MATLAB® 7
Function Reference: Volume 1 (A-E)

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Function Reference

Revision History

December 1996	First printing	For MATLAB 5.0 (Release 8)
June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online only	Revised for MATLAB 5.3 (Release 11)
June 1999	Second printing	For MATLAB 5.3 (Release 11)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for 6.5 (Release 13)
June 2004	Online only	Revised for 7.0 (Release 14)
September 2006	Online only	Revised for 7.3 (Release 2006b)
September 2007	Online only	Revised for 7.5 (Release 2007b)

Contents

1 Functions — By Category

Functions — By Category

Creating Graphical User Interfaces (p. 1-104)	GUIDE, programming graphical user interfaces
External Interfaces (p. 1-109)	Interfaces to DLLs, Java, COM and ActiveX, Web services, and serial port devices, and C and Fortran routines

Desktop Tools and Development Environment

Startup and Shutdown

exit	Terminate MATLAB (same as `quit`)
finish	MATLAB termination M-file
matlab (UNIX)	Start MATLAB (UNIX systems)
matlab (Windows)	Start MATLAB (Windows systems)
matlabrc	MATLAB startup M-file for single-user systems or system administrators
prefdir	Directory containing preferences, history, and layout files
preferences	Open Preferences dialog box for MATLAB and related products

quit	Terminate MATLAB
startup	MATLAB startup M-file for user-defined options

Command Window and History

clc	Clear Command Window
commandhistory	Open Command History window, or select it if already open
commandwindow	Open Command Window, or select it if already open
diary	Save session to file
dos	Execute DOS command and return result
format	Set display format for output
home	Move cursor to upper-left corner of Command Window
matlabcolon (matlab:)	Run specified function via hyperlink
more	Control paged output for Command Window
perl	Call Perl script using appropriate operating system executable
system	Execute operating system command and return result
unix	Execute UNIX command and return result

Help for Using MATLAB

builddocsearchdb	Build searchable documentation database
demo	Access product demos via Help browser
doc	Reference page in Help browser
docopt	Web browser for UNIX platforms
docsearch	Open Help browser **Search** pane and search for specified term
echodemo	Run M-file demo step-by-step in Command Window
help	Help for MATLAB functions in Command Window
helpbrowser	Open Help browser to access all online documentation and demos
helpwin	Provide access to M-file help for all functions
info	Information about contacting The MathWorks
lookfor	Search for keyword in all help entries
playshow	Run M-file demo (deprecated; use `echodemo` instead)
support	Open MathWorks Technical Support Web page
web	Open Web site or file in Web browser or Help browser
whatsnew	Release Notes for MathWorks products

Workspace, Search Path, and File Operations

Workspace

assignin	Assign value to variable in specified workspace
clear	Remove items from workspace, freeing up system memory
evalin	Execute MATLAB expression in specified workspace
exist	Check existence of variable, function, directory, or Java class
openvar	Open workspace variable in Array Editor or other tool for graphical editing
pack	Consolidate workspace memory
uiimport	Open Import Wizard to import data
which	Locate functions and files
workspace	Open Workspace browser to manage workspace

Search Path

addpath	Add directories to MATLAB search path
genpath	Generate path string
partialpath	Partial pathname description

path	View or change MATLAB directory search path
path2rc	Save current MATLAB search path to `pathdef.m` file
pathdef	Directories in MATLAB search path
pathsep	Path separator for current platform
pathtool	Open Set Path dialog box to view and change MATLAB path
restoredefaultpath	Restore default MATLAB search path
rmpath	Remove directories from MATLAB search path
savepath	Save current MATLAB search path to `pathdef.m` file

File Operations

See also "File I/O" on page 1-76 functions.

cd	Change working directory
copyfile	Copy file or directory
delete	Remove files or graphics objects
dir	Directory listing
exist	Check existence of variable, function, directory, or Java class
fileattrib	Set or get attributes of file or directory
filebrowser	Current Directory browser
isdir	Determine whether input is a directory
lookfor	Search for keyword in all help entries

ls	Directory contents on UNIX system
matlabroot	Root directory of MATLAB installation
mkdir	Make new directory
movefile	Move file or directory
pwd	Identify current directory
recycle	Set option to move deleted files to recycle folder
rehash	Refresh function and file system path caches
rmdir	Remove directory
toolboxdir	Root directory for specified toolbox
type	Display contents of file
web	Open Web site or file in Web browser or Help browser
what	List MATLAB files in current directory
which	Locate functions and files

Programming Tools

Edit and Debug M-Files (p. 1-9)	Edit and debug M-files
Improve Performance and Tune M-Files (p. 1-9)	Improve performance and find potential problems in M-files
Source Control (p. 1-10)	Interface MATLAB with source control system
Publishing (p. 1-10)	Publish M-file code and results

Edit and Debug M-Files

clipboard	Copy and paste strings to and from system clipboard
datatipinfo	Produce short description of input variable
dbclear	Clear breakpoints
dbcont	Resume execution
dbdown	Change local workspace context when in debug mode
dbquit	Quit debug mode
dbstack	Function call stack
dbstatus	List all breakpoints
dbstep	Execute one or more lines from current breakpoint
dbstop	Set breakpoints
dbtype	List M-file with line numbers
dbup	Change local workspace context
debug	List M-file debugging functions
edit	Edit or create M-file
keyboard	Input from keyboard

Improve Performance and Tune M-Files

memory	Help for memory limitations
mlint	Check M-files for possible problems
mlintrpt	Run `mlint` for file or directory, reporting results in browser
pack	Consolidate workspace memory
profile	Profile execution time for function

profsave	Save profile report in HTML format
rehash	Refresh function and file system path caches
sparse	Create sparse matrix
zeros	Create array of all zeros

Source Control

checkin	Check files into source control system (UNIX)
checkout	Check files out of source control system (UNIX)
cmopts	Name of source control system
customverctrl	Allow custom source control system (UNIX)
undocheckout	Undo previous checkout from source control system (UNIX)
verctrl	Source control actions (Windows)

Publishing

grabcode	MATLAB code from M-files published to HTML
notebook	Open M-book in Microsoft Word (Windows)
publish	Publish M-file containing cells, saving output to file of specified type

System

Operating System Interface (p. 1-11)	Exchange operating system information and commands with MATLAB
MATLAB Version and License (p. 1-12)	Information about MATLAB version and license

Operating System Interface

clipboard	Copy and paste strings to and from system clipboard
computer	Information about computer on which MATLAB is running
dos	Execute DOS command and return result
getenv	Environment variable
hostid	MATLAB server host identification number
maxNumCompThreads	Controls maximum number of computational threads
perl	Call Perl script using appropriate operating system executable
setenv	Set environment variable
system	Execute operating system command and return result
unix	Execute UNIX command and return result
winqueryreg	Item from Microsoft Windows registry

MATLAB Version and License

ismac	Determine whether running Macintosh OS X versions of MATLAB
ispc	Determine whether PC (Windows) version of MATLAB
isstudent	Determine whether Student Version of MATLAB
isunix	Determine whether UNIX version of MATLAB
javachk	Generate error message based on Java feature support
license	Return license number or perform licensing task
prefdir	Directory containing preferences, history, and layout files
usejava	Determine whether Java feature is supported in MATLAB
ver	Version information for MathWorks products
verLessThan	Compare toolbox version to specified version string
version	Version number for MATLAB

Mathematics

Arrays and Matrices (p. 1-14)	Basic array operators and operations, creation of elementary and specialized arrays and matrices
Linear Algebra (p. 1-19)	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
Elementary Math (p. 1-23)	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
Polynomials (p. 1-28)	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
Interpolation and Computational Geometry (p. 1-28)	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
Cartesian Coordinate System Conversion (p. 1-31)	Conversions between Cartesian and polar or spherical coordinates
Nonlinear Numerical Methods (p. 1-31)	Differential equations, optimization, integration
Specialized Math (p. 1-35)	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
Sparse Matrices (p. 1-36)	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
Math Constants (p. 1-39)	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

Arrays and Matrices

Basic Information (p. 1-14)	Display array contents, get array information, determine array type
Operators (p. 1-15)	Arithmetic operators
Elementary Matrices and Arrays (p. 1-16)	Create elementary arrays of different types, generate arrays for plotting, array indexing, etc.
Array Operations (p. 1-17)	Operate on array content, apply function to each array element, find cumulative product or sum, etc.
Array Manipulation (p. 1-17)	Create, sort, rotate, permute, reshape, and shift array contents
Specialized Matrices (p. 1-18)	Create Hadamard, Companion, Hankel, Vandermonde, Pascal matrices, etc.

Basic Information

disp	Display text or array
display	Display text or array (overloaded method)
isempty	Determine whether array is empty
isequal	Test arrays for equality
isequalwithequalnans	Test arrays for equality, treating NaNs as equal
isfinite	Array elements that are finite
isfloat	Determine whether input is floating-point array
isinf	Array elements that are infinite
isinteger	Determine whether input is integer array

islogical	Determine whether input is logical array
isnan	Array elements that are NaN
isnumeric	Determine whether input is numeric array
isscalar	Determine whether input is scalar
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
length	Length of vector
max	Largest elements in array
min	Smallest elements in array
ndims	Number of array dimensions
numel	Number of elements in array or subscripted array expression
size	Array dimensions

Operators

+	Addition
+	Unary plus
-	Subtraction
-	Unary minus
*	Matrix multiplication
^	Matrix power
\	Backslash or left matrix divide
/	Slash or right matrix divide
’	Transpose
.’	Nonconjugated transpose
.*	Array multiplication (element-wise)

.^	Array power (element-wise)
.\	Left array divide (element-wise)
./	Right array divide (element-wise)

Elementary Matrices and Arrays

blkdiag	Construct block diagonal matrix from input arguments
diag	Diagonal matrices and diagonals of matrix
eye	Identity matrix
freqspace	Frequency spacing for frequency response
ind2sub	Subscripts from linear index
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
meshgrid	Generate `X` and `Y` arrays for 3-D plots
ndgrid	Generate arrays for N-D functions and interpolation
ones	Create array of all ones
rand	Uniformly distributed pseudorandom numbers
randn	Normally distributed random numbers
sub2ind	Single index from subscripts
zeros	Create array of all zeros

Array Operations

See “Linear Algebra” on page 1-19 and “Elementary Math” on page 1-23 for other array operations.

accumarray	Construct array with accumulation
arrayfun	Apply function to each element of array
bsxfun	Apply element-by-element binary operation to two arrays with singleton expansion enabled
cast	Cast variable to different data type
cross	Vector cross product
cumprod	Cumulative product
cumsum	Cumulative sum
dot	Vector dot product
idivide	Integer division with rounding option
kron	Kronecker tensor product
prod	Product of array elements
sum	Sum of array elements
tril	Lower triangular part of matrix
triu	Upper triangular part of matrix

Array Manipulation

blkdiag	Construct block diagonal matrix from input arguments
cat	Concatenate arrays along specified dimension
circshift	Shift array circularly

diag	Diagonal matrices and diagonals of matrix
end	Terminate block of code, or indicate last array index
flipdim	Flip array along specified dimension
fliplr	Flip matrix left to right
flipud	Flip matrix up to down
horzcat	Concatenate arrays horizontally
inline	Construct inline object
ipermute	Inverse permute dimensions of N-D array
permute	Rearrange dimensions of N-D array
repmat	Replicate and tile array
reshape	Reshape array
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
vectorize	Vectorize expression
vertcat	Concatenate arrays vertically

Specialized Matrices

compan	Companion matrix
gallery	Test matrices
hadamard	Hadamard matrix
hankel	Hankel matrix

hilb	Hilbert matrix
invhilb	Inverse of Hilbert matrix
magic	Magic square
pascal	Pascal matrix
rosser	Classic symmetric eigenvalue test problem
toeplitz	Toeplitz matrix
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix

Linear Algebra

Matrix Analysis (p. 1-19)	Compute norm, rank, determinant, condition number, etc.
Linear Equations (p. 1-20)	Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.
Eigenvalues and Singular Values (p. 1-21)	Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.
Matrix Logarithms and Exponentials (p. 1-22)	Matrix logarithms, exponentials, square root
Factorization (p. 1-22)	Cholesky, LU, and QR factorizations, diagonal forms, singular value decomposition

Matrix Analysis

cond	Condition number with respect to inversion
condeig	Condition number with respect to eigenvalues

det	Matrix determinant
norm	Vector and matrix norms
normest	2-norm estimate
null	Null space
orth	Range space of matrix
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
rref	Reduced row echelon form
subspace	Angle between two subspaces
trace	Sum of diagonal elements

Linear Equations

chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cond	Condition number with respect to inversion
condest	1-norm condition number estimate
funm	Evaluate general matrix function
ilu	Sparse incomplete LU factorization
inv	Matrix inverse
linsolve	Solve linear system of equations
lscov	Least-squares solution in presence of known covariance
lsqnonneg	Solve nonnegative least-squares constraints problem
lu	LU matrix factorization

luinc	Sparse incomplete LU factorization
pinv	Moore-Penrose pseudoinverse of matrix
qr	Orthogonal-triangular decomposition
rcond	Matrix reciprocal condition number estimate

Eigenvalues and Singular Values

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
condeig	Condition number with respect to eigenvalues
eig	Find eigenvalues and eigenvectors
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
gsvd	Generalized singular value decomposition
hess	Hessenberg form of matrix
ordeig	Eigenvalues of quasitriangular matrices
ordqz	Reorder eigenvalues in QZ factorization
ordschur	Reorder eigenvalues in Schur factorization
poly	Polynomial with specified roots
polyeig	Polynomial eigenvalue problem

rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtm	Matrix square root
ss2tf	Convert state-space filter parameters to transfer function form
svd	Singular value decomposition
svds	Find singular values and vectors

Matrix Logarithms and Exponentials

expm	Matrix exponential
logm	Matrix logarithm
sqrtm	Matrix square root

Factorization

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cholupdate	Rank 1 update to Cholesky factorization
gsvd	Generalized singular value decomposition
ilu	Sparse incomplete LU factorization
lu	LU matrix factorization

luinc	Sparse incomplete LU factorization
planerot	Givens plane rotation
qr	Orthogonal-triangular decomposition
qrdelete	Remove column or row from QR factorization
qrinsert	Insert column or row into QR factorization
qrupdate	
qz	QZ factorization for generalized eigenvalues
rsf2csf	Convert real Schur form to complex Schur form
svd	Singular value decomposition

Elementary Math

Trigonometric (p. 1-24)	Trigonometric functions with results in radians or degrees
Exponential (p. 1-25)	Exponential, logarithm, power, and root functions
Complex (p. 1-26)	Numbers with real and imaginary components, phase angles
Rounding and Remainder (p. 1-27)	Rounding, modulus, and remainder
Discrete Math (e.g., Prime Factors) (p. 1-27)	Prime factors, factorials, permutations, rational fractions, least common multiple, greatest common divisor

Trigonometric

acos	Inverse cosine; result in radians
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine; result in radians
asind	Inverse sine; result in degrees
asinh	Inverse hyperbolic sine
atan	Inverse tangent; result in radians
atan2	Four-quadrant inverse tangent
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent
cos	Cosine of argument in radians
cosd	Cosine ofo argument in degrees
cosh	Hyperbolic cosine
cot	Cotangent of argument in radians
cotd	Cotangent of argument in degrees
coth	Hyperbolic cotangent
csc	Cosecant of argument in radians

cscd	Cosecant of argument in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant of argument in radians
secd	Secant of argument in degrees
sech	Hyperbolic secant
sin	Sine of argument in radians
sind	Sine of argument in degrees
sinh	Hyperbolic sine of argument in radians
tan	Tangent of argument in radians
tand	Tangent of argument in degrees
tanh	Hyperbolic tangent

Exponential

exp	Exponential
expm1	Compute `exp(x)-1` accurately for small values of `x`
log	Natural logarithm
log10	Common (base 10) logarithm
log1p	Compute `log(1+x)` accurately for small values of `x`
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
nextpow2	Next higher power of 2
nthroot	Real `n`th root of real numbers
pow2	Base 2 power and scale floating-point numbers

reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays
sqrt	Square root

Complex

abs	Absolute value and complex magnitude
angle	Phase angle
complex	Construct complex data from real and imaginary components
conj	Complex conjugate
cplxpair	Sort complex numbers into complex conjugate pairs
i	Imaginary unit
imag	Imaginary part of complex number
isreal	Determine whether input is real array
j	Imaginary unit
real	Real part of complex number
sign	Signum function
unwrap	Correct phase angles to produce smoother phase plots

Rounding and Remainder

ceil	Round toward infinity
fix	Round toward zero
floor	Round toward minus infinity
idivide	Integer division with rounding option
mod	Modulus after division
rem	Remainder after division
round	Round to nearest integer

Discrete Math (e.g., Prime Factors)

factor	Prime factors
factorial	Factorial function
gcd	Greatest common divisor
isprime	Array elements that are prime numbers
lcm	Least common multiple
nchoosek	Binomial coefficient or all combinations
perms	All possible permutations
primes	Generate list of prime numbers
rat, rats	Rational fraction approximation

Polynomials

conv	Convolution and polynomial multiplication
deconv	Deconvolution and polynomial division
poly	Polynomial with specified roots
polyder	Polynomial derivative
polyeig	Polynomial eigenvalue problem
polyfit	Polynomial curve fitting
polyint	Integrate polynomial analytically
polyval	Polynomial evaluation
polyvalm	Matrix polynomial evaluation
residue	Convert between partial fraction expansion and polynomial coefficients
roots	Polynomial roots

Interpolation and Computational Geometry

Interpolation (p. 1-29)	Data interpolation, data gridding, polynomial evaluation, nearest point search
Delaunay Triangulation and Tessellation (p. 1-30)	Delaunay triangulation and tessellation, triangular surface and mesh plots
Convex Hull (p. 1-30)	Plot convex hull, plotting functions
Voronoi Diagrams (p. 1-30)	Plot Voronoi diagram, patch graphics object, plotting functions
Domain Generation (p. 1-31)	Generate arrays for 3-D plots, or for N-D functions and interpolation

Interpolation

dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
griddata	Data gridding
griddata3	Data gridding and hypersurface fitting for 3-D data
griddatan	Data gridding and hypersurface fitting (dimension >= 2)
interp1	1-D data interpolation (table lookup)
interp1q	Quick 1-D linear interpolation
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpft	1-D interpolation using FFT method
interpn	N-D data interpolation (table lookup)
meshgrid	Generate `X` and `Y` arrays for 3-D plots
mkpp	Make piecewise polynomial
ndgrid	Generate arrays for N-D functions and interpolation
padecoef	Padé approximation of time delays
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
ppval	Evaluate piecewise polynomial
spline	Cubic spline data interpolation
tsearchn	N-D closest simplex search
unmkpp	Piecewise polynomial details

Delaunay Triangulation and Tessellation

delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search

Convex Hull

convhull	Convex hull
convhulln	N-D convex hull
patch	Create patch graphics object
plot	2-D line plot
trisurf	Triangular surface plot

Voronoi Diagrams

dsearch	Search Delaunay triangulation for nearest point
patch	Create patch graphics object
plot	2-D line plot

voronoi	Voronoi diagram
voronoin	N-D Voronoi diagram

Domain Generation

meshgrid	Generate `X` and `Y` arrays for 3-D plots
ndgrid	Generate arrays for N-D functions and interpolation

Cartesian Coordinate System Conversion

cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical
pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Nonlinear Numerical Methods

Ordinary Differential Equations (IVP) (p. 1-32)	Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations (p. 1-33)	Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Boundary Value Problems (p. 1-33)	Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution

Partial Differential Equations (p. 1-34)	Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution
Optimization (p. 1-34)	Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature) (p. 1-34)	Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

Ordinary Differential Equations (IVP)

decic	Compute consistent initial conditions for `ode15i`
deval	Evaluate solution of differential equation problem
ode15i	Solve fully implicit differential equations, variable order method
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb	Solve initial value problems for ordinary differential equations
odefile	Define differential equation problem for ordinary differential equation solvers
odeget	Ordinary differential equation options parameters
odeset	Create or alter options structure for ordinary differential equation solvers
odextend	Extend solution of initial value problem for ordinary differential equation

Delay Differential Equations

dde23	Solve delay differential equations (DDEs) with constant delays
ddeget	Extract properties from delay differential equations options structure
ddesd	Solve delay differential equations (DDEs) with general delays
ddeset	Create or alter delay differential equations options structure
deval	Evaluate solution of differential equation problem

Boundary Value Problems

bvp4c	Solve boundary value problems for ordinary differential equations
bvp5c	Solve boundary value problems for ordinary differential equations
bvpget	Extract properties from options structure created with `bvpset`
bvpinit	Form initial guess for `bvp4c`
bvpset	Create or alter options structure of boundary value problem
bvpxtend	Form guess structure for extending boundary value solutions
deval	Evaluate solution of differential equation problem

Partial Differential Equations

pdepe	Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D
pdeval	Evaluate numerical solution of PDE using output of pdepe

Optimization

fminbnd	Find minimum of single-variable function on fixed interval
fminsearch	Find minimum of unconstrained multivariable function using derivative-free method
fzero	Find root of continuous function of one variable
lsqnonneg	Solve nonnegative least-squares constraints problem
optimget	Optimization options values
optimset	Create or edit optimization options structure

Numerical Integration (Quadrature)

dblquad	Numerically evaluate double integral
quad	Numerically evaluate integral, adaptive Simpson quadrature
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature
quadl	Numerically evaluate integral, adaptive Lobatto quadrature

quadv	Vectorized quadrature
triplequad	Numerically evaluate triple integral

Specialized Math

airy	Airy functions
besselh	Bessel function of third kind (Hankel function)
besseli	Modified Bessel function of first kind
besselj	Bessel function of first kind
besselk	Modified Bessel function of second kind
bessely	Bessel function of second kind
beta	Beta function
betainc	Incomplete beta function
betaln	Logarithm of beta function
ellipj	Jacobi elliptic functions
ellipke	Complete elliptic integrals of first and second kind
erf, erfc, erfcx, erfinv, erfcinv	Error functions
expint	Exponential integral
gamma, gammainc, gammaln	Gamma functions
legendre	Associated Legendre functions
psi	Psi (polygamma) function

Sparse Matrices

Elementary Sparse Matrices (p. 1-36)	Create random and nonrandom sparse matrices
Full to Sparse Conversion (p. 1-37)	Convert full matrix to sparse, sparse matrix to full
Working with Sparse Matrices (p. 1-37)	Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern.
Reordering Algorithms (p. 1-37)	Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
Linear Algebra (p. 1-38)	Compute norms, eigenvalues, factorizations, least squares, structural rank
Linear Equations (Iterative Methods) (p. 1-38)	Methods for conjugate and biconjugate gradients, residuals, lower quartile
Tree Operations (p. 1-39)	Elimination trees, tree plotting, factorization analysis

Elementary Sparse Matrices

spdiags	Extract and create sparse band and diagonal matrices
speye	Sparse identity matrix
sprand	Sparse uniformly distributed random matrix
sprandn	Sparse normally distributed random matrix
sprandsym	Sparse symmetric random matrix

Full to Sparse Conversion

find	Find indices and values of nonzero elements
full	Convert sparse matrix to full matrix
sparse	Create sparse matrix
spconvert	Import matrix from sparse matrix external format

Working with Sparse Matrices

issparse	Determine whether input is sparse
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
nzmax	Amount of storage allocated for nonzero matrix elements
spalloc	Allocate space for sparse matrix
spfun	Apply function to nonzero sparse matrix elements
spones	Replace nonzero sparse matrix elements with ones
spparms	Set parameters for sparse matrix routines
spy	Visualize sparsity pattern

Reordering Algorithms

amd	Approximate minimum degree permutation
colamd	Column approximate minimum degree permutation

colperm	Sparse column permutation based on nonzero count
dmperm	Dulmage-Mendelsohn decomposition
ldl	Block ldl' factorization for Hermitian indefinite matrices
randperm	Random permutation
symamd	Symmetric approximate minimum degree permutation
symrcm	Sparse reverse Cuthill-McKee ordering

Linear Algebra

cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
condest	1-norm condition number estimate
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
ilu	Sparse incomplete LU factorization
luinc	Sparse incomplete LU factorization
normest	2-norm estimate
spaugment	Form least squares augmented system
sprank	Structural rank
svds	Find singular values and vectors

Linear Equations (Iterative Methods)

bicg	Biconjugate gradients method
bicgstab	Biconjugate gradients stabilized method

cgs	Conjugate gradients squared method
gmres	Generalized minimum residual method (with restarts)
lsqr	LSQR method
minres	Minimum residual method
pcg	Preconditioned conjugate gradients method
qmr	Quasi-minimal residual method
symmlq	Symmetric LQ method

Tree Operations

etree	Elimination tree
etreeplot	Plot elimination tree
gplot	Plot nodes and links representing adjacency matrix
symbfact	Symbolic factorization analysis
treelayout	Lay out tree or forest
treeplot	Plot picture of tree

Math Constants

eps	Floating-point relative accuracy
i	Imaginary unit
Inf	Infinity
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
j	Imaginary unit

NaN	Not-a-Number
pi	Ratio of circle's circumference to its diameter, π
realmax	Largest positive floating-point number
realmin	Smallest positive normalized floating-point number

Data Analysis

Basic Operations

cumprod	Cumulative product
cumsum	Cumulative sum
prod	Product of array elements
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
sum	Sum of array elements

Descriptive Statistics

corrcoef	Correlation coefficients
cov	Covariance matrix
max	Largest elements in array
mean	Average or mean value of array
median	Median value of array

min	Smallest elements in array
mode	Most frequent values in array
std	Standard deviation
var	Variance

Filtering and Convolution

conv	Convolution and polynomial multiplication
conv2	2-D convolution
convn	N-D convolution
deconv	Deconvolution and polynomial division
detrend	Remove linear trends
filter	1-D digital filter
filter2	2-D digital filter

Interpolation and Regression

interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpn	N-D data interpolation (table lookup)
mldivide \, mrdivide /	Left or right matrix division
polyfit	Polynomial curve fitting
polyval	Polynomial evaluation

Fourier Transforms

abs	Absolute value and complex magnitude
angle	Phase angle
cplxpair	Sort complex numbers into complex conjugate pairs
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftn	N-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
fftw	Interface to FFTW library run-time algorithm tuning control
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform
ifftshift	Inverse FFT shift
nextpow2	Next higher power of 2
unwrap	Correct phase angles to produce smoother phase plots

Derivatives and Integrals

cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives

gradient	Numerical gradient
polyder	Polynomial derivative
polyint	Integrate polynomial analytically
trapz	Trapezoidal numerical integration

Time Series Objects

General Purpose (p. 1-44)	Combine `timeseries` objects, query and set `timeseries` object properties, plot `timeseries` objects
Data Manipulation (p. 1-45)	Add or delete data, manipulate `timeseries` objects
Event Data (p. 1-46)	Add or delete events, create new `timeseries` objects based on event data
Descriptive Statistics (p. 1-46)	Descriptive statistics for `timeseries` objects

General Purpose

get (timeseries)	Query `timeseries` object property values
getdatasamplesize	Size of data sample in `timeseries` object
getqualitydesc	Data quality descriptions
isempty (timeseries)	Determine whether `timeseries` object is empty
length (timeseries)	Length of time vector
plot (timeseries)	Plot time series
set (timeseries)	Set properties of `timeseries` object
size (timeseries)	Size of `timeseries` object

timeseries	Create `timeseries` object
tsdata.event	Construct event object for `timeseries` object
tsprops	Help on `timeseries` object properties
tstool	Open Time Series Tools GUI

Data Manipulation

addsample	Add data sample to `timeseries` object
ctranspose (timeseries)	Transpose `timeseries` object
delsample	Remove sample from `timeseries` object
detrend (timeseries)	Subtract mean or best-fit line and all `NaN`s from time series
filter (timeseries)	Shape frequency content of time series
getabstime (timeseries)	Extract date-string time vector into cell array
getinterpmethod	Interpolation method for `timeseries` object
getsampleusingtime (timeseries)	Extract data samples into new `timeseries` object
idealfilter (timeseries)	Apply ideal (noncausal) filter to `timeseries` object
resample (timeseries)	Select or interpolate `timeseries` data using new time vector
setabstime (timeseries)	Set times of `timeseries` object as date strings
setinterpmethod	Set default interpolation method for `timeseries` object

synchronize	Synchronize and resample two `timeseries` objects using common time vector
transpose (timeseries)	Transpose `timeseries` object
vertcat (timeseries)	Vertical concatenation of `timeseries` objects

Event Data

addevent	Add event to `timeseries` object
delevent	Remove `tsdata.event` objects from `timeseries` object
gettsafteratevent	New `timeseries` object with samples occurring at or after event
gettsafterevent	New `timeseries` object with samples occurring after event
gettsatevent	New `timeseries` object with samples occurring at event
gettsbeforeatevent	New `timeseries` object with samples occurring before or at event
gettsbeforeevent	New `timeseries` object with samples occurring before event
gettsbetweenevents	New `timeseries` object with samples occurring between events

Descriptive Statistics

iqr (timeseries)	Interquartile range of `timeseries` data
max (timeseries)	Maximum value of `timeseries` data
mean (timeseries)	Mean value of `timeseries` data
median (timeseries)	Median value of `timeseries` data

min (timeseries)	Minimum value of `timeseries` data
std (timeseries)	Standard deviation of `timeseries` data
sum (timeseries)	Sum of `timeseries` data
var (timeseries)	Variance of `timeseries` data

Time Series Collections

General Purpose (p. 1-47)	Query and set `tscollection` object properties, plot `tscollection` objects
Data Manipulation (p. 1-48)	Add or delete data, manipulate `tscollection` objects

General Purpose

get (tscollection)	Query `tscollection` object property values
isempty (tscollection)	Determine whether `tscollection` object is empty
length (tscollection)	Length of time vector
plot (timeseries)	Plot time series
set (tscollection)	Set properties of `tscollection` object
size (tscollection)	Size of `tscollection` object
tscollection	Create `tscollection` object
tstool	Open Time Series Tools GUI

Data Manipulation

addsampletocollection	Add sample to `tscollection` object
addts	Add `timeseries` object to `tscollection` object
delsamplefromcollection	Remove sample from `tscollection` object
getabstime (tscollection)	Extract date-string time vector into cell array
getsampleusingtime (tscollection)	Extract data samples into new `tscollection` object
gettimeseriesnames	Cell array of names of `timeseries` objects in `tscollection` object
horzcat (tscollection)	Horizontal concatenation for `tscollection` objects
removets	Remove `timeseries` objects from `tscollection` object
resample (tscollection)	Select or interpolate data in `tscollection` using new time vector
setabstime (tscollection)	Set times of `tscollection` object as date strings
settimeseriesnames	Change name of `timeseries` object in `tscollection`
vertcat (tscollection)	Vertical concatenation for `tscollection` objects

Programming and Data Types

Data Types

Cell Arrays (p. 1-53)	Data of varying types and sizes stored in cells of array
Function Handles (p. 1-54)	Invoke a function indirectly via handle
MATLAB Classes and Objects (p. 1-55)	MATLAB object-oriented class system
Java Classes and Objects (p. 1-55)	Access Java classes through MATLAB interface
Data Type Identification (p. 1-57)	Determine data type of a variable

Numeric Types

arrayfun	Apply function to each element of array
cast	Cast variable to different data type
cat	Concatenate arrays along specified dimension
class	Create object or return class of object
find	Find indices and values of nonzero elements
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
intwarning	Control state of integer warnings
ipermute	Inverse permute dimensions of N-D array
isa	Determine whether input is object of given class
isequal	Test arrays for equality

isequalwithequalnans	Test arrays for equality, treating NaNs as equal
isfinite	Array elements that are finite
isinf	Array elements that are infinite
isnan	Array elements that are NaN
isnumeric	Determine whether input is numeric array
isreal	Determine whether input is real array
isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
permute	Rearrange dimensions of N-D array
realmax	Largest positive floating-point number
realmin	Smallest positive normalized floating-point number
reshape	Reshape array
squeeze	Remove singleton dimensions
zeros	Create array of all zeros

Characters and Strings

See “String Functions” on page 1-63 for all string-related functions.

cellstr	Create cell array of strings from character array
char	Convert to character array (string)
eval	Execute string containing MATLAB expression
findstr	Find string within another, longer string

isstr	Determine whether input is character array
regexp, regexpi	Match regular expression
sprintf	Write formatted data to string
sscanf	Read formatted data from string
strcat	Concatenate strings horizontally
strcmp, strcmpi	Compare strings
strings	MATLAB string handling
strjust	Justify character array
strmatch	Find possible matches for string
strread	Read formatted data from string
strrep	Find and replace substring
strtrim	Remove leading and trailing white space from string
strvcat	Concatenate strings vertically

Structures

arrayfun	Apply function to each element of array
cell2struct	Convert cell array to structure array
class	Create object or return class of object
deal	Distribute inputs to outputs
fieldnames	Field names of structure, or public fields of object
getfield	Field of structure array
isa	Determine whether input is object of given class
isequal	Test arrays for equality

isfield	Determine whether input is structure array field
isscalar	Determine whether input is scalar
isstruct	Determine whether input is structure array
isvector	Determine whether input is vector
orderfields	Order fields of structure array
rmfield	Remove fields from structure
setfield	Set value of structure array field
struct	Create structure array
struct2cell	Convert structure to cell array
structfun	Apply function to each field of scalar structure

Cell Arrays

cell	Construct cell array
cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array
celldisp	Cell array contents
cellfun	Apply function to each cell in cell array
cellplot	Graphically display structure of cell array
cellstr	Create cell array of strings from character array
class	Create object or return class of object
deal	Distribute inputs to outputs

isa	Determine whether input is object of given class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
isequal	Test arrays for equality
isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
struct2cell	Convert structure to cell array

Function Handles

class	Create object or return class of object
feval	Evaluate function
func2str	Construct function name string from function handle
functions	Information about function handle
function_handle (@)	Handle used in calling functions indirectly
isa	Determine whether input is object of given class
isequal	Test arrays for equality
str2func	Construct function handle from function name string

MATLAB Classes and Objects

class	Create object or return class of object
fieldnames	Field names of structure, or public fields of object
inferiorto	Establish inferior class relationship
isa	Determine whether input is object of given class
isobject	Determine whether input is MATLAB OOPs object
loadobj	User-defined extension of `load` function for user objects
methods	Information on class methods
methodsview	Information on class methods in separate window
saveobj	User-defined extension of `save` function for user objects
subsasgn	Subscripted assignment for objects
subsindex	Subscripted indexing for objects
subsref	Subscripted reference for objects
substruct	Create structure argument for `subsasgn` or `subsref`
superiorto	Establish superior class relationship

Java Classes and Objects

cell	Construct cell array
class	Create object or return class of object
clear	Remove items from workspace, freeing up system memory
depfun	List dependencies of M-file or P-file

exist	Check existence of variable, function, directory, or Java class
fieldnames	Field names of structure, or public fields of object
im2java	Convert image to Java image
import	Add package or class to current Java import list
inmem	Names of M-files, MEX-files, Java classes in memory
isa	Determine whether input is object of given class
isjava	Determine whether input is Java object
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Set and get dynamic Java class path
javaMethod	Invoke Java method
javaObject	Construct Java object
javarmpath	Remove entries from dynamic Java class path
methods	Information on class methods
methodsview	Information on class methods in separate window
usejava	Determine whether Java feature is supported in MATLAB
which	Locate functions and files

Data Type Identification

is*	Detect state
isa	Determine whether input is object of given class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isfield	Determine whether input is structure array field
isfloat	Determine whether input is floating-point array
isinteger	Determine whether input is integer array
isjava	Determine whether input is Java object
islogical	Determine whether input is logical array
isnumeric	Determine whether input is numeric array
isobject	Determine whether input is MATLAB OOPs object
isreal	Determine whether input is real array
isstr	Determine whether input is character array
isstruct	Determine whether input is structure array

validateattributes	Check validity of array
who, whos	List variables in workspace

Data Type Conversion

Numeric (p. 1-58)	Convert data of one numeric type to another numeric type
String to Numeric (p. 1-58)	Convert characters to numeric equivalent
Numeric to String (p. 1-59)	Convert numeric to character equivalent
Other Conversions (p. 1-59)	Convert to structure, cell array, function handle, etc.

Numeric

cast	Cast variable to different data type
double	Convert to double precision
int8, int16, int32, int64	Convert to signed integer
single	Convert to single precision
typecast	Convert data types without changing underlying data
uint8, uint16, uint32, uint64	Convert to unsigned integer

String to Numeric

base2dec	Convert base N number string to decimal number
bin2dec	Convert binary number string to decimal number
cast	Cast variable to different data type

hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
str2double	Convert string to double-precision value
str2num	Convert string to number
unicode2native	Convert Unicode characters to numeric bytes

Numeric to String

cast	Cast variable to different data type
char	Convert to character array (string)
dec2base	Convert decimal to base N number in string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
int2str	Convert integer to string
mat2str	Convert matrix to string
native2unicode	Convert numeric bytes to Unicode characters
num2str	Convert number to string

Other Conversions

cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array

datestr	Convert date and time to string format
func2str	Construct function name string from function handle
logical	Convert numeric values to logical
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
num2hex	Convert singles and doubles to IEEE hexadecimal strings
str2func	Construct function handle from function name string
str2mat	Form blank-padded character matrix from strings
struct2cell	Convert structure to cell array

Operators and Special Characters

Arithmetic Operators (p. 1-60)	Plus, minus, power, left and right divide, transpose, etc.
Relational Operators (p. 1-61)	Equal to, greater than, less than or equal to, etc.
Logical Operators (p. 1-61)	Element-wise and short circuit and, or, not
Special Characters (p. 1-62)	Array constructors, line continuation, comments, etc.

Arithmetic Operators

+	Plus
-	Minus

.	Decimal point
=	Assignment
*	Matrix multiplication
/	Matrix right division
\	Matrix left division
^	Matrix power
'	Matrix transpose
.*	Array multiplication (element-wise)
./	Array right division (element-wise)
.\	Array left division (element-wise)
.^	Array power (element-wise)
.'	Array transpose

Relational Operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Logical Operators

See also "Logical Functions" on page 1-66 for functions like `xor`, `all`, `any`, etc.

&&	Logical AND
||	Logical OR
&	Logical AND for arrays

|	Logical OR for arrays
~	Logical NOT

Special Characters

:	Create vectors, subscript arrays, specify for-loop iterations
( )	Pass function arguments, prioritize operators
[ ]	Construct array, concatenate elements, specify multiple outputs from function
{ }	Construct cell array, index into cell array
.	Insert decimal point, define structure field, reference methods of object
.( )	Reference dynamic field of structure
..	Reference parent directory
...	Continue statement to next line
,	Separate rows of array, separate function input/output arguments, separate commands
;	Separate columns of array, suppress output from current command
%	Insert comment line into code
%{ %}	Insert block of comments into code
!	Issue command to operating system
' '	Construct character array
@	Construct function handle, reference class directory

String Functions

Description of Strings in MATLAB

strings	MATLAB string handling

String Creation

blanks	Create string of blank characters
cellstr	Create cell array of strings from character array
char	Convert to character array (string)
sprintf	Write formatted data to string
strcat	Concatenate strings horizontally
strvcat	Concatenate strings vertically

String Identification

class	Create object or return class of object
isa	Determine whether input is object of given class
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isletter	Array elements that are alphabetic letters
isscalar	Determine whether input is scalar
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
isvector	Determine whether input is vector
validatestring	Check validity of text string

String Manipulation

deblank	Strip trailing blanks from end of string
lower	Convert string to lowercase
strjust	Justify character array
strrep	Find and replace substring
strtrim	Remove leading and trailing white space from string
upper	Convert string to uppercase

String Parsing

findstr	Find string within another, longer string
regexp, regexpi	Match regular expression
regexprep	Replace string using regular expression
regexptranslate	Translate string into regular expression
sscanf	Read formatted data from string
strfind	Find one string within another
strread	Read formatted data from string
strtok	Selected parts of string

String Evaluation

eval	Execute string containing MATLAB expression
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace

String Comparison

strcmp, strcmpi	Compare strings
strmatch	Find possible matches for string
strncmp, strncmpi	Compare first n characters of strings

Bit-wise Functions

bitand	Bitwise AND
bitcmp	Bitwise complement
bitget	Bit at specified position
bitmax	Maximum double-precision floating-point integer
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR
swapbytes	Swap byte ordering

Logical Functions

all	Determine whether all array elements are nonzero
and	Find logical AND of array or scalar inputs
any	Determine whether any array elements are nonzero
false	Logical 0 (false)
find	Find indices and values of nonzero elements
isa	Determine whether input is object of given class
iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
logical	Convert numeric values to logical

not	Find logical NOT of array or scalar input
or	Find logical OR of array or scalar inputs
true	Logical 1 (true)
xor	Logical exclusive-OR

See “Operators and Special Characters” on page 1-60 for logical operators.

Relational Functions

eq	Test for equality
ge	Test for greater than or equal to
gt	Test for greater than
le	Test for less than or equal to
lt	Test for less than
ne	Test for inequality

See “Operators and Special Characters” on page 1-60 for relational operators.

Set Functions

intersect	Find set intersection of two vectors
ismember	Array elements that are members of set
issorted	Determine whether set elements are in sorted order
setdiff	Find set difference of two vectors
setxor	Find set exclusive OR of two vectors
union	Find set union of two vectors
unique	Find unique elements of vector

Date and Time Functions

addtodate	Modify date number by field
calendar	Calendar for specified month
clock	Current time as date vector
cputime	Elapsed CPU time
date	Current date string
datenum	Convert date and time to serial date number
datestr	Convert date and time to string format
datevec	Convert date and time to vector of components
eomday	Last day of month
etime	Time elapsed between date vectors
now	Current date and time
weekday	Day of week

Programming in MATLAB

M-File Functions and Scripts (p. 1-69)	Declare functions, handle arguments, identify dependencies, etc.
Evaluation of Expressions and Functions (p. 1-70)	Evaluate expression in string, apply function to array, run script file, etc.
Timer Functions (p. 1-71)	Schedule execution of MATLAB commands
Variables and Functions in Memory (p. 1-72)	List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches

Control Flow (p. 1-73)	if-then-else, for loops, switch-case, try-catch
Error Handling (p. 1-74)	Generate warnings and errors, test for and catch errors, retrieve most recent error message
MEX Programming (p. 1-75)	Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

M-File Functions and Scripts

addOptional (inputParser)	Add optional argument to `inputParser` schema
addParamValue (inputParser)	Add parameter-value argument to `inputParser` schema
addRequired (inputParser)	Add required argument to `inputParser` schema
createCopy (inputParser)	Create copy of `inputParser` object
depdir	List dependent directories of M-file or P-file
depfun	List dependencies of M-file or P-file
echo	Echo M-files during execution
end	Terminate block of code, or indicate last array index
function	Declare M-file function
input	Request user input
inputname	Variable name of function input
inputParser	Construct input parser object
mfilename	Name of currently running M-file
namelengthmax	Maximum identifier length
nargchk	Validate number of input arguments

nargin, nargout	Number of function arguments
nargoutchk	Validate number of output arguments
parse (inputParser)	Parse and validate named inputs
pcode	Create preparsed pseudocode file (P-file)
script	Script M-file description
syntax	Two ways to call MATLAB functions
varargin	Variable length input argument list
varargout	Variable length output argument list

Evaluation of Expressions and Functions

ans	Most recent answer
arrayfun	Apply function to each element of array
assert	Generate error when condition is violated
builtin	Execute built-in function from overloaded method
cellfun	Apply function to each cell in cell array
echo	Echo M-files during execution
eval	Execute string containing MATLAB expression
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace
feval	Evaluate function

iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
pause	Halt execution temporarily
run	Run script that is not on current path
script	Script M-file description
structfun	Apply function to each field of scalar structure
symvar	Determine symbolic variables in expression
tic, toc	Measure performance using stopwatch timer

Timer Functions

delete (timer)	Remove timer object from memory
disp (timer)	Information about timer object
get (timer)	Timer object properties
isvalid (timer)	Determine whether timer object is valid
set (timer)	Configure or display timer object properties
start	Start timer(s) running
startat	Start timer(s) running at specified time
stop	Stop timer(s)
timer	Construct timer object
timerfind	Find timer objects

timerfindall	Find timer objects, including invisible objects
wait	Wait until timer stops running

Variables and Functions in Memory

ans	Most recent answer
assignin	Assign value to variable in specified workspace
datatipinfo	Produce short description of input variable
genvarname	Construct valid variable name from string
global	Declare global variables
inmem	Names of M-files, MEX-files, Java classes in memory
isglobal	Determine whether input is global variable
mislocked	Determine whether M-file or MEX-file cannot be cleared from memory
mlock	Prevent clearing M-file or MEX-file from memory
munlock	Allow clearing M-file or MEX-file from memory
namelengthmax	Maximum identifier length
pack	Consolidate workspace memory
persistent	Define persistent variable
rehash	Refresh function and file system path caches

Control Flow

break	Terminate execution of `for` or `while` loop
case	Execute block of code if condition is `true`
catch	Specify how to respond to error in `try` statement
continue	Pass control to next iteration of `for` or `while` loop
else	Execute statements if condition is false
elseif	Execute statements if additional condition is true
end	Terminate block of code, or indicate last array index
error	Display message and abort function
for	Execute block of code specified number of times
if	Execute statements if condition is true
otherwise	Default part of switch statement
return	Return to invoking function
switch	Switch among several cases, based on expression
try	Attempt to execute block of code, and catch errors
while	Repeatedly execute statements while condition is true

Error Handling

addCause (MException)	Append `MException` objects
assert	Generate error when condition is violated
catch	Specify how to respond to error in `try` statement
disp (MException)	Display `MException` object
eq (MException)	Compare `MException` objects for equality
error	Display message and abort function
ferror	Query MATLAB about errors in file input or output
getReport (MException)	Get error message for exception
intwarning	Control state of integer warnings
isequal (MException)	Compare `MException` objects for equality
last (MException)	Last uncaught exception
lasterr	Last error message
lasterror	Last error message and related information
lastwarn	Last warning message
MException	Construct `MException` object
ne (MException)	Compare `MException` objects for inequality
rethrow	Reissue error
rethrow (MException)	Reissue existing exception
throw (MException)	Terminate function and issue exception

try	Attempt to execute block of code, and catch errors
warning	Warning message

MEX Programming

dbmex	Enable MEX-file debugging
inmem	Names of M-files, MEX-files, Java classes in memory
mex	Compile MEX-function from C, C++, or Fortran source code
mexext	MEX-filename extension

File I/O

File Name Construction (p. 1-76)	Get path, directory, filename information; construct filenames
Opening, Loading, Saving Files (p. 1-77)	Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-77)	Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-77)	Low-level operations that use a file identifier
Text Files (p. 1-78)	Delimited or formatted I/O to text files
XML Documents (p. 1-79)	Documents written in Extensible Markup Language
Spreadsheets (p. 1-79)	Excel and Lotus 1-2-3 files
Scientific Data (p. 1-80)	CDF, FITS, HDF formats
Audio and Audio/Video (p. 1-81)	General audio functions; SparcStation, WAVE, AVI files
Images (p. 1-83)	Graphics files
Internet Exchange (p. 1-84)	URL, FTP, zip, tar, and e-mail

To see a listing of file formats that are readable from MATLAB, go to file formats.

File Name Construction

filemarker	Character to separate file name and internal function name
fileparts	Parts of file name and path
filesep	Directory separator for current platform
fullfile	Build full filename from parts

tempdir	Name of system's temporary directory
tempname	Unique name for temporary file

Opening, Loading, Saving Files

daqread	Read Data Acquisition Toolbox (.daq) file
filehandle	Construct file handle object
importdata	Load data from disk file
load	Load workspace variables from disk
open	Open files based on extension
save	Save workspace variables to disk
uiimport	Open Import Wizard to import data
winopen	Open file in appropriate application (Windows)

Memory Mapping

disp (memmapfile)	Information about memmapfile object
get (memmapfile)	Memmapfile object properties
memmapfile	Construct memmapfile object

Low-Level File I/O

fclose	Close one or more open files
feof	Test for end-of-file
ferror	Query MATLAB about errors in file input or output

fgetl	Read line from file, discarding newline character
fgets	Read line from file, keeping newline character
fopen	Open file, or obtain information about open files
fprintf	Write formatted data to file
fread	Read binary data from file
frewind	Move file position indicator to beginning of open file
fscanf	Read formatted data from file
fseek	Set file position indicator
ftell	File position indicator
fwrite	Write binary data to file

Text Files

csvread	Read comma-separated value file
csvwrite	Write comma-separated value file
dlmread	Read ASCII-delimited file of numeric data into matrix
dlmwrite	Write matrix to ASCII-delimited file
textread	Read data from text file; write to multiple outputs
textscan	Read formatted data from text file or string

XML Documents

xmlread	Parse XML document and return Document Object Model node
xmlwrite	Serialize XML Document Object Model node
xslt	Transform XML document using XSLT engine

Spreadsheets

Microsoft Excel Functions (p. 1-79)	Read and write Microsoft Excel spreadsheet
Lotus 1-2-3 Functions (p. 1-79)	Read and write Lotus WK1 spreadsheet

Microsoft Excel Functions

xlsfinfo	Determine whether file contains Microsoft Excel (`.xls`) spreadsheet
xlsread	Read Microsoft Excel spreadsheet file (`.xls`)
xlswrite	Write Microsoft Excel spreadsheet file (`.xls`)

Lotus 1-2-3 Functions

wk1finfo	Determine whether file contains 1-2-3 WK1 worksheet
wk1read	Read Lotus 1-2-3 WK1 spreadsheet file into matrix
wk1write	Write matrix to Lotus 1-2-3 WK1 spreadsheet file

Scientific Data

Common Data Format (CDF) (p. 1-80)	Work with CDF files
Flexible Image Transport System (p. 1-80)	Work with FITS files
Hierarchical Data Format (HDF) (p. 1-81)	Work with HDF files
Band-Interleaved Data (p. 1-81)	Work with band-interleaved files

Common Data Format (CDF)

cdfepoch	Construct `cdfepoch` object for Common Data Format (CDF) export
cdfinfo	Information about Common Data Format (CDF) file
cdfread	Read data from Common Data Format (CDF) file
cdfwrite	Write data to Common Data Format (CDF) file
todatenum	Convert CDF epoch object to MATLAB datenum

Flexible Image Transport System

fitsinfo	Information about FITS file
fitsread	Read data from FITS file

Hierarchical Data Format (HDF)

hdf	Summary of MATLAB HDF4 capabilities
hdf5	Summary of MATLAB HDF5 capabilities
hdf5info	Information about HDF5 file
hdf5read	Read HDF5 file
hdf5write	Write data to file in HDF5 format
hdfinfo	Information about HDF4 or HDF-EOS file
hdfread	Read data from HDF4 or HDF-EOS file
hdftool	Browse and import data from HDF4 or HDF-EOS files

Band-Interleaved Data

multibandread	Read band-interleaved data from binary file
multibandwrite	Write band-interleaved data to file

Audio and Audio/Video

General (p. 1-82)	Create audio player object, obtain information about multimedia files, convert to/from audio signal
SPARCstation-Specific Sound Functions (p. 1-82)	Access NeXT/SUN (`.au`) sound files

Microsoft WAVE Sound Functions (p. 1-83)	Access Microsoft WAVE (`.wav`) sound files
Audio/Video Interleaved (AVI) Functions (p. 1-83)	Access Audio/Video interleaved (`.avi`) sound files

General

audioplayer	Create audio player object
audiorecorder	Create audio recorder object
beep	Produce beep sound
lin2mu	Convert linear audio signal to mu-law
mmfileinfo	Information about multimedia file
mmreader	Create multimedia reader object for reading video files
mu2lin	Convert mu-law audio signal to linear
read	Read video frame data from multimedia reader object
sound	Convert vector into sound
soundsc	Scale data and play as sound

SPARCstation-Specific Sound Functions

aufinfo	Information about NeXT/SUN (`.au`) sound file
auread	Read NeXT/SUN (`.au`) sound file
auwrite	Write NeXT/SUN (`.au`) sound file

Microsoft WAVE Sound Functions

wavfinfo	Information about Microsoft WAVE (`.wav`) sound file
wavplay	Play recorded sound on PC-based audio output device
wavread	Read Microsoft WAVE (`.wav`) sound file
wavrecord	Record sound using PC-based audio input device
wavwrite	Write Microsoft WAVE (`.wav`) sound file

Audio/Video Interleaved (AVI) Functions

addframe	Add frame to Audio/Video Interleaved (AVI) file
avifile	Create new Audio/Video Interleaved (AVI) file
aviinfo	Information about Audio/Video Interleaved (AVI) file
aviread	Read Audio/Video Interleaved (AVI) file
close (avifile)	Close Audio/Video Interleaved (AVI) file
movie2avi	Create Audio/Video Interleaved (AVI) movie from MATLAB movie

Images

exifread	Read EXIF information from JPEG and TIFF image files
im2java	Convert image to Java image

imfinfo	Information about graphics file
imread	Read image from graphics file
imwrite	Write image to graphics file

Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-84)	Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files
FTP Functions (p. 1-84)	Connect to FTP server, download from server, manage FTP files, close server connection

URL, Zip, Tar, E-Mail

gunzip	Uncompress GNU zip files
gzip	Compress files into GNU zip files
sendmail	Send e-mail message to address list
tar	Compress files into tar file
untar	Extract contents of tar file
unzip	Extract contents of zip file
urlread	Read content at URL
urlwrite	Save contents of URL to file
zip	Compress files into zip file

FTP Functions

ascii	Set FTP transfer type to ASCII
binary	Set FTP transfer type to binary

cd (ftp)	Change current directory on FTP server
close (ftp)	Close connection to FTP server
delete (ftp)	Remove file on FTP server
dir (ftp)	Directory contents on FTP server
ftp	Connect to FTP server, creating FTP object
mget	Download file from FTP server
mkdir (ftp)	Create new directory on FTP server
mput	Upload file or directory to FTP server
rename	Rename file on FTP server
rmdir (ftp)	Remove directory on FTP server

Graphics

Basic Plots and Graphs (p. 1-86)	Linear line plots, log and semilog plots
Plotting Tools (p. 1-87)	GUIs for interacting with plots
Annotating Plots (p. 1-87)	Functions for and properties of titles, axes labels, legends, mathematical symbols
Specialized Plotting (p. 1-88)	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images (p. 1-92)	Display image object, read and write graphics file, convert to movie frames
Printing (p. 1-92)	Printing and exporting figures to standard formats
Handle Graphics (p. 1-93)	Creating graphics objects, setting properties, finding handles

Basic Plots and Graphs

box	Axes border
errorbar	Plot error bars along curve
hold	Retain current graph in figure
LineSpec	Line specification string syntax
loglog	Log-log scale plot
plot	2-D line plot
plot3	3-D line plot
plotyy	2-D line plots with y-axes on both left and right side
polar	Polar coordinate plot

semilogx, semilogy	Semilogarithmic plots
subplot	Create axes in tiled positions

Plotting Tools

figurepalette	Show or hide figure palette
pan	Pan view of graph interactively
plotbrowser	Show or hide figure plot browser
plotedit	Interactively edit and annotate plots
plottools	Show or hide plot tools
propertyeditor	Show or hide property editor
rotate3d	Rotate 3-D view using mouse
showplottool	Show or hide figure plot tool
zoom	Turn zooming on or off or magnify by factor

Annotating Plots

annotation	Create annotation objects
clabel	Contour plot elevation labels
datacursormode	Enable or disable interactive data cursor mode
datetick	Date formatted tick labels
gtext	Mouse placement of text in 2-D view
legend	Graph legend for lines and patches
line	Create line object
rectangle	Create 2-D rectangle object
texlabel	Produce TeX format from character string

title	Add title to current axes
xlabel, ylabel, zlabel	Label *x*-, *y*-, and *z*-axis

Specialized Plotting

Area, Bar, and Pie Plots (p. 1-88)	1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-89)	Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-89)	Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-89)	Stair, step, and stem plots
Function Plots (p. 1-89)	Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-90)	Plots for showing distributions of data
Polygons and Surfaces (p. 1-90)	Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-91)	Plots of point distributions
Animation (p. 1-91)	Functions to create and play movies of plots

Area, Bar, and Pie Plots

area	Filled area 2-D plot
bar, barh	Plot bar graph (vertical and horizontal)
bar3, bar3h	Plot 3-D bar chart
pareto	Pareto chart
pie	Pie chart
pie3	3-D pie chart

Contour Plots

contour	Contour plot of matrix
contour3	3-D contour plot
contourc	Low-level contour plot computation
contourf	Filled 2-D contour plot
ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter

Direction and Velocity Plots

comet	2-D comet plot
comet3	3-D comet plot
compass	Plot arrows emanating from origin
feather	Plot velocity vectors
quiver	Quiver or velocity plot
quiver3	3-D quiver or velocity plot

Discrete Data Plots

stairs	Stairstep graph
stem	Plot discrete sequence data
stem3	Plot 3-D discrete sequence data

Function Plots

ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter
ezmesh	Easy-to-use 3-D mesh plotter

ezmeshc	Easy-to-use combination mesh/contour plotter
ezplot	Easy-to-use function plotter
ezplot3	Easy-to-use 3-D parametric curve plotter
ezpolar	Easy-to-use polar coordinate plotter
ezsurf	Easy-to-use 3-D colored surface plotter
ezsurfc	Easy-to-use combination surface/contour plotter
fplot	Plot function between specified limits

Histograms

hist	Histogram plot
histc	Histogram count
rose	Angle histogram plot

Polygons and Surfaces

convhull	Convex hull
cylinder	Generate cylinder
delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
ellipsoid	Generate ellipsoid

fill	Filled 2-D polygons
fill3	Filled 3-D polygons
inpolygon	Points inside polygonal region
pcolor	Pseudocolor (checkerboard) plot
polyarea	Area of polygon
rectint	Rectangle intersection area
ribbon	Ribbon plot
slice	Volumetric slice plot
sphere	Generate sphere
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search
voronoi	Voronoi diagram
waterfall	Waterfall plot

Scatter/Bubble Plots

plotmatrix	Scatter plot matrix
scatter	Scatter plot
scatter3	3-D scatter plot

Animation

frame2im	Convert movie frame to indexed image
getframe	Capture movie frame
im2frame	Convert image to movie frame

movie	Play recorded movie frames
noanimate	Change `EraseMode` of all objects to `normal`

Bit-Mapped Images

frame2im	Convert movie frame to indexed image
im2frame	Convert image to movie frame
im2java	Convert image to Java image
image	Display image object
imagesc	Scale data and display image object
imfinfo	Information about graphics file
imformats	Manage image file format registry
imread	Read image from graphics file
imwrite	Write image to graphics file
ind2rgb	Convert indexed image to RGB image

Printing

frameedit	Edit print frames for Simulink and Stateflow block diagrams
hgexport	Export figure
orient	Hardcopy paper orientation
print, printopt	Print figure or save to file and configure printer defaults
printdlg	Print dialog box

printpreview	Preview figure to print
saveas	Save figure or Simulink block diagram using specified format

Handle Graphics

Finding and Identifying Graphics Objects (p. 1-93)	Find and manipulate graphics objects via their handles
Object Creation Functions (p. 1-94)	Constructors for core graphics objects
Plot Objects (p. 1-94)	Property descriptions for plot objects
Figure Windows (p. 1-95)	Control and save figures
Axes Operations (p. 1-96)	Operate on axes objects
Operating on Object Properties (p. 1-96)	Query, set, and link object properties

Finding and Identifying Graphics Objects

allchild	Find all children of specified objects
ancestor	Ancestor of graphics object
copyobj	Copy graphics objects and their descendants
delete	Remove files or graphics objects
findall	Find all graphics objects
findfigs	Find visible offscreen figures
findobj	Locate graphics objects with specific properties
gca	Current axes handle
gcbf	Handle of figure containing object whose callback is executing

gcbo	Handle of object whose callback is executing
gco	Handle of current object
get	Query object properties
ishandle	Is object handle valid
propedit	Open Property Editor
set	Set object properties

Object Creation Functions

axes	Create axes graphics object
figure	Create figure graphics object
hggroup	Create hggroup object
hgtransform	Create hgtransform graphics object
image	Display image object
light	Create light object
line	Create line object
patch	Create patch graphics object
rectangle	Create 2-D rectangle object
root object	Root object properties
surface	Create surface object
text	Create text object in current axes
uicontextmenu	Create context menu

Plot Objects

Annotation Arrow Properties	Define annotation arrow properties
Annotation Doublearrow Properties	Define annotation doublearrow properties

Annotation Ellipse Properties	Define annotation ellipse properties
Annotation Line Properties	Define annotation line properties
Annotation Rectangle Properties	Define annotation rectangle properties
Annotation Textarrow Properties	Define annotation textarrow properties
Annotation Textbox Properties	Define annotation textbox properties
Areaseries Properties	Define areaseries properties
Barseries Properties	Define barseries properties
Contourgroup Properties	Define contourgroup properties
Errorbarseries Properties	Define errorbarseries properties
Image Properties	Define image properties
Lineseries Properties	Define lineseries properties
Quivergroup Properties	Define quivergroup properties
Scattergroup Properties	Define scattergroup properties
Stairseries Properties	Define stairseries properties
Stemseries Properties	Define stemseries properties
Surfaceplot Properties	Define surfaceplot properties

Figure Windows

clf	Clear current figure window
close	Remove specified figure
closereq	Default figure close request function
drawnow	Flushes event queue and updates figure window
gcf	Current figure handle
hgload	Load Handle Graphics object hierarchy from file

hgsave	Save Handle Graphics object hierarchy to file
newplot	Determine where to draw graphics objects
opengl	Control OpenGL rendering
refresh	Redraw current figure
saveas	Save figure or Simulink block diagram using specified format

Axes Operations

axis	Axis scaling and appearance
box	Axes border
cla	Clear current axes
gca	Current axes handle
grid	Grid lines for 2-D and 3-D plots
ishold	Current hold state
makehgtform	Create 4-by-4 transform matrix

Operating on Object Properties

get	Query object properties
linkaxes	Synchronize limits of specified 2-D axes
linkprop	Keep same value for corresponding properties
refreshdata	Refresh data in graph when data source is specified
set	Set object properties

3-D Visualization

Surface and Mesh Plots

Creating Surfaces and Meshes

hidden	Remove hidden lines from mesh plot
mesh, meshc, meshz	Mesh plots
peaks	Example function of two variables
surf, surfc	3-D shaded surface plot
surface	Create surface object
surfl	Surface plot with colormap-based lighting

tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot

Domain Generation

griddata	Data gridding
meshgrid	Generate `X` and `Y` arrays for 3-D plots

Color Operations

brighten	Brighten or darken colormap
caxis	Color axis scaling
colorbar	Colorbar showing color scale
colordef	Set default property values to display different color schemes
colormap	Set and get current colormap
colormapeditor	Start colormap editor
ColorSpec	Color specification
graymon	Set default figure properties for grayscale monitors
hsv2rgb	Convert HSV colormap to RGB colormap
rgb2hsv	Convert RGB colormap to HSV colormap
rgbplot	Plot colormap
shading	Set color shading properties
spinmap	Spin colormap

surfnorm	Compute and display 3-D surface normals
whitebg	Change axes background color

Colormaps

contrast	Grayscale colormap for contrast enhancement

View Control

Controlling the Camera Viewpoint (p. 1-99)	Orbiting, dollying, pointing, rotating camera positions and setting fields of view
Setting the Aspect Ratio and Axis Limits (p. 1-100)	Specifying what portions of axes to view and how to scale them
Object Manipulation (p. 1-100)	Panning, rotating, and zooming views
Selecting Region of Interest (p. 1-101)	Interactively identifying rectangular regions

Controlling the Camera Viewpoint

camdolly	Move camera position and target
cameratoolbar	Control camera toolbar programmatically
camlookat	Position camera to view object or group of objects
camorbit	Rotate camera position around camera target
campan	Rotate camera target around camera position

campos	Set or query camera position
camproj	Set or query projection type
camroll	Rotate camera about view axis
camtarget	Set or query location of camera target
camup	Set or query camera up vector
camva	Set or query camera view angle
camzoom	Zoom in and out on scene
makehgtform	Create 4-by-4 transform matrix
view	Viewpoint specification
viewmtx	View transformation matrices

Setting the Aspect Ratio and Axis Limits

daspect	Set or query axes data aspect ratio
pbaspect	Set or query plot box aspect ratio
xlim, ylim, zlim	Set or query axis limits

Object Manipulation

pan	Pan view of graph interactively
reset	Reset graphics object properties to their defaults
rotate	Rotate object in specified direction
rotate3d	Rotate 3-D view using mouse
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
zoom	Turn zooming on or off or magnify by factor

Selecting Region of Interest

dragrect	Drag rectangles with mouse
rbbox	Create rubberband box for area selection

Lighting

camlight	Create or move light object in camera coordinates
diffuse	Calculate diffuse reflectance
light	Create light object
lightangle	Create or position `light` object in spherical coordinates
lighting	Specify lighting algorithm
material	Control reflectance properties of surfaces and patches
specular	Calculate specular reflectance

Transparency

alim	Set or query axes alpha limits
alpha	Set transparency properties for objects in current axes
alphamap	Specify figure alphamap (transparency)

Volume Visualization

coneplot	Plot velocity vectors as cones in 3-D vector field
contourslice	Draw contours in volume slice planes
curl	Compute curl and angular velocity of vector field
divergence	Compute divergence of vector field
flow	Simple function of three variables
interpstreamspeed	Interpolate stream-line vertices from flow speed
isocaps	Compute isosurface end-cap geometry
isocolors	Calculate isosurface and patch colors
isonormals	Compute normals of isosurface vertices
isosurface	Extract isosurface data from volume data
reducepatch	Reduce number of patch faces
reducevolume	Reduce number of elements in volume data set
shrinkfaces	Reduce size of patch faces
slice	Volumetric slice plot
smooth3	Smooth 3-D data
stream2	Compute 2-D streamline data
stream3	Compute 3-D streamline data
streamline	Plot streamlines from 2-D or 3-D vector data
streamparticles	Plot stream particles
streamribbon	3-D stream ribbon plot from vector volume data

streamslice	Plot streamlines in slice planes
streamtube	Create 3-D stream tube plot
subvolume	Extract subset of volume data set
surf2patch	Convert surface data to patch data
volumebounds	Coordinate and color limits for volume data

Creating Graphical User Interfaces

Predefined Dialog Boxes

dialog	Create and display dialog box
errordlg	Create and open error dialog box
export2wsdlg	Export variables to workspace
helpdlg	Create and open help dialog box
inputdlg	Create and open input dialog box
listdlg	Create and open list-selection dialog box
msgbox	Create and open message box
printdlg	Print dialog box
printpreview	Preview figure to print
questdlg	Create and open question dialog box
uigetdir	Open standard dialog box for selecting a directory

uigetfile	Open standard dialog box for retrieving files
uigetpref	Open dialog box for retrieving preferences
uiopen	Open file selection dialog box with appropriate file filters
uiputfile	Open standard dialog box for saving files
uisave	Open standard dialog box for saving workspace variables
uisetcolor	Open standard dialog box for setting object's `ColorSpec`
uisetfont	Open standard dialog box for setting object's font characteristics
waitbar	Open waitbar
warndlg	Open warning dialog box

Deploying User Interfaces

guidata	Store or retrieve GUI data
guihandles	Create structure of handles
movegui	Move GUI figure to specified location on screen
openfig	Open new copy or raise existing copy of saved figure

Developing User Interfaces

addpref	Add preference
getappdata	Value of application-defined data
getpref	Preference

ginput	Graphical input from mouse or cursor
guidata	Store or retrieve GUI data
guide	Open GUI Layout Editor
inspect	Open Property Inspector
isappdata	True if application-defined data exists
ispref	Test for existence of preference
rmappdata	Remove application-defined data
rmpref	Remove preference
setappdata	Specify application-defined data
setpref	Set preference
uigetpref	Open dialog box for retrieving preferences
uisetpref	Manage preferences used in `uigetpref`
waitfor	Wait for condition before resuming execution
waitforbuttonpress	Wait for key press or mouse-button click

User Interface Objects

menu	Generate menu of choices for user input
uibuttongroup	Create container object to exclusively manage radio buttons and toggle buttons
uicontextmenu	Create context menu
uicontrol	Create user interface control object

uimenu	Create menus on figure windows
uipanel	Create panel container object
uipushtool	Create push button on toolbar
uitoggletool	Create toggle button on toolbar
uitoolbar	Create toolbar on figure

Finding Objects from Callbacks

findall	Find all graphics objects
findfigs	Find visible offscreen figures
findobj	Locate graphics objects with specific properties
gcbf	Handle of figure containing object whose callback is executing
gcbo	Handle of object whose callback is executing

GUI Utility Functions

align	Align user interface controls (uicontrols) and axes
getpixelposition	Get component position in pixels
listfonts	List available system fonts
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
setpixelposition	Set component position in pixels
textwrap	Wrapped string matrix for given uicontrol
uistack	Reorder visual stacking order of objects

Controlling Program Execution

uiresume, uiwait	Control program execution

External Interfaces

Dynamic Link Libraries (p. 1-109)	Access functions stored in external shared library (.dll) files
Java (p. 1-110)	Work with objects constructed from Java API and third-party class packages
Component Object Model and ActiveX (p. 1-111)	Integrate COM components into your application
Web Services (p. 1-113)	Communicate between applications over a network using SOAP and WSDL
Serial Port Devices (p. 1-113)	Read and write to devices connected to your computer's serial port

See also MATLAB C and Fortran API Reference for functions you can use in external routines that interact with MATLAB programs and the data in MATLAB workspaces.

Dynamic Link Libraries

calllib	Call function in external library
libfunctions	Information on functions in external library
libfunctionsview	Create window displaying information on functions in external library
libisloaded	Determine whether external library is loaded
libpointer	Create pointer object for use with external libraries
libstruct	Construct structure as defined in external library

loadlibrary	Load external library into MATLAB
unloadlibrary	Unload external library from memory

Java

class	Create object or return class of object
fieldnames	Field names of structure, or public fields of object
import	Add package or class to current Java import list
inspect	Open Property Inspector
isa	Determine whether input is object of given class
isjava	Determine whether input is Java object
ismethod	Determine whether input is object method
isprop	Determine whether input is object property
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Set and get dynamic Java class path
javaMethod	Invoke Java method
javaObject	Construct Java object
javarmpath	Remove entries from dynamic Java class path
methods	Information on class methods

methodsview	Information on class methods in separate window
usejava	Determine whether Java feature is supported in MATLAB

Component Object Model and ActiveX

actxcontrol	Create ActiveX control in figure window
actxcontrollist	List all currently installed ActiveX controls
actxcontrolselect	Open GUI to create ActiveX control
actxGetRunningServer	Get handle to running instance of Automation server
actxserver	Create COM server
addproperty	Add custom property to object
class	Create object or return class of object
delete (COM)	Remove COM control or server
deleteproperty	Remove custom property from object
enableservice	Enable, disable, or report status of Automation server
eventlisteners	List of events attached to listeners
events	List of events control can trigger
Execute	Execute MATLAB command in server
Feval (COM)	Evaluate MATLAB function in server
fieldnames	Field names of structure, or public fields of object
get (COM)	Get property value from interface, or display properties

GetCharArray	Get character array from server
GetFullMatrix	Get matrix from server
GetVariable	Get data from variable in server workspace
GetWorkspaceData	Get data from server workspace
inspect	Open Property Inspector
interfaces	List custom interfaces to COM server
invoke	Invoke method on object or interface, or display methods
isa	Determine whether input is object of given class
iscom	Is input COM object
isevent	Is input event
isinterface	Is input COM interface
ismethod	Determine whether input is object method
isprop	Determine whether input is object property
load (COM)	Initialize control object from file
MaximizeCommandWindow	Open server window on Windows desktop
methods	Information on class methods
methodsview	Information on class methods in separate window
MinimizeCommandWindow	Minimize size of server window
move	Move or resize control in parent window
propedit (COM)	Open built-in property page for control
PutCharArray	Store character array in server

PutFullMatrix	Store matrix in server
PutWorkspaceData	Store data in server workspace
Quit (COM)	Terminate MATLAB server
registerevent	Register event handler with control's event
release	Release interface
save (COM)	Serialize control object to file
set (COM)	Set object or interface property to specified value
unregisterallevents	Unregister all events for control
unregisterevent	Unregister event handler with control's event

Web Services

callSoapService	Send SOAP message off to endpoint
createClassFromWsdl	Create MATLAB object based on WSDL file
createSoapMessage	Create SOAP message to send to server
parseSoapResponse	Convert response string from SOAP server into MATLAB data types

Serial Port Devices

clear (serial)	Remove serial port object from MATLAB workspace
delete (serial)	Remove serial port object from memory
disp (serial)	Serial port object summary information

fclose (serial)	Disconnect serial port object from device
fgetl (serial)	Read line of text from device and discard terminator
fgets (serial)	Read line of text from device and include terminator
fopen (serial)	Connect serial port object to device
fprintf (serial)	Write text to device
fread (serial)	Read binary data from device
fscanf (serial)	Read data from device, and format as text
fwrite (serial)	Write binary data to device
get (serial)	Serial port object properties
instrcallback	Event information when event occurs
instrfind	Read serial port objects from memory to MATLAB workspace
instrfindall	Find visible and hidden serial port objects
isvalid (serial)	Determine whether serial port objects are valid
length (serial)	Length of serial port object array
load (serial)	Load serial port objects and variables into MATLAB workspace
readasync	Read data asynchronously from device
record	Record data and event information to file
save (serial)	Save serial port objects and variables to MAT-file
serial	Create serial port object

serialbreak	Send break to device connected to serial port
set (serial)	Configure or display serial port object properties
size (serial)	Size of serial port object array
stopasync	Stop asynchronous read and write operations

2

Functions — Alphabetical List

binary
bitand
bitcmp
bitget
bitmax
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
builddocsearchdb
builtin
bsxfun
bvp4c
bvp5c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit
campan
campos
camproj
camroll
camtarget
camup

camva
camzoom
cart2pol
cart2sph
case
cast
cat
catch
caxis
cd
cd (ftp)
cdf2rdf
cdfepoch
cdfinfo
cdfread
cdfwrite
ceil
cell
cell2mat
cell2struct
celldisp
cellfun
cellplot
cellstr
cgs
char
checkin
checkout
chol
cholinc
cholupdate
circshift
cla
clabel
class
clc
clear

clear (serial)
clf
clipboard
clock
close
close (avifile)
close (ftp)
closereq
cmopts
colamd
colmmd
colorbar
colordef
colormap
colormapeditor
ColorSpec
colperm
comet
comet3
commandhistory
commandwindow
compan
compass
complex
computer
cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contourgroup Properties
contourslice

contrast
conv
conv2
convhull
convhulln
convn
copyfile
copyobj
corrcoef
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
createClassFromWsdl
createCopy (inputParser)
createSoapMessage
cross
csc
cscd
csch
csvread
csvwrite
ctranspose (timeseries)
cumprod
cumsum
cumtrapz
curl
customverctrl
cylinder
daqread
daspect
datacursormode

datatipinfo
date
datenum
datestr
datetick
datevec
dbclear
dbcont
dbdown
dblquad
dbmex
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeadv
ddeexec
ddeget
ddeinit
ddepoke
ddereq
ddesd
ddeset
ddeterm
ddeunadv
deal
deblank
debug
dec2base
dec2bin
dec2hex
decic
deconv

del2
delaunay
delaunay3
delaunayn
delete
delete (COM)
delete (ftp)
delete (serial)
delete (timer)
deleteproperty
delevent
delsample
delsamplefromcollection
demo
depdir
depfun
det
detrend
detrend (timeseries)
deval
diag
dialog
diary
diff
diffuse
dir
dir (ftp)
disp
disp (memmapfile)
disp (MException)
disp (serial)
disp (timer)
display
divergence
dlmread
dlmwrite
dmperm

eventlisteners
events
Execute
exifread
exist
exit
exp
expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh
ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
factor
factorial
false
fclose
fclose (serial)
feather
feof
ferror
feval
Feval (COM)
fft
fft2
fftn
fftshift
fftw
fgetl

fgetl (serial)
fgets
fgets (serial)
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
File Formats
filemarker
fileparts
filehandle
filesep
fill
fill3
filter
filter (timeseries)
filter2
find
findall
findfigs
findobj
findstr
finish
fitsinfo
fitsread
fix
flipdim
fliplr
flipud
floor
flops
flow
fminbnd
fminsearch
fopen

fopen (serial)
for
format
fplot
fprintf
fprintf (serial)
frame2im
frameedit
fread
fread (serial)
freqspace
frewind
fscanf
fscanf (serial)
fseek
ftell
ftp
full
fullfile
func2str
function
function_handle (@)
functions
funm
fwrite
fwrite (serial)
fzero
gallery
gamma, gammainc, gammaln
gca
gcbf
gcbo
gcd
gcf
gco
ge
genpath

genvarname
get
get (COM)
get (memmapfile)
get (serial)
get (timer)
get (timeseries)
get (tscollection)
getabstime (timeseries)
getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getenv
getfield
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getReport (MException)
getsampleusingtime (timeseries)
getsampleusingtime (tscollection)
gettimeseriesnames
gettsafteratevent
gettsafterevent
gettsatevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents
GetVariable
GetWorkspaceData
ginput
global
gmres
gplot

grabcode
gradient
graymon
grid
griddata
griddata3
griddatan
gsvd
gt
gtext
guidata
guide
guihandles
gunzip
gzip
hadamard
hankel
hdf
hdf5
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
hex2dec
hex2num
hgexport
hggroup
Hggroup Properties
hgload

hgsave
hgtransform
Hgtransform Properties
hidden
hilb
hist
histc
hold
home
horzcat
horzcat (tscollection)
hostid
hsv2rgb
hypot
i
idealfilter (timeseries)
idivide
if
ifft
ifft2
ifftn
ifftshift
ilu
im2frame
im2java
imag
image
Image Properties
imagesc
imfinfo
imformats
import
importdata
imread
imwrite
ind2rgb
ind2sub

Inf
inferiorto
info
inline
inmem
inpolygon
input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8, int16, int32, int64
interfaces
interp1
interp1q
interp2
interp3
interpft
interpn
interpstreamspeed
intersect
intmax
intmin
intwarning
inv
invhilb
invoke
ipermute
iqr (timeseries)
is*
isa
isappdata
iscell

iscellstr
ischar
iscom
isdir
isempty
isempty (timeseries)
isempty (tscollection)
isequal
isequal (MException)
isequalwithequalnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishold
isinf
isinteger
isinterface
isjava
iskeyword
isletter
islogical
ismac
ismember
ismethod
isnan
isnumeric
isobject
isocaps
isocolors
isonormals
isosurface
ispc
ispref
isprime

isprop
isreal
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
isunix
isvalid (serial)
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
javaMethod
javaObject
javarmpath
keyboard
kron
last (MException)
lasterr
lasterror
lastwarn
lcm
ldl
ldivide, rdivide
le
legend
legendre
length
length (serial)

length (timeseries)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec
linkaxes
linkprop
linsolve
linspace
listdlg
listfonts
load
load (COM)
load (serial)
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor

lower
ls
lscov
lsqnonneg
lsqr
lt
lu
luinc
magic
makehgtform
mat2cell
mat2str
material
matlabcolon (matlab:)
matlabrc
matlabroot
matlab (UNIX)
matlab (Windows)
max
max (timeseries)
MaximizeCommandWindow
maxNumCompThreads
mean
mean (timeseries)
median
median (timeseries)
memmapfile
memory
MException
menu
mesh, meshc, meshz
meshgrid
methods
methodsview
mex
mexext
mfilename

mget
min
min (timeseries)
MinimizeCommandWindow
minres
mislocked
mkdir
mkdir (ftp)
mkpp
mldivide \, mrdivide /
mlint
mlintrpt
mlock
mmfileinfo
mmreader
mod
mode
more
move
movefile
movegui
movie
movie2avi
mput
msgbox
mtimes
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin, nargout
nargoutchk
native2unicode
nchoosek

ndgrid
ndims
ne
ne (MException)
newplot
nextpow2
nnz
noanimate
nonzeros
norm
normest
not
notebook
now
nthroot
null
num2cell
num2hex
num2str
numel
nzmax
ode15i
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
odefile
odeget
odeset
odextend
ones
open
openfig
opengl
openvar
optimget
optimset
or
ordeig
orderfields

ordqz
ordschur
orient
orth
otherwise
pack
padecoef
pagesetupdlg
pan
pareto
parse (inputParser)
parseSoapResponse
partialpath
pascal
patch
Patch Properties
path
path2rc
pathdef
pathsep
pathtool
pause
pbaspect
pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie
pie3

pinv
planerot
playshow
plot
plot (timeseries)
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval
polyvalm
pow2
power
ppval
prefdir
preferences
primes
print, printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
propertyeditor
psi

publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quadgk
quadl
quadv
questdlg
quit
Quit (COM)
quiver
quiver3
Quivergroup Properties
qz
rand
randn
randperm
rank
rat, rats
rbbox
rcond
read
readasync
real
reallog
realmax
realmin
realpow
realsqrt
record

rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp, regexpi
regexprep
regexptranslate
registerevent
rehash
release
rem
removets
rename
repmat
resample (timeseries)
resample (tscollection)
reset
reshape
residue
restoredefaultpath
rethrow
rethrow (MException)
return
rgb2hsv
rgbplot
ribbon
rmappdata
rmdir
rmdir (ftp)
rmfield
rmpath
rmpref
root object

Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rref
rsf2csf
run
save
save (COM)
save (serial)
saveas
saveobj
savepath
scatter
scatter3
Scattergroup Properties
schur
script
sec
secd
sech
selectmoveresize
semilogx, semilogy
sendmail
serial
serialbreak
set
set (COM)
set (serial)
set (timer)
set (timeseries)
set (tscollection)
setabstime (timeseries)

setabstime (tscollection)
setappdata
setdiff
setenv
setfield
setinterpmethod
setpixelposition
setpref
setstr
settimeseriesnames
setxor
shading
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size (serial)
size (timeseries)
size (tscollection)
slice
smooth3
sort
sortrows
sound
soundsc
spalloc
sparse
spaugment
spconvert
spdiags
specular
speye

spfun
sph2cart
sphere
spinmap
spline
spones
spparms
sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stairs
Stairseries Properties
start
startat
startup
std
std (timeseries)
stem
stem3
Stemseries Properties
stop
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp, strcmpi
stream2

stream3
streamline
streamparticles
streamribbon
streamslice
streamtube
strfind
strings
strjust
strmatch
strncmp, strncmpi
strread
strrep
strtok
strtrim
struct
struct2cell
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substruct
subvolume
sum
sum (timeseries)
superiorto
support
surf, surfc
surf2patch
surface
Surface Properties
Surfaceplot Properties
surfl

surfnorm
svd
svds
swapbytes
switch
symamd
symbfact
symmlq
symmmd
symrcm
symvar
synchronize
syntax
system
tan
tand
tanh
tar
tempdir
tempname
tetramesh
texlabel
text
Text Properties
textread
textscan
textwrap
throw (MException)
throwAsCaller (MException)
tic, toc
timer
timerfind
timerfindall
timeseries
title
todatenum
toeplitz

toolboxdir
trace
transpose (timeseries)
trapz
treelayout
treeplot
tril
trimesh
triplequad
triplot
trisurf
triu
true
try
tscollection
tsdata.event
tsearch
tsearchn
tsprops
tstool
type
typecast
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties
uigetdir
uigetfile
uigetpref
uiimport
uimenu
Uimenu Properties
uint8, uint16, uint32, uint64
uiopen
uipanel

vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (timeseries)
vertcat (tscollection)
view
viewmtx
volumebounds
voronoi
voronoin
wait
waitbar
waitfor
waitforbuttonpress
warndlg
warning
waterfall
wavfinfo
wavplay
wavread
wavrecord
wavwrite
web
weekday
what
whatsnew
which
while
whitebg
who, whos
wilkinson
winopen
winqueryreg
wk1finfo

wk1read
wk1write
workspace
xlabel, ylabel, zlabel
xlim, ylim, zlim
xlsfinfo
xlsread
xlswrite
xmlread
xmlwrite
xor
xslt
zeros
zip
zoom

Purpose

Matrix and array arithmetic

Syntax

```
A+B
A-B
A*B
A.*B
A/B
A./B
A\B
A.\B
A^B
A.^B
A'
A.'
```

Description

MATLAB has two different types of arithmetic operations. Matrix arithmetic operations are defined by the rules of linear algebra. Array arithmetic operations are carried out element by element, and can be used with multidimensional arrays. The period character (`.`) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs `.+` and `.-` are not used.

`+` Addition or unary plus. `A+B` adds `A` and `B`. `A` and `B` must have the same size, unless one is a scalar. A scalar can be added to a matrix of any size.

`-` Subtraction or unary minus. `A-B` subtracts `B` from `A`. `A` and `B` must have the same size, unless one is a scalar. A scalar can be subtracted from a matrix of any size.

`*` Matrix multiplication. `C = A*B` is the linear algebraic product of the matrices `A` and `B`. More precisely,

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$$

For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. A scalar can multiply a matrix of any size.

`.*` Array multiplication. `A.*B` is the element-by-element product of the arrays `A` and `B`. `A` and `B` must have the same size, unless one of them is a scalar.

`/` Slash or matrix right division. `B/A` is roughly the same as `B*inv(A)`. More precisely, `B/A = (A'\B')'`. See the reference page for `mrdivide` for more information.

`./` Array right division. `A./B` is the matrix with elements `A(i,j)/B(i,j)`. `A` and `B` must have the same size, unless one of them is a scalar.

`\` Backslash or matrix left division. If `A` is a square matrix, `A\B` is roughly the same as `inv(A)*B`, except it is computed in a different way. If `A` is an `n`-by-`n` matrix and `B` is a column vector with `n` components, or a matrix with several such columns, then `X = A\B` is the solution to the equation $AX = B$ computed by Gaussian elimination. A warning message is displayed if `A` is badly scaled or nearly singular. See the reference page for `mldivide` for more information.

If A is an m-by-n matrix with `m ~= n` and B is a column vector with m components, or a matrix with several such columns, then `X = A\B` is the solution in the least squares sense to the under- or overdetermined system of equations $AX = B$. The effective rank, k, of A is determined from the QR decomposition with pivoting (see "Algorithm" on page 2-2183 for details). A solution X is computed that has at most k nonzero components per column. If `k < n`, this is usually not the same solution as `pinv(A)*B`, which is the least squares solution with the smallest norm $\|X\|$.

`.\` Array left division. `A.\B` is the matrix with elements `B(i,j)/A(i,j)`. A and B must have the same size, unless one of them is a scalar.

`^` Matrix power. `X^p` is X to the power p, if p is a scalar. If p is an integer, the power is computed by repeated squaring. If the integer is negative, X is inverted first. For other values of p, the calculation involves eigenvalues and eigenvectors, such that if `[V,D] = eig(X)`, then `X^p = V*D.^p/V`.

If x is a scalar and P is a matrix, `x^P` is x raised to the matrix power P using eigenvalues and eigenvectors. `X^P`, where X and P are both matrices, is an error.

`.^` Array power. `A.^B` is the matrix with elements `A(i,j)` to the `B(i,j)` power. A and B must have the same size, unless one of them is a scalar.

`'` Matrix transpose. `A'` is the linear algebraic transpose of A. For complex matrices, this is the complex conjugate transpose.

`.'` Array transpose. `A.'` is the array transpose of A. For complex matrices, this does not involve conjugation.

Nondouble Data Type Support

This section describes the arithmetic operators' support for data types other than `double`.

Data Type single

You can apply any of the arithmetic operators to arrays of type `single` and MATLAB returns an answer of type `single`. You can also combine an array of type `double` with an array of type `single`, and the result has type `single`.

Integer Data Types

You can apply most of the arithmetic operators to real arrays of the following integer data types:

- `int8` and `uint8`
- `int16` and `uint16`
- `int32` and `uint32`

All operands must have the same integer data type and MATLAB returns an answer of that type.

Note The arithmetic operators do not support operations on the data types `int64` or `uint64`. Except for the unary operators `+A` and `A.'`, the arithmetic operators do not support operations on complex arrays of any integer data type.

For example,

```
x = int8(3) + int8(4);
class(x)

ans =

int8
```

The following table lists the binary arithmetic operators that you can apply to arrays of the same integer data type. In the table, `A` and `B` are arrays of the same integer data type and `c` is a scalar of type `double` or the same type as `A` and `B`.

Operation	Support when A and B Have Same Integer Type
`+A, -A`	Yes
`A+B, A+c, c+B`	Yes
`A-B, A-c, c-B`	Yes
`A.*B`	Yes
`A*c, c*B`	Yes
`A*B`	No
`A/c, c/B`	Yes
`A.\B, A./B`	Yes
`A\B, A/B`	No
`A.^B`	Yes, if `B` has nonnegative integer values.
`c^k`	Yes, for a scalar `c` and a nonnegative scalar integer `k`, which have the same integer data type or one of which has type `double`
`A.', A'`	Yes

Combining Integer Data Types with Type Double

For the operations that support integer data types, you can combine a scalar or array of an integer data type with a scalar, but not an array, of type `double` and the result has the same integer data type as the input of integer type. For example,

```
y = 5 + int32(7);
class(y)
```

```
ans =

int32
```

However, you cannot combine an array of an integer data type with either of the following:

- A scalar or array of a different integer data type
- A scalar or array of type `single`

The section "Numeric Types", under "Data Types" in the MATLAB Programming documentation, provides more information about operations on nondouble data types.

Remarks

The arithmetic operators have M-file function equivalents, as shown:

Binary addition	`A+B`	`plus(A,B)`
Unary plus	`+A`	`uplus(A)`
Binary subtraction	`A-B`	`minus(A,B)`
Unary minus	`-A`	`uminus(A)`
Matrix multiplication	`A*B`	`mtimes(A,B)`
Arraywise multiplication	`A.*B`	`times(A,B)`
Matrix right division	`A/B`	`mrdivide(A,B)`
Arraywise right division	`A./B`	`rdivide(A,B)`
Matrix left division	`A\B`	`mldivide(A,B)`
Arraywise left division	`A.\B`	`ldivide(A,B)`

Matrix power	`A^B`	`mpower(A,B)`
Arraywise power	`A.^B`	`power(A,B)`
Complex transpose	`A'`	`ctranspose(A)`
Matrix transpose	`A.'`	`transpose(A)`

Note For some toolboxes, the arithmetic operators are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given operator, type `help` followed by the operator name. For example, type `help plus`. The toolboxes that overload `plus` (`+`) are listed. For information about using the operator in that toolbox, see the documentation for the toolbox.

Examples

Here are two vectors, and the results of various matrix and array operations on them, printed with `format rat`.

Matrix Operations		Array Operations	
`x`	1 2 3	`y`	4 5 6
`x'`	`1 2 3`	`y'`	`4 5 6`
`x+y`	5 7 9	`x-y`	`-3` `-3` `-3`
`x + 2`	3 4 5	`x-2`	`-1` 0 1

Matrix Operations		Array Operations	
`x * y`	Error	`x.*y`	4 10 18
`x'*y`	32	`x'.*y`	Error
`x*y'`	4 5 6 8 10 12 12 15 18	`x.*y'`	Error
`x*2`	2 4 6	`x.*2`	2 4 6
`x\y`	16/7	`x.\y`	4 5/2 2
`2\x`	1/2 1 3/2	`2./x`	2 1 2/3
`x/y`	0 0 1/6 0 0 1/3 0 0 1/2	`x./y`	1/4 2/5 1/2
`x/2`	1/2 1 3/2	`x./2`	1/2 1 3/2

Matrix Operations		Array Operations	

`x^y`	Error	`x.^y`	`1`
			`32`
			`729`

`x^2`	Error	`x.^2`	`1`
			`4`
			`9`

`2^x`	Error	`2.^x`	`2`
			`4`
			`8`

`(x+i*y)'`	`1 - 4i 2 - 5i` `3 - 6i`		
`(x+i*y).'`	`1 + 4i 2 + 5i` `3 + 6i`		

Diagnostics

- From matrix division, if a square `A` is singular,

  ```
  Warning: Matrix is singular to working precision.
  ```

- From elementwise division, if the divisor has zero elements,

  ```
  Warning: Divide by zero.
  ```

  Matrix division and elementwise division can produce `NaN`s or `Inf`s where appropriate.

- If the inverse was found, but is not reliable,

  ```
  Warning: Matrix is close to singular or badly scaled.
      Results may be inaccurate.  RCOND = xxx
  ```

- From matrix division, if a nonsquare `A` is rank deficient,

```
Warning: Rank deficient, rank = xxx tol = xxx
```

See Also

`mldivide`, `mrdivide`, `chol`, `det`, `inv`, `lu`, `orth`, `permute`, `ipermute`, `qr`, `rref`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

[2] Davis, T.A., *UMFPACK Version 4.6 User Guide* (`http://www.cise.ufl.edu/research/sparse/umfpack`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2002.

[3] Davis, T. A., *CHOLMOD Version 1.0 User Guide* (`http://www.cise.ufl.edu/research/sparse/cholmod`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2005.

Purpose

Relational operations

Syntax

```
A < B
A > B
A <= B
A >= B
A == B
A ~= B
```

Description

The relational operators are <, >, <=, >=, ==, and ~=. Relational operators perform element-by-element comparisons between two arrays. They return a `logical` array of the same size, with elements set to logical `1` (`true`) where the relation is true, and elements set to logical `0` (`false`) where it is not.

The operators <, >, <=, and >= use only the real part of their operands for the comparison. The operators == and ~= test real and imaginary parts.

To test if two strings are equivalent, use `strcmp`, which allows vectors of dissimilar length to be compared.

Note For some toolboxes, the relational operators are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given operator, type `help` followed by the operator name. For example, type `help lt`. The toolboxes that overload `lt (<)` are listed. For information about using the operator in that toolbox, see the documentation for the toolbox.

Examples

If one of the operands is a scalar and the other a matrix, the scalar expands to the size of the matrix. For example, the two pairs of statements

```
X = 5; X >= [1 2 3; 4 5 6; 7 8 10]
X = 5*ones(3,3); X >= [1 2 3; 4 5 6; 7 8 10]
```

produce the same result:

```
ans =

     1     1     1
     1     1     0
     0     0     0
```

See Also

all, any, find, strcmp

Logical Operators: Elementwise & | ~, Logical Operators: Short-circuit && ||

Purpose

Elementwise logical operations on arrays

Syntax

```
A & B
A | B
~A
```

Description

The symbols `&`, `|`, and `~` are the logical array operators `AND`, `OR`, and `NOT`. They work element by element on arrays, with logical `0` representing `false`, and logical `1` or any nonzero element representing `true`. The logical operators return a `logical` array with elements set to `1` (`true`) or `0` (`false`), as appropriate.

The `&` operator does a logical `AND`, the `|` operator does a logical `OR`, and `~A` complements the elements of `A`. The function `xor(A,B)` implements the exclusive `OR` operation. The truth table for these operators and functions is shown below.

Inputs		and	or	not	xor
A	B	A & B	A | B	~A	xor(A,B)
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

The precedence for the logical operators with respect to each other is

Operator	Operation	Priority
~	NOT	Highest
&	Elementwise AND	
|	Elementwise OR	
&&	Short-circuit AND	
||	Short-circuit OR	Lowest

Remarks

MATLAB always gives the & operator precedence over the | operator. Although MATLAB typically evaluates expressions from left to right, the expression `a|b&c` is evaluated as `a|(b&c)`. It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of & and |.

These logical operators have M-file function equivalents, as shown.

Logical Operation	Equivalent Function
`A & B`	`and(A,B)`
`A | B`	`or(A,B)`
`~A`	`not(A)`

Short-Circuiting in Elementwise Operators

When used in the context of an `if` or `while` expression, and only in this context, the elementwise | and & operators use short-circuiting in evaluating their expressions. That is, `A|B` and `A&B` ignore the second operand, `B`, if the first operand, `A`, is sufficient to determine the result.

So, although the statement `1|[]` evaluates to `false`, the same statement evaluates to `true` when used in either an `if` or `while` expression:

```
A = 1;   B = [];
if(A|B) disp 'The statement is true',  end;
   The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to `false`

```
if(B|A) disp 'The statement is true',  end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the following, which under most circumstances is invalid due to a size mismatch between `A` and `B`,

```
A = [1 1];   B = [2 0 1];
```

```
A|B        % This generates an error.
```

works within the context of an `if` or `while` expression:

```
if (A|B) disp 'The statement is true',  end;
   The statement is true
```

Examples

This example shows the logical `OR` of the elements in the vector `u` with the corresponding elements in the vector `v`:

```
u = [0 0 1 1 0 1];
v = [0 1 1 0 0 1];
u | v

ans =
   0   1   1   1   0   1
```

See Also

`all`, `any`, `find`, `logical`, `xor`, `true`, `false`

`Logical Operators:  Short-circuit && ||`

`Relational Operators < > <= >= == ~=`

Logical Operators: Short-circuit && ||

Purpose

Logical operations, with short-circuiting capability

Syntax

```
expr1 && expr2
expr1 || expr2
```

Description

`expr1 && expr2` represents a logical `AND` operation that employs short-circuiting behavior. With short-circuiting, the second operand `expr2` is evaluated only when the result is not fully determined by the first operand `expr1`. For example, if `A = 0`, then the following statement evaluates to `false`, regardless of the value of `B`, so MATLAB does not evaluate `B`:

```
A && B
```

These two expressions must each be a valid MATLAB statement that evaluates to a scalar logical result.

`expr1 || expr2` represents a logical `OR` operation that employs short-circuiting behavior.

Note Always use the `&&` and `||` operators when short-circuiting is required. Using the elementwise operators (`&` and `|`) for short-circuiting can yield unexpected results.

Examples

In the following statement, it doesn't make sense to evaluate the relation on the right if the divisor, `b`, is zero. The test on the left is put in to avoid generating a warning under these circumstances:

```
x = (b ~= 0) && (a/b > 18.5)
```

By definition, if any operands of an `AND` expression are `false`, the entire expression must be `false`. So, if `(b ~= 0)` evaluates to `false`, MATLAB assumes the entire expression to be `false` and terminates its evaluation of the expression early. This avoids the warning that would be generated if MATLAB were to evaluate the operand on the right.

See Also

all, any, find, logical, xor, true, false

Logical Operators: Elementwise & | ~

Relational Operators < > <= >= == ~=

Special Characters [ ] ( ) {} = ' . ... , ; : % ! @

Purpose

Special characters

Syntax

```
[ ]
{ }
( )
=
'
.
.
.( )
..
...
,
;
:
%
%{ %}
!
@
```

Description

`[ ]` Brackets are used to form vectors and matrices. `[6.9 9.64 sqrt(-1)]` is a vector with three elements separated by blanks. `[6.9, 9.64, i]` is the same thing. `[1+j 2-j 3]` and `[1 +j 2 -j 3]` are not the same. The first has three elements, the second has five.

`[11 12 13; 21 22 23]` is a 2-by-3 matrix. The semicolon ends the first row.

Vectors and matrices can be used inside `[ ]` brackets. `[A B;C]` is allowed if the number of rows of `A` equals the number of rows of `B` and the number of columns of `A` plus the number of columns of `B` equals the number of columns of `C`. This rule generalizes in a hopefully obvious way to allow fairly complicated constructions.

`A = [ ]` stores an empty matrix in `A`. `A(m,:) = [ ]` deletes row `m` of `A`. `A(:,n) = [ ]` deletes column `n` of `A`. `A(n) = [ ]` reshapes `A` into a column vector and deletes the third element.

`[A1,A2,A3...] = function` assigns function output to multiple variables.

For the use of `[` and `]` on the left of an "=" in multiple assignment statements, see `lu`, `eig`, `svd`, and so on.

`{ }` Curly braces are used in cell array assignment statements. For example, `A(2,1) = {[1 2 3; 4 5 6]}`, or `A{2,2} = 'str'`. See `help paren` for more information about `{ }`.

`( )` Parentheses are used to indicate precedence in arithmetic expressions in the usual way. They are used to enclose arguments of functions in the usual way. They are also used to enclose subscripts of vectors and matrices in a manner somewhat more general than usual. If `X` and `V` are vectors, then `X(V)` is `[X(V(1)), X(V(2)), ..., X(V(n))]`. The components of `V` must be integers to be used as subscripts. An error occurs if any such subscript is less than 1 or greater than the size of `X`. Some examples are

- `X(3)` is the third element of `X`.
- `X([1 2 3])` is the first three elements of `X`.

See `help paren` for more information about `( )`.

If `X` has `n` components, `X(n: 1:1)` reverses them. The same indirect subscripting works in matrices. If `V` has `m` components and `W` has `n` components, then `A(V,W)` is the `m`-by-`n` matrix formed from the elements of `A` whose subscripts are the elements of `V` and `W`. For example, `A([1,5],:) = A([5,1],:)` interchanges rows 1 and 5 of `A`.

`=` Used in assignment statements. `B = A` stores the elements of `A` in `B`. == is the relational equals operator. See the `Relational Operators < > <= >= == ~=` page.

`'` Matrix transpose. `X'` is the complex conjugate transpose of `X`. `X.'` is the nonconjugate transpose.

Quotation mark. `'any text'` is a vector whose components are the ASCII codes for the characters. A quotation mark within the text is indicated by two quotation marks.

`.` Decimal point. `314/100`, `3.14`, and `.314e1` are all the same.

Element-by-element operations. These are obtained using `.*`, `.^`, `./`, or `.\`. See the Arithmetic Operators page.

`.` Field access. `S(m).f` when `S` is a structure, accesses the contents of field `f` of that structure.

`.(` `)` Dynamic Field access. `S.(df)` when `A` is a structure, accesses the contents of dynamic field `df` of that structure. Dynamic field names are defined at runtime.

`..` Parent directory. See `cd`.

`...` Continuation. Three or more periods at the end of a line continue the current function on the next line. Three or more periods before the end of a line cause MATLAB to ignore the remaining text on the current line and continue the function on the next line. This effectively makes a comment out of anything on the current line that follows the three periods. See "Entering Long Statements (Line Continuation)" for more information.

`,` Comma. Used to separate matrix subscripts and function arguments. Used to separate statements in multistatement lines. For multistatement lines, the comma can be replaced by a semicolon to suppress printing.

`;` Semicolon. Used inside brackets to end rows. Used after an expression or statement to suppress printing or to separate statements.

`:` Colon. Create vectors, array subscripting, and `for` loop iterations. See `colon (:)` for details.

`%` Percent. The percent symbol denotes a comment; it indicates a logical end of line. Any following text is ignored. MATLAB displays the first contiguous comment lines in a M-file in response to a `help` command.

`%{` `%}` Percent-brace. The text enclosed within the `%{` and `%}` symbols is a comment block. Use these symbols to insert comments that take up more than a single line in your M-file code. Any text between these two symbols is ignored by MATLAB.

With the exception of whitespace characters, the %{ and %} operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

`!`	Exclamation point. Indicates that the rest of the input line is issued as a command to the operating system. See "Running External Programs" for more information.
`@`	Function handle. MATLAB data type that is a handle to a function. See `function_handle (@)` for details.

Remarks

Some uses of special characters have M-file function equivalents, as shown:

Horizontal concatenation	`[A,B,C...]`	`horzcat(A,B,C...)`
Vertical concatenation	`[A;B;C...]`	`vertcat(A,B,C...)`
Subscript reference	`A(i,j,k...)`	`subsref(A,S)`. See help `subsref`.
Subscript assignment	`A(i,j,k...)=B`	`subsasgn(A,S,B)`. See help `subsasgn`.

Note For some toolboxes, the special characters are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given character, type `help` followed by the character name. For example, type `help transpose`. The toolboxes that overload `transpose (.')` are listed. For information about using the character in that toolbox, see the documentation for the toolbox.

See Also

`Arithmetic Operators + - * / \ ^ '`

`Relational Operators < > <= >= == ~=`

`Logical Operators:  Elementwise & | ~,`

Purpose

Create vectors, array subscripting, and `for`-loop iterators

Description

The colon is one of the most useful operators in MATLAB. It can create vectors, subscript arrays, and specify `for` iterations.

The colon operator uses the following rules to create regularly spaced vectors:

`j:k`	is the same as `[j,j+1,...,k]`
`j:k`	is empty if `j > k`
`j:i:k`	is the same as `[j,j+i,j+2i, ...,k]`
`j:i:k`	is empty if `i == 0`, if `i > 0` and `j > k`, or if `i < 0` and `j < k`

where `i`, `j`, and `k` are all scalars.

Below are the definitions that govern the use of the colon to pick out selected rows, columns, and elements of vectors, matrices, and higher-dimensional arrays:

`A(:,j)`	is the `j`th column of `A`
`A(i,:)`	is the `i`th row of `A`
`A(:,:)`	is the equivalent two-dimensional array. For matrices this is the same as `A`.
`A(j:k)`	is `A(j), A(j+1),...,A(k)`
`A(:,j:k)`	is `A(:,j), A(:,j+1),...,A(:,k)`
`A(:,:,k)`	is the `k`th page of three-dimensional array `A`.
`A(i,j,k,:)`	is a vector in four-dimensional array `A`. The vector includes `A(i,j,k,1)`, `A(i,j,k,2)`, `A(i,j,k,3)`, and so on.
`A(:)`	is all the elements of `A`, regarded as a single column. On the left side of an assignment statement, `A(:)` fills `A`, preserving its shape from before. In this case, the right side must contain the same number of elements as `A`.

Examples

Using the colon with integers,

```
D = 1:4
```

results in

```
D =
    1    2    3    4
```

Using two colons to create a vector with arbitrary real increments between the elements,

```
E = 0:.1:.5
```

results in

```
E =
    0    0.1000    0.2000    0.3000    0.4000    0.5000
```

The command

```
A(:,:,2) = pascal(3)
```

generates a three-dimensional array whose first page is all zeros.

```
A(:,:,1) =
     0     0     0
     0     0     0
     0     0     0

A(:,:,2) =
     1     1     1
     1     2     3
     1     3     6
```

Using a colon with characters to iterate a for-loop,

```
for x='a':'d',x,end
```

results in

```
x =
     a
x =
     b
x =
     c
x =
     d
```

See Also

`for`, `linspace`, `logspace`, `reshape`

abs

Purpose

Absolute value and complex magnitude

Syntax

```
abs(X)
```

Description

`abs(X)` returns an array `Y` such that each element of `Y` is the absolute value of the corresponding element of `X`.

If `X` is complex, `abs(X)` returns the complex modulus (magnitude), which is the same as

```
sqrt(real(X).^2 + imag(X).^2)
```

Examples

```
abs(-5)
ans =
     5

abs(3+4i)
ans =
     5
```

See Also

`angle`, `sign`, `unwrap`

Purpose

Construct array with accumulation

Syntax

```
A = accumarray(subs, val)
A = accumarray(subs,val,sz)
A = accumarray(subs,val,sz,fun)
A = accumarray(subs,val,sz,fun,fillval)
A = accumarray(subs,val,sz,fun,fillval,issparse)
A = accumarray({subs1, subs2, ...}, val, ...)
```

Description

`A = accumarray(subs, val)` creates an array `A` by accumulating elements of the vector `val` using the subscript in `subs`. Each row of the `m`-by-`n` matrix `subs` defines an N-dimensional subscript into the output `A`. Each element of `val` has a corresponding row in `subs`. `accumarray` collects all elements of `val` that correspond to identical subscripts in `subs`, sums those values, and stores the result in the element of `A` that corresponds to the subscript. Elements of `A` that are not referred to by any row of `subs` contain zero.

If `subs` is a nonempty matrix with N>1 columns, then `A` is an N-dimensional array of size `max(subs,[],1)`. If `subs` is empty with N>1 columns, then `A` is an N-dimensional empty array with size 0-by-0-by-`...`-by-0. `subs` can also be a column vector, in which case a second column of ones is implied, and `A` is a column vector. `subs` must contain positive integers.

`subs` can also be a cell vector with one or more elements, each element a vector of positive integers. All the vectors must have the same length. In this case, `subs` is treated as if the vectors formed columns of an index matrix.

`val` must be a numeric, logical, or character vector with the same length as the number of rows in `subs`. `val` can also be a scalar whose value is repeated for all the rows of `subs`.

`accumarray` sums values from `val` using the default behavior of `sum`.

`A = accumarray(subs,val,sz)` creates an array `A` with size `sz`, where `sz` is a vector of positive integers. If `subs` is nonempty with N>1 columns, then `sz` must have N elements, where `all(sz >=`

`max(subs,[],1))`. If `subs` is a nonempty column vector, then `sz` must be `[M 1]`, where `M >= MAX(subs)`. Specify `sz` as `[]` for the default behavior.

`A = accumarray(subs,val,sz,fun)` applies function `fun` to each subset of elements of `val`. You must specify the `fun` input using the `@` symbol (e.g., `@sin`). The function `fun` must accept a column vector and return a numeric, logical, or character scalar, or a scalar cell. Return value `A` has the same class as the values returned by `fun`. Specify `fun` as `[]` for the default behavior. `fun` is `@sum` by default.

Note If the subscripts in `subs` are not sorted, `fun` should not depend on the order of the values in its input data.

`A = accumarray(subs,val,sz,fun,fillval)` puts the scalar value `fillval` in elements of `A` that are not referred to by any row of `subs`. For example, if `subs` is empty, then `A` is `repmat(fillval,sz)`. `fillval` and the values returned by `fun` must belong to the same class.

`A = accumarray(subs,val,sz,fun,fillval,issparse)` creates an array `A` that is sparse if the scalar input `issparse` is equal to logical 1 (i.e., `true`), or full if `issparse` is equal to logical 0 (`false`). `A` is full by default. If `issparse` is `true`, then `fillval` must be zero or `[]`, and `val` and the output of `fun` must be double.

`A = accumarray({subs1, subs2, ...}, val, ...)` passes multiple `subs` vectors in a cell array. You can use any of the four optional inputs (`sz`, `fun`, `fillval`, or `issparse`) with this syntax.

Examples

Example 1

Create a 5-by-1 vector, and sum values for repeated 1-dimensional subscripts:

```
val = 101:105;
subs = [1; 2; 4; 2; 4]
subs =
```

```
     1       % Subscript 1 of result <= val(1)
     2       % Subscript 2 of result <= val(2)
     4       % Subscript 4 of result <= val(3)
     2       % Subscript 2 of result <= val(4)
     4       % Subscript 4 of result <= val(5)

A = accumarray(subs, val)
A =
   101       % A(1) = val(1) = 101
   206       % A(2) = val(2)+val(4) = 102+104 = 206
     0       % A(3) = 0
   208       % A(4) = val(3)+val(5) = 103+105 = 208
```

Example 2

Create a 2-by-3-by-2 array, and sum values for repeated three-dimensional subscripts:

```
val = 101:105;
subs = [1 1 1; 2 1 2; 2 3 2; 2 1 2; 2 3 2];

A = accumarray(subs, val)
A(:,:,1) =
   101     0     0
     0     0     0
A(:,:,2) =
     0     0     0
   206     0   208
```

Example 3

Create a 2-by-3-by-2 array, and sum values natively:

```
val = 101:105;
subs = [1 1 1; 2 1 2; 2 3 2; 2 1 2; 2 3 2];

A = accumarray(subs, int8(val), [], @(x) sum(x,'native'))
A(:,:,1) =
  101    0    0
```

```
    0    0    0
A(:,:,2) =
    0    0    0
  127    0  127

class(A)
ans =
    int8
```

Example 4

Pass multiple subscript arguments in a cell array.

Create a 12-element vector `V`:

```
V = 101:112;
```

Create three 12-element vectors, one for each dimension of the resulting array `A`. Note how the indices of these vectors determine which elements of `V` are accumulated in `A`:

```
%        index 1   index 6 => V(1)+V(6) => A(1,3,1)
%          |         |
rowsubs = [1 3 3 2 3 1 2 2 3 3 1 2];
colsubs = [3 4 2 1 4 3 4 2 2 4 3 4];
pagsubs = [1 1 2 2 1 1 2 1 1 1 2 2];
%               |
%             index 4 => V(4) => A(2,1,2)
%
% A(1,3,1) = V(1) + V(6) = 101 + 106 = 207
% A(2,1,2) = V(4) = 104
```

Call `accumarray`, passing the subscript vectors in a cell array:

```
A = accumarray({rowsubs colsubs pagsubs}, V)
A(:,:,1) =
     0     0   207     0        % A(1,3,1) is 207
     0   108     0     0
     0   109     0   317
```

```
A(:,:,2) =
     0     0   111     0
   104     0     0   219        % A(2,1,2) is 104
     0   103     0     0
```

Example 5

Create an array with the `max` function, and fill all empty elements of that array with `NaN`:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @max, NaN)
A =
   101   NaN   NaN   NaN
   104   NaN   105   NaN
```

Example 6

Create a sparse matrix using the `prod` function:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @prod, 0, true)
A =
   (1,1)           101
   (2,1)         10608
   (2,3)         10815
```

Example 7

Count the number of subscripts for each bin:

```
val = 1;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4])
A =
```

```
     1     0     0     0
     2     0     2     0
```

Example 8

Create a logical array that shows which bins have two or more values:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @(x) length(x) > 1)
A =
     0     0     0     0
     1     0     1     0
```

Example 9

Group values in a cell array:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @(x) {x})
A =
    [       101]     []               []     []
    [2x1 double]     []    [2x1 double]     []

A{2}
ans =
   104
   102
```

See Also

`full`, `sparse`, `sum`

Purpose

Inverse cosine; result in radians

Syntax

```
Y = acos(X)
```

Description

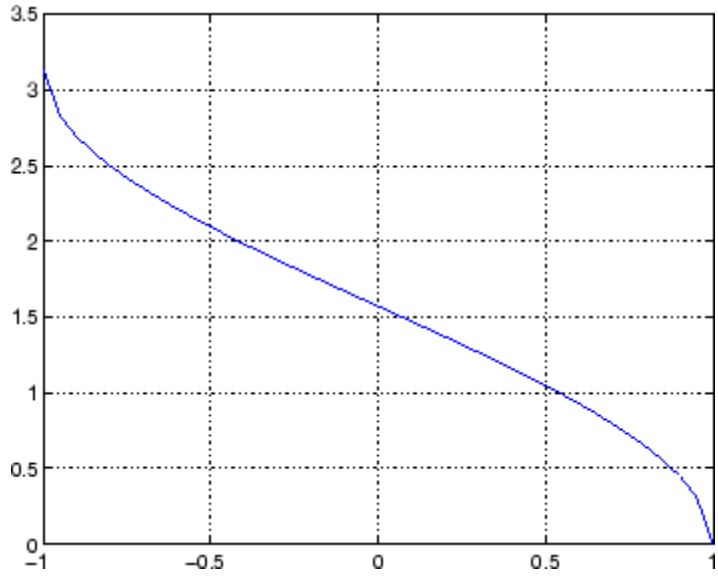
`Y = acos(X)` returns the inverse cosine (arccosine) for each element of `X`. For real elements of `X` in the domain $[-1, 1]$, `acos(X)` is real and in the range $[0, \pi]$. For real elements of `X` outside the domain $[-1, 1]$, `acos(X)` is complex.

The `acos` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse cosine function over the domain $-1 \le x \le 1$.

```
x = -1:.05:1;
plot(x,acos(x)), grid on
```



Definition

The inverse cosine can be defined as

$$\cos^{-1}(z) = -i \log\left[z + i(1 - z^2)^{\frac{1}{2}} \right]$$

Algorithm

`acos` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc., business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`acosd`, `acosh`, `cos`

Purpose

Inverse cosine; result in degrees

Syntax

```
Y = acosd(X)
```

Description

`Y = acosd(X)` is the inverse cosine, expressed in degrees, of the elements of `X`.

See Also

`cosd`, `acos`

Purpose

Inverse hyperbolic cosine

Syntax

```
Y = acosh(X)
```

Description

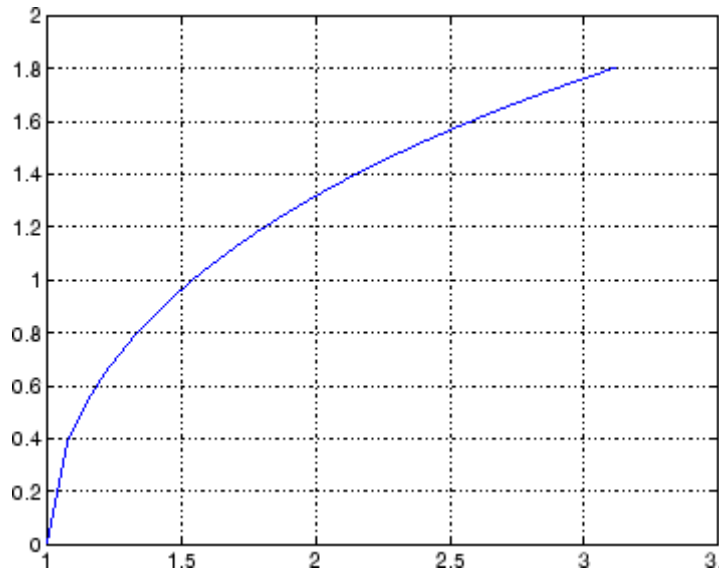
`Y = acosh(X)` returns the inverse hyperbolic cosine for each element of `X`.

The `acosh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse hyperbolic cosine function over the domain $1 \le x \le \pi$.

```
x = 1:pi/40:pi;
plot(x,acosh(x)), grid on
```



Definition

The hyperbolic inverse cosine can be defined as

$$\cosh^{-1}(z) = \log\left[z + (z^2 - 1)^{\frac{1}{2}} \right]$$

Algorithm

`acosh` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc., business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`acos`, `cosh`

Purpose

Inverse cotangent; result in radians

Syntax

```
Y = acot(X)
```

Description

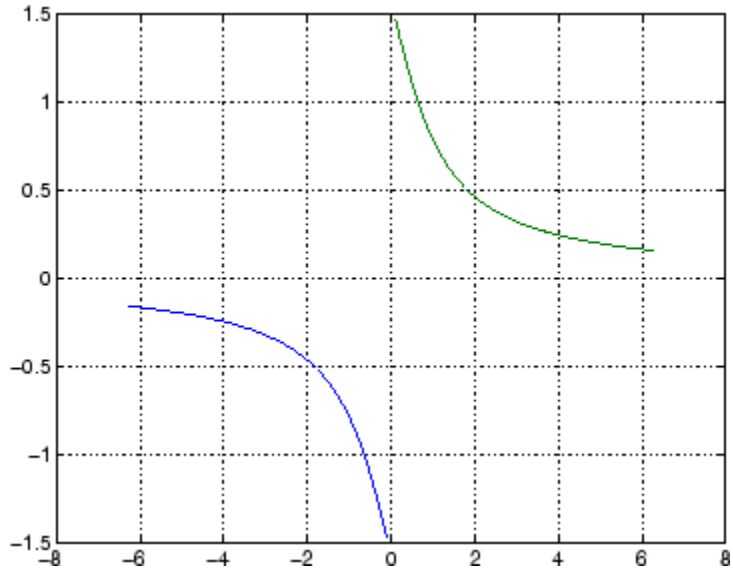
`Y = acot(X)` returns the inverse cotangent (arccotangent) for each element of `X`.

The `acot` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse cotangent over the domains $-2\pi \le x < 0$ and $0 < x \le 2\pi$.

```
x1 = -2*pi:pi/30:-0.1;
x2 = 0.1:pi/30:2*pi;
plot(x1,acot(x1),x2,acot(x2)), grid on
```



Definition

The inverse cotangent can be defined as

$$\cot^{-1}(z) = \tan^{-1}\left(\frac{1}{z}\right)$$

Algorithm

`acot` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc., business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`cot`, `acotd`, `acoth`

acotd

Purpose

Inverse cotangent; result in degrees

Syntax

```
Y = acosd(X)
```

Description

`Y = acosd(X)` is the inverse cotangent, expressed in degrees, of the elements of `X`.

See Also

`cotd`, `acot`

Purpose

Inverse hyperbolic cotangent

Syntax

```
Y = acoth(X)
```

Description

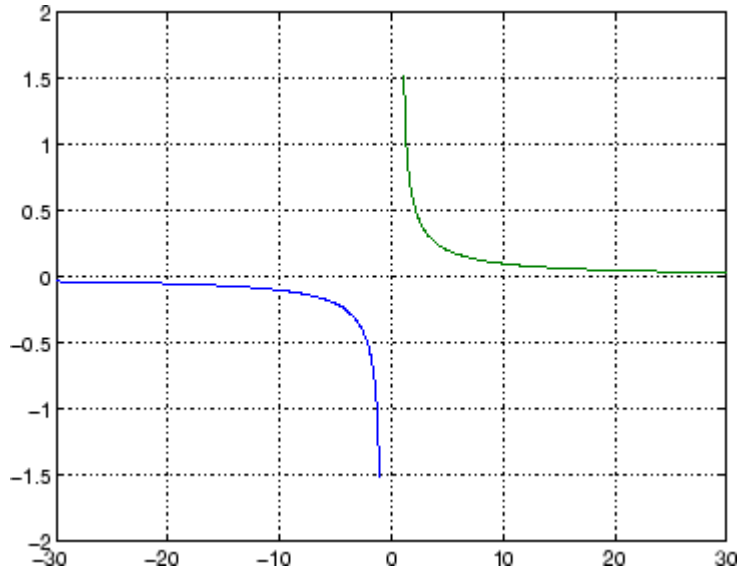
`Y = acoth(X)` returns the inverse hyperbolic cotangent for each element of `X`.

The `acoth` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse hyperbolic cotangent over the domains $-30 \le x < -1$ and $1 < x \le 30$.

```
x1 = -30:0.1:-1.1;
x2 = 1.1:0.1:30;
plot(x1,acoth(x1),x2,acoth(x2)), grid on
```



Definition

The hyperbolic inverse cotangent can be defined as

$$\coth^{-1}(z) = \tanh^{-1}\left(\frac{1}{z}\right)$$

Algorithm

`acoth` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`acot`, `coth`

Purpose

Inverse cosecant; result in radians

Syntax

```
Y = acsc(X)
```

Description

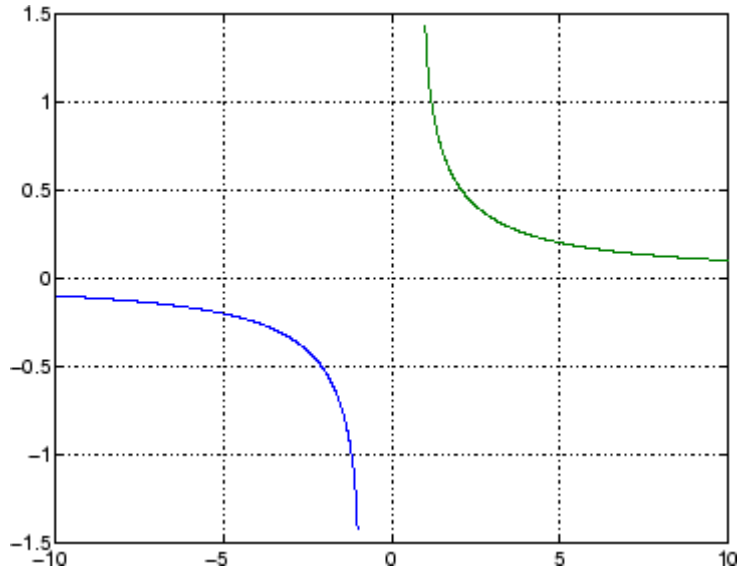
`Y = acsc(X)` returns the inverse cosecant (arccosecant) for each element of `X`.

The `acsc` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse cosecant over the domains $-10 \le x < -1$ and $1 < x \le 10$.

```
x1 = -10:0.01:-1.01;
x2 = 1.01:0.01:10;
plot(x1,acsc(x1),x2,acsc(x2)), grid on
```



Definition

The inverse cosecant can be defined as

$$\csc^{-1}(z) = \sin^{-1}\left(\frac{1}{z}\right)$$

Algorithm

`acsc` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`csc`, `acscd`, `acsch`

Purpose

Inverse cosecant; result in degrees

Syntax

```
Y = acscd(X)
```

Description

`Y = acscd(X)` is the inverse cotangent, expressed in degrees, of the elements of `X`.

See Also

`cscd`, `acsc`

Purpose

Inverse hyperbolic cosecant

Syntax

```
Y = acsch(X)
```

Description

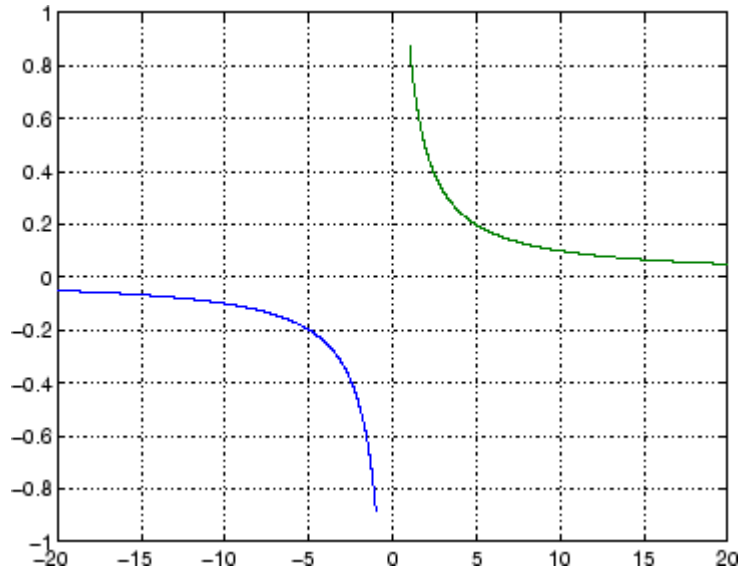
`Y = acsch(X)` returns the inverse hyperbolic cosecant for each element of `X`.

The `acsch` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse hyperbolic cosecant over the domains $-20 \le x \le -1$ and $1 \le x \le 20$.

```
x1 = -20:0.01:-1;
x2 = 1:0.01:20;
plot(x1,acsch(x1),x2,acsch(x2)), grid on
```



Definition

The hyperbolic inverse cosecant can be defined as

$$\operatorname{csch}^{-1}(z) = \operatorname{sinh}^{-1}\left(\frac{1}{z}\right)$$

Algorithm

`acsc` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`acsc`, `csch`

Purpose

Create ActiveX control in figure window

Syntax

```
h = actxcontrol('progid')
h = actxcontrol('progid','param1',value1,...)
h = actxcontrol('progid', position)
h = actxcontrol('progid', position, fig_handle)
h = actxcontrol('progid',position,fig_handle,event_handler)
h = actxcontrol('progid',position,fig_handle,event_handler,
   'filename')
```

Description

`h = actxcontrol('progid')` creates an ActiveX control in a figure window. The type of control created is determined by the string `progid`, the programmatic identifier (`progid`) for the control. (See the documentation provided by the control vendor to get this string.) The returned object, `h`, represents the default interface for the control.

Note that `progid` cannot be an ActiveX server because MATLAB cannot insert ActiveX servers in a figure. See `actxserver` for use with ActiveX servers.

`h = actxcontrol('progid','param1',value1,...)` creates an ActiveX control using the optional parameter name/value pairs. Parameter names include:

- `position` — MATLAB position vector specifying the control's position. The format is [left, bottom, width, height] using pixel units.
- `parent` — Handle to parent figure, model, or command window.
- `callback` — Name of event handler. Specify a single name to use the same handler for all events. Specify a cell array of event name/event handler pairs to handle specific events.
- `filename` — Sets the control's initial conditions to those in the previously saved control.
- `licensekey` — License key to create licensed ActiveX controls that require design-time licenses. See "Deploying ActiveX Controls Requiring Run-Time Licenses" for information on how to use controls that require run-time licenses.

For example:

```
h = actxcontrol('progid','position',[0 0 200 200],...
 'parent',gcf,...
 'callback',{`Click' 'myClickHandler';...
 'DblClick' 'myDblClickHandler';...
 'MouseDown' 'myMouseDownHandler'});
```

The following syntaxes are deprecated and will not become obsolete. They are included for reference, but the above syntaxes are preferred.

`h = actxcontrol('progid', position)` creates an ActiveX control having the location and size specified in the vector, `position`. The format of this vector is

```
[x y width height]
```

The first two elements of the vector determine where the control is placed in the figure window, with `x` and `y` being offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control. The last two elements, `width` and `height`, determine the size of the control itself.

The default `position` vector is `[20 20 60 60]`.

`h = actxcontrol('progid', position, fig_handle)` creates an ActiveX control at the specified `position` in an existing figure window. This window is identified by the Handle Graphics handle, `fig_handle`.

The current figure handle is returned by the `gcf` command.

Note If the figure window designated by `fig_handle` is invisible, the control is invisible. If you want the control you are creating to be invisible, use the handle of an invisible figure window.

`h = actxcontrol('progid',position,fig_handle,event_handler)` creates an ActiveX control that responds to events. Controls respond to events by invoking an M-file function whenever an event (such

as clicking a mouse button) is fired. The `event_handler` argument identifies one or more M-file functions to be used in handling events (see "Specifying Event Handlers" on page 2-86 below).

```
h =
actxcontrol('progid',position,fig_handle,event_handler,'filename')
```

creates an ActiveX control with the first four arguments, and sets its initial state to that of a previously saved control. MATLAB loads the initial state from the file specified in the string `filename`.

If you don't want to specify an `event_handler`, you can use an empty string (`''`) as the fourth argument.

The `progid` argument must match the `progid` of the saved control.

Specifying Event Handlers

There is more than one valid format for the `event_handler` argument. Use this argument to specify one of the following:

- A different event handler routine for each event supported by the control
- One common routine to handle selected events
- One common routine to handle all events

In the first case, use a cell array for the `event_handler` argument, with each row of the array specifying an event and handler pair:

```
{'event' 'eventhandler'; 'event2' 'eventhandler2'; ...}
```

`event` can be either a string containing the event name or a numeric event identifier (see Example 2 below), and `eventhandler` is a string identifying the M-file function you want the control to use in handling the event. Include only those events that you want enabled.

In the second case, use the same cell array syntax just described, but specify the same `eventhandler` for each event. Again, include only those events that you want enabled.

In the third case, make `event_handler` a string (instead of a cell array) that contains the name of the one M-file function that is to handle all events for the control.

There is no limit to the number of event and handler pairs you can specify in the `event_handler` cell array.

Event handler functions should accept a variable number of arguments.

Strings used in the `event_handler` argument are not case sensitive.

Note Although using a single handler for all events may be easier in some cases, specifying an individual handler for each event creates more efficient code that results in better performance.

Remarks

If the control implements any custom interfaces, use the `interfaces` function to list them, and the `invoke` function to get a handle to a selected interface.

When you no longer need the control, call `release` to release the interface and free memory and other resources used by the interface. Note that releasing the interface does not delete the control itself. Use the `delete` function to do this.

For more information on handling control events, see the section, "Writing Event Handlers" in the External Interfaces documentation.

For an example event handler, see the file `sampev.m` in the `toolbox\matlab\winfun\comcli` directory.

Note If you encounter problems creating Microsoft Forms 2.0 controls in MATLAB or other non-VBA container applications, see "Using Microsoft Forms 2.0 Controls" in the External Interfaces documentation.

Examples

Example 1 — Basic Control Methods

Start by creating a `figure` window to contain the control. Then create a control to run a Microsoft Calendar application in the window. Position the control at a `[0 0]` x-y offset from the bottom left of the figure window, and make it the same size (600 x 500 pixels) as the figure window.

```
f = figure('position', [300 300 600 500]);
cal = actxcontrol('mscal.calendar', [0 0 600 500], f)
cal =
    COM.mscal.calendar
```

Call the `get` method on `cal` to list all properties of the calendar:

```
cal.get
        BackColor: 2.1475e+009
              Day: 23
          DayFont: [1x1 Interface.Standard_OLE_Types.Font]
            Value: '8/20/2001'
                  .
                  .
```

Read just one property to record today's date:

```
date = cal.Value
date =
   8/23/2001
```

Set the `Day` property to a new value:

```
cal.Day = 5;
date = cal.Value
date =
   8/5/2001
```

Call `invoke` with no arguments to list all available methods:

```
meth = cal.invoke
meth =
```

```
          NextDay: 'HRESULT NextDay(handle)'
        NextMonth: 'HRESULT NextMonth(handle)'
         NextWeek: 'HRESULT NextWeek(handle)'
         NextYear: 'HRESULT NextYear(handle)'
            .
            .
```

Invoke the `NextWeek` method to advance the current date by one week:

```
cal.NextWeek;
date = cal.Value
date =
   8/12/2001
```

Call `events` to list all calendar events that can be triggered:

```
cal.events
ans =
   Click = void Click()
   DblClick = void DblClick()
   KeyDown = void KeyDown(int16 KeyCode, int16 Shift)
   KeyPress = void KeyPress(int16 KeyAscii)
   KeyUp = void KeyUp(int16 KeyCode, int16 Shift)
   BeforeUpdate = void BeforeUpdate(int16 Cancel)
   AfterUpdate = void AfterUpdate()
   NewMonth = void NewMonth()
   NewYear = void NewYear()
```

Example 2 — Event Handling

The `event_handler` argument specifies how you want the control to handle any events that occur. The control can handle all events with one common handler function, selected events with a common handler function, or each type of event can be handled by a separate function.

This command creates an `mwsamp` control that uses one event handler, `sampev`, to respond to all events:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
```

```
    gcf, 'sampev')
```

The next command also uses a common event handler, but will only invoke the handler when selected events, `Click` and `DblClick` are fired:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
   gcf, {'Click' 'sampev'; 'DblClick' 'sampev'})
```

This command assigns a different handler routine to each event. For example, `Click` is an event, and `myclick` is the routine that executes whenever a `Click` event is fired:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
   gcf, {'Click', 'myclick'; 'DblClick' 'my2click'; ...
   'MouseDown' 'mymoused'});
```

The next command does the same thing, but specifies the events using numeric event identifiers:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
gcf, {-600, 'myclick'; -601 'my2click'; -605 'mymoused'});
```

See the section, “Sample Event Handlers” in the External Interfaces documentation for examples of event handler functions and how to register them with MATLAB.

See Also

`actxserver`, `release`, `delete`, `save`, `load`, `interfaces`

Purpose

List all currently installed ActiveX controls

Syntax

```
C = actxcontrollist
```

Description

`C = actxcontrollist` returns a list of each control, including its name, programmatic identifier (or ProgID), and filename, in output cell array `C`.

Examples

Here is an example of the information that might be returned for several controls:

```
list = actxcontrollist;

for k = 1:2
   sprintf(' Name = %s\n ProgID = %s\n File = %s\n', ...
           list{k,:})
end

ans =
   Name = ActiveXPlugin Object
   ProgID = Microsoft.ActiveXPlugin.1
   File = C:\WINNT\System32\plugin.ocx

ans =
   Name = Adaptec CD Guide
   ProgID = Adaptec.EasyCDGuide
   File = D:\APPLIC~1\Adaptec\Shared\CDGuide\CDGuide.ocx
```

See Also

`actxcontrolselect`, `actxcontrol`

actxcontrolselect

Purpose

Open GUI to create ActiveX control

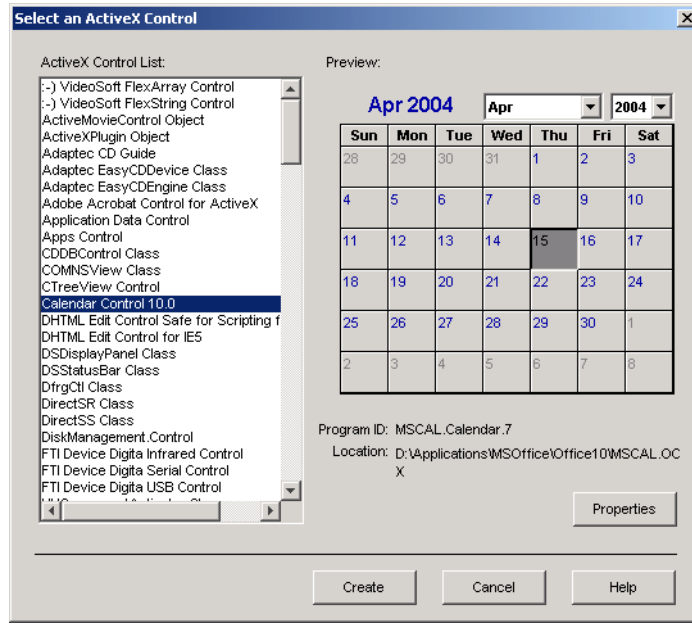
Syntax

```
h = actxcontrolselect
[h, info] = actxcontrolselect
```

Description

`h = actxcontrolselect` displays a graphical interface that lists all ActiveX controls installed on the system and creates the one that you select from the list. The function returns a handle `h` for the object. Use the handle to identify this particular control object when calling other MATLAB COM functions.

`[h, info] = actxcontrolselect` returns the handle `h` and also the 1-by-3 cell array `info` containing information about the control. The information returned in the cell array shows the name, programmatic identifier (or ProgID), and filename for the control.

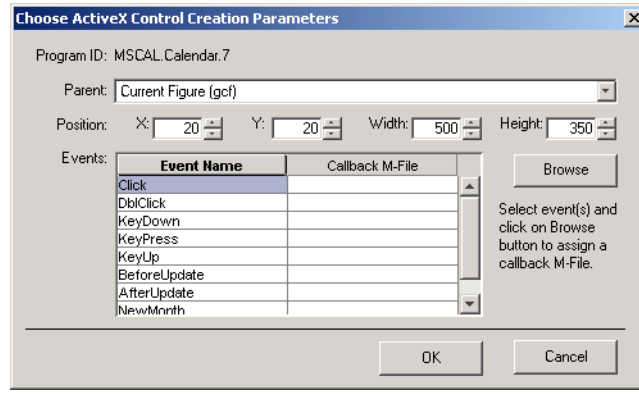


The `actxcontrolselect` interface has a selection panel at the left of the window and a preview panel at the right. Click on one of the control names in the selection panel to see a preview of the control displayed. (If MATLAB cannot create the control, an error message is displayed in the preview panel.) Select an item from the list and click the **Create** button at the bottom.

Remarks

Click the **Properties** button on the `actxcontrolselect` window to enter nondefault values for properties when creating the control. You can select which figure window to put the control in (**Parent** field), where to position it in the window (**X** and **Y** fields), and what size to make the control (**Width** and **Height**).

You can also register any events you want the control to respond to and what event handling routines to use when any of these events fire. Do this by entering the name of the appropriate event handling routine to the right of the event, or clicking the **Browse** button to search for the event handler file.



Note If you encounter problems creating Microsoft Forms 2.0 controls in MATLAB or other non-VBA container applications, see “Using Microsoft Forms 2.0 Controls” in the External Interfaces documentation.

Examples

Select `Calendar Control 9.0` in the `actxcontrolselect` window and then click **Properties** to open the window shown above. Enter new values for the size of the control, setting **Width** to `500` and **Height** to `350`, then click **OK**. Click **Create** in the `actxcontrolselect` window to create the control.

The control appears in a MATLAB figure window and the `actxcontrolselect` function returns these values:

```
h =
   COM.mscal.calendar.7
info =
    [1x20 char]    'MSCAL.Calendar.7'    [1x41 char]
```

Expand the `info` cell array to show the control name, ProgID, and filename:

```
info{:}
ans =
   Calendar Control 9.0
ans =
   MSCAL.Calendar.7
ans =
   D:\Applications\MSOffice\Office\MSCAL.OCX
```

See Also

`actxcontrollist`, `actxcontrol`

Purpose

Get handle to running instance of Automation server

Syntax

```
h = actxGetRunningServer('progid')
```

Description

`h = actxGetRunningServer('progid')` gets a reference to a running instance of the OLE Automation server, where `progid` is the programmatic identifier of the Automation server object and `h` is the handle to the server object's default interface.

The function issues an error if the server specified by `progid` is not currently running or if the server object is not registered. When there are multiple instances of the Automation server already running, the behavior of this function is controlled by the operating system.

Example

```
h = actxGetRunningServer('Excel.Application')
```

See Also

`actxcontrol`, `actxserver`

Purpose

Create COM server

Syntax

```
h = actxserver('progid')
h = actxserver('progid', 'machine', 'machineName')
h = actxserver('progid', 'interface', 'interfaceName')
h = actxserver('progid', 'machine', 'machineName',
   'interface', 'interfaceName')
h = actxserver('progid', machine)
```

Description

`h = actxserver('progid')` creates a local OLE Automation server, where `progid` is the programmatic identifier of the COM server, and `h` is the handle of the server's default interface.

Get `progid` from the control or server vendor's documentation. To see the `progid` values for MATLAB, refer to "Programmatic Identifiers" in the MATLAB External Interfaces documentation.

`h = actxserver('progid', 'machine', 'machineName')` creates an OLE Automation server on a remote machine, where `machineName` is a string specifying the name of the machine on which to launch the server.

`h = actxserver('progid', 'interface', 'interfaceName')` creates a Custom interface server, where `interfaceName` is a string specifying the interface name of the COM object. Values for `interfaceName` are

- `IUnknown` — Use the IUnknown interface.
- The Custom interface name

You must know the name of the interface and have the server vendor's documentation in order to use the `interfaceName` value. See "COM Server Types" in the MATLAB External Interfaces documentation for information about Custom COM servers and interfaces.

`h = actxserver('progid', 'machine', 'machineName', 'interface', 'interfaceName')` creates a Custom interface server on a remote machine.

The following syntaxes are deprecated and will not become obsolete. They are included for reference, but the syntaxes described earlier are preferred:

`h = actxserver('progid', machine)` creates a COM server running on the remote system named by the `machine` argument. This can be an IP address or a DNS name. Use this syntax only in environments that support Distributed Component Object Model (DCOM).

Remarks

For components implemented in a dynamic link library (DLL), `actxserver` creates an in-process server. For components implemented as an executable (EXE), `actxserver` creates an out-of-process server. Out-of-process servers can be created either on the client system or on any other system on a network that supports DCOM.

If the control implements any Custom interfaces, use the `interfaces` function to list them, and the `invoke` function to get a handle to a selected interface.

You can register events for COM servers.

Run Microsoft Excel Example

This example creates an OLE Automation server, Microsoft Excel version 9.0, and manipulates a workbook in the application:

```
%  Create a COM server running Microsoft Excel
e = actxserver ('Excel.Application')

% e =
%     COM.excel.application

% Make the Excel frame window visible
e.Visible = 1;

% Use the get method on the Excel object "e"
% to list all properties of the application:
e.get

% ans =
```

```
%        Application: [1x1Interface.Microsoft_Excel_9.0_
%Object_Library._Application]
%            Creator: 'xlCreatorCode'
%          Workbooks: [1x1 Interface.Microsoft_Excel_9.0_
%Object_Library.Workbooks]
%            Caption: 'Microsoft Excel - Book1'
%    CellDragAndDrop: 0
%   ClipboardFormats: {3x1 cell}
%             Cursor: 'xlNorthwestArrow'
%               .
%               .

% Create an interface "eWorkBooks"
eWorkbooks = e.Workbooks

% eWorkbooks =
%    Interface.Microsoft_Excel_9.0_Object_Library.Workbooks

% List all methods for that interface
eWorkbooks.invoke

% ans =
%       Add: 'handle Add(handle, [Optional]Variant)'
%     Close: 'void Close(handle)'
%      Item: 'handle Item(handle, Variant)'
%      Open: 'handle Open(handle, string, [Optional]Variant)'
%  OpenText: 'void OpenText(handle, string, [Optional]Variant)'

% Add a new workbook "w",
% also creating a new interface
w = eWorkbooks.Add

% w =
%    Interface.Microsoft_Excel_9.0_Object_Library._Workbook

% Close Excel and delete the object
e.Quit;
```

```
e.delete;
```

See Also

actxcontrol, release, delete, save, load, interfaces

addCause (MException)

Purpose

Append MException objects

Syntax

```
new_ME = addCause(base_ME, cause_ME)
base_ME = addCause(base_ME, cause_ME)
```

Description

`new_ME = addCause(base_ME, cause_ME)` creates a new `MException` object `new_ME` from two existing `MException` objects, `base_ME` and `cause_ME`. `addCause` constructs `new_ME` by making a copy of the `base_ME` object and appending `cause_ME` to the `cause` property of that object.

If other errors have contributed to the exception currently being thrown, you can add the `MException` objects that represent these errors to the `cause` field of the current `MException` to provide further information for diagnosing the error at hand. All objects of the `MException` class have a property called `cause` which is defined as a vector of additional `MException` objects that can be added onto a base object of that class.

`base_ME = addCause(base_ME, cause_ME)` modifies existing `MException` object `base_ME` by appending `cause_ME` to the `cause` property of that object.

Examples

Example 1

This example attempts to assign data from array `D`. If `D` does not exist, the code attempts to recreate `D` by loading it from a MAT-file. The code constructs a new `MException` object `new_ME` to store the causes of the first two errors, `cause1_ME` and `cause2_ME`:

```
try
    x = D(1:25);
catch cause1_ME
    try
        filename = 'test2O4';
        testdata = load(filename);
        x = testdata.D(1:25)
    catch cause2_ME
        base_ME = MException('MATLAB:LoadErr', ...
               'Unable to load from file %s', filename);
```

```
        new_ME = addCause(base_ME, cause1_ME);
        new_ME = addCause(new_ME, cause2_ME);
        throw(new_ME);
    end
end
```

When you run the code, MATLAB displays the following message:

```
??? Unable to load from file test204
```

There are two exceptions in the cause field of `new_ME`:

```
new_ME.cause
ans =
    [1x1 MException]
    [1x1 MException]
```

Examine the `cause` field of `new_ME` to see the related errors:

```
new_ME.cause{:}
ans =

 MException object with properties:

    identifier: 'MATLAB:UndefinedFunction'
       message: 'Undefined function or method 'D' for input
            arguments of type 'double'.'
         stack: [0x1 struct]
         cause: {}
ans =

 MException object with properties:

    identifier: 'MATLAB:load:couldNotReadFile'
       message: 'Unable to read file test204: No such file
            or directory.'
         stack: [0x1 struct]
         cause: {}
```

Example 2

This example attempts to open a file in a directory that is not on the MATLAB path. It uses a nested `try-catch` block to give the user the opportunity to extend the path. If the still cannot be found, the program issues an exception with the first error appended to the second using `addCause`:

```
function data = read_it(filename);
try
   fid = fopen(filename, 'r');
   data = fread(fid);
catch ME1
   if strcmp(ME1.identifier, 'MATLAB:FileIO:InvalidFid')
      msg = sprintf('\n%s%s%s', 'Cannot open file ', ...
         filename, '. Try another location?  ');
      reply = input(msg, 's')
      if reply(1) == 'y'
          newdir = input('Enter directory name:  ', 's');
      else
          throw(ME1);
      end
      addpath(newdir);
      try
         fid = fopen(filename, 'r');
         data = fread(fid);
      catch ME2
         ME3 = addCause(ME2, ME1)
         throw(ME3);
      end
      rmpath(newdir);
   end
end
fclose(fid);
```

If you run this function in a `try-catch` block at the command line, you can look at the `MException` object by assigning it to a variable (`e`) with the `catch` command.

```
try
   d = read_it('anytextfile.txt');
catch e
end

e
e =
 MException object with properties:

    identifier: 'MATLAB:FileIO:InvalidFid'
       message: 'Invalid file identifier.  Use fopen to
generate a valid file identifier.'
         stack: [1x1 struct]
         cause: {[1x1 MException]}

  Cannot open file anytextfile.txt. Try another location?y
Enter directory name:  xxxxxxx
Warning: Name is nonexistent or not a directory: xxxxxxx.
> In path at 110
  In addpath at 89
```

See Also

try, catch, error, assert, , MException, throw(MException), rethrow(MException), throwAsCaller(MException), getReport(MException), disp(MException), isequal(MException), eq(MException), ne(MException), last(MException)

addevent

Purpose

Add event to timeseries object

Syntax

```
ts = addevent(ts,e)
ts = addevent(ts,Name,Time)
```

Description

`ts = addevent(ts,e)` adds one or more `tsdata.event` objects, `e`, to the `timeseries` object `ts`. `e` is either a single `tsdata.event` object or an array of `tsdata.event` objects.

`ts = addevent(ts,Name,Time)` constructs one or more `tsdata.event` objects and adds them to the `Events` property of `ts`. `Name` is a cell array of event name strings. `Time` is a cell array of event times.

Examples

Create a time-series object and add an event to this object.

```
%% Import the sample data
load count.dat

%% Create time-series object
count1=timeseries(count(:,1),1:24,'name', 'data');

%% Modify the time units to be 'hours' ('seconds' is default)
count1.TimeInfo.Units = 'hours';

%% Construct and add the first event at 8 AM
e1 = tsdata.event('AMCommute',8);

%% Specify the time units of the time
e1.Units = 'hours';
```

View the properties (`EventData`, `Name`, `Time`, `Units`, and `StartDate`) of the event object.

```
get(e1)
```

MATLAB responds with

```
    EventData: []
```

```
           Name: 'AMCommute'
           Time: 8
          Units: 'hours'
      StartDate: ''
  %% Add the event to count1
  count1 = addevent(count1,e1);
```

An alternative syntax for adding two events to the time series `count1` is as follows:

```
count1 = addevent(count1,{'AMCommute' 'PMCommute'},{8 18})
```

See Also

`timeseries`, `tsdata.event`, `tsprops`

addframe

Purpose

Add frame to Audio/Video Interleaved (AVI) file

Syntax

```
aviobj = addframe(aviobj,frame)
aviobj = addframe(aviobj,frame1,frame2,frame3,...)
aviobj = addframe(aviobj,mov)
aviobj = addframe(aviobj,h)
```

Description

`aviobj = addframe(aviobj,frame)` appends the data in `frame` to the AVI file identified by `aviobj`, which was created by a previous call to `avifile`. `frame` can be either an indexed image (`m`-by-`n`) or a truecolor image (`m`-by-`n`-by-3) of `double` or `uint8` precision. If `frame` is not the first frame added to the AVI file, it must be consistent with the dimensions of the previous frames.

`addframe` returns a handle to the updated AVI file object, `aviobj`. For example, `addframe` updates the `TotalFrames` property of the AVI file object each time it adds a frame to the AVI file.

`aviobj = addframe(aviobj,frame1,frame2,frame3,...)` adds multiple frames to an AVI file.

`aviobj = addframe(aviobj,mov)` appends the frames contained in the MATLAB movie `mov` to the AVI file `aviobj`. MATLAB movies that store frames as indexed images use the colormap in the first frame as the colormap for the AVI file, unless the colormap has been previously set.

`aviobj = addframe(aviobj,h)` captures a frame from the figure or axis handle `h` and appends this frame to the AVI file. `addframe` renders the figure into an offscreen array before appending it to the AVI file. This ensures that the figure is written correctly to the AVI file even if the figure is obscured on the screen by another window or screen saver.

Note If an animation uses XOR graphics, you must use `getframe` to capture the graphics into a frame of a MATLAB movie. You can then add the frame to an AVI movie using the `addframe` syntax `aviobj = addframe(aviobj,mov)`. See the example for an illustration.

Example

This example calls `addframe` to add frames to the AVI file object `aviobj`.

```
fig=figure;
set(fig,'DoubleBuffer','on');
set(gca,'xlim',[-80 80],'ylim',[-80 80],...
 'nextplot','replace','Visible','off')

aviobj = avifile('example.avi')

x = -pi:.1:pi;
radius = 0:length(x);
for i=1:length(x)
 h = patch(sin(x)*radius(i),cos(x)*radius(i),...
   [abs(cos(x(i))) 0 0]);
 set(h,'EraseMode','xor');
 frame = getframe(gca);
 aviobj = addframe(aviobj,frame);
end

aviobj = close(aviobj);
```

See Also

`avifile`, `close`, `movie2avi`

addOptional (inputParser)

Purpose

Add optional argument to `inputParser` schema

Syntax

```
p.addOptional(argname, default, validator)
addOptional(p, argname, default, validator)
```

Description

`p.addOptional(argname, default, validator)` updates the schema for `inputParser` object `p` by adding an optional argument, `argname`. Specify the argument name in a string enclosed within single quotation marks. The `default` input specifies the value to use when the optional argument `argname` is not present in the actual inputs to the function. The optional `validator` input is a handle to a function that MATLAB uses during parsing to validate the input arguments. If the validator function returns `false` or errors, the parsing fails and MATLAB throws an error.

MATLAB parses parameter-value arguments after required arguments and optional arguments.

`addOptional(p, argname, default, validator)` is functionally the same as the syntax above.

Note For more information on the `inputParser` class, see Parsing Inputs with inputParser in the MATLAB Programming documentation.

Examples

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class.

There are three calling syntaxes for this function:

```
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

From these three syntaxes, you can see that there is one required argument (`script`), one optional argument (`format`), and some number

of optional arguments that are specified as parameter-value pairs (`options`).

Begin writing the example `publish_ip` M-file by entering the following two statements. The second statement calls the class constructor for `inputParser` to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.
```

Following the constructor, add this block of code to the M-file. This code uses the `addRequired(inputParser)`, `addOptional`, and `addParamValue(inputParser)` methods to define the input arguments to the function:

```
p.addRequired('script', @ischar);
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Also add the next two lines to the M-file. The `Parameters` property of `inputParser` lists all of the arguments that belong to the object `p`:

```
disp 'The input parameters for this program are
disp(p.Parameters)'
```

Save the M-file using the **Save** option on the MATLAB **File** menu, and then run it to see the following list displayed:

```
The input parameters for this program are
    'format'
    'maxHeight'
    'maxWidth'
    'outputDir'
    'script'
```

See Also

`inputParser`, `addRequired(inputParser)`, `addParamValue(inputParser)`, `parse(inputParser)`, `createCopy(inputParser)`

Purpose

Add parameter-value argument to `inputParser` schema

Syntax

```
p.addParamValue(argname, default, validator)
addParamValue(p, argname, default, validator)
```

Description

`p.addParamValue(argname, default, validator)` updates the schema for `inputParser` object `p` by adding a parameter-value argument, `argname`. Specify the argument name in a string enclosed within single quotation marks. The `default` input specifies the value to use when the optional argument name is not present in the actual inputs to the function. The optional `validator` is a handle to a function that MATLAB uses during parsing to validate the input arguments. If the validator function returns `false` or errors, the parsing fails and MATLAB throws an error.

MATLAB parses parameter-value arguments after required arguments and optional arguments.

`addParamValue(p, argname, default, validator)` is functionally the same as the syntax above.

Note For more information on the `inputParser` class, see Parsing Inputs with inputParser in the MATLAB Programming documentation.

Examples

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class. There are three calling syntaxes for this function:

```
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

From these calling syntaxes, you can see that there is one required argument (`script`), one optional argument (`format`), and a number of optional arguments that are specified as parameter-value pairs (`options`).

Begin writing the example `publish_ip` M-file by entering the following two statements. Call the class constructor for `inputParser` to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.
```

After calling the constructor, add the following lines to the M-file. This code uses the `addRequired(inputParser)`, `addOptional(inputParser)`, and `addParamValue` methods to define the input arguments to the function:

```
p.addRequired('script', @ischar);
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Also add the next two lines to the M-file. The `Parameters` property of `inputParser` lists all of the arguments that belong to the object `p`:

```
disp 'The input parameters for this program are
disp(p.Parameters)'
```

Save the M-file using the **Save** option on the MATLAB **File** menu, and then run it to see the following list displayed:

```
The input parameters for this program are
    'format'
    'maxHeight'
    'maxWidth'
    'outputDir'
    'script'
```

See Also

`inputParser`, `addRequired(inputParser)`, `addOptional(inputParser)`, `parse(inputParser)`, `createCopy(inputParser)`

addpath

Purpose

Add directories to MATLAB search path

GUI Alternatives

As an alternative to the `addpath` function, use **File > Set Path** to open the Set Path dialog box.

Syntax

```
addpath('directory')
addpath('dir','dir2','dir3' ...)
addpath('dir','dir2','dir3' ...'-flag')
addpath dir1 dir2 dir3 ... -flag
```

Description

`addpath('directory')` adds the specified directory to the top (also called front) of the current MATLAB search path. Use the full pathname for `directory`.

`addpath('dir','dir2','dir3' ...)` adds all the specified directories to the top of the path. Use the full pathname for each `dir`.

`addpath('dir','dir2','dir3' ...'-flag')` adds the specified directories to either the top or bottom of the path, depending on the value of `flag`.

flag Argument	Result
`0` or `begin`	Add specified directories to the top of the path
`1` or `end`	Add specified directories to the bottom (also called end) of the path

`addpath dir1 dir2 dir3 ... -flag` is the unquoted form of the syntax.

Remarks

To recursively add subdirectories of your directory in addition to the directory itself, run

```
addpath(genpath('directory'))
```

Use addpath statements in your startup.m file to use the modified path in future sessions. For details, see "Modifying the Path in a startup.m File" in the MATLAB Desktop Tools and Development Environment Documentation.

Examples

For the current path, viewed by typing path,

```
MATLABPATH
c:\matlab\toolbox\general
c:\matlab\toolbox\ops
c:\matlab\toolbox\strfun
```

you can add c:/matlab/mymfiles to the front of the path by typing

```
addpath('c:/matlab/mymfiles')
```

Verify that the files were added to the path by typing

```
path
```

and MATLAB returns

```
MATLABPATH
c:\matlab\mymfiles
c:\matlab\toolbox\general
c:\matlab\toolbox\ops
c:\matlab\toolbox\strfun
```

You can also use genpath in conjunction with addpath to add subdirectories to the path from the command line. For example, to add /control and its subdirectories to the path, use

```
addpath(genpath(fullfile(matlabroot,'toolbox/control')))
```

See Also

genpath, path, pathdef, pathsep, pathtool, rehash, restoredefaultpath, rmpath, savepath, startup

"Search Path" in the MATLAB Desktop Tools and Development Environment Documentation

addpref

Purpose

Add preference

Syntax

```
addpref('group','pref',val)
addpref('group',{'pref1','pref2',...'prefn'},{val1,val2,
   ...valn})
```

Description

`addpref('group','pref',val)` creates the preference specified by `group` and `pref` and sets its value to `val`. It is an error to add a preference that already exists.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g. `'ApplicationOnePrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`addpref('group',{'pref1','pref2',...'prefn'},{val1,val2,...valn})` creates the preferences specified by the cell array of names `'pref1'`, `'pref2'`,`...`,`'prefn'`, setting each to the corresponding value.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

This example adds a preference called `version` to the `mytoolbox` group of preferences and sets its value to the string `1.0`.

```
addpref('mytoolbox','version','1.0')
```

See Also

`getpref`, `ispref`, `rmpref`, `setpref`, `uigetpref`, `uisetpref`

Purpose

Add custom property to object

Syntax

```
h.addproperty('propertyname')
addproperty(h, 'propertyname')
```

Description

`h.addproperty('propertyname')` adds the custom property specified in the string, `propertyname`, to the object or interface, `h`. Use `set` to assign a value to the property.

`addproperty(h, 'propertyname')` is an alternate syntax for the same operation.

Examples

Create an `mwsamp` control and add a new property named `Position` to it. Assign an array value to the property:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.get
     Label: 'Label'
    Radius: 20

h.addproperty('Position');
h.Position = [200 120];
h.get
     Label: 'Label'
    Radius: 20
  Position: [200 120]

h.get('Position')
ans =
   200   120
```

Delete the custom `Position` property:

```
h.deleteproperty('Position');
h.get
     Label: 'Label'
    Radius: 20
```

See Also deleteproperty, get, set, inspect

Purpose

Add required argument to `inputParser` schema

Syntax

```
p.addRequired(argname, validator)
addRequired(p, argname, validator)
```

Description

`p.addRequired(argname, validator)` updates the schema for `inputParser` object `p` by adding a required argument, `argname`. Specify the argument name in a string enclosed within single quotation marks. The optional `validator` is a handle to a function that MATLAB uses during parsing to validate the input arguments. If the validator function returns `false` or errors, the parsing fails and MATLAB throws an error.

MATLAB parses required arguments before optional or parameter-value arguments.

`addRequired(p, argname, validator)` is functionally the same as the syntax above.

Note For more information on the `inputParser` class, see Parsing Inputs with inputParser in the MATLAB Programming documentation.

Examples

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class. There are three calling syntaxes for this function:

```
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

From these calling syntaxes, you can see that there is one required argument (`script`), one optional argument (`format`), and a number of optional arguments that are specified as parameter-value pairs (`options`).

Begin writing the example `publish_ip` M-file by entering the following two statements. Call the class constructor for `inputParser` to create an

instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.
```

After calling the constructor, add the following lines to the M-file. This code uses the `addRequired`, `addOptional(inputParser)`, and `addParamValue(inputParser)` methods to define the input arguments to the function:

```
p.addRequired('script', @ischar);
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Also add the next two lines to the M-file. The `Parameters` property of `inputParser` lists all of the arguments that belong to the object `p`:

```
disp 'The input parameters for this program are
disp(p.Parameters)'
```

Save the M-file using the **Save** option on the MATLAB **File** menu, and then run it to see the following list displayed:

```
The input parameters for this program are
    'format'
    'maxHeight'
    'maxWidth'
    'outputDir'
    'script'
```

See Also

`inputParser`, `addOptional(inputParser)`, `addParamValue(inputParser)`, `parse(inputParser)`, `createCopy(inputParser)`

Purpose

Add data sample to `timeseries` object

Syntax

```
ts = addsample(ts,'Field1',Value1,'Field2',Value2,...)
ts = addsample(ts,s)
```

Description

`ts = addsample(ts,'Field1',Value1,'Field2',Value2,...)` adds one or more data samples to the `timeseries` object `ts`, where one field must specify `Time` and another must specify `Data`. You can also specify the following optional property-value pairs:

- `'Quality'` — Array of data quality codes
- `'OverwriteFlag'` — Logical value that controls whether to overwrite a data sample at the same time with the new sample you are adding to your `timeseries` object. When set to `true`, the new sample overwrites the old sample at the same time.

`ts = addsample(ts,s)` adds one or more new samples stored in a structure `s` to the `timeseries` object `ts`. You must define the fields of the structure `s` before passing it as an argument to `addsample` by assigning values to the following optional `s` fields:

- `s.data`
- `s.time`
- `s.quality`
- `s.overwriteflag`

Remarks

A time-series *data sample* consists of one or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

The `Time` value must be a valid time vector.

Suppose that `N` is the number of samples. The sample size of each time series is given by `SampleSize = getsamplesize(ts)`. When

`ts.IsTimeFirst` is `true`, the size of the data is `N`-by-`SampleSize`. When `ts.IsTimeFirst` is `false`, the size of the data is `SampleSize`-by-`N`.

Examples

Add a data value of `420` at time `3`.

```
ts = ts.addsample('Time',3,'Data',420);
```

Add a data value of `420` at time `3` and specify quality code `1` for this data value. Set the flag to overwrite an existing value at time `3`.

```
ts = ts.addsample('Data',3.2,'Quality',1,'OverwriteFlag',...
        true,'Time',3);
```

See Also

`delsample`, `getdatasamplesize`, `tsprops`

Purpose

Add sample to tscollection object

Syntax

```
tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data,
   TSnName,TSnData)
```

Description

`tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data, TSnName,TSnData)` adds data samples `TSnData` to the collection member `TSnName` in the `tscollection` object `tsc` at one or more `Time` values. Here, `TSnName` is the string that represents the name of a time series in `tsc`, and `TSnData` is an array containing data samples.

Remarks

If you do not specify data samples for a time-series member in tsc, that time-series member will contain missing data at the times given by Time (for numerical time-series data), `NaN` values, or (for logical time-series data) `false` values.

When a time-series member requires `Quality` values, you can specify data quality codes together with the data samples by using the following syntax:

```
tsc = addsampletocollection(tsc,'time',time,TS1Name,...
          ts1cellarray,TS2Name,ts2cellarray,...)
```

Specify data in the first cell array element and `Quality` in the second cell array element.

Note If a time-series member already has `Quality` values but you only provide data samples, `0`s are added to the existing `Quality` array at the times given by `Time`.

Examples

The following example shows how to create a `tscollection` that consists of two `timeseries` objects, where one `timeseries` does not have quality codes and the other does. The final step of the example adds a sample to the `tscollection`.

1 Create two `timeseries` objects, `ts1` and `ts2`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                 'name','acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
                 'name','speed');
```

2 Define a dictionary of quality codes and descriptions for `ts2`.

```
ts2.QualityInfo.Code = [0 1];
ts2.QualityInfo.Description = {'bad','good'};
```

3 Assign a quality of code of `1`, which is equivalent to `'good'`, to each data value in `ts2`.

```
ts2.Quality = ones(5,1);
```

4 Create a time-series collection `tsc`, which includes time series `ts1` and `ts2`.

```
tsc = tscollection({ts1,ts2});
```

5 Add a data sample to the collection `tsc` at `3.5` seconds.

```
tsc = addsampletocollection(tsc,'time',3.5,'acceleration',10,
        'speed',{5 1});
```

The cell array for the `timeseries` object `'speed'` specifies both the data value 5 and the quality code `1`.

Note If you do not specify a quality code when adding a data sample to a time series that has quality codes, then the lowest quality code is assigned to the new sample by default.

See Also

`delsamplefromcollection`, `tscollection`, `tsprops`

Purpose

Modify date number by field

Syntax

```
R = addtodate(D, Q, F)
```

Description

`R = addtodate(D, Q, F)` adds quantity `Q` to the indicated date field `F` of a scalar serial date number `D`, returning the updated date number `R`.

The quantity `Q` to be added must be a double scalar whole number, and can be either positive or negative. The date field `F` must be a 1-by-`N` character array equal to one of the following: `'year'`, `'month'`, or `'day'`.

If the addition to the date field causes the field to roll over, MATLAB adjusts the next more significant fields accordingly. Adding a negative quantity to the indicated date field rolls back the calender on the indicated field. If the addition causes the field to roll back, MATLAB adjusts the next less significant fields accordingly.

Examples

Adding 20 days to the given date in late December causes the calendar to roll over to January of the next year:

```
R = addtodate(datenum('12/24/1984 12:45'), 20, 'day');

datestr(R)
ans =
   13-Jan-1985 12:45:00
```

See Also

`date`, `datenum`, `datestr`, `datevec`

addts

Purpose

Add timeseries object to tscollection object

Syntax

```
tsc = addts(tsc,ts)
tsc = addts(tsc,ts)
tsc = addts(tsc,ts,Name)
tsc = addts(tsc,Data,Name)
```

Description

`tsc = addts(tsc,ts)` adds the `timeseries` object ts to `tscollection` object tsc.

`tsc = addts(tsc,ts)` adds a cell array of `timeseries` objects ts to the `tscollection` tsc.

`tsc = addts(tsc,ts,Name)` adds a cell array of `timeseries` objects ts to `tscollection` tsc. Name is a cell array of strings that gives the names of the `timeseries` objects in `ts`.

`tsc = addts(tsc,Data,Name)` creates a new `timeseries` object from Data with the name Name and adds it to the `tscollection` object tsc. Data is a numerical array and Name is a string.

Remarks

The `timeseries` objects you add to the collection must have the same time vector as the collection. That is, the time vectors must have the same time values and units.

Suppose that the time vector of a `timeseries` object is associated with calendar dates. When you add this `timeseries` to a collection with a time vector without calendar dates, the time vectors are compared based on the units and the values relative to the `StartDate` property. For more information about properties, see the `timeseries` reference page.

Examples

The following example shows how to add a time series to a time-series collection:

1. Create two `timeseries` objects, `ts1` and `ts2`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                 'name','acceleration');
```

```
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
                 'name','speed');
```

2 Create a time-series collection `tsc`, which includes `ts1`.

```
tsc = tscollection(ts1);
```

3 Add `ts2` to the `tsc` collection.

```
tsc = addts(tsc, ts2);
```

4 To view the members of `tsc`, type

```
tsc
```

at the MATLAB prompt. MATLAB responds with

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:

      acceleration
      speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of the `timeseries` objects `ts1` and `ts2`, respectively.

See Also

`removets`, `tscollection`

Purpose

Airy functions

Syntax

```
W = airy(Z)
W = airy(k,Z)
[W,ierr] = airy(k,Z)
```

Definition

The Airy functions form a pair of linearly independent solutions to

$$\frac{d^2W}{dZ^2} - ZW = 0$$

The relationship between the Airy and modified Bessel functions is

$$Ai(Z) = \left[\frac{1}{\pi}\sqrt{Z/3}\right] K_{1/3}(\zeta)$$

$$Bi(Z) = \sqrt{Z/3}\ \ [I_{-1/3}(\zeta) + I_{1/3}(\zeta)]$$

where

$$\zeta = \frac{2}{3}Z^{3/2}$$

Description

`W = airy(Z)` returns the Airy function, $Ai(Z)$, for each element of the complex array `Z`.

`W = airy(k,Z)` returns different results depending on the value of `k`.

k	Returns
0	The same result as `airy(Z)`
1	The derivative, $Ai'(Z)$

k	Returns
2	The Airy function of the second kind, $Bi(Z)$
3	The derivative, $Bi'(Z)$

`[W,ierr] = airy(k,Z)` also returns completion flags in an array the same size as `W`.

ierr	Description
0	`airy` successfully computed the Airy function for this element.
1	Illegal arguments
2	Overflow. Returns `Inf`
3	Some loss of accuracy in argument reduction
4	Unacceptable loss of accuracy, `Z` too large
5	No convergence. Returns `NaN`

See Also

`besseli`, `besselj`, `besselk`, `bessely`

References

[1] Amos, D. E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[2] Amos, D. E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

align

Purpose

Align user interface controls (uicontrols) and axes

Syntax

```
align(HandleList,'HorizontalAlignment','VerticalAlignment')
Positions = align(HandleList,'HorizontalAlignment',
   'VerticalAlignment')
Positions = align(CurPositions,'HorizontalAlignment',
   'VerticalAlignment')
```

Description

`align(HandleList,'HorizontalAlignment','VerticalAlignment')` aligns the `uicontrol` and `axes` objects in `HandleList`, a vector of handles, according to the options `HorizontalAlignment` and `VerticalAlignment`. The following table shows the possible values for `HorizontalAlignment` and `VerticalAlignment`.

Argument	Possible Values
HorizontalAlignment	`None`, `Left`, `Center`, `Right`, `Distribute`, `Fixed`
VerticalAlignment	`None`, `Top`, `Middle`, `Bottom`, `Distribute`, `Fixed`

All alignment options align the objects within the bounding box that encloses the objects. `Distribute` and `Fixed` align objects to the bottom left of the bounding box. `Distribute` evenly distributes the objects while `Fixed` distributes the objects with a fixed distance (in points) between them.

If you use `Fixed` for `Horizontal Alignment` or `Vertical Alignment`, then you must specify the distance, in points, as an extra argument. These are some examples:

```
align(HandleList,'Fixed',Distance,'VerticalAlignment')
```

distributes the specified components `Distance` points horizontally and aligns them vertically as specified.

```
align(HandleList,'HorizontalAlignment','Fixed',Distance)
```

aligns the specified components horizontally as specified and distributes them `Distance` points vertically.

```
align(HandleList,'Fixed','HorizontalDistance',...
      'Fixed','VerticalDistance')
```

distributes the specified components `HorizontalDistance` points horizontally and distributes them `VerticalDistance` points vertically.

Note 72 points equals 1 inch.

`Positions = align(HandleList,'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the specified objects as a vector of `Position` vectors. The position of the objects on the figure does not change.

`Positions = align(CurPositions,'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the objects whose positions are contained in `CurPositions`, where `CurPositions` is a vector of `Position` vectors. The position of the objects on the figure does not change.

alim

Purpose

Set or query axes alpha limits

Syntax

```
alpha_limits = alim
alim([amin amax])
alim_mode = alim('mode')
alim('alim_mode')
alim(axes_handle,...)
```

Description

`alpha_limits = alim` returns the alpha limits (the axes `ALim` property) of the current axes.

`alim([amin amax])` sets the alpha limits to the specified values. `amin` is the value of the data mapped to the first alpha value in the alphamap, and `amax` is the value of the data mapped to the last alpha value in the alphamap. Data values in between are linearly interpolated across the alphamap, while data values outside are clamped to either the first or last alphamap value, whichever is closest.

`alim_mode = alim('mode')` returns the alpha limits mode (the axes `ALimMode` property) of the current axes.

`alim('alim_mode')` sets the alpha limits mode on the current axes. `alim_mode` can be

- auto — MATLAB automatically sets the alpha limits based on the alpha data of the objects in the axes.
- `manual` — MATLAB does not change the alpha limits.

`alim(axes_handle,...)` operates on the specified axes.

See Also

`alpha`, `alphamap`, `caxis`

Axes `ALim` and `ALimMode` properties

Patch `FaceVertexAlphaData` property

Image and surface `AlphaData` properties

Transparency for related functions

"Transparency" in 3-D Visualization for examples

Purpose

Determine whether all array elements are nonzero

Syntax

```
B = all(A)
B = all(A, dim)
```

Description

`B = all(A)` tests whether *all* the elements along various dimensions of an array are nonzero or logical `1` (`true`).

If `A` is a vector, `all(A)` returns logical `1` (`true`) if all the elements are nonzero and returns logical `0` (`false`) if one or more elements are zero.

If `A` is a matrix, `all(A)` treats the columns of `A` as vectors, returning a row vector of logical 1's and 0's.

If `A` is a multidimensional array, `all(A)` treats the values along the first nonsingleton dimension as vectors, returning a logical condition for each vector.

`B = all(A, dim)` tests along the dimension of `A` specified by scalar *`dim`*.

Examples

Given

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
```

then `B = (A < 0.5)` returns logical `1` (`true`) only where `A` is less than one half:

```
0    0    1    1    1    1    0
```

The `all` function reduces such a vector of logical conditions to a single condition. In this case, `all(B)` yields `0`.

This makes `all` particularly useful in `if` statements:

```
if all(A < 0.5)
    do something
end
```

where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

Applying the `all` function twice to a matrix, as in `all(all(A))`, always reduces it to a scalar condition.

```
all(all(eye(3)))
ans =
   0
```

See Also

`any`, logical operators (elementwise and short-circuit), relational operators, `colon`

Other functions that collapse an array's dimensions include `max`, `mean`, `median`, `min`, `prod`, `std`, `sum`, and `trapz`.

allchild

Purpose

Find all children of specified objects

Syntax

```
child_handles = allchild(handle_list)
```

Description

`child_handles = allchild(handle_list)` returns the list of all children (including ones with hidden handles) for each handle. If `handle_list` is a single element, `allchild` returns the output in a vector. Otherwise, the output is a cell array.

Examples

Compare the results returned by these two statements.

```
get(gca,'Children')
allchild(gca)
```

See Also

`findall`, `findobj`

Purpose

Set transparency properties for objects in current axes

Syntax

```
alpha
alpha(face_alpha)
alpha(alpha_data)
alpha(alpha_data)
alpha(alpha_data)
alpha(alpha_data_mapping)
alpha(object_handle,value)
```

Description

`alpha` sets one of three transparency properties, depending on what arguments you specify with the call to this function.

FaceAlpha

`alpha(face_alpha)` sets the `FaceAlpha` property of all image, patch, and surface objects in the current axes. You can set `face_alpha` to

- A scalar — Set the `FaceAlpha` property to the specified value (for images, set the `AlphaData` property to the specified value).
- `'flat'` — Set the `FaceAlpha` property to `flat`.
- `'interp'` — Set the `FaceAlpha` property to `interp`.
- `'texture'` — Set the `FaceAlpha` property to `texture`.
- `'opaque'` — Set the `FaceAlpha` property to `1`.
- `'clear'` — Set the `FaceAlpha` property to `0`.

See for more information.

AlphaData (Surface Objects)

`alpha(alpha_data)` sets the `AlphaData` property of all surface objects in the current axes. You can set `alpha_data` to

- A matrix the same size as `CData` — Set the `AlphaData` property to the specified values.
- `'x'` — Set the `AlphaData` property to be the same as `XData`.

- `'y'` — Set the `AlphaData` property to be the same as `YData`.
- `'z'` — Set the `AlphaData` property to be the same as `ZData`.
- `'color'` — Set the `AlphaData` property to be the same as `CData`.
- `'rand'` — Set the `AlphaData` property to a matrix of random values equal in size to `CData`.

AlphaData (Image Objects)

`alpha(alpha_data)` sets the `AlphaData` property of all image objects in the current axes. You can set `alpha_data` to

- A matrix the same size as `CData` — Set the `AlphaData` property to the specified value.
- `'x'` — Ignored.
- `'y'` — Ignored.
- `'z'` — Ignored.
- `'color'` — Set the `AlphaData` property to be the same as `CData`.
- `'rand'` — Set the `AlphaData` property to a matrix of random values equal in size to `CData`.

FaceVertexAlphaData (Patch Objects)

`alpha(alpha_data)` sets the `FaceVertexAlphaData` property of all patch objects in the current axes. You can set `alpha_data` to

- A matrix the same size as `FaceVertexCData` — Set the `FaceVertexAlphaData` property to the specified value.
- `'x'` — Set the `FaceVertexAlphaData` property to be the same as `Vertices(:,1)`.
- `'y'` — Set the `FaceVertexAlphaData` property to be the same as `Vertices(:,2)`.
- `'z'` — Set the `FaceVertexAlphaData` property to be the same as `Vertices(:,3)`.

- `'color'` — Set the `FaceVertexAlphaData` property to be the same as `FaceVertexCData`.
- `'rand'` — Set the `FaceVertexAlphaData` property to random values.

See Mapping Data to Transparency for more information.

AlphaDataMapping

`alpha(alpha_data_mapping)` sets the `AlphaDataMapping` property of all image, patch, and surface objects in the current axes. You can set `alpha_data_mapping` to

- `'scaled'` — Set the `AlphaDataMapping` property to `scaled`.
- `'direct'` — Set the `AlphaDataMapping` property to `direct`.
- `'none'` — Set the `AlphaDataMapping` property to `none`.

`alpha(object_handle,value)` sets the transparency property only on the object identified by `object_handle`.

See Also

`alim`, `alphamap`

Image: `AlphaData`, `AlphaDataMapping`

Patch: `FaceAlpha`, `FaceVertexAlphaData`, `AlphaDataMapping`

Surface: `FaceAlpha`, `AlphaData`, `AlphaDataMapping`

Transparency for related functions

“Transparency” in 3-D Visualization for examples

alphamap

Purpose

Specify figure alphamap (transparency)

Syntax

```
alphamap
alphamap(alpha_map)
alphamap('parameter')
alphamap('parameter',length)
alphamap('parameter',delta)
alphamap(figure_handle,...)
alpha_map = alphamap
alpha_map = alphamap(figure_handle)
alpha_map = alphamap('parameter')
```

Description

`alphamap` enables you to set or modify a figure's `Alphamap` property. Unless you specify a figure handle as the first argument, `alphamap` operates on the current figure.

`alphamap(alpha_map)` sets the `AlphaMap` of the current figure to the specified m-by-1 array of alpha values.

`alphamap('parameter')` creates a new alphamap or modifies the current alphamap. You can specify the following parameters:

- `default` — Set the `AlphaMap` property to the figure's default alphamap.
- `rampup` — Create a linear alphamap with increasing opacity (default `length` equals the current alphamap length).
- `rampdown` — Create a linear alphamap with decreasing opacity (default `length` equals the current alphamap length).
- `vup` — Create an alphamap that is opaque in the center and becomes more transparent linearly towards the beginning and end (default `length` equals the current alphamap length).
- `vdown` — Create an alphamap that is transparent in the center and becomes more opaque linearly towards the beginning and end (default `length` equals the current alphamap length).

- `increase` — Modify the alphamap making it more opaque (default `delta` is `.1`, which is added to the current values).
- `decrease` — Modify the alphamap making it more transparent (default `delta` is `.1`, which is subtracted from the current values).
- `spin` — Rotate the current alphamap (default `delta` is 1; note that `delta` must be an integer).

`alphamap('parameter',length)` creates a new alphamap with the length specified by `length` (used with parameters `rampup`, `rampdown`, `vup`, `vdown`).

`alphamap('parameter',delta)` modifies the existing alphamap using the value specified by `delta` (used with parameters `increase`, `decrease`, `spin`).

`alphamap(figure_handle,...)` performs the operation on the alphamap of the figure identified by `figure_handle`.

`alpha_map = alphamap` returns the current alphamap.

`alpha_map = alphamap(figure_handle)` returns the current alphamap from the figure identified by `figure_handle`.

`alpha_map = alphamap('parameter')` returns the alphamap modified by the `parameter`, but does not set the `AlphaMap` property.

See Also

`alim`, `alpha`

Image: `AlphaData`, `AlphaDataMapping`

Patch: `FaceAlpha`, `FaceVertexAlphaData`, `AlphaDataMapping`

Surface: `FaceAlpha`, `AlphaData`, `AlphaDataMapping`

Transparency for related functions

“Transparency” in 3-D Visualization for examples

Purpose

Approximate minimum degree permutation

Syntax

```
P = amd(A)
P = amd(A,opts)
```

Description

`P = amd(A)` returns the approximate minimum degree permutation vector for the sparse matrix `C = A + A'`. The Cholesky factorization of `C(P,P)` or `A(P,P)` tends to be sparser than that of `C` or `A`. The `amd` function tends to be faster than `symamd`, and also tends to return better orderings than `symamd`. Matrix `A` must be square. If `A` is a full matrix, then `amd(A)` is equivalent to `amd(sparse(A))`.

`P = amd(A,opts)` allows additional options for the reordering. The `opts` input is a structure with the two fields shown below. You only need to set the fields of interest:

- **dense** — A nonnegative scalar value that indicates what is considered to be dense. If A is n-by-n, then rows and columns with more than `max(16,(dense*sqrt(n)))` entries in `A + A'` are considered to be "dense" and are ignored during the ordering. MATLAB places these rows and columns last in the output permutation. The default value for this field is 10.0 if this option is not present.
- **aggressive** — A scalar value controlling aggressive absorption. If this field is set to a nonzero value, then aggressive absorption is performed. This is the default if this option is not present.

MATLAB performs an assembly tree post-ordering, which is typically the same as an elimination tree post-ordering. It is not always identical because of the approximate degree update used, and because “dense” rows and columns do not take part in the post-order. It well-suited for a subsequent `chol` operation, however, If you require a precise elimination tree post-ordering, you can use the following code:

```
P = amd(S);
C = spones(S)+spones(S'); % Skip this line if S is already symmetric
[ignore, Q] = etree(C(P,P));
P = P(Q);
```

Examples

This example constructs a sparse matrix and computes a two Cholesky factors: one of the original matrix and one of the original matrix preordered by `amd`. Note how much sparser the Cholesky factor of the preordered matrix is compared to the factor of the matrix in its natural ordering:

```
A = gallery('wathen',50,50);
p = amd(A);
L = chol(A,'lower');
Lp = chol(A(p,p),'lower');

figure;
subplot(2,2,1);    spy(A);
title('Sparsity structure of A');

subplot(2,2,2); spy(A(p,p));
title('Sparsity structure of AMD ordered A');

subplot(2,2,3); spy(L);
title('Sparsity structure of Cholesky factor of A');

subplot(2,2,4); spy(Lp);
title('Sparsity structure of Cholesky factor of AMD ordered A');

set(gcf,'Position',[100 100 800 700]);
```

See Also

`colamd`, `colperm`, `symamd`, `symrcm`, `/`

References

AMD Version 1.2 is written and copyrighted by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. It is available at `http://www.cise.ufl.edu/research/sparse/amd`.

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (`davis@cise.ufl.edu`), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory.

Sparse Matrix Algorithms Research at the University of Florida:
`http://www.cise.ufl.edu/research/sparse/`

Purpose

Ancestor of graphics object

Syntax

```
p = ancestor(h,type)
p = ancestor(h,type,'toplevel')
```

Description

`p = ancestor(h,type)` returns the handle of the closest ancestor of `h`, if the ancestor is one of the types of graphics objects specified by `type`. `type` can be:

- a string that is the name of a single type of object. For example, `'figure'`
- a cell array containing the names of multiple objects. For example, `{'hgtransform','hggroup','axes'}`

If MATLAB cannot find an ancestor of `h` that is one of the specified types, then `ancestor` returns `p` as empty.

Note that `ancestor` returns `p` as empty but does not issue an error if `h` is not the handle of a Handle Graphics object.

`p = ancestor(h,type,'toplevel')` returns the highest-level ancestor of `h`, if this type appears in the `type` argument.

Examples

Create some line objects and parent them to an hggroup object.

```
hgg = hggroup;
hgl = line(randn(5),randn(5),'Parent',hgg);
```

Now get the ancestor of the lines.

```
p = ancestor(hgg,{'figure','axes','hggroup'});
get(p,'Type')
ans =

hggroup
```

Now get the top-level ancestor

```
p=ancestor(hgg,{'figure','axes','hggroup'},'toplevel');
get(p,'type')
ans =

figure
```

See Also

findobj

Purpose

Find logical AND of array or scalar inputs

Syntax

```
A & B & ...
and(A, B)
```

Description

`A & B & ...` performs a logical AND of all input arrays `A`, `B`, etc., and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if all input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input `A` is a 3-by-5 matrix and input `B` is the number 1, then `B` is treated as if it were a 3-by-5 matrix of ones.

`and(A, B)` is called for the syntax `A & B` when either `A` or `B` is an object.

Note The symbols `&` and `&&` perform different operations in MATLAB. The element-wise AND operator described here is `&`. The short-circuit AND operator is `&&`.

Examples

If matrix `A` is

```
    0.4235    0.5798         0    0.7942         0
    0.5155         0    0.7833    0.0592    0.8744
    0.3340         0         0         0    0.0150
    0.4329    0.6405    0.6808    0.0503         0
```

and matrix `B` is

```
     0     1     0     1     0
     1     1     1     0     1
     0     1     1     1     0
     0     1     0     0     1
```

then

```
A & B
ans =
     0     1     0     1     0
     1     0     1     0     1
     0     0     0     0     0
     0     1     0     0     0
```

See Also

`bitand`, `or`, `xor`, `not`, `any`, `all`, logical operators, logical types, bitwise functions

Purpose

Phase angle

Syntax

```
P = angle(Z)
```

Description

`P = angle(Z)` returns the phase angles, in radians, for each element of complex array `Z`. The angles lie between $\pm\pi$.

For complex `Z`, the magnitude `R` and phase angle `theta` are given by

```
R = abs(Z)
theta = angle(Z)
```

and the statement

```
Z = R.*exp(i*theta)
```

converts back to the original complex `Z`.

Examples

```
Z = [ 1 - 1i   2 + 1i   3 - 1i   4 + 1i
      1 + 2i   2 - 2i   3 + 2i   4 - 2i
      1 - 3i   2 + 3i   3 - 3i   4 + 3i
      1 + 4i   2 - 4i   3 + 4i   4 - 4i ]

P = angle(Z)

P =
   -0.7854    0.4636   -0.3218    0.2450
    1.1071   -0.7854    0.5880   -0.4636
   -1.2490    0.9828   -0.7854    0.6435
    1.3258   -1.1071    0.9273   -0.7854
```

Algorithm

The `angle` function can be expressed as `angle(z) = imag(log(z)) = atan2(imag(z),real(z))`.

See Also

`abs`, `atan2`, `unwrap`

Purpose

Create annotation objects

GUI Alternatives

Create several types of annotations with the Figure Palette and modify annotations with the Property Editor, components of the plotting tools. Directly manipulate annotations in *plot edit* mode. For details, see "How to Annotate Graphs" and "Working in Plot Edit Mode" in the MATLAB Graphics documentation.

Syntax

```
annotation(annotation_type)
annotation('line',x,y)
annotation('arrow',x,y)
annotation('doublearrow',x,y)
annotation('textarrow',x,y)
annotation('textbox',[x y w h])
annotation('ellipse',[x y w h])
annotation('rectangle',[x y w h])
annotation(figure_handle,...)
annotation(...,'PropertyName',PropertyValue,...)
anno_obj_handle = annotation(...)
```

Description

`annotation(annotation_type)` creates the specified annotation type using default values for all properties. *`annotation_type`* can be one of the following strings:

- `'line'`
- `'arrow'`
- `'doublearrow'` (two-headed arrow),
- `'textarrow'` (arrow with attached text box),
- `'textbox'`
- `'ellipse'`
- `'rectangle'`

`annotation('line',x,y)` creates a line annotation object that extends from the point defined by `x(1),y(1)` to the point defined by `x(2),y(2)`, specified in normalized figure units.

`annotation('arrow',x,y)` creates an arrow annotation object that extends from the point defined by `x(1),y(1)` to the point defined by `x(2),y(2)`, specified in normalized figure units.

`annotation('doublearrow',x,y)` creates a two-headed annotation object that extends from the point defined by `x(1),y(1)` to the point defined by `x(2),y(2)`, specified in normalized figure units.

`annotation('textarrow',x,y)` creates a textarrow annotation object that extends from the point defined by `x(1),y(1)` to the point defined by `x(2),y(2)`, specified in normalized figure units. The tail end of the arrow is attached to an editable text box.

`annotation('textbox',[x y w h])` creates an editable text box annotation with its lower left corner at the point `x,y`, a width `w`, and a height `h`, specified in normalized figure units. Specify `x`, `y`, `w`, and `h` in a single vector.

To type in the text box, enable plot edit mode (`plotedit`) and double-click within the box.

`annotation('ellipse',[x y w h])` creates an ellipse annotation with the lower left corner of the bounding rectangle at the point `x,y`, a width `w`, and a height `h`, specified in normalized figure units. Specify `x`, `y`, `w`, and `h` in a single vector.

`annotation('rectangle',[x y w h])` creates a rectangle annotation with the lower left corner of the rectangle at the point `x,y`, a width `w`, and a height `h`, specified in normalized figure units. Specify `x`, `y`, `w`, and `h` in a single vector.

`annotation(figure_handle,...)` creates the annotation in the specified figure.

`annotation(...,'`*`PropertyName`*`',PropertyValue,...)` creates the annotation and sets the specified properties to the specified values.

`anno_obj_handle = annotation(...)` returns the handle to the annotation object that is created.

Annotation Layer

All annotation objects are displayed in an overlay axes that covers the figure. This layer is designed to display only annotation objects. You should not parent objects to this axes nor set any properties of this axes. See the See Also section for information on the properties of annotation objects that you can set.

Objects in the Plotting Axes

You can create lines, text, rectangles, and ellipses in data coordinates in the axes of a graph using the `line`, `text`, and `rectangle` functions. These objects are not placed in the annotation axes and must be located inside their parent axes.

Deleting Annotations

Existing annotations persist on a plot when you replace its data. This might not be what you want to do. If it is not, or if you want to remove annotation objects for any reason, you can do so manually, or sometimes programmatically, in several ways:

- To manually delete, click the **Edit Plot** tool or invoke `plottools`, select the annotation(s) you want to remove, and do one of the following:
  - Press the **Delete** key.
  - Press the **Backspace** key.
  - Select **Clear** from the **Edit** menu.
  - Select **Delete** from the context menu (one annotation at a time).
- If you obtained a handle for the annotation when you created it, use the `delete` function:

  ```
  delete(anno_obj_handle)
  ```

  There is no reliable way to obtain handles for annotations from a figure's property set; you must keep track of them yourself.

- To delete all annotations at once (as well as all plot contents), type

```
clf
```

Normalized Coordinates

By default, annotation objects use normalized coordinates to specify locations within the figure. In normalized coordinates, the point 0,0 is always the lower left corner and the point 1,1 is always the upper right corner of the figure window, regardless of the figure size and proportions. Set the `Units` property of annotation objects to change their coordinates from `normalized` to `inches`, `centimeters`, `points`, `pixels`, or `characters`.

When their `Units` property is other than `normalized`, annotation objects have absolute positions with respect to the figure's origin, and fixed sizes. Therefore, they will shift position with respect to axes when you resize figures. When units are normalized, annotations shrink and grow when you resize figures; this can cause lines of text in textbox annotations to wrap. However, if you set the `FontUnits` property of an annotation textbox object to `normalized`, the text changes size rather than wraps if the textbox size changes.

You can use either the `set` command or the Inspector to change a selected annotation object's `Units` property:

```
set(gco,'Units','inches')  % or
inspect(gco)
```

See Also

Properties for the annotation objects `Annotation Arrow Properties`, `Annotation Doublearrow Properties`, `Annotation Ellipse Properties`, `Annotation Line Properties`, `Annotation Rectangle Properties`, `Annotation Textarrow Properties`, `Annotation Textbox Properties`

See "Annotating Graphs" and "Annotation Objects" for more information.

Annotation Arrow Properties

Purpose

Define annotation arrow properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Arrow Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation arrow object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

See the `ColorSpec` reference page for more information on specifying color.

`HeadLength`
scalar value in points

Length of the arrowhead. Specify this property in points (1 point = 1/72 inch). See also `HeadWidth`.

`HeadStyle`
select string from list

Style of the arrowhead. Specify this property as one of the strings from the following table.

Head Style String	Head	Head Style String	Head
none		star4	
plain		rectangle	
ellipse		diamond	
vback1		rose	
vback2 (Default)		hypocycloid	
vback3		astroid	
cback1		deltoid	
cback2			
cback3			

`HeadWidth`
scalar value in points

Width of the arrowhead. Specify this property in points (1 point = 1/72 inch). See also `HeadLength`.

`LineStyle`
`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Position`
four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x, y* in units normalized to the figure (when `Units` property is `normalized`). The third and fourth elements specify the object's *dx* and *dy*, respectively, in units normalized to the figure.

`Units`
`{normalized} | inches | centimeters | points | pixels`

position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured

from the lower left corner of the figure window. Normalized units interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`X`

vector $[X_{begin}\ X_{end}]$

X-coordinates of the beginning and ending points for line. Specify this property as a vector of *x*-axis (horizontal) values that specify the beginning and ending points of the line, units normalized to the figure.

`Y`

vector $[Y_{begin}\ Y_{end}]$

Y-coordinates of the beginning and ending points for line. Specify this property as a vector of *y*-axis (vertical) values that specify the beginning and ending points of the line, units normalized to the figure.

Annotation Doublearrow Properties

Purpose

Define annotation doublearrow properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Doublearrow Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation doublearrow object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

See the `ColorSpec` reference page for more information on specifying color.

`Head1Length`
scalar value in points

Length of the first arrowhead. Specify this property in points (1 point = 1/72 inch). See also `Head1Width`.

The first arrowhead is located at the end defined by the point `x(1)`, `y(1)`. See also the `X` and `Y` properties.

`Head2Length`
scalar value in points

Length of the second arrowhead. Specify this property in points (1 point = 1/72 inch). See also `Head1Width`.

The first arrowhead is located at the end defined by the point `x(end)`, `y(end)`. See also the `X` and `Y` properties.

`Head1Style`
select string from list

Style of the first arrowhead. Specify this property as one of the strings from the following table

`Head2Style`
select string from list

Style of the second arrowhead. Specify this property as one of the strings from the following table.

Head Style String	Head	Head Style String	Head
`none`		`star4`	
`plain`		`rectangle`	
`ellipse`		`diamond`	
`vback1`		`rose`	
`vback2` (Default)		`hypocycloid`	
`vback3`		`astroid`	
`cback1`		`deltoid`	

Head Style String	Head	Head Style String	Head
`cback2`			
`cback3`			

`Head1Width`
scalar value in points

Width of the first arrowhead. Specify this property in points (1 point = 1/72 inch). See also Head1Length.

`Head2Width`
scalar value in points

Width of the second arrowhead. Specify this property in points (1 point = 1/72 inch). See also Head2Length.

`LineStyle`
`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Position`
four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x, y* in units normalized to the figure (when `Units` property is `normalized`). The third and fourth elements specify the object's *dx* and *dy*, respectively, in units normalized to the figure.

`Units`
`{normalized} | inches | centimeters | points | pixels`

position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window. Normalized units interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`X`
vector [X_{begin} X_{end}]

X-coordinates of the beginning and ending points for line. Specify this property as a vector of *x*-axis (horizontal) values that specify the beginning and ending points of the line, units normalized to the figure.

Y

vector [Y_{begin} Y_{end}]

Y-coordinates of the beginning and ending points for line. Specify this property as a vector of *y*-axis (vertical) values that specify the beginning and ending points of the line, units normalized to the figure.

Purpose

Define annotation ellipse properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Ellipse Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation ellipse object.

`EdgeColor`

ColorSpec `{[0 0 0]} | none |`

Color of the object's edges. A three-element RGB vector or one of the MATLAB predefined names, specifying the edge color.

See the `ColorSpec` reference page for more information on specifying color.

`FaceColor`

`{flat} | none | ColorSpec`

Color of filled areas. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that `EdgeColor` is drawn independently of `FaceColor`
- `flat` — The color of the filled areas is determined by the figure colormap. See `colormap` for information on setting the colormap.

See the `ColorSpec` reference page for more information on specifying color.

`LineStyle`

`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Position`

four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x, y* in units normalized to the figure (when `Units` property is `normalized`). The third and fourth elements specify the object's *dx* and *dy*, respectively, in units normalized to the figure.

`Units`

`{normalized} | inches | centimeters | points | pixels`

position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window. Normalized units interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

Annotation Line Properties

Purpose

Define annotation line properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Line Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation line object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

See the `ColorSpec` reference page for more information on specifying color.

`LineStyle`
`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line

Specifier String	Line Style
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Position`
four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x, y* in units normalized to the figure (when `Units` property is `normalized`). The third and fourth elements specify the object's *dx* and *dy*, respectively, in units normalized to the figure.

`Units`
`{normalized} | inches | centimeters | points | pixels`

position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window. Normalized units interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

X

vector [X_{begin} X_{end}]

X-coordinates of the beginning and ending points for line. Specify this property as a vector of *x*-axis (horizontal) values that specify the beginning and ending points of the line, units normalized to the figure.

Y

vector [Y_{begin} Y_{end}]

Y-coordinates of the beginning and ending points for line. Specify this property as a vector of *y*-axis (vertical) values that specify the beginning and ending points of the line, units normalized to the figure.

Purpose

Define annotation rectangle properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Rectangle Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation rectangle object.

`EdgeColor`
ColorSpec `{[0 0 0]} | none |`

Color of the object's edges. A three-element RGB vector or one of the MATLAB predefined names, specifying the edge color.

See the `ColorSpec` reference page for more information on specifying color.

`FaceAlpha`
Scalar alpha value in range [0 1]

Transparency of object background. This property defines the degree to which the object's background color is transparent. A value of `1` (the default) makes to color opaque, a value of `0` makes the background completely transparent (i.e., invisible). The default FaceAlpha is `1`.

`FaceColor`
`{flat} | none | ColorSpec`

Color of filled areas. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that `EdgeColor` is drawn independently of `FaceColor`
- `flat` — The color of the filled areas is determined by the figure colormap. See `colormap` for information on setting the colormap.

  See the `ColorSpec` reference page for more information on specifying color.

`LineStyle`

```
{-} | -- | :  | -.  | none
```

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^{1}/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Position`

four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x, y* in units normalized to the figure (when `Units` property is `normalized`). The third and fourth elements specify the object's *dx* and *dy*, respectively, in units normalized to the figure.

`Units`

`{normalized} | inches | centimeters | points | pixels`

position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window. Normalized units interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

Annotation Textarrow Properties

Purpose

Define annotation textarrow properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Textarrow Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation textarrow object.

`Color`

ColorSpec Default: [0 0 0]

Color of the arrow, text and text border. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the arrow, the color of the text (`TextColor` property), and the rectangle enclosing the text (`TextEdgeColor` property).

Setting the `Color` property also sets the `TextColor` and `TextEdgeColor` properties to the same color. However, if the value of the `TextEdgeColor` is `none`, it remains `none` and the text box is not displayed. You can set `TextColor` or `TextEdgeColor` independently without affecting other properties.

For example, if you want to create a textarrow with a red arrow and black text in a black box, you must

1. Set the `Color` property to red — `set(h,'Color','r')`
2. Set the `TextColor` to black — `set(h,'TextColor','k')`
3. Set the `TextEdgeColor` to black .— `set(h,'TextEdgeColor','k')`

If you do not want display the text box, set the `TextEdgeColor` to `none`.

See the `ColorSpec` reference page for more information on specifying color.

`FontAngle`
`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

`FontName`
A name, such as `Helvetica`

Font family. A string specifying the name of the font to use for the text. To display and print properly, this font must be supported on your system. The default font is `Helvetica`.

`FontSize`
size in points

Approximate size of text characters. A value specifying the font size to use in points. The default size is 10 (1 point = 1/72 inch).

`FontUnits`
`{points} | normalized | inches | centimeters | pixels`

Font size units. MATLAB uses this property to determine the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your system. Generally, setting this property to `bold` or `demi` causes MATLAB to use a bold font.

`HeadLength`
scalar value in points

Length of the arrowhead. Specify this property in points (1 point = 1/72 inch). See also `HeadWidth`.

`HeadStyle`
select string from list

Style of the arrowhead. Specify this property as one of the strings from the following table.

Head Style String	Head	Head Style String	Head
`none`		`star4`	
`plain`		`rectangle`	
`ellipse`		`diamond`	
`vback1`		`rose`	
`vback2` (Default)		`hypocycloid`	
`vback3`		`astroid`	
`cback1`		`deltoid`	

Head Style String	Head	Head Style String	Head
`cback2`			
`cback3`			

`HeadWidth`
scalar value in points

Width of the arrowhead. Specify this property in points (1 point = 1/72 inch). See also `HeadLength`.

`HorizontalAlignment`
`{left} | center | right`

Horizontal alignment of text. This property specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the `Position` property. The following picture illustrates the alignment options.

`HorizontalAlignment` viewed with the `VerticalAlignment` set to middle (the default).

See the `Extent` property for related information.

`Interpreter`
latex | `{tex}` | `none`

Interpret T_EX instructions. This property controls whether MATLAB interprets certain characters in the `String` property as T_EX instructions (default) or displays all characters literally. The options are:

- `latex` — Supports the full L_AT_EX markup language.
- `tex` — Supports a subset of plain T_EX markup language. See the `String` property for a list of supported T_EX instructions.
- `none` — Displays literal characters.

`LineStyle`

`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Position`
four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x, y* in units normalized to the figure (when `Units` property is `normalized`). The third and fourth elements specify the object's *dx* and *dy*, respectively, in units normalized to the figure.

`String`
string

The text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

When the text `Interpreter` property is set to `Tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\beta`	β	`\phi`	Φ	`\leq`	≤
`\gamma`	γ	`\chi`	χ	`\infty`	∞
`\delta`	δ	`\psi`	ψ	`\clubsuit`	♣
`\epsilon`	ε	`\omega`	ω	`\diamondsuit`	♦
`\zeta`	ζ	`\Gamma`	Γ	`\heartsuit`	♥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	θ	\Theta	Θ	\leftrightarrow	↔
\vartheta	ϑ	\Lambda	Λ	\leftarrow	→
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	↔
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	μ	\Upsilon	ϒ	\circ	º
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∍	\div	÷
\tau	τ	\cong	≅	\neq	≠
\equiv	≡	\approx	∼	\aleph	
\Im	ℑ	\Re	ℜ	\wp	℘
\otimes	⊗	\oplus	⊕	\oslash	∅
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in		\o	o
\rfloor	⌋	\lceil	⌈	\nabla	∇
\lfloor	⌊	\cdot	·	\ldots	...

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\perp	⊥	\neg	¬	\prime	´
\wedge	∧	\times	x	\0	∅
\rceil	⌉	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	∠	\rangle	∠		

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers:

`TextBackgroundColor`
`ColorSpec` Default: `none`

Color of text background rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the `ColorSpec` reference page for more information on specifying color.

`TextColor`
`ColorSpec` Default: `[0 0 0]`

Color of text. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the `ColorSpec` reference page for more information on specifying color. Setting the `Color` property also sets this property.

`TextEdgeColor`
`ColorSpec` or `none` Default: `none`

Color of edge of text rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the rectangle that encloses the text.

See the `ColorSpec` reference page for more information on specifying color. Setting the `Color` property also sets this property.

`TextLineWidth`
width in points

The width of the text rectangle edge. Specify this value in points (1 point = $^1/_{72}$ inch). The default `TextLineWidth` is 0.5 points.

`TextMargin`
dimension in pixels default: 5

Space around text. Specify a value in pixels that defines the space around the text string, but within the rectangle.

`TextRotation`
rotation angle in degrees (default = 0)

Text orientation. This property determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation). Angles are absolute and not relative to previous rotations; a rotation of 0 degrees is always horizontal.

`Units`
`{normalized} | inches | centimeters | points | pixels`

position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window. Normalized units interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

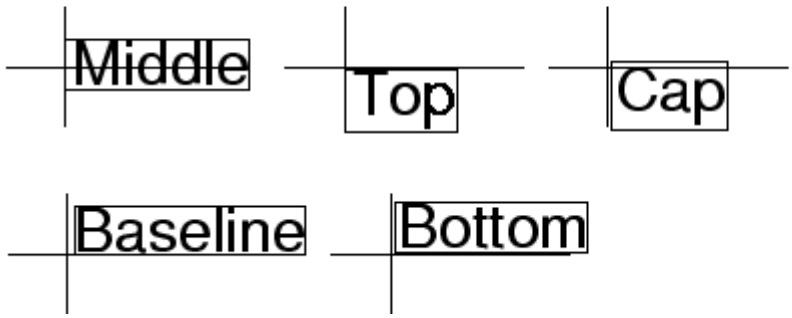
```
VerticalAlignment
     top | cap | {middle} | baseline |
     bottom
```

Vertical alignment of text. This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean

- top — Place the top of the string' s `Extent` rectangle at the specified *y*-position.
- `cap` — Place the string so that the top of a capital letter is at the specified *y*-position.
- `middle` — Place the middle of the string at the specified *y*-position.
- `baseline` — Place font baseline at the specified *y*-position.
- `bottom` — Place the bottom of the string's `Extent` rectangle at the specified *y*-position.

The following picture illustrates the alignment options.

Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



X

vector [X_{begin} X_{end}]

X-coordinates of the beginning and ending points for line. Specify this property as a vector of *x*-axis (horizontal) values that specify the beginning and ending points of the line, units normalized to the figure.

Y

vector [Y_{begin} Y_{end}]

Y-coordinates of the beginning and ending points for line. Specify this property as a vector of *y*-axis (vertical) values that specify the beginning and ending points of the line, units normalized to the figure.

Purpose

Define annotation textbox properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Textbox Property Descriptions

Properties You Can Modify

This section lists the properties you can modify on an annotation textbox object.

`BackgroundColor`
`ColorSpec` Default: `none`

Color of text background rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the `ColorSpec` reference page for more information on specifying color.

`Color`
`ColorSpec` Default: `[0 0 0]`

Color of text. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the `ColorSpec` reference page for more information on specifying color. Setting the `Color` property also sets this property.

`EdgeColor`
`ColorSpec` or `none` Default: `none`

Color of edge of text rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the rectangle that encloses the text.

See the `ColorSpec` reference page for more information on specifying color. Setting the `Color` property also sets this property.

`FaceAlpha`
Scalar alpha value in range [0 1]

Transparency of object background. This property defines the degree to which the object's background color is transparent. A value of `1` (the default) makes to color opaque, a value of `0` makes the background completely transparent (i.e., invisible). The default FaceAlpha is `1`.

`FitBoxToText`
`on | off`

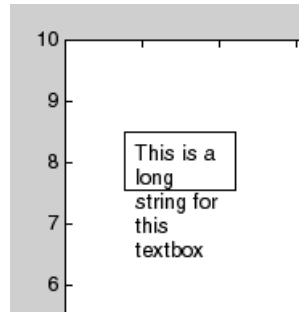
Automatically adjust text box width and height to fit text. When this property is `on` (the default), MATLAB automatically resizes textboxes to fit the *x*-extents and *y*-extents of the text strings they contain. When it is `off`, text strings are wrapped to fit the width of their textboxes, which can cause them to extend below the bottom of the box.

If you resize a textbox in plot edit mode or change the width or height of its `position` property directly, MATLAB sets the object's `FitBoxToText` property to `'off'`. You can toggle this property with `set`, with the Property Inspector, or in plot edit mode via the object's context menu.

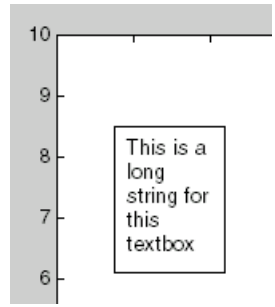
`FitHeightToText`
`on | off`

Automatically adjust text box width and height to fit text. MATLAB automatically wraps text strings to fit the width of the text box. However, if the text string is long enough, it can extend beyond the bottom of the text box.

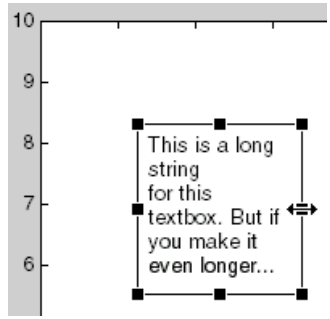
Note The `FitHeightToText` property is obsolete. To control line wrapping behavior in textboxes, use `fitBoxToText` instead.



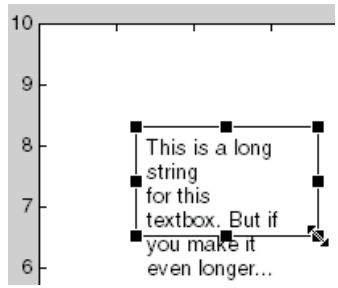
When you set this mode to `on`, MATLAB automatically adjusts the height of the text box to accommodate the string, doing so as you create or edit the string.



The fit-size-to-text behavior turns off if you resize the text box programmatically or manually in plot edit mode.



However, if you resize the text box from any other handles, the position you set is honored without regard to how the text fits the box.



`FontAngle`

`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

`FontName`

A name, such as `Helvetica`

Font family. A string specifying the name of the font to use for the text. To display and print properly, this font must be supported on your system. The default font is `Helvetica`.

`FontSize`
size in points

Approximate size of text characters. A value specifying the font size to use in points. The default size is 10 (1 point = 1/72 inch).

`FontUnits`
`{points} | normalized | inches | centimeters | pixels`

Font size units. MATLAB uses this property to determine the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your system. Generally, setting this property to `bold` or `demi` causes MATLAB to use a bold font.

`HorizontalAlignment`
`{left} | center | right`

Horizontal alignment of text. This property specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the `Position` property. The following picture illustrates the alignment options.

`HorizontalAlignment` viewed with the `VerticalAlignment` set to middle (the default).

See the `Extent` property for related information.

`Interpreter`
latex | {tex} | none

Interpret T_EX instructions. This property controls whether MATLAB interprets certain characters in the `String` property as T_EX instructions (default) or displays all characters literally. The options are:

- `latex` — Supports the full L_AT_EX markup language.
- `tex` — Supports a subset of plain T_EX markup language. See the `String` property for a list of supported T_EX instructions.
- `none` — Displays literal characters.

`LineStyle`
{-} | -- | : | -. | none

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^{1}/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Margin`

dimension in pixels default: 5

Space around text. Specify a value in pixels that defines the space around the text string, but within the rectangle.

`Position`

four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x, y* in units normalized to the figure (when `Units` property is `normalized`). The third and fourth elements specify the object's *dx* and *dy*, respectively, in units normalized to the figure.

`String`

string

The text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

When the text `Interpreter` property is set to `Tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\beta	β	\phi	Φ	\leq	≤
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	♣
\epsilon	ε	\omega	ω	\diamondsuit	♦
\zeta	ζ	\Gamma	Γ	\heartsuit	♥
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	Θ	\Theta	Θ	\leftrightarrow	↔
\vartheta	ϑ	\Lambda	Λ	\leftarrow	→
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	↔
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	μ	\Upsilon	ϒ	\circ	º
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∍	\div	÷
\tau	τ	\cong	≅	\neq	≠
\equiv	≡	\approx	≈	\aleph	
\Im	ℑ	\Re	ℜ	\wp	℘

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\otimes	⊗	\oplus	⊕	\oslash	∅
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in		\o	o
\rfloor	⌋	\lceil	⌈	\nabla	∇
\lfloor	⌊	\cdot	·	\ldots	...
\perp	⊥	\neg	¬	\prime	´
\wedge	∧	\times	x	\0	∅
\rceil	⌉	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	∠	\rangle	∠		

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers:

`Units`

`{normalized} | inches | centimeters | points | pixels`

position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window. Normalized units interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

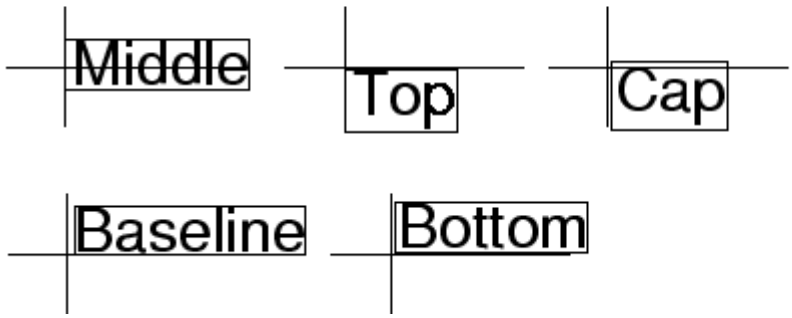
```
VerticalAlignment
     top | cap | {middle} | baseline |
     bottom
```

Vertical alignment of text. This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean

- top — Place the top of the string' s `Extent` rectangle at the specified *y*-position.
- `cap` — Place the string so that the top of a capital letter is at the specified *y*-position.
- `middle` — Place the middle of the string at the specified *y*-position.
- `baseline` — Place font baseline at the specified *y*-position.
- `bottom` — Place the bottom of the string's `Extent` rectangle at the specified *y*-position.

The following picture illustrates the alignment options.

Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



Purpose

Most recent answer

Syntax

```
ans
```

Description

MATLAB creates the ans variable automatically when you specify no output argument.

Examples

The statement

```
2+2
```

is the same as

```
ans = 2+2
```

See Also

display

Purpose

Determine whether any array elements are nonzero

Syntax

```
B = any(A)
B = any(A,dim)
```

Description

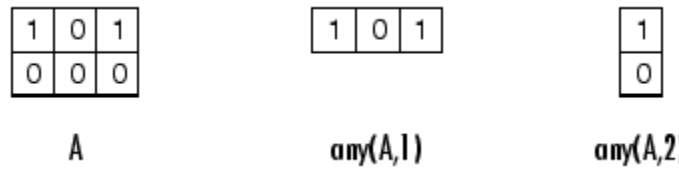
`B = any(A)` tests whether *any* of the elements along various dimensions of an array is a nonzero number or is logical `1` (`true`). `any` ignores entries that are `NaN` (Not a Number).

If `A` is a vector, `any(A)` returns logical `1` (`true`) if any of the elements of `A` is a nonzero number or is logical `1` (`true`), and returns logical `0` (`false`) if all the elements are zero.

If `A` is a matrix, `any(A)` treats the columns of `A` as vectors, returning a row vector of logical 1's and 0's.

If `A` is a multidimensional array, `any(A)` treats the values along the first nonsingleton dimension as vectors, returning a logical condition for each vector.

`B = any(A,dim)` tests along the dimension of `A` specified by scalar *`dim`*.



Examples

Example 1 – Reducing a Logical Vector to a Scalar Condition

Given

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
```

then `B = (A < 0.5)` returns logical `1` (`true`) only where `A` is less than one half:

```
0    0    1    1    1    1    0
```

The any function reduces such a vector of logical conditions to a single condition. In this case, any(B) yields logical 1.

This makes any particularly useful in if statements:

```
if any(A < 0.5)do something
end
```

where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

Example 2– Reducing a Logical Matrix to a Scalar Condition

Applying the any function twice to a matrix, as in any(any(A)), always reduces it to a scalar condition.

```
any(any(eye(3)))
ans =
   1
```

Example 3 – Testing Arrays of Any Dimension

You can use the following type of statement on an array of any dimensions. This example tests a 3-D array to see if any of its elements are greater than 3:

```
x = rand(3,7,5) * 5;

any(x(:) > 3)
ans =
     1
```

or less than zero:

```
any(x(:) < 0)
ans =
     0
```

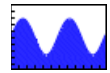
See Also

all, logical operators (elementwise and short-circuit), relational operators, colon

Other functions that collapse an array's dimensions include `max`, `mean`, `median`, `min`, `prod`, `std`, `sum`, and `trapz`.

Purpose

Filled area 2-D plot



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
area(Y)
area(X,Y)
area(...,basevalue)
area(...,'PropertyName',PropertyValue,...)
area(axes_handle,...)
h = area(...)
hpatches = area('v6',...)
```

Description

An area graph displays elements in `Y` as one or more curves and fills the area beneath each curve. When `Y` is a matrix, the curves are stacked showing the relative contribution of each row element to the total height of the curve at each x interval.

`area(Y)` plots the vector `Y` or the sum of each column in matrix `Y`. The *x*-axis automatically scales to `1:size(Y,1)`.

`area(X,Y)` For vectors `X` and `Y`, `area(X,Y)` is the same as `plot(X,Y)` except that the area between `0` and `Y` is filled. When `Y` is a matrix, `area(X,Y)` plots the columns of `Y` as filled areas. For each `X`, the net result is the sum of corresponding values from the columns of `Y`.

If `X` is a vector, `length(X)` must equal `length(Y)`. If `X` is a matrix, `size(X)` must equal `size(Y)`.

`area(...,basevalue)` specifies the base value for the area fill. The default `basevalue` is `0`. See the `BaseValue` property for more information.

`area(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the patch graphics object created by `area`.

`area(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = area(...)` returns handles of `areaseries` graphics objects.

Backward-Compatible Version

`hpatches = area('v6',...)` returns the handles of patch objects instead of areaseries objects for compatibility with MATLAB 6.5 and earlier.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Areaseries Objects

Creating an area graph of an *m*-by-*n* matrix creates *n* areaseries objects (i.e., one per column), whereas a 1-by-*n* vector creates one area object.

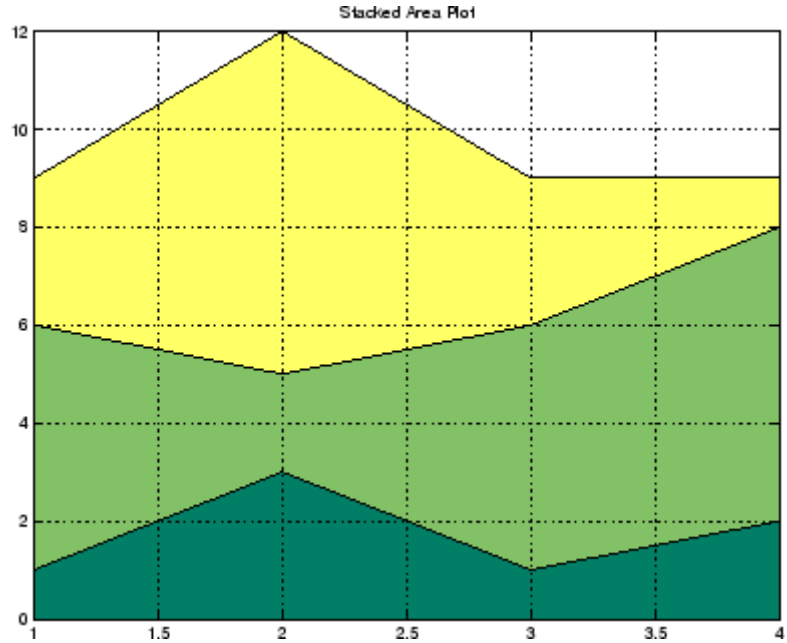
Some areaseries object properties that you set on an individual areaseries object set the values for all areaseries objects in the graph. See the property descriptions for information on specific properties.

Examples

Stacked Area Graph

This example plots the data in the variable `Y` as an area graph. Each subsequent column of `Y` is stacked on top of the previous data. The figure colormap controls the coloring of the individual areas. You can explicitly set the color of an area using the `EdgeColor` and `FaceColor` properties.

```
Y = [1, 5, 3;
 3, 2, 7;
 1, 5, 3;
 2, 6, 1];
area(Y)
grid on
colormap summer
set(gca,'Layer','top')
title 'Stacked Area Plot'
```

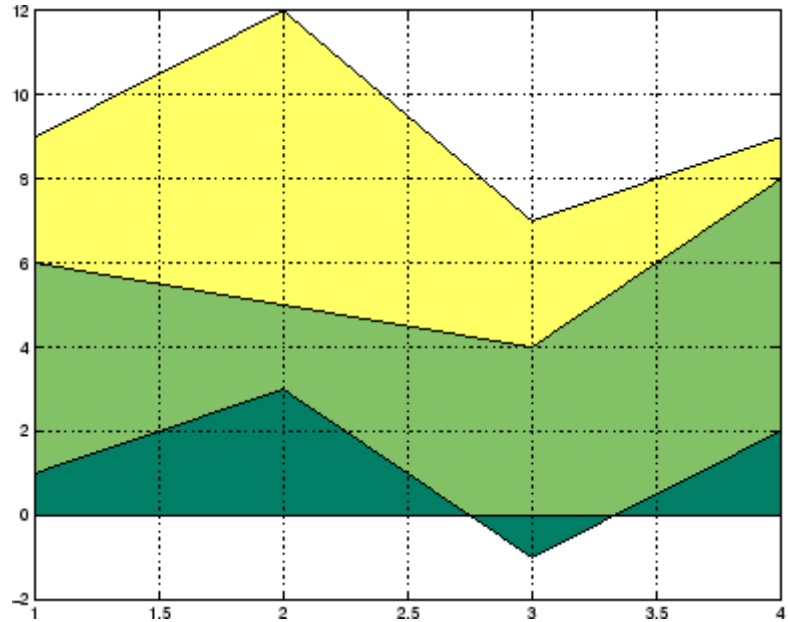


Adjusting the Base Value

The area function uses a *y*-axis value of 0 as the base of the filled areas. You can change this value by setting the area `BaseValue` property. For example, negate one of the values of Y from the previous example and replot the data.

```
Y(3,1) = -1; % Was 1
h = area(Y);
set(gca,'Layer','top')
grid on
colormap summer
```

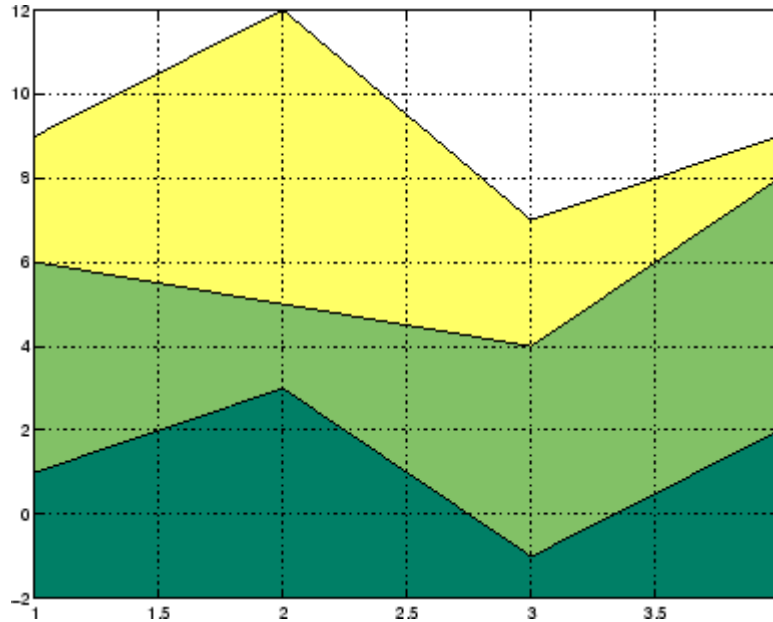
The area graph now looks like this:



Adjusting the `BaseValue` property improves the appearance of the graph:

```
set(h,'BaseValue',-2)
```

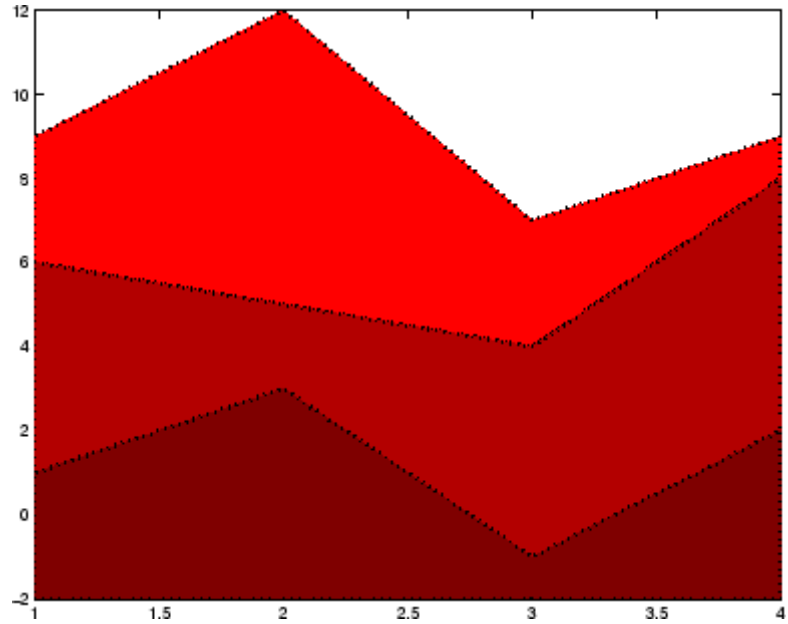
Setting the `BaseValue` property on one areaseries object sets the values of all objects.



Specifying Colors and Line Styles

You can specify the colors of the filled areas and the type of lines used to separate them.

```
h = area(Y,-2); % Set BaseValue via argument
set(h(1),'FaceColor',[.5 0 0])
set(h(2),'FaceColor',[.7 0 0])
set(h(3),'FaceColor',[1 0 0])
set(h,'LineStyle',':','LineWidth',2) % Set
all to same value
```



See Also

`bar`, `plot`, `sort`

"Area, Bar, and Pie Plots" on page 1-88 for related functions

"Area Graphs" for more examples

`Areaseries Properties` for property descriptions

Purpose

Define areaseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

Note that you cannot define default properties for areaseries objects.

See "Plot Objects" for more information on areaseries objects.

Areaseries Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of areaseries objects in legends. The `Annotation` property enables you to specify whether this areaseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the areaseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the areaseries object in a legend as one entry, but not its children objects
`off`	Do not include the areaseries or its children in a legend (default)
`children`	Include only the children of the areaseries as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BaseValue`
double: *y*-axis value

Value where filled area base is drawn. Specify the value along the *y*-axis at which MATLAB draws the baseline of the bottommost filled area.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`

string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

String used by legend for this areaseries object. The `legend` function uses the string defined by the `DisplayName` property to label this areaseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this areaseries object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EdgeColor`

`{[0 0 0]} | none | ColorSpec`

Color of line that separates filled areas. You can set the color of the edges of filled areas to a three-element RGB vector or one of the MATLAB predefined names, including the string `none`. The default edge color is black. See `ColorSpec` for more information on specifying color.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode` `none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceColor`
`{flat} | none | ColorSpec`

Color of filled areas. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that `EdgeColor` is drawn independently of `FaceColor`
- `flat` — The color of the filled areas is determined by the figure colormap. See `colormap` for information on setting the colormap.

  See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions

invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.

- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`

`on | {off}`

Select areaseries object on filled area or extent of graph. This property enables you to select areaseries objects in two ways:

- Select by clicking bars (default).
- Select by clicking anywhere in the extent of the area plot.

  When `HitTestArea` is `off`, you must click the bars to select the bar object. When `HitTestArea` is `on`, you can select the bar object by clicking anywhere within the extent of the bar graph (i.e., anywhere within a rectangle that encloses all the bars).

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For areaseries objects, `Type` is 'hggroup'.

The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`

handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
vector or matrix

The x-axis values for a graph. The *x*-axis values for graphs are specified by the `X` input argument. If `XData` is a vector, `length(XData)` must equal `length(YData)` and must be monotonic. If `XData` is a matrix, `size(XData)` must equal `size(YData)` and each column must be monotonic.

You can use `XData` to define meaningful coordinates for an underlying surface whose topography is being mapped. See "Setting the Axis Limits on Contour Plots" on page 2-640 for more information.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `x` input

argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
vector or matrix

Area plot data. `YData` contains the data plotted as filled areas (the `Y` input argument). If `YData` is a vector, `area` creates a single filled area whose upper boundary is defined by the elements of `YData`. If `YData` is a matrix, `area` creates one filled area per column, stacking each on the previous plot.

The input argument `Y` in the `area` function calling syntax assigns values to `YData`.

`YDataSource`
string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Apply function to each element of array

Syntax

```
A = arrayfun(fun, S)
A = arrayfun(fun, S, T, ...)
[A, B, ...] = arrayfun(fun, S, ...)
[A, ...] = arrayfun(fun, S, ..., 'param1', value1, ...)
```

Description

`A = arrayfun(fun, S)` applies the function specified by `fun` to each element of array `S`, and returns the results in array `A`. The value `A` returned by `arrayfun` is the same size as `S`, and the `(I,J,...)`th element of `A` is equal to `fun(S(I,J,...))`. The first input argument `fun` is a function handle to a function that takes one input argument and returns a scalar value. `fun` must return values of the same class each time it is called.

If `fun` is bound to more than one built-in or M-file (that is, if it represents a set of overloaded functions), then the class of the values that `arrayfun` actually provides as input arguments to `fun` determines which functions are executed.

The order in which `arrayfun` computes elements of `A` is not specified and should not be relied upon.

`A = arrayfun(fun, S, T, ...)` evaluates `fun` using elements of the arrays `S`, `T`, `...` as input arguments. The `(I,J,...)`th element of `A` is equal to `fun(S(I,J,...), T(I,J,...), ...)`. All input arguments must be of the same size.

`[A, B, ...] = arrayfun(fun, S, ...)` evaluates `fun`, which is a function handle to a function that returns multiple outputs, and returns arrays `A`, `B`, `...`, each corresponding to one of the output arguments of `fun`. `arrayfun` calls `fun` each time with as many outputs as there are in the call to `arrayfun`. `fun` can return output arguments having different classes, but the class of each output must be the same each time `fun` is called.

`[A, ...] = arrayfun(fun, S, ..., 'param1', value1, ...)` enables you to specify optional parameter name and value pairs.

Parameters recognized by arrayfun are shown below. Enclose each parameter name with single quotes.

Parameter Name	Parameter Value
UniformOutput	A logical 1 (true) or 0 (false), indicating whether or not the outputs of fun can be returned without encapsulation in a cell array. If true (the default), fun must return scalar values that can be concatenated into an array. These values can also be a cell array. If false, arrayfun returns a cell array (or multiple cell arrays), where the (I,J,...)th cell contains the value fun(S(I,J,...), ...).
ErrorHandler	A function handle, specifying the function that arrayfun is to call if the call to fun fails. If an error handler is not specified, arrayfun rethrows the error from the call to fun.

Remarks

MATLAB provides two functions that are similar to arrayfun; these are structfun and cellfun. With structfun, you can apply a given function to all fields of one or more structures. With cellfun, you apply the function to all cells of one or more cell arrays.

Examples

Example 1 — Operating on a Single Input.

Create a 1-by-15 structure array with fields f1 and f2, each field containing an array of a different size. Make each f1 field be unequal to the f2 field at that same array index:

```
for k=1:15
   s(k).f1 = rand(k+3,k+7) * 10;
   s(k).f2 = rand(k+3,k+7) * 10;
```

```
end
```

Set three `f1` fields to be equal to the `f2` field at that array index:

```
s(3).f2 = s(3).f1;
s(9).f2 = s(9).f1;
s(12).f2 = s(12).f1;
```

Use `arrayfun` to compare the fields at each array index. This compares the array of `s(1).f1` with that of `s(1).f2`, the array of `s(2).f1` with that of `s(2).f2`, and so on through the entire structure array.

The first argument in the call to `arrayfun` is an anonymous function. Anonymous functions return a function handle, which is the required first input to `arrayfun`:

```
z = arrayfun(@(x)isequal(x.f1, x.f2), s)
z =
   0  0  1  0  0  0  0  0  1  0  0  1  0  0  0
```

Example 2 — Operating on Multiple Inputs.

This example performs the same array comparison as in the previous example, except that it compares the some field of more than one structure array rather than different fields of the same structure array. This shows how you can use more than one array input with `arrayfun`.

Make copies of array `s`, created in the last example, to arrays `t` and `u`.

```
t = s;   u = s;
```

Make one element of structure array `t` unequal to the same element of `s`. Do the same with structure array `u`:

```
t(4).f1(12)=0;
u(14).f1(6)=0;
```

Compare field `f1` of the three arrays `s`, `t`, and `u`:

```
z = arrayfun(@(a,b,c)isequal(a.f1, b.f1, c.f1), s, t, u)
z =
```

```
     1   1   1   0   1   1   1   1   1   1   1   1   1   0   1
```

Example 3 — Generating Nonuniform Output.

Generate a 1-by-3 structure array `s` having random matrices in field `f1`:

```
rand('state', 0);
s(1).f1 = rand(7,4) * 10;
s(2).f1 = rand(3,7) * 10;
s(3).f1 = rand(5,5) * 10;
```

Find the maximum for each `f1` vector. Because the output is nonscalar, specify the `UniformOutput` option as `false`:

```
sMax = arrayfun(@(x) max(x.f1), s, 'UniformOutput', false)
sMax =
    [1x4 double]    [1x7 double]    [1x5 double]

sMax{:}
ans =
  9.5013  9.2181  9.3547  8.1317
ans =
  2.7219  9.3181  8.4622  6.7214  8.3812   8.318  7.0947
ans =
  6.8222  8.6001  8.9977  8.1797  8.385
```

Find the mean for each `f1` vector:

```
sMean = arrayfun(@(x) mean(x.f1), s, ...
                 'UniformOutput', false)
sMean =
    [1x4 double]    [1x7 double]    [1x5 double]

sMean{:}
ans =
  6.2628  6.2171  5.4231  3.3144
ans =
  1.6209   7.079  5.7696  4.6665  5.1301  5.7136  4.8099
ans =
```

```
    3.8195  5.8816  6.9128  4.9022  5.9541
```

Example 4 — Assigning to More Than One Output Variable.

The next example uses the `lu` function on the same structure array, returning three outputs from `arrayfun`:

```
[l u p] = arrayfun(@(x)lu(x.f1), s, 'UniformOutput', false)
l =
    [7x4 double]    [3x3 double]    [5x5 double]
u =
    [4x4 double]    [3x7 double]    [5x5 double]
p =
    [7x7 double]    [3x3 double]    [5x5 double]

l{3}
ans =
       1          0          0          0          0
 0.44379          1          0          0          0
 0.79398    0.79936          1          0          0
 0.27799   0.066014   -0.77517          1          0
 0.28353    0.85338    0.29223    0.67036          1

u{3}
ans =
   6.8222     3.7837     8.9977     3.4197     3.0929
        0     6.9209     4.2232     1.3796     7.0124
        0          0    -4.0708   -0.40607    -2.3804
        0          0          0     6.8232     2.1729
        0          0          0          0   -0.35098

p{3}
ans =
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
     1     0     0     0     0
     0     1     0     0     0
```

See Also `structfun`, `cellfun`, `spfun`, `function_handle`, `cell2mat`

Purpose

Set FTP transfer type to ASCII

Syntax

```
ascii(f)
```

Description

`ascii(f)` sets the download and upload FTP mode to ASCII, which converts new lines, where `f` was created using `ftp`. Use this function for text files only, including HTML pages and Rich Text Format (RTF) files.

Examples

Connect to the MathWorks FTP server, and display the FTP object.

```
tmw=ftp('ftp.mathworks.com');
disp(tmw)
FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: binary
```

Note that the FTP object defaults to binary mode.

Use the `ascii` function to set the FTP mode to ASCII, and use the `disp` function to display the FTP object.

```
ascii(tmw)
disp(tmw)
FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: ascii
```

Note that the FTP object is now set to ASCII mode.

See Also

`ftp`, `binary`

Purpose

Inverse secant; result in radians

Syntax

```
Y = asec(X)
```

Description

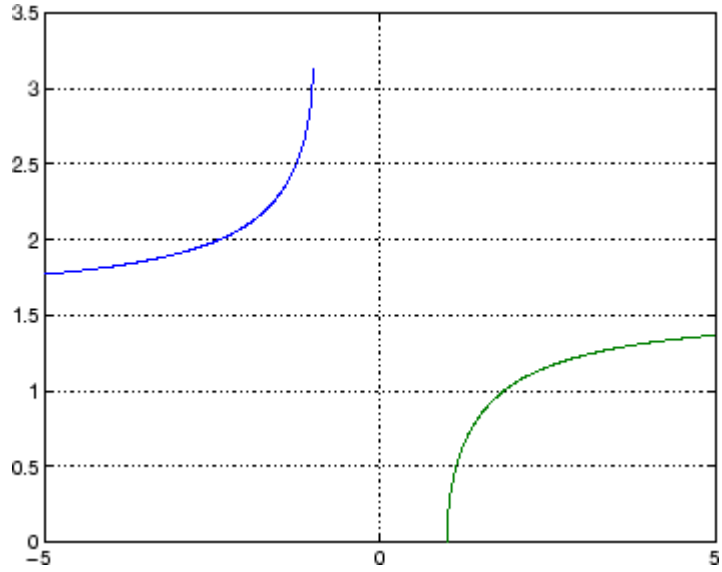
`Y = asec(X)` returns the inverse secant (arcsecant) for each element of `X`.

The `asec` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse secant over the domains $1 \le x \le 5$ and $-5 \le x \le -1$.

```
x1 = -5:0.01:-1;
x2 = 1:0.01:5;
plot(x1,asec(x1),x2,asec(x2)), grid on
```



Definition

The inverse secant can be defined as

$$\sec^{-1}(z) = \cos^{-1}\left(\frac{1}{z}\right)$$

Algorithm

`asec` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`asecd`, `asech`, `sec`

asecd

Purpose Inverse secant; result in degrees

Syntax `Y = asecd(X)`

Description `Y = asecd(X)` is the inverse secant, expressed in degrees, of the elements of `X`.

See Also `secd`, `asec`

Purpose

Inverse hyperbolic secant

Syntax

```
Y = asech(X)
```

Description

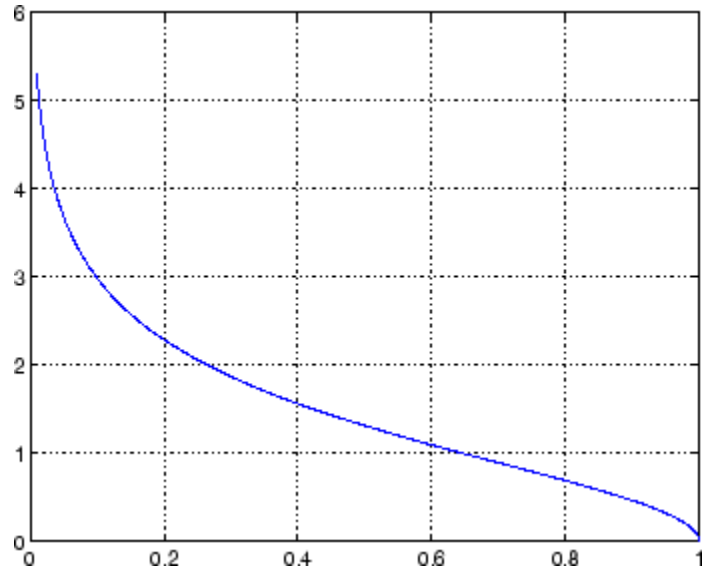
`Y = asech(X)` returns the inverse hyperbolic secant for each element of `X`.

The `asech` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse hyperbolic secant over the domain $0.01 \le x \le 1$.

```
x = 0.01:0.001:1;
plot(x,asech(x)), grid on
```



Definition

The hyperbolic inverse secant can be defined as

$$\operatorname{sech}^{-1}(z) = \cosh^{-1}\left(\frac{1}{z}\right)$$

Algorithm

`asech` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

See Also

`asec`, `sech`

Purpose

Inverse sine; result in radians

Syntax

```
Y = asin(X)
```

Description

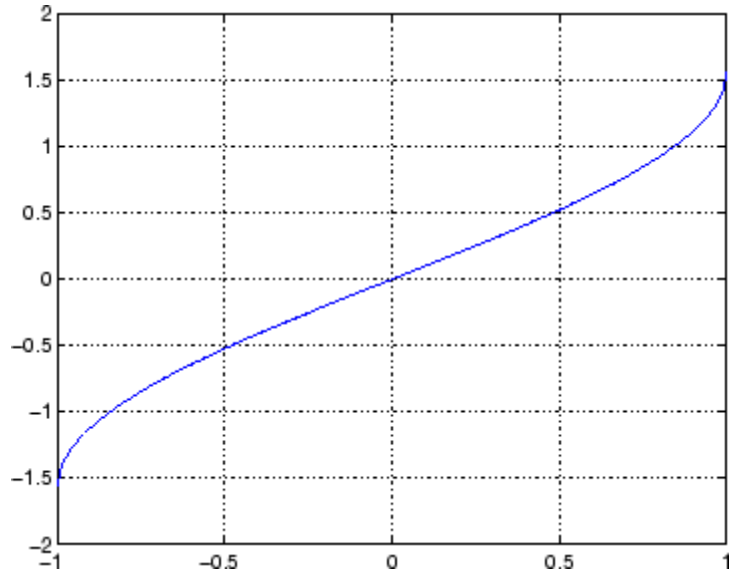
`Y = asin(X)` returns the inverse sine (arcsine) for each element of `X`. For real elements of `X` in the domain $[-1, 1]$, `asin(X)` is in the range $[-\pi/2, \pi/2]$. For real elements of `x` outside the range $[-1, 1]$, `asin(X)` is complex.

The `asin` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse sine function over the domain $-1 \le x \le 1$.

```
x = -1:.01:1;
plot(x,asin(x)), grid on
```



Definition

The inverse sine can be defined as

$$\sin^{-1}(z) = -i \log\left[iz + (1 - z^2)^{\frac{1}{2}} \right]$$

Algorithm

`asin` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`asind`, `asinh`, `sin`, `sind`, `sinh`

Purpose

Inverse sine; result in degrees

Syntax

Description

`Y = asind(X)` is the inverse sine, expressed in degrees, of the elements of `X`.

See Also

`asin`, `asinh`, `sin`, `sind`, `sinh`

Purpose

Inverse hyperbolic sine

Syntax

```
Y = asinh(X)
```

Description

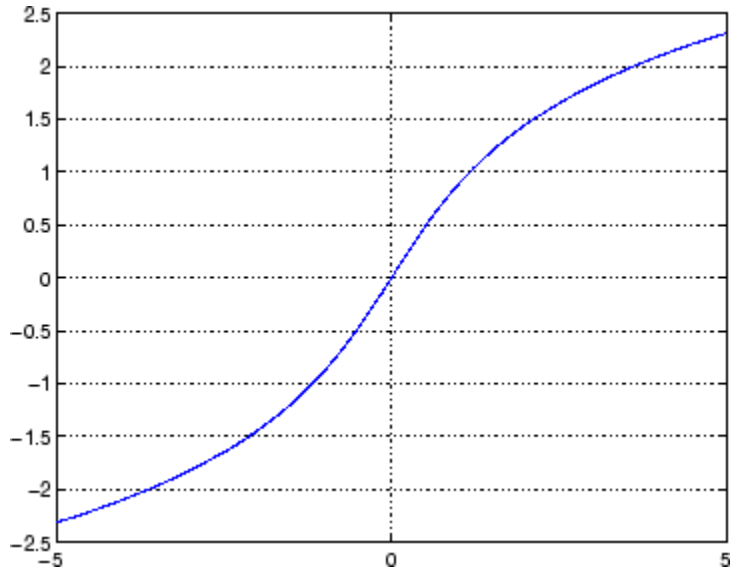
`Y = asinh(X)` returns the inverse hyperbolic sine for each element of `X`.

The `asinh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse hyperbolic sine function over the domain $-5 \le x \le 5$.

```
x = -5:.01:5;
plot(x,asinh(x)), grid on
```



Definition

The hyperbolic inverse sine can be defined as

$$\sinh^{-1}(z) = \log\left[z + (z^2 + 1)^{\frac{1}{2}} \right]$$

Algorithm

`asinh` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`asin`, `asind`, `sin`, `sinh`, `sind`

Purpose

Generate error when condition is violated

Syntax

```
assert(expression)
assert(expression, 'errmsg')
assert(expression, 'errmsg', value1, value2, ...)
assert(expression, 'msg_id', 'errmsg', value1, value2, ...)
```

Description

`assert(expression)` evaluates `expression` and, if it is `false`, displays the error message: `Assertion Failed`.

`assert(expression, 'errmsg')` evaluates `expression` and, if it is `false`, displays the string contained in `errmsg`. This string must be enclosed in single quotation marks. When `errmsg` is the last input to `assert`, MATLAB displays it literally, without performing any substitutions on the characters in `errmsg`.

`assert(expression, 'errmsg', value1, value2, ...)` evaluates `expression` and, if it is `false`, displays the formatted string contained in `errmsg`. The `errmsg` string can include escape sequences such as `\t` or `\n`, as well as any of the C language conversion operators supported by the `sprintf` function (e.g., `%s` or `%d`). Additional arguments `value1`, `value2`, etc. provide values that correspond to and replace the conversion operators.

See “Formatting Strings” in the MATLAB Programming documentation for more detailed information on using string formatting commands.

MATLAB makes substitutions for escape sequences and conversion operators in `errmsg` in the same way that it does for the `sprintf` function.

`assert(expression, 'msg_id', 'errmsg', value1, value2, ...)` evaluates `expression` and, if it is `false`, displays the formatted string `errmsg`, also tagging the error with the message identifier `msg_id`. See in the MATLAB Programming documentation for information.

Examples

This function tests input arguments using `assert`:

```
function write2file(varargin)
```

```
min_inputs = 3;
assert(nargin >= min_inputs, ...
  'You must call function %s with at least %d inputs', ...
  mfilename, min_inputs)

infile = varargin{1};
assert(ischar(infile), ...
      'First argument must be a filename.')
assert(exist(infile)~=0, 'File %s not found.', infile)

fid = fopen(infile, 'w');
assert(fid > 0, 'Cannot open file %s for writing', infile)

fwrite(fid, varargin{2}, varargin{3});
```

See Also

`error`, `eval`, `sprintf`

assignin

Purpose

Assign value to variable in specified workspace

Syntax

```
assignin(ws, 'var', val)
```

Description

`assignin(ws, 'var', val)` assigns the value `val` to the variable *`var`* in the workspace `ws`. *`var`* is created if it doesn't exist. `ws` can have a value of `'base'` or `'caller'` to denote the MATLAB base workspace or the workspace of the caller function.

The `assignin` function is particularly useful for these tasks:

- Exporting data from a function to the MATLAB workspace
- Within a function, changing the value of a variable that is defined in the workspace of the caller function (such as a variable in the function argument list)

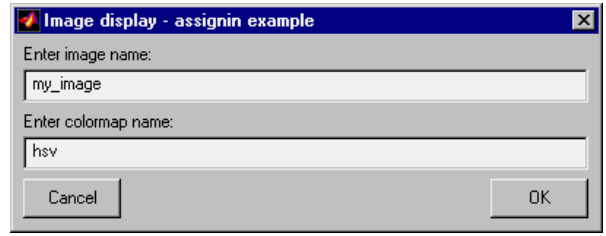
Remarks

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the M-file. Note that the base and caller workspaces are equivalent in the context of an M-file that is invoked from the MATLAB command line.

Examples

This example creates a dialog box for the image display function, prompting a user for an image name and a colormap name. The `assignin` function is used to export the user-entered values to the MATLAB workspace variables `imfile` and `cmap`.

```
prompt = {'Enter image name:','Enter colormap name:'};
title = 'Image display - assignin example';
lines = 1;
def = {'my_image','hsv'};
answer = inputdlg(prompt,title,lines,def);
assignin('base','imfile',answer{1});
assignin('base','cmap',answer{2});
```



See Also

`evalin`

Purpose

Inverse tangent; result in radians

Syntax

```
Y = atan(X)
```

Description

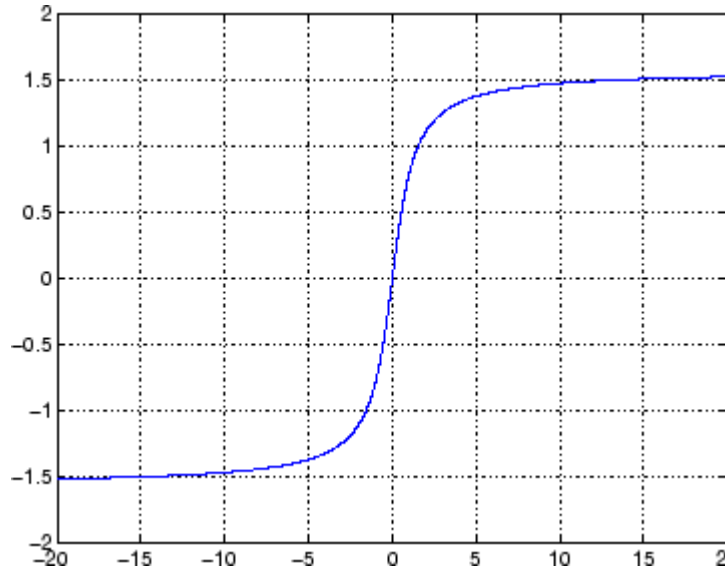
`Y = atan(X)` returns the inverse tangent (arctangent) for each element of `X`. For real elements of `X`, `atan(X)` is in the range $[-\pi/2, \pi/2]$.

The `atan` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the inverse tangent function over the domain $-20 \le x \le 20$.

```
x = -20:0.01:20;
plot(x,atan(x)), grid on
```



Definition

The inverse tangent can be defined as

$$\tan^{-1}(z) = \frac{i}{2}\log\left(\frac{i+z}{i-z}\right)$$

Algorithm

`atan` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`atan2`, `tan`, `atand`, `atanh`

Purpose

Four-quadrant inverse tangent

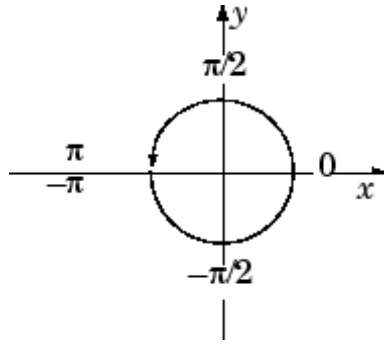
Syntax

```
P = atan2(Y,X)
```

Description

`P = atan2(Y,X)` returns an array `P` the same size as `X` and `Y` containing the element-by-element, four-quadrant inverse tangent (arctangent) of the real parts of `Y` and `X`. Any imaginary parts of the inputs are ignored.

Elements of `P` lie in the closed interval `[-pi,pi]`, where `pi` is the MATLAB floating-point representation of π. `atan` uses `sign(Y)` and `sign(X)` to determine the specific quadrant.



`atan2(Y,X)` contrasts with `atan(Y/X)`, whose results are limited to the interval $[-\pi/2, \pi/2]$, or the right side of this diagram.

Examples

Any complex number $z = x + iy$ is converted to polar coordinates with

```
r = abs(z)
theta = atan2(imag(z),real(z))
```

For example,

```
z = 4 + 3i;
r = abs(z)
theta = atan2(imag(z),real(z))
```

```
r =
     5

theta =
    0.6435
```

This is a common operation, so MATLAB provides a function, `angle(z)`, that computes `theta = atan2(imag(z),real(z))`.

To convert back to the original complex number

```
z = r *exp(i *theta)
z =

   4.0000 + 3.0000i
```

Algorithm

`atan2` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`angle`, `atan`, `atanh`

atand

Purpose

Inverse tangent; result in degrees

Syntax

```
Y = atand(X)
```

Description

`Y = atand(X)` is the inverse tangent, expressed in degrees, of the elements of `X`.

See Also

`tand`, `atan`

Purpose

Inverse hyperbolic tangent

Syntax

```
Y = atanh(X)
```

Description

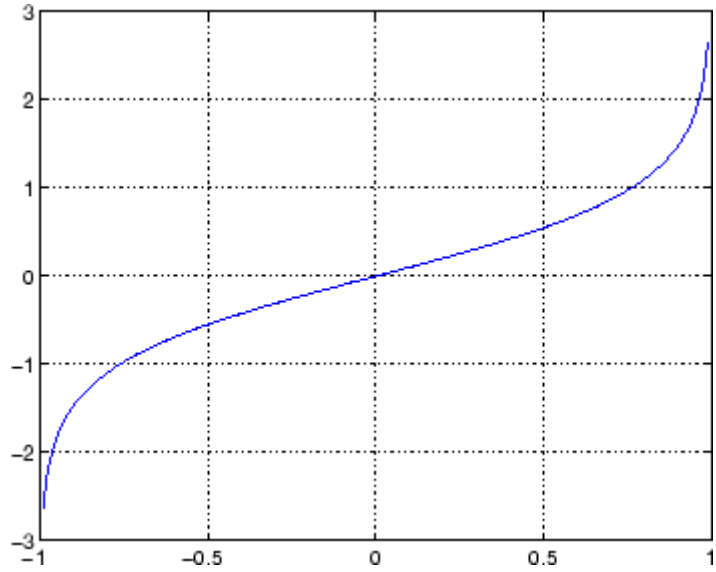
The `atanh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = atanh(X)` returns the inverse hyperbolic tangent for each element of `X`.

Examples

Graph the inverse hyperbolic tangent function over the domain $-1 < x < 1$.

```
x = -0.99:0.01:0.99;
plot(x,atanh(x)), grid on
```



Definition

The hyperbolic inverse tangent can be defined as

$$\tanh^{-1}(z) = \frac{1}{2}\log\left(\frac{1+z}{1-z}\right)$$

Algorithm

`atanh` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`atan2`, `atan`, `tanh`

Purpose

Create audio player object

Syntax

```
player = audioplayer(Y, Fs)
player = audioplayer(Y, Fs, nBits)
player = audioplayer(Y, Fs, nBits, ID)
player = audioplayer(R)
player = audioplayer(R, ID)
```

Description

Note To use all of the features of the audio player object, your system needs a properly installed and configured sound card with 8- and 16-bit I/O, two channels, and support for sampling rates of up to 48 kHz.

`player = audioplayer(Y, Fs)` creates an audio player object for signal `Y`, using sample rate `Fs`. The function returns `player`, a handle to the audio player object. The audio player object supports methods and properties that you can use to control how the audio data is played.

The input signal `Y` can be a vector or two-dimensional array containing `single`, `double`, `int8`, `uint8`, or `int16` MATLAB data types. `Fs` is the sampling rate in Hz to use for playback. Valid values for `Fs` depend on the specific audio hardware installed. Typical values supported by most sound cards are 8000, 11025, 22050, and 44100 Hz.

`player = audioplayer(Y, Fs, nBits)` creates an audio player object and uses `nBits` bits per sample for floating point signal `Y`. Valid values for `nBits` are 8, 16, and 24 on Windows, 8 and 16 on UNIX. The default number of bits per sample for floating point signals is 16.

`player = audioplayer(Y, Fs, nBits, ID)` creates an audio player object using audio device identifier `ID` for output. If `ID` equals -1, the default output device will be used. This option is only available on Windows.

`player = audioplayer(R)` creates an audio player object using audio recorder object `R`.

`player = audioplayer(R, ID)` creates an audio player object from audio recorder object `R` using audio device identifier `ID` for output. This option is only available on Windows.

Remarks

The value range of the input sample depends on the MATLAB data type. The following table lists these ranges.

Data Type	Input Sample Value Range
`int8`	-128 to 127
`uint8`	0 to 255
`int16`	-32768 to 32767
`single`	-1 to 1
`double`	-1 to 1

Example

Load a sample audio file of Handel's Hallelujah Chorus, create an audio player object, and play back only the first three seconds. `y` contains the audio samples and `Fs` is the sampling rate. You can use any of the `audioplayer` functions listed above on the player:

```
load handel;
player = audioplayer(y, Fs);
play(player,[1 (get(player, 'SampleRate')*3)]);
```

To stop the playback, use this command:

```
stop(player); % Equivalent to player.stop
```

Methods

After you create an audio player object, you can use the methods listed below on that object. `player` represents a handle to the audio player object.

Method	Description

`play(player)`
`play(player, start)`
`play(player, [start stop])`
`play(player, range)`
	Starts playback from the beginning and plays to the end of audio player object `player`. Play audio from the sample indicated by `start` to the end, or from the sample indicated by `start` up to the sample indicated by `stop`. The values of `start` and `stop` can also be specified in a two-element vector `range`.

`playblocking(player)`
`playblocking(player, start)`
`playblocking(player, [start stop])`
`playblocking(player, range)`
	Same as play, but does not return control until playback completes.

`stop(player)`	Stops playback.
`pause(player)`	Pauses playback.
`resume(player)`	Restarts playback from where playback was paused.
`isplaying(player)`	Indicates whether playback is in progress. If `0`, playback is not in progress. If `1`, playback is in progress.

`display(player)`
`disp(player)`
`get(player)`
	Displays all property information about audio player `player`.

Properties

Audio player objects have the properties listed below. To set a user-settable property, use this syntax:

```
set(player, 'property1', value,'property2',value,...)
```

To view a read-only property,

```
get(player,'property')  % Displays 'property' setting.
```

Property	Description	Type
`Type`	Name of the object's class.	Read-only
`SampleRate`	Sampling frequency in Hz.	User-settable
`BitsPerSample`	Number of bits per sample.	Read-only
`NumberOfChannels`	Number of channels.	Read-only
`TotalSamples`	Total length, in samples, of the audio data.	Read-only
`Running`	Status of the audio player (`'on'` or `'off'`).	Read-only
`CurrentSample`	Current sample being played by the audio output device (if it is not playing, `CurrentSample` is the next sample to be played with `play` or `resume`).	Read-only
`UserData`	User data of any type.	User-settable
`Tag`	User-specified object label string.	User-settable
For information on using the following four properties, see Creating Timer Callback Functions in the MATLAB documentation. Note that for audio player object callbacks, `eventStruct (event)` is currently empty (`[]`).		

Property	Description	Type
`TimerFcn`	Handle to a user-specified callback function that is executed repeatedly (at `TimerPeriod` intervals) during playback.	User-settable
`TimerPeriod`	Time, in seconds, between `TimerFcn` callbacks.	User-settable
`StartFcn`	Handle to a user-specified callback function that is executed once when playback starts.	User-settable
`StopFcn`	Handle to a user-specified callback function that is executed once when playback stops.	User-settable

See Also

`audiorecorder`, `sound`, `wavplay`, `wavwrite`, `wavread`, `get`, `set`, `methods`

audiorecorder

Purpose

Create audio recorder object

Syntax

```
y = audiorecorder
y = audiorecorder(Fs, nbits, nchans)
y = audiorecorder(Fs, nbits, channels, id)
```

Description

Note To use all of the features of the audiorecorder object, your system must have a properly installed and configured sound card with 8- and 16-bit I/O and support for sampling rates of up to 48 kHz.

`y = audiorecorder` creates an 8000 Hz, 8-bit, 1 channel audiorecorder object. `y` is a handle to the object. The audiorecorder object supports methods and properties that you can use to record audio data.

`y = audiorecorder(Fs, nbits, nchans)` creates an audiorecorder object using the sampling rate `Fs` (in Hz), the sample size `nbits`, and the number of channels`nchans`. `Fs` can be any sampling rate supported by the audio hardware. Common sampling rates are 8000, 11025, 22050, and 44100 (only 44100, 48000, and 96000 on a Macintosh). The value of `nbits` must be `8`, `16`, or `24`, on Windows, and `8` or `16` on UNIX. The number of channels, `nchans` must be `1` (mono) or `2` (stereo).

`y = audiorecorder(Fs, nbits, channels, id)` creates an audiorecorder object using the audio device specified by its `id` for input. If `id` equals -1, the default input device will be used. This option is only available on Windows.

Examples

Example 1

Using a microphone, record your voice, using a sample rate of 44100 Hz, 16 bits per sample, and one channel. Speak into the microphone, then pause the recording. Play back what you've recorded so far. Record some more, then stop the recording. Finally, return the recorded data to MATLAB as an int16 array.

```
r = audiorecorder(44100, 16, 1);
```

```
record(r);     % speak into microphone...
pause(r);
p = play(r);   % listen
resume(r);     % speak again
stop(r);
p = play(r);   % listen to complete recording
mySpeech = getaudiodata(r, 'int16'); % get data as int16 array
```

Remarks

The current implementation of `audiorecorder` is not intended for long, high-sample-rate recording because it uses system memory for storage and does not use disk buffering. When large recordings are attempted, MATLAB performance may degrade.

Methods

After you create an audiorecorder object, you can use the methods listed below on that object. `y` represents the name of the returned audiorecorder object

Method	Description
`record(y)` `record(y,length)`	Starts recording. Records for `length` number of seconds.
`recordblocking(y,length)`	Same as record, but does not return control until recording completes.
`stop(y)`	Stops recording.
`pause(y)`	Pauses recording.
`resume(y)`	Restarts recording from where recording was paused.
`isrecording(y)`	Indicates the status of recording. If `0`, recording is not in progress. If `1`, recording is in progress.
`play(y)`	Creates an `audioplayer`, plays the recorded audio data, and returns a handle to the created `audioplayer`.

Method	Description
`getplayer(y)`	Creates an `audioplayer` and returns a handle to the created `audioplayer`.
`getaudiodata(y)` `getaudiodata(y,'type')`	Returns the recorded audio data to the MATLAB workspace. `type` is a string containing the desired data type. Supported data types are `double`, `single`, `int16`, `int8`, or `uint8`. If `type` is omitted, it defaults to `'double'`. For `double` and `single`, the array contains values between -1 and 1. For `int8`, values are between -128 to 127. For `uint8`, values are from 0 to 255. For `int16`, values are from -32768 to 32767. If the recording is in mono, the returned array has one column. If it is in stereo, the array has two columns, one for each channel.

`display(y)`
`disp(y)`
`get(y)`
	Displays all property information about audio recorder `y`.

Properties

Audio recorder objects have the properties listed below. To set a user-settable property, use this syntax:

```
set(y, 'property1', value,'property2',value,...)
```

To view a read-only property,

```
get(y,'property')  %displays 'property' setting.
```

Property	Description	Type
`Type`	Name of the object's class.	Read-only
`SampleRate`	Sampling frequency in Hz.	Read-only
`BitsPerSample`	Number of bits per recorded sample.	Read-only
`NumberOfChannels`	Number of channels of recorded audio.	Read-only
`TotalSamples`	Total length, in samples, of the recording.	Read-only
`Running`	Status of the audio recorder (`'on'` or `'off'`).	Read-only
`CurrentSample`	Current sample being recorded by the audio output device (if it is not recording, `currentsample` is the next sample to be recorded with `record` or `resume`).	Read-only
`UserData`	User data of any type.	User-settable
For information on using the following four properties, see Creating Timer Callback Functions in the MATLAB documentation. Note that for audio object callbacks, `eventStruct` (`event`) is currently empty (`[]`).		
`TimerFcn`	Handle to a user-specified callback function that is executed repeatedly (at TimerPeriod intervals) during recording.	User-settable
`TimerPeriod`	Time, in seconds, between `TimerFcn` callbacks.	User-settable

Property	Description	Type
`StartFcn`	Handle to a user-specified callback function that is executed once when recording starts.	User-settable
`StopFcn`	Handle to a user-specified callback function that is executed once when recording stops.	User-settable
`NumberOfBuffers`	Number of buffers used for recording (you should adjust this only if you have skips, dropouts, etc., in your recording).	User-settable
`BufferLength`	Length in seconds of buffer (you should adjust this only if you have skips, dropouts, etc., in your recording).	User-settable
`Tag`	User-specified object label string.	User-settable

See Also

`audioplayer`, `wavread`, `wavrecord`, `wavwrite`, `get`, `set`, `methods`

Purpose

Information about NeXT/SUN (`.au`) sound file

Syntax

```
[m d] = aufinfo(aufile)
```

Description

`[m d] = aufinfo(aufile)` returns information about the contents of the AU sound file specified by the string `aufile`.

`m` is the string `'Sound (AU) file'`, if `filename` is an AU file. Otherwise, it contains an empty string (`''`).

`d` is a string that reports the number of samples in the file and the number of channels of audio data. If `filename` is not an AU file, it contains the string `'Not an AU file'`.

See Also

`auread`

auread

Purpose

Read NeXT/SUN (`.au`) sound file

Graphical Interface

As an alternative to `auread`, use the Import Wizard. To activate the Import Wizard, select **Import data** from the **File** menu.

Syntax

```
y = auread('aufile')
[y,Fs,bits] = auread('aufile')
[...] = auread('aufile',N)
[...] = auread('aufile',[N1 N2])
siz = auread('aufile','size')
```

Description

`y = auread('aufile')` loads a sound file specified by the string `aufile`, returning the sampled data in `y`. The `.au` extension is appended if no extension is given. Amplitude values are in the range `[-1,+1]`. `auread` supports multichannel data in the following formats:

- 8-bit mu-law
- 8-, 16-, and 32-bit linear
- Floating-point

`[y,Fs,bits] = auread('aufile')` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.

`[...] = auread('aufile',N)` returns only the first `N` samples from each channel in the file.

`[...] = auread('aufile',[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

`siz = auread('aufile','size')` returns the size of the audio data contained in the file in place of the actual audio data, returning the vector `siz = [samples channels]`.

See Also

`auwrite`, `wavread`

Purpose

Write NeXT/SUN (.au) sound file

Syntax

```
auwrite(y,'aufile')
auwrite(y,Fs,'aufile')
auwrite(y,Fs,N,'aufile')
auwrite(y,Fs,N,'method','aufile')
```

Description

`auwrite(y,'aufile')` writes a sound file specified by the string `aufile`. The data should be arranged with one channel per column. Amplitude values outside the range `[-1,+1]` are clipped prior to writing. `auwrite` supports multichannel data for 8-bit mu-law and 8- and 16-bit linear formats.

`auwrite(y,Fs,'aufile')` specifies the sample rate of the data in Hertz.

`auwrite(y,Fs,N,'aufile')` selects the number of bits in the encoder. Allowable settings are `N = 8` and `N = 16`.

`auwrite(y,Fs,N,'method','aufile')` allows selection of the encoding method, which can be either `mu` or `linear`. Note that mu-law files must be 8-bit. By default, `method = 'mu'`.

See Also

`auread`, `wavwrite`

Purpose

Create new Audio/Video Interleaved (AVI) file

Syntax

```
aviobj = avifile(filename)
aviobj = avifile(filename, 'Param1', Val1, 'Param2', Val2,
   ...)
```

Description

`aviobj = avifile(filename)` creates an `avifile` object, giving it the name specified in `filename`, using default values for all `avifile` object properties. AVI is a file format for storing audio and video data. If `filename` does not include an extension, `avifile` appends `.avi` to the filename. To close all open AVI files, use the `clear mex` command.

`avifile` returns a handle to an AVI file object `aviobj`. You use this object to refer to the AVI file in other functions. An AVI file object supports properties and methods that control aspects of the AVI file created.

`aviobj = avifile(filename, 'Param1', Val1, 'Param2', Val2,...)` creates an `avifile` object with the property values specified by parameter/value pairs. This table lists available parameters.

Parameter	Value	Default
`'colormap'`	An `m`-by-3 matrix defining the colormap to be used for indexed AVI movies, where `m` must be no greater than 256 (236 if using `Indeo` compression). You must set this parameter before calling `addframe`, unless you are using `addframe` with the MATLAB movie syntax.	There is no default colormap.
`'compression'`	A text string specifying the compression codec to use.	

Parameter	Value		Default

	On Windows:
	`'Indeo3'`
	`'Indeo5'`
	`'Cinepak'`
	`'MSVC'`
	`'RLE'`
	`'None'`
		On UNIX:
		`'None'`
			`'Indeo5'` on Windows.
			`'None'` on UNIX.

	To use a custom compression codec, specify the four-character code that identifies the codec (typically included in the codec documentation). The `addframe` function reports an error if it cannot find the specified custom compressor. You must set this parameter before calling `addframe`.		
`'fps'`	A scalar value specifying the speed of the AVI movie in frames per second (fps).		15 fps
`'keyframe'`	For compressors that support temporal compression, this is the number of key frames per second.		2.1429 key frames per second.

Parameter	Value	Default
'quality'	A number between 0 and 100. This parameter has no effect on uncompressed movies. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes. You must set this parameter before calling addframe.	75
'videoname'	A descriptive name for the video stream. This parameter must be no greater than 64 characters long and must be set before using addframe.	The default is the filename.

You can also use structure syntax (also called dot notation) to set avifile object properties. The property name must be typed in full, however it is not case sensitive. For example, to set the quality property to 100, use the following syntax:

```
aviobj = avifile('myavifile');
aviobj.quality = 100;
```

All the field names of an avifile object are the same as the parameter names listed in the table, except for the keyframe parameter. To set this property using dot notation, specify the KeyFramePerSec property. For example, to change the value of keyframe to 2.5, type

```
aviobj.KeyFramePerSec = 2.5;
```

Example

This example shows how to use the avifile function to create the AVI file example.avi.

```
fig=figure;
set(fig,'DoubleBuffer','on');
```

```
set(gca,'xlim',[-80 80],'ylim',[-80 80],...
    'NextPlot','replace','Visible','off')
mov = avifile('example.avi')
x = -pi:.1:pi;
radius = 0:length(x);
for k=1:length(x)
 h = patch(sin(x)*radius(k),cos(x)*radius(k),...
    [abs(cos(x(k))) 0 0]);
 set(h,'EraseMode','xor');
 F = getframe(gca);
 mov = addframe(mov,F);
end
mov = close(mov);
```

See Also

addframe, close, movie2avi

Purpose

Information about Audio/Video Interleaved (AVI) file

Syntax

```
fileinfo = aviinfo(filename)
```

Description

`fileinfo = aviinfo(filename)` returns a structure whose fields contain information about the AVI file specified in the string `filename`. If `filename` does not include an extension, then `.avi` is used. The file must be in the current working directory or in a directory on the MATLAB path.

The set of fields in the `fileinfo` structure is shown below.

Field Name	Description
`AudioFormat`	String containing the name of the format used to store the audio data, if audio data is present
`AudioRate`	Integer indicating the sample rate in Hertz of the audio stream, if audio data is present
`Filename`	String specifying the name of the file
`FileModDate`	String containing the modification date of the file
FileSize	Integer indicating the size of the file in bytes
`FramesPerSecond`	Integer indicating the desired frames per second
`Height`	Integer indicating the height of the AVI movie in pixels
`ImageType`	String indicating the type of image. Either `'truecolor'` for a truecolor (RGB) image, or `'indexed'` for an indexed image.

Field Name	Description
`NumAudioChannels`	Integer indicating the number of channels in the audio stream, if audio data is present
`NumFrames`	Integer indicating the total number of frames in the movie
`NumColormapEntries`	Integer specifying the number of colormap entries. For a truecolor image, this value is 0 (zero).
`Quality`	Number between 0 and 100 indicating the video quality in the AVI file. Higher quality numbers indicate higher video quality; lower quality numbers indicate lower video quality. This value is not always set in AVI files and therefore can be inaccurate.
`VideoCompression`	String containing the compressor used to compress the AVI file. If the compressor is not Microsoft Video 1, Run Length Encoding (RLE), Cinepak, or Intel Indeo, `aviinfo` returns the four-character code that identifies the compressor.
`Width`	Integer indicating the width of the AVI movie in pixels

See also

`avifile`, `aviread`

Purpose

Read Audio/Video Interleaved (AVI) file

Syntax

```
mov = aviread(filename)
mov = aviread(filename, index)
```

Description

`mov = aviread(filename)` reads the AVI movie `filename` into the MATLAB movie structure `mov`. If `filename` does not include an extension, then `.avi` is used. Use the `movie` function to view the movie `mov`. On UNIX, `filename` must be an uncompressed AVI file.

`mov` has two fields, `cdata` and `colormap`. The content of these fields varies depending on the type of image.

Image Type	cdata Field	colormap Field
`Truecolor`	Height-by-width-by-3 array of `uint8` values	Empty
`Indexed`	Height-by-width array of `uint8` values	`m`-by-3 array of `double` values

`aviread` supports 8-bit frames, for indexed and grayscale images, 16-bit grayscale images, or 24-bit truecolor images. Note, however, that `movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale image frames.

`mov = aviread(filename, index)` reads only the frames specified by `index`. `index` can be a single index or an array of indices into the video stream. In AVI files, the first frame has the index value 1, the second frame has the index value 2, and so on.

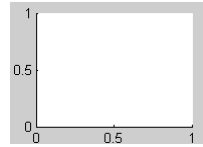
Note If you are using MATLAB on a Windows platform, consider using the new `mmreader` function, which adds support for more video formats and codecs.

See also

`avifile`, `aviinfo`, `mmreader`, `movie`

Purpose

Create axes graphics object



GUI Alternatives

To create a figure select **New > Figure** from the MATLAB Desktop or a figure's **File** menu. To add an axes to a figure, click one of the *New Subplots* icons in the Figure Palette, and slide right to select an arrangement of new axes. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation.

Syntax

```
axes
axes('PropertyName',propertyvalue,...)
axes(h)
h = axes(...)
```

Description

`axes` is the low-level function for creating axes graphics objects.

`axes` creates an axes graphics object in the current figure using default property values.

`axes('PropertyName',propertyvalue,...)` creates an axes object having the specified property values. MATLAB uses default values for any properties that you do not explicitly define as arguments.

`axes(h)` makes existing axes `h` the current axes and brings the figure containing it into focus. It also makes `h` the first axes listed in the figure's `Children` property and sets the figure's `CurrentAxes` property to `h`. The current axes is the target for functions that draw image, line, patch, rectangle, surface, and text graphics objects.

If you want to make an axes the current axes without changing the state of the parent figure, set the `CurrentAxes` property of the figure containing the axes:

```
set(figure_handle,'CurrentAxes',axes_handle)
```

This is useful if you want a figure to remain minimized or stacked below other figures, but want to specify the current axes.

`h = axes(...)` returns the handle of the created axes object.

Remarks

MATLAB automatically creates an axes, if one does not already exist, when you issue a command that creates a graph.

The `axes` function accepts property name/property value pairs, structure arrays, and cell arrays as input arguments (see the `set` and `get` commands for examples of how to specify these data types). These properties, which control various aspects of the axes object, are described in the `Axes Properties` section.

Use the `set` function to modify the properties of an existing axes or the `get` function to query the current values of axes properties. Use the `gca` command to obtain the handle of the current axes.

The `axis` (not `axes`) function provides simplified access to commonly used properties that control the scaling and appearance of axes.

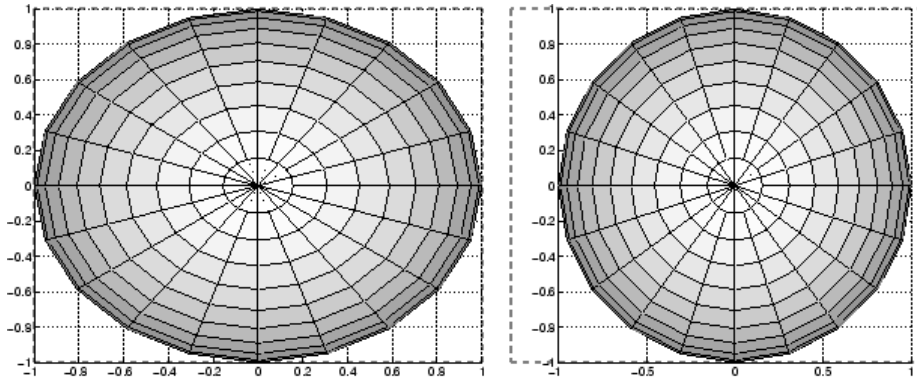
While the basic purpose of an axes object is to provide a coordinate system for plotted data, axes properties provide considerable control over the way MATLAB displays data.

Stretch-to-Fill

By default, MATLAB stretches the axes to fill the axes position rectangle (the rectangle defined by the last two elements in the `Position` property). This results in graphs that use the available space in the rectangle. However, some 3-D graphs (such as a sphere) appear distorted because of this stretching, and are better viewed with a specific three-dimensional aspect ratio.

Stretch-to-fill is active when the `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto` (the default). However, stretch-to-fill is turned off when the `DataAspectRatio`, `PlotBoxAspectRatio`, or `CameraViewAngle` is user-specified, or when one or more of the corresponding modes is set to `manual` (which happens automatically when you set the corresponding property value).

This picture shows the same sphere displayed both with and without the stretch-to-fill. The dotted lines show the axes rectangle.



Stretch-to-fill active | Stretch-to-fill disabled

When stretch-to-fill is disabled, MATLAB sets the size of the axes to be as large as possible within the constraints imposed by the `Position` rectangle without introducing distortion. In the picture above, the height of the rectangle constrains the axes size.

Examples

Zooming

Zoom in using aspect ratio and limits:

```
sphere
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1],'ZLim',[-0.6 0.6])
```

Zoom in and out using the `CameraViewAngle`:

```
sphere
set(gca,'CameraViewAngle',get(gca,'CameraViewAngle')-5)
set(gca,'CameraViewAngle',get(gca,'CameraViewAngle')+5)
```

Note that both examples disable the MATLAB stretch-to-fill behavior.

Positioning the Axes

The axes `Position` property enables you to define the location of the axes within the figure window. For example,

```
h = axes('Position',position_rectangle)
```

creates an axes object at the specified position within the current figure and returns a handle to it. Specify the location and size of the axes with a rectangle defined by a four-element vector,

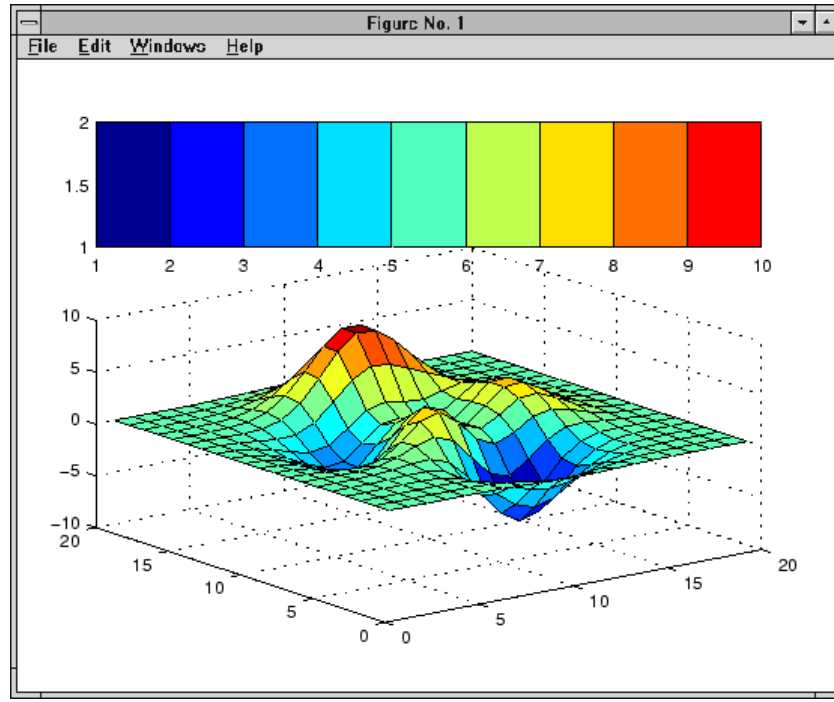
```
position_rectangle = [left, bottom, width, height];
```

The `left` and `bottom` elements of this vector define the distance from the lower left corner of the figure to the lower left corner of the rectangle. The `width` and `height` elements define the dimensions of the rectangle. You specify these values in units determined by the `Units` property. By default, MATLAB uses normalized units where (0,0) is the lower left corner and (1.0,1.0) is the upper right corner of the figure window.

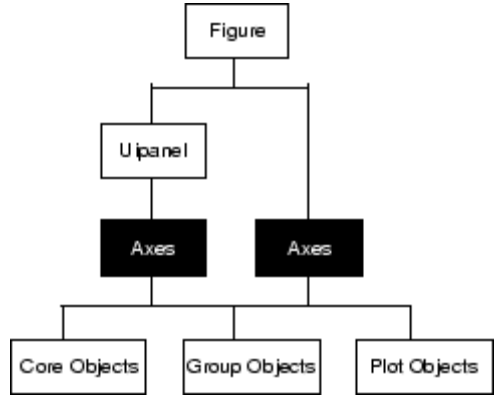
You can define multiple axes in a single figure window:

```
axes('position',[.1  .1  .8  .6])
mesh(peaks(20));
axes('position',[.1  .7  .8  .2])
pcolor([1:10;1:10]);
```

In this example, the first plot occupies the bottom two-thirds of the figure, and the second occupies the top third.



Object Hierarchy



Setting Default Properties

You can set default axes properties on the figure and root levels:

```
set(0,'DefaultAxes<PropertyName>',PropertyValue,...)
set(gcf,'DefaultAxes<PropertyName>',PropertyValue,...)
```

where *PropertyName* is the name of the axes property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access axes properties.

See Also

`axis`, `cla`, `clf`, `figure`, `gca`, `grid`, `subplot`, `title`, `xlabel`, `ylabel`, `zlabel`, `view`

“Axes Operations” on page 1-96 for related functions

“Axes Properties” for more examples

See “Types of Graphics Objects” for information on core, group, plot, and annotation objects.

Purpose

Axes properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see Setting Default Property Values.

Axes Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces { } enclose default values.

`ActivePositionProperty`

`{outerposition} | position`

Use `OuterPosition` or `Position` property for resize. `ActivePositionProperty` specifies which property MATLAB uses to determine the size of the axes when the figure is resized (interactively or during a printing or exporting operation).

See `OuterPosition` and `Position` for related properties.

See Automatic Axes Resize for a discussion of how to use axes positioning properties.

`ALim`

`[amin, amax]`

Alpha axis limits. A two-element vector that determines how MATLAB maps the `AlphaData` values of surface, patch, and image objects to the figure's alphamap. `amin` is the value of the data mapped to the first alpha value in the alphamap, and `amax` is the value of the data mapped to the last alpha value in the alphamap. Data values in between are linearly interpolated

across the alphamap, while data values outside are clamped to either the first or last alphamap value, whichever is closest.

When `ALimMode` is `auto` (the default), MATLAB assigns `amin` the minimum data value and `amax` the maximum data value in the graphics object's `AlphaData`. This maps `AlphaData` elements with minimum data values to the first alphamap entry and those with maximum data values to the last alphamap entry. Data values in between are mapped linearly to the values

If the axes contains multiple graphics objects, MATLAB sets `ALim` to span the range of all objects' `AlphaData` (or `FaceVertexAlphaData` for patch objects).

See the `alpha` function reference page for additional information.

`ALimMode`

`{auto} | manual`

Alpha axis limits mode. In `auto` mode, MATLAB sets the `ALim` property to span the `AlphaData` limits of the graphics objects displayed in the axes. If `ALimMode` is `manual`, MATLAB does not change the value of `ALim` when the `AlphaData` limits of axes children change. Setting the `ALim` property sets `ALimMode` to `manual`.

`AmbientLightColor`

`ColorSpec`

The background light in a scene. Ambient light is a directionless light that shines uniformly on all objects in the axes. However, if there are no visible light objects in the axes, MATLAB does not use `AmbientLightColor`. If there are light objects in the axes, the `AmbientLightColor` is added to the other light sources.

`AspectRatio`

(Obsolete)

This property produces a warning message when queried or changed. It has been superseded by the `DataAspectRatio[Mode]` and `PlotBoxAspectRatio[Mode]` properties.

`BeingDeleted`

`on | {off}`

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

See the `close` and `delete` function reference pages for related information.

`Box`

`on | {off}`

Axes box mode. This property specifies whether to enclose the axes extent in a box for 2-D views or a cube for 3-D views. The default is to not display the box.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback executing, callback invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is

executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed.

If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is within the axes, but not over another graphics object parented to the axes. For 3-D views, the active area is defined by a rectangle that encloses the axes.

See the figure's `SelectionType` property to determine whether modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of axes associated with the button down event and an event structure, which is empty for this property)

See Function Handle Callbacks for information on how to use function handles to define the callback function.

Some Plotting Functions Reset the ButtonDownFcn

Most MATLAB plotting functions clear the axes and reset a number of axes properties, including the `ButtonDownFcn` before

plotting data. If you want to create an interface that enables users to plot data interactively, consider using a control device such as a push button (`uicontrol`), which is not affected by plotting functions. See "Example — Using Function Handles in GUIs" for an example.

If you must use the axes `ButtonDownFcn` to plot data, then you should use low-level functions such as `line` `patch`, and `surface` and manage the process with the figure and axes `NextPlot` properties.

See "High-Level Versus Low-Level" for information on how plotting functions behave.

See "Preparing Figures and Axes for Graphics" for more information.

Camera Properties

See View Control with the Camera Toolbar for information related to the Camera properties

`CameraPosition`
`[x, y, z]` axes coordinates

The location of the camera. This property defines the position from which the camera views the scene. Specify the point in axes coordinates.

If you fix `CameraViewAngle`, you can zoom in and out on the scene by changing the `CameraPosition`, moving the camera closer to the `CameraTarget` to zoom in and farther away from the `CameraTarget` to zoom out. As you change the `CameraPosition`, the amount of perspective also changes, if `Projection` is `perspective`. You can also zoom by changing the `CameraViewAngle`; however, this does not change the amount of perspective in the scene.

`CameraPositionMode`
`{auto} | manual`

Auto or manual `CameraPosition`. When set to `auto`, MATLAB automatically calculates the `CameraPosition` such that the camera lies a fixed distance from the `CameraTarget` along the azimuth and elevation specified by `view`. Setting a value for `CameraPosition` sets this property to manual.

`CameraTarget`
`[x, y, z]` axes coordinates

Camera aiming point. This property specifies the location in the axes that the camera points to. The `CameraTarget` and the `CameraPosition` define the vector (the view axis) along which the camera looks.

`CameraTargetMode`
`{auto} | manual`

Auto or manual `CameraTarget` placement. When this property is `auto`, MATLAB automatically positions the `CameraTarget` at the centroid of the axes plot box. Specifying a value for `CameraTarget` sets this property to `manual`.

`CameraUpVector`
`[x, y, z]` axes coordinates

Camera rotation. This property specifies the rotation of the camera around the viewing axis defined by the `CameraTarget` and the `CameraPosition` properties. Specify `CameraUpVector` as a three-element array containing the *x*, *y*, and *z* components of the vector. For example, `[0 1 0]` specifies the positive *y*-axis as the up direction.

The default `CameraUpVector` is `[0 0 1]`, which defines the positive *z*-axis as the up direction.

`CameraUpVectorMode`
`auto} | manual`

Default or user-specified up vector. When `CameraUpVectorMode` is `auto`, MATLAB uses a value of `[0 0 1]` (positive *z*-direction is up) for 3-D views and `[0 1 0]` (positive *y*-direction is up) for 2-D views. Setting a value for `CameraUpVector` sets this property to `manual`.

`CameraViewAngle`
scalar greater than 0 and less than or equal to 180 (angle in degrees)

The field of view. This property determines the camera field of view. Changing this value affects the size of graphics objects displayed in the axes, but does not affect the degree of perspective distortion. The greater the angle, the larger the field of view, and the smaller objects appear in the scene.

`CameraViewAngleMode`
`{auto} | manual`

Auto or manual `CameraViewAngle`. When in `auto` mode, MATLAB sets `CameraViewAngle` to the minimum angle that captures the entire scene (up to 180°).

The following table summarizes MATLAB automatic camera behavior.

CameraViewAngle	Camera Target	Camera Position	Behavior
auto	auto	auto	CameraTarget is set to plot box centroid, CameraViewAngle is set to capture entire scene, CameraPosition is set along the view axis.
auto	auto	manual	CameraTarget is set to plot box centroid, CameraViewAngle is set to capture entire scene.
auto	manual	auto	CameraViewAngle is set to capture entire scene, CameraPosition is set along the view axis.
auto	manual	manual	CameraViewAngle is set to capture entire scene.
manual	auto	auto	CameraTarget is set to plot box centroid, CameraPosition is set along the view axis.
manual	auto	manual	CameraTarget is set to plot box centroid

CameraViewAngle	Camera Target	Camera Position	Behavior
manual	manual	auto	CameraPosition is set along the view axis.
manual	manual	manual	All camera properties are user-specified.

`Children`

vector of graphics object handles

. A vector containing the handles of all graphics objects rendered within the axes (whether visible or not). The graphics objects that can be children of axes are `image`, `light`, `line`, `patch`, `rectangle`, `surface`, and `text`. You can change the order of the handles and thereby change the stacking of the objects on the display.

The text objects used to label the *x*-, *y*-, and *z*-axes and the title are also children of axes, but their `HandleVisibility` properties are set to `off`. This means their handles do not show up in the axes `Children` property unless you set the Root `ShowHiddenHandles` property to `on`.

When an object's `HandleVisibility` property is set to `off`, it is not listed in its parent's `Children` property. See `HandleVisibility` for more information.

`CLim`

`[cmin, cmax]`

Color axis limits. A two-element vector that determines how MATLAB maps the `CData` values of surface and patch objects to the figure's colormap. `cmin` is the value of the data mapped to the first color in the colormap, and `cmax` is the value of the data mapped to the last color in the colormap. Data values in between are linearly interpolated across the colormap, while data

values outside are clamped to either the first or last colormap color, whichever is closest.

When `CLimMode` is `auto` (the default), MATLAB assigns `cmin` the minimum data value and `cmax` the maximum data value in the graphics object's `CData`. This maps `CData` elements with minimum data value to the first colormap entry and with maximum data value to the last colormap entry.

If the axes contains multiple graphics objects, MATLAB sets `CLim` to span the range of all objects' `CData`.

See the `caxis` function reference page for related information.

`CLimMode`

`{auto} | manual`

Color axis limits mode. In `auto` mode, MATLAB sets the `CLim` property to span the `CData` limits of the graphics objects displayed in the axes. If `CLimMode` is `manual`, MATLAB does not change the value of `CLim` when the `CData` limits of axes children change. Setting the `CLim` property sets this property to `manual`.

`Clipping`

`{on} | off`

This property has no effect on axes.

`Color`

`{none} | ColorSpec`

Color of the axes back planes. Setting this property to `none` means the axes is transparent and the figure color shows through. A `ColorSpec` is a three-element RGB vector or one of the MATLAB predefined names. Note that while the default value is `none`, the `matlabrc.m` file may set the axes `color` to a specific color.

`ColorOrder`

m-by-3 matrix of RGB values

Colors to use for multiline plots. `ColorOrder` is an *m*-by-3 matrix of RGB values that define the colors used by the `plot` and `plot3` functions to color each line plotted. If you do not specify a line color with `plot` and `plot3`, these functions cycle through the `ColorOrder` to obtain the color for each line plotted. To obtain the current `ColorOrder`, which may be set during startup, get the property value:

```
get(gca,'ColorOrder')
```

Note that if the axes `NextPlot` property is set to `replace` (the default), high-level functions like `plot` reset the `ColorOrder` property before determining the colors to use. If you want MATLAB to use a `ColorOrder` that is different from the default, set `NextPlot` to `replacechildren`. You can also specify your own default `ColorOrder`.

`CreateFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during object creation. A callback function that executes when MATLAB creates an axes object. You must define this property as a default value for axes. For example, the statement

```
set(0,'DefaultAxesCreateFcn',@ax_create)
```

defines a default value on the `Root` level that sets axes properties whenever you (or MATLAB) create an axes.

```
function ax_create(src,evnt)
 set(src,'Color','b',...
 'XLim',[1 10],...
 'YLim',[0 100])
end
```

MATLAB executes this function after setting all properties for the axes. Setting the `CreateFcn` property on an existing axes object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the Root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`CurrentPoint`
2-by-3 matrix

Location of last button click, in axes data units. A 2-by-3 matrix containing the coordinates of two points defined by the location of the pointer when the mouse was last clicked. MATLAB returns the coordinates with respect to the requested axes.

Clicking Within the Axes — Orthogonal Projection

The two points lie on the line that is perpendicular to the plane of the screen and passes through the pointer. This is true for both 2-D and 3-D views.

The 3-D coordinates are the points, in the axes coordinate system, where this line intersects the front and back surfaces of the axes volume (which is defined by the axes x, y, and z limits).

The returned matrix is of the form:

$$\begin{bmatrix} x_{front} & y_{front} & z_{front} \\ x_{back} & y_{back} & z_{back} \end{bmatrix}$$

where *front* defines the point nearest to the camera position. Therefore, if cp is the matrix returned by the CurrentPoint property, then the first row,

```
cp(1,:)
```

specifies the point nearest the viewer and the second row,

```
cp(2,:)
```

specifies the point furthest from the viewer.

Clicking Outside the Axes — Orthogonal Projection

When you click outside the axes volume, but within the figure, the values returned are:

- Back point — a point in the plane of the camera target (which is perpendicular to the viewing axis).
- Front point — a point in the camera position plane (which is perpendicular to the viewing axis).

These points lie on a line that passes through the pointer and is perpendicular to the camera target and camera position planes.

Clicking Within the Axes — Perspective Projection

The values of the current point when using perspective project can be different from the same point in orthographic projection because the shape of the axes volume can be different.

Clicking Outside the Axes — Perspective Projection

Clicking outside of the axes volume causes the front point to be returned as the current camera position at all times. Only the back point updates with the coordinates of a point that lies on a line extending from the camera position through the pointer and intersecting the camera target at the point.

Related Information

See Defining Scenes with Camera Graphics for information on the camera properties.

See View Projection Types for information on orthogonal and perspective projections.

`DataAspectRatio`
`[dx dy dz]`

Relative scaling of data units. A three-element vector controlling the relative scaling of data units in the *x*, *y*, and *z* directions. For example, setting this property to `[1 2 1]` causes the length of one unit of data in the *x* direction to be the same length as two units of data in the *y* direction and one unit of data in the *z* direction.

Note that the `DataAspectRatio` property interacts with the `PlotBoxAspectRatio`, `XLimMode`, `YLimMode`, and `ZLimMode` properties to control how MATLAB scales the *x*-, *y*-, and *z*-axis. Setting the `DataAspectRatio` will disable the stretch-to-fill behavior if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`. The following table describes the interaction between properties when `stretch-to-fill` behavior is disabled.

X-, Y-, Z-Limits	DataAspect Ratio	PlotBox AspectRatio	Behavior
`auto`	`auto`	`auto`	Limits chosen to span data range in all dimensions.

X-, Y-, Z-Limits	DataAspect Ratio	PlotBox AspectRatio	Behavior
`auto`	`auto`	`manual`	Limits chosen to span data range in all dimensions. `DataAspectRatio` is modified to achieve the requested `PlotBoxAspectRatio` within the limits selected by MATLAB.
`auto`	`manual`	`auto`	Limits chosen to span data range in all dimensions. `PlotBoxAspectRatio` is modified to achieve the requested `DataAspectRatio` within the limits selected by MATLAB.
`auto`	`manual`	`manual`	Limits chosen to completely fit and center the plot within the requested `PlotBoxAspectRatio` given the requested `DataAspectRatio` (this may produce empty space around 2 of the 3 dimensions).

X-, Y-, Z-Limits	DataAspect Ratio	PlotBox AspectRatio	Behavior
`manual`	`auto`	`auto`	Limits are honored. The `DataAspectRatio` and `PlotBoxAspectRatio` are modified as necessary.
`manual`	`auto`	`manual`	Limits and `PlotBoxAspectRatio` are honored. The `DataAspectRatio` is modified as necessary.
`manual`	`manual`	`auto`	Limits and `DataAspectRatio` are honored. The `PlotBoxAspectRatio` is modified as necessary.
`1 manual` `2 auto`	`manual`	`manual`	The 2 automatic limits are selected to honor the specified aspect ratios and limit. See "Examples."
`2 or 3 manual`	`manual`	`manual`	Limits and `DataAspectRatio` are honored; the `PlotBoxAspectRatio` is ignored.

See “Understanding Axes Aspect Ratio” for more information.

`DataAspectRatioMode`
`{auto} | manual`

User or MATLAB controlled data scaling. This property controls whether the values of the `DataAspectRatio` property are user defined or selected automatically by MATLAB. Setting values for the `DataAspectRatio` property automatically sets this property to `manual`. Changing `DataAspectRatioMode` to `manual` disables the stretch-to-fill behavior if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all auto.

`DeleteFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Delete axes callback function. A callback function that executes when the axes object is deleted (e.g., when you issue a `delete` or `clf` command). MATLAB executes the routine before destroying the object's properties so the callback can query these values.

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the Root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DrawMode`
`{normal} | fast`

Rendering mode. This property controls the way MATLAB renders graphics objects displayed in the axes when the figure `Renderer` property is `painters`.

- `normal` mode draws objects in back to front ordering based on the current view in order to handle hidden surface elimination and object intersections.
- `fast` mode draws objects in the order in which you specify the drawing commands, without considering the relationships of the objects in three dimensions. This results in faster rendering because it requires no sorting of objects according to location in the view, but can produce undesirable results because it bypasses the hidden surface elimination and object intersection handling provided by `normal DrawMode`.

When the figure `Renderer` is `zbuffer`, `DrawMode` is ignored, and hidden surface elimination and object intersection handling are always provided.

`FontAngle`

`{normal} | italic | oblique`

Select italic or normal font. This property selects the character slant for axes text. `normal` specifies a nonitalic font. `italic` and `oblique` specify italic font.

`FontName`

A name such as `Courier` or the string `FixedWidth`

Font family name. The font family name specifying the font to use for axes labels. To display and print properly, `FontName` must be a font that your system supports. Note that the *x*-, *y*-, and *z*-axis labels are not displayed in a new font until you manually reset them (by setting the `XLabel`, `YLabel`, and `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` command). Tick mark labels change immediately.

Specifying a Fixed-Width Font

If you want an axes to use a fixed-width font that looks good in any locale, you should set `FontName` to the string `FixedWidth`:

```
set(axes_handle,'FontName','FixedWidth')
```

This eliminates the need to hardcode the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding (such as in Japan, where multibyte character sets are used). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` (note that this string is case sensitive) and rely on `FixedWidthFontName` to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`.

Note that setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`

Font size specified in `FontUnits`

Font size. An integer specifying the font size to use for axes labels and titles, in units determined by the `FontUnits` property. The default point size is 12. The *x*-, *y*-, and *z*-axis text labels are not displayed in a new font size until you manually reset them (by setting the `XLabel`, `YLabel`, or `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` command). Tick mark labels change immediately.

`FontUnits`

`{points} | normalized | inches | centimeters | pixels`

Units used to interpret the `FontSize` property. When set to `normalized`, MATLAB interprets the value of `FontSize` as a fraction of the height of the axes. For example, a `normalized` `FontSize` of 0.1 sets the text characters to a font whose height is one tenth of the axes' height. The default units (`points`), are equal to 1/72 of an inch.

Note that if you are setting both the `FontSize` and the `FontUnits` in one function call, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`.

`FontWeight`

`{normal} | bold | light | demi`

Select bold or normal font. The character weight for axes text. The *x*-, *y*-, and *z*-axis text labels are not displayed in bold until you manually reset them (by setting the `XLabel`, `YLabel`, and `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` commands). Tick mark labels change immediately.

`GridLineStyle`

`- | --| {:} | -.  | none`

Line style used to draw grid lines. The line style is a string consisting of a character, in quotes, specifying solid lines (-), dashed lines (--), dotted lines(`:`), or dash-dot lines (-.). The default grid line style is dotted. To turn on grid lines, use the `grid` command.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from

command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the Root `ShowHiddenHandles` property to on to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines if the axes can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click

on the axes. If `HitTest` is `off`, clicking the axes selects the object below it (which is usually the figure containing it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an axes callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback routine to interrupt callback routines originating from an axes property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`Layer`
`{bottom} | top`

Draw axis lines below or above graphics objects. This property determines if axis lines and tick marks are drawn on top or below axes children objects for any 2-D view (i.e., when you are looking along the *x*-, *y*-, or *z*-axis). This is useful for placing grid lines and tick marks on top of images.

`LineStyleOrder`
`LineSpec` (default: a solid line '-')

Order of line styles and markers used in a plot. This property specifies which line styles and markers to use and in what order when creating multiple-line plots. For example,

```
set(gca,'LineStyleOrder', '-*|:|o')
```

sets LineStyleOrder to solid line with asterisk marker, dotted line, and hollow circle marker. The default is (-), which specifies a solid line for all data plotted. Alternatively, you can create a cell array of character strings to define the line styles:

```
set(gca,'LineStyleOrder',{'-*',':','o'})
```

MATLAB supports four line styles, which you can specify any number of times in any order. MATLAB cycles through the line styles only after using all colors defined by the ColorOrder property. For example, the first eight lines plotted use the different colors defined by ColorOrder with the first line style. MATLAB then cycles through the colors again, using the second line style specified, and so on.

You can also specify line style and color directly with the plot and plot3 functions or by altering the properties of theline or lineseries objects after creating the graph.

High-Level Functions and LineStyleOrder

Note that, if the axes NextPlot property is set to replace (the default), high-level functions like plot reset the LineStyleOrder property before determining the line style to use. If you want MATLAB to use a LineStyleOrder that is different from the default, set NextPlot to replacechildren.

Specifying a Default LineStyleOrder

You can also specify your own default LineStyleOrder. For example, this statement

```
set(0,'DefaultAxesLineStyleOrder',{'-*',':','o'})
```

creates a default value for the axes LineStyleOrder that is not reset by high-level plotting functions.

`LineWidth`
line width in points

Width of axis lines. This property specifies the width, in points, of the *x*-, *y*-, and *z*-axis lines. The default line width is 0.5 points (1 point = $^1/_{72}$ inch).

`MinorGridLineStyle`
`- | --| {:} | -. | none`

Line style used to draw minor grid lines. The line style is a string consisting of one or more characters, in quotes, specifying solid lines (-), dashed lines (--), dotted lines (:), or dash-dot lines (-.). The default minor grid line style is dotted. To turn on minor grid lines, use the `grid minor` command.

`NextPlot`
`add | {replace} | replacechildren`

Where to draw the next plot. This property determines how high-level plotting functions draw into an existing axes.

- `add` — Use the existing axes to draw graphics objects.
- `replace` — Reset all axes properties except `Position` to their defaults and delete all axes children before displaying graphics (equivalent to `cla reset`).
- `replacechildren` — Remove all child objects, but do not reset axes properties (equivalent to `cla`).

The `newplot` function simplifies the use of the `NextPlot` property and is used by M-file functions that draw graphs using only low-level object creation routines. See the M-file `pcolor.m` for an example. Note that figure graphics objects also have a `NextPlot` property.

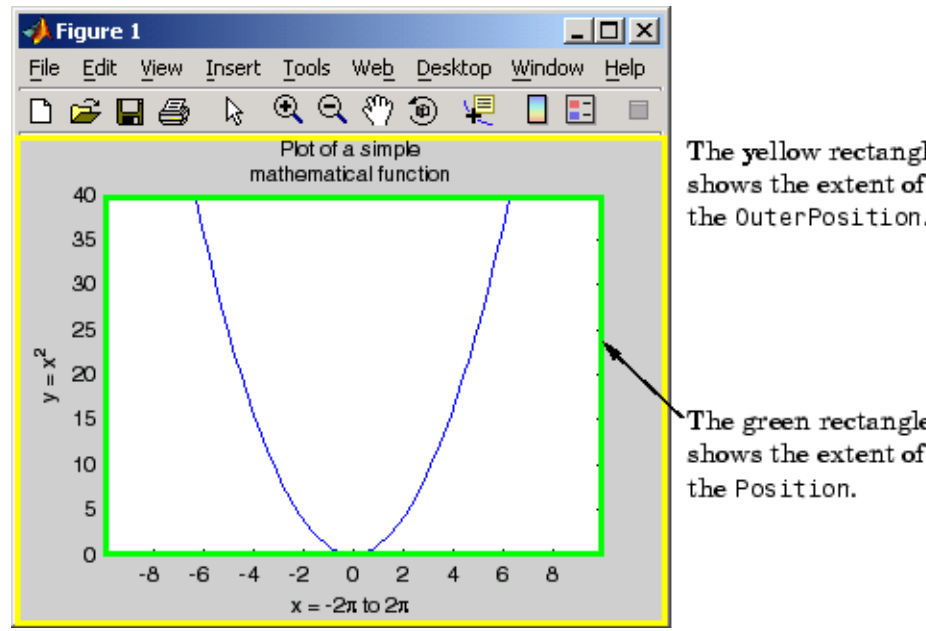
`OuterPosition`
four-element vector

Position of axes including labels, title, and a margin. A four-element vector specifying a rectangle that locates the outer bounds of the axes, including axis labels, the title, and a margin. The vector is defined as follows:

```
[left bottom width height]
```

where `left` and `bottom` define the distance from the lower-left corner of the figure window to the lower-left corner of the rectangle. `width` and `height` are the dimensions of the rectangle

The following picture shows the region defined by the `OuterPosition` enclosed in a yellow rectangle.



When `ActivePositionProperty` is set to `OuterPosition` (the default), none of the text is clipped when you resize the figure.

The default value of `[0 0 1 1]` (normalized units) includes the interior of the figure.

All measurements are in units specified by the `Units` property.

See the `TightInset` property for related information.

See "Automatic Axes Resize" for a discussion of how to use axes positioning properties.

`Parent`
figure or uipanel handle

Axes parent. The handle of the axes' parent object. The parent of an axes object is the figure in which it is displayed or the uipanel object that contains it. The utility function `gcf` returns the handle of the current axes `Parent`. You can reparent axes to other figure or uipanel objects.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`PlotBoxAspectRatio`
`[px py pz]`

Relative scaling of axes plot box. A three-element vector controlling the relative scaling of the plot box in the *x*, *y*, and *z* directions. The plot box is a box enclosing the axes data region as defined by the *x*-, *y*-, and *z*-axis limits.

Note that the `PlotBoxAspectRatio` property interacts with the `DataAspectRatio`, `XLimMode`, `YLimMode`, and `ZLimMode` properties to control the way graphics objects are displayed in the axes. Setting the `PlotBoxAspectRatio` disables stretch-to-fill behavior, if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`.

`PlotBoxAspectRatioMode`
`{auto} | manual`

User or MATLAB controlled axis scaling. This property controls whether the values of the `PlotBoxAspectRatio` property are user defined or selected automatically by MATLAB. Setting values for the `PlotBoxAspectRatio` property automatically sets this property to `manual`. Changing the `PlotBoxAspectRatioMode` to `manual` disables stretch-to-fill behavior if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`.

`Position`
four-element vector

Position of axes. A four-element vector specifying a rectangle that locates the axes within its parent container (figure or uipanel). The vector is of the form

```
[left bottom width height]
```

where `left` and `bottom` define the distance from the lower-left corner of the container to the lower-left corner of the rectangle. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

When axes stretch-to-fill behavior is enabled (when `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`), the axes are stretched to fill the `Position` rectangle. When stretch-to-fill is disabled, the axes are made as large as possible, while obeying all other properties, without extending outside the `Position` rectangle.

See the `OuterPosition` poperty for related information.

See “Automatic Axes Resize” for a discussion of how to use axes positioning properties.

`Projection`
`{orthographic} | perspective`

Type of projection. This property selects between two projection types:

- `orthographic` — This projection maintains the correct relative dimensions of graphics objects with regard to the distance a given point is from the viewer. Parallel lines in the data are drawn parallel on the screen.
- `perspective` — This projection incorporates foreshortening, which allows you to perceive depth in 2-D representations of 3-D objects. Perspective projection does not preserve the relative dimensions of objects; a distant line segment is displayed smaller than a nearer line segment of the same length. Parallel lines in the data may not appear parallel on screen.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that the axes has been selected.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines.

For example, suppose you want to direct all graphics output from an M-file to a particular axes, regardless of user actions that may have changed the current axes. To do this, identify the axes with a `Tag`:

```
axes('Tag','Special Axes')
```

Then make that axes the current axes before drawing by searching for the `Tag` with `findobj`:

```
axes(findobj('Tag','Special Axes'))
```

`TickDir`

```
in | out
```

Direction of tick marks. For 2-D views, the default is to direct tick marks inward from the axis lines; 3-D views direct tick marks outward from the axis line.

`TickDirMode`

```
{auto} | manual
```

Automatic tick direction control. In `auto` mode, MATLAB directs tick marks inward for 2-D views and outward for 3-D views. When you specify a setting for `TickDir`, MATLAB sets `TickDirMode` to `manual`. In `manual` mode, MATLAB does not change the specified tick direction.

`TickLength`

```
[2DLength 3DLength]
```

Length of tick marks. A two-element vector specifying the length of axes tick marks. The first element is the length of tick marks used for 2-D views and the second element is the length of tick marks used for 3-D views. Specify tick mark lengths in units normalized relative to the longest of the visible X-, Y-, or Z-axis annotation lines.

`TightInset`
[left bottom right top] Read only

Margins added to Position to include text labels. The values of this property are the distances between the bounds of the `Position` property and the extent of the axes text labels and title. When added to the `Position` width and height values, the `TightInset` defines the tightest bounding box that encloses the axes and it's labels and title.

See "Automatic Axes Resize" for more information.

`Title`
handle of text object

Axes title. The handle of the `text` object that is used for the axes title. You can use this handle to change the properties of the title text or you can set `Title` to the handle of an existing text object. For example, the following statement changes the color of the current title to red:

```
set(get(gca,'Title'),'Color','r')
```

To create a new title, set this property to the handle of the text object you want to use:

```
set(gca,'Title',text('String','New Title','Color','r'))
```

However, it is generally simpler to use the `title` command to create or replace an axes title:

```
title('New Title','Color','r') % Make text color red
```

```
title({'This title','has 2 lines'}) % Two line title
```

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For axes objects, `Type` is always set to `'axes'`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the axes. Assign this property the handle of a uicontextmenu object created in the axes' parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the axes.

`Units`
`inches | centimeters | {normalized} | points | pixels | characters`

Axes position units. The units used to interpret the `Position` property. All units are measured from the lower left corner of the figure window.

Note The `Units` property controls the positioning of the axes within the figure. This property does not affect the data units used for graphing. See the axes XLim, YLim, and ZLim properties to set the limits of each axis data units.

- `normalized` units map the lower left corner of the figure window to (0,0) and the upper right corner to (1.0, 1.0).
- `inches`, `centimeters`, and `points` are absolute units (one point equals $^1/_{72}$ of an inch).

- `Character` units are defined by characters from the default system font; the width of one character is the width of the letter x, and the height of one character is the distance between the baselines of two lines of text.

When specifying the units as property/value pairs during object creation, you must set the `Units` property before specifying the properties that you want to use these units.

`UserData`
matrix

User-specified data. This property can be any data you want to associate with the axes object. The axes does not use this property, but you can access it using the `set` and `get` functions.

`View`
Obsolete

The functionality provided by the `View` property is now controlled by the axes camera properties — `CameraPosition`, `CameraTarget`, `CameraUpVector`, and `CameraViewAngle`. See the `view` command.

`Visible`
`{on} | off`

Visibility of axes. By default, axes are visible. Setting this property to `off` prevents axis lines, tick marks, and labels from being displayed. The `Visible` property does not affect children of axes.

`XAxisLocation`
`top | {bottom}`

Location of x-axis tick marks and labels. This property controls where MATLAB displays the *x*-axis tick marks and labels. Setting this property to `top` moves the *x*-axis to the top of the plot from its default position at the bottom. This property applies to 2–D views only.

`YAxisLocation`
`right | {left}`

Location of y-axis tick marks and labels. This property controls where MATLAB displays the *y*-axis tick marks and labels. Setting this property to `right` moves the *y*-axis to the right side of the plot from its default position on the left side. This property applies to 2–D views only. See the `plotyy` function for a simple way to use two *y*-axes.

Properties That Control the X-, Y-, or Z-Axis

`XColor`
`YColor`
`ZColor`
`ColorSpec`

Color of axis lines. A three-element vector specifying an RGB triple, or a predefined MATLAB color string. This property determines the color of the axis lines, tick marks, tick mark labels, and the axis grid lines of the respective *x*-, *y*-, and *z*-axis. The default color axis color is black. See`ColorSpec` for details on specifying colors.

`XDir`
`YDir`
`ZDir`
`{normal} | reverse`

Direction of increasing values. A mode controlling the direction of increasing axis values. Axes form a right-hand coordinate system. By default,

- *x*-axis values increase from left to right. To reverse the direction of increasing *x* values, set this property to `reverse`.

```
set(gca,'XDir','reverse')
```

- *y*-axis values increase from bottom to top (2-D view) or front to back (3-D view). To reverse the direction of increasing *y* values, set this property to `reverse`.

  ```
  set(gca,'YDir','reverse')
  ```

- *z*-axis values increase pointing out of the screen (2-D view) or from bottom to top (3-D view). To reverse the direction of increasing *z* values, set this property to `reverse`.

  ```
  set(gca,'ZDir','reverse')
  ```

`XGrid`
`YGrid`
`ZGrid`

`on | {off}`

Axis gridline mode. When you set any of these properties to `on`, MATLAB draws grid lines perpendicular to the respective axis (i.e., along lines of constant *x*, *y*, or *z* values). Use the `grid` command to set all three properties `on` or `off` at once.

```
set(gca,'XGrid','on')
```

`XLabel`
`YLabel`
`ZLabel`

handle of text object

Axis labels. The handle of the `text` object used to label the *x*-, *y*-, or *z*-axis, respectively. To assign values to any of these properties, you must obtain the handle to the text string you want to use as a label. This statement defines a text object and assigns its handle to the `XLabel` property:

```
set(get(gca,'XLabel'),'String','axis label')
```

MATLAB places the string `'axis label'` appropriately for an *x*-axis label. Any text object whose handle you specify as an `XLabel`, `YLabel`, or `ZLabel` property is moved to the appropriate location for the respective label.

Alternatively, you can use the `xlabel`, `ylabel`, and `zlabel` functions, which generally provide a simpler means to label axis lines.

Note that using a bitmapped font (e.g., Courier is usually a bitmapped font) might cause the labels to be rotated improperly. As a workaround, use a TrueType font (e.g., Courier New) for axis labels. See your system documentation to determine the types of fonts installed on your system.

`XLim`
`YLim`
`ZLim`

```
[minimum maximum]
```

Axis limits. A two-element vector specifying the minimum and maximum values of the respective axis. These values are determined by the data you are plotting.

Changing these properties affects the scale of the *x*-, *y*-, or *z*-dimension as well as the placement of labels and tick marks on the axis. The default values for these properties are [0 1].

See the `axis`, `datetick`, `xlim`, `ylim`, and `zlim` commands to set these properties.

`XLimMode`
`YLimMode`
`ZLimMode`

```
{auto} | manual
```

MATLAB or user-controlled limits. The axis limits mode determines whether MATLAB calculates axis limits based on the

data plotted (i.e., the `XData`, `YData`, or `ZData` of the axes children) or uses the values explicitly set with the `XLim`, `YLim`, or `ZLim` property, in which case, the respective limits mode is set to `manual`.

`XMinorGrid`
`YMinorGrid`
`ZMinorGrid`

`on | {off}`

Enable or disable minor gridlines. When set to `on`, MATLAB draws gridlines aligned with the minor tick marks of the respective axis. Note that you do not have to enable minor ticks to display minor grids.

`XMinorTick`
`YMinorTick`
`ZMinorTick`

`on | {off}`

Enable or disable minor tick marks. When set to `on`, MATLAB draws tick marks between the major tick marks of the respective axis. MATLAB automatically determines the number of minor ticks based on the space between the major ticks.

`XScale`
`YScale`
`ZScale`

`{linear} | log`

Axis scaling. Linear or logarithmic scaling for the respective axis. See also `loglog`, `semilogx`, and `semilogy`.

`XTick`
`YTick`
`ZTick`

vector of data values locating tick marks

Tick spacing. A vector of *x*-, *y*-, or *z*-data values that determine the location of tick marks along the respective axis. If you do

not want tick marks displayed, set the respective property to the empty vector, [ ]. These vectors must contain monotonically increasing values.

`XTickLabel`
`YTickLabel`
`ZTickLabel`
string

Tick labels. A matrix of strings to use as labels for tick marks along the respective axis. These labels replace the numeric labels generated by MATLAB. If you do not specify enough text labels for all the tick marks, MATLAB uses all of the labels specified, then reuses the specified labels.

For example, the statement

```
set(gca,'XTickLabel',{'One';'Two';'Three';'Four'})
```

labels the first four tick marks on the *x*-axis and then reuses the labels until all ticks are labeled.

Labels can be specified as cell arrays of strings, padded string matrices, string vectors separated by vertical slash characters, or as numeric vectors (where each number is implicitly converted to the equivalent string using `num2str`). All of the following are equivalent:

```
set(gca,'XTickLabel',{'1';'10';'100'})
set(gca,'XTickLabel','1|10|100')
set(gca,'XTickLabel',[1;10;100])
set(gca,'XTickLabel',['1  ';'10 ';'100'])
```

Note that tick labels do not interpret TeX character sequences (however, the `Title`, `XLabel`, `YLabel`, and `ZLabel` properties do).

`XTickMode`
`YTickMode`
`ZTickMode`
`{auto} | manual`

MATLAB or user-controlled tick spacing. The axis tick modes determine whether MATLAB calculates the tick mark spacing based on the range of data for the respective axis (`auto` mode) or uses the values explicitly set for any of the `XTick`, `YTick`, and `ZTick` properties (`manual` mode). Setting values for the `XTick`, `YTick`, or `ZTick` properties sets the respective axis tick mode to `manual`.

`XTickLabelMode`
`YTickLabelMode`
`ZTickLabelMode`
`{auto} | manual`

MATLAB or user-determined tick labels. The axis tick mark labeling mode determines whether MATLAB uses numeric tick mark labels that span the range of the plotted data (`auto` mode) or uses the tick mark labels specified with the `XTickLabel`, `YTickLabel`, or `ZTickLabel` property (`manual` mode). Setting values for the `XTickLabel`, `YTickLabel`, or `ZTickLabel` property sets the respective axis tick label mode to `manual`.

Purpose

Axis scaling and appearance

Syntax

```
axis([xmin xmax ymin ymax])
axis([xmin xmax ymin ymax zmin zmax cmin cmax])
v = axis
axis auto
axis manual
axis tight
axis fill
axis ij
axis xy
axis equal
axis image
axis square
axis vis3d
axis normal
axis off
axis on
axis(axes_handles,...)
[mode,visibility,direction] = axis('state')
```

Description

`axis` manipulates commonly used axes properties. (See Algorithm section.)

`axis([xmin xmax ymin ymax])` sets the limits for the *x*- and *y*-axis of the current axes.

`axis([xmin xmax ymin ymax zmin zmax cmin cmax])` sets the *x*-, *y*-, and *z*-axis limits and the color scaling limits (see `caxis`) of the current axes.

`v = axis` returns a row vector containing scaling factors for the *x*-, *y*-, and *z*-axis. `v` has four or six components depending on whether the current axes is 2-D or 3-D, respectively. The returned values are the current axes `XLim`, `Ylim`, and `ZLim` properties.

`axis auto` sets MATLAB to its default behavior of computing the current axes limits automatically, based on the minimum and maximum values of *x*, *y*, and *z* data. You can restrict this automatic behavior to

a specific axis. For example, `axis 'auto x'` computes only the *x*-axis limits automatically; `axis 'auto yz'` computes the *y*- and *z*-axis limits automatically.

`axis manual` and axis(axis) freezes the scaling at the current limits, so that if `hold` is `on`, subsequent plots use the same limits. This sets the `XLimMode`, `YLimMode`, and `ZLimMode` properties to `manual`.

`axis tight` sets the axis limits to the range of the data.

`axis fill` sets the axis limits and `PlotBoxAspectRatio` so that the axes fill the position rectangle. This option has an effect only if `PlotBoxAspectRatioMode` or `DataAspectRatioMode` is `manual`.

`axis ij` places the coordinate system origin in the upper left corner. The *i*-axis is vertical, with values increasing from top to bottom. The *j*-axis is horizontal with values increasing from left to right.

`axis xy` draws the graph in the default Cartesian axes format with the coordinate system origin in the lower left corner. The *x*-axis is horizontal with values increasing from left to right. The *y*-axis is vertical with values increasing from bottom to top.

`axis equal` sets the aspect ratio so that the data units are the same in every direction. The aspect ratio of the *x*-, *y*-, and *z*-axis is adjusted automatically according to the range of data units in the *x*, *y*, and *z* directions.

`axis image` is the same as `axis equal` except that the plot box fits tightly around the data.

`axis square` makes the current axes region square (or cubed when three-dimensional). MATLAB adjusts the *x*-axis, *y*-axis, and *z*-axis so that they have equal lengths and adjusts the increments between data units accordingly.

`axis vis3d` freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretch-to-fill.

`axis normal` automatically adjusts the aspect ratio of the axes and the relative scaling of the data units so that the plot fits the figure's shape as well as possible.

`axis off` turns off all axis lines, tick marks, and labels.

`axis on` turns on all axis lines, tick marks, and labels.

`axis(axes_handles,...)` applies the `axis` command to the specified axes. For example, the following statements

```
h1 = subplot(221);
h2 = subplot(222);
axis([h1 h2],'square')
```

set both axes to `square`.

`[mode,visibility,direction] = axis('state')` returns three strings indicating the current setting of axes properties:

Output Argument	Strings Returned
`mode`	`'auto' | 'manual'`
`visibility`	`'on' | 'off'`
`direction`	`'xy' | 'ij'`

`mode` is `auto` if `XLimMode`, `YLimMode`, and `ZLimMode` are all set to `auto`. If `XLimMode`, `YLimMode`, or `ZLimMode` is `manual`, `mode` is `manual`.

Keywords to `axis` can be combined, separated by a space (e.g., `axis tight equal`). These are evaluated from left to right, so subsequent keywords can overwrite properties set by prior ones.

Remarks

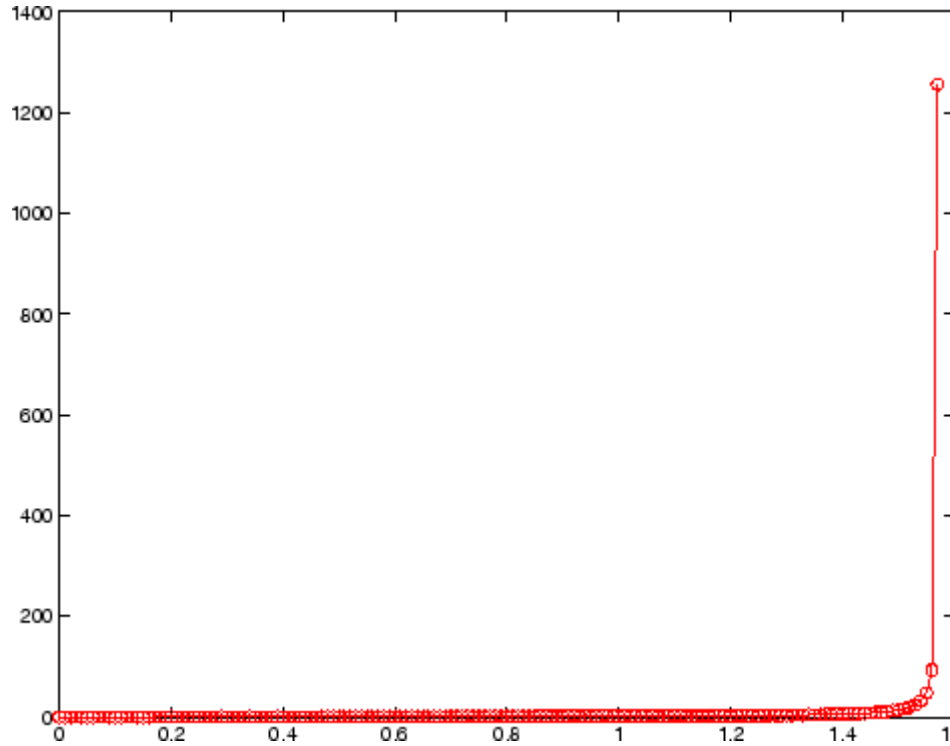
You can create an axes (and a figure for it) if none exists with the `axis` command. However, if you specify non-default limits or formatting for the axes when doing this, such as `[4 8 2 9]`, `square`, `equal`, or `image`, the property is ignored because there are no axis limits to adjust in the absence of plotted data. To use `axis` in this manner, you can set `hold on` to keep preset axes limits from being overridden.

Examples

The statements

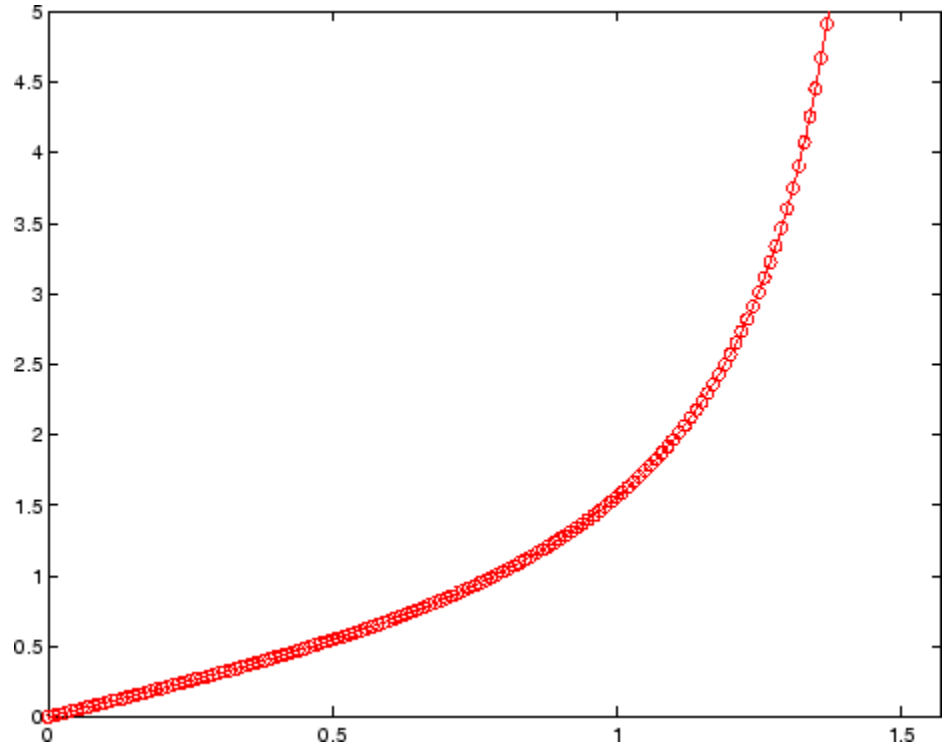
```
x = 0:.025:pi/2;
plot(x,tan(x),'-ro')
```

use the automatic scaling of the *y*-axis based on `ymax = tan(1.57)`, which is well over 1000:



The right figure shows a more satisfactory plot after typing

```
axis([0  pi/2  0  5])
```



Algorithm

When you specify minimum and maximum values for the *x*-, *y*-, and *z*-axes, `axis` sets the `XLim`, `Ylim`, and `ZLim` properties for the current axes to the respective minimum and maximum values in the argument list. Additionally, the `XLimMode`, `YLimMode`, and `ZLimMode` properties for the current axes are set to `manual`.

`axis auto` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to `'auto'`.

`axis manual` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to `'manual'`.

The following table shows the values of the axes properties set by `axis equal`, `axis normal`, `axis square`, and `axis image`.

Axes Property or Behavior	axis equal	axis normal	axis square	axis image
`DataAspectRatio` property	`[1 1 1]`	`not set`	`not set`	`[1 1 1]`
`DataAspectRatioMode` property	`manual`	`auto`	`auto`	`manual`
`PlotBoxAspectRatio` property	`[3 4 4]`	`not set`	`[1 1 1]`	`auto`
`PlotBoxAspectRatioMode` property	`manual`	`auto`	`manual`	`auto`
Stretch-to-fill behavior;	`disabled`	`active`	`disabled`	`disabled`

See Also

`axes`, `grid`, `subplot`, `xlim`, `ylim`, `zlim`

Properties of axes graphics objects

"Axes Operations" on page 1-96 for related functions

For aspect ratio behavior, see in the axes properties reference page.

Purpose

Diagonal scaling to improve eigenvalue accuracy

Syntax

```
[T,B] = balance(A)
[S,P,B] = balance(A)
B = balance(A)
B = balance(A,'noperm')
```

Description

`[T,B] = balance(A)` returns a similarity transformation `T` such that `B = T\A*T`, and `B` has, as nearly as possible, approximately equal row and column norms. `T` is a permutation of a diagonal matrix whose elements are integer powers of two to prevent the introduction of roundoff error. If `A` is symmetric, then `B == A` and `T` is the identity matrix.

`[S,P,B] = balance(A)` returns the scaling vector `S` and the permutation vector `P` separately. The transformation `T` and balanced matrix `B` are obtained from `A`, `S`, and `P` by `T(:,P) = diag(S)` and `B(P,P) = diag(1./S)*A*diag(S)`.

`B = balance(A)` returns just the balanced matrix `B`.

`B = balance(A,'noperm')` scales `A` without permuting its rows and columns.

Remarks

Nonsymmetric matrices can have poorly conditioned eigenvalues. Small perturbations in the matrix, such as roundoff errors, can lead to large perturbations in the eigenvalues. The condition number of the eigenvector matrix,

```
cond(V) = norm(V)*norm(inv(V))
```

where

```
[V,T] = eig(A)
```

relates the size of the matrix perturbation to the size of the eigenvalue perturbation. Note that the condition number of `A` itself is irrelevant to the eigenvalue problem.

Balancing is an attempt to concentrate any ill conditioning of the eigenvector matrix into a diagonal scaling. Balancing usually cannot turn a nonsymmetric matrix into a symmetric matrix; it only attempts to make the norm of each row equal to the norm of the corresponding column.

Note The MATLAB eigenvalue function, `eig(A)`, automatically balances `A` before computing its eigenvalues. Turn off the balancing with `eig(A,'nobalance')`.

Examples

This example shows the basic idea. The matrix `A` has large elements in the upper right and small elements in the lower left. It is far from being symmetric.

```
A = [1  100  10000; .01  1  100; .0001  .01  1]
A =
   1.0e+04 *
    0.0001    0.0100    1.0000
    0.0000    0.0001    0.0100
    0.0000    0.0000    0.0001
```

Balancing produces a diagonal matrix `T` with elements that are powers of two and a balanced matrix `B` that is closer to symmetric than `A`.

```
[T,B] = balance(A)
T =
   1.0e+03 *
    2.0480         0         0
         0    0.0320         0
         0         0    0.0003
B =
    1.0000    1.5625    1.2207
    0.6400    1.0000    0.7813
    0.8192    1.2800    1.0000
```

To see the effect on eigenvectors, first compute the eigenvectors of A, shown here as the columns of V.

```
[V,E] = eig(A); V
V =
   -1.0000    0.9999    0.9937
    0.0050    0.0100   -0.1120
    0.0000    0.0001    0.0010
```

Note that all three vectors have the first component the largest. This indicates V is badly conditioned; in fact cond(V) is 8.7766e+003. Next, look at the eigenvectors of B.

```
[V,E] = eig(B); V
V =
   -0.8873    0.6933    0.0898
    0.2839    0.4437   -0.6482
    0.3634    0.5679   -0.7561
```

Now the eigenvectors are well behaved and cond(V) is 1.4421. The ill conditioning is concentrated in the scaling matrix; cond(T) is 8192.

This example is small and not really badly scaled, so the computed eigenvalues of A and B agree within roundoff error; balancing has little effect on the computed results.

Algorithm

Inputs of Type Double

For inputs of type double, balance uses the linear algebra package (LAPACK) routines DGEBAL (real) and ZGEBAL (complex). If you request the output T, balance also uses the LAPACK routines DGEBAK (real) and ZGEBAK (complex).

Inputs of Type Single

For inputs of type single, balance uses the LAPACK routines SGEBAL (real) and CGEBAL (complex). If you request the output T, balance also uses the LAPACK routines SGEBAK (real) and CGEBAK (complex).

Limitations

Balancing can destroy the properties of certain matrices; use it with some care. If a matrix contains small elements that are due to roundoff error, balancing might scale them up to make them as significant as the other elements of the original matrix.

See Also

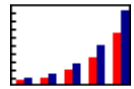
`eig`

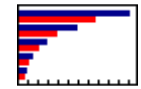
References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Plot bar graph (vertical and horizontal)





GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation and "Creating Plots from the Workspace Browser" in the MATLAB Desktop Tools documentation.

Syntax

```
bar(Y)
bar(x,Y)
bar(...,width)
bar(...,'style')
bar(...,'bar_color')
bar(axes_handle,...)
barh(axes_handle,...)
h = bar(...)
barh(...)
h = barh(...)
hpatches = bar('v6',...)
hpatches = barh('v6',...)
```

Description

A bar graph displays the values in a vector or matrix as horizontal or vertical bars.

`bar(Y)` draws one bar for each element in `Y`. If `Y` is a matrix, `bar` groups the bars produced by the elements in each row. The *x*-axis scale ranges from 1 up to `length(Y)` when `Y` is a vector, and `1` to `size(Y,1)`, which is the number of rows, when `Y` is a matrix. The default is to scale the *x*-axis to the highest x-tick on the plot, (a multiple of 10, 100, etc.). If you want the *x*-axis scale to end exactly at the last bar, you can use the default, and then, for example, type

```
set(gca,'xlim',[1 length(Y)])
```

at the MATLAB prompt.

`bar(x,Y)` draws a bar for each element in `Y` at locations specified in `x`, where `x` is a vector defining the *x*-axis intervals for the vertical bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `bar` groups the elements of each row in `Y` at corresponding locations in `x`.

`bar(...,width)` sets the relative bar width and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, the bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar(...,'style')` specifies the style of the bars. `'style'` is `'grouped'` or `'stacked'`. `'group'` is the default mode of display.

- `'grouped'` displays *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar(...,'bar_color')` displays all bars using the color specified by the single-letter abbreviation `'r'`, `'g'`, `'b'`, `'c'`, `'m'`, `'y'`, `'k'`, or `'w'`.

`bar(axes_handle,...)` and `barh(axes_handle,...)` plot into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = bar(...)` returns a vector of handles to barseries graphics objects, one for each created. When `Y` is a matrix, `bar` creates one barseries graphics object per column in `Y`.

`barh(...)` and `h = barh(...)` create horizontal bars. `Y` determines the bar length. The vector `x` is a vector defining the *y*-axis intervals for horizontal bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values.

Backward-Compatible Versions

`hpatches = bar('v6',...)` and `hpatches = barh('v6',...)` return the handles of patch objects instead of barseries objects for compatibility with MATLAB 6.5 and earlier. See patch object properties for a discussion of the properties you can set to control the appearance of these bar graphs.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See

Plot Objects and Backward Compatibility for more information.

Barseries Objects

Creating a bar graph of an *m*-by-*n* matrix creates *m* groups of *n* barseries objects. Each barseries object contains the data for corresponding `x` values of each bar group (as indicated by the coloring of the bars).

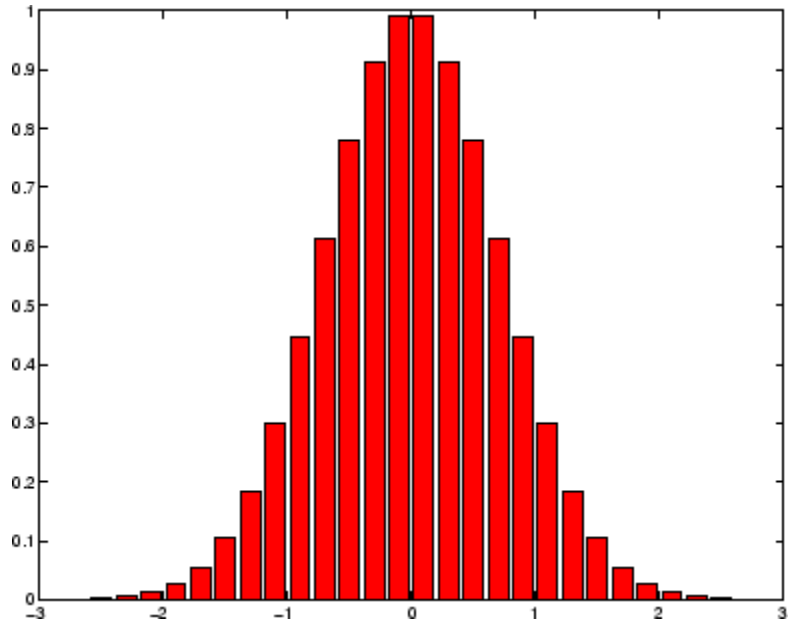
Note that some barseries object properties set on an individual barseries object set the values for all barseries objects in the graph. See the barseries property descriptions for information on specific properties.

Examples

Single Series of Data

This example plots a bell-shaped curve as a bar graph and sets the colors of the bars to red.

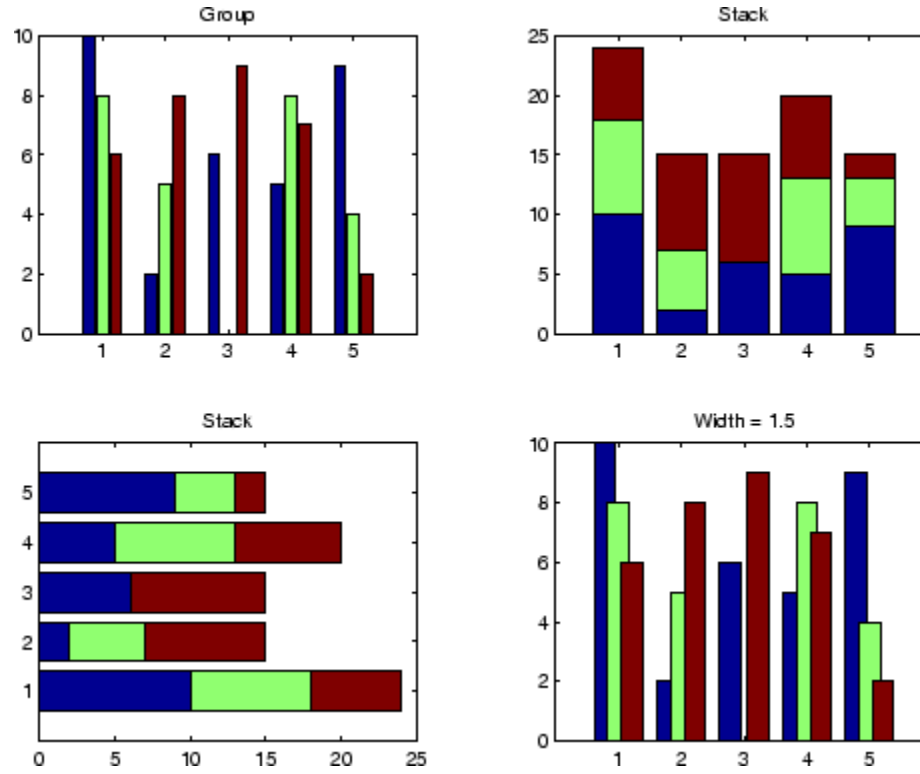
```
x = -2.9:0.2:2.9;
bar(x,exp(-x.*x),'r')
```



Bar Graph Options

This example illustrates some bar graph options.

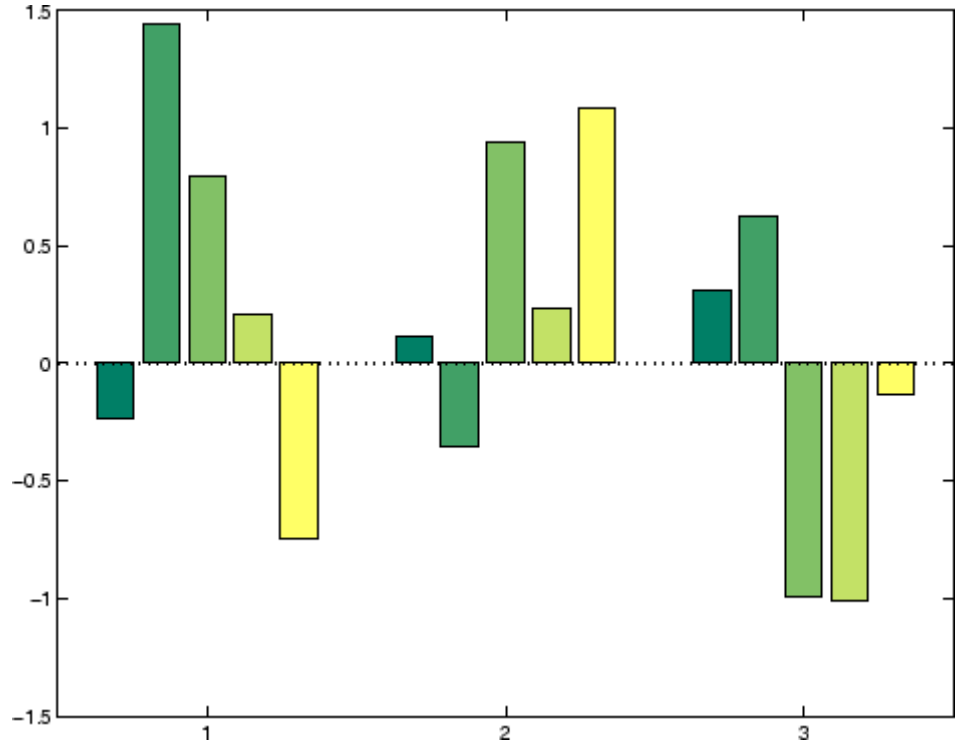
```
Y = round(rand(5,3)*10);
subplot(2,2,1)
bar(Y,'group')
title 'Group'
subplot(2,2,2)
bar(Y,'stack')
title 'Stack'
subplot(2,2,3)
barh(Y,'stack')
title 'Stack'
subplot(2,2,4)
bar(Y,1.5)
title 'Width = 1.5'
```



Setting Properties with Multiobject Graphs

This example creates a graph that displays three groups of bars and contains five barseries objects. Since all barseries objects in a graph share the same baseline, you can set values using any barseries object's `BaseLine` property. This example uses the first handle returned in `h`.

```
Y = randn(3,5);
h = bar(Y);
set(get(h(1),'BaseLine'),'LineWidth',2,'LineStyle',':')
colormap summer % Change the color scheme
```



See Also

`bar3`, `ColorSpec`, `patch`, `stairs`, `hist`

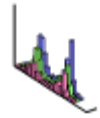
“Area, Bar, and Pie Plots” on page 1-88 for related functions

`Barseries Properties`

“Bar and Area Graphs” for more examples

Purpose

Plot 3-D bar chart



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation and "Creating Graphics from the Workspace Browser" in the MATLAB Desktop Tools documentation.

Syntax

```
bar3(Y)
bar3(x,Y)
bar3(...,width)
bar3(...,'style')
bar3(...,LineSpec)
bar3(axes_handle,...)
h = bar3(...)
bar3h(...)
h = bar3h(...)
```

Description

`bar3` and `bar3h` draw three-dimensional vertical and horizontal bar charts.

`bar3(Y)` draws a three-dimensional bar chart, where each element in `Y` corresponds to one bar. When `Y` is a vector, the *x*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *x*-axis scale ranges from `1` to `size(Y,2)`, which is the number of columns, and the elements in each row are grouped together.

`bar3(x,Y)` draws a bar chart of the elements in `Y` at the locations specified in `x`, where `x` is a vector defining the *y*-axis intervals for vertical bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `bar3` clusters elements from the

same row in `Y` at locations corresponding to an element in `x`. Values of elements in each row are grouped together.

`bar3(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar3(...,'style')` specifies the style of the bars. `'style'` is `'detached'`, `'grouped'`, or `'stacked'`. `'detached'` is the default mode of display.

- `'detached'` displays the elements of each row in `Y` as separate blocks behind one another in the *x* direction.
- `'grouped'` displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar3(...,LineSpec)` displays all bars using the color specified by `LineSpec`.

`bar3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

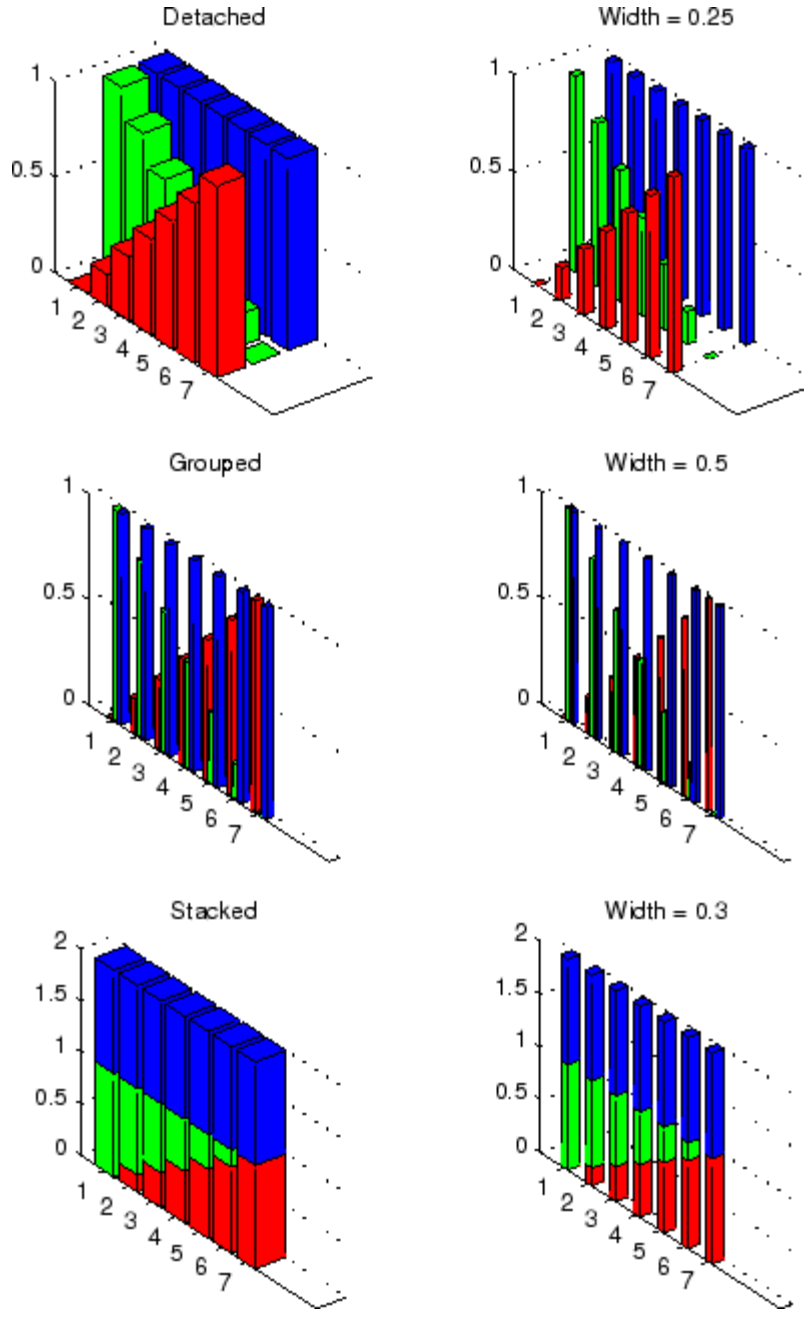
`h = bar3(...)` returns a vector of handles to patch graphics objects, one for each created. `bar3` creates one patch object per column in `Y`. When `Y` is a matrix, `bar3` creates one patch graphics object per column in `Y`.

`bar3h(...)` and `h = bar3h(...)` create horizontal bars. `Y` determines the bar length. The vector `x` is a vector defining the *y*-axis intervals for horizontal bars.

Examples

This example creates six subplots showing the effects of different arguments for `bar3`. The data `Y` is a 7-by-3 matrix generated using the `cool` colormap:

```
Y = cool(7);
subplot(3,2,1)
bar3(Y,'detached')
title('Detached')
subplot(3,2,2)
bar3(Y,0.25,'detached')
title('Width = 0.25')
subplot(3,2,3)
bar3(Y,'grouped')
title('Grouped')
subplot(3,2,4)
bar3(Y,0.5,'grouped')
title('Width = 0.5')
subplot(3,2,5)
bar3(Y,'stacked')
title('Stacked')
subplot(3,2,6)
bar3(Y,0.3,'stacked')
title('Width = 0.3')
colormap([1 0 0;0 1 0;0 0 1])
```


Detached
Width = 0.25
Grouped
Width = 0.5
Stacked
Width = 0.3

See Also

`bar`, `LineSpec`, `patch`

"Area, Bar, and Pie Plots" on page 1-88 for related functions

"Bar and Area Graphs" for more examples

Barseries Properties

Purpose

Define barseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default properties for barseries objects.

See “Plot Objects” for more information on barseries objects.

Barseries Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of barseries objects in legends. The `Annotation` property enables you to specify whether this barseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the barseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the barseries object in a legend as one entry, but not its children objects
`off`	Do not include the barseries or its children in a legend (default)
`children`	Include only the children of the barseries as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BarLayout`

`{grouped} | stacked`

Specify grouped or stacked bars. Grouped bars display *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in the input argument `Y`. The group contains one bar per column in `Y`.

Stacked bars display one bar for each row in the input argument `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`BarWidth`

scalar in range [0 1]

Width of individual bars. `BarWidth` specifies the relative bar width and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, the bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`BaseLine`

handle of baseline

Handle of the baseline object. This property contains the handle of the line object used as the baseline. You can set the properties of this line using its handle. For example, the following statements create a bar graph, obtain the handle of the baseline from the barseries object, and then set line properties that make the baseline a dashed, red line.

```
bar_handle = bar(randn(10,1));
baseline_handle = get(bar_handle,'BaseLine');
set(baseline_handle,'LineStyle','--','Color','red')
```

`BaseValue`
double: *y*-axis value

Value where baseline is drawn. You can specify the value along the *y*-axis (vertical bars) or *x*-axis (horizontal bars) at which MATLAB draws the baseline.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this barseries object. The `legend` function uses the string defined by the `DisplayName` property to label this barseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this barseries object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EdgeColor`

`{[0 0 0]} | none | ColorSpec`

Color of line that separates filled areas. You can set the color of the edges of filled areas to a three-element RGB vector or one of the MATLAB predefined names, including the string `none`. The default edge color is black. See `ColorSpec` for more information on specifying color.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode` `none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceColor`
`{flat} | none | ColorSpec`

Color of filled areas. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that `EdgeColor` is drawn independently of `FaceColor`
- `flat` — The color of the filled areas is determined by the figure colormap. See `colormap` for information on setting the colormap.

  See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions

invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.

- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`

`on | {off}`

Select barseries object on bars or area of extent. This property enables you to select barseries objects in two ways:

- Select by clicking bars (default).
- Select by clicking anywhere in the extent of the bar graph.

When `HitTestArea` is `off`, you must click the bars to select the barseries object. When `HitTestArea` is `on`, you can select the barseries object by clicking anywhere within the extent of the bar graph (i.e., anywhere within a rectangle that encloses all the bars).

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`ShowBaseLine`
`{on} | off`

Turn baseline display on or off. This property determines whether bar plots display a baseline from which the bars are drawn. By default, the baseline is displayed.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks.

For example, you might create a barseries object and set the `Tag` property:

```
t = bar(Y,'Tag','bar1')
```

When you want to access the barseries object, you can use `findobj` to find the barseries object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `bar1`.

```
set(findobj('Tag','bar1'),'FaceColor','red')
```

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For barseries objects, `Type` is `hggroup`.

The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`

handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
array

Location of bars. The *x*-axis intervals for the vertical bars or *y*-axis intervals for horizontal bars (as specified by the `x` input argument). If `YData` is a vector, `XData` must be the same size. If `YData` is a matrix, the length of `XData` must be equal to the number of rows in `YData`.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `x` input

argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
scalar, vector, or matrix

Bar plot data. YData contains the data plotted as bars (the Y input argument). Each value in YData is represented by a bar in the bar graph. If XYData is a matrix, the bar function creates a "group" or a "stack" of bars for each column in the matrix. See “Bar Graph Options” in the bar, barh reference page for examples of grouped and stacked bar graphs.

The input argument Y in the bar function calling syntax assigns values to YData.

YDataSource
string (MATLAB variable)

Link YData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change YData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

base2dec

Purpose

Convert base N number string to decimal number

Syntax

```
d = base2dec('strn', base)
```

Description

`d = base2dec('strn', base)` converts the string number *strn* of the specified `base` into its decimal (base 10) equivalent. `base` must be an integer between 2 and 36. If `'strn'` is a character array, each row is interpreted as a string in the specified `base`.

Examples

The expression `base2dec('212',3)` converts 212_3 to decimal, returning 23.

See Also

`dec2base`

Purpose

Produce beep sound

Syntax

```
beep
beep on
beep off
s = beep
```

Description

`beep` produces your computer's default beep sound.

`beep on` turns the beep on.

`beep off` turns the beep off.

`s = beep` returns the current beep mode (`on` or `off`).

besselh

Purpose

Bessel function of third kind (Hankel function)

Syntax

```
H = besselh(nu,K,Z)
H = besselh(nu,Z)
H = besselh(nu,K,Z,1)
[H,ierr] = besselh(...)
```

Definitions

The differential equation

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} + (z^2 - \nu^2)y = 0$$

where ν is a nonnegative constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*. $J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν. $Y_\nu(z)$ is a second solution of Bessel's equation – linearly independent of $J_\nu(z)$ – defined by

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

The relationship between the Hankel and Bessel functions is

$$H_\nu^{(1)}(z) = J_\nu(z) + i\ Y_\nu(z)$$

$$H_\nu^{(2)}(z) = J_\nu(z) - i\ Y_\nu(z)$$

where $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`.

Description

`H = besselh(nu,K,Z)` computes the Hankel function $H_\nu^{(K)}(z)$, where `K` = 1 or 2, for each element of the complex array `Z`. If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, `besselh` expands it to the other input's size. If one input is a row

vector and the other is a column vector, the result is a two-dimensional table of function values.

`H = besselh(nu,Z)` uses `K = 1`.

`H = besselh(nu,K,Z,1)` scales $H_\nu^{(K)}(z)$ by `exp(-i*Z)` if `K = 1`, and by `exp(+i*Z)` if `K = 2`.

`[H,ierr] = besselh(...)` also returns completion flags in an array the same size as `H`.

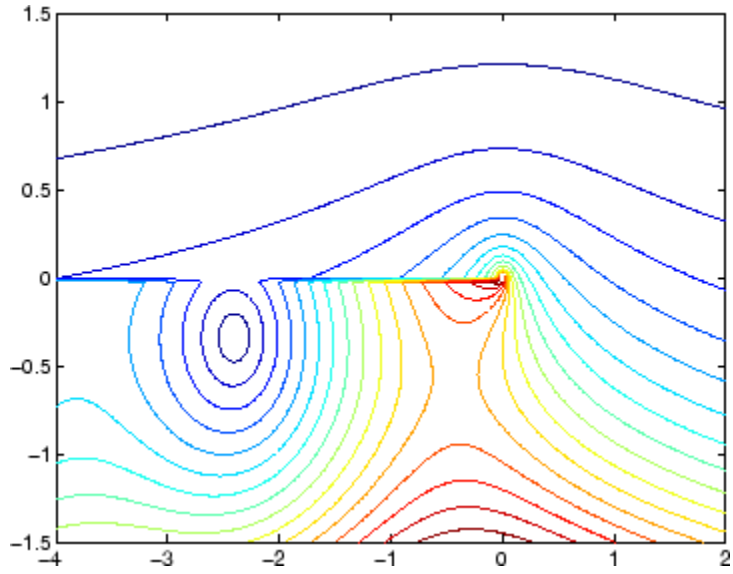
ierr	Description
0	`besselh` successfully computed the Hankel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Examples

This example generates the contour plots of the modulus and phase of the Hankel function $H_0^{(1)}(z)$ shown on page 359 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

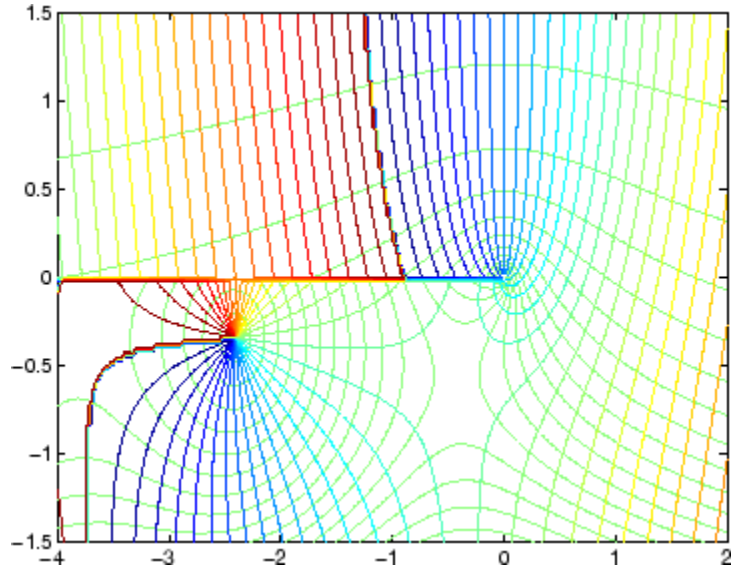
It first generates the modulus contour plot

```
[X,Y] = meshgrid(-4:0.025:2,-1.5:0.025:1.5);
H = besselh(0,1,X+i*Y);
contour(X,Y,abs(H),0:0.2:3.2), hold on
```



then adds the contour plot of the phase of the same function.

```
contour(X,Y,(180/pi)*angle(H),-180:10:180); hold off
```



See Also

`besselj`, `bessely`, `besseli`, `besselk`

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965.

Purpose

Modified Bessel function of first kind

Syntax

```
I = besseli(nu,Z)
I = besseli(nu,Z,1)
[I,ierr] = besseli(...)
```

Definitions

The differential equation

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} - (z^2+\nu^2)y = 0$$

where ν is a real constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

$I_\nu(z)$ and $I_{-\nu}(z)$ form a fundamental set of solutions of the modified Bessel's equation for noninteger ν. $I_\nu(z)$ is defined by

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\ \Gamma(\nu+k+1)}$$

where $\Gamma(a)$ is the gamma function.

$K_\nu(z)$ is a second solution, independent of $I_\nu(z)$. It can be computed using `besselk`.

Description

`I = besseli(nu,Z)` computes the modified Bessel function of the first kind, $I_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`I = besseli(nu,Z,1)` computes `besseli(nu,Z).*exp(-abs(real(Z)))`.

`[I,ierr] = besseli(...)` also returns completion flags in an array the same size as `I`.

ierr	Description
0	`besseli` successfully computed the modified Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Examples

Example 1

```
format long
z = (0:0.2:1)';

besseli(1,z)

ans =
                   0
   0.10050083402813
   0.20402675573357
   0.31370402560492
   0.43286480262064
   0.56515910399249
```

Example 2

`besseli(3:9,(0:.2,10)',1)` generates the entire table on page 423 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*

Algorithm

The `besseli` functions use a Fortran MEX-file to call a library developed by D.E. Amos [3] [4].

See Also

`airy`, `besselh`, `besselj`, `besselk`, `bessely`

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

Purpose

Bessel function of first kind

Syntax

```
J = besselj(nu,Z)
J = besselj(nu,Z,1)
[J,ierr] = besselj(nu,Z)
```

Definition

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} + (z^2 - \nu^2)y = 0$$

where ν is a real constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*.

$J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν. $J_\nu(z)$ is defined by

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{z^2}{4}\right)^k}{k!\ \Gamma(\nu + k + 1)}$$

where $\Gamma(a)$ is the gamma function.

$Y_\nu(z)$ is a second solution of Bessel's equation that is linearly independent of $J_\nu(z)$. It can be computed using `bessely`.

Description

`J = besselj(nu,Z)` computes the Bessel function of the first kind, $J_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`J = besselj(nu,Z,1)` computes
`besselj(nu,Z).*exp(-abs(imag(Z)))`.

`[J,ierr] = besselj(nu,Z)` also returns completion flags in an array the same size as `J`.

ierr	Description
0	`besselj` successfully computed the Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Remarks

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind,

$$H_\nu^{(1)}(z) = J_\nu(z) + i\ Y_\nu(z)$$

$$H_\nu^{(2)}(z) = J_\nu(z) - i\ Y_\nu(z)$$

where $H_\nu^{(K)}(z)$ is `besselh`, $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

Examples

Example 1

```
format long
z = (0:0.2:1)';

besselj(1,z)
```

```
ans =
                    0
   0.09950083263924
   0.19602657795532
   0.28670098806392
   0.36884204609417
   0.44005058574493
```

Example 2

`besselj(3:9,(0:.2:10)')` generates the entire table on page 398 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithm

The `besselj` function uses a Fortran MEX-file to call a library developed by D.E. Amos [3] [4].

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

See Also

`besselh`, `besseli`, `besselk`, `bessely`

besselk

Purpose

Modified Bessel function of second kind

Syntax

```
K = besselk(nu,Z)
K = besselk(nu,Z,1)
[K,ierr] = besselk(...)
```

Definitions

The differential equation

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} - (z^2+\nu^2)y = 0$$

where ν is a real constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

A solution $K_\nu(z)$ of the second kind can be expressed as

$$K_\nu(z) = \left(\frac{\pi}{2}\right)\frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}$$

where $I_\nu(z)$ and $I_{-\nu}(z)$ form a fundamental set of solutions of the modified Bessel's equation for noninteger ν

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\ \Gamma(\nu+k+1)}$$

and $\Gamma(a)$ is the gamma function. $K_\nu(z)$ is independent of $I_\nu(z)$. $I_\nu(z)$ can be computed using `besseli`.

Description

`K = besselk(nu,Z)` computes the modified Bessel function of the second kind, $K_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If nu and Z are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`K = besselk(nu,Z,1)` computes `besselk(nu,Z).*exp(Z)`.

`[K,ierr] = besselk(...)` also returns completion flags in an array the same size as `K`.

ierr	Description
0	`besselk` successfully computed the modified Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Examples

Example 1

```
format long
z = (0:0.2:1)';

besselk(1,z)

ans =
                Inf
   4.77597254322047
   2.18435442473269
   1.30283493976350
   0.86178163447218
   0.60190723019723
```

Example 2

`besselk(3:9,(0:.2:10)',1)` generates part of the table on page 424 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithm

The `besselk` function uses a Fortran MEX-file to call a library developed by D.E. Amos [3][4].

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

See Also

`airy`, `besselh`, `besseli`, `besselj`, `bessely`

Purpose

Bessel function of second kind

Syntax

```
Y = bessely(nu,Z)
Y = bessely(nu,Z,1)
[Y,ierr] = bessely(nu,Z)
```

Definition

The differential equation

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + (z^2 - \nu^2) y = 0$$

where ν is a real constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*.

A solution $Y_\nu(z)$ of the second kind can be expressed as

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

where $J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{z^2}{4}\right)^k}{k!\ \Gamma(\nu + k + 1)}$$

and $\Gamma(a)$ is the gamma function. $Y_\nu(z)$ is linearly independent of $J_\nu(z)$.

$J_\nu(z)$ can be computed using `besselj`.

Description

`Y = bessely(nu,Z)` computes Bessel functions of the second kind, $Y_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If nu and Z are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`Y = bessely(nu,Z,1)` computes `bessely(nu,Z).*exp(-abs(imag(Z)))`.

`[Y,ierr] = bessely(nu,Z)` also returns completion flags in an array the same size as `Y`.

ierr	Description
0	`bessely` successfully computed the Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Remarks

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind,

$$H_\nu^{(1)}(z) = J_\nu(z) + i\ Y_\nu(z)$$

$$H_\nu^{(2)}(z) = J_\nu(z) - i\ Y_\nu(z)$$

where $H_\nu^{(K)}(z)$ is `besselh`, $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

Examples

Example 1

```
format long
z = (0:0.2:1)';

bessely(1,z)

ans =
                -Inf
  -3.32382498811185
  -1.78087204427005
  -1.26039134717739
  -0.97814417668336
  -0.78121282130029
```

Example 2

`bessely(3:9,(0:.2:10)')` generates the entire table on page 399 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithm

The `bessely` function uses a Fortran MEX-file to call a library developed by D. E Amos [3] [4].

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

See Also

besselh, besseli, besselj, besselk

Purpose

Beta function

Syntax

```
B = beta(Z,W)
```

Definition

The beta function is

$$B(z, w) = \int_0^1 t^{z-1}(1-t)^{w-1}\,dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

where $\Gamma(z)$ is the gamma function.

Description

`B = beta(Z,W)` computes the beta function for corresponding elements of arrays `Z` and `W`. The arrays must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

In this example, which uses integer arguments,

```
beta(n,3)
   = (n-1)!*2!/(n+2)!
   = 2/(n*(n+1)*(n+2))
```

is the ratio of fairly small integers, and the rational format is able to recover the exact result.

```
format rat
beta((0:10)',3)

ans =

     1/0
     1/3
     1/12
     1/30
     1/60
     1/105
     1/168
     1/252
```

```
1/360
1/495
1/660
```

Algorithm

```
beta(z,w) = exp(gammaln(z)+gammaln(w)-gammaln(z+w))
```

See Also

betainc, betaln, gammaln

Purpose

Incomplete beta function

Syntax

```
I = betainc(X,Z,W)
I = betainc(X,Z,tail)
```

Definition

The incomplete beta function is

$$I_x(z, w) = \frac{1}{B(z, w)} \int_0^x t^{z-1}(1-t)^{w-1}\, dt$$

where $B(z, w)$, the beta function, is defined as

$$B(z, w) = \int_0^1 t^{z-1}(1-t)^{w-1}\, dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

and $\Gamma(z)$ is the gamma function.

Description

`I = betainc(X,Z,W)` computes the incomplete beta function for corresponding elements of the arrays `X`, `Z`, and `W`. The elements of `X` must be in the closed interval $[0,1]$. The arrays `Z` and `W` must be nonnegative and real. All arrays must be the same size, or any of them can be scalar.

`I = betainc(X,Z,tail)` specifies the tail of the incomplete beta function. Choices are:

`'lower'` (the default)	Computes the integral from `0` to `x`
`'upper'`	Computes the integral from `x` to `1`

These functions are related as follows:

```
1-betainc(X,Z,W) = betainc(X,Z,W,'upper')
```

Note that especially when the upper tail value is close to `0`, it is more accurate to use the `'upper'` option than to subtract the `'lower'` value from `1`.

Examples

```
format long
betainc(.5,(0:10)',3)

ans =
   1.00000000000000
   0.87500000000000
   0.68750000000000
   0.50000000000000
   0.34375000000000
   0.22656250000000
   0.14453125000000
   0.08984375000000
   0.05468750000000
   0.03271484375000
   0.01928710937500
```

See Also

beta, betaln

Purpose

Logarithm of beta function

Syntax

```
L = betaln(Z,W)
```

Description

`L = betaln(Z,W)` computes the natural logarithm of the `beta` function `log(beta(Z,W))`, for corresponding elements of arrays `Z` and `W`, without computing `beta(Z,W)`. Since the beta function can range over very large or very small values, its logarithm is sometimes more useful.

`Z` and `W` must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

```
x = 510
betaln(x,x)

ans =
      -708.8616
```

`-708.8616` is slightly less than `log(realmin)`. Computing `beta(x,x)` directly would underflow (or be denormal).

Algorithm

```
betaln(z,w) = gammaln(z)+gammaln(w)-gammaln(z+w)
```

See Also

`beta`, `betainc`, `gammaln`

bicg

Purpose

Biconjugate gradients method

Syntax

```
x = bicg(A,b)
bicg(A,b,tol)
bicg(A,b,tol,maxit)
bicg(A,b,tol,maxit,M)
bicg(A,b,tol,maxit,M1,M2)
bicg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicg(A,b,...)
[x,flag,relres] = bicg(A,b,...)
[x,flag,relres,iter] = bicg(A,b,...)
[x,flag,relres,iter,resvec] = bicg(A,b,...)
```

Description

`x = bicg(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicg` converges, it displays a message to that effect. If `bicg` fails to converge after the maximum number of iterations or halts for any reason, it prints a warning message that includes the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicg` uses the default, `1e-6`.

`bicg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicg` uses the default, `min(n,20)`.

`bicg(A,b,tol,maxit,M)` and `bicg(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system

`inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicg` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`bicg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `bicg` uses the default, an all-zero vector.

`[x,flag] = bicg(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`bicg` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicg` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicg` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicg` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = bicg(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicg(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = bicg(A,b,...)` also returns a vector of the residual norms at each iteration including `norm(b-A*x0)`.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
```

```
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = bicg(A,b,tol,maxit,M1,M2);
```

displays this message:

```
bicg converged at iteration 9 to a solution with relative
residual 5.3e-009
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_bicg` that

- Calls `bicg` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_bicg` are available to `afun`.

The following shows the code for `run_bicg`:

```
function x1 = run_bicg
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = bicg(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
```

```
            y = 4 * x;
            y(1:n-1) = y(1:n-1) - 2 * x(2:n);
            y(2:n) = y(2:n) - x(1:n-1);
         elseif strcmp(transp_flag,'notransp') % y = A*x
            y = 4 * x;
            y(2:n) = y(2:n) - 2 * x(1:n-1);
            y(1:n-1) = y(1:n-1) - x(2:n);
         end
      end
   end
```

When you enter

```
x1=run_bicg;
```

MATLAB displays the message

```
bicg converged at iteration 9 to a solution with ...
relative residual
5.3e-009
```

Example 3

This example demonstrates the use of a preconditioner. Start with `A = west0479`, a real 479-by-479 sparse matrix, and define `b` so that the true solution is a vector of all ones.

```
load west0479;
A = west0479;
b = sum(A,2);
```

You can accurately solve `A*x = b` using backslash since `A` is not so large.

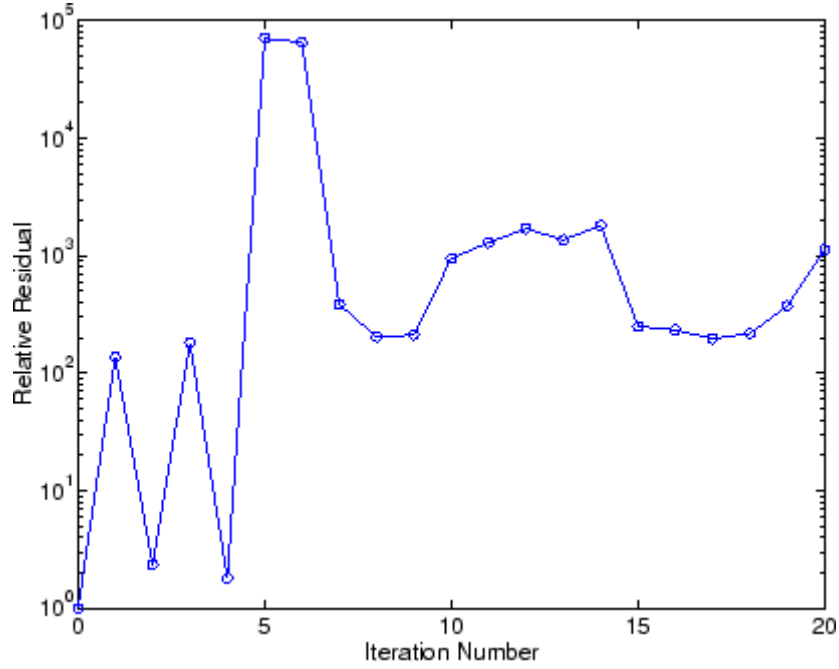
```
x = A \ b;
norm(b-A*x) / norm(b)

ans =
    8.3154e-017
```

Now try to solve `A*x = b` with `bicg`.

```
[x,flag,relres,iter,resvec] = bicg(A,b)

flag =
          1
relres =
          1
iter =
          0
```

The value of `flag` indicates that `bicg` iterated the default 20 times without converging. The value of `iter` shows that the method behaved so badly that the initial all-zero guess was better than all the subsequent iterates. The value of `relres` supports this: `relres = norm(b-A*x)/norm(b) = norm(b)/norm(b) = 1`. You can confirm that the unpreconditioned method oscillates rather wildly by plotting the relative residuals at each iteration.

```
semilogy(0:20,resvec/norm(b),'-o')
xlabel('Iteration Number')
ylabel('Relative Residual')
```



Now, try an incomplete LU factorization with a drop tolerance of `1e-5` for the preconditioner.

```
[L1,U1] = luinc(A,1e-5);
Warning: Incomplete upper triangular factor has 1 zero diagonal.
         It cannot be used as a preconditioner for an iterative
         method.

nnz(A), nnz(L1), nnz(U1)

ans =
         1887
ans =
         5562
ans =
         4320
```

The zero on the main diagonal of the upper triangular `U1` indicates that `U1` is singular. If you try to use it as a preconditioner,

```
[x,flag,relres,iter,resvec] = bicg(A,b,1e-6,20,L1,U1)

flag =
         2
relres =
         1
iter =
         0
resvec =
         7.0557e+005
```

the method fails in the very first iteration when it tries to solve a system of equations involving the singular `U1` using backslash. `bicg` is forced to return the initial estimate since no other iterates were produced.

Try again with a slightly less sparse preconditioner.

```
[L2,U2] = luinc(A,1e-6);

nnz(L2), nnz(U2)

ans =
         6231
ans =
         4559
```

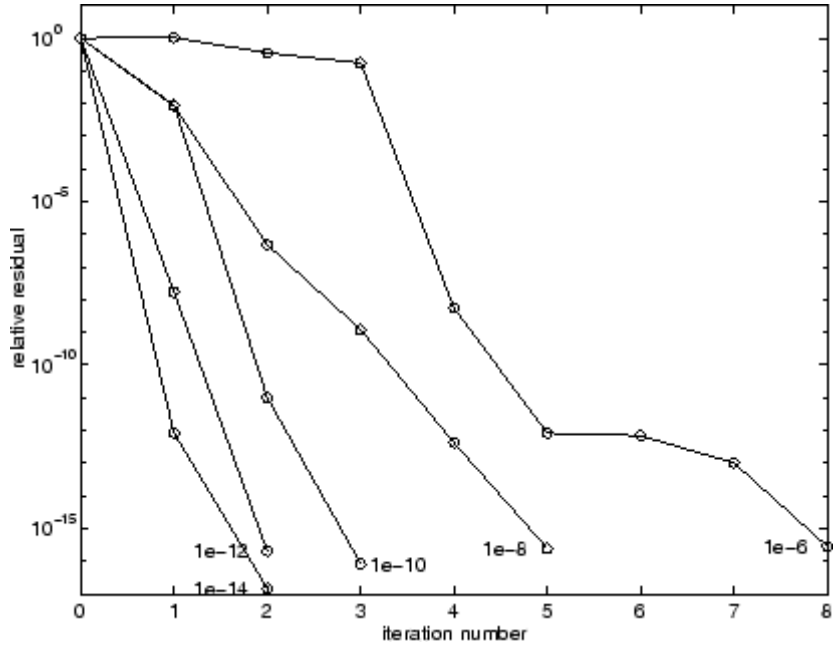
This time `U2` is nonsingular and may be an appropriate preconditioner.

```
[x,flag,relres,iter,resvec] = bicg(A,b,1e-15,10,L2,U2)

flag =
         0
relres =
         2.8664e-016
iter =
```

```
     8
```

and `bicg` converges to within the desired tolerance at iteration number 8. Decreasing the value of the drop tolerance increases the fill-in of the incomplete factors but also increases the accuracy of the approximation to the original matrix. Thus, the preconditioned system becomes closer to `inv(U)*inv(L)*L*U*x = inv(U)*inv(L)*b`, where `L` and `U` are the true LU factors, and closer to being solved within a single iteration.

The next graph shows the progress of `bicg` using six different incomplete LU factors as preconditioners. Each line in the graph is labeled with the drop tolerance of the preconditioner used in `bicg`.



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

See Also

bicgstab, cgs, gmres, ilu, lsqr, luinc, minres, pcg, qmr, symmlq, function_handle (@), mldivide (\)

Purpose

Biconjugate gradients stabilized method

Syntax

```
x = bicgstab(A,b)
bicgstab(A,b,tol)
bicgstab(A,b,tol,maxit)
bicgstab(A,b,tol,maxit,M)
bicgstab(A,b,tol,maxit,M1,M2)
bicgstab(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstab(A,b,...)
[x,flag,relres] = bicgstab(A,b,...)
[x,flag,relres,iter] = bicgstab(A,b,...)
[x,flag,relres,iter,resvec] = bicgstab(A,b,...)
```

Description

`x = bicgstab(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicgstab` converges, a message to that effect is displayed. If `bicgstab` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicgstab(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicgstab` uses the default, `1e-6`.

`bicgstab(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicgstab` uses the default, `min(n,20)`.

`bicgstab(A,b,tol,maxit,M)` and `bicgstab(A,b,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system

`inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicgstab` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`bicgstab(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `bicgstab` uses the default, an all zero vector.

`[x,flag] = bicgstab(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`bicgstab` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicgstab` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicgstab` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicgstab` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = bicgstab(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicgstab(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`. `iter` can be an integer + 0.5, indicating convergence halfway through an iteration.

`[x,flag,relres,iter,resvec] = bicgstab(A,b,...)` also returns a vector of the residual norms at each half iteration, including `norm(b-A*x0)`.

Example

Example 1

This example first solves `Ax = b` by providing `A` and the preconditioner `M1` directly as arguments.

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = bicgstab(A,b,tol,maxit,M1);
```

displays the message

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 6.7e-014
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in an M-file `run_bicgstab` that

- Calls `bicgstab` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_bicgstab` are available to `afun` and `mfun`.

The following shows the code for `run_bicgstab`:

```
function x1 = run_bicgstab
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);
x1 = bicgstab(@afun,b,tol,maxit,@mfun);
```

```
    function y = afun(x)
       y = [0; x(1:n-1)] + ...
           [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
           [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_bicgstab;
```

MATLAB displays the message

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 6.7e-014
```

Example 3

This examples demonstrates the use of a preconditioner. Start with `A = west0479`, a real 479-by-479 sparse matrix, and define `b` so that the true solution is a vector of all ones.

```
load west0479;
A = west0479;
b = sum(A,2);
[x,flag] = bicgstab(A,b)
```

`flag` is `1` because `bicgstab` does not converge to the default tolerance `1e-6` within the default 20 iterations.

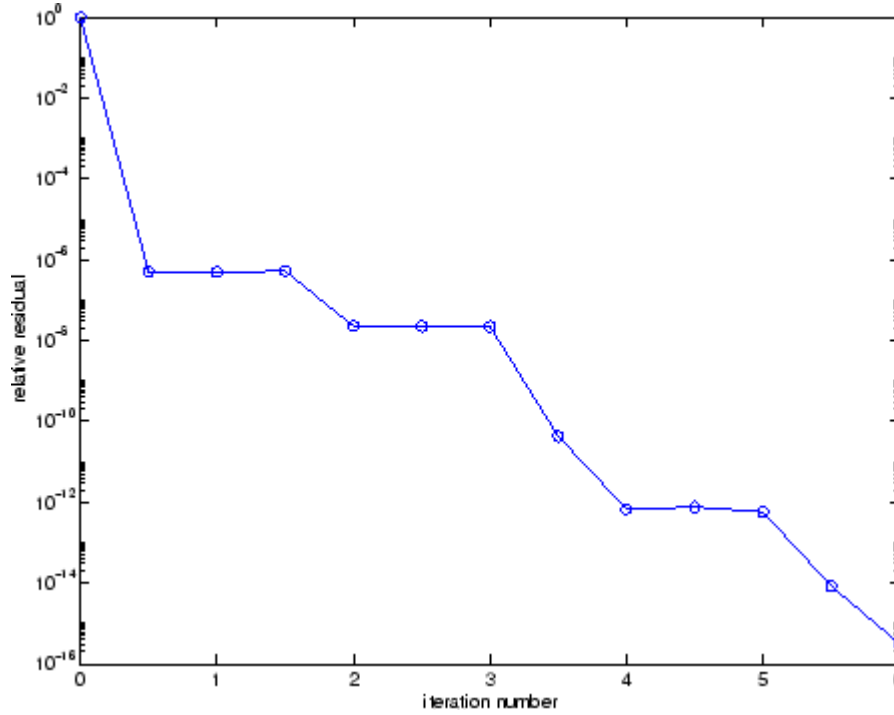
```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = bicgstab(A,b,1e-6,20,L1,U1)
```

`flag1` is `2` because the upper triangular `U1` has a zero on its diagonal. This causes `bicgstab` to fail in the first iteration when it tries to solve a system such as `U1*y = r` using backslash.

```
[L2,U2] = luinc(A,1e-6);
[x2,flag2,relres2,iter2,resvec2] = bicgstab(A,b,1e-15,10,L2,U2)
```

`flag2` is `0` because `bicgstab` converges to the tolerance of `3.1757e-016` (the value of `relres2`) at the sixth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. `resvec2(1) = norm(b)` and `resvec2(13) = norm(b-A*x2)`. You can follow the progress of `bicgstab` by plotting the relative residuals at the halfway point and end of each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:0.5:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] van der Vorst, H.A., "BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, March 1992, Vol. 13, No. 2, pp. 631-644.

See Also

`bicg`, `cgs`, `gmres`, `lsqr`, `luinc`, `minres`, `pcg`, `qmr`, `symmlq`, `function_handle` (`@`), `mldivide` (`\`)

Purpose

Convert binary number string to decimal number

Syntax

```
bin2dec(binarystr)
```

Description

`bin2dec(binarystr)` interprets the binary string *`binarystr`* and returns the equivalent decimal number.

`bin2dec` ignores any space (`' '`) characters in the input string.

Examples

Binary `010111` converts to decimal `23`:

```
bin2dec('010111')
ans =
    23
```

Because space characters are ignored, this string yields the same result:

```
bin2dec(' 010   111 ')
ans =
    23
```

See Also

`dec2bin`

binary

Purpose

Set FTP transfer type to binary

Syntax

```
binary(f)
```

Description

`binary(f)` sets the FTP download and upload mode to binary, which does not convert new lines, where `f` was created using `ftp`. Use this function when downloading or uploading any nontext file, such as an executable or ZIP archive.

Examples

Connect to the MathWorks FTP server, and display the FTP object.

```
tmw=ftp('ftp.mathworks.com');
disp(tmw)
FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: binary
```

Note that the FTP object defaults to binary mode.

Use the `ascii` function to set the FTP mode to ASCII, and use the `disp` function to display the FTP object.

```
ascii(tmw)
disp(tmw)
FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: ascii
```

Note that the FTP object is now set to ASCII mode.

Use the `binary` function to set the FTP mode to binary, and use the `disp` function to display the FTP object.

```
binary(tmw)
```

```
disp(tmw)
FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: binary
```

Note that the FTP object's mode is again set to binary.

See Also

`ftp`, `ascii`

bitand

Purpose

Bitwise AND

Syntax

```
C = bitand(A, B)
```

Description

`C = bitand(A, B)` returns the bitwise AND of arguments `A` and `B`, where `A` and `B` are unsigned integers or arrays of unsigned integers.

Examples

Example 1

The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bitwise AND on these numbers yields 01001, or 9:

```
C = bitand(uint8(13), uint8(27))
C =
    9
```

Example 2

Create a truth table for a logical AND operation:

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);

TT = bitand(A, B)
TT =
    0    0
    0    1
```

See Also

`bitcmp`, `bitget`, `bitmax`, `bitor`, `bitset`, `bitshift`, `bitxor`

Purpose

Bitwise complement

Syntax

```
C = bitcmp(A)
C = bitcmp(A, n)
```

Description

`C = bitcmp(A)` returns the bitwise complement of `A`, where `A` is an unsigned integer or an array of unsigned integers.

`C = bitcmp(A, n)` returns the bitwise complement of `A` as an `n`-bit unsigned integer `C`. Input `A` may not have any bits set higher than `n` (that is, `A` may not have a value greater than `2^n-1`). The value of `n` can be no greater than the number of bits in the unsigned integer class of `A`. For example, if the class of `A` is `uint32`, then `n` must be a positive integer less than 32.

Examples

Example 1

With eight-bit arithmetic, the one's complement of 01100011 (decimal 99) is 10011100 (decimal 156):

```
C = bitcmp(uint8(99))
C =
   156
```

Example 2

The complement of hexadecimal A5 (decimal 165) is 5A:

```
x = hex2dec('A5')
x =
   165

dec2hex(bitcmp(x, 8))
ans =
5A
```

Next, find the complement of hexadecimal 000000A5:

```
dec2hex(bitcmp(x, 32))
```

```
ans =
FFFFFF5A
```

See Also

bitand, bitget, bitmax, bitor, bitset, bitshift, bitxor

Purpose

Bit at specified position

Syntax

```
C = bitget(A, bit)
```

Description

`C = bitget(A, bit)` returns the value of the bit at position *`bit`* in `A`. Operand `A` must be an unsigned integer or an array of unsigned integers, and *`bit`* must be a number between 1 and the number of bits in the unsigned integer class of `A` (e.g., 32 for the `uint32` class).

Examples

Example 1

The `dec2bin` function converts decimal numbers to binary. However, you can also use the `bitget` function to show the binary representation of a decimal number. Just test successive bits from most to least significant:

```
disp(dec2bin(13))
1101

C = bitget(uint8(13), 4:-1:1)
C =
     1     1     0     1
```

Example 2

Prove that `intmax` sets all the bits to 1:

```
a = intmax('uint8');
if all(bitget(a, 1:8))
   disp('All the bits have value 1.')
   end

All the bits have value 1.
```

See Also

`bitand`, `bitcmp`, `bitmax`, `bitor`, `bitset`, `bitshift`, `bitxor`

bitmax

Purpose

Maximum double-precision floating-point integer

Syntax

```
bitmax
```

Description

`bitmax` returns the maximum unsigned double-precision floating-point integer for your computer. It is the value when all bits are set, namely the value $2^{53} - 1$.

Note Instead of integer-valued double-precision variables, use unsigned integers for bit manipulations and replace `bitmax` with `intmax`.

Examples

Display in different formats the largest floating point integer and the largest 32 bit unsigned integer:

```
format long e
bitmax
ans =
   9.007199254740991e+015

intmax('uint32')
ans =
   4294967295

format hex
bitmax
ans =
   433fffffffffffff

intmax('uint32')
ans =
   ffffffff
```

In the second bitmax statement, the last 13 hex digits of `bitmax` are `f`, corresponding to 52 1's (all 1's) in the mantissa of the binary

representation. The first 3 hex digits correspond to the sign bit 0 and the 11 bit biased exponent 10000110011 in binary (1075 in decimal), and the actual exponent is (1075-1023) = 52. Thus the binary value of bitmax is 1.111...111 x 2^52 with 52 trailing 1's, or 2^53-1.

See Also

bitand, bitcmp, bitget, bitor, bitset, bitshift, bitxor

bitor

Purpose

Bitwise OR

Syntax

```
C = bitor(A, B)
```

Description

`C = bitor(A, B)` returns the bitwise OR of arguments `A` and `B`, where `A` and `B` are unsigned integers or arrays of unsigned integers.

Examples

Example 1

The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bitwise OR on these numbers yields 11111, or 31.

```
C = bitor(uint8(13), uint8(27))
C =
    31
```

Example 2

Create a truth table for a logical OR operation:

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);

TT = bitor(A, B)
TT =
    0    1
    1    1
```

See Also

`bitand`, `bitcmp`, `bitget`, `bitmax`, `bitset`, `bitshift`, `bitxor`

Purpose

Set bit at specified position

Syntax

```
C = bitset(A, bit)
C = bitset(A, bit, v)
```

Description

`C = bitset(A, bit)` sets bit position *`bit`* in `A` to 1 (on). `A` must be an unsigned integer or an array of unsigned integers, and *`bit`* must be a number between 1 and the number of bits in the unsigned integer class of `A` (e.g., 32 for the `uint32` class).

`C = bitset(A, bit, v)` sets the bit at position *`bit`* to the value `v`, which must be either 0 or 1.

Examples

Example 1

Setting the fifth bit in the five-bit binary representation of the integer 9 (01001) yields 11001, or 25:

```
C = bitset(uint8(9), 5)
C =
    25
```

Example 2

Repeatedly subtract powers of 2 from the largest `uint32` value:

```
a = intmax('uint32')
for k = 1:32
   a = bitset(a, 32-k+1, 0)
   end
```

See Also

`bitand`, `bitcmp`, `bitget`, `bitmax`, `bitor`, `bitshift`, `bitxor`

bitshift

Purpose

Shift bits specified number of places

Syntax

```
C = bitshift(A, k)
C = bitshift(A, k, n)
```

Description

`C = bitshift(A, k)` returns the value of `A` shifted by `k` bits. Input argument `A` must be an unsigned integer or an array of unsigned integers. Shifting by `k` is the same as multiplication by `2^k`. Negative values of `k` are allowed and this corresponds to shifting to the right, or dividing by `2^abs(k)` and truncating to an integer. If the shift causes `C` to overflow the number of bits in the unsigned integer class of `A`, then the overflowing bits are dropped.

`C = bitshift(A, k, n)` causes any bits that overflow `n` bits to be dropped. The value of `n` must be less than or equal to the length in bits of the unsigned integer class of `A` (e.g., `n <= 32` for `uint32`).

Instead of using `bitshift(A, k, 8)` or another power of 2 for `n`, consider using `bitshift(uint8(A), k)` or the appropriate unsigned integer class for `A`.

Examples

Example 1

Shifting 1100 (12, decimal) to the left two bits yields 110000 (48, decimal).

```
C = bitshift(12, 2)
C =
    48
```

Example 2

Repeatedly shift the bits of an unsigned 16 bit value to the left until all the nonzero bits overflow. Track the progress in binary:

```
a = intmax('uint16');
disp(sprintf( ...
   'Initial uint16 value %5d is %16s in binary', ...
   a, dec2bin(a)))
```

```
for k = 1:16
   a = bitshift(a, 1);
   disp(sprintf( ...
      'Shifted uint16 value %5d is %16s in binary',...
      a, dec2bin(a)))
end
```

See Also

bitand, bitcmp, bitget, bitmax, bitor, bitset, bitxor, fix

bitxor

Purpose

Bitwise XOR

Syntax

```
C = bitxor(A, B)
```

Description

`C = bitxor(A, B)` returns the bitwise XOR of arguments `A` and `B`, where `A` and `B` are unsigned integers or arrays of unsigned integers.

Examples

Example 1

The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bitwise XOR on these numbers yields 10110, or 22.

```
C = bitxor(uint8(13), uint8(27))
C =
    22
```

Example 2

Create a truth table for a logical XOR operation:

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);

TT = bitxor(A, B)
TT =
    0    1
    1    0
```

See Also

`bitand`, `bitcmp`, `bitget`, `bitmax`, `bitor`, `bitset`, `bitshift`

Purpose

Create string of blank characters

Syntax

```
blanks(n)
```

Description

`blanks(n)` is a string of `n` blanks.

Examples

`blanks` is useful with the `display` function. For example,

```
disp(['xxx' blanks(20) 'yyy'])
```

displays twenty blanks between the strings `'xxx'` and `'yyy'`.

`disp(blanks(n)')` moves the cursor down `n` lines.

See Also

`clc`, `format`, `home`

blkdiag

Purpose

Construct block diagonal matrix from input arguments

Syntax

```
out = blkdiag(a,b,c,d,...)
```

Description

`out = blkdiag(a,b,c,d,...)`, where `a`, `b`, `c`, `d`, `...` are matrices, outputs a block diagonal matrix of the form

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & \ldots \end{bmatrix}$$

The input matrices do not have to be square, nor do they have to be of equal size.

See Also

`diag`, `horzcat`, `vertcat`

Purpose

Axes border

Syntax

```
box on
box off
box
box(axes_handle,...)
```

Description

`box on` displays the boundary of the current axes.

`box off` does not display the boundary of the current axes.

`box` toggles the visible state of the current axes boundary.

`box(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

Algorithm

The `box` function sets the axes `Box` property to `on` or `off`.

See Also

`axes`, `grid`

"Axes Operations" on page 1-96 for related functions

break

Purpose

Terminate execution of `for` or `while` loop

Syntax

```
break
```

Description

`break` terminates the execution of a `for` or `while` loop. Statements in the loop that appear after the `break` statement are not executed.

In nested loops, `break` exits only from the loop in which it occurs. Control passes to the statement that follows the `end` of that loop.

Remarks

`break` is not defined outside a `for` or `while` loop. Use `return` in this context instead.

Examples

The example below shows a `while` loop that reads the contents of the file `fft.m` into a MATLAB character array. A `break` statement is used to exit the `while` loop when the first empty line is encountered. The resulting character array contains the M-file help for the `fft` program.

```
fid = fopen('fft.m','r');
s = '';
while ~feof(fid)
   line = fgetl(fid);
   if isempty(line), break, end
   s = strvcat(s,line);
end
disp(s)
```

See Also

`for`, `while`, `end`, `continue`, `return`

Purpose

Brighten or darken colormap

Syntax

```
brighten(beta)
brighten(h,beta)
newmap = brighten(beta)
newmap = brighten(cmap,beta)
```

Description

`brighten` increases or decreases the color intensities in a colormap. The modified colormap is brighter if `0 < beta < 1` and darker if `1 < beta < 0`.

`brighten(beta)` replaces the current colormap with a brighter or darker colormap of essentially the same colors. `brighten(beta)`, followed by `brighten(-beta)`, where `beta < 1`, restores the original map.

`brighten(h,beta)` brightens all objects that are children of the figure having the handle `h`.

`newmap = brighten(beta)` returns a brighter or darker version of the current colormap without changing the display.

`newmap = brighten(cmap,beta)` returns a brighter or darker version of the colormap `cmap` without changing the display.

Examples

Brighten and then darken the current colormap:

```
beta = .5; brighten(beta);
beta = -.5; brighten(beta);
```

Algorithm

The values in the colormap are raised to the power of gamma, where gamma is

$$\gamma = \begin{cases} 1-\beta, & \beta > 0 \\ \dfrac{1}{1+\beta}, & \beta \le 0 \end{cases}$$

`brighten` has no effect on graphics objects defined with true color.

See Also

`colormap`, `rgbplot`

"Color Operations" on page 1-98 for related functions

"Altering Colormaps" for more information

Purpose

Build searchable documentation database

Syntax

```
builddocsearchdb help_location
```

Description

`builddocsearchdb help_location` builds a searchable database of user-added HTML and related help files in the specified help location. The `help_location` argument is the full path to the directory containing the help files. The database enables the Help browser to search for content within the help files.

`builddocsearchdb` creates a directory named `helpsearch` under `help_location`. The `helpsearch` directory contains the search database files. Add the location of the `helpsearch` directory to your `info.xml` file.

The `helpsearch` directory works only with the version of MATLAB used to create it.

For a full discussion of this process, refer to "Adding HTML Help Files for Your Own Toolboxes to the Help Browser".

Examples

Build a search database for the documentation files found at `D:\work\mytoolbox\help`.

```
builddocsearchdb D:\work\mytoolbox\help
```

See Also

`doc`, `help`

builtin

Purpose

Execute built-in function from overloaded method

Syntax

```
builtin(function, x1, ..., xn)
[y1, ..., yn] = builtin(function, x1, ..., xn)
```

Description

`builtin` is used in methods that overload built-in functions to execute the original built-in function. If *`function`* is a string containing the name of a built-in function, then

`builtin(`*`function`*`, x1, ..., xn)` evaluates the specified `function` at the given arguments `x1` through `xn`. The `function` argument must be a string containing a valid function name. `function` cannot be a function handle.

`[y1, ..., yn] = builtin(`*`function`*`, x1, ..., xn)` returns multiple output arguments.

Remarks

`builtin(...)` is the same as `feval(...)` except that it calls the original built-in version of the function even if an overloaded one exists. (For this to work you must never overload `builtin`.)

See Also

`feval`

Purpose

Apply element-by-element binary operation to two arrays with singleton expansion enabled

Syntax

```
C = bsxfun(fun,A,B)
```

Description

`C = bsxfun(fun,A,B)` applies an element-by-element binary operation to arrays `A` and `B`, with singleton expansion enabled. `fun` is a function handle, and can either be an M-file function or one of the following built-in functions:

`@plus`	Plus
`@minus`	Minus
`@times`	Array multiply
`@rdivide`	Right array divide
`@ldivide`	Left array divide
`@power`	Array power
`@max`	Binary maximum
`@min`	Binary minimum
`@rem`	Remainder after division
`@mod`	Modulus after division
`@atan2`	Four quadrant inverse tangent
`@hypot`	Square root of sum of squares
`@eq`	Equal
`@ne`	Not equal
`@lt`	Less than
`@le`	Less than or equal to
`@gt`	Greater than
`@ge`	Greater than or equal to

`@and`	Element-wise logical AND
`@or`	Element-wise logical OR
`@xor`	Logical exclusive OR

If an M-file function is specified, it must be able to accept either two column vectors of the same size, or one column vector and one scalar, and return as output a column vector of the size as the input values.

Each dimension of `A` and `B` must either be equal to each other, or equal to `1`. Whenever a dimension of `A` or `B` is singleton (equal to `1`), the array is virtually replicated along the dimension to match the other array. The array may be diminished if the corresponding dimension of the other array is `0`.

The size of the output array `C` is equal to: `max(size(A),size(B)).*(size(A)>0 & size(B)>0)`.

Examples

In this example, `bsxfun` is used to subtract the column means from the matrix `A`.

```
A = magic(5);
A = bsxfun(@minus, A, mean(A))
A =

     4    11   -12    -5     2
    10    -8    -6     1     3
    -9    -7     0     7     9
    -3    -1     6     8   -10
    -2     5    12   -11    -4
```

See Also

`repmat`, `arrayfun`

Purpose

Solve boundary value problems for ordinary differential equations

Syntax

```
sol = bvp4c(odefun,bcfun,solinit)
sol = bvp4c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

Arguments

`odefun`	A function handle that evaluates the differential equations $f(x, y)$. It can have the form
	
	`dydx = odefun(x,y)`
	`dydx = odefun(x,y,parameters)`
	
	where `x` is a scalar corresponding to x, and `y` is a column vector corresponding to y. `parameters` is a vector of unknown parameters. The output `dydx` is a column vector.

`bcfun`	A function handle that computes the residual in the boundary conditions. For two-point boundary value conditions of the form $bc(y(a), y(b))$, `bcfun` can have the form
	
	`res = bcfun(ya,yb)`
	`res = bcfun(ya,yb,parameters)`
	
	where `ya` and `yb` are column vectors corresponding to $y(a)$ and $y(b)$. `parameters` is a vector of unknown parameters. The output `res` is a column vector.
	
	See "Multipoint Boundary Value Problems" on page 2-416 for a description of `bcfun` for multipoint boundary value problems.

`solinit`	A structure containing the initial guess for a solution. You create `solinit` using the function `bvpinit`. `solinit` has the following fields.

	x	Ordered nodes of the initial mesh. Boundary conditions are imposed at a = solinit.x(1) and b = solinit.x(end).
	y	Initial guess for the solution such that solinit.y(:,i) is a guess for the solution at the node solinit.x(i).
	parameters	Optional. A vector that provides an initial guess for unknown parameters.
	The structure can have any name, but the fields must be named x, y, and parameters. You can form solinit with the helper function bvpinit. See bvpinit for details.	
options	Optional integration argument. A structure you create using the bvpset function. See bvpset for details.	

Description

sol = bvp4c(odefun,bcfun,solinit) integrates a system of ordinary differential equations of the form

$$y' = f(x, y)$$

on the interval [a,b] subject to two-point boundary value conditions

$$bc(y(a), y(b)) = 0$$

odefun and bcfun are function handles. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions" in the MATLAB mathematics documentation, explains how to provide additional parameters to the function odefun, as well as the boundary condition function bcfun, if necessary.

bvp4c can also solve multipoint boundary value problems. See "Multipoint Boundary Value Problems" on page 2-416. You can use the function bvpinit to specify the boundary points, which are stored in the input argument solinit. See the reference page for bvpinit for more information.

The bvp4c solver can also find unknown parameters p for problems of the form

$$y' = f(x, y, p)$$
$$0 = bc(y(a), y(b), p)$$

where p corresponds to parameters. You provide bvp4c an initial guess for any unknown parameters in solinit.parameters. The bvp4c solver returns the final values of these unknown parameters in sol.parameters.

bvp4c produces a solution that is continuous on [a,b] and has a continuous first derivative there. Use the function deval and the output sol of bvp4c to evaluate the solution at specific points xint in the interval [a,b].

```
sxint = deval(sol,xint)
```

The structure sol returned by bvp4c has the following fields:

sol.x	Mesh selected by bvp4c
sol.y	Approximation to $y(x)$ at the mesh points of sol.x
sol.yp	Approximation to $y'(x)$ at the mesh points of sol.x
sol.parameters	Values returned by bvp4c for the unknown parameters, if any
sol.solver	'bvp4c'

The structure sol can have any name, and bvp4c creates the fields x, y, yp, parameters, and solver.

sol = bvp4c(odefun,bcfun,solinit,options) solves as above with default integration properties replaced by the values in options, a structure created with the bvpset function. See bvpset for details.

`solinit = bvpinit(x, yinit, params)` forms the initial guess `solinit` with the vector `params` of guesses for the unknown parameters.

Singular Boundary Value Problems

`bvp4c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form

$$y' = S \cdot y/x + f(x, y, p)$$
$$0 = bc(y(0), y(b), p)$$

The interval is required to be [0, *b*] with b > 0. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only *f*(*x*, *y*, *p*). The boundary conditions must be consistent with the necessary condition $S \cdot y(0) = 0$ and the initial guess should satisfy this condition.

Multipoint Boundary Value Problems

`bvp4c` can solve multipoint boundary value problems where $a = a_0 < a_1 < a_2 < \ldots < a_n = b$ are boundary points in the interval $[a, b]$. The points $a_1, a_2, \ldots, a_{n-1}$ represent interfaces that divide $[a, b]$ into regions. `bvp4c` enumerates the regions from left to right (from *a* to *b*), with indices starting from 1. In region *k*, $[a_{k-1}, a_k]$, `bvp4c` evaluates the derivative as

```
yp = odefun(x, y, k)
```

In the boundary conditions function

```
bcfun(yleft, yright)
```

yleft(:, k) is the solution at the left boundary of $[a_{k-1}, a_k]$. Similarly, `yright(:, k)` is the solution at the right boundary of region *k*. In particular,

```
yleft(:, 1) = y(a)
```

and

```
yright(:, end) = y(b)
```

When you create an initial guess with

```
solinit = bvpinit(xinit, yinit),
```

use double entries in `xinit` for each interface point. See the reference page for `bvpinit` for more information.

If `yinit` is a function, `bvpinit` calls `y = yinit(x, k)` to get an initial guess for the solution at `x` in region `k`. In the solution structure `sol` returned by `bpv4c`, `sol.x` has double entries for each interface point. The corresponding columns of `sol.y` contain the left and right solution at the interface, respectively.

For an example of solving a three-point boundary value problem, type `threebvp` at the MATLAB command prompt to run a demonstration.

Note The `bvp5c` function is used exactly like `bvp4c`, with the exception of the meaning of error tolerances between the two solvers. If *S(x)* approximates the solution *y(x)*, `bvp4c` controls the residual $|S'(x) - f(x,S(x))|$. This controls indirectly the true error $|y(x) - S(x)|$. `bvp5c` controls the true error directly. `bvp5c` is more efficient than `bvp4c` for small error tolerances.

Examples

Example 1

Boundary value problems can have multiple solutions and one purpose of the initial guess is to indicate which solution you want. The second-order differential equation

$$y'' + |y| = 0$$

has exactly two solutions that satisfy the boundary conditions

$$y(0) = 0$$
$$y(4) = -2$$

Prior to solving this problem with `bvp4c`, you must write the differential equation as a system of two first-order ODEs

$$y_1' = y_2$$
$$y_2' = -|y_1|$$

Here $y_1 = y$ and $y_2 = y'$. This system has the required form

$$y' = f(x, y)$$
$$bc(y(a), y(b)) = 0$$

The function f and the boundary conditions bc are coded in MATLAB as functions `twoode` and `twobc`.

```
function dydx = twoode(x,y)
  dydx = [ y(2)
           -abs(y(1))];

function res = twobc(ya,yb)
  res = [ ya(1)
          yb(1) + 2];
```

Form a guess structure consisting of an initial mesh of five equally spaced points in [0,4] and a guess of constant values $y_1(x) \equiv 1$ and $y_2(x) \equiv 0$ with the command

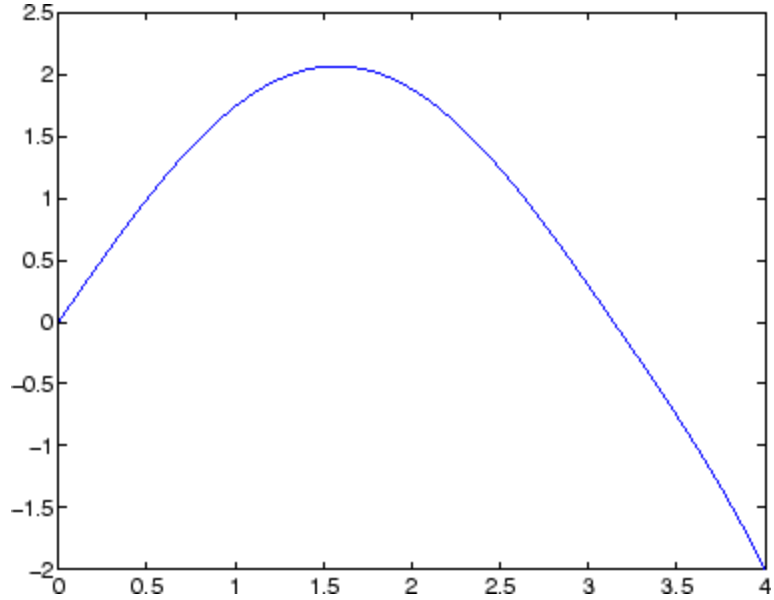
```
solinit = bvpinit(linspace(0,4,5),[1 0]);
```

Now solve the problem with

```
sol = bvp4c(@twoode,@twobc,solinit);
```

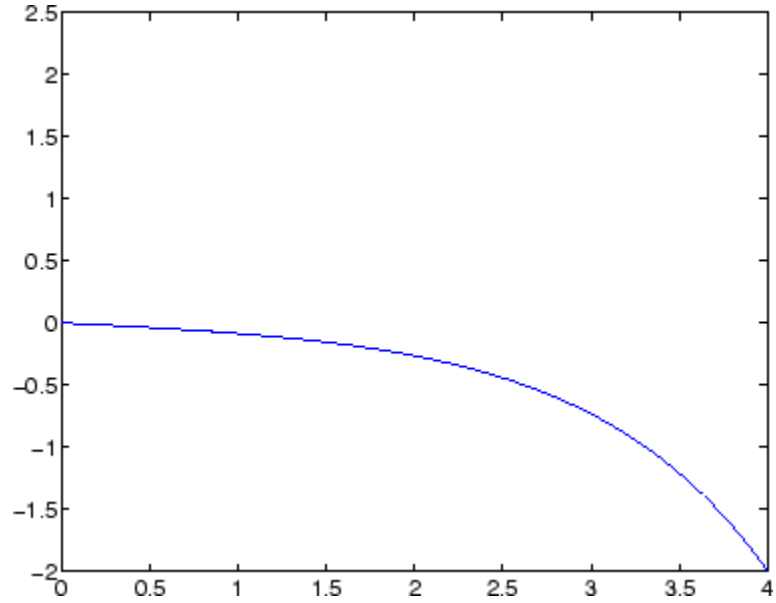
Evaluate the numerical solution at 100 equally spaced points and plot $y(x)$ with

```
x = linspace(0,4);
y = deval(sol,x);
plot(x,y(1,:));
```



You can obtain the other solution of this problem with the initial guess

```
solinit = bvpinit(linspace(0,4,5),[-1 0]);
```



Example 2

This boundary value problem involves an unknown parameter. The task is to compute the fourth ($q = 5$) eigenvalue λ of Mathieu's equation

$$y'' + (\lambda - 2\,q\cos 2x)\,y = 0$$

Because the unknown parameter λ is present, this second-order differential equation is subject to *three* boundary conditions

$$y'(0) = 0$$
$$y'(\pi) = 0$$
$$y(0) = 1$$

It is convenient to use subfunctions to place all the functions required by `bvp4c` in a single M-file.

```
function mat4bvp
```

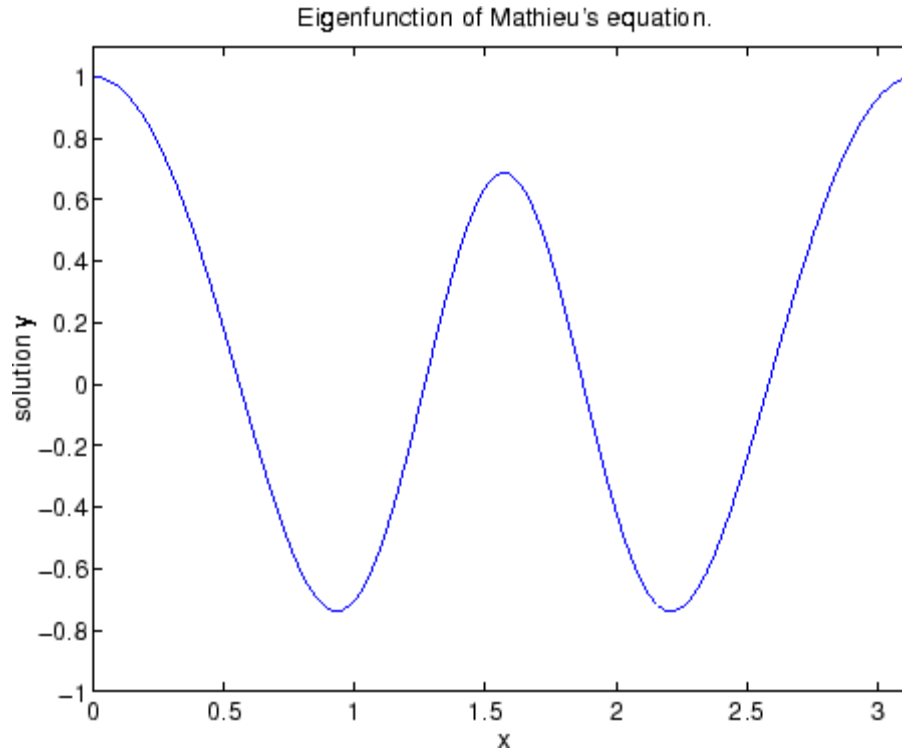
```
lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
sol = bvp4c(@mat4ode,@mat4bc,solinit);

fprintf('The fourth eigenvalue is approximately %7.3f.\n',...
        sol.parameters)

xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
axis([0 pi -1 1.1])
title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')
% --------------------------------------------------------------
function dydx = mat4ode(x,y,lambda)
q = 5;
dydx = [  y(2)
         -(lambda - 2*q*cos(2*x))*y(1) ];
% --------------------------------------------------------------
function res = mat4bc(ya,yb,lambda)
res = [  ya(2)
         yb(2)
        ya(1)-1 ];
% --------------------------------------------------------------
function yinit = mat4init(x)
yinit = [  cos(4*x)
          -4*sin(4*x) ];
```

The differential equation (converted to a first-order system) and the boundary conditions are coded as subfunctions `mat4ode` and `mat4bc`, respectively. Because unknown parameters are present, these functions must accept three input arguments, even though some of the arguments are not used.

The guess structure `solinit` is formed with `bvpinit`. An initial guess for the solution is supplied in the form of a function `mat4init`. We chose

$y = \cos 4x$ because it satisfies the boundary conditions and has the correct qualitative behavior (the correct number of sign changes). In the call to `bvpinit`, the third argument (`lambda = 15`) provides an initial guess for the unknown parameter λ.

After the problem is solved with `bvp4c`, the field `sol.parameters` returns the value $\lambda = 17.097$, and the plot shows the eigenfunction associated with this eigenvalue.



Algorithms

`bvp4c` is a finite difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fourth-order

accurate uniformly in `[a,b]`. Mesh selection and error control are based on the residual of the continuous solution.

References

[1] Shampine, L.F., M.W. Reichelt, and J. Kierzenka, "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c," available at `http://www.mathworks.com/bvp_tutorial`

See Also

`function_handle` (`@`), `bvp5c`,`bvpget`, `bvpinit`, `bvpset`, `bvpxtend`, `deval`

Purpose

Solve boundary value problems for ordinary differential equations

Syntax

```
sol = bvp5c(odefun,bcfun,solinit)
sol = bvp5c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

Arguments

`odefun`	A function handle that evaluates the differential equations $f(x, y)$. It can have the form
	`dydx = odefun(x,y)`
	`dydx = odefun(x,y,parameters)`
	where `x` is a scalar corresponding to x, and `y` is a column vector corresponding to y. `parameters` is a vector of unknown parameters. The output `dydx` is a column vector.	

`bcfun`	A function handle that computes the residual in the boundary conditions. For two-point boundary value conditions of the form $bc(y(a), y(b))$, `bcfun` can have the form
	`res = bcfun(ya,yb)`
	`res = bcfun(ya,yb,parameters)`
	where `ya` and `yb` are column vectors corresponding to $y(a)$ and $y(b)$. `parameters` is a vector of unknown parameters. The output `res` is a column vector.	

`solinit`	A structure containing the initial guess for a solution. You create `solinit` using the function `bvpinit`. `solinit` has the following fields.	
	`x`	Ordered nodes of the initial mesh. Boundary conditions are imposed at a = `solinit.x(1)` and b = `solinit.x(end)`.

	`y`	Initial guess for the solution such that `solinit.y(:,i)` is a guess for the solution at the node `solinit.x(i)`.
	`parameters`	Optional. A vector that provides an initial guess for unknown parameters.
	The structure can have any name, but the fields must be named `x`, `y`, and `parameters`. You can form `solinit` with the helper function `bvpinit`. See `bvpinit` for details.	
`options`	Optional integration argument. A structure you create using the `bvpset` function. See `bvpset` for details.	

Description

`sol = bvp5c(odefun,bcfun,solinit)` integrates a system of ordinary differential equations of the form

$$y' = f(x, y)$$

on the interval [a,b] subject to two-point boundary value conditions

$$bc(y(a), y(b)) = 0$$

`odefun` and `bcfun` are function handles. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parameterizing Functions Called by Function Functions” in the MATLAB mathematics documentation, explains how to provide additional parameters to the function `odefun`, as well as the boundary condition function `bcfun`, if necessary. You can use the function `bvpinit` to specify the boundary points, which are stored in the input argument `solinit`. See the reference page for `bvpinit` for more information.

The `bvp5c` solver can also find unknown parameters p for problems of the form

$$y' = f(x, y, p)$$
$$0 = bc(y(a), y(b), p)$$

where p corresponds to `parameters`. You provide `bvp5c` an initial guess for any unknown parameters in `solinit.parameters`. The `bvp5c` solver returns the final values of these unknown parameters in `sol.parameters`.

`bvp5c` produces a solution that is continuous on `[a,b]` and has a continuous first derivative there. Use the function `deval` and the output `sol` of `bvp5c` to evaluate the solution at specific points `xint` in the interval `[a,b]`.

```
sxint = deval(sol,xint)
```

The structure `sol` returned by `bvp5c` has the following fields:

`sol.x`	Mesh selected by `bvp5c`
`sol.y`	Approximation to $y(x)$ at the mesh points of `sol.x`
`sol.parameters`	Values returned by `bvp5c` for the unknown parameters, if any
`sol.solver`	'bvp5c'

The structure `sol` can have any name, and `bvp5c` creates the fields `x`, `y`, `parameters`, and `solver`.

`sol = bvp5c(odefun,bcfun,solinit,options)` solves as above with default integration properties replaced by the values in `options`, a structure created with the `bvpset` function. See `bvpset` for details.

`solinit = bvpinit(x, yinit, params)` forms the initial guess `solinit` with the vector `params` of guesses for the unknown parameters.

Note The `bvp5c` function is used exactly like `bvp4c`, with the exception of the meaning of error tolerances between the two solvers. If *S(x)* approximates the solution *y(x)*, `bvp4c` controls the residual $|S'(x) - f(x,S(x))|$. This controls indirectly the true error $|y(x) - S(x)|$. `bvp5c` controls the true error directly. `bvp5c` is more efficient than `bvp4c` for small error tolerances.

Singular Boundary Value Problems

`bvp5c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form

$$y' = S \cdot y/x + f(x, y, p)$$
$$0 = bc(y(0), y(b), p)$$

The interval is required to be [0, *b*] with b > 0. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only *f*(*x*, *y*, *p*). The boundary conditions must be consistent with the necessary condition $S \cdot y(0) = 0$ and the initial guess should satisfy this condition.

Algorithms

`bvp5c` is a finite difference code that implements the four-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fifth-order accurate uniformly in `[a,b]`. The formula is implemented as an implicit Runge-Kutta formula. `bvp5c` solves the algebraic equations directly; `bvp4c` uses analytical condensation. `bvp4c` handles unknown parameters directly; while `bvp5c` augments the system with trivial differential equations for unknown parameters.

References

[1] Shampine, L.F., M.W. Reichelt, and J. Kierzenka “Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with

bvp4c" `http://www.mathworks.com/bvp_tutorial`. Note that this tutorial uses the bvp4c function, however in most cases the solvers can be used interchangeably.

See Also

`function_handle` (`@`), `bvp4c`, `bvpget`, `bvpinit`, `bvpset`, `bvpxtend`, `deval`

Purpose

Extract properties from options structure created with `bvpset`

Syntax

```
val = bvpget(options,'name')
val = bvpget(options,'name',default)
```

Description

`val = bvpget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = bvpget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = bvpget(opts,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `opts`.

See Also

`bvp4c`, `bvp5c`, `bvpinit`, `bvpset`, `deval`

bvpinit

Purpose

Form initial guess for `bvp4c`

Syntax

```
solinit = bvpinit(x,yinit)
solinit = bvpinit(x,yinit,parameters)
solinit = bvpinit(sol,[anew bnew])
solinit = bvpinit(sol,[anew bnew],parameters)
```

Description

`solinit = bvpinit(x,yinit)` forms the initial guess for the boundary value problem solver `bvp4c`.

`x` is a vector that specifies an initial mesh. If you want to solve the boundary value problem (BVP) on $[a, b]$, then specify `x(1)` as a and `x(end)` as b. The function `bvp4c` adapts this mesh to the solution, so a guess like `xb=nlinspace(a,b,10)` often suffices. However, in difficult cases, you should place mesh points where the solution changes rapidly. The entries of `x` must be in

- Increasing order if $a < b$
- Decreasing order if $a > b$

For two-point boundary value problems, the entries of `x` must be distinct. That is, if $a < b$, the entries must satisfy `x(1) < x(2) < ... < x(end)`. If $a > b$, the entries must satisfy `x(1) > x(2) > ... > x(end)`

For multipoint boundary value problem, you can specify the points in $[a, b]$ at which the boundary conditions apply, other than the endpoints *a* and *b*, by repeating their entries in `x`. For example, if you set

```
x = [0, 0.5, 1, 1,  1.5, 2];
```

the boundary conditions apply at three points: the endpoints `0` and `2`, and the repeated entry 1. In general, repeated entries represent boundary points between regions in $[a, b]$. In the preceding example, the repeated entry 1 divides the interval `[0,2]` into two regions: `[0,1]` and `[1,2]`.

`yinit` is a guess for the solution. It can be either a vector, or a function:

- Vector – For each component of the solution, `bvpinit` replicates the corresponding element of the vector as a constant guess across all mesh points. That is, `yinit(i)` is a constant guess for the `i`th component `yinit(i,:)` of the solution at all the mesh points in `x`.
- Function – For a given mesh point, the guess function must return a vector whose elements are guesses for the corresponding components of the solution. The function must be of the form

  ```
  y = guess(x)
  ```

  where `x` is a mesh point and `y` is a vector whose length is the same as the number of components in the solution. For example, if the guess function is an M-file function, `bvpinit` calls

  ```
  y(:,j) = guess(x(j))
  ```

  at each mesh point.

  For multipoint boundary value problems, the guess function must be of the form

  ```
  y = guess(x, k)
  ```

  where `y` an initial guess for the solution at `x` in region `k`. The function must accept the input argument `k`, which is provided for flexibility in writing the guess function. However, the function is not required to use `k`.

`solinit = bvpinit(x,yinit,parameters)` indicates that the boundary value problem involves unknown parameters. Use the vector `parameters` to provide a guess for all unknown parameters.

`solinit` is a structure with the following fields. The structure can have any name, but the fields must be named `x`, `y`, and `parameters`.

`x`	Ordered nodes of the initial mesh.
`y`	Initial guess for the solution with `solinit.y(:,i)` a guess for the solution at the node `solinit.x(i)`.
`parameters`	Optional. A vector that provides an initial guess for unknown parameters.

`solinit = bvpinit(sol,[anew bnew])` forms an initial guess on the interval `[anew bnew]` from a solution `sol` on an interval $[a, b]$. The new interval must be larger than the previous one, so either `anew <=`a `<`b `<= bnew` or `anew >=`a `>`b `>= bnew`. The solution `sol` is extrapolated to the new interval. If `sol` contains `parameters`, they are copied to `solinit`.

`solinit = bvpinit(sol,[anew bnew],parameters)` forms `solinit` as described above, but uses `parameters` as a guess for unknown parameters in `solinit`.

See Also

`@(function_handle)`, `bvp4c`,`bvp5c`, `bvpget`, `bvpset`, `bvpxtend`, `deval`

Purpose

Create or alter options structure of boundary value problem

Syntax

```
options = bvpset('name1',value1,'name2',value2,...)
options = bvpset(oldopts,'name1',value1,...)
options = bvpset(oldopts,newopts)
bvpset
```

Description

`options = bvpset('name1',value1,'name2',value2,...)` creates a structure `options` that you can supply to the boundary value problem solver `bvp4c`, in which the named properties have the specified values. Any unspecified properties retain their default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property. `bvpset` ignores case for property names.

`options = bvpset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = bvpset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`bvpset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `bvpget` to query the `options` structure for the value of a specific property.

BVP Properties

`bvpset` enables you to specify properties for the boundary value problem solver `bvp4c`. There are several categories of properties that you can set:

- "Error Tolerance Properties" on page 2-434
- "Vectorization" on page 2-435
- "Analytical Partial Derivatives" on page 2-436
- "Singular BVPs" on page 2-439

Error Tolerance Properties

Because `bvp4c` uses a collocation formula, the numerical solution is based on a mesh of points at which the collocation equations are satisfied. Mesh selection and error control are based on the residual of this solution, such that the computed solution $S(x)$ is the exact solution of a perturbed problem $S'(x) = f(x, S(x)) + res(x)$. On each subinterval of the mesh, a norm of the residual in the `i`th component of the solution, `res(i)`, is estimated and is required to be less than or equal to a tolerance. This tolerance is a function of the relative and absolute tolerances, `RelTol` and `AbsTol`, defined by the user.

$$\| (\text{res(i)}/\max(\text{abs(f(i))}, \text{AbsTol(i)}/\text{RelTol})) \| \leq \text{RelTol}$$

The following table describes the error tolerance properties.

BVP Error Tolerance Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the residual vector. It is a measure of the residual relative to the size of $f(x, y)$. The default, `1e-3`, corresponds to 0.1% accuracy. The computed solution $S(x)$ is the exact solution of $S'(x) = F(x, S(x)) + \mathtt{res}(x)$. On each subinterval of the mesh, the residual $\mathtt{res}(x)$ satisfies $\|(\mathrm{res(i)/max(abs(F(i)),AbsTol(i)/RelTol))}\| \le \mathrm{RelTol}$
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the corresponding components of the residual vector. `AbsTol(i)` is a threshold below which the values of the corresponding components are unimportant. If a scalar value is specified, it applies to all components.

Vectorization

The following table describes the BVP vectorization property. Vectorization of the ODE function used by `bvp4c` differs from the vectorization used by the ODE solvers:

- For `bvp4c`, the ODE function must be vectorized with respect to the first argument as well as the second one, so that `F([x1 x2 ...],[y1 y2 ...])` returns `[F(x1,y1) F(x2,y2)...]`.
- `bvp4c` benefits from vectorization even when analytical Jacobians are provided. For stiff ODE solvers, vectorization is ignored when analytical Jacobians are used.

Vectorization Properties

Property	Value	Description

`Vectorized`	`on` | {`off`}	Set on to inform `bvp4c` that you have coded the ODE function F so that `F([x1 x2 ...],[y1 y2 ...])` returns `[F(x1,y1) F(x2,y2) ...]`. That is, your ODE function can pass to the solver a whole array of column vectors at once. This enables the solver to reduce the number of function evaluations and may significantly reduce solution time.
		
		With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. In the `shockbvp` example shown previously, the shockODE function has been vectorized using colon notation into the subscripts and by using the array multiplication (.*) operator.
		
		`function dydx = shockODE(x,y,e)`
		`pix = pi*x;`
		`dydx = [ y(2,:)...`
		`-x/e.*y(2,:)-pi^2*cos(pix)-`
		`pix/e.*sin(pix)];`

Analytical Partial Derivatives

By default, the `bvp4c` solver approximates all partial derivatives with finite differences. `bvp4c` can be more efficient if you provide analytical partial derivatives $\partial f / \partial y$ of the differential equations,

and analytical partial derivatives, $\partial bc/\partial ya$ and $\partial bc/\partial yb$, of the boundary conditions. If the problem involves unknown parameters, you must also provide partial derivatives, $\partial f/\partial p$ and $\partial bc/\partial p$, with respect to the parameters.

The following table describes the analytical partial derivatives properties.

BVP Analytical Partial Derivative Properties

Property	Value	Description
`FJacobian`	Function handle	Handle to a function that computes the analytical partial derivatives of $f(x, y)$. When solving $y' = f(x, y)$, set this property to `@fjac` if `dfdy = fjac(x,y)` evaluates the Jacobian $\partial f / \partial y$. If the problem involves unknown parameters p, `[dfdy,dfdp] = fjac(x,y,p)` must also return the partial derivative $\partial f / \partial p$. For problems with constant partial derivatives, set this property to the value of `dfdy` or to a cell array `{dfdy,dfdp}`. See "Function Handles" in the MATLAB Programming documentation for more information.
`BCJacobian`	Function handle	Handle to a function that computes the analytical partial derivatives of $bc(ya, yb)$. For boundary conditions $bc(ya, yb)$, set this property to `@bcjac` if `[dbcdya,dbcdyb] = bcjac(ya,yb)` evaluates the partial derivatives $\partial bc / \partial ya$, and $\partial bc / \partial yb$. If the problem involves unknown parameters p, `[dbcdya,dbcdyb,dbcdp] = bcjac(ya,yb,p)` must also return the partial derivative $\partial bc / \partial p$. For problems with constant partial derivatives, set this property to a cell array `{dbcdya,dbcdyb}` or `{dbcdya,dbcdyb,dbcdp}`.

Singular BVPs

bvp4c can solve singular problems of the form

$$y' = S\frac{y}{x} + f(x, y, p)$$

posed on the interval $[0, b]$ where $b > 0$. For such problems, specify the constant matrix S as the value of SingularTerm. For equations of this form, odefun evaluates only the $f(x, y, p)$ term, where p represents unknown parameters, if any.

Singular BVP Property

Property	Value	Description
SingularTerm	Constant matrix	Singular term of singular BVPs. Set to the constant matrix S for equations of the form $y' = S\frac{y}{x} + f(x, y, p)$ posed on the interval $[0, b]$ where $b > 0$.

Mesh Size Property

bvp4c solves a system of algebraic equations to determine the numerical solution to a BVP at each of the mesh points. The size of the algebraic system depends on the number of differential equations (n) and the number of mesh points in the current mesh (N). When the allowed number of mesh points is exhausted, the computation stops, bvp4c displays a warning message and returns the solution it found so far. This solution does not satisfy the error tolerance, but it may provide an

excellent initial guess for computations restarted with relaxed error tolerances or an increased value of NMax.

The following table describes the mesh size property.

BVP Mesh Size Property

Property	Value	Description
NMax	positive integer {floor(1000/n)}	Maximum number of mesh points allowed when solving the BVP, where n is the number of differential equations in the problem. The default value of NMax limits the size of the algebraic system to about 1000 equations. For systems of a few differential equations, the default value of NMax should be sufficient to obtain an accurate solution.

Solution Statistic Property

The Stats property lets you view solution statistics.

The following table describes the solution statistics property.

BVP Solution Statistic Property

Property	Value	Description
`Stats`	`on` | {`off`}	Specifies whether statistics about the computations are displayed. If the `stats` property is `on`, after solving the problem, `bvp4c` displays: • The number of points in the mesh • The maximum residual of the solution • The number of times it called the differential equation function `odefun` to evaluate $f(x, y)$ • The number of times it called the boundary condition function `bcfun` to evaluate $bc(y(a), y(b))$

Example

To create an options structure that changes the relative error tolerance of `bvp4c` from the default value of `1e-3` to `1e-4`, enter

```
options = bvpset('RelTol', 1e-4);
```

To recover the value of `'RelTol'` from `options`, enter

```
bvpget(options, 'RelTol')

ans =

   1.0000e-004
```

See Also

`@` (function_handle), `bvp4c`,`bvp5c`, `bvpget`, `bvpinit`, `deval`

bvpxtend

Purpose

Form guess structure for extending boundary value solutions

Syntax

```
solinit = bvpxtend(sol,xnew,ynew)
solinit = bvpxtend(sol,xnew,extrap)
solinit = bvpxtend(sol,xnew)
solinit = bvpxtend(sol,xnew,ynew,pnew)
solinit = bvpxtend(sol,xnew,extrap,pnew)
```

Description

`solinit = bvpxtend(sol,xnew,ynew)` uses solution `sol` computed on `[a,b]` to form a solution guess for the interval extended to `xnew`. The extension point `xnew` must be outside the interval `[a,b]`, but on either side. The vector `ynew` provides an initial guess for the solution at `xnew`.

`solinit = bvpxtend(sol,xnew,extrap)` forms the guess at `xnew` by extrapolating the solution `sol`. `extrap` is a string that determines the extrapolation method. `extrap` has three possible values:

- `'constant'` — `ynew` is a value nearer to end point of solution in `sol`.
- `'linear'` — `ynew` is a value at `xnew` of linear interpolant to the value and slope at the nearer end point of solution in `sol`.
- `'solution'` — `ynew` is the value of (cubic) solution in `sol` at `xnew`.

The value of `extrap` is case-insensitive and only the leading, unique portion needs to be specified.

`solinit = bvpxtend(sol,xnew)` uses the extrapolating solution where `extrap` is `'constant'`. If there are unknown parameters, values present in `sol` are used as the initial guess for parameters in `solinit`.

`solinit = bvpxtend(sol,xnew,ynew,pnew)` specifies a different guess `pnew`. `pnew` can be used with extrapolation, using the syntax `solinit = bvpxtend(sol,xnew,extrap,pnew)`. To modify parameters without changing the interval, use `[]` as place holder for `xnew` and `ynew`.

See Also

`bvp4c`, `bvp5c`, `bvpinit`

Purpose

Calendar for specified month

Syntax

```
c = calendar
c = calendar(d)
c = calendar(y, m)
```

Description

`c = calendar` returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.

`c = calendar(d)`, where `d` is a serial date number or a date string, returns a calendar for the specified month.

`c = calendar(y, m)`, where `y` and `m` are integers, returns a calendar for the specified month of the specified year.

Examples

The command

```
calendar(1957,10)
```

reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

```
                Oct 1957
   S     M     Tu     W     Th     F     S
   0     0      1     2      3     4     5
   6     7      8     9     10    11    12
  13    14     15    16     17    18    19
  20    21     22    23     24    25    26
  27    28     29    30     31     0     0
   0     0      0     0      0     0     0
```

See Also

`datenum`

Purpose

Call function in external library

Syntax

```
[x1, ..., xN] = calllib('libname', 'funcname', arg1, ...,
   argN)
```

Description

`[x1, ..., xN] = calllib('libname', 'funcname', arg1, ..., argN)` calls the function `funcname` in library `libname`, passing input arguments `arg1` through `argN`. `calllib` returns output values obtained from function `funcname` in `x1` through `XN`.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

Ways to Call calllib

The following examples show ways calls to `calllib`. By using `libfunctionsview`, you determined that the `addStructByRef` function in the shared library `shrlibsample` requires a pointer to a `c_struct` data type as its argument.

Load the library:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h
```

Create a MATLAB structure and use `libstruct` to create a C structure of the proper type (`c_struct` here):

```
struct.p1 = 4; struct.p2 = 7.3; struct.p3 = -290;
[res,st] = calllib('shrlibsample','addStructByRef',...
 libstruct('c_struct',struct));
```

Let MATLAB convert `struct` to the proper type of C structure:

```
[res,st] = calllib('shrlibsample','addStructByRef',struct);
```

Pass an empty array to `libstruct` and assign the values from your C function:

```
[res,st] = calllib('shrlibsample','addStructByRef',...
```

```
libstruct('c_struct',[]);
```

Let MATLAB create the proper type of structure and assign values from your C function:

```
[res,st] = calllib('shrlibsample','addStructByRef',[]);
```

Examples

This example calls functions from the `libmx` library to test the value stored in `y`:

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile)

y = rand(4, 7, 2);

calllib('libmx', 'mxGetNumberOfElements', y)
ans =
    56

calllib('libmx', 'mxGetClassID', y)
ans =
   mxDOUBLE_CLASS

unloadlibrary libmx
```

See Also

`loadlibrary`, `libfunctions`, `libfunctionsview`, `libpointer`, `libstruct`, `libisloaded`, `unloadlibrary`

See Passing Arguments for information on defining the correct data types for library function arguments.

callSoapService

Purpose

Send SOAP message off to endpoint

Syntax

```
callSoapService(endpoint, soapAction, message)
```

Description

`callSoapService(endpoint, soapAction, message)` sends `message`, a Java document object model (DOM), to the `soapAction` service at the `endpoint`.

Example

```
message = createSoapMessage(...
'urn:xmethods-delayed-quotes','getQuote',{'GOOG'},{'symbol'},...
{'{http://www.w3.org/2001/XMLSchema}string'},'rpc')
response = callSoapService('http://64.124.140.30:9090/soap',...
'urn:xmethods-delayed-quotes#getQuote',message)
price = parseSoapResponse(response)
```

See Also

`createClassFromWsdl`, `CreateSoapMessage`, `parseSoapResponse`

Purpose

Move camera position and target

Syntax

```
camdolly(dx,dy,dz)
camdolly(dx,dy,dz,'targetmode')
camdolly(dx,dy,dz,'targetmode','coordsys')
camdolly(axes_handle,...)
```

Description

`camdolly` moves the camera position and the camera target by the specified amounts.

`camdolly(dx,dy,dz)` moves the camera position and the camera target by the specified amounts (see Coordinate Systems).

`camdolly(dx,dy,dz,'targetmode')` The *`targetmode`* argument can take on two values that determine how MATLAB moves the camera:

- `movetarget` (default) — Move both the camera and the target.
- `fixtarget` — Move only the camera.

`camdolly(dx,dy,dz,'targetmode','coordsys')` The *`coordsys`* argument can take on three values that determine how MATLAB interprets `dx`, `dy`, and `dz`:

Coordinate Systems

- `camera` (default) — Move in the camera's coordinate system. `dx` moves left/right, `dy` moves down/up, and `dz` moves along the viewing axis. The units are normalized to the scene.

  For example, setting `dx` to 1 moves the camera to the right, which pushes the scene to the left edge of the box formed by the axes position rectangle. A negative value moves the scene in the other direction. Setting `dz` to 0.5 moves the camera to a position halfway between the camera position and the camera target.

- `pixels` — Interpret `dx` and `dy` as pixel offsets. `dz` is ignored.
- `data` — Interpret `dx`, `dy`, and `dz` as offsets in axes data coordinates.

`camdolly(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camdolly` operates on the current axes.

Remarks

`camdolly` sets the axes `CameraPosition` and`CameraTarget` properties, which in turn causes the `CameraPositionMode` and `CameraTargetMode` properties to be set to `manual`.

Examples

This example moves the camera along the *x*- and *y*-axes in a series of steps.

```
surf(peaks)
axis vis3d
t = 0:pi/20:2*pi;
dx = sin(t)./40;
dy = cos(t)./40;
for i = 1:length(t);
    camdolly(dx(i),dy(i),0)
    drawnow
end
```

See Also

`axes`, `campos`, `camproj`, `camtarget`, `camup`, `camva`

The axes properties `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`, `Projection`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

See "Defining Scenes with Camera Graphics" for more information on camera properties.

Purpose

Control camera toolbar programmatically

Syntax

```
cameratoolbar
cameratoolbar('NoReset')
cameratoolbar('SetMode',mode)
cameratoolbar('SetCoordSys',coordsys)
cameratoolbar('Show')
cameratoolbar('Hide')
cameratoolbar('Toggle')
cameratoolbar('ResetCameraAndSceneLight')
cameratoolbar('ResetCamera')
cameratoolbar('ResetSceneLight')
cameratoolbar('ResetTarget')
mode = cameratoolbar('GetMode')
paxis = cameratoolbar('GetCoordsys')
vis = cameratoolbar('GetVisible')
cameratoolbar(fig,...)
h = cameratoolbar
cameratoolbar('Close')
```

Description

`cameratoolbar` creates a new toolbar that enables interactive manipulation of the axes camera and light when users drag the mouse on the figure window. Several axes camera properties are set when the toolbar is initialized.

`cameratoolbar('NoReset')` creates the toolbar without setting any camera properties.

`cameratoolbar('SetMode',mode)` sets the toolbar mode (depressed button). *`mode`* can be `'orbit'`, `'orbitscenelight'`, `'pan'`, `'dollyhv'`, `'dollyfb'`, `'zoom'`, `'roll'`, `'nomode'`.

`cameratoolbar('SetCoordSys',coordsys)` sets the principal axis of the camera motion. *`coordsys`* can be: `'x'`, `'y'`, `'z'`, `'none'`.

`cameratoolbar('Show')` shows the toolbar on the current figure.

`cameratoolbar('Hide')` hides the toolbar on the current figure.

`cameratoolbar('Toggle')` toggles the visibility of the toolbar.

`cameratoolbar('ResetCameraAndSceneLight')` resets the current camera and scenelight.

`cameratoolbar('ResetCamera')` resets the current camera.

`cameratoolbar('ResetSceneLight')` resets the current scenelight.

`cameratoolbar('ResetTarget')` resets the current camera target.

`mode = cameratoolbar('GetMode')` returns the current mode.

`paxis = cameratoolbar('GetCoordsys')` returns the current principal axis.

`vis = cameratoolbar('GetVisible')` returns the visibility of the toolbar (`1` if visible, `0` if not visible).

`cameratoolbar(fig,...)` specifies the figure to operate on by passing the figure handle as the first argument.

`h = cameratoolbar` returns the handle to the toolbar.

`cameratoolbar('Close')` removes the toolbar from the current figure.

Note that, in general, the use of OpenGL hardware improves rendering performance.

See Also

`rotate3d`, `zoom`

Purpose

Create or move light object in camera coordinates

Syntax

```
camlight('headlight')
camlight('right')
camlight('left')
camlight
camlight(az,el)
camlight(...,'style')
camlight(light_handle,...)
light_handle = camlight(...)
```

Description

`camlight('headlight')` creates a light at the camera position.

`camlight('right')` creates a light right and up from camera.

`camlight('left')` creates a light left and up from camera.

`camlight` with no arguments is the same as `camlight('right')`.

`camlight(az,el)` creates a light at the specified azimuth (`az`) and elevation (`el`) with respect to the camera position. The camera target is the center of rotation and `az` and `el` are in degrees.

`camlight(...,'style')` The style argument can take on two values:

- `local` (default) — The light is a point source that radiates from the location in all directions.
- `infinite` — The light shines in parallel rays.

`camlight(light_handle,...)` uses the light specified in `light_handle`.

`light_handle = camlight(...)` returns the light's handle.

Remarks

`camlight` sets the light object `Position` and `Style` properties. A light created with `camlight` will not track the camera. In order for the light to stay in a constant position relative to the camera, you must call `camlight` whenever you move the camera.

Examples

This example creates a light positioned to the left of the camera and then repositions the light each time the camera is moved:

```
surf(peaks)
axis vis3d
h = camlight('left');
for i = 1:20;
 camorbit(10,0)
 camlight(h,'left')
 drawnow;
end
```

See Also

`light`, `lightangle`

"Lighting" on page 1-101 for related functions

"Lighting as a Visualization Tool" for more information on using lights

Purpose

Position camera to view object or group of objects

Syntax

```
camlookat(object_handles)
camlookat(axes_handle)
camlookat
```

Description

`camlookat(object_handles)` views the objects identified in the vector `object_handles`. The vector can contain the handles of axes children.

`camlookat(axes_handle)` views the objects that are children of the axes identified by `axes_handle`.

`camlookat` views the objects that are in the current axes.

Remarks

`camlookat` moves the camera position and camera target while preserving the relative view direction and camera view angle. The object (or objects) being viewed roughly fill the axes position rectangle.

`camlookat` sets the axes `CameraPosition` and `CameraTarget` properties.

Examples

This example creates three spheres at different locations and then progressively positions the camera so that each sphere is the object around which the scene is composed:

```
[x y z] = sphere;
s1 = surf(x,y,z);
hold on
s2 = surf(x+3,y,z+3);
s3 = surf(x,y,z+6);
daspect([1 1 1])
view(30,10)
camproj perspective
camlookat(gca) % Compose the scene around the current axes
pause(2)
camlookat(s1)  % Compose the scene around sphere s1
pause(2)
camlookat(s2)  % Compose the scene around sphere s2
pause(2)
```

```
camlookat(s3)  % Compose the scene around sphere s3
pause(2)
camlookat(gca)
```

See Also

`campos`, `camtarget`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

"Defining Scenes with Camera Graphics" for more information

Purpose

Rotate camera position around camera target

Syntax

```
camorbit(dtheta,dphi)
camorbit(dtheta,dphi,'coordsys')
camorbit(dtheta,dphi,'coordsys','direction')
camorbit(axes_handle,...)
```

Description

`camorbit(dtheta,dphi)` rotates the camera position around the camera target by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`camorbit(dtheta,dphi,'coordsys')` The *`coordsys`* argument determines the center of rotation. It can take on two values:

- `data` (default) — Rotate the camera around an axis defined by the camera target and the `direction` (default is the positive *z* direction).
- `camera` — Rotate the camera about the point defined by the camera target.

`camorbit(dtheta,dphi,'coordsys','direction')` The `direction` argument, in conjunction with the camera target, defines the axis of rotation for the data coordinate system. Specify `direction` as a three-element vector containing the `x`, `y`, and `z` components of the direction or one of the characters, `x`, `y`, or `z`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`camorbit(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camorbit` operates on the current axes.

Examples

Compare rotation in the two coordinate systems with these `for` loops. The first rotates the camera horizontally about a line defined by the camera target point and a direction that is parallel to the *y*-axis. Visualize this rotation as a cone formed with the camera target at the apex and the camera position forming the base:

```
surf(peaks)
```

```
axis vis3d
for i=1:36
 camorbit(10,0,'data',[0 1 0])
 drawnow
end
```

Rotation in the `camera` coordinate system orbits the camera around the axes along a circle while keeping the center of a circle at the camera target.

```
surf(peaks)
axis vis3d
for i=1:36
 camorbit(10,0,'camera')
 drawnow
end
```

See Also

`axes`, `axis('vis3d')`, `camdolly`, `campan`, `camzoom`, `camroll`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

"Defining Scenes with Camera Graphics" for more information

Purpose

Rotate camera target around camera position

Syntax

```
campan(dtheta,dphi)
campan(dtheta,dphi,'coordsys')
campan(dtheta,dphi,'coordsys','direction')
campan(axes_handle,...)
```

Description

`campan(dtheta,dphi)` rotates the camera target around the camera position by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`campan(dtheta,dphi,'coordsys')` The `coordsys` argument determines the center of rotation. It can take on two values:

- `data` (default) — Rotate the camera target around an axis defined by the camera position and the `direction` (default is the positive *z* direction)
- `camera` — Rotate the camera about the point defined by the camera target.

`campan(dtheta,dphi,'coordsys','direction')` The `direction` argument, in conjunction with the camera position, defines the axis of rotation for the data coordinate system. Specify `direction` as a three-element vector containing the x, y, and z components of the direction or one of the characters, `x`, `y`, or `z`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`campan(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `campan` operates on the current axes.

See Also

`axes`, `camdolly`, `camorbit`, `camtarget`, `camzoom`, `camroll`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

"Defining Scenes with Camera Graphics" for more information

campos

Purpose

Set or query camera position

Syntax

```
campos
campos([camera_position])
campos('mode')
campos('auto')
campos('manual')
campos(axes_handle,...)
```

Description

`campos` with no arguments returns the camera position in the current axes.

`campos([camera_position])` sets the position of the camera in the current axes to the specified value. Specify the position as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`campos('mode')` returns the value of the camera position mode, which can be either `auto` (the default) or `manual`.

`campos('auto')` sets the camera position mode to `auto`.

`campos('manual')` sets the camera position mode to `manual`.

`campos(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `campos` operates on the current axes.

Remarks

`campos` sets or queries values of the axes `CameraPosition` and `CameraPositionMode` properties. The camera position is the point in the Cartesian coordinate system of the axes from which you view the scene.

Examples

This example moves the camera along the *x*-axis in a series of steps:

```
surf(peaks)
axis vis3d off
for x = -200:5:200
    campos([x,5,10])
    drawnow
```

```
end
```

See Also

`axis`, `camproj`, `camtarget`, `camup`, `camva`

The axes properties `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`, `Projection`

“Controlling the Camera Viewpoint” on page 1-99 for related functions

“Defining Scenes with Camera Graphics” for more information

camproj

Purpose

Set or query projection type

Syntax

```
camproj
camproj('projection_type')
camproj(axes_handle,...)
```

Description

The projection type determines whether MATLAB uses a perspective or orthographic projection for 3-D views.

`camproj` with no arguments returns the projection type setting in the current axes.

`camproj('projection_type')` sets the projection type in the current axes to the specified value. Possible values for *`projection_type`* are `orthographic` and `perspective`.

`camproj(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camproj` operates on the current axes.

Remarks

`camproj` sets or queries values of the axes object `Projection` property.

See Also

`campos`, `camtarget`, `camup`, `camva`

The axes properties `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`, `Projection`

“Controlling the Camera Viewpoint” on page 1-99 for related functions

“Defining Scenes with Camera Graphics” for more information

Purpose

Rotate camera about view axis

Syntax

```
camroll(dtheta)
camroll(axes_handle,dtheta)
```

Description

`camroll(dtheta)` rotates the camera around the camera viewing axis by the amounts specified in `dtheta` (in degrees). The viewing axis is defined by the line passing through the camera position and the camera target.

`camroll(axes_handle,dtheta)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camroll` operates on the current axes.

Remarks

`camroll` sets the axes `CameraUpVector` property and thereby also sets the `CameraUpVectorMode` property to `manual`.

See Also

`axes`, `axis('vis3d')`, `camdolly`, `camorbit`, `camzoom`, `campan`

“Controlling the Camera Viewpoint” on page 1-99 for related functions

“Defining Scenes with Camera Graphics” for more information

camtarget

Purpose

Set or query location of camera target

Syntax

```
camtarget
camtarget([camera_target])
camtarget('mode')
camtarget('auto')
camtarget('manual')
camtarget(axes_handle,...)
```

Description

The camera target is the location in the axes that the camera points to. The camera remains oriented toward this point regardless of its position.

`camtarget` with no arguments returns the location of the camera target in the current axes.

`camtarget([camera_target])` sets the camera target in the current axes to the specified value. Specify the target as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`camtarget('mode')` returns the value of the camera target mode, which can be either `auto` (the default) or `manual`.

`camtarget('auto')` sets the camera target mode to `auto`.

`camtarget('manual')` sets the camera target mode to `manual`.

`camtarget(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camtarget` operates on the current axes.

Remarks

`camtarget` sets or queries values of the axes object `CameraTarget` and `CameraTargetMode` properties.

When the camera target mode is `auto`, MATLAB positions the camera target at the center of the axes plot box.

Examples

This example moves the camera position and the camera target along the *x*-axis in a series of steps:

```
surf(peaks);
axis vis3d
xp = linspace(-150,40,50);
xt = linspace(25,50,50);
for i=1:50
     campos([xp(i),25,5]);
     camtarget([xt(i),30,0])
     drawnow
end
```

See Also

`axis`, `camproj`, `campos`, `camup`, `camva`

The axes properties `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`, `Projection`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

"Defining Scenes with Camera Graphics" for more information

camup

Purpose

Set or query camera up vector

Syntax

```
camup
camup([up_vector])
camup('mode')
camup('auto')
camup('manual')
camup(axes_handle,...)
```

Description

The camera up vector specifies the direction that is oriented up in the scene.

`camup` with no arguments returns the camera up vector setting in the current axes.

`camup([up_vector])` sets the up vector in the current axes to the specified value. Specify the up vector as *x*, *y*, and *z* components. See Remarks.

`camup('mode')` returns the current value of the camera up vector mode, which can be either `auto` (the default) or `manual`.

`camup('auto')` sets the camera up vector mode to `auto`. In `auto` mode, MATLAB uses a value for the up vector of `[0 1 0]` for 2-D views. This means the *z*-axis points up.

`camup('manual')` sets the camera up vector mode to `manual`. In `manual` mode, MATLAB does not change the value of the camera up vector.

`camup(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camup` operates on the current axes.

Remarks

`camup` sets or queries values of the axes object `CameraUpVector` and `CameraUpVectorMode` properties.

Specify the camera up vector as the *x*-, *y*-, and *z*-coordinates of a point in the axes coordinate system that forms the directed line segment PQ, where P is the point (0,0,0) and Q is the specified *x*-, *y*-, and *z*-coordinates. This line always points up. The length of the line PQ has

no effect on the orientation of the scene. This means a value of `[0 0 1]` produces the same results as `[0 0 25]`.

See Also

`axis`, `camproj`, `campos`, `camtarget`, `camva`

The axes properties `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`, `Projection`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

"Defining Scenes with Camera Graphics" for more information

camva

Purpose

Set or query camera view angle

Syntax

```
camva
camva(view_angle)
camva('mode')
camva('auto')
camva('manual')
camva(axes_handle,...)
```

Description

The camera view angle determines the field of view of the camera. Larger angles produce a smaller view of the scene. You can implement zooming by changing the camera view angle.

`camva` with no arguments returns the camera view angle setting in the current axes.

`camva(view_angle)` sets the view angle in the current axes to the specified value. Specify the view angle in degrees.

`camva('mode')` returns the current value of the camera view angle mode, which can be either `auto` (the default) or `manual`. See Remarks.

`camva('auto')` sets the camera view angle mode to `auto`.

`camva('manual')` sets the camera view angle mode to `manual`. See Remarks.

`camva(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camva` operates on the current axes.

Remarks

`camva` sets or queries values of the axes object `CameraViewAngle` and `CameraViewAngleMode` properties.

When the camera view angle mode is `auto`, MATLAB adjusts the camera view angle so that the scene fills the available space in the window. If you move the camera to a different position, MATLAB changes the camera view angle to maintain a view of the scene that fills the available area in the window.

Setting a camera view angle or setting the camera view angle to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the camera view angle to its current value,

```
camva(camva)
```

can cause a change in the way the graph looks. See the Remarks section of the `axes` reference page for more information.

Examples

This example creates two pushbuttons, one that zooms in and another that zooms out.

```
uicontrol('Style','pushbutton',...
  'String','Zoom In',...
  'Position',[20 20 60 20],...
  'Callback','if camva <= 1;return;else;camva(camva-1);end');
uicontrol('Style','pushbutton',...
  'String','Zoom Out',...
  'Position',[100 20 60 20],...
  'Callback','if camva >= 179;return;else;camva(camva+1);end');
```

Now create a graph to zoom in and out on:

```
surf(peaks);
```

Note the range checking in the callback statements. This keeps the values for the camera view angle in the range greater than zero and less than 180.

See Also

`axis`, `camproj`, `campos`, `camup`, `camtarget`

The axes properties `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`, `Projection`

“Controlling the Camera Viewpoint” on page 1-99 for related functions

“Defining Scenes with Camera Graphics” for more information

camzoom

Purpose

Zoom in and out on scene

Syntax

```
camzoom(zoom_factor)
camzoom(axes_handle,...)
```

Description

`camzoom(zoom_factor)` zooms in or out on the scene depending on the value specified by `zoom_factor`. If `zoom_factor` is greater than 1, the scene appears larger; if `zoom_factor` is greater than zero and less than 1, the scene appears smaller.

`camzoom(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camzoom` operates on the current axes.

Remarks

`camzoom` sets the axes `CameraViewAngle` property, which in turn causes the `CameraViewAngleMode` property to be set to `manual`. Note that setting the `CameraViewAngle` property disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This may result in a change to the aspect ratio of your graph. See the `axes` function for more information on this behavior.

See Also

`axes`, `camdolly`, `camorbit`, `campan`, `camroll`, `camva`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

"Defining Scenes with Camera Graphics" for more information

Purpose

Transform Cartesian coordinates to polar or cylindrical

Syntax

```
[THETA,RHO,Z] = cart2pol(X,Y,Z)
[THETA,RHO] = cart2pol(X,Y)
```

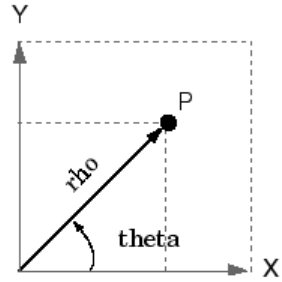
Description

`[THETA,RHO,Z] = cart2pol(X,Y,Z)` transforms three-dimensional Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z`, into cylindrical coordinates. `THETA` is a counterclockwise angular displacement in radians from the positive *x*-axis, `RHO` is the distance from the origin to a point in the *x-y* plane, and `Z` is the height above the *x-y* plane. Arrays `X`, `Y`, and `Z` must be the same size (or any can be scalar).

`[THETA,RHO] = cart2pol(X,Y)` transforms two-dimensional Cartesian coordinates stored in corresponding elements of arrays `X` and `Y` into polar coordinates.

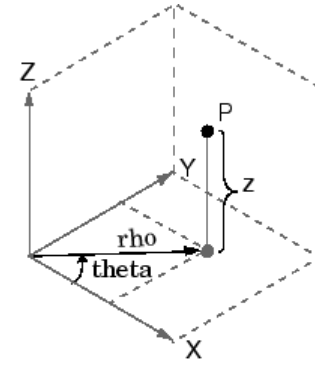
Algorithm

The mapping from two-dimensional Cartesian coordinates to polar coordinates, and from three-dimensional Cartesian coordinates to cylindrical coordinates is



Two-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
```



Three-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
```

See Also

`cart2sph`, `pol2cart`, `sph2cart`

cart2sph

Purpose

Transform Cartesian coordinates to spherical

Syntax

```
[THETA,PHI,R] = cart2sph(X,Y,Z)
```

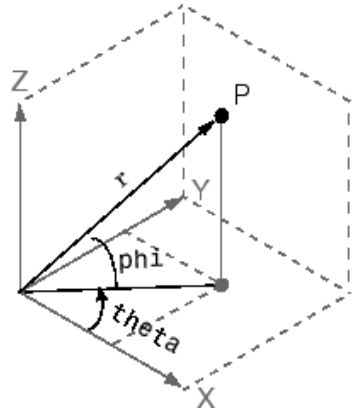
Description

`[THETA,PHI,R] = cart2sph(X,Y,Z)` transforms Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z` into spherical coordinates. Azimuth `THETA` and elevation `PHI` are angular displacements in radians measured from the positive *x*-axis, and the *x-y* plane, respectively; and `R` is the distance from the origin to a point.

Arrays `X`, `Y`, and `Z` must be the same size.

Algorithm

The mapping from three-dimensional Cartesian coordinates to spherical coordinates is




```
theta = atan2(y,x)
  phi = atan2(z, sqrt(x.^2 + y.^2))
    r = sqrt(x.^2+y.^2+z.^2)
```

The notation for spherical coordinates is not standard. For the `cart2sph` function, the angle PHI is measured from the *x-y* plane. Notice that if PHI = 0 then the point is in the *x-y* plane and if PHI = pi/2 then the point is on the positive *z*-axis.

See Also

`cart2pol`, `pol2cart`, `sph2cart`

Purpose

Execute block of code if condition is `true`

Syntax

```
switch switch_expr
 case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

Description

`case` is part of the `switch` statement syntax which allows for conditional execution. A particular case consists of the `case` statement itself followed by a case expression and one or more statements.

`case case_expr` compares the value of the expression `switch_expr` declared in the preceding `switch` statement with one or more values in `case_expr`, and executes the block of code that follows if any of the comparisons yield a `true` result.

You typically use multiple `case` statements in the evaluation of a single `switch` statement. The block of code associated with a particular `case` statement is executed only if its associated `case` expression (`case_expr`) is the first to match the switch expression (`switch_expr`).

To enter more than one `case` expression in a `switch` statement, put the expressions in a cell array, as shown above.

Examples

To execute a certain block of code based on what the string, `method`, is set to,

```
method = 'Bilinear';

switch lower(method)
   case {'linear','bilinear'}
      disp('Method is linear')
   case 'cubic'
```

```
      disp('Method is cubic')
   case 'nearest'
      disp('Method is nearest')
   otherwise
      disp('Unknown method.')
end

Method is linear
```

See Also

switch, otherwise, end, if, else, elseif, while

Purpose

Cast variable to different data type

Syntax

```
B = cast(A, newclass)
```

Description

`B = cast(A, newclass)` casts `A` to class `newclass`. `A` must be convertible to class `newclass`. `newclass` must be the name of one of the built in data types.

Examples

```
a = int8(5);
b = cast(a,'uint8');
class(b)

ans =

uint8
```

See Also

`class`

Purpose

Concatenate arrays along specified dimension

Syntax

```
C = cat(dim, A, B)
C = cat(dim, A1, A2, A3, A4, ...)
```

Description

`C = cat(dim, A, B)` concatenates the arrays `A` and `B` along `dim`.

`C = cat(dim, A1, A2, A3, A4, ...)` concatenates all the input arrays (`A1`, `A2`, `A3`, `A4`, and so on) along `dim`.

`cat(2, A, B)` is the same as `[A, B]`, and `cat(1, A, B)` is the same as `[A; B]`.

Remarks

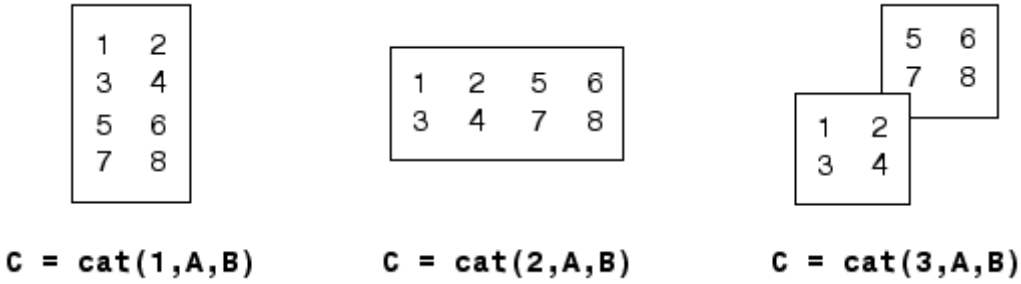
When used with comma-separated list syntax, `cat(dim, C{:})` or `cat(dim, C.field)` is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.

Examples

Given

```
A =                 B =
     1     2               5     6
     3     4               7     8
```

concatenating along different dimensions produces



The commands

```
A = magic(3); B = pascal(3);
C = cat(4, A, B);
```

produce a 3-by-3-by-1-by-2 array.

See Also

`num2cell`

The `special character []`

catch

Purpose

Specify how to respond to error in `try` statement

Syntax

```
catch ME
catch
```

Description

`catch ME` marks the start of a *catch block* in a `try-catch` statement. It returns object `ME`, which is an instance of the MATLAB class `MException`. This object contains information about an error caught in the preceding *try block* and can be useful in helping your program respond to the error appropriately.

A `try-catch` statement is a programming device that enables you to define how certain errors are to be handled in your program. This bypasses the default MATLAB error-handling mechanism when these errors are detected. The `try-catch` statement consists of two blocks of MATLAB code, a *try block* and a *catch block*, delimited by the keywords `try`, `catch`, and `end`:

```
try
   MATLAB commands     % Try block
catch ME
   MATLAB commands     % Catch block
end
```

Each of these blocks consists of one or more MATLAB commands. The `try` block is just another piece of your program code; the commands in this block execute just like any other part of your program. Any errors MATLAB encounters in the `try` block are dealt with by the respective `catch` block. This is where you write your error-handling code. If the `try` block executes without error, MATLAB skips the `catch` block entirely. If an error occurs while executing the `catch` block, the program terminates unless this error is caught by another `try-catch` block.

`catch` marks the start of a catch block but does not return an `MException` object. You can obtain the error string that was generated by calling the `lasterror` function.

Specifying the `try`, `catch`, and `end` commands, as well as the commands that make up the `try` and `catch` blocks, on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

Examples

The `catch` block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension. This is done using a `try-catch` statement that is nested within the original `try-catch`.

```
function d_in = read_image(filename)
file_format = regexp(filename, '(?<=\.)\w+$', 'match');

try
  fid = fopen(filename, 'r');
  d_in = fread(fid);
catch ME1
  % Get last segment of the error message identifier.
  idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

  % Did the read fail because the file could not be found?
  if strcmp(idSegLast, 'InvalidFid') && ~exist(filename, 'file')

    % Yes. Try modifying the filename extension.
    switch file_format
    case 'jpg'    % Change jpg to jpeg
      filename = regexprep(filename, '(?<=\.)\w+$', 'jpeg');
    case 'jpeg'   % Change jpeg to jpg
      filename = regexprep(filename, '(?<=\.)\w+$', 'jpg');
```

```
        case 'tif'    % Change tif to tiff
          filename = regexprep(filename, '(?<=\.)\w+$', 'tiff');
        case 'tiff'   % Change tiff to tif
          filename = regexprep(filename, '(?<=\.)\w+$', 'tif');
        otherwise
          disp(sprintf('File %s not found', filename));
           rethrow(ME1);
        end

        % Try again, with modifed filenames.
        try
          fid = fopen(filename, 'r');
          d_in = fread(fid);
        catch ME2
          disp(sprintf('Unable to access file %s', filename));
          ME2 = addCause(ME2, ME1);
          rethrow(ME2)
        end
      end
    end
```

See Also

try, rethrow, end, lasterror, eval, evalin

Purpose

Color axis scaling

Syntax

```
caxis([cmin cmax])
caxis auto
caxis manual
caxis(caxis) freeze
v = caxis
caxis(axes_handle,...)
```

Description

`caxis` controls the mapping of data values to the colormap. It affects any surfaces, patches, and images with indexed `CData` and `CDataMapping` set to scaled. It does not affect surfaces, patches, or images with true color `CData` or with `CDataMapping` set to `direct`.

`caxis([cmin cmax])` sets the color limits to specified minimum and maximum values. Data values less than `cmin` or greater than `cmax` map to `cmin` and `cmax`, respectively. Values between `cmin` and `cmax` linearly map to the current colormap.

`caxis auto` lets MATLAB compute the color limits automatically using the minimum and maximum data values. This is the default behavior. Color values set to `Inf` map to the maximum color, and values set to `-Inf` map to the minimum color. Faces or edges with color values set to `NaN` are not drawn.

`caxis manual` and `caxis(caxis) freeze` the color axis scaling at the current limits. This enables subsequent plots to use the same limits when `hold` is `on`.

`v = caxis` returns a two-element row vector containing the `[cmin cmax]` currently in use.

`caxis(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

Remarks

`caxis` changes the `CLim` and `CLimMode` properties of axes graphics objects.

How Color Axis Scaling Works

Surface, patch, and image graphics objects having indexed `CData` and `CDataMapping` set to `scaled` map `CData` values to colors in the figure colormap each time they render. `CData` values equal to or less than `cmin` map to the first color value in the colormap, and `CData` values equal to or greater than `cmax` map to the last color value in the colormap. MATLAB performs the following linear transformation on the intermediate values (referred to as `C` below) to map them to an entry in the colormap (whose length is `m`, and whose row index is referred to as `index` below).

```
index = fix((C-cmin)/(cmax-cmin)*m)+1
```

Examples

Create `(X,Y,Z)` data for a sphere and view the data as a surface.

```
[X,Y,Z] = sphere;
C = Z;
surf(X,Y,Z,C)
```

Values of `C` have the range [-1 1]. Values of `C` near -1 are assigned the lowest values in the colormap; values of `C` near 1 are assigned the highest values in the colormap.

To map the top half of the surface to the highest value in the color table, use

```
caxis([-1 0])
```

To use only the bottom half of the color table, enter

```
caxis([-1 3])
```

which maps the lowest `CData` values to the bottom of the colormap, and the highest values to the middle of the colormap (by specifying a `cmax` whose value is equal to `cmin` plus twice the range of the `CData`).

The command

```
caxis auto
```

resets axis scaling back to autoranging and you see all the colors in the surface. In this case, entering

```
caxis
```

returns

```
[-1 1]
```

Adjusting the color axis can be useful when using images with scaled color data. For example, load the image data and colormap for Cape Cod, Massachusetts.

```
load cape
```

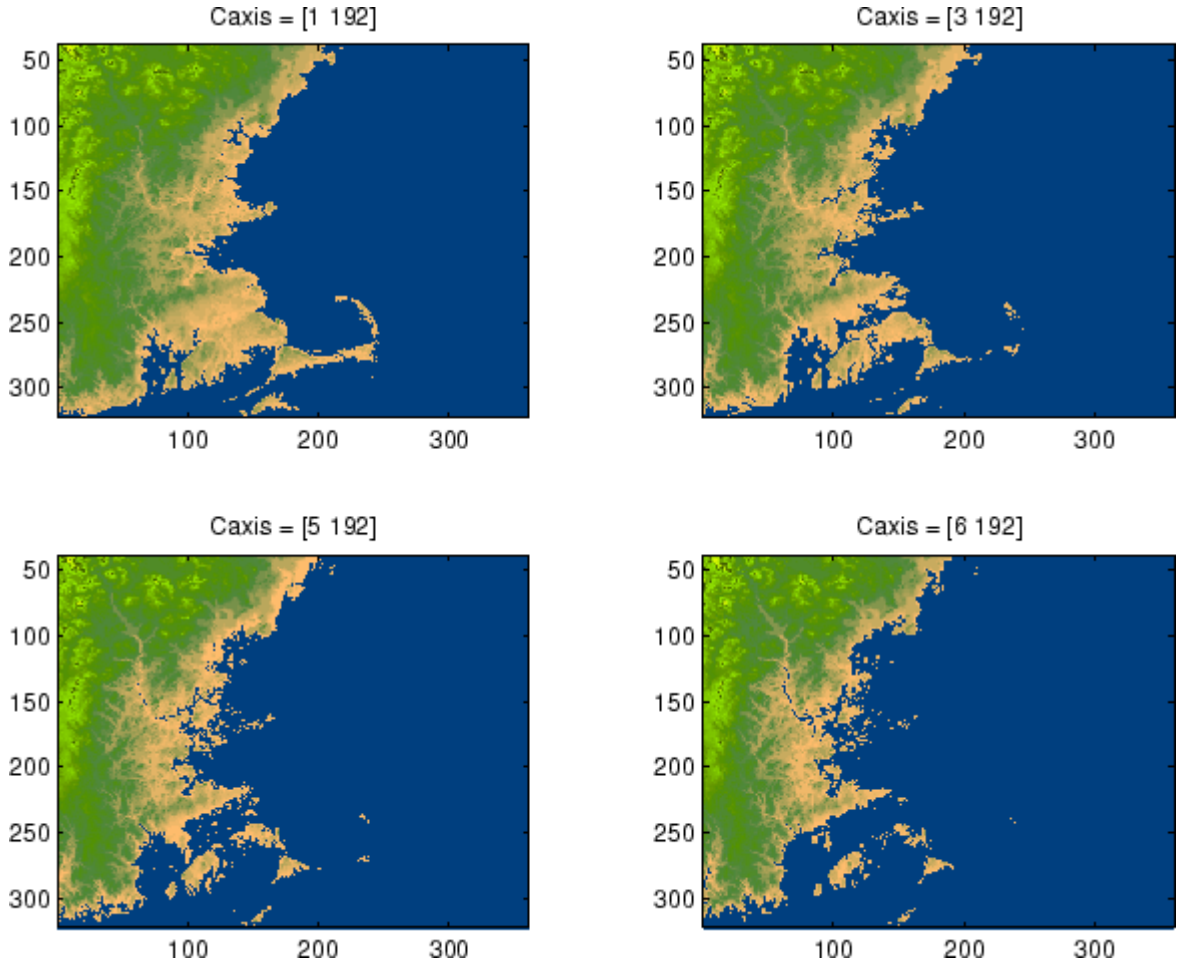
This command loads the image's data `X` and the image's colormap `map` into the workspace. Now display the image with `CDataMapping` set to `scaled` and install the image's colormap.

```
image(X,'CDataMapping','scaled')colormap(map)
```

MATLAB sets the color limits to span the range of the image data, which is 1 to 192:

```
caxis
ans =
     1   192
```

The blue color of the ocean is the first color in the colormap and is mapped to the lowest data value (1). You can effectively move sea level by changing the lower color limit value. For example,



See Also

`axes`, `axis`, `colormap`, `get`, `mesh`, `pcolor`, `set`, `surf`

The `CLim` and `CLimMode` properties of axes graphics objects

The `Colormap` property of figure graphics objects

“Color Operations” on page 1-98 for related functions

"Axes Color Limits — the CLim Property" for more examples

Purpose

Change working directory

Graphical Interface

As an alternative to the `cd` function, use the current directory field

in the MATLAB desktop toolbar.

Syntax

```
cd
w = cd
cd('directory')
cd('..')
cd directory
```

Description

`cd` displays the current working directory.

`w = cd` assigns the current working directory to `w`.

`cd('directory')` sets the current working directory to `directory`. Use the full pathname for `directory`. On UNIX platforms, the character ~ is interpreted as the user's root directory.

`cd('..')` changes the current working directory to the directory above it.

`cd directory` or cd .. is the unquoted form of the syntax.

Examples

On UNIX

```
cd('/usr/local/matlab/toolbox/control/ctrldemos')
```

changes the current working directory to `ctrldemos` for the Control System Toolbox.

On Windows

```
cd('c:/matlab/toolbox/control/ctrldemos')
```

changes the current working directory to `ctrldemos` for the Control System Toolbox. Then typing

```
cd ..
```

changes the current working directory to `control`, and typing

```
cd ..
```

again, changes the current working directory to `toolbox`.

On any platform, use `cd` with the `matlabroot` function to change to a directory relative to the directory in which MATLAB is installed. For example

```
cd([matlabroot '/toolbox/control/ctrldemos'])
```

changes the current working directory to `ctrldemos` for the Control System Toolbox.

See Also

`dir`, `fileparts`, `mfilename`, `path`, `pwd`, `what`

cd (ftp)

Purpose

Change current directory on FTP server

Syntax

```
cd(f)
cd(f,'dirname')
cd(f,'..')
```

Description

`cd(f)` Displays the current directory on the FTP server `f`, where `f` was created using `ftp`.

`cd(f,'dirname')` Changes the current directory on the FTP server `f` to `dirname`, where `f` was created using `ftp`. After running cd, the object `f` remembers the current directory on the FTP server. You can then perform file operations functions relative to `f` using the methods `delete`, `dir`, `mget`, `mkdir`, `mput`, `rename`, and `rmdir`.

`cd(f,'..')` changes the current directory on the FTP server `f` to the directory above the current one.

Examples

Connect to the MathWorks FTP server.

```
tmw=ftp('ftp.mathworks.com');
```

View the contents.

```
dir(tmw)
```

Change the current directory to `pub`.

```
cd(tmw,'pub');
```

View the contents of `pub`.

```
dir(tmw)
```

See Also

`dir (ftp)`, `ftp`

Purpose

Convert complex diagonal form to real block diagonal form

Syntax

```
[V,D] = cdf2rdf(V,D)
[V,D] = cdf2rdf(V,D)
```

Description

If the eigensystem `[V,D] = eig(X)` has complex eigenvalues appearing in complex-conjugate pairs, `cdf2rdf` transforms the system so `D` is in real diagonal form, with 2-by-2 real blocks along the diagonal replacing the complex pairs originally there. The eigenvectors are transformed so that

```
X = V*D/V
```

continues to hold. The individual columns of `V` are no longer eigenvectors, but each pair of vectors associated with a 2-by-2 block in `D` spans the corresponding invariant vectors.

Examples

The matrix

```
X =
     1     2     3
     0     4     5
     0    -5     4
```

has a pair of complex eigenvalues.

```
[V,D] = eig(X)

V =

    1.0000      -0.0191 - 0.4002i      -0.0191 + 0.4002i
         0            0 - 0.6479i            0 + 0.6479i
         0       0.6479                 0.6479

D =

    1.0000            0                      0
```

```
         0       4.0000 + 5.0000i            0
         0            0                 4.0000 - 5.0000i
```

Converting this to real block diagonal form produces

```
[V,D] = cdf2rdf(V,D)

V =

    1.0000    -0.0191     -0.4002
         0          0     -0.6479
         0     0.6479           0

D =

    1.0000          0           0
         0     4.0000      5.0000
         0    -5.0000      4.0000
```

Algorithm

The real diagonal form for the eigenvalues is obtained from the complex form using a specially constructed similarity transformation.

See Also

`eig`, `rsf2csf`

Purpose

Construct cdfepoch object for Common Data Format (CDF) export

Syntax

```
E = cdfepoch(date)
```

Description

`E = cdfepoch(date)` constructs a `cdfepoch` object, where `date` is a valid string (`datestr`), a number (`datenum`) representing a date, or a `cdfepoch` object.

When writing data to a CDF using `cdfwrite`, use `cdfepoch` to convert MATLAB formatted dates to CDF formatted dates. The MATLAB `cdfepoch` object simulates the `CDFEPOCH` data type in CDF files.

Use the `todatenum` function to convert a `cdfepoch` object into a MATLAB serial date number.

Note A CDF epoch is the number of milliseconds since 1-Jan-0000. MATLAB `datenums` are the number of days since 0-Jan-0000.

See Also

`cdfinfo`, `cdfread`, `cdfwrite`, `datenum`

Purpose

Information about Common Data Format (CDF) file

Syntax

```
info = cdfinfo(filename)
```

Description

`info = cdfinfo(filename)` returns information about the Common Data Format (CDF) file specified in the string `filename`.

Note Because `cdfinfo` creates temporary files, the current working directory must be writeable.

The return value, `info`, is a structure that contains the fields listed alphabetically in the following table.

Field	Description
`FileModDate`	Text string indicating the date the file was last modified
`Filename`	Text string specifying the name of the file
`FileSettings`	Structure array containing library settings used to create the file
`FileSize`	Double scalar specifying the size of the file, in bytes
`Format`	Text string specifying the file format
`FormatVersion`	Text string specifying the version of the CDF library used to create the file
`GlobalAttributes`	Structure array that contains one field for each global attribute. The name of each field corresponds to the name of an attribute. The data in each field, contained in a cell array, represents the entry values for that attribute.

Field	Description
`Subfiles`	Filenames containing the CDF file's data, if it is a multifile CDF
`VariableAttributes`	Structure array that contains one field for each variable attribute. The name of each field corresponds to the name of an attribute. The data in each field is contained in a *n*-by-2 cell array, where *n* is the number of variables. The first column of this cell array contains the variable names associated with the entries. The second column contains the entry values.

Field	Description	
`Variables`	N-by-6 cell array, where N is the number of variables, containing information about the variables in the file. The columns present the following information:	
	Column 1	Text string specifying name of variable
	Column 2	Double array specifying the dimensions of the variable, as returned by the `size` function
	Column 3	Double scalar specifying the number of records assigned for the variable
	Column 4	Text string specifying the data type of the variable, as stored in the CDF file

	Column 5	Text string specifying the record and dimension variance settings for the variable. The single `T` or `F` to the left of the slash designates whether values vary by record. The zero or more `T` or `F` letters to the right of the slash designate whether values vary at each dimension. Here are some examples.
		`T/ (scalar variable`
		`F/T (one-dimensional variable)`
		T/TFF (three-dimensional variable)

	Column 6	Text string specifying the sparsity of the variable's records, with these possible values: `'Full' 'Sparse (padded)' 'Sparse (nearest)'`

Note Attribute names returned by `cdfinfo` might not match the names of the attributes in the CDF file exactly. Attribute names can contain characters that are illegal in MATLAB field names. `cdfinfo` removes illegal characters that appear at the beginning of attributes and replaces other illegal characters with underscores ('_'). When `cdfinfo` modifies an attribute name, it appends the attribute's internal number to the end of the field name. For example, the attribute name `Variable%Attribute` becomes `Variable_Attribute_013`.

Examples

```
info = cdfinfo('example.cdf')
info =
               Filename: 'example.cdf'
            FileModDate: '09-Mar-2001 15:45:22'
               FileSize: 1240
                 Format: 'CDF'
          FormatVersion: '2.7.0'
           FileSettings: [1x1 struct]
               Subfiles: {}
              Variables: {5x6 cell}
       GlobalAttributes: [1x1 struct]
    VariableAttributes: [1x1 struct]

info.Variables
ans =
  'Time'         [1x2 double] [24] 'epoch'  'T/'     'Full'
  'Longitude'    [1x2 double] [ 1] 'int8'   'F/FT'   'Full'
  'Latitude'     [1x2 double] [ 1] 'int8'   'F/TF'   'Full'
  'Data'         [1x3 double] [ 1] 'double' 'T/TTT'  'Full'
  'multidim'     [1x4 double] [ 1] 'uint8'  'T/TTTT' 'Full'
```

See Also

`cdfread`

Purpose

Read data from Common Data Format (CDF) file

Syntax

```
data = cdfread(filename)
data = cdfread(filename, param1, val1, param2, val2, ...)
[data, info] = cdfread(filename, ...)
```

Description

`data = cdfread(filename)` reads all the data from the Common Data Format (CDF) file specified in the string `filename`. CDF data sets typically contain a set of variables, of a specific data type, each with an associated set of records. The variable might represent time values with each record representing a specific time that an observation was recorded. `cdfread` returns all the data in a cell array where each column represents a variable and each row represents a record associated with a variable. If the variables have varying numbers of associated records, `cdfread` pads the rows to create a rectangular cell array, using pad values defined in the CDF file.

Note Because `cdfread` creates temporary files, the current working directory must be writeable.

`data = cdfread(filename, param1, val1, param2, val2, ...)` reads data from the file, where *param1*, *param2*, and so on, can be any of the following parameters.

Parameter	Value
`'Records'`	A vector specifying which records to read. Record numbers are zero-based. `cdfread` returns a cell array with the same number of rows as the number of records read and as many columns as there are variables.

Parameter	Value
`'Variables'`	A 1-by-*n* or *n*-by-1 cell array specifying the names of the variables to read from the file. *n* must be less than or equal to the total number of variables in the file. `cdfread` returns a cell array with the same number of columns as the number of variables read, and a row for each record read.
`'Slices'`	An *m*-by-3 array, where each row specifies where to start reading along a particular dimension of a variable, the skip interval to use on that dimension (every item, every other item, etc.), and the total number of values to read on that dimension. *m* must be less than or equal to the number of dimensions of the variable. If *m* is less than the total number of dimensions, `cdfread` reads every value from the unspecified dimensions (`[0 1 n]`, where *n* is the total number of elements in the dimension. Note: Because the `'Slices'` parameter describes how to process a single variable, it must be used in conjunction with the `'Variables'` parameter.

Parameter	Value
'ConvertEpochToDatenum'	A Boolean value that determines whether cdfread automatically converts CDF epoch data types to MATLAB serial date numbers. If set to false (the default), cdfread wraps epoch values in MATLABcdfepoch objects. Note: For better performance when reading large data sets, set this parameter to true.
'CombineRecords'	A Boolean value that determines how cdfread returns the CDF data sets read from the file. If set to false (the default), cdfread stores the data in an *m*-by-*n* cell array, where *m* is the number of records and *n* is the number of variables requested. If set to true, cdfread combines all records for a particular variable into one cell in the output cell array. In this cell, cdfread stores scalar data as a column array. cdfread extends the dimensionality of nonscalar and string data. For example, instead of creating 1000 elements containing 20-by-30 arrays for each record, cdfread stores all the records in one cell as a 1000-by-20-by-30 array Note: If you use the 'Records' parameter to specify which records to read, you cannot use the 'CombineRecords' parameter. Note: When using the 'Variable' parameter to read one variable, if the 'CombineRecords' parameter is true, cdfread returns the data as an M-by-N numeric or character array; it does not put the data into a cell array.

`[data, info] = cdfread(`*`filename`*`, ...)` returns details about the CDF file in the `info` structure.

Note To maximize performance, specify both the `'ConvertEpochToDatenum'` and `'CombineRecords'` parameters, setting their values to `'true'`.

Examples

Read all the data from a CDF file.

```
data = cdfread('example.cdf');
```

Read the data from the variable `'Time'`.

```
data = cdfread('example.cdf', 'Variable', {'Time'});
```

Read the first value in the first dimension, the second value in the second dimension, the first and third values in the third dimension, and all values in the remaining dimension of the variable `'multidimensional'`.

```
data = cdfread('example.cdf', ...
               'Variable', {'multidimensional'}, ...
               'Slices', [0 1 1; 1 1 1; 0 2 2]);
```

This is similar to reading the whole variable into `data` and then using matrix indexing, as in the following.

```
data{1}(1, 2, [1 3], :)
```

Collapse the records from a data set and convert CDF epoch data types to MATLAB serial date numbers.

```
data = cdfread('example.cdf', ...
               'CombineRecords', true, ...
               'ConvertEpochToDatenum', true);
```

See Also

`cdfepoch`, `cdfinfo`, `cdfwrite`

For more information about using this function, see “Common Data Format (CDF) Files”.

Purpose

Write data to Common Data Format (CDF) file

Syntax

```
cdfwrite(filename,variablelist)
cdfwrite(...,'PadValues',padvals)
cdfwrite(...,'GlobalAttributes',gattrib)
cdfwrite(..., 'VariableAttributes', vattrib)
cdfwrite(...,'WriteMode',mode)
cdfwrite(...,'Format',format)
```

Description

`cdfwrite(filename,variablelist)` writes out a Common Data Format (CDF) file, specified in `filename`. The `filename` input is a string enclosed in single quotes. The `variablelist` argument is a cell array of ordered pairs, each of which comprises a CDF variable name (a string) and the corresponding CDF variable value. To write out multiple records for a variable, put the values in a cell array where each element in the cell array represents a record.

Note Because `cdfwrite` creates temporary files, both the destination directory for the file and the current working directory must be writeable.

`cdfwrite(...,'PadValues',padvals)` writes out pad values for given variable names. `padvals` is a cell array of ordered pairs, each of which comprises a variable name (a string) and a corresponding pad value. Pad values are the default values associated with the variable when an out-of-bounds record is accessed. Variable names that appear in `padvals` must appear in `variablelist`.

`cdfwrite(...,'GlobalAttributes',gattrib)` writes the structure `gattrib` as global metadata for the CDF file. Each field of the structure is the name of a global attribute. The value of each field contains the value of the attribute. To write out multiple values for an attribute, put the values in a cell array where each element in the cell array represents a record.

Note To specify a global attribute name that is illegal in MATLAB, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `GlobalAttributes` structure, and the corresponding name of the attribute to be written to the CDF file.

`cdfwrite(..., 'VariableAttributes', vattrib)` writes the structure `vattrib` as variable metadata for the CDF. Each field of the struct is the name of a variable attribute. The value of each field should be an M-by-2 cell array where M is the number of variables with attributes. The first element in the cell array should be the name of the variable and the second element should be the value of the attribute for that variable.

Note To specify a variable attribute name that is illegal in MATLAB, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `VariableAttributes` struct, and the corresponding name of the attribute to be written to the CDF file. If you are specifying a variable attribute of a CDF variable that you are renaming, the name of the variable in the `VariableAttributes` structure must be the same as the renamed variable.

`cdfwrite(...,'WriteMode',`*`mode`*`)`, where *`mode`* is either `'overwrite'` or `'append'`, indicates whether or not the specified variables should be appended to the CDF file if the file already exists. By default, `cdfwrite` overwrites existing variables and attributes.

`cdfwrite(...,'Format',`*`format`*`)`, where *`format`* is either `'multifile'` or `'singlefile'`, indicates whether or not the data is written out as a multifile CDF. In a multifile CDF, each variable is stored in a separate

file with the name `*.vN`, where `N` is the number of the variable that is written out to the CDF. By default, `cdfwrite` writes out a single file CDF. When `'WriteMode'` is set to `'Append'`, the `'Format'` option is ignored, and the format of the preexisting CDF is used.

Examples

Write out a file `'example.cdf'` containing a variable `'Longitude'` with the value `[0:360]`.

```
cdfwrite('example', {'Longitude', 0:360});
```

Write out a file `'example.cdf'` containing variables `'Longitude'` and `'Latitude'` with the variable `'Latitude'` having a pad value of 10 for all out-of-bounds records that are accessed.

```
cdfwrite('example', {'Longitude', 0:360, 'Latitude', 10:20}, ...
         'PadValues', {'Latitude', 10});
```

Write out a file `'example.cdf'`, containing a variable `'Longitude'` with the value `[0:360]`, and with a variable attribute of `'validmin'` with the value 10.

```
varAttribStruct.validmin = {'longitude' [10]};
cdfwrite('example', {'Longitude' 0:360}, 'VarAttribStruct', ...
         varAttribStruct);
```

See Also

`cdfread`, `cdfinfo`, `cdfepoch`

Purpose

Round toward infinity

Syntax

```
B = ceil(A)
```

Description

`B = ceil(A)` rounds the elements of `A` to the nearest integers greater than or equal to `A`. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000        -0.2000        3.4000        5.6000

  Columns 5 through 6
   7.0000        2.4000 + 3.6000i

ceil(a)

ans =
  Columns 1 through 4
  -1.0000        0              4.0000        6.0000

  Columns 5 through 6
   7.0000        3.0000 + 4.0000i
```

See Also

`fix`, `floor`, `round`

Purpose

Construct cell array

Syntax

```
c = cell(n)
c = cell(m, n)
c = cell([m, n])
c = cell(m, n, p,...)
c = cell([m n p ...])
c = cell(size(A))
c = cell(javaobj)
```

Description

`c = cell(n)` creates an `n`-by-`n` cell array of empty matrices. An error message appears if `n` is not a scalar.

`c = cell(m, n)` or `c = cell([m, n])` creates an `m`-by-`n` cell array of empty matrices. Arguments `m` and `n` must be scalars.

`c = cell(m, n, p,...)` or `c = cell([m n p ...])` creates an `m`-by-`n`-by-`p`-... cell array of empty matrices. Arguments `m`, `n`, `p`,`...` must be scalars.

`c = cell(size(A))` creates a cell array the same size as `A` containing all empty matrices.

`c = cell(javaobj)` converts a Java array or Java object `javaobj` into a MATLAB cell array. Elements of the resulting cell array will be of the MATLAB type (if any) closest to the Java array elements or Java object.

Remarks

This type of cell is not related to “cell mode,” a MATLAB feature used in debugging and publishing.

Examples

This example creates a cell array that is the same size as another array, `A`.

```
A = ones(2,2)

A =
     1     1
     1     1
```

```
c = cell(size(A))

c =
     []     []
     []     []
```

The next example converts an array of `java.lang.String` objects into a MATLAB cell array.

```
strArray = java_array('java.lang.String', 3);
strArray(1) = java.lang.String('one');
strArray(2) = java.lang.String('two');
strArray(3) = java.lang.String('three');

cellArray = cell(strArray)
cellArray =
    'one'
    'two'
    'three'
```

See Also

`num2cell`, `ones`, `rand`, `randn`, `zeros`

Purpose

Convert cell array of matrices to single matrix

Syntax

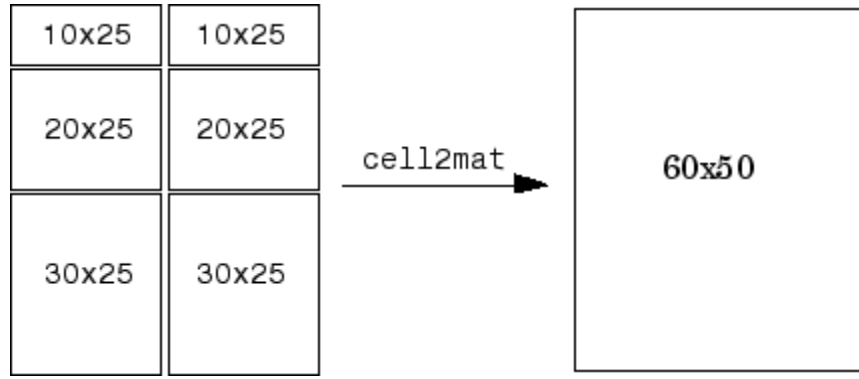
```
m = cell2mat(c)
```

Description

`m = cell2mat(c)` converts a multidimensional cell array `c` with contents of the same data type into a single matrix, `m`. The contents of `c` must be able to concatenate into a hyperrectangle. Moreover, for each pair of neighboring cells, the dimensions of the cells' contents must match, excluding the dimension in which the cells are neighbors.

The example shown below combines matrices in a 3-by-2 cell array into a single 60-by-50 matrix:

```
cell2mat(c)
```



Remarks

The dimensionality (or number of dimensions) of `m` will match the highest dimensionality contained in the cell array.

`cell2mat` is not supported for cell arrays containing cell arrays or objects.

Examples

Combine the matrices in four cells of cell array `C` into the single matrix, `M`:

```
C = {[1] [2 3 4]; [5; 9] [6 7 8; 10 11 12]}
```

```
C =
    [         1]    [1x3 double]
    [2x1 double]    [2x3 double]

C{1,1}                      C{1,2}
ans =                       ans =
     1                           2     3     4

C{2,1}                      C{2,2}
ans =                       ans =
     5                          6     7     8
     9                         10    11    12

M = cell2mat(C)
M =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

See Also

mat2cell, num2cell

Purpose

Convert cell array to structure array

Syntax

```
s = cell2struct(c, fields, dim)
```

Description

`s = cell2struct(c, fields, dim)` creates a structure array `s` from the information contained within cell array `c`.

The `fields` argument specifies field names for the structure array. `fields` can be a character array or a cell array of strings.

The `dim` argument controls which axis of the cell array is to be used in creating the structure array. The length of `c` along the specified dimension must match the number of fields named in `fields`. In other words, the following must be true.

```
size(c,dim) == length(fields)   % If fields is a cell array
size(c,dim) == size(fields,1)   % If fields is a char array
```

Examples

The cell array `c` in this example contains information on trees. The three columns of the array indicate the common name, genus, and average height of a tree.

```
c = {'birch', 'betula', 65;  'maple', 'acer', 50}
c =
    'birch'    'betula'    [65]
    'maple'    'acer'      [50]
```

To put this information into a structure with the fields `name`, `genus`, and `height`, use `cell2struct` along the second dimension of the 2-by-3 cell array.

```
fields = {'name', 'genus', 'height'};
s = cell2struct(c, fields, 2);
```

This yields the following 2-by-1 structure array.

```
s(1)                        s(2)
ans =                       ans =
      name: 'birch'               name: 'maple'
```

```
      genus: 'betula'             genus: 'acer'
     height: 65                  height: 50
```

See Also

`struct2cell`, `cell`, `iscell`, `struct`, `isstruct`, `fieldnames`, dynamic field names

celldisp

Purpose

Cell array contents

Syntax

```
celldisp(C)
celldisp(C, name)
```

Description

`celldisp(C)` recursively displays the contents of a cell array.

`celldisp(C, name)` uses the string *name* for the display instead of the name of the first input (or `ans`).

Examples

Use `celldisp` to display the contents of a 2-by-3 cell array:

```
C = {[1 2] 'Tony' 3+4i; [1 2;3 4] -5 'abc'};
celldisp(C)

C{1,1} =
     1     2

C{2,1} =
     1     2
     3     4

C{1,2} =
Tony

C{2,2} =
    -5

C{1,3} =
   3.0000+ 4.0000i

C{2,3} =
abc
```

See Also

`cellplot`

Purpose

Apply function to each cell in cell array

Syntax

```
A = cellfun(fun, C)
A = cellfun(fun, C, D, ...)
[A, B, ...] = cellfun(fun, C, ...)
[A, ...] = cellfun(fun, C, ..., 'param1', value1, ...)
A = cellfun('fname', C)
A = cellfun('size', C, k)
A = cellfun('isclass', C, 'classname')
```

Description

`A = cellfun(fun, C)` applies the function specified by `fun` to the contents of each cell of cell array `C`, and returns the results in array `A`. The value `A` returned by `cellfun` is the same size as `C`, and the `(I,J,...)`th element of `A` is equal to `fun(C{I,J,...})`. The first input argument `fun` is a function handle to a function that takes one input argument and returns a scalar value. `fun` must return values of the same class each time it is called. The order in which `cellfun` computes elements of `A` is not specified and should not be relied upon.

If `fun` is bound to more than one built-in or M-file (that is, if it represents a set of overloaded functions), then the class of the values that `cellfun` actually provides as input arguments to `fun` determines which functions are executed.

`A = cellfun(fun, C, D, ...)` evaluates `fun` using the contents of the cells of cell arrays `C`, `D`, `...` as input arguments. The `(I,J,...)`th element of `A` is equal to `fun(C{I,J,...}, D{I,J,...}, ...)`. All input arguments must be of the same size and shape.

`[A, B, ...] = cellfun(fun, C, ...)` evaluates `fun`, which is a function handle to a function that returns multiple outputs, and returns arrays `A`, `B`, `...`, each corresponding to one of the output arguments of `fun`. `cellfun` calls `fun` each time with as many outputs as there are in the call to `cellfun`. `fun` can return output arguments having different classes, but the class of each output must be the same each time `fun` is called.

`[A, ...] = cellfun(fun, C, ..., 'param1', value1, ...)` enables you to specify optional parameter name and value pairs.

Parameters recognized by `cellfun` are shown below. Enclose each parameter name with single quotes.

Parameter Name	Parameter Value
`UniformOutput`	Logical 1 (`true`) or 0 (`false`), indicating whether or not the outputs of `fun` can be returned without encapsulation in a cell array. See "UniformOutput Parameter" on page 2-510 below.
`ErrorHandler`	Function handle, specifying the function that `cellfun` is to call if the call to `fun` fails. See "ErrorHandler Parameter" on page 2-510 below.

UniformOutput Parameter

If you set the `UniformOutput` parameter to `true` (the default), `fun` must return scalar values that can be concatenated into an array. These values can also be a cell array.

If `UniformOutput` is `false`, `cellfun` returns a cell array (or multiple cell arrays), where the `(I,J,...)`th cell contains the value

```
fun(C{I,J,...}, ...)
```

ErrorHandler Parameter

MATLAB calls the function represented by the `ErrorHandler` parameter with two input arguments:

- A structure having three fields, named `identifier`, `message`, and `index`, respectively containing the identifier of the error that occurred, the text of the error message, and a linear index into the input array or arrays for which the error occurred
- The set of input arguments for which the call to the function failed

The error handling function must either rethrow the error that was caught, or it must return the output values from the call to `fun`. Error

handling functions that do not rethrow the error must have the same number of outputs as `fun`. MATLAB places these output values in the output variables used in the call to `arrayfun`.

Shown here is an example of a simple error handling function, `errorfun`:

```
function [A, B] = errorfun(S, varargin)
warning(S.identifier, S.message);
A = NaN;  B = NaN;
```

If `'UniformOutput'` is set to logical 1 (`true`), the outputs of the error handler must be scalars and of the same data type as the outputs of function `fun`.

If you do not specify an error handler, `cellfun` rethrows the error.

Backward Compatibility

The following syntaxes are also accepted for backward compatibility:

`A = cellfun('fname', C)` applies the function `fname` to the elements of cell array `C` and returns the results in the double array `A`. Each element of `A` contains the value returned by `fname` for the corresponding element in `C`. The output array `A` is the same size as the cell array `C`.

These functions are supported:

Function	Return Value
`isempty`	`true` for an empty cell element
`islogical`	`true` for a logical cell element
`isreal`	`true` for a real cell element
`length`	Length of the cell element
`ndims`	Number of dimensions of the cell element
`prodofsize`	Number of elements in the cell element

`A = cellfun('size', C, k)` returns the size along the `k`th dimension of each element of `C`.

`A = cellfun('isclass', C, 'classname')` returns logical `1` (`true`) for each element of `C` that matches `classname`. This function syntax returns logical 0 (`false`) for objects that are a subclass of `classname`.

Note For the previous three syntaxes, if `C` contains objects, `cellfun` does not call any overloaded versions of MATLAB functions corresponding to the above strings.

Examples

Compute the mean of several data sets:

```
C = {1:10, [2; 4; 6], []};

Cmeans = cellfun(@mean, C)
Cmeans =
    5.5000    4.0000       NaN
```

Compute the size of these data sets:

```
[Cnrows, Cncols] = cellfun(@size, C)
Cnrows =
     1     3     0
Cncols =
    10     1     0
```

Again compute the size, but with `UniformOutput` set to `false`:

```
Csize = cellfun(@size, C, 'UniformOutput', false)
Csize =
    [1x2 double]    [1x2 double]    [1x2 double]

Csize{:}
ans =
     1    10
ans =
     3     1
ans =
```

```
     0     0
```

Find the positive values in several data sets.

```
C = {randn(10,1), randn(20,1), randn(30,1)};

Cpositives = cellfun(@(x) x(x>0), C, 'UniformOutput',false)
Cpositives =
    [6x1 double]    [11x1 double]    [15x1 double]

Cpositives{:}
ans =
    0.1253
    0.2877
    1.1909
     etc.
ans =
    0.7258
    2.1832
    0.1139
     etc.
ans =
    0.6900
    0.8156
    0.7119
     etc.
```

Compute the covariance between several pairs of data sets:

```
C = {randn(10,1), randn(20,1), randn(30,1)};
D = {randn(10,1), randn(20,1), randn(30,1)};

CDcovs = cellfun(@cov, C, D, 'UniformOutput', false)
CDcovs =
    [2x2 double]    [2x2 double]    [2x2 double]

CDcovs{:}
ans =
```

```
    0.7353   -0.2148
   -0.2148    0.6080
ans =
    0.5743   -0.2912
   -0.2912    0.8505
ans =
    0.7130    0.1750
    0.1750    0.6910
```

See Also

arrayfun, spfun, function_handle, cell2mat

Purpose

Graphically display structure of cell array

Syntax

```
cellplot(c)
cellplot(c, 'legend')
handles = cellplot(c)
```

Description

`cellplot(c)` displays a figure window that graphically represents the contents of `c`. Filled rectangles represent elements of vectors and arrays, while scalars and short text strings are displayed as text.

`cellplot(c, 'legend')` places a colorbar next to the plot labelled to identify the data types in `c`.

`handles = cellplot(c)` displays a figure window and returns a vector of surface handles.

Limitations

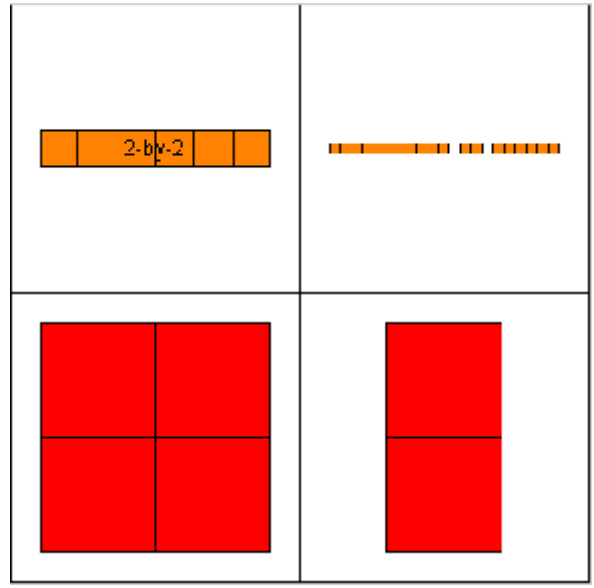
The `cellplot` function can display only two-dimensional cell arrays.

Examples

Consider a 2-by-2 cell array containing a matrix, a vector, and two text strings:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces



Purpose

Create cell array of strings from character array

Syntax

```
c = cellstr(S)
```

Description

`c = cellstr(S)` places each row of the character array `S` into separate cells of `c`. Any trailing spaces in the rows of `S` are removed.

Use the `char` function to convert back to a string matrix.

Examples

Given the string matrix

```
S = ['abc '; 'defg'; 'hi  ']

S =
    abc
    defg
    hi

whos S
  Name      Size          Bytes  Class
  S         3x4              24  char array
```

The following command returns a 3-by-1 cell array.

```
c = cellstr(S)

c =
    'abc'
    'defg'
    'hi'

whos c
  Name      Size          Bytes  Class
  c         3x1              294  cell array
```

See Also

`iscellstr`, `strings`, `char`, `isstrprop`

Purpose

Conjugate gradients squared method

Syntax

```
x = cgs(A,b)
cgs(A,b,tol)
cgs(A,b,tol,maxit)
cgs(A,b,tol,maxit,M)
cgs(A,b,tol,maxit,M1,M2)
cgs(A,b,tol,maxit,M1,M2,x0)
[x,flag] = cgs(A,b,...)
[x,flag,relres] = cgs(A,b,...)
[x,flag,relres,iter] = cgs(A,b,...)
[x,flag,relres,iter,resvec] = cgs(A,b,...)
```

Description

`x = cgs(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `cgs` converges, a message to that effect is displayed. If `cgs` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`cgs(A,b,tol)` specifies the tolerance of the method, `tol`. If `tol` is `[]`, then `cgs` uses the default, `1e-6`.

`cgs(A,b,tol,maxit)` specifies the maximum number of iterations, `maxit`. If `maxit` is `[]` then `cgs` uses the default, `min(n,20)`.

`cgs(A,b,tol,maxit,M)` and `cgs(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `cgs` applies no

preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`cgs(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess `x0`. If `x0` is `[]`, then `cgs` uses the default, an all-zero vector.

`[x,flag] = cgs(A,b,...)` returns a solution `x` and a flag that describes the convergence of `cgs`.

Flag	Convergence
0	`cgs` converged to the desired tolerance `tol` within `maxit` iterations.
1	`cgs` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`cgs` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `cgs` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = cgs(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres <= tol`.

`[x,flag,relres,iter] = cgs(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = cgs(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Example

```
A = gallery('wilk',21);
b = sum(A,2);
```

```
tol = 1e-12;  maxit = 15;
M1 = diag([10:-1:1 1 1:10]);
x = cgs(A,b,tol,maxit,M1);
```

displays the message

```
cgs converged at iteration 13 to a solution with relative residual
1.3e-016
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in an M-file `run_cgs` that

- Calls `cgs` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_cgs` are available to `afun` and `myfun`.

The following shows the code for `run_cgs`:

```
function x1 = run_cgs
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;  maxit = 15;
x1 = cgs(@afun,b,tol,maxit,@mfun);

    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
```

```
end
```

When you enter

```
x1 = run_cgs
```

MATLAB returns

```
cgs converged at iteration 13 to a solution with relative residual
1.3e-016
```

Example 3

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = cgs(A,b)
```

`flag` is `1` because `cgs` does not converge to the default tolerance `1e-6` within the default 20 iterations.

```
[L1,U1] = luinc(A,1e-5)
[x1,flag1] = cgs(A,b,1e-6,20,L1,U1)
```

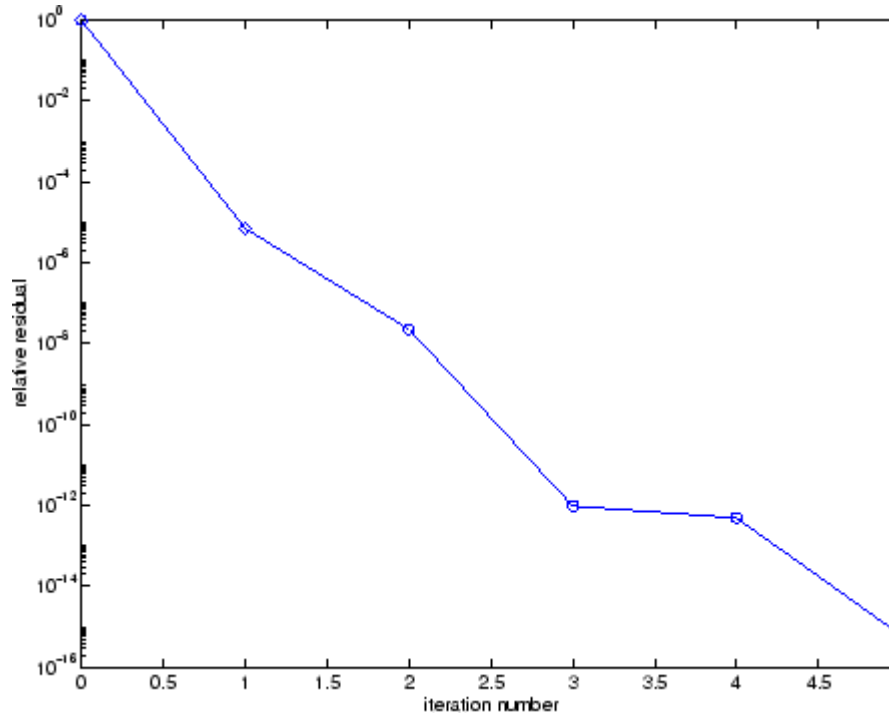
`flag1` is `2` because the upper triangular `U1` has a zero on its diagonal, and `cgs` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` with backslash.

```
[L2,U2] = luinc(A,1e-6)
[x2,flag2,relres2,iter2,resvec2] = cgs(A,b,1e-15,10,L2,U2)
```

`flag2` is `0` because `cgs` converges to the tolerance of `6.344e-16` (the value of `relres2`) at the fifth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. `resvec2(1) = norm(b)` and `resvec2(6) = norm(b-A*x2)`. You can follow the progress of `cgs` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0) with

```
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
```

```
ylabel('relative residual')
```



See Also

`bicg`, `bicgstab`, `gmres`, `lsqr`, `luinc`, `minres`, `pcg`, `qmr`, `symmlq`

`function_handle` (`@`), `mldivide` (`\`)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Sonneveld, Peter, "CGS: A fast Lanczos-type solver for nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, January 1989, Vol. 10, No. 1, pp. 36-52.

Purpose

Convert to character array (string)

Syntax

```
S = char(X)
S = char(C)
S = char(t1, t2, t3, ...)
```

Description

`S = char(X)` converts the array `X` that contains nonnegative integers representing character codes into a MATLAB character array. The actual characters displayed depend on the character encoding scheme for a given font. The result for any elements of `X` outside the range from 0 to 65535 is not defined (and can vary from platform to platform). Use `double` to convert a character array into its numeric codes.

`S = char(C)`, when `C` is a cell array of strings, places each element of `C` into the rows of the character array `s`. Use `cellstr` to convert back.

`S = char(t1, t2, t3, ...)` forms the character array `S` containing the text strings `T1`, `T2`, `T3`, `...` as rows, automatically padding each string with blanks to form a valid matrix. Each text parameter, `T`*`i`*, can itself be a character array. This allows the creation of arbitrarily large character arrays. Empty strings are significant.

Examples

To print a 3-by-32 display of the printable ASCII characters,

```
ascii = char(reshape(32:127, 32, 3)')
ascii =
   !"#$%&'()*+,-./0123456789:;<=>?
  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
  'abcdefghijklmnopqrstuvwxyz{|}~
```

See Also

`ischar`, `isletter`, `isspace`, `isstrprop`, `cellstr`, `iscellstr`, `get`, `set`, `strings`, `strvcat`, `text`

checkin

Purpose

Check files into source control system (UNIX)

GUI Alternatives

As an alternative to the `checkin` function, use **File > Source Control > Check In** in the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser. For more information, see "Checking Files Into the Source Control System on UNIX".

Syntax

```
checkin('filename','comments','comment_text')
checkin({'filename1','filename2'},'comments','comment_text')
checkin('filename','comments', 'comment_text','option',
   'value')
```

Description

`checkin('filename','comments','comment_text')` checks in the file named `filename` to the source control system. Use the full path for `filename` and include the file extension. You must save the file before checking it in, but the file can be open or closed. The comment_text argument is a MATLAB string containing checkin comments for the source control system. You must supply **`comments`** and comment_text.

`checkin({'filename1','filename2'},'comments','comment_text')` checks in the files `filename1` through `filenamen` to the source control system. Use the full paths for the files and include file extensions. Comments apply to all files checked in.

`checkin('filename','comments', 'comment_text','option','value')` provides additional `checkin` options. For multiple filenames, use an array of strings instead of `filename`, that is, `{'filename1','filename2',...}`. Options apply to all filenames. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	`filename` is checked in even if the file has not changed since it was checked out.
`'force'`	`'off'` (default)	`filename` is not checked in if there were no changes since checkout.
`'lock'`	`'on'`	`filename` is checked in with comments, and is automatically checked out.
`'lock'`	`'off'` (default)	`filename` is checked in with comments but does not remain checked out.

Examples

Check In a File

Typing

```
checkin('/myserver/mymfiles/clock.m','comments',...
'Adjustment for leapyear')
```

checks the file `/myserver/mymfiles/clock.m` into the source control system, with the comment `Adjustment for leapyear`.

Check In Multiple Files

Typing

```
checkin({'/myserver/mymfiles/clock.m', ...
'/myserver/mymfiles/calendar.m'},'comments',...
'Adjustment for leapyear')
```

checks the two files into the source control system, using the same comment for each.

Check In a File and Keep It Checked Out

Typing

```
checkin('/myserver/mymfiles/clock.m','comments',...
'Adjustment for leapyear','lock','on')
```

checks the file `/myserver/mymfiles/clock.m` into the source control system and keeps the file checked out.

See Also

`checkout`, `cmopts`, `undocheckout`

For Windows platforms, use `verctrl`.

Purpose

Check files out of source control system (UNIX)

GUI Alternatives

As an alternative to the `checkout` function, select **Source Control > Check Out** from the **File** menu in the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser. For details, see "Checking Files Out of the Source Control System on UNIX".

Syntax

```
checkout('filename')
checkout({'filename1','filename2', ...})
checkout('filename','option','value',...)
```

Description

`checkout('filename')` checks out the file named `filename` from the source control system. Use the full path for `filename` and include the file extension. The file can be open or closed when you use `checkout`.

`checkout({'filename1','filename2', ...})` checks out the files named `filename1` through `filenamen` from the source control system. Use the full paths for the files and include the file extensions.

`checkout('filename','option','value',...)` provides additional `checkout` options. For multiple filenames, use an array of strings instead of `filename`, that is, `{'filename1','filename2', ...}`. Options apply to all filenames. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	The checkout is forced, even if you already have the file checked out. This is effectively an `undocheckout` followed by a `checkout`.

option Argument	value Argument	Purpose
`'force'`	`'off'` (default)	Prevents you from checking out the file if you already have it checked out.
`'lock'`	`'on'` (default)	The checkout gets the file, allows you to write to it, and locks the file so that access to the file for others is read only.
`'lock'`	`'off'`	The checkout gets a read-only version of the file, allowing another user to check out the file for updating. You do not have to check the file in after checking it out with this option.
`'revision'`	`'version_num'`	Checks out the specified revision of the file.

If you end the MATLAB session, the file remains checked out. You can check in the file from within MATLAB during a later session, or directly from your source control system.

Examples

Check Out a File

Typing

```
checkout('/myserver/mymfiles/clock.m')
```

checks out the file `/myserver/mymfiles/clock.m` from the source control system.

Check Out Multiple Files

Typing

```
checkout({'/myserver/mymfiles/clock.m',...
'/myserver/mymfiles/calendar.m'})
```

checks out `/matlab/mymfiles/clock.m` and `/matlab/mymfiles/calendar.m` from the source control system.

Force a Checkout, Even If File Is Already Checked Out

Typing

```
checkout('/myserver/mymfiles/clock.m','force','on')
```

checks out `/matlab/mymfiles/clock.m` even if `clock.m` is already checked out to you.

Check Out Specified Revision of File

Typing

```
checkout('/matlab/mymfiles/clock.m','revision','1.1')
```

checks out revision `1.1` of `clock.m`.

See Also

`checkin`, `cmopts`, `undocheckout`, `customverctrl`

For Windows platforms, use `verctrl`.

Purpose

Cholesky factorization

Syntax

```
R = chol(A)
L = chol(A,'lower')
[R,p] = chol(A)
[L,p] = chol(A,'lower')
[R,p,S] = chol(A)
[R,p,s] = chol(A,'vector')
[L,p,s] = chol(A,'lower','vector')
```

Description

`R = chol(A)` produces an upper triangular matrix `R` from the diagonal and upper triangle of matrix `A`, satisfying the equation `R'*R=A`. The lower triangle is assumed to be the (complex conjugate) transpose of the upper triangle. Matrix `A` must be positive definite; otherwise, MATLAB displays an error message.

`L = chol(A,'lower')` produces a lower triangular matrix `L` from the diagonal and lower triangle of matrix `A`, satisfying the equation `L*L'=A`. When `A` is sparse, this syntax of `chol` is typically faster. Matrix `A` must be positive definite; otherwise MATLAB displays an error message.

`[R,p] = chol(A)` for positive definite `A`, produces an upper triangular matrix `R` from the diagonal and upper triangle of matrix `A`, satisfying the equation `R'*R=A` and `p` is zero. If `A` is not positive definite, then `p` is a positive integer and MATLAB does not generate an error. When `A` is full, `R` is an upper triangular matrix of order `q=p-1` such that `R'*R=A(1:q,1:q)`. When `A` is sparse, `R` is an upper triangular matrix of size `q`-by-`n` so that the `L`-shaped region of the first `q` rows and first `q` columns of `R'*R` agree with those of `A`.

`[L,p] = chol(A,'lower')` for positive definite `A`, produces a lower triangular matrix `L` from the diagonal and lower triangle of matrix `A`, satisfying the equation `L'*L=A` and `p` is zero. If `A` is not positive definite, then `p` is a positive integer and MATLAB does not generate an error. When `A` is full, `L` is a lower triangular matrix of order `q=p-1` such that `L'*L=A(1:q,1:q)`. When `A` is sparse, `L` is a lower triangular matrix of size `q`-by-`n` so that the `L`-shaped region of the first `q` rows and first `q` columns of `L'*L` agree with those of `A`.

`[R,p,S] = chol(A)`, when `A` is sparse, returns a permutation matrix `S`. Note that the preordering `S` may differ from that obtained from `amd` since `chol` will slightly change the ordering for increased performance. When `p=0`, `R` is an upper triangular matrix such that `R'*R=S'*A*S`. When `p` is not zero, `R` is an upper triangular matrix of size `q`-by-`n` so that the `L`-shaped region of the first `q` rows and first `q` columns of `R'*R` agree with those of `S'*A*S`. The factor of `S'*A*S` tends to be sparser than the factor of `A`.

`[R,p,s] = chol(A,'vector')` returns the permutation information as a vector `s` such that `A(s,s)=R'*R`, when `p=0`. You can use the `'matrix'` option in place of `'vector'` to obtain the default behavior.

`[L,p,s] = chol(A,'lower','vector')` uses only the diagonal and the lower triangle of `A` and returns a lower triangular matrix `L` and a permutation vector `s` such that `A(s,s)=L*L'`, when `p=0`. As above, you can use the `'matrix'` option in place of `'vector'` to obtain a permutation matrix.

For sparse `A`, CHOLMOD is used to compute the Cholesky factor.

Note Using `chol` is preferable to using `eig` for determining positive definiteness.

Examples

The binomial coefficients arranged in a symmetric array create an interesting positive definite matrix.

```
n = 5;
X = pascal(n)
X =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70
```

It is interesting because its Cholesky factor consists of the same coefficients, arranged in an upper triangular matrix.

```
R = chol(X)
R =
     1     1     1     1     1
     0     1     2     3     4
     0     0     1     3     6
     0     0     0     1     4
     0     0     0     0     1
```

Destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element.

```
X(n,n) = X(n,n)-1

X =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    69
```

Now an attempt to find the Cholesky factorization fails.

Algorithm

For full matrices `X`, `chol` uses the LAPACK routines listed in the following table.

	Real	Complex
`X double`	DPOTRF	ZPOTRF
`X single`	SPOTRF	CPOTRF

For sparse matrices, MATLAB uses CHOLMOD to compute the Cholesky factor.

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

[2] Davis, T. A., *CHOLMOD Version 1.0 User Guide* (`http://www.cise.ufl.edu/research/sparse/cholmod`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2005.

See Also

`cholinc`, `cholupdate`

cholinc

Purpose

Sparse incomplete Cholesky and Cholesky-Infinity factorizations

Syntax

```
R = cholinc(X,droptol)
R = cholinc(X,options)
R = cholinc(X,'0')
[R,p] = cholinc(X,'0')
R = cholinc(X,'inf')
```

Description

`cholinc` produces two different kinds of incomplete Cholesky factorizations: the drop tolerance and the 0 level of fill-in factorizations. These factors may be useful as preconditioners for a symmetric positive definite system of linear equations being solved by an iterative method such as `pcg` (Preconditioned Conjugate Gradients). `cholinc` works only for sparse matrices.

`R = cholinc(X,droptol)` performs the incomplete Cholesky factorization of `X`, with drop tolerance `droptol`.

`R = cholinc(X,options)` allows additional options to the incomplete Cholesky factorization. `options` is a structure with up to three fields:

`droptol`	Drop tolerance of the incomplete factorization
`michol`	Modified incomplete Cholesky
`rdiag`	Replace zeros on the diagonal of `R`

Only the fields of interest need to be set.

`droptol` is a non-negative scalar used as the drop tolerance for the incomplete Cholesky factorization. This factorization is computed by performing the incomplete LU factorization with the pivot threshold option set to `0` (which forces diagonal pivoting) and then scaling the rows of the incomplete upper triangular factor, `U`, by the square root of the diagonal entries in that column. Since the nonzero entries `U(i,j)` are bounded below by `droptol*norm(X(:,j))` (see `luinc`), the nonzero entries `R(i,j)` are bounded below by the local drop tolerance `droptol*norm(X(:,j))/R(i,i)`.

Setting `droptol = 0` produces the complete Cholesky factorization, which is the default.

`michol` stands for modified incomplete Cholesky factorization. Its value is either `0` (unmodified, the default) or `1` (modified). This performs the modified incomplete LU factorization of `X` and scales the returned upper triangular factor as described above.

`rdiag` is either `0` or `1`. If it is `1`, any zero diagonal entries of the upper triangular factor R are replaced by the square root of the local drop tolerance in an attempt to avoid a singular factor. The default is `0`.

`R = cholinc(X,'0')` produces the incomplete Cholesky factor of a real sparse matrix that is symmetric and positive definite using no fill-in. The upper triangular `R` has the same sparsity pattern as `triu(X)`, although `R` may be zero in some positions where `X` is nonzero due to cancellation. The lower triangle of `X` is assumed to be the transpose of the upper. Note that the positive definiteness of `X` does not guarantee the existence of a factor with the required sparsity. An error message results if the factorization is not possible. If the factorization is successful, `R'*R` agrees with `X` over its sparsity pattern.

`[R,p] = cholinc(X,'0')` with two output arguments, never produces an error message. If `R` exists, `p` is `0`. If `R` does not exist, then `p` is a positive integer and `R` is an upper triangular matrix of size `q`-by-`n` where `q = p-1`. In this latter case, the sparsity pattern of `R` is that of the `q`-by-`n` upper triangle of `X`. `R'*R` agrees with `X` over the sparsity pattern of its first `q` rows and first `q` columns.

`R = cholinc(X,'inf')` produces the Cholesky-Infinity factorization. This factorization is based on the Cholesky factorization, and additionally handles real positive semi-definite matrices. It may be useful for finding a solution to systems which arise in interior-point methods. When a zero pivot is encountered in the ordinary Cholesky factorization, the diagonal of the Cholesky-Infinity factor is set to `Inf` and the rest of that row is set to `0`. This forces a `0` in the corresponding entry of the solution vector in the associated system of linear equations. In practice, `X` is assumed to be positive semi-definite so even negative pivots are replaced with a value of `Inf`.

Remarks

The incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. A single `0` on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the `rdiag` option to replace a zero diagonal only gets rid of the symptoms of the problem, but it does not solve it. The preconditioner may not be singular, but it probably is not useful, and a warning message is printed.

The Cholesky-Infinity factorization is meant to be used within interior-point methods. Otherwise, its use is not recommended.

Examples

Example 1

Start with a symmetric positive definite matrix, `S`.

```
S = delsq(numgrid('C',15));
```

S is the two-dimensional, five-point discrete negative Lapacian on the grid generated by numgrid('C',15).

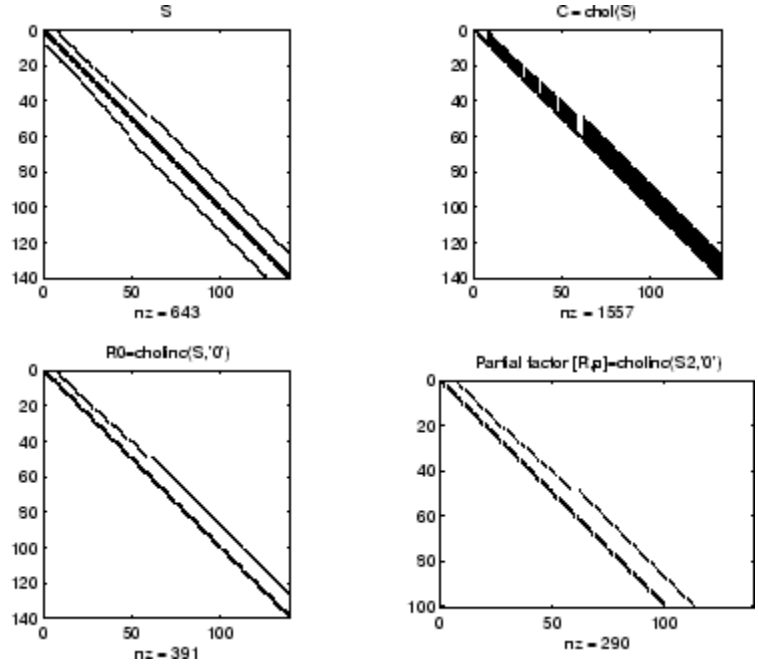
Compute the Cholesky factorization and the incomplete Cholesky factorization of level 0 to compare the fill-in. Make `S` singular by zeroing out a diagonal entry and compute the (partial) incomplete Cholesky factorization of level `0`.

```
C = chol(S);
R0 = cholinc(S,'0');
S2 = S; S2(101,101) = 0;
[R,p] = cholinc(S2,'0');
```

Fill-in occurs within the bands of `S` in the complete Cholesky factor, but none in the incomplete Cholesky factor. The incomplete factorization of the singular `S2` stopped at row `p = 101` resulting in a 100-by-139 partial factor.

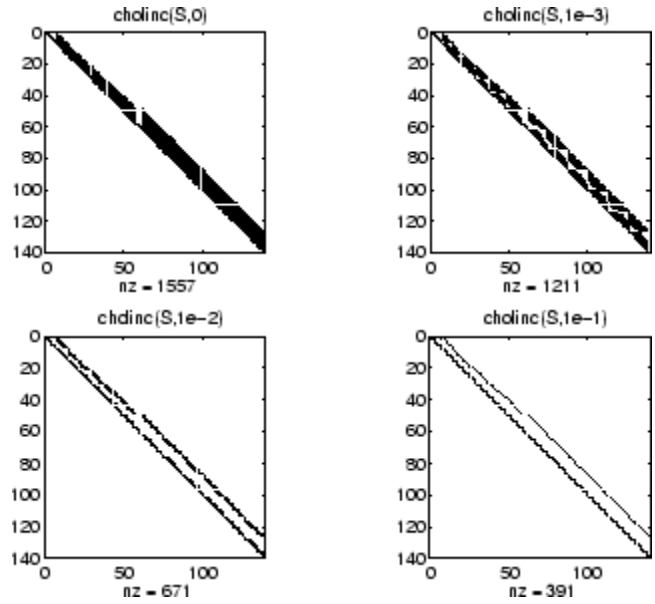
```
D1 = (R0'*R0).*spones(S)-S;
D2 = (R'*R).*spones(S2)-S2;
```

`D1` has elements of the order of `eps`, showing that `R0'*R0` agrees with `S` over its sparsity pattern. `D2` has elements of the order of `eps` over its first 100 rows and first 100 columns, `D2(1:100,:)` and `D2(:,1:100)`.

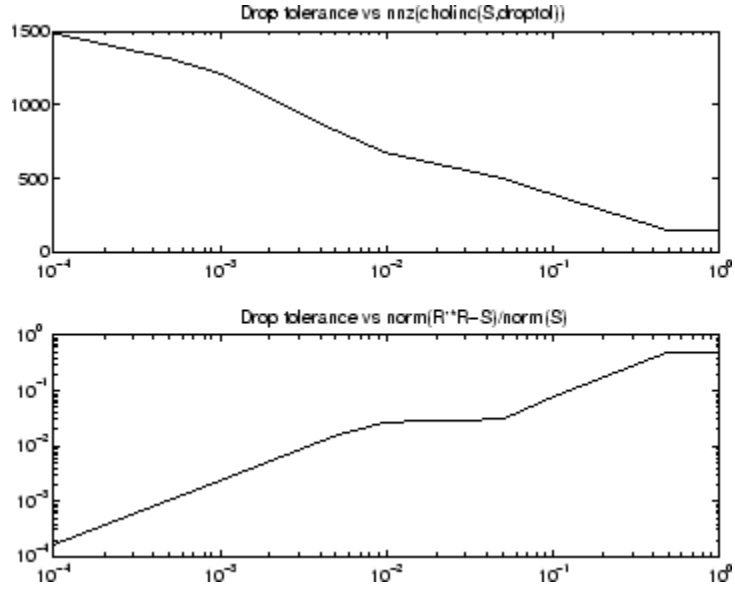


Example 2

The first subplot below shows that `cholinc(S,0)`, the incomplete Cholesky factor with a drop tolerance of `0`, is the same as the Cholesky factor of `S`. Increasing the drop tolerance increases the sparsity of the incomplete factors, as seen below.



Unfortunately, the sparser factors are poor approximations, as is seen by the plot of drop tolerance versus `norm(R'*R-S,1)/norm(S,1)` in the next figure.



Example 3

The Hilbert matrices have (i,j) entries 1/(i+j-1) and are theoretically positive definite:

```
H3 = hilb(3)
H3 =
    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
R3 = chol(H3)
R3 =
    1.0000    0.5000    0.3333
         0    0.2887    0.2887
         0         0    0.0745
```

In practice, the Cholesky factorization breaks down for larger matrices:

```
H20 = sparse(hilb(20));
```

```
[R,p] = chol(H20);
p =
    14
```

For `hilb(20)`, the Cholesky factorization failed in the computation of row 14 because of a numerically zero pivot. You can use the Cholesky-Infinity factorization to avoid this error. When a zero pivot is encountered, `cholinc` places an `Inf` on the main diagonal, zeros out the rest of the row, and continues with the computation:

```
Rinf = cholinc(H20,'inf');
```

In this case, all subsequent pivots are also too small, so the remainder of the upper triangular factor is:

```
full(Rinf(14:end,14:end))
ans =
   Inf     0     0     0     0     0     0
     0   Inf     0     0     0     0     0
     0     0   Inf     0     0     0     0
     0     0     0   Inf     0     0     0
     0     0     0     0   Inf     0     0
     0     0     0     0     0   Inf     0
     0     0     0     0     0     0   Inf
```

Limitations

`cholinc` works on square sparse matrices only. For `cholinc(X,'0')` and `cholinc(X,'inf')`, `X` must be real.

Algorithm

`R = cholinc(X,droptol)` is obtained from `[L,U] = luinc(X,options)`, where `options.droptol = droptol` and `options.thresh = 0`. The rows of the uppertriangular `U` are scaled by the square root of the diagonal in that row, and this scaled factor becomes `R`.

`R = cholinc(X,options)` is produced in a similar manner, except the `rdiag` option translates into the `udiag` option and the `milu` option takes the value of the `michol` option.

`R = cholinc(X,'0')` is based on the "KJI" variant of the Cholesky factorization. Updates are made only to positions which are nonzero in the upper triangle of `X`.

`R = cholinc(X,'inf')` is based on the algorithm in Zhang [2].

See Also

`chol`, `ilu`, `luinc`, `pcg`

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996. Chapter 10, "Preconditioning Techniques"

[2] Zhang, Yin, *Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment*, Department of Mathematics and Statistics, University of Maryland Baltimore County, Technical Report TR96-01

Purpose

Rank 1 update to Cholesky factorization

Syntax

```
R1 = cholupdate(R,x)
R1 = cholupdate(R,x,'+')
R1 = cholupdate(R,x,'-')
[R1,p] = cholupdate(R,x,'-')
```

Description

`R1 = cholupdate(R,x)` where `R = chol(A)` is the original Cholesky factorization of `A`, returns the upper triangular Cholesky factor of `A + x*x'`, where `x` is a column vector of appropriate length. `cholupdate` uses only the diagonal and upper triangle of `R`. The lower triangle of `R` is ignored.

`R1 = cholupdate(R,x,'+')` is the same as `R1 = cholupdate(R,x)`.

`R1 = cholupdate(R,x,'-')` returns the Cholesky factor of `A - x*x'`. An error message reports when R is not a valid Cholesky factor or when the downdated matrix is not positive definite and so does not have a Cholesky factorization.

`[R1,p] = cholupdate(R,x,'-')` will not return an error message. If `p` is `0`, `R1` is the Cholesky factor of `A - x*x'`. If `p` is greater than `0`, `R1` is the Cholesky factor of the original `A`. If `p` is `1`, `cholupdate` failed because the downdated matrix is not positive definite. If `p` is `2`, `cholupdate` failed because the upper triangle of `R` was not a valid Cholesky factor.

Remarks

`cholupdate` works only for full matrices.

Example

```
A = pascal(4)
A =

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20

R = chol(A)
R =
```

```
     1     1     1     1
     0     1     2     3
     0     0     1     3
     0     0     0     1
x = [0 0 0 1]';
```

This is called a rank one update to `A` since `rank(x*x')` is `1`:

```
A + x*x'
ans =


     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    21
```

Instead of computing the Cholesky factor with `R1 = chol(A + x*x')`, we can use `cholupdate`:

```
R1 = cholupdate(R,x)
R1 =

    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    1.4142
```

Next destroy the positive definiteness (and actually make the matrix singular) by subtracting `1` from the last element of `A`. The downdated matrix is:

```
A - x*x'
ans =

     1     1     1     1
     1     2     3     4
```

```
     1     3     6    10
     1     4    10    19
```

Compare `chol` with `cholupdate`:

```
R1 = chol(A-x*x')
??? Error using ==> chol
Matrix must be positive definite.
R1 = cholupdate(R,x,'-')
??? Error using ==> cholupdate
Downdated matrix must be positive definite.
```

However, subtracting `0.5` from the last element of `A` produces a positive definite matrix, and we can use `cholupdate` to compute its Cholesky factor:

```
x = [0 0 0 1/sqrt(2)]';
R1 = cholupdate(R,x,'-')
R1 =
    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    0.7071
```

Algorithm

`cholupdate` uses the algorithms from the LINPACK subroutines `ZCHUD` and `ZCHDD`. `cholupdate` is useful since computing the new Cholesky factor from scratch is an $O(N^3)$ algorithm, while simply updating the existing factor in this way is an $O(N^2)$ algorithm.

See Also

`chol`, `qrupdate`

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Purpose

Shift array circularly

Syntax

```
B = circshift(A,shiftsize)
```

Description

`B = circshift(A,shiftsize)` circularly shifts the values in the array, `A`, by `shiftsize` elements. `shiftsize` is a vector of integer scalars where the `n`-th element specifies the shift amount for the `n`-th dimension of array `A`. If an element in `shiftsize` is positive, the values of `A` are shifted down (or to the right). If it is negative, the values of `A` are shifted up (or to the left). If it is `0`, the values in that dimension are not shifted.

Example

Circularly shift first dimension values down by 1.

```
A = [ 1 2 3;4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9

B = circshift(A,1)
B =
     7     8     9
     1     2     3
     4     5     6
```

Circularly shift first dimension values down by 1 and second dimension values to the left by 1.

```
B = circshift(A,[1 -1]);
B =
    8     9     7
    2     3     1
    5     6     4
```

See Also

`fftshift`, `shiftdim`

cla

Purpose

Clear current axes

GUI Alternatives

Remove axes and clear objects from them in *plot edit* mode. For details, see "Working in Plot Edit Mode" in the MATLAB Graphics documentation.

Syntax

```
cla
cla reset
cla(ax)
cla(ax,'reset')
```

Description

`cla` deletes from the current axes all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to `on`).

`cla reset` deletes from the current axes all graphics objects regardless of the setting of their `HandleVisibility` property and resets all axes properties, except `Position` and `Units`, to their default values.

`cla(ax)` or `cla(ax,'reset')` clears the single axes with handle `ax`.

Remarks

The `cla` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `cla` deletes only those objects whose `HandleVisibility` property is set to `on`.

See Also

`clf`, `hold`, `newplot`, `reset`

"Axes Operations" on page 1-96 for related functions

Purpose

Contour plot elevation labels

Syntax

```
clabel(C,h)
clabel(C,h,v)
clabel(C,h,'manual')
clabel(C)
clabel(C,v)
clabel(C,'manual')
text_handles = clabel(...)
clabel(...,'PropertyName',propertyvalue,...)
clabel(...'LabelSpacing',points)
```

Description

The `clabel` function adds height labels to a 2-D contour plot.

`clabel(C,h)` rotates the labels and inserts them in the contour lines. The function inserts only those labels that fit within the contour, depending on the size of the contour.

`clabel(C,h,v)` creates labels only for those contour levels given in vector `v`, then rotates the labels and inserts them in the contour lines.

`clabel(C,h,'manual')` places contour labels at locations you select with a mouse. Press the left mouse button (the mouse button on a single-button mouse) or the space bar to label a contour at the closest location beneath the center of the cursor. Press the **Return** key while the cursor is within the figure window to terminate labeling. The labels are rotated and inserted in the contour lines.

`clabel(C)` adds labels to the current contour plot using the contour array `C` output from `contour`. The function labels all contours displayed and randomly selects label positions.

`clabel(C,v)` labels only those contour levels given in vector `v`.

`clabel(C,'manual')` places contour labels at locations you select with a mouse.

`text_handles = clabel(...)` returns the handles of text objects created by clabel. The `UserData` properties of the text objects contain the contour values displayed. If you call `clabel` without the `h` argument,

`text_handles` also contains the handles of line objects used to create the `'+'` symbols.

`clabel(...,'`*`PropertyName`*`',propertyvalue,...)` enables you to specify text object property/value pairs for the label strings. (See `Text Properties`.)

`clabel(...'LabelSpacing',`*`points`*`)` specifies the spacing between labels on the same contour line, in units of points (72 points equal one inch).

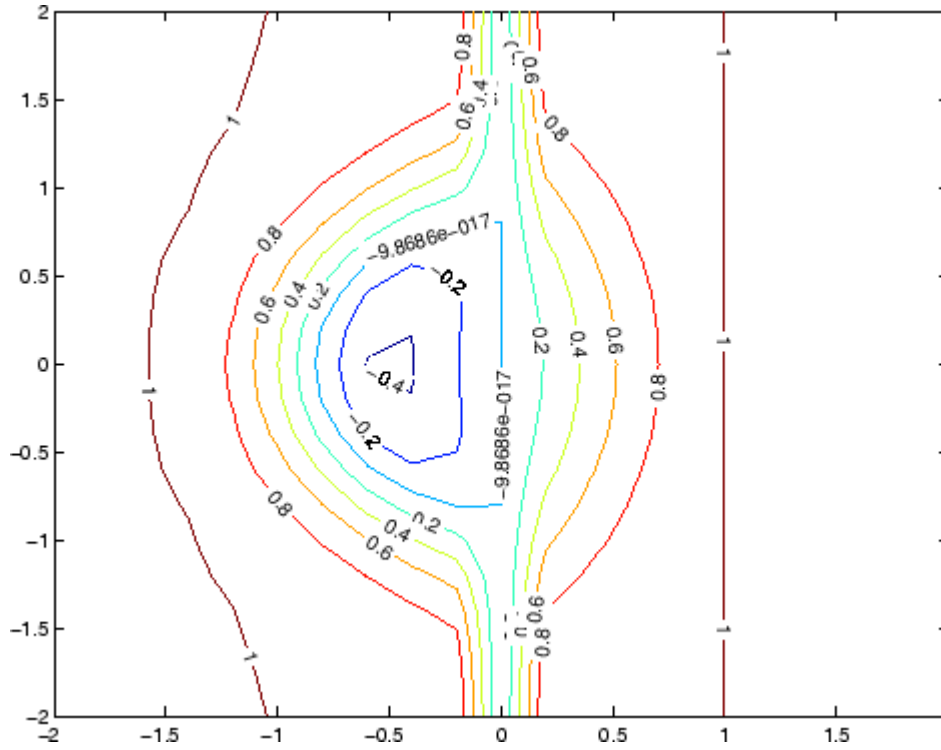
Remarks

When the syntax includes the argument `h`, this function rotates the labels and inserts them in the contour lines (see Examples). Otherwise, the labels are displayed upright and a `'+'` indicates which contour line the label is annotating.

Examples

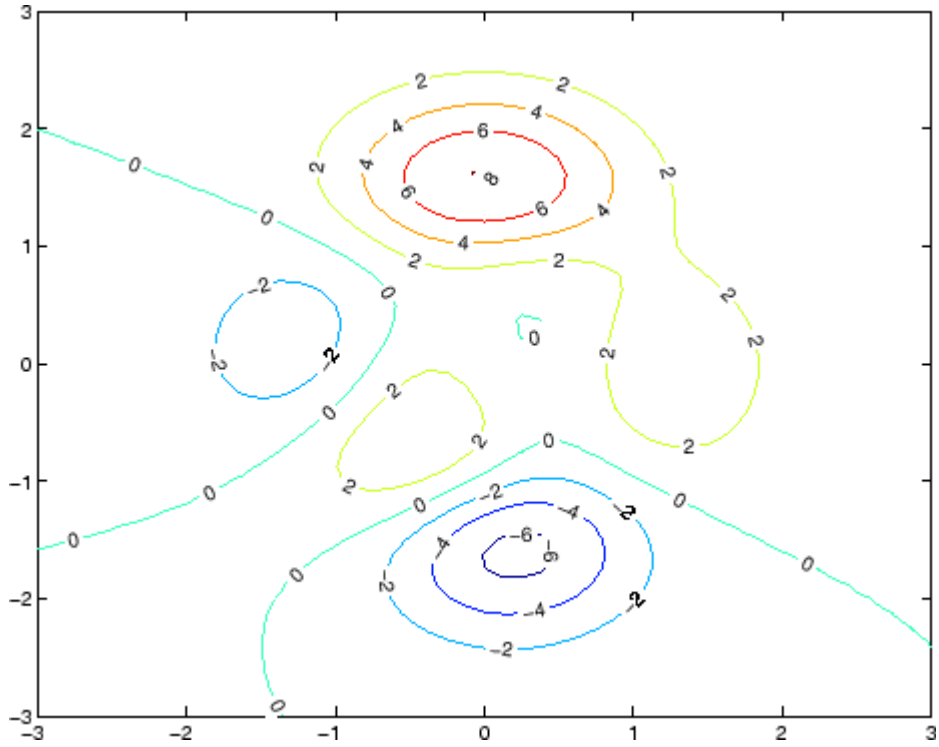
Generate, draw, and label a simple contour plot.

```
[x,y] = meshgrid(-2:.2:2);
z = x.^exp(-x.^2-y.^2);
[C,h] = contour(x,y,z);
clabel(C,h);
```



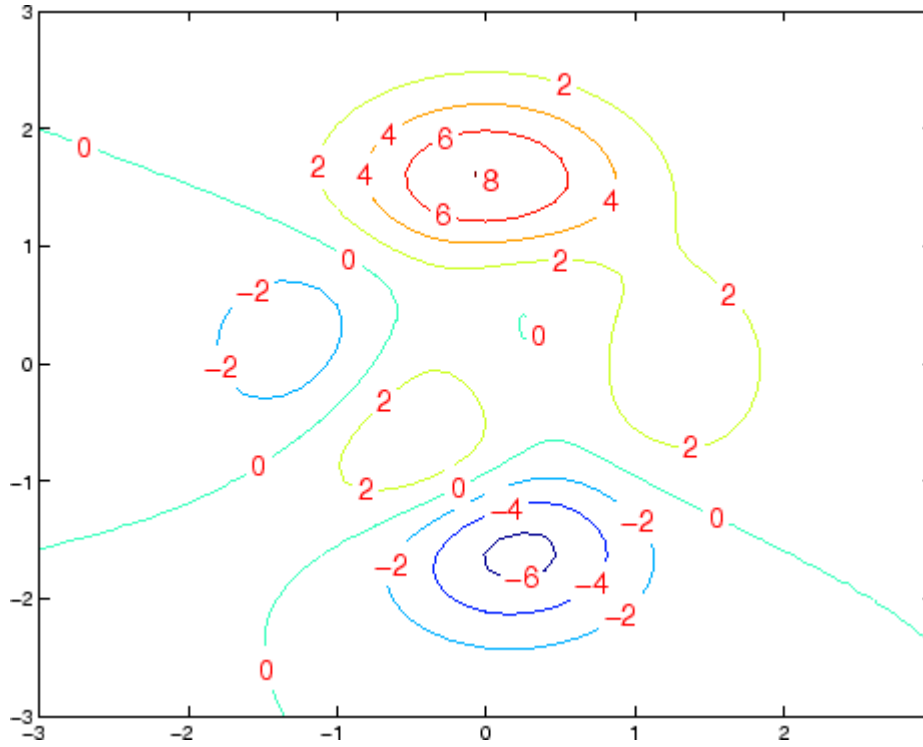
Label a contour plot with label spacing set to 72 points (one inch).

```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'LabelSpacing',72)
```



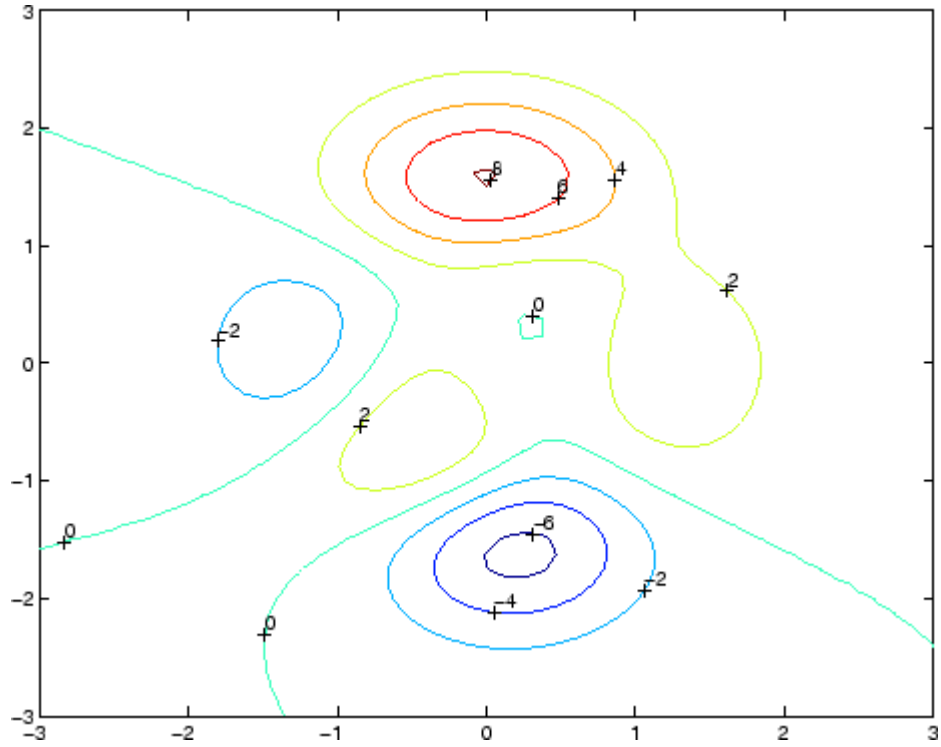
Label a contour plot with 15 point red text.

```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'FontSize',15,'Color','r','Rotation',0)
```



Label a contour plot with upright text and `'+'` symbols indicating which contour line each label annotates.

```
[x,y,z] = peaks;
C = contour(x,y,z);
clabel(C)
```



See Also

`contour`, `contourc`, `contourf`

“Annotating Plots” on page 1-87 for related functions

“Drawing Text in a Box” for an example that illustrates the use of contour labels

Purpose

Create object or return class of object

Syntax

```
str = class(object)
obj = class(s, 'class_name')
obj = class(s, 'class_name', parent1, parent2, ...)
obj = class(struct([]), 'class_name', parent1, parent2, ...)
```

Description

`str = class(object)` returns a string specifying the class of `object`.

The following table lists the object class names that can be returned. All except the last one are MATLAB classes.

`logical`	Logical array of `true` and `false` values
`char`	Character array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array
`single`	Single-precision floating-point number array
`double`	Double-precision floating-point number array
`cell`	Cell array
`struct`	Structure array
`function handle`	Array of values for calling functions indirectly
'*`class_name`*'	Custom MATLAB object class or Java class

`obj = class(s, 'class_name')` creates an object of MATLAB class `'class_name'` using structure `s` as a template. This syntax is valid only in a function named `class_name.m` in a directory named `@class_name` (where `'class_name'` is the same as the string passed in to `class`).

`obj = class(s, 'class_name', parent1, parent2, ...)` creates an object of MATLAB class `'class_name'` that inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. Structure `s` is used as a template for the object.

`obj = class(struct([]), 'class_name', parent1, parent2, ...)` creates an object of MATLAB class `'class_name'` that inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. Specifying the empty structure `struct([])` as the first argument ensures that the object created contains no fields other than those that are inherited from the parent objects.

Examples

To return in `nameStr` the name of the class of Java object `j`,

```
nameStr = class(j)
```

To create a user-defined MATLAB object of class `polynom`,

```
p = class(p, 'polynom')
```

See Also

`inferiorto`, `isa`, `superiorto`

The “Classes and Objects” and the “Calling Java from MATLAB” chapters in MATLAB Programming and Data Types documentation.

Purpose

Clear Command Window

GUI Alternatives

As an alternative to the `clc` function, select **Edit > Clear Command Window** in the MATLAB desktop.

Syntax

```
clc
```

Description

`clc` clears all input and output from the Command Window display, giving you a "clean screen."

After using `clc`, you cannot use the scroll bar to see the history of functions, but you still can use the up arrow to recall statements from the command history.

Examples

Use `clc` in an M-file to always display output in the same starting position on the screen.

See Also

`clear`, `clf`, `close`, `home`

Purpose

Remove items from workspace, freeing up system memory

Graphical Interface

As an alternative to the `clear` function, use **Edit > Clear Workspace** in the MATLAB desktop.

Syntax

```
clear
clear name
clear name1 name2 name3 ...
clear global name
clear -regexp expr1 expr2 ...
clear global -regexp expr1 expr2 ...
clear keyword
clear('name1','name2','name3',...)
```

Description

`clear` removes all variables from the workspace. This frees up system memory.

`clear name` removes just the M-file or MEX-file function or variable `name` from the workspace. You can use wildcards (`*`) to remove items selectively. For example, `clear my*` removes any variables whose names begin with the string `my`. It removes debugging breakpoints in M-files and reinitializes persistent variables, since the breakpoints for a function and persistent variables are cleared whenever the M-file is changed or cleared. If `name` is global, it is removed from the current workspace, but left accessible to any functions declaring it global. If `name` has been locked by `mlock`, it remains in memory.

Use a `partial path` to distinguish between different overloaded versions of a function. For example, `clear polynom/display` clears only the `display` method for `polynom` objects, leaving any other implementations in memory.

`clear name1 name2 name3 ...` removes `name1`, `name2`, and `name3` from the workspace.

`clear global name` removes the global variable `name`. If `name` is global, `clear name` removes `name` from the current workspace, but leaves it

accessible to any functions declaring it global. Use `clear global name` to completely remove a global variable.

`clear -regexp expr1 expr2 ...` clears all variables that match any of the regular expressions `expr1`, `expr2`, etc. This option only clears variables.

`clear global -regexp expr1 expr2 ...` clears all global variables that match any of the regular expressions `expr1`, `expr2`, etc.

`clear keyword` clears the items indicated by *`keyword`*.

Keyword	Items Cleared
`all`	Removes all variables, functions, and MEX-files from memory, leaving the workspace empty. Using `clear all` removes debugging breakpoints in M-files and reinitializes persistent variables, since the breakpoints for a function and persistent variables are cleared whenever the M-file is changed or cleared. When issued from the Command Window prompt, also removes the Java packages import list.
`classes`	The same as `clear all`, but also clears MATLAB class definitions. If any objects exist outside the workspace (for example, in user data or persistent variables in a locked M-file), a warning is issued and the class definition is not cleared. Issue a `clear classes` function if the number or names of fields in a class are changed.
`functions`	Clears all the currently compiled M-functions and MEX-functions from memory. Using `clear function` removes debugging breakpoints in the function M-file and reinitializes persistent variables, since the breakpoints for a function and persistent variables are cleared whenever the M-file is changed or cleared.

Keyword	Items Cleared
`global`	Clears all global variables from the workspace.
`import`	Removes the Java packages import list. It can only be issued from the Command Window prompt. It cannot be used in a function.
`java`	The same as `clear all`, but also clears the definitions of all Java classes defined by files on the Java dynamic class path (see "The Java Class Path" in the External Interfaces documentation). If any java objects exist outside the workspace (for example, in user data or persistent variables in a locked M-file), a warning is issued and the Java class definition is not cleared. Issue a `clear java` command after modifying any files on the Java dynamic class path.
`variables`	Clears all variables from the workspace.

`clear('name1','name2','name3',...)` is the function form of the syntax. Use this form when the variable name or function name is stored in a string.

Remarks

When you use `clear` in a function, it has the following effect on items in your function and base workspaces:

- `clear name` — If `name` is the name of a function, the function is cleared in both the function workspace and in your base workspace.
- `clear` **`functions`** — All functions are cleared in both the function workspace and in your base workspace.
- `clear` **`global`** — All global variables are cleared in both the function workspace and in your base workspace.
- `clear` **`all`** — All functions, global variables, and classes are cleared in both the function workspace and in your base workspace.

Limitations

`clear` does not affect the amount of memory allocated to the MATLAB process under UNIX.

The `clear` function does not clear Simulink models. Use `close` instead.

Examples

Given a workspace containing the following variables

```
  Name        Size          Bytes  Class

  c           3x4            1200  cell array
  frame       1x1                  java.awt.Frame
  gbl1        1x1               8  double array (global)
  gbl2        1x1               8  double array (global)
  xint        1x1               1  int8 array
```

you can clear a single variable, `xint`, by typing

```
clear xint
```

To clear all global variables, type

```
clear global
whos
  Name       Size          Bytes  Class

  c          3x4            1200  cell array
  frame      1x1                  java.awt.Frame
```

Using regular expressions, clear those variables with names that begin with `Mon`, `Tue`, or `Wed`:

```
clear('-regexp', '^Mon|^Tue|^Wed');
```

To clear all compiled M- and MEX-functions from memory, type `clear functions`. In the case shown below, `clear functions` was unable to clear one M-file function from memory, `testfun`, because the function is locked.

```
clear functions    % Attempt to clear all functions.
```

```
inmem

ans =
   'testfun'       % One M-file function remains in memory.

mislocked testfun
ans =
    1              % This function is locked in memory.
```

Once you unlock the function from memory, you can clear it.

```
munlock testfun
clear functions

inmem
ans =
   Empty cell array: 0-by-1
```

See Also

clc, close, import, inmem, load, mlock, munlock, pack, persistent, save, who, whos, workspace

Purpose

Remove serial port object from MATLAB workspace

Syntax

```
clear obj
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`clear obj` removes `obj` from the MATLAB workspace.

Remarks

If `obj` is connected to the device and it is cleared from the workspace, then `obj` remains connected to the device. You can restore `obj` to the workspace with the `instrfind` function. A serial port object connected to the device has a `Status` property value of `open`.

To disconnect `obj` from the device, use the `fclose` function. To remove `obj` from memory, use the `delete` function. You should remove invalid serial port objects from the workspace with `clear`.

Example

This example creates the serial port object `s`, copies `s` to a new variable `scopy`, and clears `s` from the MATLAB workspace. `s` is then restored to the workspace with `instrfind` and is shown to be identical to `scopy`.

```
s = serial('COM1');
scopy = s;
clear s
s = instrfind;
isequal(scopy,s)
ans =
     1
```

See Also

Functions

`delete`, `fclose`, `instrfind`, `isvalid`

Properties

`Status`

Purpose

Clear current figure window

GUI Alternatives

Use **Clear Figure** from the figure window's **File** menu to clear the contents of a figure. You can also create a *desktop shortcut* to clear the current figure with one mouse click. See "Shortcuts for MATLAB — Easily Run a Group of Statements" in the MATLAB Desktop Environment documentation.

Syntax

```
clf('reset')
clf(fig)
clf(fig,'reset')
figure_handle = clf(...)
```

Description

`clf` deletes from the current figure all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to `on`).

`clf('reset')` deletes from the current figure all graphics objects regardless of the setting of their `HandleVisibility` property and resets all figure properties except `Position`, `Units`, `PaperPosition`, and `PaperUnits` to their default values.

`clf(fig)` or `clf(fig,'reset')` clears the single figure with handle `fig`.

`figure_handle = clf(...)` returns the handle of the figure. This is useful when the figure `IntegerHandle` property is `off` because the noninteger handle becomes invalid when the reset option is used (i.e., `IntegerHandle` is reset to `on`, which is the default).

Remarks

The `clf` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `clf` deletes only those objects whose `HandleVisibility` property is set to `on`.

See Also

`cla`, `clc`, `hold`, `reset`

"Figure Windows" on page 1-95 for related functions

Purpose

Copy and paste strings to and from system clipboard

Graphical Interface

As an alternative to `clipboard`, use the Import Wizard. To use the Import Wizard to copy data from the clipboard, select **Paste to Workspace** from the **Edit** menu.

Syntax

```
clipboard('copy', data)
str = clipboard('paste')
data = clipboard('pastespecial')
```

Description

`clipboard('copy', data)` sets the clipboard contents to `data`. If `data` is not a character array, the clipboard uses `mat2str` to convert it to a string.

`str = clipboard('paste')` returns the current contents of the clipboard as a string or as an empty string (`' '`), if the current clipboard contents cannot be converted to a string.

`data = clipboard('pastespecial')` returns the current contents of the clipboard as an array using `uiimport`.

Note Requires an active X display on UNIX, and Java elsewhere.

See Also

`load`, `uiimport`

clock

Purpose

Current time as date vector

Syntax

```
c = clock
```

Description

`c = clock` returns a 6-element date vector containing the current date and time in decimal form:

```
c = [year month day hour minute seconds]
```

The first five elements are integers. The `seconds` element is accurate to several digits beyond the decimal point. The statement `fix(clock)` rounds to integer display format.

Remarks

When timing the duration of an event, use the `tic` and `toc` functions instead of `clock` or `etime`. These latter two functions are based on the system time which can be adjusted periodically by the operating system and thus might not be reliable in time comparison operations.

See Also

`cputime`, `datenum`, `datevec`, `etime`, `tic`, `toc`

Purpose

Remove specified figure

Syntax

```
close
close(h)
close name
close all
close all hidden
status = close(...)
```

Description

`close` deletes the current figure or the specified figure(s). It optionally returns the status of the close operation.

`close` deletes the current figure (equivalent to `close(gcf)`).

`close(h)` deletes the figure identified by h. If h is a vector or matrix, `clse` deletes all figures identified by h.

`close name` deletes the figure with the specified name.

`close all` deletes all figures whose handles are not hidden.

`close all hidden` deletes all figures including those with hidden handles.

`status = close(...)` returns `1` if the specified windows have been deleted and `0` otherwise.

Remarks

The `close` function works by evaluating the specified figure's `CloseRequestFcn` property with the statement

```
eval(get(h,'CloseRequestFcn'))
```

The default `CloseRequestFcn`, `closereq`, deletes the current figure using `delete(get(0,'CurrentFigure'))`. If you specify multiple figure handles, `close` executes each figure's `CloseRequestFcn` in turn. If MATLAB encounters an error that terminates the execution of a `CloseRequestFcn`, the figure is not deleted. Note that using your computer's window manager (i.e., the **Close** menu item) also calls the figure's `CloseRequestFcn`.

If a figure's handle is hidden (i.e., the figure's `HandleVisibility` property is set to `callback` or `off` and the root `ShowHiddenHandles` property is set to `on`), you must specify the `hidden` option when trying to access a figure using the `all` option.

To delete all figures unconditionally, use the statements

```
set(0,'ShowHiddenHandles','on')
delete(get(0,'Children'))
```

The `delete` function does not execute the figure's `CloseRequestFcn`; it simply deletes the specified figure.

The figure `CloseRequestFcn` allows you to either delay or abort the closing of a figure once the `close` function has been issued. For example, you can display a dialog box to see if the user really wants to delete the figure or save and clean up before closing.

See Also

`delete`, `figure`, `gcf`

The figure `HandleVisibility` property

The root `ShowHiddenHandles` property

"Figure Windows" on page 1-95 for related functions

Purpose

Close Audio/Video Interleaved (AVI) file

Syntax

```
aviobj = close(aviobj)
```

Description

`aviobj = close(aviobj)` finishes writing and closes the AVI file associated with `aviobj`, which is an AVI file object created using the `avifile` function.

See Also

`avifile`, `addframe`, `movie2avi`

close (ftp)

Purpose

Close connection to FTP server

Syntax

```
close(f)
```

Description

`close(f)` closes the connection to the FTP server, represented by object `f`, which was created using `ftp`. Be sure to use `close` after completing work on the server. If you do not run `close`, the connection will be terminated automatically either because of the server's time-out feature or by exiting MATLAB.

Examples

Connect to the MathWorks FTP server and then disconnect.

```
tmw=ftp('ftp.mathworks.com');
close(tmw)
```

See Also

`ftp`

Purpose

Default figure close request function

Syntax

```
closereq
```

Description

`closereq` deletes the current figure.

See Also

The figure `CloseRequestFcn` property

"Figure Windows" on page 1-95 for related functions

Purpose

Name of source control system

GUI Alternatives

As an alternative to `cmopts`, select **File > Preferences > General > Source Control** to view the currently selected source control system.

Syntax

```
cmopts
```

Description

`cmopts` displays the name of the source control system you selected using preferences, which is one of the following:

- `clearcase` (UNIX only)
- `customverctrl` (UNIX only)
- `cvs` (UNIX only)
- `pvcs` (UNIX only, used for PVCS and ChangeMan)
- `rcs` (UNIX only)
- `sourcesafe` (Windows only)

If you have not selected a source control system, `cmopts` displays

```
none
```

For more information, see "Specify Source Control System in MATLAB" for PC platforms, and "Specifying the Source Control System on UNIX" for UNIX platforms in the MATLAB Desktop Tools and Development Environment documentation.

Examples

Type

```
cmopts
```

and MATLAB returns

```
ans =
Microsoft Visual SourceSafe
```

which is the source control system specified in preferences.

See Also

`checkin`, `checkout`, `customverctrl`, `verctrl`

Purpose

Column approximate minimum degree permutation

Syntax

```
p = colamd(S)
```

Description

`p = colamd(S)` returns the column approximate minimum degree permutation vector for the sparse matrix `S`. For a non-symmetric matrix `S`, `S(:,p)` tends to have sparser LU factors than `S`. The Cholesky factorization of `S(:,p)' * S(:,p)` also tends to be sparser than that of `S'*S`.

`knobs` is a two-element vector. If S is `m`-by-`n`, then rows with more than `(knobs(1))*n` entries are ignored. Columns with more than `(knobs(2))*m` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs(1) = knobs(2) = spparms('wh_frac')`.

`stats` is an optional vector that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `colamd`
`stats(2)`	Number of dense or empty columns ignored by `colamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `colamd` (roughly of size `2.2*nnz(S) + 4*m + 7*n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid
`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists

`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `colamd`. For this reason, `colamd` verifies that `S` is valid:

- If a row index appears two or more times in the same column, `colamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `colamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `colamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`) .

The ordering is followed by a column elimination tree post-ordering.

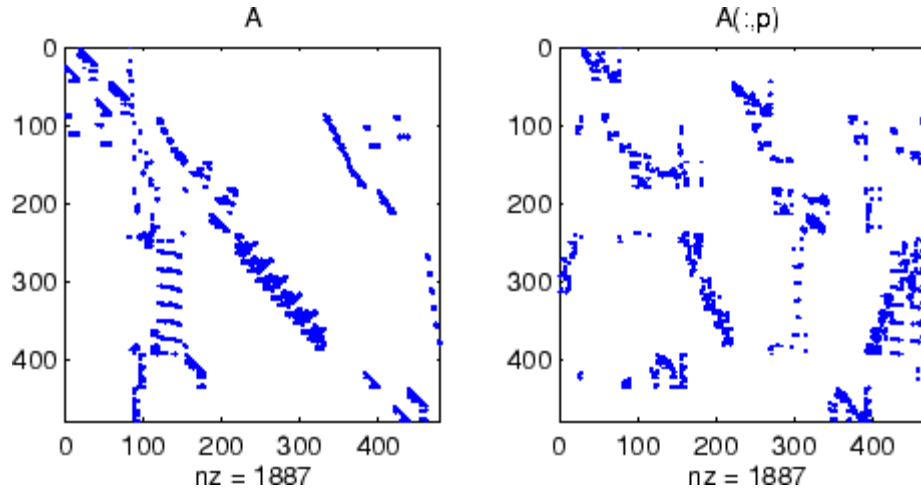
Note `colamd` tends to be faster than `colmmd` and tends to return a better ordering.

Examples

The Harwell-Boeing collection of sparse matrices and the MATLAB demos directory include a test matrix `west0479`. It is a matrix of order 479 resulting from a model due to Westerberg of an eight-stage chemical distillation column. The spy plot shows evidence of the eight stages. The `colamd` ordering scrambles this structure.

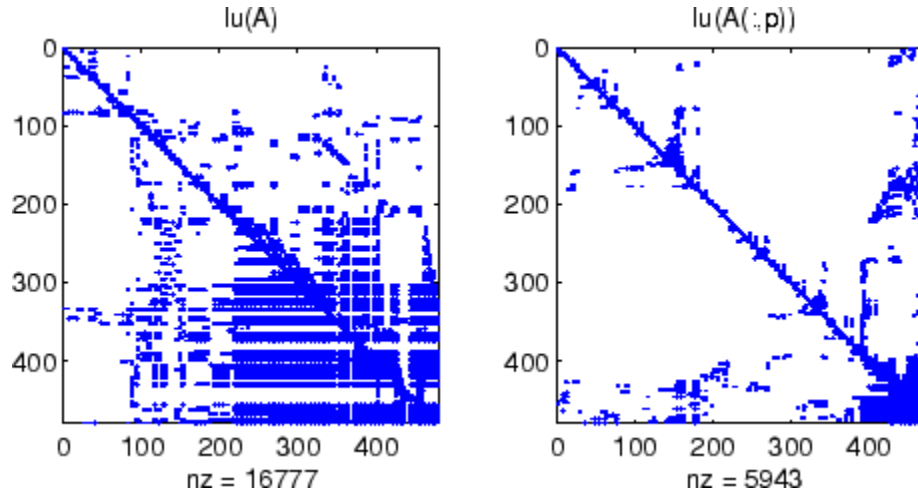
```
load west0479
```

```
A = west0479;
p = colamd(A);
subplot(1,2,1), spy(A,4), title('A')
subplot(1,2,2), spy(A(:,p),4), title('A(:,p)')
```



Comparing the spy plot of the LU factorization of the original matrix with that of the reordered matrix shows that minimum degree reduces the time and storage requirements by better than a factor of 2.8. The nonzero counts are 16777 and 5904, respectively.

```
spy(lu(A),4)
spy(lu(A(:,p)),4)
```



See Also

`colperm`, `spparms`, `symamd`, `symrcm`

References

[1] The authors of the code for "colamd" are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. Sparse Matrix Algorithms Research at the University of Florida: `http://www.cise.ufl.edu/research/sparse/`

colmmd

Purpose

Sparse column minimum degree permutation

Syntax

```
p = colmmd(S)
```

Note `colmmd` is obsolete and will be removed from a future version of MATLAB. Use `colamd` instead.

Description

`p = colmmd(S)` returns the column minimum degree permutation vector for the sparse matrix `S`. For a nonsymmetric matrix `S`, this is a column permutation `p` such that `S(:,p)` tends to have sparser LU factors than `S`.

The `colmmd` permutation is automatically used by `\` and `/` for the solution of nonsymmetric and symmetric indefinite sparse linear systems.

Use `spparms` to change some options and parameters associated with heuristics in the algorithm.

Algorithm

The minimum degree algorithm for symmetric matrices is described in the review paper by George and Liu [1]. For nonsymmetric matrices, the MATLAB minimum degree algorithm is new and is described in the paper by Gilbert, Moler, and Schreiber [2]. It is roughly like symmetric minimum degree for `A'*A`, but does not actually form `A'*A`.

Each stage of the algorithm chooses a vertex in the graph of `A'*A` of lowest degree (that is, a column of `A` having nonzero elements in common with the fewest other columns), eliminates that vertex, and updates the remainder of the graph by adding fill (that is, merging rows). If the input matrix `S` is of size `m`-by-`n`, the columns are all eliminated and the permutation is complete after `n` stages. To speed up the process, several heuristics are used to carry out multiple stages simultaneously.

See Also

`colamd`, `colperm`, `lu`, `spparms`, `symamd`, `symmmd`, `symrcm`

The arithmetic operator `\`

References

[1] George, Alan and Liu, Joseph, "The Evolution of the Minimum Degree Ordering Algorithm," *SIAM Review*, 1989, 31:1-19.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications 13*, 1992, pp. 333-356.

colorbar

Purpose

Colorbar showing color scale

GUI Alternatives

Add a colorbar to a plot with the colorbar tool  on the figure toolbar, or use **Insert —> Colorbar** from the figure menu. Use the Property Editor to modify the position, font and other properties of a legend. . For details, see “Working in Plot Edit Mode” in the MATLAB Graphics documentation.

Syntax

```
colorbar
colorbar(...,'peer',axes_handle)
colorbar(...,'location')
colorbar(...,'PropertyName',propertyvalue)
cbar_axes = colorbar(...)
colorbar(axes_handle)
```

Description

The `colorbar` function displays the current colormap in the current figure and resizes the current axes to accommodate the colorbar.

`colorbar` adds a new vertical colorbar on the right side of the current axes. If a colorbar exists in that location, `colorbar` replaces it with a new one. If a colorbar exists at a nondefault location, it is retained along with the new colorbar

`colorbar(...,'peer',axes_handle)` creates a colorbar associated with the axes `axes_handle` instead of the current axes.

`colorbar(...,'location')` adds a colorbar in the specified orientation with respect to the axes. If a colorbar exists at the location specified, it is replaced. Any colorbars not occupying the specified location are retained. Possible values for *`location`* are

`North`	Inside plot box near top
`South`	Inside bottom
`East`	Inside right
`West`	Inside left

`NorthOutside`	Outside plot box near top
`SouthOutside`	Outside bottom
`EastOutside`	Outside right
`WestOutside`	Outside left

Using one of the `...Outside` values for *`location`* ensures that the colorbar does not overlap the plot, whereas overlaps can occur when you specify any of the other four values.

`colorbar(...,'`*`PropertyName`*`',propertyvalue)` specifies property names and values for the axes object used to create the colorbar. See axes properties for a description of the properties you can set. The *`location`* property applies only to colorbars and legends, not to axes.

`cbar_axes = colorbar(...)` returns a handle to the colorbar, which is an axes graphics object that contains one additional property, `Location`.

Backward-Compatible Version

`h = colorbar('v6',...)` creates a colorbar compatible with MATLAB 6.5 and earlier. It returns the handles of patch objects instead of a colorbar object.

`colorbar(axes_handle)` adds the colorbar to the axes `axes_handle` in the default (`right`) orientation. As in Version 6 and earlier releases, no new axes is created.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

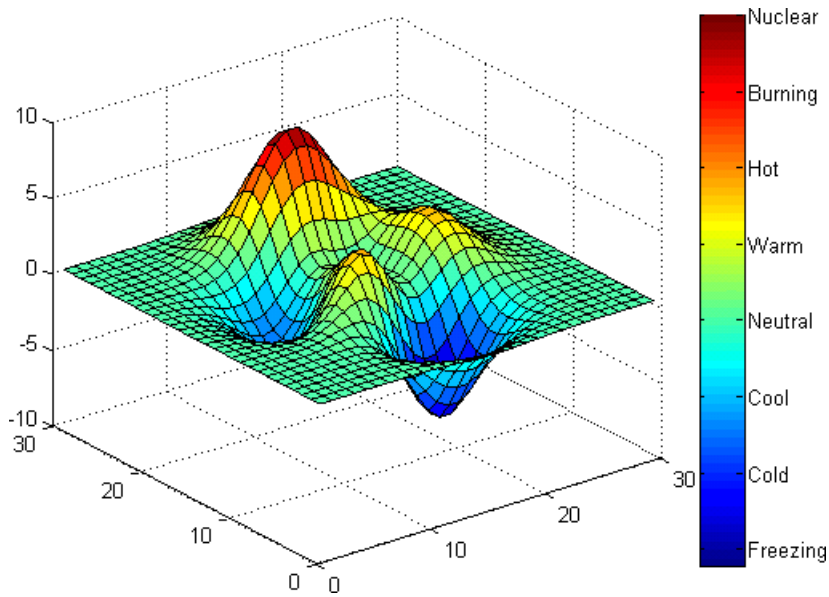
You can use `colorbar` with 2-D and 3-D plots.

Examples

Example 1

Display a colorbar beside the axes and use descriptive text strings as *y*-tick labels. Note that labels will repeat cyclically when the number of *y*-ticks is greater than the number of labels, and not all labels will appear if there are fewer *y*-ticks than labels you have specified. Also note that when colorbars are horizontal, their ticks and labels are governed by the `XTick` property rather than the `YTick` property. For more information, see "Labeling Colorbar Ticks".

```
surf(peaks(30))
colorbar('YTickLabel',...
    {'Freezing','Cold','Cool','Neutral',...
     'Warm','Hot','Burning','Nuclear'})
```

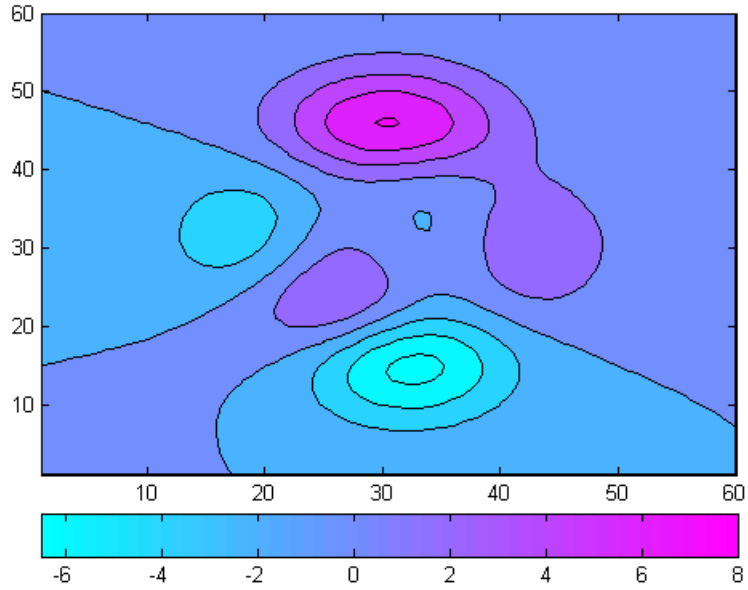


Example 2

Display a horizontal colorbar beneath the axes of a filled contour plot:

```
contourf(peaks(60))
```

```
colormap cool
colorbar('location','southoutside')
```



See Also

`colormap`

"Color Operations" on page 1-98 for related functions

colordef

Purpose

Set default property values to display different color schemes

Syntax

```
colordef white
colordef black
colordef none
colordef(fig,color_option)
h = colordef('new',color_option)
```

Description

`colordef` enables you to select either a white or black background for graphics display. It sets axis lines and labels so that they contrast with the background color.

`colordef white` sets the axis background color to white, the axis lines and labels to black, and the figure background color to light gray.

`colordef black` sets the axis background color to black, the axis lines and labels to white, and the figure background color to dark gray.

`colordef none` sets the figure coloring to that used by MATLAB Version 4. The most noticeable difference is that the axis background is set to `'none'`, making the axis background and figure background colors the same. The figure background color is set to black.

`colordef(fig,color_option)` sets the color scheme of the figure identified by the handle `fig` to one of the color options `'white'`, `'black'`, or `'none'`. When you use this syntax to apply `colordef` to an existing figure, the figure must have no graphic content. If it does, you should first clear it (via `clf`) before using this form of the command.

`h = colordef('new',color_option)` returns the handle to a new figure created with the specified color options (i.e., `'white'`, `'black'`, or `'none'`). This form of the command is useful for creating GUIs when you may want to control the default environment. The figure is created with `'visible','off'` to prevent flashing.

Remarks

`colordef` affects only subsequently drawn figures, not those currently on the display. This is because `colordef` works by setting default property values (on the root or figure level). You can list the currently set default values on the root level with the statement

```
get(0,'defaults')
```

You can remove all default values using the `reset` command:

```
reset(0)
```

See the `get` and `reset` references pages for more information.

See Also

`whitebg`, `clf`

"Color Operations" on page 1-98 for related functions

colormap

Purpose

Set and get current colormap

GUI Alternatives

Select a built-in colormap with the Property Editor. To modify the current colormap, use the Colormap Editor, accessible from **Edit —> Colormap** on the figure menu.

Syntax

```
colormap(map)
colormap('default')
cmap = colormap
```

Description

A colormap is an *m*-by-3 matrix of real numbers between 0.0 and 1.0. Each row is an RGB vector that defines one color. The *k*th row of the colormap defines the *k*th color, where `map(k,:) = [r(k) g(k) b(k)]` specifies the intensity of red, green, and blue.

`colormap(map)` sets the colormap to the matrix `map`. If any values in `map` are outside the interval [0 1], MATLAB returns the error `Colormap must have values in [0,1]`.

`colormap('default')` sets the current colormap to the default colormap.

`cmap = colormap` retrieves the current colormap. The values returned are in the interval [0 1].

Specifying Colormaps

M-files in the `color` directory generate a number of colormaps. Each M-file accepts the colormap size as an argument. For example,

```
colormap(hsv(128))
```

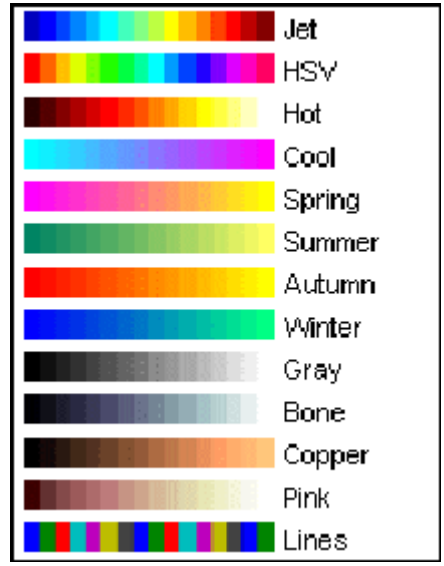
creates an `hsv` colormap with 128 colors. If you do not specify a size, MATLAB creates a colormap the same size as the current colormap.

Supported Colormaps

MATLAB supports a number of built-in colormaps, illustrated and described below. In addition to specifying built-in colormaps

programmatically, you can use the **Colormap** menu in the **Figure Properties** pane of the **Plot Tools** GUI to select one interactively.

The named built-in colormaps are the following:



- `autumn` varies smoothly from red, through orange, to yellow.
- `bone` is a grayscale colormap with a higher value for the blue component. This colormap is useful for adding an “electronic” look to grayscale images.
- `colorcube` contains as many regularly spaced colors in RGB colorspace as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
- `cool` consists of colors that are shades of cyan and magenta. It varies smoothly from cyan to magenta.
- `copper` varies smoothly from black to bright copper.

- `flag` consists of the colors red, white, blue, and black. This colormap completely changes color with each index increment.
- `gray` returns a linear grayscale colormap.
- `hot` varies smoothly from black through shades of red, orange, and yellow, to white.
- `hsv` varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for displaying periodic functions. `hsv(m)` is the same as `hsv2rgb([h ones(m,2)])` where `h` is the linear ramp, `h = (0:m 1)'/m`.
- `jet` ranges from blue to red, and passes through the colors cyan, yellow, and orange. It is a variation of the `hsv` colormap. The `jet` colormap is associated with an astrophysical fluid jet simulation from the National Center for Supercomputer Applications. See the "Examples" on page 2-586 section.
- `lines` produces a colormap of colors specified by the axes `ColorOrder` property and a shade of gray.
- `pink` contains pastel shades of pink. The pink colormap provides sepia tone colorization of grayscale photographs.
- `prism` repeats the six colors red, orange, yellow, green, blue, and violet.
- `spring` consists of colors that are shades of magenta and yellow.
- `summer` consists of colors that are shades of green and yellow.
- `white` is an all white monochrome colormap.
- `winter` consists of colors that are shades of blue and green.

Examples

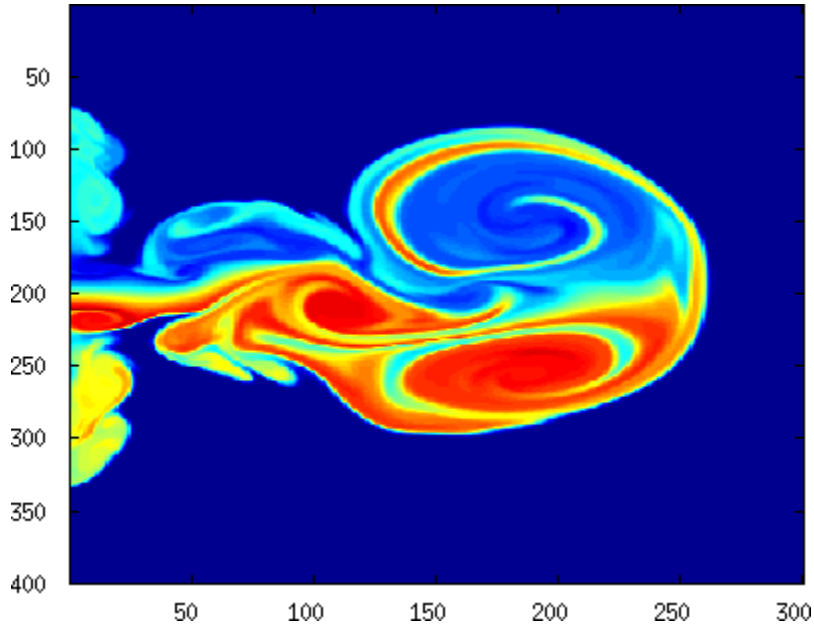
The images and colormaps demo, `imagedemo`, provides an introduction to colormaps. Select **Color Spiral** from the menu. This uses the `pcolor` function to display a 16-by-16 matrix whose elements vary from 0 to 255 in a rectilinear spiral. The `hsv` colormap starts with red in the center,

then passes through yellow, green, cyan, blue, and magenta before returning to red at the outside end of the spiral. Selecting **Colormap Menu** gives access to a number of other colormaps.

The `rgbplot` function plots colormap values. Try `rgbplot(hsv)`, `rgbplot(gray)`, and `rgbplot(hot)`.

The following commands display the `flujet` data using the `jet` colormap.

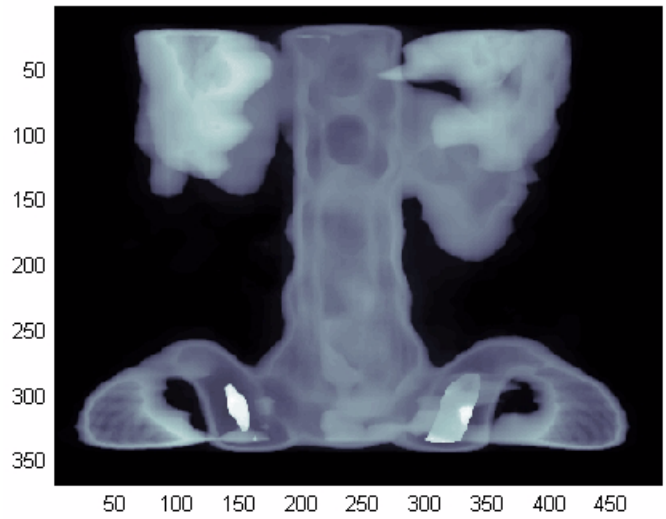
```
load flujet
image(X)
colormap(jet)
```



The `demos` directory contains a CAT scan image of a human spine. To view the image, type the following commands:

```
load spine
image(X)
```

```
colormap bone
```



Algorithm

Each figure has its own `Colormap` property. `colormap` is an M-file that sets and gets this property.

See Also

`brighten`, `caxis`, `colormapeditor`, `colorbar`, `contrast`, `hsv2rgb`, `pcolor`, `rgb2hsv`, `rgbplot`

The `Colormap` property of figure graphics objects

“Color Operations” on page 1-98 for related functions

“Coloring Mesh and Surface Plots” for more information about colormaps and other coloring methods

Purpose

Start colormap editor

Syntax

```
colormapeditor
```

Description

`colormapeditor` displays the current figure's colormap as a strip of rectangular cells in the colormap editor. Node pointers are colored cells below the colormap strip that indicate points in the colormap where the rate of the variation of R, G, and B values changes. You can also work in the HSV colorspace by setting the **Interpolating Colorspace** selector to HSV.

You can also start the colormap editor by selecting **Colormap** from the **Edit** menu.

Node Pointer Operations

You can select and move node pointers to change a range of colors in the colormap. The color of a node pointer remains constant as you move it, but the colormap changes by linearly interpolating the RGB values between nodes.

Change the color at a node by double-clicking the node pointer. MATLAB displays a color picker from which you can select a new color. After you select a new color at a node, MATLAB reinterpolates the colors in between nodes.

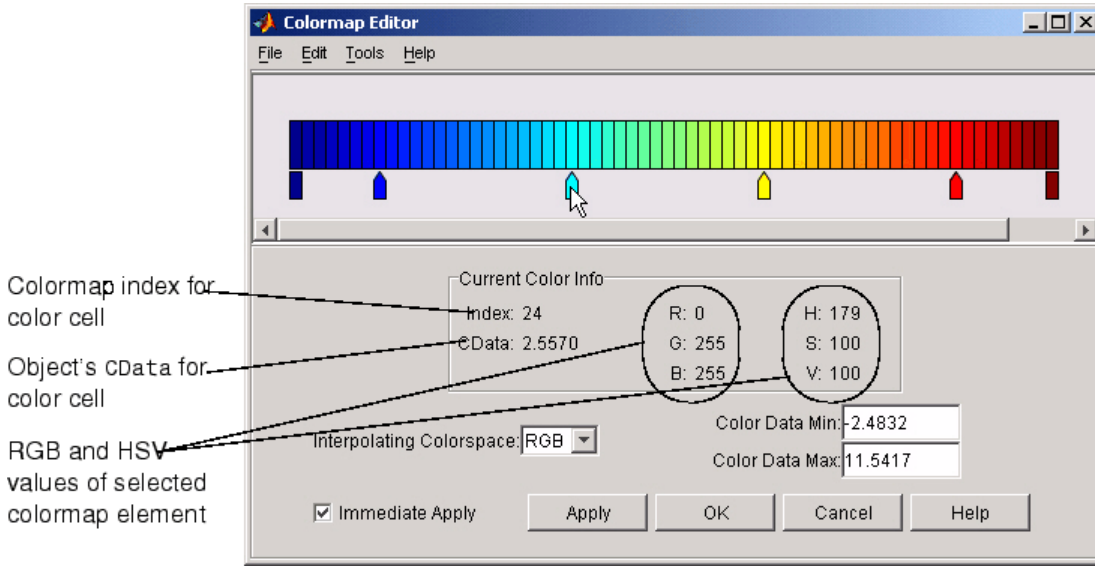
Operation	How to Perform
Add a node	Click below the corresponding cell in the colormap strip.
Select a node	Left-click the node.
Select multiple nodes	Adjacent: left-click first node, **Shift+click** the last node. Nonadjacent: left-click first node, **Ctrl+click** subsequent nodes.

Operation	How to Perform
Move a node	Select and drag with the mouse or select and use the left and right arrow keys.
Move multiple nodes	Select multiple nodes and use the left and right arrow keys to move nodes as a group. Movement stops when one of the selected nodes hits an unselected node or an end node.
Delete a node	Select the node and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Delete multiple nodes	Select the nodes and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Display color picker for a node	Double-click the node pointer.

Current Color Info

When you put the mouse over a color cell or node pointer, the colormap editor displays the following information about that colormap element:

- The element's index in the colormap
- The value from the graphics object color data that is mapped to the node's color (i.e., data from the `CData` property of any image, patch, or surface objects in the figure)
- The color's RGB and HSV color value



Interpolating Colorspace

The colorspace determines what values are used to calculate the colors of cells between nodes. For example, in the RGB colorspace, internode colors are calculated by linearly interpolating the red, green, and blue intensity values from one node to the next. Switching to the HSV colorspace causes the colormap editor to recalculate the colors between nodes using the hue, saturation, and value components of the color definition.

Note that when you switch from one colorspace to another, the color editor preserves the number, color, and location of the node pointers, which can cause the colormap to change.

Interpolating in HSV. Since hue is conceptually mapped about a color circle, the interpolation between hue values can be ambiguous. To minimize this ambiguity, the interpolation uses the shortest distance around the circle. For example, interpolating between two nodes, one with hue of 2 (slightly orange red) and another with a hue of 356 (slightly magenta red), does not result in hues 3,4,5...353,354,355 (orange/red-yellow-green-cyan-blue-magenta/red). Taking the shortest distance around the circle gives 357,358,1,2 (orange/red-red-magenta/red).

Color Data Min and Max

The **Color Data Min** and **Color Data Max** text fields enable you to specify values for the axes `CLim` property. These values change the mapping of object color data (the `CData` property of images, patches, and surfaces) to the colormap. See "Axes Color Limits — the CLim Property" for discussion and examples of how to use this property.

Examples

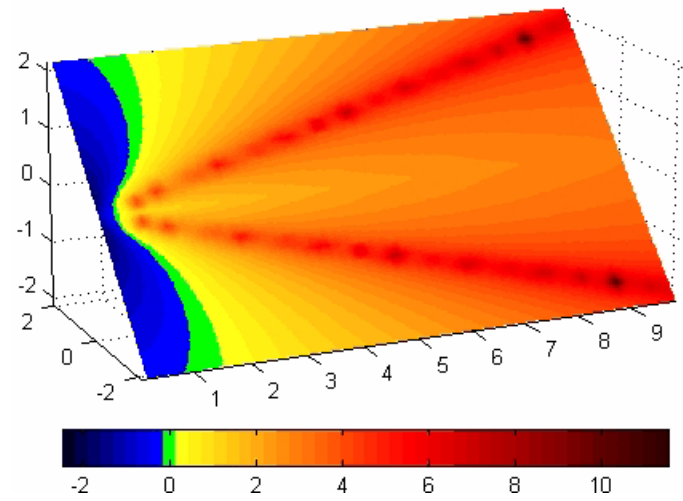
This example modifies a default MATLAB colormap so that ranges of data values are displayed in specific ranges of color. The graph is a slice plane illustrating a cross section of fluid flow through a jet nozzle. See the `slice` reference page for more information on this type of graph.

Example Objectives

The objectives are as follows:

- Regions of flow from left to right (positive data) are mapped to colors from yellow through orange to dark red. Yellow is slowest and dark red is the fastest moving fluid.
- Regions that have a speed close to zero are colored green.
- Regions where the fluid is actually moving right to left (negative data) are shades of blue (darker blue is faster).

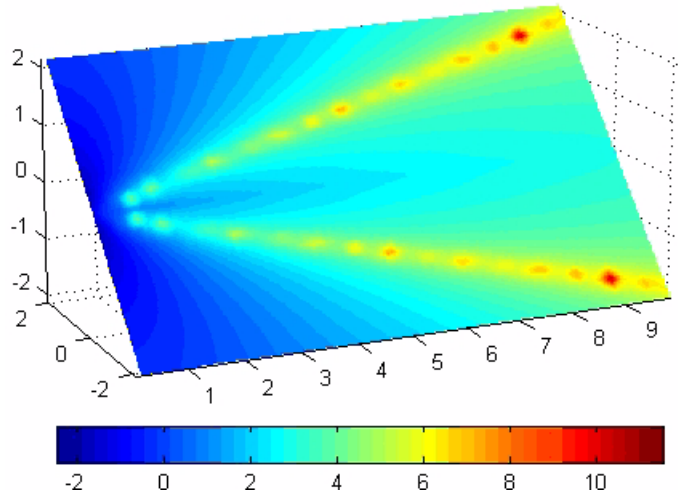
The following picture shows the desired coloring of the slice plane. The colorbar shows the data to color mapping.



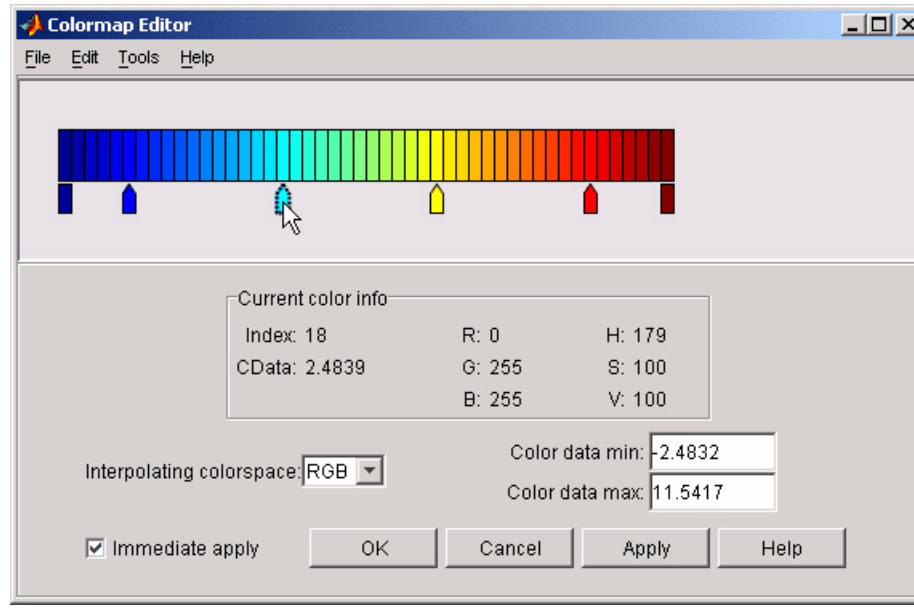
Running the Example

Note If you are viewing this documentation in the MATLAB help browser, you can display the graph used in this example by running this M-file from the MATLAB editor (select **Run** from the **Debug** menu).

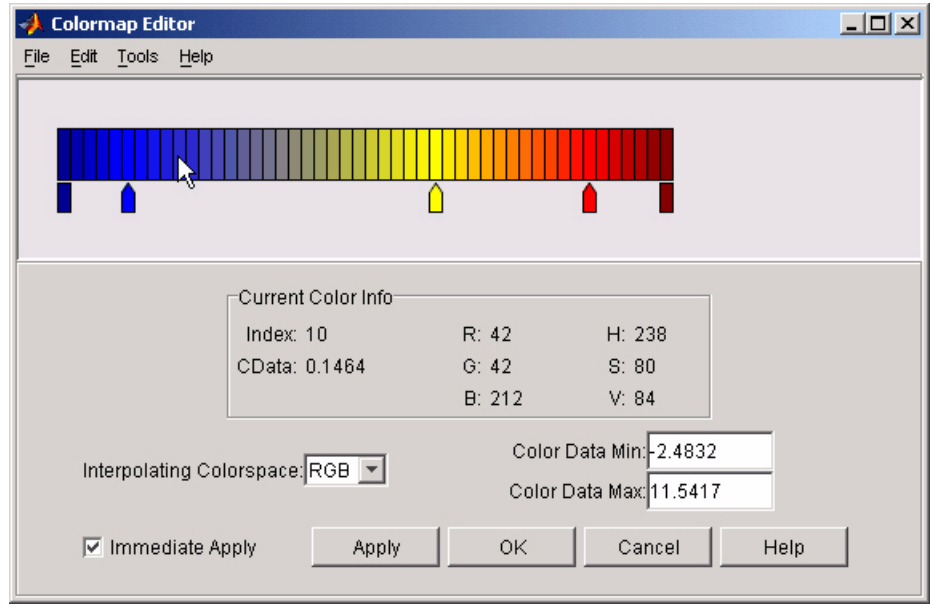
Initially, the default colormap (`jet`) colored the slice plane, as illustrated in the following picture. Note that this example uses a colormap that is 48 elements to display wider bands of color (the default is 64 elements).



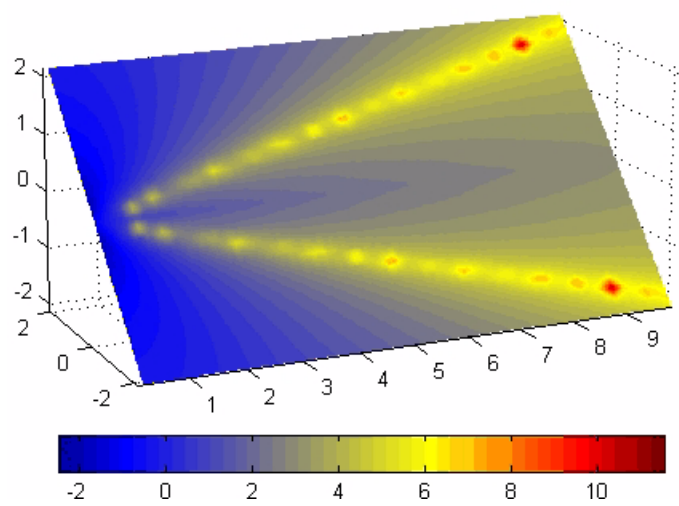
1 Start the colormap editor using the `colormapeditor` command. The color map editor displays the current figure' s colormap, as shown in the following picture.



2 Since we want the regions of left-to-right flow (positive speed) to range from yellow to dark red, we can delete the cyan node pointer. To do this, first select it by clicking with the left mouse button and press **Delete**. The colormap now looks like this.

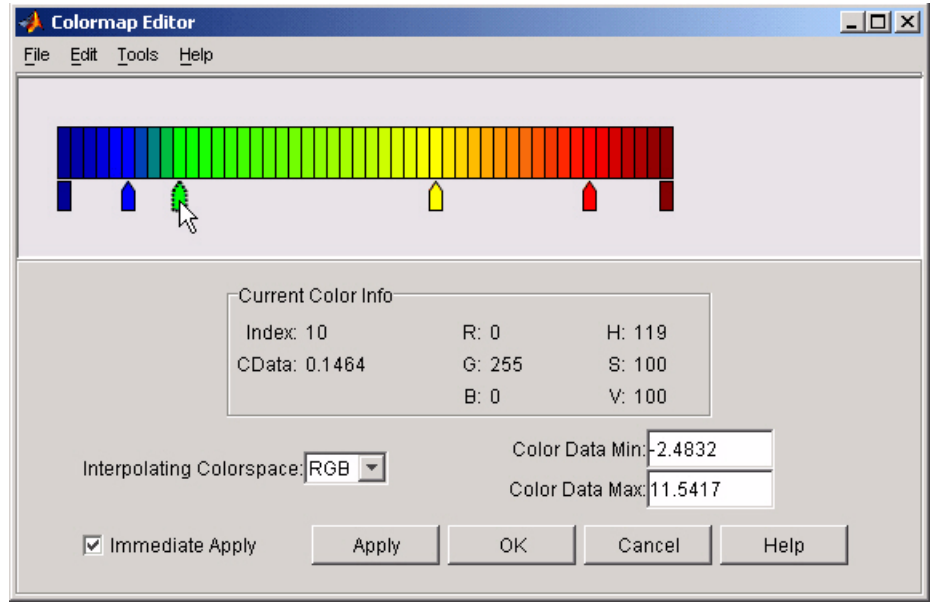


The **Immediate Apply** box is checked, so the graph displays the results of the changes made to the colormap.

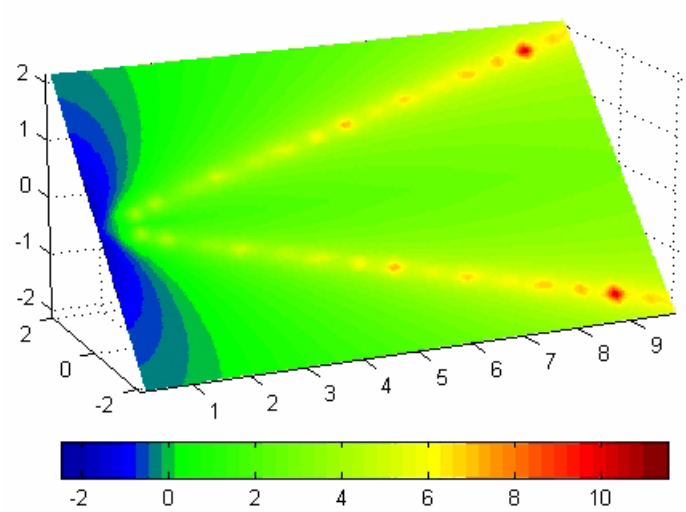


3 We want the fluid speed values around zero to stand out, so we need to find the color cell where the negative-to-positive transition occurs. Dragging the cursor over the color strip enables you to read the data values in the **Current Color Info** panel.

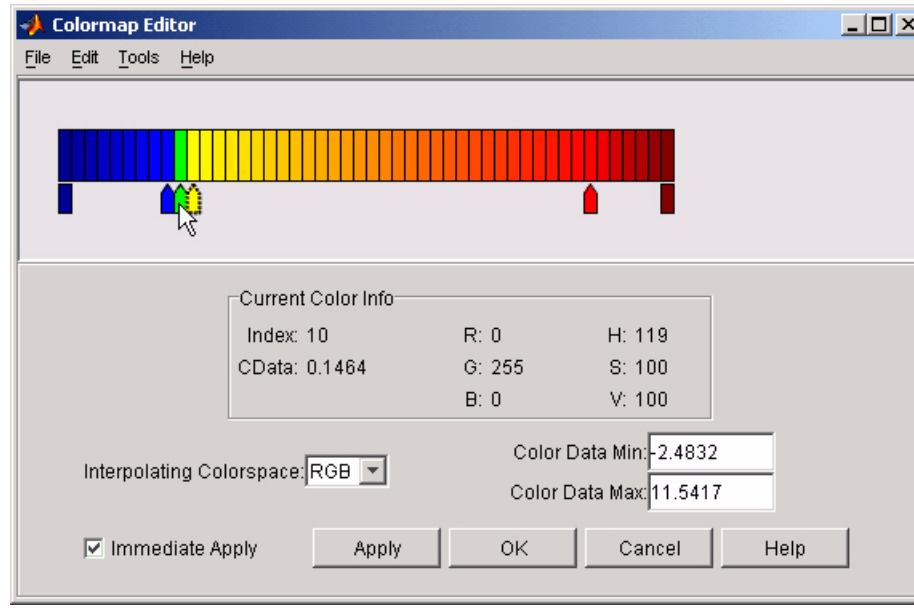
In this case, cell 10 is the first positive value, so we click below that cell and create a node pointer. Double-clicking the node pointer displays the color picker. Set the color of this node to green.



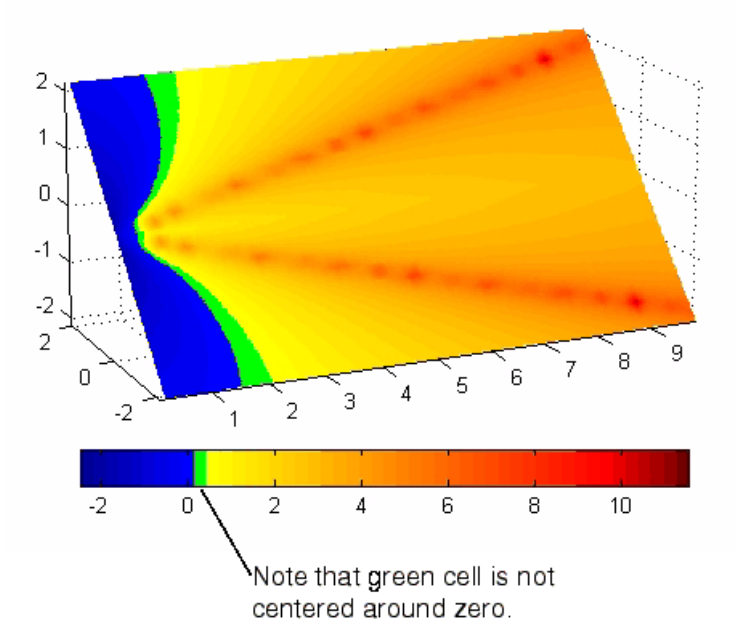
The graph continues to update to the modified colormap.



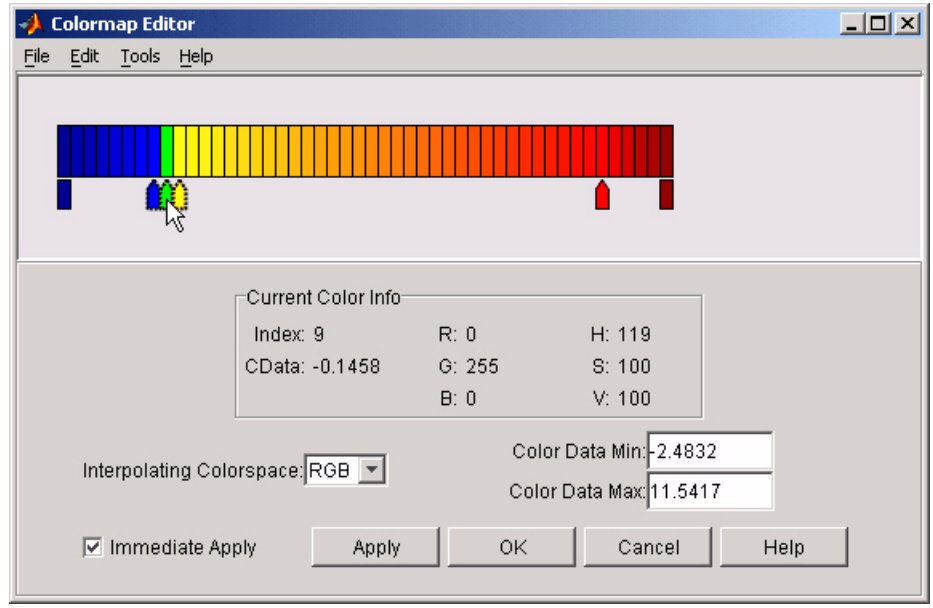
4 In the current state, the colormap colors are interpolated from the green node to the yellowish node about 20 cells away. We actually want only the single cell that is centered around zero to be colored green. To limit the color green to one cell, move the blue and yellow node pointers next to the green pointer.



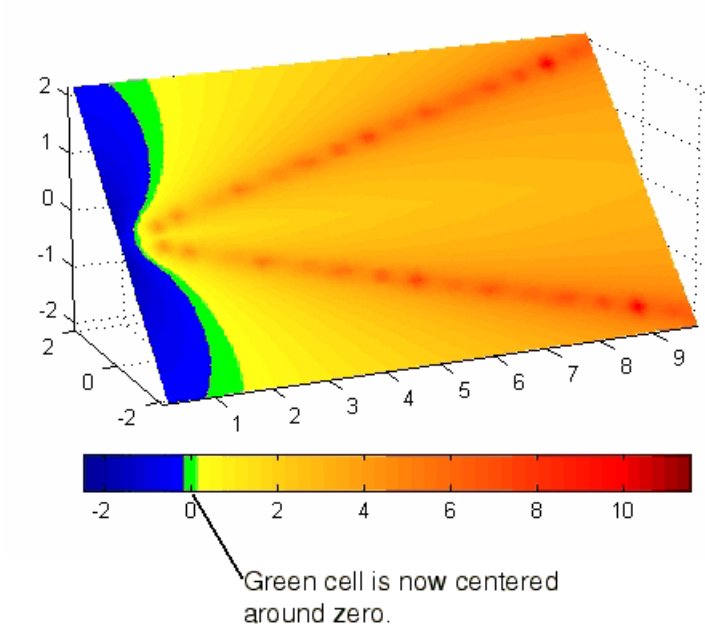
5 Before making further adjustments to the colormap, we need to move the green cell so that it is centered around zero. Use the colorbar to locate the green cell.



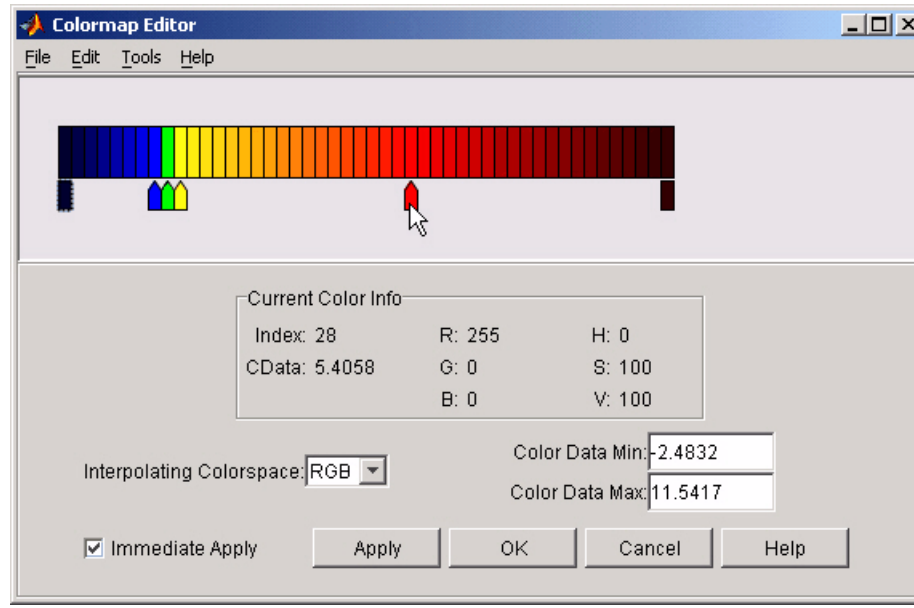
To recenter the green cell around zero, select the blue, green, and yellow node pointers (left-click blue, **Shift+click** yellow) and move them as a group using the left arrow key. Watch the colorbar in the figure window to see when the green color is centered around zero.



The slice plane now has the desired range of colors for negative, zero, and positive data.

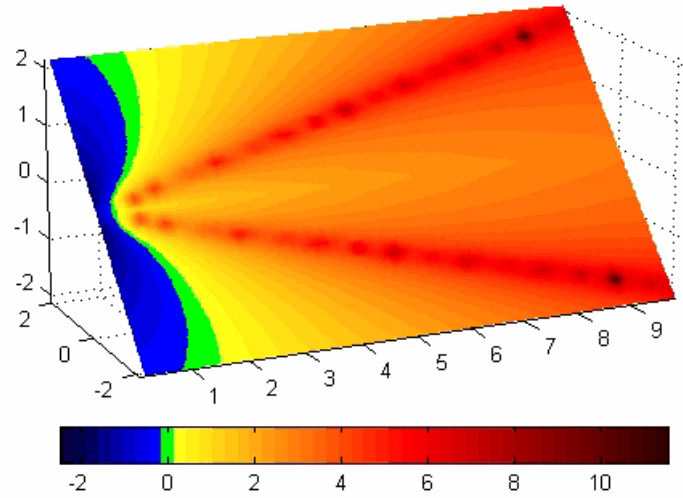


6 Increase the orange-red coloring in the slice by moving the red node pointer toward the yellow node.



7 Darken the endpoints to bring out more detail in the extremes of the data. Double-click the end nodes to display the color picker. Set the red endpoint to the RGB value [50 0 0] and set the blue endpoint to the RGB value [0 0 50].

The slice plane coloring now matches the example objectives.



Saving the Modified Colormap

You can save the modified colormap using the `colormap` function or the figure `Colormap` property.

After you have applied your changes, save the current figure colormap in a variable:

```
mycmap = get(fig,'Colormap'); % fig is figure
handle or use gcf
```

To use this colormap in another figure, set that figure's `Colormap` property:

```
set(new_fig,'Colormap',mycmap)
```

To save your modified colormap in a MAT-file, use the `save` command to save the `mycmap` workspace variable:

```
save('MyColormaps','mycmap')
```

To use your saved colormap in another MATLAB session, `load` the variable into the workspace and assign the colormap to the figure:

```
load('MyColormaps','mycmap')
set(fig,'Colormap',mycmap)
```

See Also

`colormap`, `get`, `load`, `save`, `set`

Color Operations for related functions

See “Colormaps” for more information on using MATLAB colormaps.

Purpose

Color specification

Description

`ColorSpec` is not a function; it refers to the three ways in which you specify color in MATLAB:

- RGB triple
- Short name
- Long name

The short names and long names are MATLAB strings that specify one of eight predefined colors. The RGB triple is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1]. The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Remarks

The eight predefined colors and any colors you specify as RGB values are not part of a figure's colormap, nor are they affected by changes to the figure's colormap. They are referred to as *fixed* colors, as opposed to *colormap* colors.

Some high-level functions (for example, `scatter`) accept a colorspec as an input argument and use it to set the `CData` of graphic objects they

create. When using such functions, take care not to specify a colorspec in a property/value pair that sets `CData`; values for `CData` are always n-length vectors or n-by-3 matrices, where n is the length of `XData` and `YData`, never strings.

Examples

To change the background color of a figure to green, specify the color with a short name, a long name, or an RGB triple. These statements generate equivalent results:

```
whitebg('g')
whitebg('green')
whitebg([0 1 0]);
```

You can use `ColorSpec` anywhere you need to define a color. For example, this statement changes the figure background color to pink:

```
set(gcf,'Color',[1,0.4,0.6])
```

See Also

`bar`, `bar3`, `colordef`, `colormap`, `fill`, `fill3`, `whitebg`

"Color Operations" on page 1-98 for related functions

Purpose

Sparse column permutation based on nonzero count

Syntax

```
j = colperm(S)
```

Description

`j = colperm(S)` generates a permutation vector `j` such that the columns of `S(:,j)` are ordered according to increasing count of nonzero entries. This is sometimes useful as a preordering for LU factorization; in this case use `lu(S(:,j))`.

If `S` is symmetric, then `j = colperm(S)` generates a permutation `j` so that both the rows and columns of `S(j,j)` are ordered according to increasing count of nonzero entries. If `S` is positive definite, this is sometimes useful as a preordering for Cholesky factorization; in this case use `chol(S(j,j))`.

Algorithm

The algorithm involves a sort on the counts of nonzeros in each column.

Examples

The `n`-by-`n` *arrowhead* matrix

```
A = [ones(1,n); ones(n-1,1) speye(n-1,n-1)]
```

has a full first row and column. Its LU factorization, `lu(A)`, is almost completely full. The statement

```
j = colperm(A)
```

returns `j = [2:n 1]`. So `A(j,j)` sends the full row and column to the bottom and the rear, and `lu(A(j,j))` has the same nonzero structure as `A` itself.

On the other hand, the Bucky ball example,

```
B = bucky
```

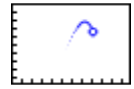
has exactly three nonzero elements in each row and column, so `j = colperm(B)` is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.

See Also

chol, colamd, lu, spparms, symamd, symrcm

Purpose

2-D comet plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
comet(y)
comet(x,y)
comet(x,y,p)
comet(axes_handle,...)
```

Description

A comet graph is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet(y)` displays a comet graph of the vector `y`.

`comet(x,y)` displays a comet graph of vector `y` versus vector `x`.

`comet(x,y,p)` specifies a comet body of length `p*length(y)`. `p` defaults to `0.1`.

`comet(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

Remarks

The trace left by `comet` is created by using an `EraseMode` of `none`, which means you cannot print the graph (you get only the comet head), and it disappears if you cause a redraw (e.g., by resizing the window).

Examples

Create a simple comet graph:

```
t = 0:.01:2*pi;
x = cos(2*t).*(cos(t).^2);
y = sin(2*t).*(sin(t).^2);
comet(x,y);
```

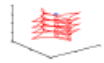
See Also

`comet3`

"Direction and Velocity Plots" on page 1-89 for related functions

Purpose

3-D comet plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
comet3(z)
comet3(x,y,z)
comet3(x,y,z,p)
comet3(axes_handle,...)
```

Description

A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet3(z)` displays a 3-D comet graph of the vector `z`.

`comet3(x,y,z)` displays a comet graph of the curve through the points `[x(i),y(i),z(i)]`.

`comet3(x,y,z,p)` specifies a comet body of length `p*length(y)`.

`comet3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

Remarks

The trace left by `comet3` is created by using an `EraseMode` of `none`, which means you cannot print the graph (you get only the comet head), and it disappears if you cause a redraw (e.g., by resizing the window).

Examples

Create a 3-D comet graph.

```
t = -10*pi:pi/250:10*pi;
comet3((cos(2*t).^2).*sin(t),(sin(2*t).^2).*cos(t),t);
```

See Also

`comet`

"Direction and Velocity Plots" on page 1-89 for related functions

Purpose

Open Command History window, or select it if already open

GUI Alternatives

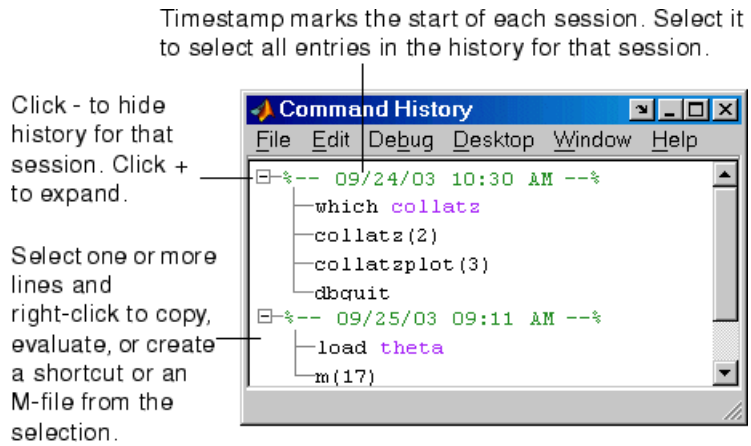
As an alternative to `commandhistory`, select **Desktop > Command History** to open it, or **Window > Command History** to select it.

Syntax

```
commandhistory
```

Description

`commandhistory` opens the MATLAB Command History window when it is closed, and selects the Command History window when it is open. The Command History window presents a log of the statements most recently run in the Command Window.



See Also

`diary`, `prefdir`, `startup`

MATLAB Desktop Tools and Development Environment Documentation

- “Recalling Previous Lines”
- “Command History Window”

commandwindow

Purpose

Open Command Window, or select it if already open

GUI Alternatives

As an alternative to `commandwindow`, select **Desktop > Command Window** to open it, or **Window > Command Window** to select it.

Syntax

```
commandwindow
```

Description

`commandwindow` opens the MATLAB Command Window when it is closed, and selects the Command Window when it is open.

Remarks

To determine the number of columns and rows that display in the Command Window, given its current size, use

```
get(0,'CommandWindowSize')
```

The number of columns is based on the width of the Command Window. With the matrix display width preference set to 80 columns, the number of columns is always 80.

See Also

`commandhistory`, `input`, `inputdlg`

MATLAB Desktop Tools and Development Environment documentation

- “Opening and Arranging Tools”
- “Running Functions and Programs, and Entering Variables”
- “Preferences for the Command Window”

Purpose

Companion matrix

Syntax

```
A = compan(u)
```

Description

`A = compan(u)` returns the corresponding companion matrix whose first row is `-u(2:n)/u(1)`, where `u` is a vector of polynomial coefficients. The eigenvalues of `compan(u)` are the roots of the polynomial.

Examples

The polynomial $(x-1)(x-2)(x+3) = x^3 - 7x + 6$ has a companion matrix given by

```
u = [1  0  -7  6]
A = compan(u)
A =
     0     7    -6
     1     0     0
     0     1     0
```

The eigenvalues are the polynomial roots:

```
eig(compan(u))

ans =
    -3.0000
     2.0000
     1.0000
```

This is also `roots(u)`.

See Also

`eig`, `poly`, `polyval`, `roots`

Purpose

Plot arrows emanating from origin

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
compass(U,V)
compass(Z)
compass(...,LineSpec)
compass(axes_handle,...)
h = compass(...)
```

Description

A compass graph displays the vectors with components `(U,V)` as arrows emanating from the origin. `U`, `V`, and `Z` are in Cartesian coordinates and plotted on a circular grid.

`compass(U,V)` displays a compass graph having *n* arrows, where *n* is the number of elements in `U` or `V`. The location of the base of each arrow is the origin. The location of the tip of each arrow is a point relative to the base and determined by `[U(i),V(i)]`.

`compass(Z)` displays a compass graph having *n* arrows, where *n* is the number of elements in `Z`. The location of the base of each arrow is the origin. The location of the tip of each arrow is relative to the base as determined by the real and imaginary components of `Z`. This syntax is equivalent to `compass(real(Z),imag(Z))`.

`compass(...,LineSpec)` draws a compass graph using the line type, marker symbol, and color specified by `LineSpec`.

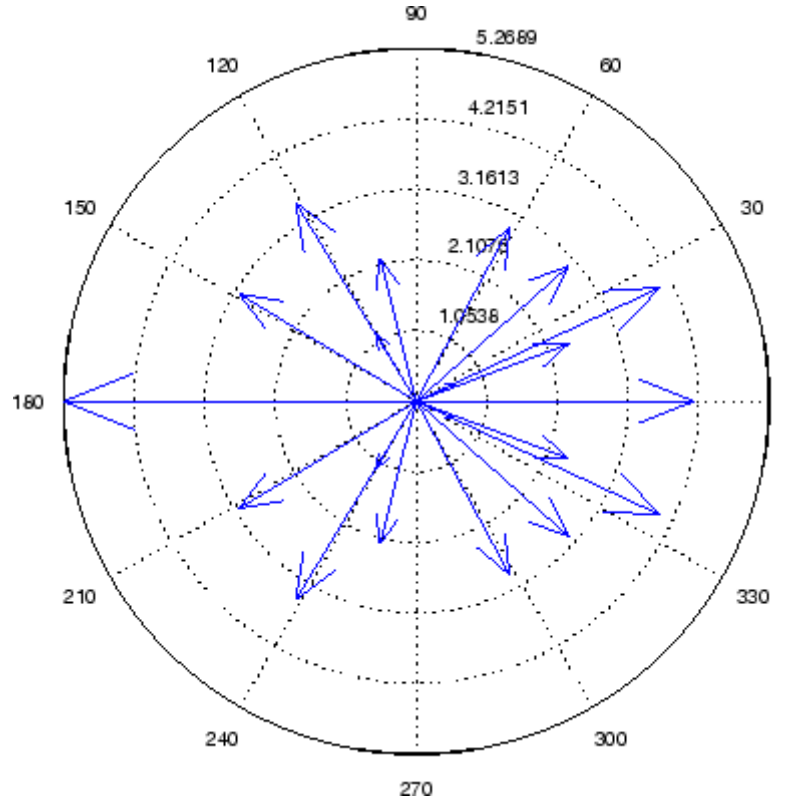
`compass(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = compass(...)` returns handles to line objects.

Examples

Draw a compass graph of the eigenvalues of a matrix.

```
Z = eig(randn(20,20));
compass(Z)
```



See Also

`feather`, `LineSpec`, `quiver`, `rose`

"Direction and Velocity Plots" on page 1-89 for related functions

"Compass Plots" for another example

complex

Purpose

Construct complex data from real and imaginary components

Syntax

```
c = complex(a,b)
```

Description

`c = complex(a,b)` creates a complex output, `c`, from the two real inputs.

```
c = a + bi
```

The output is the same size as the inputs, which must be scalars or equally sized vectors, matrices, or multi-dimensional arrays.

Note If `b` is all zeros, `c` is complex and the value of all its imaginary components is `0`. In contrast, the result of the addition `a+0i` returns a strictly real result.

The following describes when `a` and `b` can have different data types, and the resulting data type of the output `c`:

- If either of `a` or `b` has type `single`, `c` has type `single`.
- If either of `a` or `b` has an integer data type, the other must have the same integer data type or type scalar `double`, and `c` has the same integer data type.

`c = complex(a)` for real `a` returns the complex result `c` with real part `a` and `0` as the value of all imaginary components. Even though the value of all imaginary components is `0`, `c` is complex and `isreal(c)` returns false.

The `complex` function provides a useful substitute for expressions such as

```
a + i*b   or   a + j*b
```

in cases when the names "i" and "j" may be used for other variables (and do not equal $\sqrt{-1}$), when a and b are not single or double, or when b is all zero.

Example

Create complex uint8 vector from two real uint8 vectors.

```
a = uint8([1;2;3;4])
b = uint8([2;2;7;7])
c = complex(a,b)
c =
   1.0000 + 2.0000i
   2.0000 + 2.0000i
   3.0000 + 7.0000i
   4.0000 + 7.0000i
```

See Also

abs, angle, conj, i, imag, isreal, j, real

Purpose

Information about computer on which MATLAB is running

Syntax

```
str = computer
[str,maxsize] = computer
[str,maxsize,endian] = computer
```

Description

`str = computer` returns the string `str` with the computer type on which MATLAB is running.

`[str,maxsize] = computer` returns the integer `maxsize`, which contains the maximum number of elements allowed in an array with this version of MATLAB.

`[str,maxsize,endian] = computer` also returns either 'L' for little endian byte ordering or 'B' for big endian byte ordering.

The list of supported computers changes as new computers are added and others become obsolete. A typical list follows.

32–bit Platforms

str	Computer	ispc	isunix	ismac
GLNX86	GNU Linux on x86	0	1	0
MAC	Apple Macintosh OS X on PPC	0	1	1
MACI	Apple Macintosh OS X on x86	0	1	1
PCWIN	Microsoft Windows on x86	1	0	0

64–bit Platforms

str	Computer	ispc	isunix	ismac
GLNXA64	GNU Linux on x86_64	0	1	0

64-bit Platforms (Continued)

str	Computer	ispc	isunix	ismac
PCWIN64	Microsoft Windows on x64	1	0	0
SOL64	Sun Solaris on SPARC	0	1	0

See Also

getenv, setenv, ispc, isunix, ismac

Purpose

Condition number with respect to inversion

Syntax

```
c = cond(X)
c = cond(X,p)
```

Description

The *condition number* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and the linear equation solution. Values of `cond(X)` and `cond(X,p)` near 1 indicate a well-conditioned matrix.

`c = cond(X)` returns the 2-norm condition number, the ratio of the largest singular value of `X` to the smallest.

`c = cond(X,p)` returns the matrix condition number in `p`-norm:

```
norm(X,p) * norm(inv(X),p
```

If p is...	Then cond(X,p) returns the...
1	1-norm condition number
2	2-norm condition number
'fro'	Frobenius norm condition number
inf	Infinity norm condition number

Algorithm

The algorithm for `cond` (when `p = 2`) uses the singular value decomposition, `svd`.

See Also

`condeig`, `condest`, `norm`, `normest`, `rank`, `rcond`, `svd`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Condition number with respect to eigenvalues

Syntax

```
c = condeig(A)
[V,D,s] = condeig(A)
```

Description

`c = condeig(A)` returns a vector of condition numbers for the eigenvalues of `A`. These condition numbers are the reciprocals of the cosines of the angles between the left and right eigenvectors.

`[V,D,s] = condeig(A)` is equivalent to

```
[V,D] = eig(A);
s = condeig(A);
```

Large condition numbers imply that `A` is near a matrix with multiple eigenvalues.

See Also

`balance`, `cond`, `eig`

condest

Purpose

1-norm condition number estimate

Syntax

```
c = condest(A)
c = condest(A,t)
[c,v] = condest(A)
```

Description

`c = condest(A)` computes a lower bound C for the 1-norm condition number of a square matrix A.

`c = condest(A,t)` changes `t`, a positive integer parameter equal to the number of columns in an underlying iteration matrix. Increasing the number of columns usually gives a better condition estimate but increases the cost. The default is `t = 2`, which almost always gives an estimate correct to within a factor 2.

`[c,v] = condest(A)` also computes a vector `v` which is an approximate null vector if `c` is large. `v` satisfies `norm(A*v,1) = norm(A,1)*norm(v,1)/c`.

Note `condest` invokes `rand`. If repeatable results are required then invoke `rand('state',j)`, for some `j`, before calling this function.

This function is particularly useful for sparse matrices.

Algorithm

`condest` is based on the 1-norm condition estimator of Hager [1] and a block oriented generalization of Hager's estimator given by Higham and Tisseur [2]. The heart of the algorithm involves an iterative search to estimate $\|A^{-1}\|_1$ without computing A^{-1}. This is posed as the convex, but nondifferentiable, optimization problem

$$\max \|A^{-1} x\|_1 \text{ subject to } \|x\|_1 = 1$$

See Also

`cond`, `norm`, `normest`

Reference

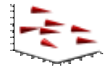
[1] William W. Hager, "Condition Estimates," *SIAM J. Sci. Stat. Comput. 5*, 1984, 311-316, 1984.

[2] Nicholas J. Higham and Françoise Tisseur, "A Block Algorithm for Matrix 1-Norm Estimation with an Application to 1-Norm Pseudospectra, "*SIAM J. Matrix Anal. Appl.*, Vol. 21, 1185-1201, 2000.

coneplot

Purpose

Plot velocity vectors as cones in 3-D vector field



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)
coneplot(U,V,W,Cx,Cy,Cz)
coneplot(...,s)
coneplot(...,color)
coneplot(...,'quiver')
coneplot(...,'method')
coneplot(X,Y,Z,U,V,W,'nointerp')
coneplot(axes_handle,...)
h = coneplot(...)
```

Description

`coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)` plots velocity vectors as cones pointing in the direction of the velocity vector and having a length proportional to the magnitude of the velocity vector.

- `X`, `Y`, `Z` define the coordinates for the vector field.
- `U`, `V`, `W` define the vector field. These arrays must be the same size, monotonic, and 3-D plaid (such as the data produced by `meshgrid`).
- `Cx`, `Cy`, `Cz` define the location of the cones in the vector field. The section “Specifying Starting Points for Stream Plots” in Visualization Techniques provides more information on defining starting points.

`coneplot(U,V,W,Cx,Cy,Cz)` (omitting the `X`, `Y`, and `Z` arguments) assumes `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p]= size(U)`.

`coneplot(...,s)` MATLAB automatically scales the cones to fit the graph and then stretches them by the scale factor `s`. If you do not specify a value for `s`, MATLAB uses a value of 1. Use `s = 0` to plot the cones without automatic scaling.

`coneplot(...,color)` interpolates the array `color` onto the vector field and then colors the cones according to the interpolated values. The size of the `color` array must be the same size as the `U`, `V`, `W` arrays. This option works only with cones (i.e., not with the `quiver` option).

`coneplot(...,'quiver')` draws arrows instead of cones (see `quiver3` for an illustration of a quiver plot).

`coneplot(...,'`*`method`*`')` specifies the interpolation method to use. *`method`* can be `linear`, `cubic`, or `nearest`. `linear` is the default. (See `interp3` for a discussion of these interpolation methods.)

`coneplot(X,Y,Z,U,V,W,'nointerp')` does not interpolate the positions of the cones into the volume. The cones are drawn at positions defined by `X`, `Y`, `Z` and are oriented according to `U`, `V`, `W`. Arrays `X`, `Y`, `Z`, `U`, `V`, `W` must all be the same size.

`coneplot(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = coneplot(...)` returns the handle to the `patch` object used to draw the cones. You can use the `set` command to change the properties of the cones.

Remarks

`coneplot` automatically scales the cones to fit the graph, while keeping them in proportion to the respective velocity vectors.

It is usually best to set the data aspect ratio of the axes before calling `coneplot`. You can set the ratio using the `daspect` command.

```
daspect([1,1,1])
```

Examples

This example plots the velocity vector cones for vector volume data representing the motion of air through a rectangular region of space. The final graph employs a number of enhancements to visualize the data more effectively:

- Cone plots indicate the magnitude and direction of the wind velocity.
- Slice planes placed at the limits of the data range provide a visual context for the cone plots within the volume.
- Directional lighting provides visual cues to the orientation of the cones.
- View adjustments compose the scene to best reveal the information content of the data by selecting the view point, projection type, and magnification.

1. Load and Inspect Data

The `winds` data set contains six 3-D arrays: `u`, `v`, and `w` specify the vector components at each of the coordinates specified in `x`, `y`, and `z`. The coordinates define a lattice grid structure where the data is sampled within the volume.

It is useful to establish the range of the data to place the slice planes and to specify where you want the cone plots (`min`, `max`).

```
load wind
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
ymax = max(y(:));
zmin = min(z(:));
```

2. Create the Cone Plot

- Decide where in data space you want to plot cones. This example selects the full range of `x` and `y` in eight steps and the range 3 to 15 in four steps in `z` (`linspace`, `meshgrid`).

- Use `daspect` to set the data aspect ratio of the axes before calling `coneplot` so MATLAB can determine the proper size of the cones.
- Draw the cones, setting the scale factor to 5 to make the cones larger than the default size.
- Set the coloring of each cone (`FaceColor`, `EdgeColor`).

```
daspect([2,2,1])
xrange = linspace(xmin,xmax,8);
yrange = linspace(ymin,ymax,8);
zrange = 3:4:15;
[cx cy cz] = meshgrid(xrange,yrange,zrange);
hcones = coneplot(x,y,z,u,v,w,cx,cy,cz,5);
set(hcones,'FaceColor','red','EdgeColor','none')
```

3. Add the Slice Planes

- Calculate the magnitude of the vector field (which represents wind speed) to generate scalar data for the `slice` command.
- Create slice planes along the *x*-axis at `xmin` and `xmax`, along the *y*-axis at `ymax`, and along the *z*-axis at `zmin`.
- Specify interpolated face color so the slice coloring indicates wind speed, and do not draw edges (`hold`, `slice`, `FaceColor`, `EdgeColor`).

```
hold on
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hsurfaces = slice(x,y,z,wind_speed,[xmin,xmax],ymax,zmin);
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
hold off
```

4. Define the View

- Use the `axis` command to set the axis limits equal to the range of the data.
- Orient the `view` to azimuth = 30 and elevation = 40. (`rotate3d` is a useful command for selecting the best view.)

- Select perspective projection to provide a more realistic looking volume (`camproj`).
- Zoom in on the scene a little to make the plot as large as possible (`camzoom`).

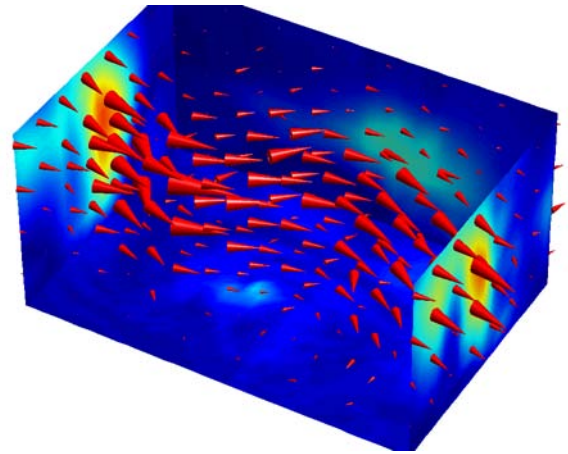
```
axis tight; view(30,40); axis off
camproj perspective; camzoom(1.5)
```

5. Add Lighting to the Scene

The light source affects both the slice planes (surfaces) and the cone plots (patches). However, you can set the lighting characteristics of each independently:

- Add a light source to the right of the camera and use Phong lighting to give the cones and slice planes a smooth, three-dimensional appearance (`camlight`, `lighting`).
- Increase the value of the `AmbientStrength` property for each slice plane to improve the visibility of the dark blue colors. (Note that you can also specify a different `colormap` to change the coloring of the slice planes.)
- Increase the value of the `DiffuseStrength` property of the cones to brighten particularly those cones not showing specular reflections.

```
camlight right; lighting phong
set(hsurfaces,'AmbientStrength',.6)
set(hcones,'DiffuseStrength',.8)
```



See Also

`isosurface`, `patch`, `reducevolume`, `smooth3`, `streamline`, `stream2`, `stream3`, `subvolume`

"Volume Visualization" on page 1-102 for related functions

Purpose

Complex conjugate

Syntax

```
ZC = conj(Z)
```

Description

`ZC = conj(Z)` returns the complex conjugate of the elements of `Z`.

Algorithm

If `Z` is a complex array:

```
conj(Z) = real(Z) - i*imag(Z)
```

See Also

`i`, `j`, `imag`, `real`

Purpose

Pass control to next iteration of `for` or `while` loop

Syntax

```
continue
```

Description

`continue` passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for `continue` statements in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

Examples

The example below shows a `continue` loop that counts the lines of code in the file `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered.

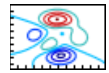
```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line,'%',1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines',count));
```

See Also

`for`, `while`, `end`, `break`, `return`

Purpose

Contour plot of matrix



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation and "Creating Graphics from the Workspace Browser" in the MATLAB Desktop Tools documentation.

Syntax

```
contour(Z)
contour(Z,n)
contour(Z,v)
contour(X,Y,Z)
contour(X,Y,Z,n)
contour(X,Y,Z,v)
contour(...,LineSpec)
contour(ax,...)
[C,h] = contour(...)
[C,h] = contour('v6',...)
```

Description

A contour plot displays isolines of matrix `Z`. Label the contour lines using `clabel`.

`contour(Z)` draws a contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The number of contour levels and the values of the contour levels are chosen automatically based on the minimum and maximum values of `Z`. The ranges of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contour(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels.

`contour(Z,v)` draws a contour plot of matrix `Z` with contour lines at the data values specified in vector `v`. The number of contour levels is equal

to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i])`.

`contour(X,Y,Z)`, `contour(X,Y,Z,n)`, and `contour(X,Y,Z,v)` draw contour plots of `Z`. `X` and `Y` specify the *x*- and *y*-axis limits. When `X` and `Y` are matrices, they must be the same size as `Z`, in which case they specify a surface, as defined by the `surf` function. `X` and `Y` must be monotonically increasing.

If `X` or `Y` is irregularly spaced, `contour` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

`contour(...,LineSpec)` draws the contours using the line type and color specified by `LineSpec`. `contour` ignores marker symbols.

`contour(ax,...)` plots into axes `ax` instead of `gca`.

`[C,h] = contour(...)` returns a contour matrix, `C`, derived from the matrix returned by the low-level `contourc` function, and a handle, `h`, to a contourgroup object. `clabel` uses the contour matrix `C` to create the labels. (See descriptions of `contourgroup properties`.)

Backward Compatible Version

`[C,h] = contour('v6',...)` returns the contour matrix `C` (see `contourc`) and a vector of handles, `h`, to graphics objects. `clabel` uses the contour matrix `C` to create the labels. When called with the `'v6'` flag, `contour` creates patch graphics objects, unless you specify a `LineSpec`, in which case `contour` creates line graphics objects. In this case, contour lines are not mapped to colors in the figure colormap, but are colored using the colors defined in the axes `ColorOrder` property. If you do not specify a `LineSpec` argument, the figure colormap (`colormap`) and the color limits (`caxis`) control the color of the contour lines (patch objects).

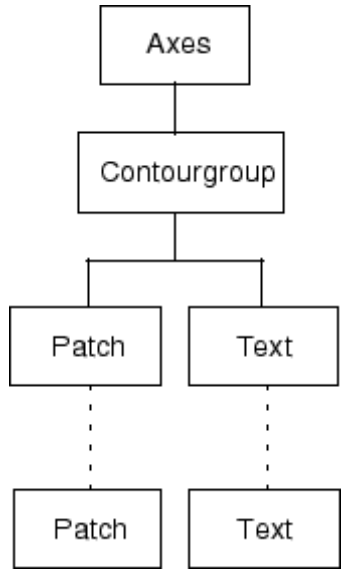
Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

Use contourgroup object properties to control the contour plot appearance.

The following diagram illustrates the parent-child relationship in contour plots.



Examples

Contour Plot of a Function

To view a contour plot of the function

$$z = xe^{(-x^2 - y^2)}$$

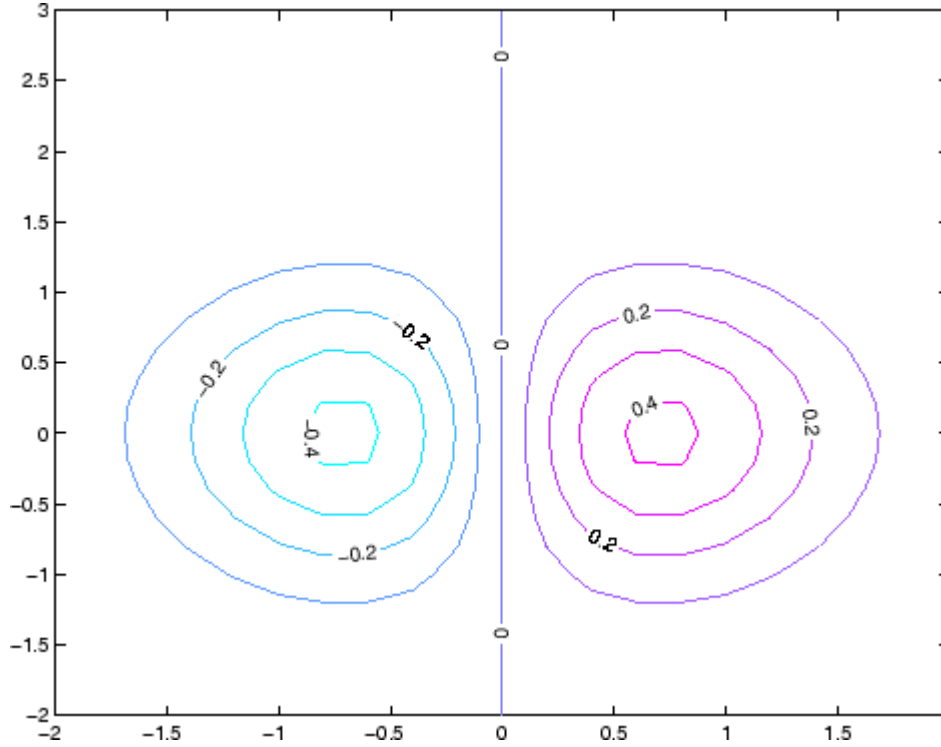
over the range $-2 \le x \le 2$, $-2 \le y \le 3$, create matrix `Z` using the statements

```
[X,Y] = meshgrid(-2:.2:2,-2:.2:3);
Z = X.*exp(-X.^2-Y.^2);
```

Then, generate a contour plot of `Z`.

- Display contour labels by setting the `ShowText` property to `on`.
- Label every other contour line by setting the `TextStep` property to twice the contour interval (i.e., two times the `LevelStep` property).
- Use a smoothly varying colormap.

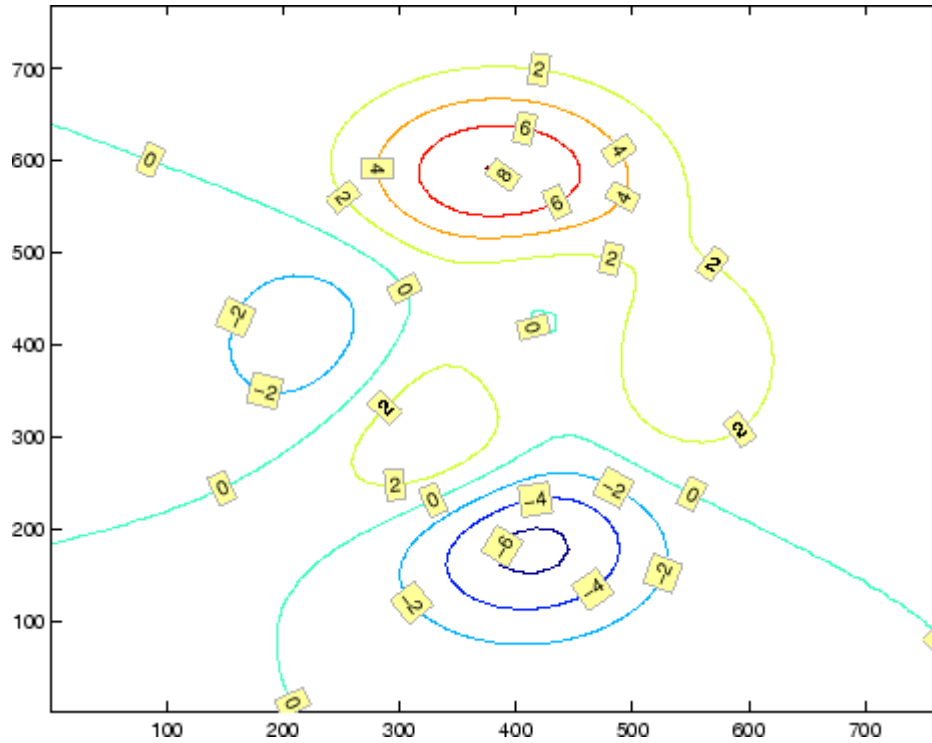
```
[C,h] = contour(X,Y,Z);
set(h,'ShowText','on','TextStep',get(h,'LevelStep')*2)
colormap cool
```



Smoothing Contour Data

Use `interp2` to create smoother contours. Also set the contour label text `BackgroundColor` to a light yellow and the `EdgeColor` to light gray.

```
Z = peaks;
[C,h] = contour(interp2(Z,4));
text_handle = clabel(C,h);
set(text_handle,'BackgroundColor',[1 1 .6],...
    'Edgecolor',[.7 .7 .7])
```



Setting the Axis Limits on Contour Plots

Suppose, for example, your data represents a region that is 1000 meters in the *x* dimension and 3000 meters in the *y* dimension. Use the following statements to set the axis limits correctly:

```
Z = rand(24,36); % assume data is a 24-by-36 matrix
X = linspace(0,1000,size(Z,2));
```

```
Y = linspace(0,3000,size(Z,1));
[c,h] = contour(X,Y,Z);
axis equal tight % set the axes aspect ratio
```

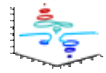
See Also

contour3, contourc, contourf, contourslice

See Contourgroup Properties for property descriptions.

Purpose

3-D contour plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation and "Creating Graphics from the Workspace Browser" in the MATLAB Desktop Tools documentation.

Syntax

```
contour3(Z)
contour3(Z,n)
contour3(Z,v)
contour3(X,Y,Z)
contour3(X,Y,Z,n)
contour3(X,Y,Z,v)
contour3(...,LineSpec)
contour3(axes_handle,...)
[C,h] = contour3(...)
```

Description

`contour3` creates a 3-D contour plot of a surface defined on a rectangular grid.

`contour3(Z)` draws a contour plot of matrix `Z` in a 3-D view. `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The number of contour levels and the values of contour levels are chosen automatically. The ranges of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contour3(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels in a 3-D view.

`contour3(Z,v)` draws a contour plot of matrix `Z` with contour lines at the values specified in vector `v`. The number of contour levels is equal to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i])`.

`contour3(X,Y,Z)`, `contour3(X,Y,Z,n)`, and `contour3(X,Y,Z,v)` use `X` and `Y` to define the *x*- and *y*-axis limits. If `X` is a matrix, `X(1,:)` defines the *x*-axis. If `Y` is a matrix, `Y(:,1)` defines the *y*-axis. When `X` and `Y` are matrices, they must be the same size as `Z`, in which case they specify a surface as `surf` does.

`contour3(...,LineSpec)` draws the contours using the line type and color specified by `LineSpec`.

`contour3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`[C,h] = contour3(...)` returns the contour matrix `C`, as described in the function `contourc` and a column vector `h`, containing handles to graphics objects. `contour3` creates `patch` graphics objects unless you specify `LineSpec`, in which case `contour3` creates `line` graphics objects.

Remarks

If `X` or `Y` is irregularly spaced, `contour3` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

If you do not specify `LineSpec`, `colormap` and `caxis` control the color.

`contour3(...)` works the same as `contour(...)` with these exceptions:

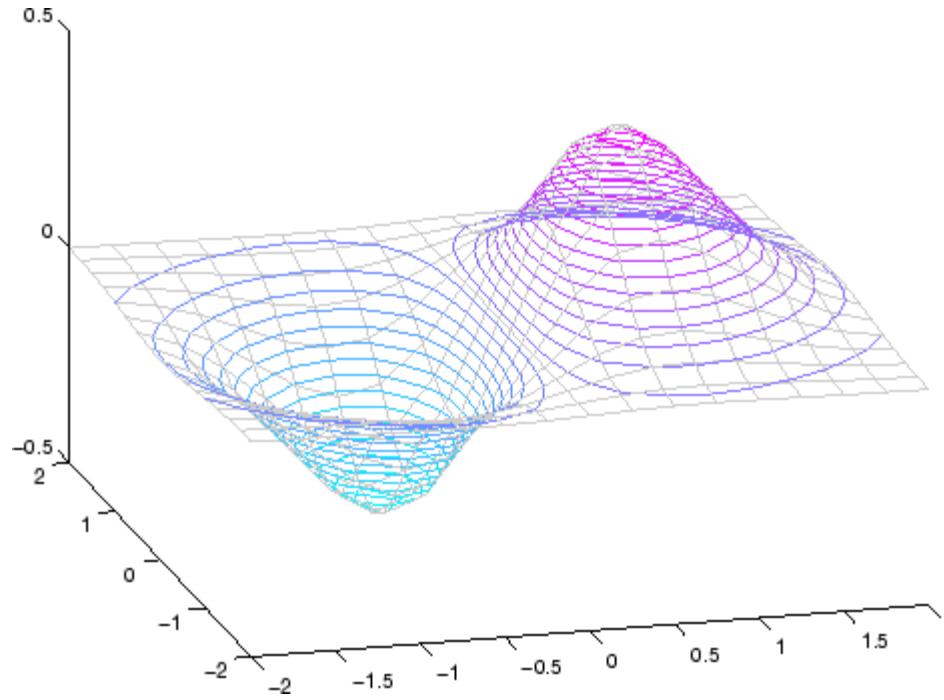
- The contours are drawn at their corresponding Z level.
- Multiple `patch` objects are created instead of a `contourgroup`.
- Calling `contour3` with trailing property-value pairs is not allowed.

Examples

Plot the three-dimensional contour of a function and superimpose a surface plot to enhance visualization of the function.

```
[X,Y] = meshgrid([-2:.25:2]);
Z = X.*exp(-X.^2-Y.^2);
contour3(X,Y,Z,30)
surface(X,Y,Z,'EdgeColor',[.8 .8 .8],'FaceColor','none')
grid off
view(-15,25)
```

```
colormap cool
```



See Also

`contour`, `contourc`, `meshc`, `meshgrid`, `surfc`

"Contour Plots" on page 1-89 category for related functions

"Contour Plots" section for more examples

Purpose

Low-level contour plot computation

Syntax

```
C = contourc(Z)
C = contourc(Z,n)
C = contourc(Z,v)
C = contourc(x,y,Z)
C = contourc(x,y,Z,n)
C = contourc(x,y,Z,v)
```

Description

`contourc` calculates the contour matrix `C` used by `contour`, `contour3`, and `contourf`. The values in `Z` determine the heights of the contour lines with respect to a plane. The contour calculations use a regularly spaced grid determined by the dimensions of `Z`.

`C = contourc(Z)` computes the contour matrix from data in matrix `Z`, where `Z` must be at least a 2-by-2 matrix. The contours are isolines in the units of `Z`. The number of contour lines and the corresponding values of the contour lines are chosen automatically.

`C = contourc(Z,n)` computes contours of matrix `Z` with `n` contour levels.

`C = contourc(Z,v)` computes contours of matrix `Z` with contour lines at the values specified in vector `v`. The length of `v` determines the number of contour levels. To compute a single contour of level `i`, use `contourc(Z,[i i])`.

`C = contourc(x,y,Z)`, `C = contourc(x,y,Z,n)`, and `C = contourc(x,y,Z,v)` compute contours of `Z` using vectors `x` and `y` to determine the *x*- and *y*-axis limits. `x` and `y` must be monotonically increasing.

Remarks

`C` is a two-row matrix specifying all the contour lines. Each contour line defined in matrix `C` begins with a column that contains the value of the contour (specified by `v` and used by `clabel`), and the number of `(x,y)` vertices in the contour line. The remaining columns contain the data for the `(x,y)` pairs.

```
C = [value1xdata(1)xdata(2)..value2xdata(1)xdata(2)...;
```

```
dim1ydata(1)ydata(2)...dim2 ydata(1)ydata(2)...]
```

Specifying irregularly spaced `x` and `y` vectors is not the same as contouring irregularly spaced data. If `x` or `y` is irregularly spaced, `contourc` calculates contours using a regularly spaced contour grid, then transforms the data to `x` or `y`.

See Also

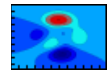
`clabel`, `contour`, `contour3`, `contourf`

"Contour Plots" on page 1-89 for related functions

"The Contouring Algorithm" for more information

Purpose

Filled 2-D contour plot



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation and "Creating Graphics from the Workspace Browser" in the MATLAB Desktop Tools documentation.

Syntax

```
contourf(Z)
contourf(Z,n)
contourf(Z,v)
contourf(X,Y,Z)
contourf(X,Y,Z,n)
contourf(X,Y,Z,v)
contourf(axes_handle,...)
C = contourf(...)
[C,h] = contourf(...)
[C,h,CF] = contourf(...)
```

Description

A filled contour plot displays isolines calculated from matrix `Z` and fills the areas between the isolines using constant colors. The color of the filled areas depends on the current figure's colormap. `NaN`s in the Z-data leave white holes with black borders in the contour plot. The function creates and optionally returns a handle to a `Contourgroup Properties` object containing the filled contours.

`contourf(Z)` draws a contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to a plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The number of contour lines and the values of the contour lines are chosen automatically.

`contourf(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels.

`contourf(Z,v)` draws a contour plot of matrix `Z` with contour levels at the values specified in vector `v`. When you use the `contourf(Z,v)` syntax to specify a vector of contour levels (`v` must increase monotonically), contour regions with `Z`-values less than `v(1)` are not filled (they are rendered in white). To fill such regions with a color, make `v(1)` less than or equal to the minimum `Z`-data value.

`contourf(X,Y,Z)`, `contourf(X,Y,Z,n)`, and `contourf(X,Y,Z,v)` produce contour plots of `Z` using `X` and `Y` to determine the *x*- and *y*-axis limits. When `X` and `Y` are matrices, they must be the same size as `Z`, in which case they specify a surface as `surf` does. `X` and `Y` must be monotonically increasing.

`contourf(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`C = contourf(...)` returns the contour matrix `C` as calculated by the function `contourc` and used by `clabel`.

`[C,h] = contourf(...)` also returns a handle `h` for the contourgroup object.

Backward-Compatible Version

`[C,h,CF] = contourf(...)` returns the contour matrix `C` as calculated by the function `contourc` and used by `clabel`, a vector of handles `h` to patch graphics objects (instead of a contourgroup object, for compatibility with MATLAB 6.5 and earlier) and a contour matrix `CF` for the filled areas.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

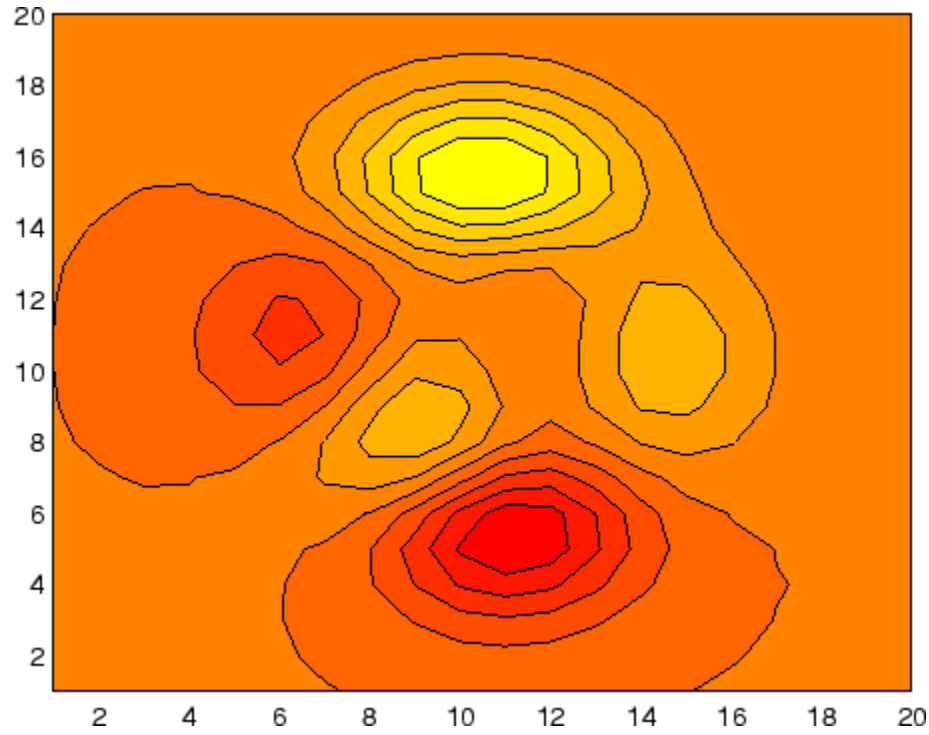
Remarks

If `X` or `Y` is irregularly spaced, `contourf` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

Examples

Create a filled contour plot of the peaks function.

```
[C,h] = contourf(peaks(20),10);
colormap autumn
```



See Also

`clabel`, `contour`, `contour3`, `contourc`, `quiver`

"Contour Plots" on page 1-89 for related functions

Contourgroup Properties

Purpose

Define contourgroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default properties for contourgroup objects.

See “Plot Objects” for more information on contourgroup objects.

Contourgroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of contourgroup objects in legends. The `Annotation` property enables you to specify whether this contourgroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the contourgroup object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the contourgroup object in a legend as one entry, but not its children objects
`off`	Do not include the contourgroup or its children in a legend (default)
`children`	Include only the children of the contourgroup as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See "Controlling Legends" for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See "Function Handle Callbacks" for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`ContourMatrix`
2-by-n matrix Read Only

A two-row matrix specifying all the contour lines. Each contour line defined in the `ContourMatrix` begins with a column that contains the value of the contour (specified by the `LevelList` property and is used by `clabel`), and the number of `(x,y)` vertices in the contour line. The remaining columns contain the data for the `(x,y)` pairs:

```
C = [value1 xdata(1) xdata(2)...value2 xdata(1) xdata(2)...;
       dim1 ydata(1) ydata(2)...  dim2 ydata(1) ydata(2)...]
```

That is,

```
C = [C(1) C(2)...C(I)...C(N)]
```

where `N` is the number of contour levels, and

```
C(i) = [ level(i) x(1) x(2)...x(  numel(i));
```

```
numel(i) y(1) y(2)...y(  numel(i))];
```

For further information, see The Contouring Algorithm.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which can be queried using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the BeingDeleted property for related information.

DisplayName

string (default is empty string)

String used by legend for this contourgroup object. The legend function uses the string defined by the DisplayName property to label this contourgroup object in the legend.

- If you specify string arguments with the legend function, DisplayName is set to this contourgroup object’s corresponding string and that string is used for the legend.
- If DisplayName is empty, legend creates a string of the form, ['data' *n*], where *n* is the number assigned to the object based on its location in the list of legend entries. However, legend does not set DisplayName to this string.
- If you edit the string directly in an existing legend, DisplayName is set to the edited string.
- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.
- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See “Controlling Legends” for more examples.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine

layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`Fill`

`{off} | on`

Color spaces between contour lines. By default, `contour` draws only the contour lines of the surface. If you set `Fill` to `on`, `contour` colors the regions in between the contour lines according to the Z-value of the region and changes the contour lines to black.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.

- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LabelSpacing`
distance in points (default = 144)

Spacing between labels on each contour line. When you display contour line labels using either the `ShowText` property or the `clabel` command, the labels are spaced 144 points (2 inches) apart on each line. You can specify the spacing by setting the `LabelSpacing` property to a value in points. If the length of an individual contour line is less than the specified value, MATLAB displays only one contour label on that line.

`LevelList`
vector of `ZData`-values

Values at which contour lines are drawn. When the `LevelListMode` property is `auto`, the `contour` function automatically chooses contour values that span the range of values in `ZData` (the input argument `Z`). You can set this property to the values at which you want contour lines drawn.

To specify the contour interval (space between contour lines) use the `LevelStep` property.

`LevelListMode`
{auto} | manual

User-specified or autogenerated `LevelList` *values*. By default, the `contour` function automatically generates the values at which contours are drawn. If you set this property to `manual`, contour does not change the values in `LevelList` as you change the values of `ZData`.

`LevelStep`
scalar

Spacing of contour lines. The `contour` function draws contour lines at regular intervals determined by the value of `LevelStep`. When the `LevelStepMode` property is set to `auto`, `contour` determines the contour interval automatically based on the `ZData`.

`LevelStepMode`
{auto} | manual

User-specified or autogenerated `LevelStep` *values*. By default, the `contour` function automatically determines a value for the `LevelStep` property. If you set this property to `manual`, contour does not change the value of `LevelStep` as you change the values of `ZData`.

`LineColor`
`{auto} | ColorSpec | none`

Color of the contour lines. This property determines how MATLAB colors the contour lines.

- `auto`— Each contour line is a single color determined by its contour value, the figure colormap, and the color axis (`caxis`).
- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default edge color is black. See `ColorSpec` for more information on specifying color.
- `none` — No contour lines are drawn.

`LineStyle`

`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Parent`

handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`ShowText`
`on | {off}`

Display labels on contour lines. When you set this property to on, MATLAB displays text labels on each contour line indicating the contour value. See also `LevelList`, `clabel`, and the example “Contour Plot of a Function” on page 2-638.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`TextList`

vector of contour values

Contour values to label. This property contains the contour values where text labels are placed. By default, these values are the same as those contained in the `LevelList` property, which define where the contour lines are drawn. Note that there must be an equivalent contour line to display a text label.

For example, the following statements create and label a contour plot:

```
[c,h]=contour(peaks);
clabel(c,h)
```

You can get the `LevelList` property to see the contour line values:

```
get(h,'LevelList')
```

Suppose you want to view the contour value `4.375` instead of the value of `4` that the contour function used. To do this, you need to set both the `LevelList` and `TextList` properties:

```
set(h,'LevelList',[-6 -4 -2 0 2 4.375 6 8],...
 'TextList',[-6 -4 -2 0 2 4.375 6 8])
```

See the example "Contour Plot of a Function" on page 2-638 for additional information.

`TextListMode`
`{auto} | manual`

User-specified or auto `TextList` values. When this property is set to `auto`, MATLAB sets the `TextList` property equal to the values of the `LevelList` property (i.e., a text label for each contour line). When this property is set to `manual`, MATLAB does not set the values of the `TextList` property. Note that specifying values for the `TextList` property causes the `TextListMode` property to be set to `manual`.

`TextStep`
scalar

Determines which contour line have numeric labels. The `contour` function labels contour lines at regular intervals which are determined by the value of the `TextStep` property. When the `TextStepMode` property is set to `auto`, `contour` labels every contour line when the `ShowText` property is `on`. See "Contour Plot of a Function" on page 2-638 for an example that uses the `TextStep` property.

`TextStepMode`
{auto} | manual

User-specified or autogenerated `TextStep` values. By default, the `contour` function automatically determines a value for the `TextStep` property. If you set this property to `manual`, `contour` does not change the value of `TextStep` as you change the values of `ZData`.

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For contourgroup objects, `Type` is '`hggroup`'. This statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
vector or matrix

The x-axis values for a graph. The *x*-axis values for graphs are specified by the `X` input argument. If `XData` is a vector, `length(XData)` must equal `length(YData)` and must be monotonic. If `XData` is a matrix, `size(XData)` must equal `size(YData)` and each column must be monotonic.

You can use XData to define meaningful coordinates for an underlying surface whose topography is being mapped. See "Setting the Axis Limits on Contour Plots" on page 2-640 for more information.

XDataMode
{auto} | manual

Use automatic or user-specified x-axis values. If you specify XData (by setting the XData property or specifying the x input argument), MATLAB sets this property to manual and uses the specified values to label the *x*-axis.

If you set XDataMode to auto after having specified XData, MATLAB resets the *x*-axis ticks to 1:size(YData,1) or to the column indices of the ZData, overwriting any previous values for XData.

XDataSource
string (MATLAB variable)

Link XData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change XData.

You can use the refreshdata function to force an update of the object's data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

scalar, vector, or matrix

Y-axis limits. This property determines the *y*-axis limits used in the contour plot. If you do not specify a `Y` argument, the `contour` function calculates *y*-axis limits based on the size of the input argument `Z`.

`YData` can be either a matrix equal in size to `ZData` or a vector equal in length to the number of columns in `ZData`.

Use `YData` to define meaningful coordinates for the underlying surface whose topography is being mapped. See "Setting the Axis Limits on Contour Plots" on page 2-640 for more information.

`YDataMode`

{auto} | manual

Use automatic or user-specified y-axis values. In `auto` mode (the default) the `contour` function automatically determines the *y*-axis limits. If you set this property to `manual`, specify a value for `YData`, or specify a `Y` argument, then `contour` sets this property to `manual` and does not change the axis limits.

`YDataSource`

string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`

matrix

Contour data. This property contains the data from which the contour lines are generated (specified as the input argument `Z`). `ZData` must be at least a 2-by-2 matrix. The number of contour levels and the values of the contour levels are chosen automatically based on the minimum and maximum values of `ZData`. The limits of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(ZData)`.

`ZDataSource`

string (MATLAB variable)

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `ZData`.

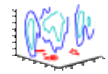
You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Draw contours in volume slice planes



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
contourslice(X,Y,Z,V,Sx,Sy,Sz)
contourslice(X,Y,Z,V,Xi,Yi,Zi)
contourslice(V,Sx,Sy,Sz)
contourslice(V,Xi,Yi,Zi)
contourslice(...,n)
contourslice(...,cvals)
contourslice(...,[cv cv])
contourslice(...,'method')
contourslice(axes_handle,...)
h = contourslice(...)
```

Description

`contourslice(X,Y,Z,V,Sx,Sy,Sz)` draws contours in the *x*-, *y*-, and *z*-axis aligned planes at the points in the vectors `Sx`, `Sy`, `Sz`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V` and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). The color at each contour is determined by the volume `V`, which must be an m-by-n-by-p volume array.

`contourslice(X,Y,Z,V,Xi,Yi,Zi)` draws contours through the volume `V` along the surface defined by the 2-D arrays `Xi,Yi,Zi`. The surface should lie within the bounds of the volume.

`contourslice(V,Sx,Sy,Sz)` and `contourslice(V,Xi,Yi,Zi)` (omitting the `X`, `Y`, and `Z` arguments) assume `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p]= size(v)`.

`contourslice(...,n)` draws `n` contour lines per plane, overriding the automatic value.

`contourslice(...,cvals)` draws `length(cval)` contour lines per plane at the values specified in vector `cvals`.

`contourslice(...,[cv cv])` computes a single contour per plane at the level `cv`.

`contourslice(...,'method')` specifies the interpolation method to use. *`method`* can be `linear`, `cubic`, or `nearest`. `nearest` is the default except when the contours are being drawn along the surface defined by `Xi`, `Yi`, `Zi`, in which case `linear` is the default. (See `interp3` for a discussion of these interpolation methods.)

`contourslice(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = contourslice(...)` returns a vector of handles to `patch` objects that are used to implement the contour lines.

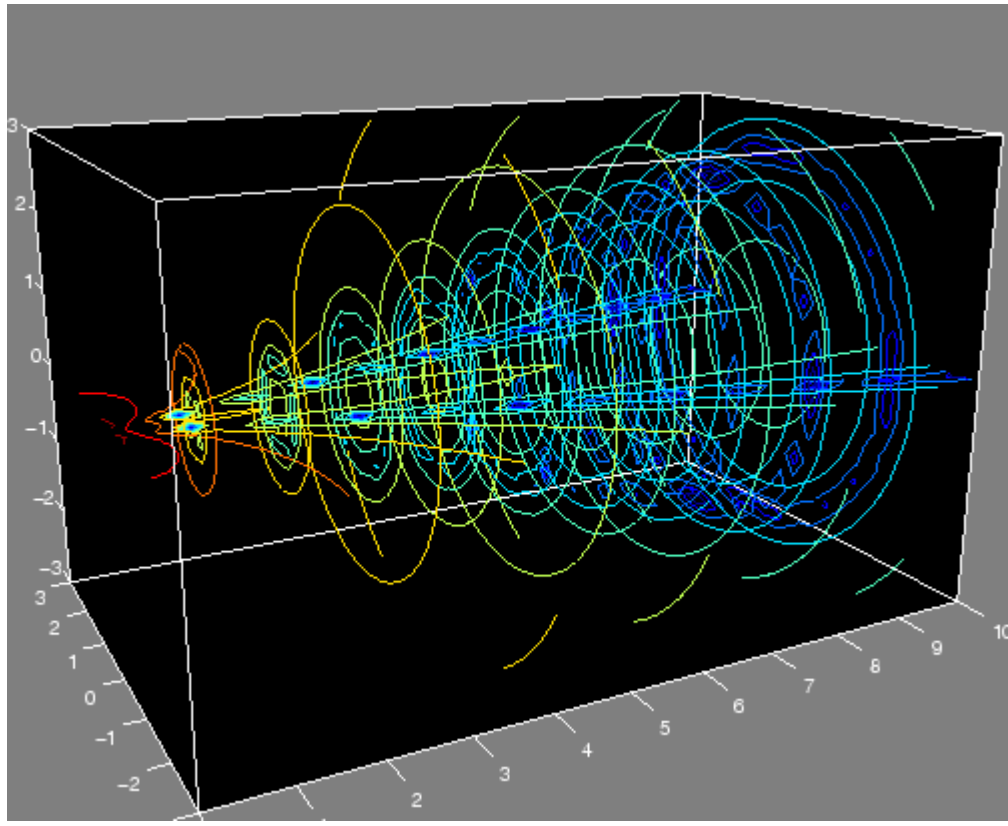
Examples

This example uses the `flow` data set to illustrate the use of contoured slice planes. (Type `doc flow` for more information on this data set.) Notice that this example

- Specifies a vector of `length = 9` for `Sx`, an empty vector for the `Sy`, and a scalar value (0) for `Sz`. This creates nine contour plots along the x direction in the y-z plane, and one in the x-y plane at z = 0.
- Uses `linspace` to define a 10-element vector of linearly spaced values from -8 to 2. This vector specifies that 10 contour lines be drawn, one at each element of the vector.
- Defines the view and projection type (`camva`, `camproj`, `campos`).
- Sets figure (`gcf`) and axes (`gca`) characteristics.

```
[x y z v] = flow;
h = contourslice(x,y,z,v,[1:9],[],[0],linspace(-8,2,10));
axis([0,10,-3,3,-3,3]); daspect([1,1,1])
camva(24); camproj perspective;
```

```
campos([-3,-15,5])
set(gcf,'Color',[.5,.5,.5],'Renderer','zbuffer')
set(gca,'Color','black','XColor','white', ...
 'YColor','white','ZColor','white')
box on
```



This example draws contour slices along a spherical surface within the volume.

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2); % Create volume data
```

```
[xi,yi,zi] = sphere; % Plane to contour
contourslice(x,y,z,v,xi,yi,zi)
view(3)
```

See Also

isosurface, slice, smooth3, subvolume, reducevolume

"Volume Visualization" on page 1-102 for related functions

Purpose

Grayscale colormap for contrast enhancement

Syntax

```
cmap = contrast(X)
cmap = contrast(X,m)
```

Description

The `contrast` function enhances the contrast of an `image`. It creates a new gray colormap, `cmap`, that has an approximately equal intensity distribution. All three elements in each row are identical.

`cmap = contrast(X)` returns a gray colormap that is the same length as the current colormap.

`cmap = contrast(X,m)` returns an `m`-by-3 gray colormap.

Examples

Add contrast to the clown image defined by `X`.

```
load clown;
cmap = contrast(X);
image(X);
colormap(cmap);
```

See Also

`brighten`, `colormap`, `image`

"Colormaps" on page 1-99 for related functions

Purpose

Convolution and polynomial multiplication

Syntax

```
w = conv(u,v)
```

Description

`w = conv(u,v)` convolves vectors `u` and `v`. Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of `u` and `v`.

Definition

Let `m = length(u)` and `n = length(v)` . Then `w` is the vector of length `m+n-1` whose `k`th element is

$$w(k) = \sum_j u(j)v(k+1-j)$$

The sum is over all the values of `j` which lead to legal subscripts for `u(j)` and `v(k+1-j)`, specifically `j = max(1,k+1-n): min(k,m)`. When `m = n`, this gives

```
w(1) = u(1)*v(1)
w(2) = u(1)*v(2)+u(2)*v(1)
w(3) = u(1)*v(3)+u(2)*v(2)+u(3)*v(1)
...
w(n) = u(1)*v(n)+u(2)*v(n-1)+ ... +u(n)*v(1)
...
w(2*n-1) = u(n)*v(n)
```

Algorithm

The convolution theorem says, roughly, that convolving two sequences is the same as multiplying their Fourier transforms. In order to make this precise, it is necessary to pad the two vectors with zeros and ignore roundoff error. Thus, if

```
X = fft([x zeros(1,length(y)-1)])
```

and

```
Y = fft([y zeros(1,length(x)-1)])
```

then `conv(x,y) = ifft(X.*Y)`

See Also

`conv2`, `convn`, `deconv`, `filter`

`convmtx` and `xcorr` in the Signal Processing Toolbox

Purpose

2-D convolution

Syntax

```
C = conv2(A,B)
C = conv2(hcol,hrow,A)
C = conv2(...,'shape')
```

Description

`C = conv2(A,B)` computes the two-dimensional convolution of matrices `A` and `B`. If one of these matrices describes a two-dimensional finite impulse response (FIR) filter, the other matrix is filtered in two dimensions.

The size of `C` in each dimension is equal to the sum of the corresponding dimensions of the input matrices, minus one. That is, if the size of `A` is `[ma,na]` and the size of `B` is `[mb,nb]`, then the size of `C` is `[ma+mb-1,na+nb-1]`.

The indices of the center element of B are defined as `floor(([mb nb]+1)/2)`.

`C = conv2(hcol,hrow,A)` convolves `A` first with the vector `hcol` along the rows and then with the vector `hrow` along the columns. If `hcol` is a column vector and `hrow` is a row vector, this case is the same as `C = conv2(hcol*hrow,A)`.

`C = conv2(...,'shape')` returns a subsection of the two-dimensional convolution, as specified by the `shape` parameter:

`full`	Returns the full two-dimensional convolution (default).
`same`	Returns the central part of the convolution of the same size as `A`.
`valid`	Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, `C` has size `[ma-mb+1,na-nb+1]` when `all(size(A) >= size(B))`. Otherwise `conv2` returns `[]`.

Note If any of `A`, `B`, `hcol`, and `hrow` are empty, then `C` is an empty matrix `[]`.

Algorithm

`conv2` uses a straightforward formal implementation of the two-dimensional convolution equation in spatial form. If a and b are functions of two discrete variables, n_1 and n_2, then the formula for the two-dimensional convolution of a and b is

$$c(n_1, n_2) = \sum_{k_1 = -\infty}^{\infty} \sum_{k_2 = -\infty}^{\infty} a(k_1, k_2)\, b(n_1 - k_1, n_2 - k_2)$$

In practice however, `conv2` computes the convolution for finite intervals.

Note that matrix indices in MATLAB always start at 1 rather than 0. Therefore, matrix elements `A(1,1)`, `B(1,1)`, and `C(1,1)` correspond to mathematical quantities a (0,0), b (0,0), and c (0,0).

Examples

Example 1

For the `'same'` case, `conv2` returns the central part of the convolution. If there are an odd number of rows or columns, the "center" leaves one more at the beginning than the end.

This example first computes the convolution of A using the default (`'full'`) shape, then computes the convolution using the `'same'` shape. Note that the array returned using `'same'` corresponds to the underlined elements of the array returned using the default shape.

```
A = rand(3);
B = rand(4);
C = conv2(A,B)           % C is 6-by-6

C =
   0.1838   0.2374   0.9727   1.2644   0.7890   0.3750
   0.6929   1.2019   1.5499   2.1733   1.3325   0.3096
   0.5627   1.5150   2.3576   3.1553   2.5373   1.0602
```

```
    0.9986   2.3811   3.4302   3.5128   2.4489   0.8462
    0.3089   1.1419   1.8229   2.1561   1.6364   0.6841
    0.3287   0.9347   1.6464   1.7928   1.2422   0.5423

Cs = conv2(A,B,'same')   % Cs is the same size as A: 3-by-3
Cs =
    2.3576   3.1553   2.5373
    3.4302   3.5128   2.4489
    1.8229   2.1561   1.6364
```

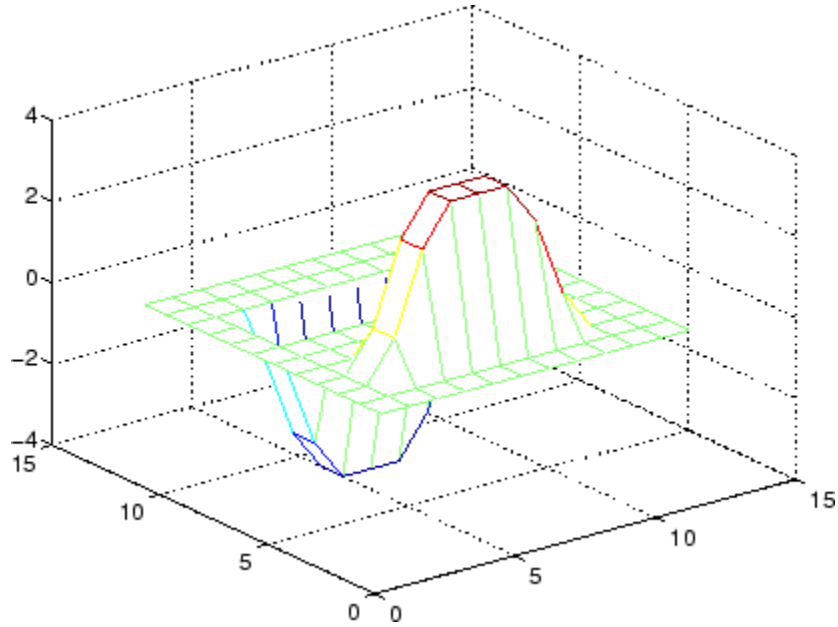
Example 2

In image processing, the Sobel edge finding operation is a two-dimensional convolution of an input array with the special matrix

```
s = [1 2 1; 0 0 0; -1 -2 -1];
```

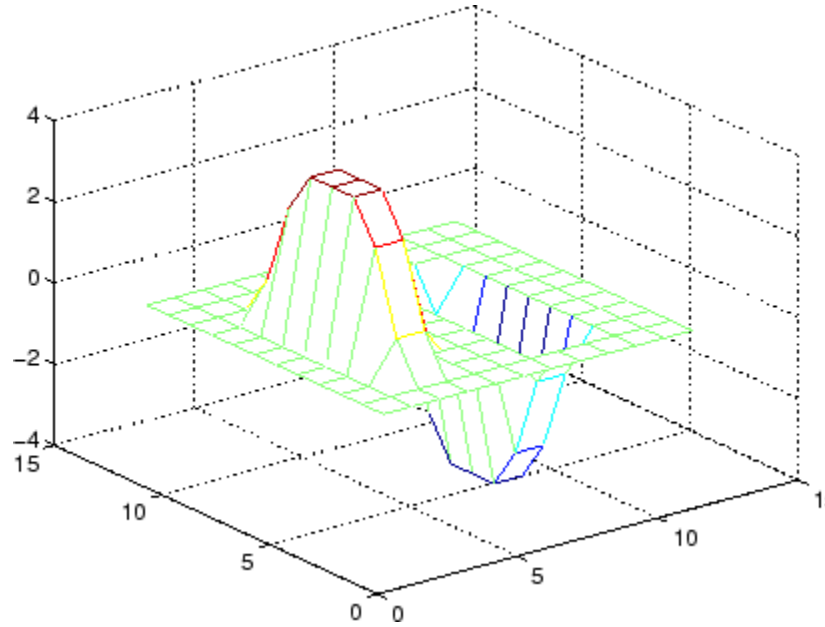
These commands extract the horizontal edges from a raised pedestal.

```
A = zeros(10);
A(3:7,3:7) = ones(5);
H = conv2(A,s);
mesh(H)
```



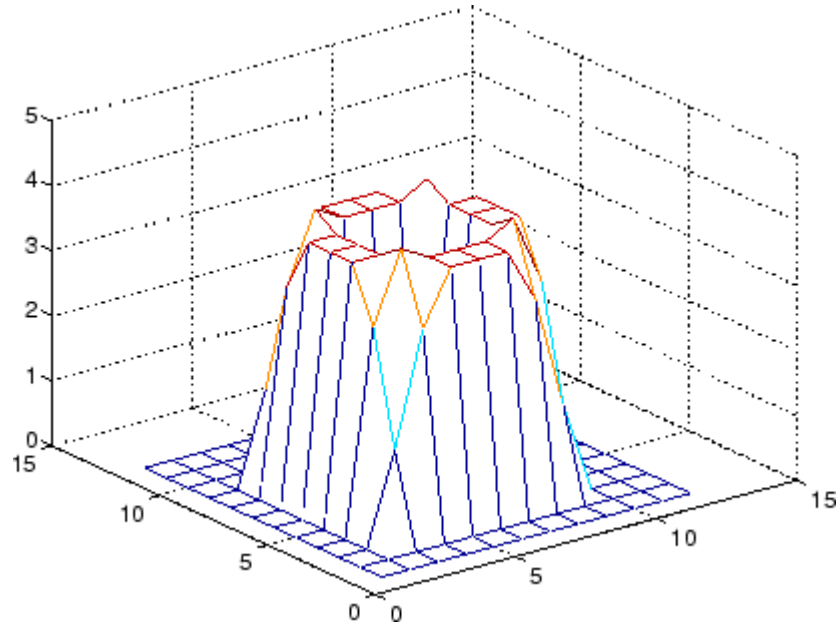
Transposing the filter `s` extracts the vertical edges of `A`.

```
V = conv2(A,s');
figure, mesh(V)
```



This figure combines both horizontal and vertical edges.

```
figure
mesh(sqrt(H.^2 + V.^2))
```



See Also

`conv`, `convn`, `filter2`

`xcorr2` in the Signal Processing Toolbox

convhull

Purpose

Convex hull

Syntax

```
K = convhull(x,y)
K = convhull(x,y,options)
[K,a] = convhull(...)
```

Description

`K = convhull(x,y)` returns indices into the `x` and `y` vectors of the points on the convex hull.

`convhull` uses Qhull.

`K = convhull(x,y,options)` specifies a cell array of strings `options` to be used in Qhull via `convhulln`. The default option is `{'Qt'}`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options will be used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org`.

`[K,a] = convhull(...)` also returns the area of the convex hull.

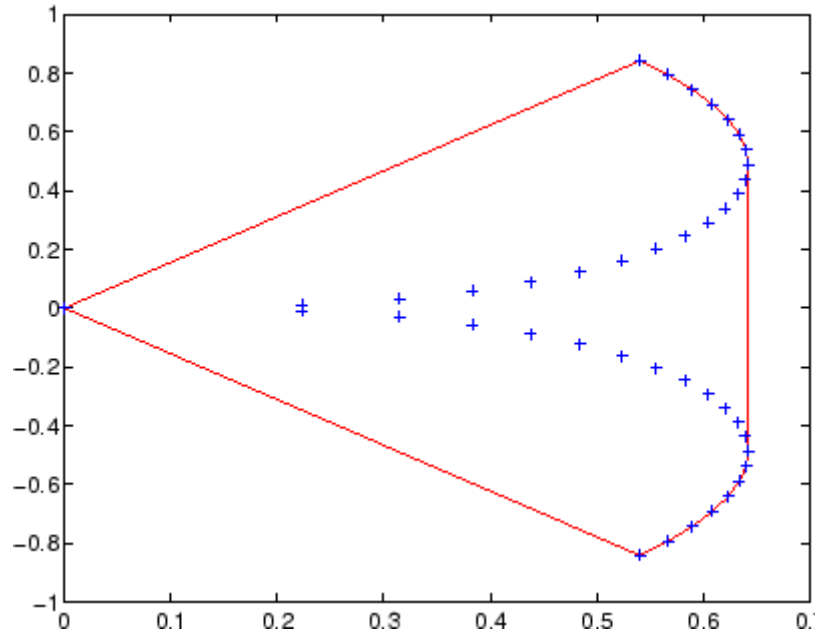
Visualization

Use `plot` to plot the output of `convhull`.

Examples

Example 1

```
xx = -1:.05:1; yy = abs(sqrt(xx));
[x,y] = pol2cart(xx,yy);
k = convhull(x,y);
plot(x(k),y(k),'r-',x,y,'b+')
```



Example 2

The following example illustrates the `options` input for `convhull`. The following commands

```
X = [0 0 0 1];
Y = [0 1e-10 0 1];
K = convhull(X,Y)
```

return a warning.

```
Warning: qhull precision warning:
The initial hull is narrow (cosine of min. angle is
0.9999999999999998).
A coplanar point may lead to a wide facet. Options 'QbB' (scale
to unit box)
or 'Qbb' (scale last coordinate) may remove this warning. Use 'Pp'
to skip
```

```
this warning.
```

To suppress this warning, use the option `'Pp'`. The following command passes the option `'Pp'`, along with the default `'Qt'`, to `convhull`.

```
K = convhull(X,Y,{'Qt','Pp'})

K =

     2
     1
     4
     2
```

Algorithm

`convhull` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`convhulln`, `delaunay`, `plot`, `polyarea`, `voronoi`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber`.

[2] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), *University of Minnesota*, 1993.

Purpose

N-D convex hull

Syntax

```
K = convhulln(X)
K = convulln(X, options)
[K, v] = convhulln(...)
```

Description

`K = convhulln(X)` returns the indices `K` of the points in `X` that comprise the facets of the convex hull of `X`. `X` is an `m`-by-`n` array representing `m` points in N-dimensional space. If the convex hull has `p` facets then `K` is `p`-by-`n`.

`convhulln` uses Qhull.

`K = convulln(X, options)` specifies a cell array of strings `options` to be used as options in Qhull. The default options are:

- `{'Qt'}` for 2-, 3-. and 4-dimensional input
- `{'Qt','Qx'}` for 5-dimensional input and higher.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org/`.

`[K, v] = convhulln(...)` also returns the volume `v` of the convex hull.

Visualization

Plotting the output of `convhulln` depends on the value of `n`:

- For `n = 2`, use `plot` as you would for `convhull`.
- For `n = 3`, you can use `trisurf` to plot the output. The calling sequence is

  ```
  K = convhulln(X);
  trisurf(K,X(:,1),X(:,2),X(:,3))
  ```

  For more control over the color of the facets, use `patch` to plot the output. For an example, see “Tessellation and Interpolation

of Scattered Data in Higher Dimensions" in the MATLAB documentation.

- You cannot plot `convhulln` output for `n > 3`.

Example

The following example illustrates the `options` input for `convhulln`. The following commands

```
X = [0 0; 0 1e-10; 0 0; 1 1];
K = convhulln(X)
```

return a warning.

```
Warning: qhull precision warning:
The initial hull is narrow
(cosine of min. angle is 0.9999999999999998).
A coplanar point may lead to a wide facet.
Options 'QbB' (scale to unit box) or 'Qbb'
(scale last coordinate) may remove this warning.
Use 'Pp' to skip this warning.
```

To suppress the warning, use the option `'Pp'`. The following command passes the option `'Pp'`, along with the default `'Qt'`, to `convhulln`.

```
K = convhulln(X,{'Qt','Pp'})

K =

     1     4
     1     2
     4     2
```

Algorithm

`convhulln` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`convhull`, `delaunayn`, `dsearchn`, `tsearchn`, `voronoin`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM *Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483.

Purpose

N-D convolution

Syntax

```
C = convn(A,B)
C = convn(A,B,'shape')
```

Description

`C = convn(A,B)` computes the N-dimensional convolution of the arrays `A` and `B`. The size of the result is `size(A)+size(B)-1`.

`C = convn(A,B,'shape')` returns a subsection of the N-dimensional convolution, as specified by the `shape` parameter:

`'full'`	Returns the full N-dimensional convolution (default).
`'same'`	Returns the central part of the result that is the same size as `A`.
`'valid'`	Returns only those parts of the convolution that can be computed without assuming that the array A is zero-padded. The size of the result is `max(size(A)-size(B) + 1, 0)`

See Also

`conv`, `conv2`

Purpose

Copy file or directory

Graphical Interface

In the Current Directory browser, select **Edit > Copy**, then **Paste**. See details.

Syntax

```
copyfile('source','destination')
copyfile('source','destination','f')
[status,message,messageid] = copyfile('source','destination',
   'f')
```

Description

`copyfile('source','destination')` copies the file or directory, `source` (and all its contents) to the file or directory, `destination`, where `source` and `destination` are the absolute or relative pathnames for the directory or file. If `source` is a directory, `destination` cannot be a file. If `source` is a directory, `copyfile` copies the contents of `source`, not the directory itself. To rename a file or directory when copying it, make `destination` a different name than `source`. If `destination` already exists, `copyfile` replaces it without warning. Use the wildcard `*` at the end of `source` to copy all matching files. Note that the read-only and archive attributes of `source` are not preserved in `destination`.

`copyfile('source','destination','f')` copies `source` to `destination`, regardless of the read-only attribute of `destination`.

`[status,message,messageid] = copyfile('source','destination','f')` copies `source` to `destination`, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterror`). Here, `status` is `1` for success and `0` for error. Only one output argument is required and the `f` input argument is optional.

Remarks

The `*` wildcard in a path string is supported. Current behavior of `copyfile` differs between UNIX and Windows when using the wildcard `*` or copying directories.

The timestamp given to the `destination` file is identical to that taken from the `source` file.

Examples

Copy File in Current Directory, Assigning a New Name to It

To make a copy of a file `myfun.m` in the current directory, assigning it the name `myfun2.m`, type

```
copyfile('myfun.m','myfun2.m')
```

Copy File to Another Directory

To copy `myfun.m` to the directory `d:/work/myfiles`, keeping the same filename, type

```
copyfile('myfun.m','d:/work/myfiles')
```

Copy All Matching Files by Using a Wildcard

To copy all files in the directory `myfiles` whose names begin with my to the directory `newprojects`, where `newprojects` is at the same level as the current directory, type

```
copyfile('myfiles/my*','../newprojects')
```

Copy Directory and Return Status

In this example, all files and subdirectories in the current directory's `myfiles` directory are copied to the `directory d:/work/myfiles`. Note that before running the `copyfile` function, `d:/work` does not contain the directory `myfiles`. It is created because `myfiles` is appended to `destination` in the `copyfile` function:

```
[s,mess,messid]=copyfile('myfiles','d:/work/myfiles')
s =
     1

mess =
     ''

messid =
     ''
```

The message returned indicates that `copyfile` was successful.

Copy File to Read-Only Directory

Copy `myfile.m` from the current directory to `d:/work/restricted`, where `restricted` is a read-only directory:

```
copyfile('myfile.m','d:/work/restricted','f')
```

After the copy, `myfile.m` exists in `d:/work/restricted`.

See Also

`cd`, `delete`, `dir`, `fileattrib`, `filebrowser`, `fileparts`, `mkdir`, `movefile`, `rmdir`

copyobj

Purpose

Copy graphics objects and their descendants

Syntax

```
new_handle = copyobj(h,p)
```

Description

`copyobj` creates copies of graphics objects. The copies are identical to the original objects except the copies have different values for their `Parent` property and a new handle. The new parent must be appropriate for the copied object (e.g., you can copy a line object only to another axes object).

`new_handle = copyobj(h,p)` copies one or more graphics objects identified by `h` and returns the handle of the new object or a vector of handles to new objects. The new graphics objects are children of the graphics objects specified by `p`.

Remarks

`h` and `p` can be scalars or vectors. When both are vectors, they must be the same length, and the output argument, `new_handle`, is a vector of the same length. In this case, `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p(i)`.

When `h` is a scalar and `p` is a vector, `h` is copied once to each of the parents in `p`. Each `new_handle(i)` is a copy of `h` with its `Parent` property set to `p(i)`, and `length(new_handle)` equals `length(p)`.

When `h` is a vector and `p` is a scalar, each `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p`. The length of `new_handle` equals `length(h)`.

Graphics objects are arranged as a hierarchy. See “Handle Graphics Objects” for more information.

Examples

Copy a surface to a new axes within a different figure.

```
h = surf(peaks);
colormap hot
figure    % Create a new figure
axes    % Create an axes object in the figure
new_handle = copyobj(h,gca);
```

```
colormap hot
view(3)
grid on
```

Note that while the surface is copied, the `colormap` (`figure` property), `view`, and `grid` (`axes` properties) are not copies.

See Also

`findobj`, `gcf`, `gca`, `gco`, `get`, `set`

`Parent` property for all graphics objects

"Finding and Identifying Graphics Objects" on page 1-93 for related functions

Purpose

Correlation coefficients

Syntax

```
R = corrcoef(X)
R = corrcoef(x,y)
[R,P]=corrcoef(...)
[R,P,RLO,RUP]=corrcoef(...)
[...]=corrcoef(...,'param1',val1,'param2',val2,...)
```

Description

`R = corrcoef(X)` returns a matrix `R` of correlation coefficients calculated from an input matrix `X` whose rows are observations and whose columns are variables. The matrix `R = corrcoef(X)` is related to the covariance matrix `C = cov(X)` by

$$R(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}}$$

`corrcoef(X)` is the zeroth lag of the normalized covariance function, that is, the zeroth lag of `xcov(x,'coeff')` packed into a square array.

`R = corrcoef(x,y)` where `x` and `y` are column vectors is the same as `corrcoef([x y])`. If `x` and `y` are not column vectors, `corrcoef` converts them to column vectors. For example, in this case `R=corrcoef(x,y)` is equivalent to `R=corrcoef([x(:) y(:)])`.

`[R,P]=corrcoef(...)` also returns `P`, a matrix of p-values for testing the hypothesis of no correlation. Each p-value is the probability of getting a correlation as large as the observed value by random chance, when the true correlation is zero. If `P(i,j)` is small, say less than `0.05`, then the correlation `R(i,j)` is significant.

`[R,P,RLO,RUP]=corrcoef(...)` also returns matrices `RLO` and `RUP`, of the same size as `R`, containing lower and upper bounds for a 95% confidence interval for each coefficient.

`[...]=corrcoef(...,'param1',val1,'param2',val2,...)` specifies additional parameters and their values. Valid parameters are the following.

'alpha'	A number between 0 and 1 to specify a confidence level of 100*(1 - alpha)%. Default is 0.05 for 95% confidence intervals.
'rows'	Either 'all' (default) to use all rows, 'complete' to use rows with no NaN values, or 'pairwise' to compute R(i,j) using rows with no NaN values in either column i or j.

The p-value is computed by transforming the correlation to create a t statistic having n-2 degrees of freedom, where n is the number of rows of X. The confidence bounds are based on an asymptotic normal distribution of 0.5*log((1+R)/(1-R)), with an approximate variance equal to 1/(n-3). These bounds are accurate for large samples when X has a multivariate normal distribution. The 'pairwise' option can produce an R matrix that is not positive definite.

Examples

Generate random data having correlation between column 4 and the other columns.

```
x = randn(30,4);     % Uncorrelated data
x(:,4) = sum(x,2);   % Introduce correlation.
[r,p] = corrcoef(x)  % Compute sample correlation and p-values.
[i,j] = find(p<0.05);  % Find significant correlations.
[i,j]                % Display their (row,col) indices.

r =
    1.0000   -0.3566    0.1929    0.3457
   -0.3566    1.0000   -0.1429    0.4461
    0.1929   -0.1429    1.0000    0.5183
    0.3457    0.4461    0.5183    1.0000

p =
    1.0000    0.0531    0.3072    0.0613
    0.0531    1.0000    0.4511    0.0135
    0.3072    0.4511    1.0000    0.0033
    0.0613    0.0135    0.0033    1.0000
```

```
ans =
     4     2
     4     3
     2     4
     3     4
```

See Also

`cov`, `mean`, `median`, `std`, `var`

`xcorr`, `xcov` in the Signal Processing Toolbox

Purpose

Cosine of argument in radians

Syntax

```
Y = cos(X)
```

Description

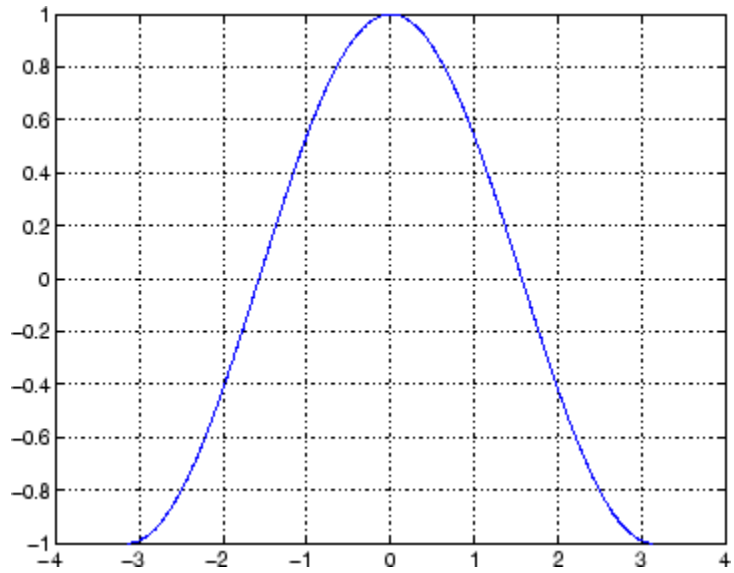
The `cos` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = cos(X)` returns the circular cosine for each element of `X`.

Examples

Graph the cosine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,cos(x)), grid on
```



The expression `cos(pi/2)` is not exactly zero but a value the size of the floating-point accuracy, `eps`, because `pi` is only a floating-point approximation to the exact value of π.

Definition

The cosine can be defined as

$$\cos(x+iy) = \cos(x)\cosh(y) - i\sin(x)\sinh(y)$$

$$\cos(z) = \frac{e^{iz}+e^{-iz}}{2}$$

Algorithm

`cos` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`cosd`, `cosh`, `acos`, `acosd`, `acosh`

Purpose

Cosine ofo argument in degrees

Syntax

```
Y = cosd(X)
```

Description

`Y = cosd(X)` is the cosine of the elements of `X`, expressed in degrees. For odd integers `n`, `cosd(n*90)` is exactly zero, whereas `cos(n*pi/2)` reflects the accuracy of the floating point value of `pi`.

See Also

`cos`, `cosh`, `acos`, `acosd`, `acosh`

cosh

Purpose

Hyperbolic cosine

Syntax

```
Y = cosh(X)
```

Description

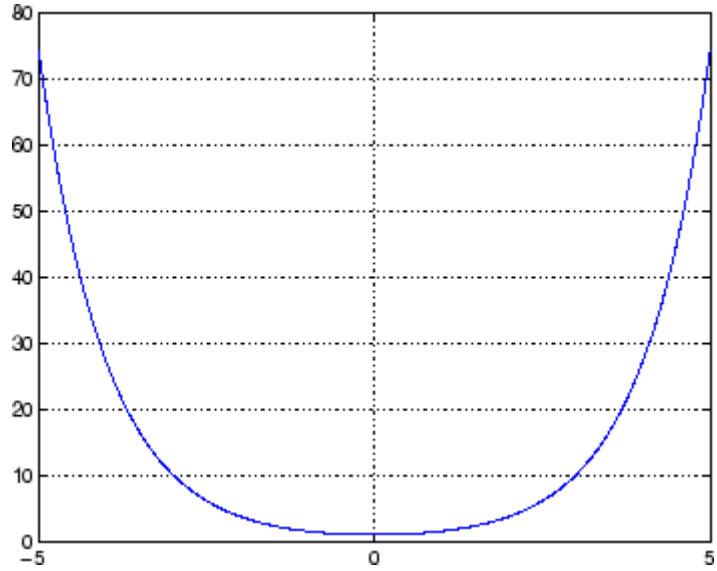
The `cosh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = cosh(X)` returns the hyperbolic cosine for each element of `X`.

Examples

Graph the hyperbolic cosine function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,cosh(x)), grid on
```



Definition

The hyperbolic cosine can be defined as

$$\cosh(z) = \frac{e^{z} + e^{-z}}{2}$$

Algorithm

`cosh` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`acos`, `acosh`, `cos`

Purpose

Cotangent of argument in radians

Syntax

```
Y = cot(X)
```

Description

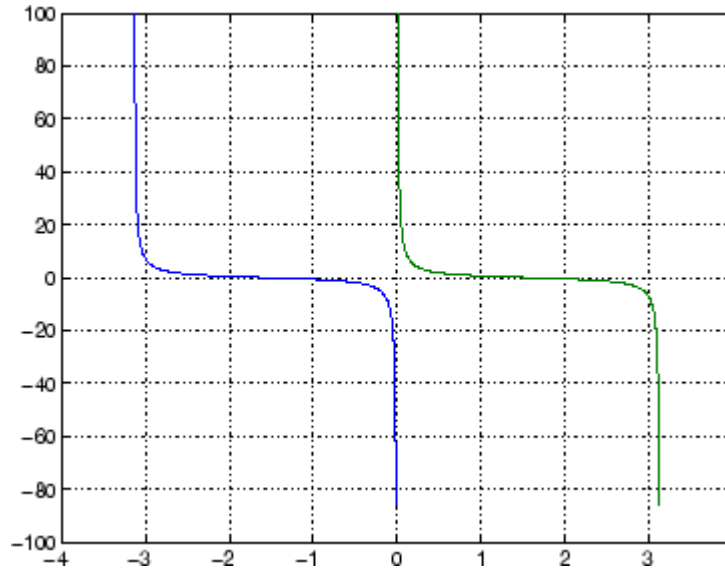
The `cot` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = cot(X)` returns the cotangent for each element of `X`.

Examples

Graph the cotangent the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,cot(x1),x2,cot(x2)), grid on
```



Definition

The cotangent can be defined as

$$\cot(z) = \frac{1}{\tan(z)}$$

Algorithm

`cot` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`cotd`, `coth`, `acot`, `acotd`, `acoth`

cotd

Purpose

Cotangent of argument in degrees

Syntax

```
Y = cotd(X)
```

Description

`Y = cotd(X)` is the cotangent of the elements of `X`, expressed in degrees. For integers `n`, `cotd(n*180)` is infinite, whereas `cot(n*pi)` is large but finite, reflecting the accuracy of the floating point value of `pi`.

See Also

`cot`, `coth`, `acot`, `acotd`, `acoth`

Purpose

Hyperbolic cotangent

Syntax

```
Y = coth(X)
```

Description

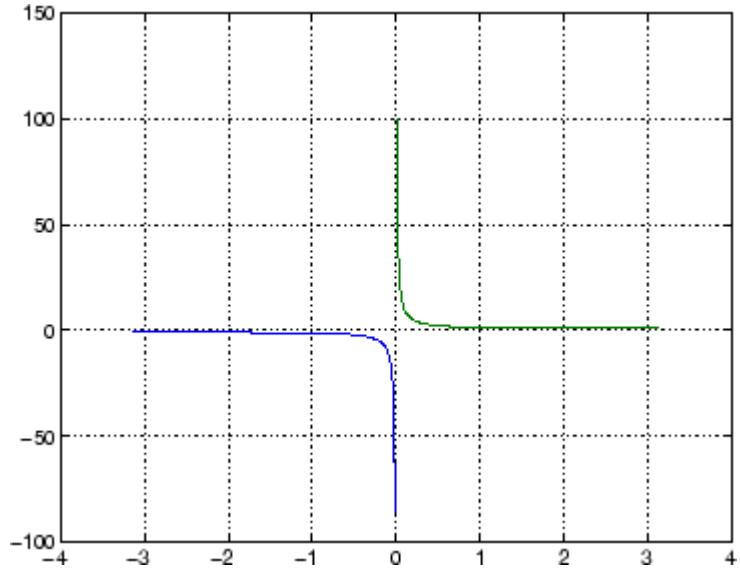
The `coth` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = coth(X)` returns the hyperbolic cotangent for each element of `X`.

Examples

Graph the hyperbolic cotangent over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,coth(x1),x2,coth(x2)), grid on
```



Definition

The hyperbolic cotangent can be defined as

$$\coth(z) = \frac{1}{\tanh(z)}$$

Algorithm

`coth` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`acot`, `acoth`, `cot`

Purpose

Covariance matrix

Syntax

```
cov(x)
cov(x) or cov(x,y)
cov(x,1) or cov(x,y,1)
```

Description

`cov(x)`, if `X` is a vector, returns the variance. For matrices, where each row is an observation, and each column is a variable, `cov(X)` is the covariance matrix. `diag(cov(X))` is a vector of variances for each column, and `sqrt(diag(cov(X)))` is a vector of standard deviations. `cov(X,Y)`, where `X` and `Y` are matrices with the same number of elements, is equivalent to `cov([X(:)  Y(:)])`.

`cov(x) or cov(x,y)` normalizes by N-1, if N>1, where N is the number of observations. This makes `cov(X)` the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For N=1, `cov normalizes by N.`

`cov(x,1) or cov(x,y,1)` normalizes by N and produces the second moment matrix of the observations about their mean. `cov(X,Y,0)` is the same as `cov(X,Y)` and `cov(X,0)` is the same as `cov(X)`.

Remarks

`cov` removes the mean from each column before calculating the result.

The *covariance* function is defined as

$$\mathrm{cov}(x_1,x_2) = E[(x_1-\mu_1)(x_2-\mu_2)]$$

where E is the mathematical expectation and $\mu_i = Ex_i$.

Examples

Consider `A = [-1 1 2 ; -2 3 1 ; 4 0 3]`. To obtain a vector of variances for each column of `A`:

```
v = diag(cov(A))'
v =
   10.3333    2.3333    1.0000
```

Compare vector `v` with covariance matrix `C`:

```
C =
   10.3333   -4.1667    3.0000
   -4.1667    2.3333   -1.5000
    3.0000   -1.5000    1.0000
```

The diagonal elements `C(i,i)` represent the variances for the columns of `A`. The off-diagonal elements `C(i,j)` represent the covariances of columns `i` and `j`.

See Also

`corrcoef`, `mean`, `median`, `std`, `var`

`xcorr`, `xcov` in the Signal Processing Toolbox

Purpose

Sort complex numbers into complex conjugate pairs

Syntax

```
B = cplxpair(A)
B = cplxpair(A,tol)
B = cplxpair(A,[],dim)
B = cplxpair(A,tol,dim)
```

Description

`B = cplxpair(A)` sorts the elements along different dimensions of a complex array, grouping together complex conjugate pairs.

The conjugate pairs are ordered by increasing real part. Within a pair, the element with negative imaginary part comes first. The purely real values are returned following all the complex pairs. The complex conjugate pairs are forced to be exact complex conjugates. A default tolerance of `100*eps` relative to `abs(A(i))` determines which numbers are real and which elements are paired complex conjugates.

If `A` is a vector, `cplxpair(A)` returns `A` with complex conjugate pairs grouped together.

If `A` is a matrix, `cplxpair(A)` returns `A` with its columns sorted and complex conjugates paired.

If `A` is a multidimensional array, `cplxpair(A)` treats the values along the first non-singleton dimension as vectors, returning an array of sorted elements.

`B = cplxpair(A,tol)` overrides the default tolerance.

`B = cplxpair(A,[],dim)` sorts `A` along the dimension specified by scalar dim.

`B = cplxpair(A,tol,dim)` sorts `A` along the specified dimension and overrides the default tolerance.

Diagnostics

If there are an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message

```
Complex numbers can't be paired.
```

cputime

Purpose

Elapsed CPU time

Syntax

```
cputime
```

Description

`cputime` returns the total CPU time (in seconds) used by MATLAB from the time it was started. This number can overflow the internal representation and wrap around.

Remarks

Although it is possible to measure performance using the `cputime` function, it is recommended that you use the `tic` and `toc` functions for this purpose exclusively. See Using tic and toc Versus the cputime Function in the MATLAB Programming documentation for more information.

Examples

The following code returns the CPU time used to run `surf(peaks(40))`.

```
t = cputime; surf(peaks(40)); e = cputime-t

e =
     0.4667
```

See Also

`clock`, `etime`, `tic`, `toc`

Purpose

Create MATLAB object based on WSDL file

Syntax

```
createClassFromWsdl('source')
```

Description

`createClassFromWsdl('source')` creates a MATLAB object based on a Web Services Description Language (WSDL) application program interface (API). The `source` argument specifies a URL or path to a WSDL API, which defines Web service methods, arguments, and transactions. It returns the name of the new class.

Based on the WSDL API, the `createClassFromWsdl` function creates a new folder in the current directory. The folder contains an M-file for each Web service method. In addition, two default M-files are created: the object's display method (`display.m`) and its constructor (`servicename.m`).

For example, if `myWebService` offers two methods (`method1` and `method2`), the `createClassFromWsdl` function creates

- `@myWebService` folder in the current directory
- `method1.m` — M-file for `method1`
- `method2.m` — M-file for `method2`
- `display.m` — Default M-file for `display` method
- `myWebService.m` — Default M-file for the `myWebService` MATLAB object

Remarks

For more information about WSDL and Web services, see the following resources:

- World Wide Web Consortium (W3C) WSDL specification
- W3C SOAP specification
- XMethods

Example

The following example calls a Web service that returns the stock price for an stock symbol.

```
cd(tempdir)
% Create a class for the Web service
% provided by xmethods.net
url = 'http://services.xmethods.net/soap/
  urn:xmethods-delayed-quotes.wsdl';
createClassFromWsdl(url);
% Instantiate the object
service = StockQuoteService;
% getQuote returns the price of a stock
getQuote(service, 'GOOG');
```

See Also

`callSoapService`, `createSoapMessage`, `parseSoapResponse`

Purpose

Create copy of `inputParser` object

Syntax

```
p.createCopy
createCopy(p)
```

Description

`p.createCopy` creates a copy of `inputParser` object `p`. Because the `inputParser` class uses handle semantics, a normal assignment statement does not create a copy.

`createCopy(p)` is functionally the same as the syntax above.

Note For more information on the `inputParser` class, see Parsing Inputs with inputParser in the MATLAB Programming documentation.

Examples

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class. Construct an instance of `inputParser` and assign it to variable `p`:

```
function publish_ip(script, varargin)
p = inputParser;   % Create an instance of the inputParser class.
```

Add arguments to the schema. See the reference pages for the `addRequired`, `addOptional`, and `addParamValue` methods for help with this:

```
p.addRequired('script', @ischar);
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Make a copy of object `p`, assigning it to variable `x`. Use the `Parameters` property of `inputParser` to list the arguments belonging to each object:

```
disp(' ')
```

```
disp 'The input parameters for object p are'
disp(p.Parameters')

x = p.createCopy;

disp(' ')
disp 'The input parameters for the copy of object p are'
disp(x.Parameters')
```

Save the M-file using the **Save** option on the MATLAB **File** menu, and then run it:

```
publish_ip('ipscript.m', 'ppt', 'maxWidth', 500, 'MAXHeight', 300);

The input parameters for object p are
    'format'
    'maxHeight'
    'maxWidth'
    'outputDir'
    'script'

The input parameters for the copy of object p are
    'format'
    'maxHeight'
    'maxWidth'
    'outputDir'
    'script'
```

See Also

`inputParser`, `addRequired(inputParser)`, `addOptional(inputParser)`, `addParamValue(inputParser)`, `parse(inputParser)`

Purpose

Create SOAP message to send to server

Syntax

```
createSoapMessage(namespace, method, values, names, types,
   style)
```

Description

`createSoapMessage(namespace, method, values, names, types, style)` creates a SOAP message. `values`, `names`, and `types` are cell arrays. `names` defaults to dummy names and `types` defaults to unspecified. The optional *`style`* argument specifies '**`document`**' or '**`rpc`**' messages; **`rpc`** is the default.

Example

```
message = createSoapMessage(...
 'urn:xmethods-delay-quotes',...
 'getQuote', ...
 {'GOOG'}, ...
 {'symbol'}, ...
 {'http://www.w3.org/2001/XMLSchema}string'}, ...
 'rpc');
response = callSoapService( ...
 'http://64.124.140.30:9090/soap', ...
 'urn:xmethods-delayed-quotes#getQuote' ...
 message);
price = parseSoapResponse(response)
```

See Also

`callSoapService`, `createClassFromWsdl`, `parseSoapResponse`

Purpose

Vector cross product

Syntax

```
C = cross(A,B)
C = cross(A,B,dim)
```

Description

`C = cross(A,B)` returns the cross product of the vectors `A` and `B`. That is, `C = A x B`. `A` and `B` must be 3-element vectors. If `A` and `B` are multidimensional arrays, `cross` returns the cross product of `A` and `B` along the first dimension of length 3.

`C = cross(A,B,dim)` where `A` and `B` are multidimensional arrays, returns the cross product of `A` and `B` in dimension `dim`. `A` and `B` must have the same size, and both `size(A,dim)` and `size(B,dim)` must be 3.

Remarks

To perform a dot (scalar) product of two vectors of the same size, use `c = dot(a,b)`.

Examples

The cross and dot products of two vectors are calculated as shown:

```
a = [1 2 3];
b = [4 5 6];
c = cross(a,b)

c =
      -3     6    -3

d = dot(a,b)

d =
      32
```

See Also

`dot`

Purpose

Cosecant of argument in radians

Syntax

```
Y = csc(x)
```

Description

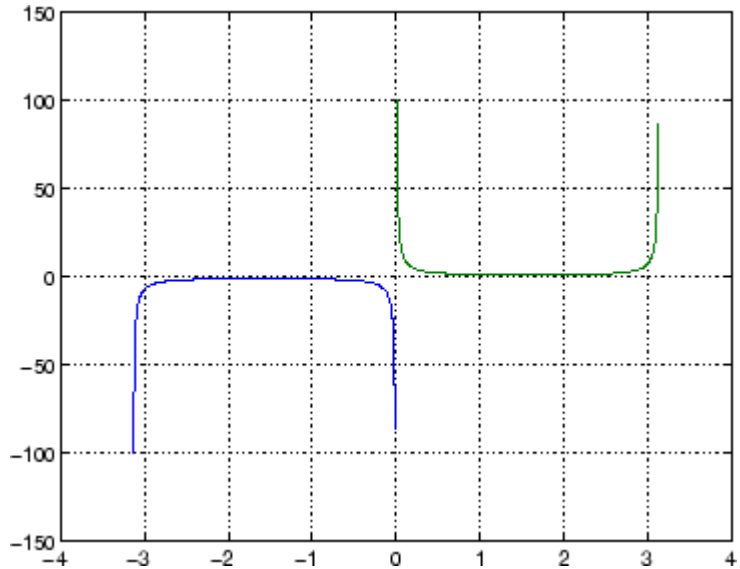
The `csc` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = csc(x)` returns the cosecant for each element of `x`.

Examples

Graph the cosecant over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csc(x1),x2,csc(x2)), grid on
```



Definition

The cosecant can be defined as

$$\csc(z) = \frac{1}{\sin(z)}$$

Algorithm

`csc` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`cscd`, `csch`, `acsc`, `acscd`, `acsch`

Purpose

Cosecant of argument in degrees

Syntax

```
Y = cscd(X)
```

Description

`Y = cscd(X)` is the cosecant of the elements of `X`, expressed in degrees. For integers `n`, `cscd(n*180)` is infinite, whereas `csc(n*pi)` is large but finite, reflecting the accuracy of the floating point value of `pi`.

See Also

`csc`, `csch`, `acsc`, `acscd`, `acsch`

Purpose

Hyperbolic cosecant

Syntax

```
Y = csch(x)
```

Description

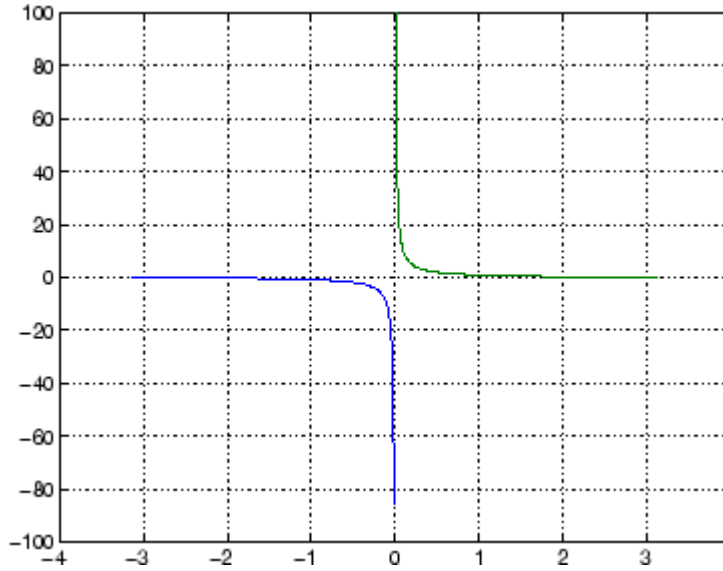
The `csch` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = csch(x)` returns the hyperbolic cosecant for each element of `x`.

Examples

Graph the hyperbolic cosecant over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csch(x1),x2,csch(x2)), grid on
```



Definition

The hyperbolic cosecant can be defined as

$$\operatorname{csch}(z) = \frac{1}{\sinh(z)}$$

Algorithm

`csch` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`acsc`, `acsch`, `csc`

Purpose

Read comma-separated value file

Syntax

```
M = csvread(filename)
M = csvread(filename, row, col)
M = csvread(filename, row, col, range)
```

Description

`M = csvread(filename)` reads a comma-separated value formatted file, `filename`. The `filename` input is a string enclosed in single quotes. The result is returned in `M`. The file can only contain numeric values.

`M = csvread(filename, row, col)` reads data from the comma-separated value formatted file starting at the specified row and column. The row and column arguments are zero based, so that `row=0` and `col=0` specify the first value in the file.

`M = csvread(filename, row, col, range)` reads only the range specified. Specify `range` using the notation `[R1 C1 R2 C2]` where (`R1`,`C1`) is the upper left corner of the data to be read and (`R2`,`C2`) is the lower right corner. You can also specify the range using spreadsheet notation, as in `range = 'A1..B7'`.

Remarks

`csvread` fills empty delimited fields with zero. Data files having lines that end with a nonspace delimiter, such as a semicolon, produce a result that has an additional last column of zeros.

`csvread` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. Valid forms for a complex number are

Form	Example
`–<real>–<imag>i|j`	`5.7-3.1i`
`–<imag>i|j`	`-7j`

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

Examples

Given the file `csvlist.dat` that contains the comma-separated values

```
02, 04, 06, 08, 10, 12
03, 06, 09, 12, 15, 18
05, 10, 15, 20, 25, 30
07, 14, 21, 28, 35, 42
11, 22, 33, 44, 55, 66
```

To read the entire file, use

```
csvread('csvlist.dat')

ans =

     2     4     6     8    10    12
     3     6     9    12    15    18
     5    10    15    20    25    30
     7    14    21    28    35    42
    11    22    33    44    55    66
```

To read the matrix starting with zero-based row 2, column 0, and assign it to the variable `m`,

```
m = csvread('csvlist.dat', 2, 0)

m =

     5    10    15    20    25    30
     7    14    21    28    35    42
    11    22    33    44    55    66
```

To read the matrix bounded by zero-based (2,0) and (3,3) and assign it to `m`,

```
m = csvread('csvlist.dat', 2, 0, [2,0,3,3])

m =
```

```
     5    10    15    20
     7    14    21    28
```

See Also

csvwrite, dlmread, textscan, wk1read, file formats, importdata, uiimport

Purpose

Write comma-separated value file

Syntax

```
csvwrite(filename,M)
csvwrite(filename,M,row,col)
```

Description

`csvwrite(filename,M)` writes matrix `M` into `filename` as comma-separated values. The `filename` input is a string enclosed in single quotes.

`csvwrite(filename,M,row,col)` writes matrix `M` into `filename` starting at the specified row and column offset. The row and column arguments are zero based, so that `row=0` and `C=0` specify the first value in the file.

Remarks

`csvwrite` terminates each line with a line feed character and no carriage return.

Examples

The following example creates a comma-separated value file from the matrix `m`.

```
m = [3 6 9 12 15; 5 10 15 20 25; ...
     7 14 21 28 35; 11 22 33 44 55];

csvwrite('csvlist.dat',m)
type csvlist.dat

3,6,9,12,15
5,10,15,20,25
7,14,21,28,35
11,22,33,44,55
```

The next example writes the matrix to the file, starting at a column offset of `2`.

```
csvwrite('csvlist.dat',m,0,2)
type csvlist.dat
```

```
,,3,6,9,12,15
,,5,10,15,20,25
,,7,14,21,28,35
,,11,22,33,44,55
```

See Also

csvread, dlmwrite, wk1write, file formats, importdata, uiimport

Purpose

Transpose `timeseries` object

Syntax

```
ts1 = ctranspose(ts)
```

Description

`ts1 = ctranspose(ts)` returns a new `timeseries` object `ts1` with `IsTimeFirst` value set to the opposite of what it is for `ts`. For example, if `ts` has the first data dimension aligned with the time vector, `ts1` has the last data dimension aligned with the time vector as a result of this operation.

Remarks

The `ctranspose` function that is overloaded for `timeseries` objects does not transpose the data. Instead, this function changes whether the first or the last dimension of the data is aligned with the time vector.

Note To transpose the data, you must transpose the `Data` property of the `timeseries` object. For example, you can use the syntax `ctranspose(ts.Data)` or `(ts.Data)'`. `Data` must be a 2-D array.

Consider a `timeseries` object with 10 samples with the property `IsTimeFirst = True`. When you transpose this object, the data size is changed from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

The following table summarizes how MATLAB displays the size for Data property of the `timeseries` object (up to three dimensions) before and after transposing.

Data Size Before and After Transposing

Size of Original Data	Size of Transposed Data
N-by-1	1-by-1-by-N

Data Size Before and After Transposing (Continued)

Size of Original Data	Size of Transposed Data
N-by-M	M-by-1-by-N
N-by-M-by-L	M-by-L-by-N

Examples

Suppose that a `timeseries` object `ts` has `ts.data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is set to `true`, which means that the first dimension of the data is aligned with the time vector. `ctranspose(ts)` modifies `ts` such that the last dimension of the data is now aligned with the time vector. This permutes the data such that the size of `ts.Data` becomes 3-by-2-by-10.

See Also

`transpose (timeseries)`, `tsprops`

Purpose

Cumulative product

Syntax

```
B = cumprod(A)
B = cumprod(A,dim)
```

Description

`B = cumprod(A)` returns the cumulative product along different dimensions of an array.

If `A` is a vector, `cumprod(A)` returns a vector containing the cumulative product of the elements of `A`.

If `A` is a matrix, `cumprod(A)` returns a matrix the same size as `A` containing the cumulative products for each column of `A`.

If `A` is a multidimensional array, `cumprod(A)` works on the first nonsingleton dimension.

`B = cumprod(A,dim)` returns the cumulative product of the elements along the dimension of `A` specified by scalar `dim`. For example, `cumprod(A,1)` increments the first (row) index, thus working along the rows of `A`.

Examples

```
cumprod(1:5)
ans =
     1  2  6  24  120

A = [1 2 3; 4 5 6];

cumprod(A)
ans =
     1     2     3
     4    10    18

cumprod(A,2)
ans =
     1     2     6
     4    20   120
```

See Also

cumsum, prod, sum

Purpose

Cumulative sum

Syntax

```
B = cumsum(A)
B = cumsum(A,dim)
```

Description

`B = cumsum(A)` returns the cumulative sum along different dimensions of an array.

If `A` is a vector, `cumsum(A)` returns a vector containing the cumulative sum of the elements of `A`.

If `A` is a matrix, `cumsum(A)` returns a matrix the same size as `A` containing the cumulative sums for each column of `A`.

If `A` is a multidimensional array, `cumsum(A)` works on the first nonsingleton dimension.

`B = cumsum(A,dim)` returns the cumulative sum of the elements along the dimension of `A` specified by scalar `dim`. For example, `cumsum(A,1)` works across the first dimension (the rows).

Examples

```
cumsum(1:5)
ans =
       [1  3  6  10  15]

A = [1 2 3; 4 5 6];

cumsum(A)
ans =
       1     2     3
       5     7     9

cumsum(A,2)
ans =
       1     3     6
       4     9    15
```

See Also

`cumprod`, `prod`, `sum`

Purpose

Cumulative trapezoidal numerical integration

Syntax

```
Z = cumtrapz(Y)
Z = cumtrapz(X,Y)
Z = cumtrapz(X,Y,dim) or cumtrapz(Y,dim)
```

Description

`Z = cumtrapz(Y)` computes an approximation of the cumulative integral of `Y` via the trapezoidal method with unit spacing. To compute the integral with other than unit spacing, multiply `Z` by the spacing increment. Input `Y` can be complex.

For vectors, `cumtrapz(Y)` is a vector containing the cumulative integral of `Y`.

For matrices, `cumtrapz(Y)` is a matrix the same size as `Y` with the cumulative integral over each column.

For multidimensional arrays, `cumtrapz(Y)` works across the first nonsingleton dimension.

`Z = cumtrapz(X,Y)` computes the cumulative integral of `Y` with respect to `X` using trapezoidal integration. `X` and `Y` must be vectors of the same length, or `X` must be a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`. `cumtrapz` operates across this dimension. Inputs `X` and `Y` can be complex.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `cumtrapz(X,Y)` operates across this dimension.

`Z = cumtrapz(X,Y,dim) or cumtrapz(Y,dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X` must be the same as `size(Y,dim)`.

Example

Example 1

```
Y = [0 1 2; 3 4 5];

cumtrapz(Y,1)
ans =
0         0         0
```

```
    1.5000   2.5000    3.5000

cumtrapz(Y,2)
ans =
0    0.5000    2.0000
        0    3.5000    8.0000
```

Example 2

This example uses two complex inputs:

```
z = exp(1i*pi*(0:100)/100);

ct = cumtrapz(z,1./z);
ct(end)
ans =
   0.0000 + 3.1411i
```

See Also

cumsum, trapz

curl

Purpose

Compute curl and angular velocity of vector field

Syntax

```
[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)
[curlx,curly,curlz,cav] = curl(U,V,W)
[curlz,cav]= curl(X,Y,U,V)
[curlz,cav]= curl(U,V)
[curlx,curly,curlz] = curl(...), curlx,curly] = curl(...)
cav = curl(...)
```

Description

`[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)` computes the curl and angular velocity perpendicular to the flow (in radians per time unit) of a 3-D vector field `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`).

`[curlx,curly,curlz,cav] = curl(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`[curlz,cav]= curl(X,Y,U,V)` computes the curl z-component and the angular velocity perpendicular to `z` (in radians per time unit) of a 2-D vector field `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`[curlz,cav]= curl(U,V)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[m,n] = size(U)`.

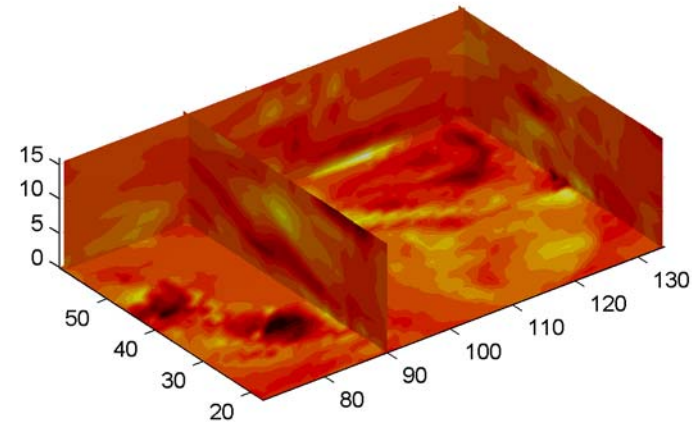
`[curlx,curly,curlz] = curl(...), curlx,curly] = curl(...)` returns only the curl.

`cav = curl(...)` returns only the curl angular velocity.

Examples

This example uses colored slice planes to display the curl angular velocity at specified locations in the vector field.

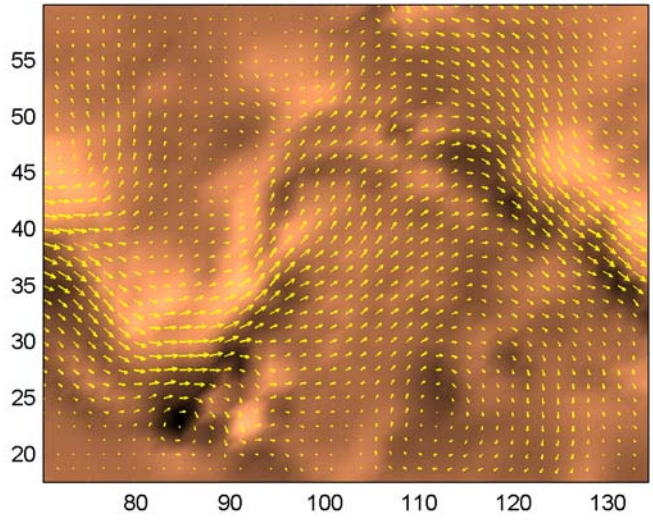
```
load wind
cav = curl(x,y,z,u,v,w);
slice(x,y,z,cav,[90 134],[59],[0]);
shading interp
daspect([1 1 1]); axis tight
colormap hot(16)
camlight
```



This example views the curl angular velocity in one plane of the volume and plots the velocity vectors (`quiver`) in the same plane.

```
load wind
k = 4;
x = x(:,:,k); y = y(:,:,k); u = u(:,:,k); v = v(:,:,k);
cav = curl(x,y,u,v);
pcolor(x,y,cav); shading interp
hold on;
quiver(x,y,u,v,'y')
```

```
hold off
colormap copper
```



See Also

`streamribbon`, `divergence`

"Volume Visualization" on page 1-102 for related functions

"Example — Displaying Curl with Stream Ribbons" for another example

Purpose

Allow custom source control system (UNIX)

Syntax

```
customerverctrl
```

Description

`customerverctrl` function is for customers who want to integrate a source control system that is not supported with MATLAB. When using this function, conform to the structure of one of the supported version control systems, for example, RCS. For examples, see the files `clearcase.m`, `cvs.m`, `pvcs.m`, and `rcs.m` in *`matlabroot`*`\toolbox\matlab\verctrl`.

See Also

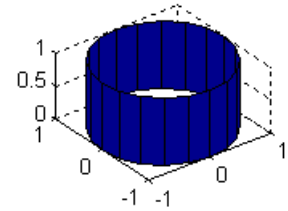
`checkin`, `checkout`, `cmopts`, `undocheckout`

For Windows platforms, use `verctrl`.

cylinder

Purpose

Generate cylinder



Syntax

```
[X,Y,Z] = cylinder
[X,Y,Z] = cylinder(r)
[X,Y,Z] = cylinder(r,n)
cylinder(axes_handle,...)
cylinder(...)
```

Description

`cylinder` generates *x*-, *y*-, and *z*-coordinates of a unit cylinder. You can draw the cylindrical object using `surf` or `mesh`, or draw it immediately by not providing output arguments.

`[X,Y,Z] = cylinder` returns the *x*-, *y*-, and *z*-coordinates of a cylinder with a radius equal to `1`. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder using `r` to define a profile curve. `cylinder` treats each element in `r` as a radius at equally spaced heights along the unit height of the cylinder. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r,n)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder based on the profile curve defined by vector `r`. The cylinder has `n` equally spaced points around its circumference.

`cylinder(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`cylinder(...)`, with no output arguments, plots the cylinder using `surf`.

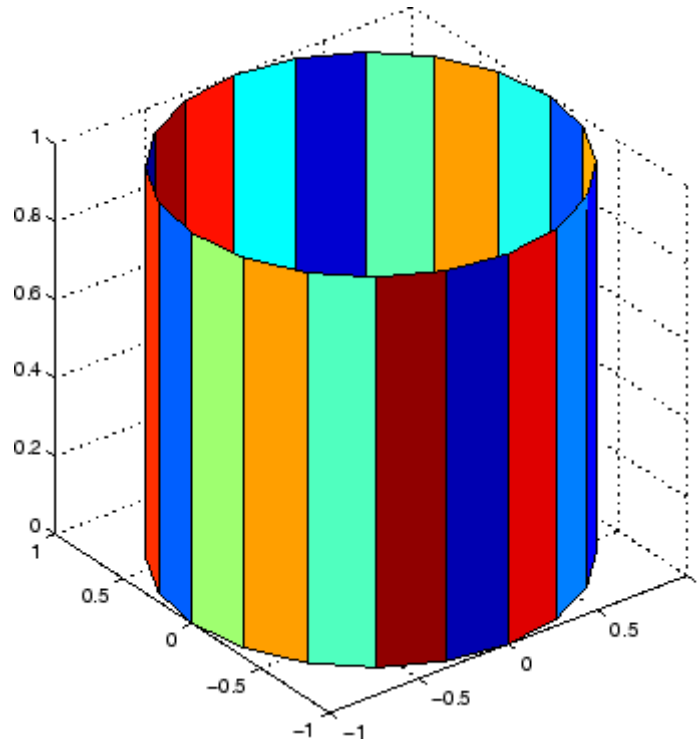
Remarks

`cylinder` treats its first argument as a profile curve. The resulting surface graphics object is generated by rotating the curve about the *x*-axis, and then aligning it with the *z*-axis.

Examples

Create a cylinder with randomly colored faces.

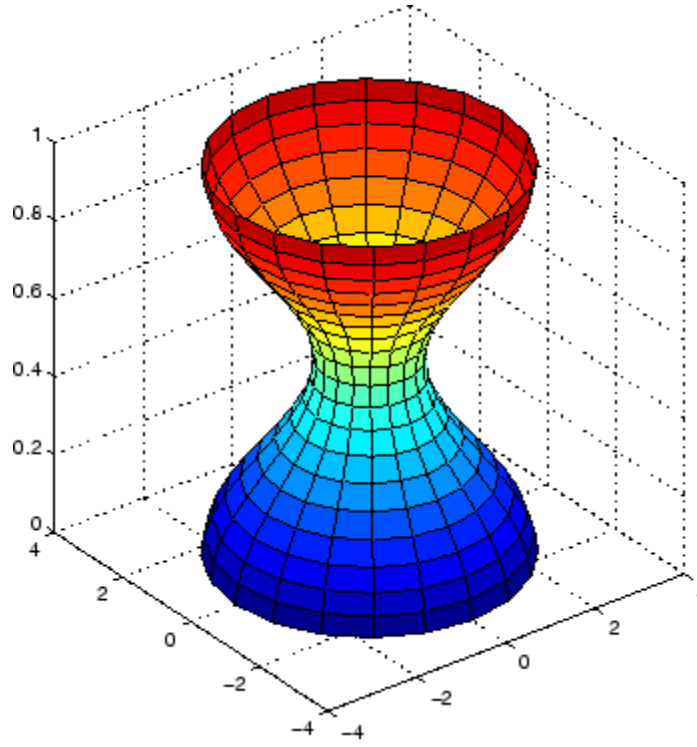
```
cylinder
axis square
h = findobj('Type','surface');
set(h,'CData',rand(size(get(h,'CData'))))
```



Generate a cylinder defined by the profile function `2+sin(t)`.

```
t = 0:pi/10:2*pi;
```

```
[X,Y,Z] = cylinder(2+cos(t));
surf(X,Y,Z)
axis square
```



See Also

`sphere`, `surf`

“Polygons and Surfaces” on page 1-90 for related functions

Purpose

Read Data Acquisition Toolbox (.daq) file

Syntax

```
data = daqread('filename')
[data, time] = daqread(...)
[data, time, abstime] = daqread(...)
[data, time, abstime, events] = daqread(...)
[data, time, abstime, events, daqinfo] = daqread(...)
data = daqread(...,'Param1', Val1,...)
daqinfo = daqread('filename','info')
```

Description

`data = daqread('filename')` reads all the data from the Data Acquisition Toolbox (`.daq`) file specified by `filename`. `daqread` returns `data`, an *m*-by-*n* data matrix, where *m* is the number of samples and *n* is the number of channels. If `data` includes data from multiple triggers, the data from each trigger is separated by a `NaN`. If you set the `OutputFormat` property to `tscollection`, `daqread` returns a time series collection object. See below for more information.

`[data, time] = daqread(...)` returns time/value pairs. `time` is an *m*-by-1 vector, the same length as `data`, that contains the relative time for each sample. Relative time is measured with respect to the first trigger that occurs.

`[data, time, abstime] = daqread(...)` returns the absolute time of the first trigger. `abstime` is returned as a `clock` vector.

`[data, time, abstime, events] = daqread(...)` returns a log of events. `events` is a structure containing event information. If you specify either the`Samples`, `Time`, or `Triggers` parameters (see below), the events structure contains only the specified events.

`[data, time, abstime, events, daqinfo] = daqread(...)` returns a structure, `daqinfo`, that contains two fields: `ObjInfo` and `HwInfo`. `ObjInfo` is a structure containing property name/property value pairs and `HwInfo` is a structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

`data = daqread(...,'Param1', Val1,...)` specifies the amount of data returned and the format of the data, using the following parameters.

Parameter	Description
`Samples`	Specify the sample range.
`Time`	Specify the relative time range.
`Triggers`	Specify the trigger range.
`Channels`	Specify the channel range. Channel names can be specified as a cell array.
`DataFormat`	Specify the data format as `doubles` (default) or `native`.
`TimeFormat`	Specify the time format as `vector` (default) or `matrix`.
`OutputFormat`	Specify the output format as `matrix` (the default) or `tscollection`. When you specify `tscollection`, `daqread` only returns `data`.

The `Samples`, `Time`, and `Triggers` properties are mutually exclusive; that is, either `Samples`, `Triggers` or `Time` can be defined at once.

`daqinfo = daqread('filename','info')` returns metadata from the file in the `daqinfo` structure, without incurring the overhead of reading the data from the file as well. The `daqinfo` structure contains two fields:

`daqinfo.ObjInfo`
: a structure containing parameter/value pairs for the data acquisition object used to create the file, `filename`. Note: The `UserData` property value is not restored.

`daqinfo.HwInfo`
: a structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

Remarks

More About .daq Files

- The format used by `daqread` to return data, relative time, absolute time, and event information is identical to the format used by the `getdata` function that is part of Data Acquisition Toolbox. For more information, see the Data Acquisition Toolbox documentation.
- If data from multiple triggers is read, then the size of the resulting data array is increased by the number of triggers issued because each trigger is separated by a `NaN`.
- `ObjInfo.EventLog` always contains the entire event log regardless of the value specified by `Samples`, `Time`, or `Triggers`.
- The `UserData` property value is not restored when you return device object (`ObjInfo`) information.
- When reading a `.daq` file, the `daqread` function does not return property values that were specified as a cell array.
- Data Acquisition Toolbox (`.daq`) files are created by specifying a value for the `LogFileName` property (or accepting the default value), and configuring the `LoggingMode` property to `Disk` or `Disk&Memory`.

More About Time Series Collection Object Returned

When `OutputFormat` is set to `tscollection`, `daqread` returns a time series collection object. This times series collection object contains an absolute time series object for each channel in the file. The following describes how `daqread` sets some of the properties of the times series collection object and the time series objects.

- The `time` property of the time series collection object is set to the value of the `InitialTriggerTime` property specified in the file.
- The `name` property of each time series object is set to the value of the `Name` property of a channel in the file. If this name cannot be used as a time series object name, `daqread` sets the name to `'Channel'` with the `HwChannel` property of the channel appended.

- The value of the Units property of the time series object depends on the value of the DataFormat parameter. If the DataFormat parameter is set to 'double', daqread sets the DataInfo property of each time series object in the collection to the value of the Units property of the corresponding channel in the file. If the DataFormat parameter is set to 'native', daqread sets the Units property to 'native'. See the Data Acquisition Toolbox documentation for more information on these properties.
- Each time series object will have tsdata.event objects attached corresponding to the log of events associated with the channel.

If daqread returns data from multiple triggers, the data from each trigger is separated by a NaN in the time series data. This increases the length of data and time vectors in the time series object by the number of triggers.

Examples

Use Data Acquisition Toolbox to acquire data. The analog input object, ai, acquires one second of data for four channels, and saves the data to the output file data.daq.

```
ai = analoginput('nidaq','Dev1');
chans = addchannel(ai,0:3);
set(ai,'SampleRate',1000)
ActualRate = get(ai,'SampleRate');
set(ai,'SamplesPerTrigger, ActualRate)
set(ai,'LoggingMode','Disk&Memory')
set(ai,'LogFileName','data.daq')
start(ai)
```

After the data has been collected and saved to a disk file, you can retrieve the data and other acquisition-related information using daqread. To read all the sample-time pairs from data.daq:

```
[data,time] = daqread('data.daq');
```

To read samples 500 to 1000 for all channels from data.daq:

```
data = daqread('data.daq','Samples',[500 1000]);
```

To read only samples 1000 to 2000 of channel indices 2, 4 and 7 in native format from the file, `data.daq`:

```
data = daqread('data.daq', 'Samples', [1000 2000],...
               'Channels', [2 4 7], 'DataFormat', 'native');
```

To read only the data which represents the first and second triggers on all channels from the file, `data.daq`:

```
[data, time] = daqread('data.daq', 'Triggers', [1 2]);
```

To obtain the channel property information from `data.daq`:

```
daqinfo = daqread('data.daq','info');
chaninfo = daqinfo.ObjInfo.Channel;
```

To obtain a list of event types and event data contained by `data.daq`:

```
daqinfo = daqread('data.daq','info');
events = daqinfo.ObjInfo.EventLog;
event_type = {events.Type};
event_data = {events.Data};
```

To read all the data from the file `data.daq` and return it as a time series collection object:

```
data = daqread('data.daq','OutputFormat','tscollection');
```

See Also

Functions

`timeseries`, `tscollection`

For more information about using this function, see the Data Acquisition Toolbox documentation.

daspect

Purpose

Set or query axes data aspect ratio

Syntax

```
daspect
daspect([aspect_ratio])
daspect('mode')
daspect('auto')
daspect('manual')
daspect(axes_handle,...)
```

Description

The data aspect ratio determines the relative scaling of the data units along the *x*-, *y*-, and *z*-axes.

`daspect` with no arguments returns the data aspect ratio of the current axes.

`daspect([aspect_ratio])` sets the data aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the *x*-, *y*-, and *z*-axis scaling (e.g., `[1 1 3]` means one unit in *x* is equal in length to one unit in *y* and three units in *z*).

`daspect('mode')` returns the current value of the data aspect ratio mode, which can be either `auto` (the default) or `manual`. See Remarks.

`daspect('auto')` sets the data aspect ratio mode to `auto`.

`daspect('manual')` sets the data aspect ratio mode to `manual`.

`daspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `daspect` operates on the current axes.

Remarks

`daspect` sets or queries values of the axes object `DataAspectRatio` and `DataAspectRatioMode` properties.

When the data aspect ratio mode is `auto`, MATLAB adjusts the data aspect ratio so that each axis spans the space available in the figure window. If you are displaying a representation of a real-life object, you should set the data aspect ratio to `[1 1 1]` to produce the correct proportions.

Setting a value for data aspect ratio or setting the data aspect ratio mode to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the data aspect ratio to a value, including its current value,

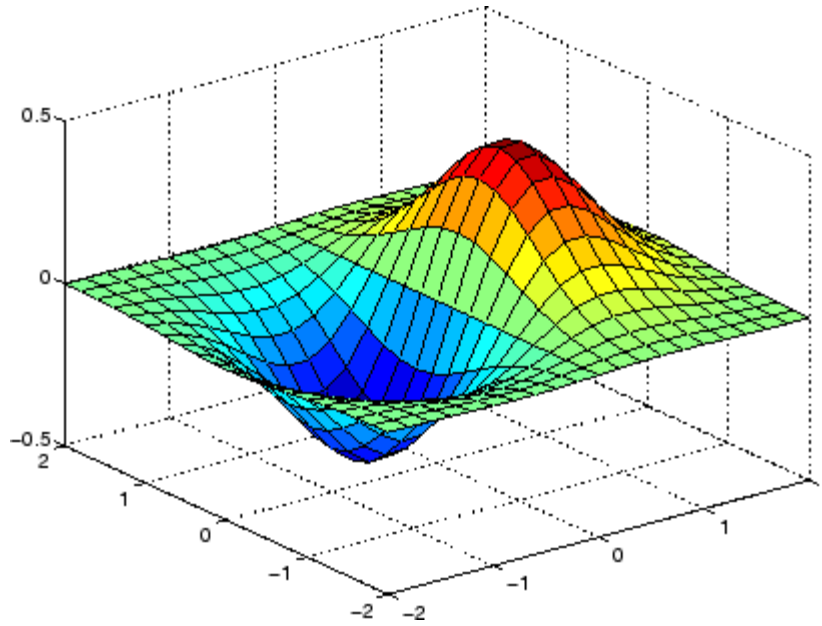
```
daspect(daspect)
```

can cause a change in the way the graphs look. See the Remarks section of the `axes` description for more information.

Examples

The following surface plot of the function $z = xe^{(-x^2 - y^2)}$ is useful to illustrate the data aspect ratio. First plot the function over the range $-2 \le x \le 2$, $-2 \le y \le 2$,

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```

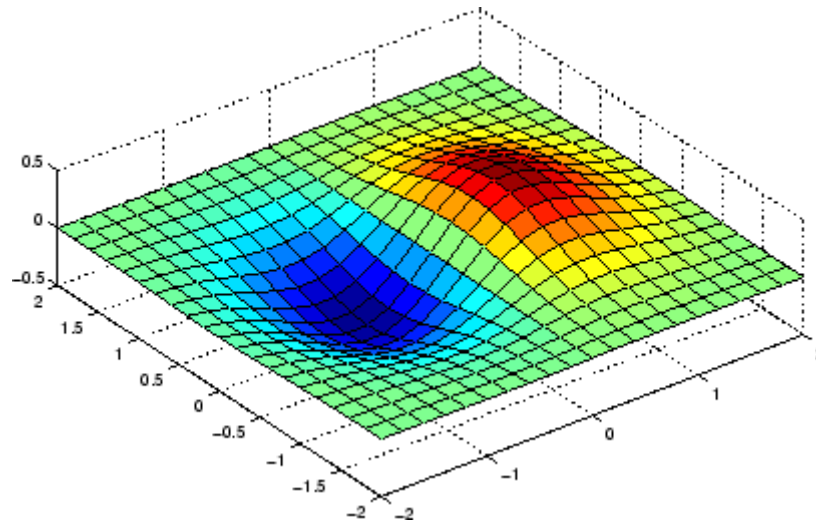


Querying the data aspect ratio shows how MATLAB has drawn the surface.

```
daspect
ans =
     4   4   1
```

Setting the data aspect ratio to `[1 1 1]` produces a surface plot with equal scaling along each axis.

```
daspect([1 1 1])
```



See Also

`axis`, `pbaspect`, `xlim`, `ylim`, `zlim`

The axes properties `DataAspectRatio`, `PlotBoxAspectRatio`, `XLim`, `YLim`, `ZLim`

“Setting the Aspect Ratio and Axis Limits” on page 1-100 for related functions

“Understanding Axes Aspect Ratio” for more information

Purpose

Enable or disable interactive data cursor mode

GUI Alternatives

Use the Data Cursor tool  to label x, y, and z values on graphs and surfaces. For details, see Data Cursor — Displaying Data Values Interactively in the MATLAB Graphics documentation.

Syntax

```
datacursormode on
datacursormode off
datacursormode
datacursormode(figure_handle,...)
dcm_obj = datacursormode(figure_handle)
```

Description

`datacursormode on` enables data cursor mode on the current figure.

`datacursormode off` disables data cursor mode on the current figure.

`datacursormode` toggles data cursor mode on the current figure.

`datacursormode(figure_handle,...)` enables or disables data cursor mode on the specified figure.

`dcm_obj = datacursormode(figure_handle)` returns the figure's data cursor mode object, which enables you to customize the data cursor. See "Data Cursor Mode Object" on page 2-751.

Data Cursor Mode Object

The data cursor mode object has properties that enable you to controls certain aspects of the data cursor. You can use the `set` and `get` commands and the returned object (`dcm_obj` in the above syntax) to set and query property values.

Data Cursor Mode Properties

`Enable`
on | off

Specifies whether this mode is currently enabled on the figure.

`SnapToDataVertex`
on | off

Specifies whether the data cursor snaps to the nearest data value or is located at the actual pointer position.

`DisplayStyle`
datatip | window

Determines how the data is displayed.

- `datatip` displays cursor information in a yellow text box next to a marker indicating the actual data point being displayed.
- `window` displays cursor information in a floating window within the figure.

`Updatefcn`
function handle

This property references a function that customizes the text appearing in the data cursor. The function handle must reference a function that has two implicit arguments (these arguments are automatically passed to the function by MATLAB when the function executes). For example, the following function definition line uses the required arguments:

```
function output_txt = myfunction(obj,event_obj)
% obj    Currently not used (empty)
% event_obj    Handle to event object
% output_txt    Data cursor text string (string or cell array of
%     strings).
```

`event_obj` is an object having the following read-only properties.

- `Target` — Handle of the object the data cursor is referencing (the object on which the user clicked).
- `Position` — An array specifying the x, y, (and z for 3-D graphs) coordinates of the cursor.

You can query these properties within your function. For example,

```
pos = get(event_obj,'Position');
```

returns the coordinates of the cursor.

See `Function Handles` for more information on creating a function handle.

See “Change Data Cursor Text” on page 2-755 for an example.

Data Cursor Method

You can use the `getCursorInfo` function with the data cursor mode object (`dcm_obj` in the above syntax) to obtain information about the data cursor. For example,

```
info_struct = getCursorInfo(dcm_obj);
```

returns a vector of structures, one for each data cursor on the graph. Each structure has the following fields:

- `Target` — The handle of the graphics object containing the data point.
- `Position` — An array specifying the x, y, (and z) coordinates of the cursor.

Line and lineseries objects have an additional field:

- `DataIndex` — A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

Examples

This example creates a plot and enables data cursor mode from the command line.

```
surf(peaks)
datacursormode on
% Click mouse on surface to display data cursor
```

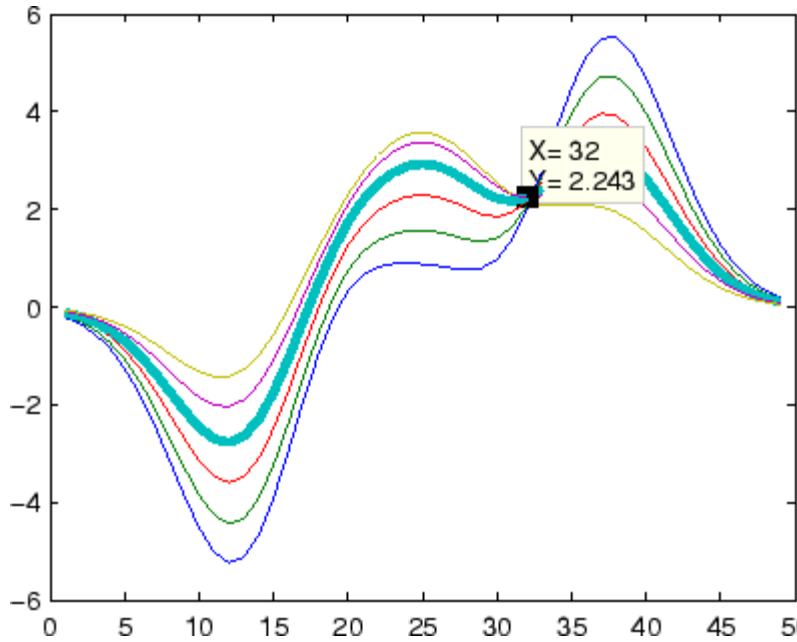
Setting Data Cursor Mode Options

This example enables data cursor mode on the current figure and sets data cursor mode options. The following statements

- Create a graph

- Toggle data cursor mode to on
- Save the data cursor mode object to specify options and get the handle of the line to which the datatip is attached

```
fig = figure;
z = peaks;
plot(z(:,30:35))
dcm_obj = datacursormode(fig);
set(dcm_obj,'DisplayStyle','datatip',...
'SnapToDataVertex','off','Enable','on')

% Click on line to place datatip

c_info = getCursorInfo(dcm_obj);
set(c_info.Target,'LineWidth',2) % Make
selected line wider
```



Change Data Cursor Text

This example shows you how to customize the text that is displayed by the data cursor. Suppose you want to replace the text displayed in the datatip and data window with “Time:” and “Amplitude:”

```
function doc_datacursormode
fig = figure;
a = -16; t = 0:60;
plot(t,sin(a*t))
dcm_obj = datacursormode(fig);
set(dcm_obj,'UpdateFcn',@myupdatefcn)

% Click on line to select data point

function txt = myupdatefcn(empt,event_obj)
pos = get(event_obj,'Position');
txt = {['Time: ',num2str(pos(1))],...
 ['Amplitude: ',num2str(pos(2))]};
```

datatipinfo

Purpose

Produce short description of input variable

Syntax

```
datatipinfo(var)
```

Description

`datatipinfo(var)` displays a short description of a variable, similar to what is displayed in a datatip in the MATLAB debugger.

Examples

Get datatip information for a 5-by-5 matrix:

```
A = rand(5);

datatipinfo(A)
A: 5x5 double =
    0.4445    0.3567    0.7458    0.0767    0.4400
    0.7962    0.6575    0.3918    0.8289    0.9746
    0.5641    0.9808    0.0265    0.4838    0.6722
    0.9099    0.9653    0.2508    0.4859    0.4054
    0.2857    0.5198    0.7383    0.9301    0.9604
```

Get datatip information for a 50-by-50 matrix. For this larger matrix, `datatipinfo` displays just the size and data type:

```
A = rand(50);

datatipinfo(A)
A: 50x50 double
```

Also for multidimensional matrices, `datatipinfo` displays just the size and data type:

```
A = rand(5);
A(:,:,2) = A(:,:,1);

datatipinfo(A)
A: 5x5x2 double
```

See Also

`debug`

Purpose

Current date string

Syntax

```
str = date
```

Description

`str = date` returns a string containing the date in `dd-mmm-yyyy` format.

See Also

`clock`, `datenum`, `now`

datenum

Purpose

Convert date and time to serial date number

Syntax

```
N = datenum(V)
N = datenum(S, F)
N = datenum(S, F, P)
N = datenum([S, P, F])
N = datenum(Y, M, D)
N = datenum(Y, M, D, H, MN, S)
N = datenum(S)
N = datenum(S, P)
```

Description

`datenum` is one of three conversion functions that enable you to express dates and times in any of three formats in MATLAB: a string (or *date string*), a vector of date and time components (or *date vector*), or as a numeric offset from a known date in time (or *serial date number*). Here is an example of a date and time expressed in the three MATLAB formats:

```
Date String:           '24-Oct-2003 12:45:07'
Date Vector:           [2003  10  24  12  45  07]
Serial Date Number:    7.3188e+005
```

A serial date number represents the whole and fractional number of days from a specific date and time, where `datenum('Jan-1-0000 00:00:00')` returns the number 1. (The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time.)

`N = datenum(V)` converts one or more date vectors `V` to serial date numbers `N`. Input `V` can be an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors respectively. A full date vector has six elements, specifying year, month, day, hour, minute, and second, in that order. A partial date vector has three elements, specifying year, month, and day, in that order. Each element of `V` must be a positive double-precision number. `datenum` returns a column vector of `m` date numbers, where `m` is the total number of date vectors in `V`.

`N = datenum(S, F)` converts one or more date strings `S` to serial date numbers `N` using format string `F` to interpret each date string. Input `S`

can be a one-dimensional character array or cell array of date strings. All date strings in S must have the same format, and that format must match one of the date string formats shown in the help for the datestr function. datenum returns a column vector of m date numbers, where m is the total number of date strings in S. MATLAB considers date string years that are specified with only two characters (e.g., '79') to fall within 100 years of the current year.

See the datestr reference page to find valid string values for F. These values are listed in Table 1 in the column labeled "Dateform String." You can use any string from that column except for those that include the letter Q in the string (for example, 'QQ-YYYY'). Certain formats may not contain enough information to compute a date number. In these cases, hours, minutes, seconds, and milliseconds default to 0, the month defaults to January, the day to 1, and the year to the current year.

N = datenum(S, F, P) converts one or more date strings S to date numbers N using format F and pivot year P. The pivot year is used in interpreting date strings that have the year specified as two characters. It is the starting year of the 100-year range in which a two-character date string year resides. The default pivot year is the current year minus 50 years.

N = datenum([S, P, F]) is the same as the syntax shown above, except the order of the last two arguments are switched.

N = datenum(Y, M, D) returns the serial date numbers for corresponding elements of the Y, M, and D (year, month, day) arrays. Y, M, and D must be arrays of the same size (or any can be a scalar) of type double. You can also specify the input arguments as a date vector, [Y M D].

For this and the following syntax, values outside the normal range of each array are automatically carried to the next unit. Values outside the normal range of each array are automatically carried to the next unit. For example, month values greater than 12 are carried to years. Month values less than 1 are set to be 1. All other units can wrap and have valid negative values.

`N = datenum(Y, M, D, H, MN, S)` returns the serial date numbers for corresponding elements of the `Y`, `M`, `D`, `H`, `MN`, and `S` (year, month, day, hour, minute, and second) array values. `datenum` does not accept milliseconds in a separate input, but as a fractional part of the seconds (`S`) input. Inputs `Y`, `M`, `D`, `H`, `MN`, and `S` must be arrays of the same size (or any can be a scalar) of type `double`. You can also specify the input arguments as a date vector, `[Y M D H MN S]`.

`N = datenum(S)` converts date string `S` into a serial date number. String `S` must be in one of the date formats 0, 1, 2, 6, 13, 14, 15, 16, or 23, as defined in the reference page for the `datestr` function. MATLAB considers date string years that are specified with only two characters (e.g., `'79'`) to fall within 100 years of the current year. If the format of date string `S` is known, use the syntax `N = datenum(S, F)`.

`N = datenum(S, P)` converts date string `S`, using pivot year `P`. If the format of date string `S` is known, use the syntax `N = datenum(S, F, P)`.

Note The last two calling syntaxes are provided for backward compatibility and are significantly slower than the syntaxes that include a format argument `F`.

Examples

Convert a date string to a serial date number:

```
n = datenum('19-May-2001', 'dd-mmm-yyyy')

n =
      730990
```

Specifying year, month, and day, convert a date to a serial date number:

```
n = datenum(2001, 12, 19)

n =
      731204
```

Convert a date vector to a serial date number:

```
format bank
datenum('March 28, 2005  3:37:07.033 PM')
ans =
     732399.65
```

Convert a date string to a serial date number using the default pivot year:

```
n = datenum('12-jun-17', 'dd-mmm-yy')

n =
       736858
```

Convert the same date string to a serial date number using 1400 as the pivot year:

```
n = datenum('12-jun-17', 'dd-mmm-yy', 1400)

n =
       517712
```

Specify format `'dd.mm.yyyy'` to be used in interpreting a nonstandard date string:

```
n = datenum('19.05.2000', 'dd.mm.yyyy')

n =
       730625
```

See Also

`datestr`, `datevec`, `date`, `clock`, `now`, `datetick`

datestr

Purpose

Convert date and time to string format

Syntax

```
S = datestr(V)
S = datestr(N)
S = datestr(D, F)
S = datestr(S1, F, P)
S = datestr(..., 'local')
```

Description

`datestr` is one of three conversion functions that enable you to express dates and times in any of three formats in MATLAB: a string (or *date string*), a vector of date and time components (or *date vector*), or as a numeric offset from a known date in time (or *serial date number*). Here is an example of a date and time expressed in the three MATLAB formats:

```
Date String:           '24-Oct-2003 12:45:07'
Date Vector:           [2003  10  24  12  45  07]
Serial Date Number:    7.3188e+005
```

A serial date number represents the whole and fractional number of days from 1-Jan-0000 to a specific date. The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time.

`S = datestr(V)` converts one or more date vectors `V` to date strings `S`. Input `V` must be an `m`-by-6 matrix containing `m` full (six-element) date vectors. Each element of `V` must be a positive double-precision number. `datestr` returns a column vector of `m` date strings, where `m` is the total number of date vectors in `V`.

`S = datestr(N)` converts one or more serial date numbers `N` to date strings `S`. Input argument `N` can be a scalar, vector, or multidimensional array of positive double-precision numbers. `datestr` returns a column vector of `m` date strings, where `m` is the total number of date numbers in `N`.

`S = datestr(D, F)` converts one or more date vectors, serial date numbers, or date strings `D` into the same number of date strings `S`.

Input argument F is a format number or string that determines the format of the date string output. Valid values for F are given in the table Standard MATLAB Date Format Definitions on page 2-763, below. Input F may also contain a free-form date format string consisting of format tokens shown in the table Free-Form Date Format Specifiers on page 2-766, below.

Date strings with 2-character years are interpreted to be within the 100 years centered around the current year.

`S = datestr(S1, F, P)` converts date string `S1` to date string `S`, applying format `F` to the output string, and using pivot year `P` as the starting year of the 100-year range in which a two-character year resides. The default pivot year is the current year minus 50 years. All date strings in `S1` must have the same format.

`S = datestr(..., 'local')` returns the string in a localized format. The default is US English ('**en_US**'). This argument must come last in the argument sequence.

Note The vectorized calling syntax can offer significant performance improvement for large arrays.

Standard MATLAB Date Format Definitions

dateform (number)	dateform (string)	Example
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M

Standard MATLAB Date Format Definitions (Continued)

dateform (number)	dateform (string)	Example
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1-01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd,yyyy'	Mar.01,2000
23	'mm/dd/yyyy'	03/01/2000
24	'dd/mm/yyyy'	01/03/2000
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01

Standard MATLAB Date Format Definitions (Continued)

dateform (number)	dateform (string)	Example
27	'QQ-YYYY'	Q1-2001
28	'mmmyyyy'	Mar2000
29 (ISO 8601)	'yyyy-mm-dd'	2000-03-01
30 (ISO 8601)	'yyyymmddTHHMMSS'	20000301T154517
31	'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

Note `dateform` numbers 0, 1, 2, 6, 13, 14, 15, 16, and 23 produce a string suitable for input to `datenum` or `datevec`. Other date string formats do not work with these functions unless you specify a date form in the function call.

Note For date formats that specify only a time (i.e., `dateform` numbers 13, 14, 15, and 16), MATLAB sets the date to January 1 of the current year.

Time formats like `'h:m:s'`, `'h:m:s.s'`, `'h:m pm'`, ... can also be part of the input array `S`. If you do not specify a format string `F`, or if you specify `F` as `-1`, the date string format defaults to the following:

1	If `S` contains date information only, e.g., 01-Mar-1995
16	If `S` contains time information only, e.g., 03:45 PM
0	If `S` is a date vector, or a string that contains both date and time information, e.g., 01-Mar-1995 03:45

The following table shows the string symbols to use in specifying a free-form format for the output date string. MATLAB interprets these symbols according to your computer's language setting and the current MATLAB language setting.

Note You cannot use more than one format specifier for any date or time field. For example, `datestr(n, 'dddd dd mmmm')` specifies two formats for the day of the week, and thus returns an error.

Free-Form Date Format Specifiers

Symbol	Interpretation	Example
`yyyy`	Show year in full.	`1990`, `2002`
`yy`	Show year in two digits.	`90`, `02`
`mmmm`	Show month using full name.	`March`, `December`
`mmm`	Show month using first three letters.	`Mar`, `Dec`
`mm`	Show month in two digits.	`03`, `12`
`m`	Show month using capitalized first letter.	`M`, `D`
`dddd`	Show day using full name.	`Monday`, `Tuesday`
`ddd`	Show day using first three letters.	`Mon`, `Tue`
`dd`	Show day in two digits.	`05`, `20`
`d`	Show day using capitalized first letter.	`M`, `T`

Free-Form Date Format Specifiers (Continued)

Symbol	Interpretation	Example
`HH`	Show hour in two digits (no leading zeros when free-form specifier `AM` or `PM` is used (see last entry in this table)).	`05`, `5 AM`
`MM`	Show minute in two digits.	`12`, `02`
`SS`	Show second in two digits.	`07`, `59`
`FFF`	Show millisecond in three digits.	`.057`
`AM or PM`	Append `AM` or `PM` to date string (see note below).	`3:45:02 PM`

Note Free-form specifiers `AM` and `PM` from the table above are identical. They do not influence which characters are displayed following the time (`AM` versus `PM`), but only whether or not they are displayed. MATLAB selects `AM` or `PM` based on the time entered.

Remarks

A vector of three or six numbers could represent either a single date vector, or a vector of individual serial date numbers. For example, the vector `[2000 12 15 11 45 03]` could represent either 11:45:03 on December 15, 2000 or a vector of date numbers 2000, 12, 15, etc.. MATLAB uses the following general rule in interpreting vectors associated with dates:

- A 3- or 6-element vector having a first element within an approximate range of 500 greater than or less than the current year is considered by MATLAB to be a date vector. Otherwise, it is considered to be a vector of serial date numbers.

To specify dates outside of this range as a date vector, first convert the vector to a serial date number using the `datenum` function as shown here:

```
datestr(datenum([1400 12 15 11 45 03]), ...
        'mmm.dd,yyyy HH:MM:SS')
ans =
   Dec.15,1400 11:45:03
```

Examples

Return the current date and time in a string using the default format, `0`:

```
datestr(now)

ans =
   28-Mar-2005 15:36:23
```

Reformat the date and time, and also show milliseconds:

```
dt = datestr(now, 'mmmm dd, yyyy HH:MM:SS.FFF AM')
dt =
   March 28, 2005  3:37:07.952 PM
```

Format the same showing only the date and in the `mm/dd/yy` format. Note that you can specify this format either by number or by string.

```
datestr(now, 2)     -or-     datestr(now, 'mm/dd/yy')

ans =
   03/28/05
```

Display the returned date string using your own format made up of symbols shown in the Free-Form Date Format Specifiers on page 2-766 table above.

```
datestr(now, 'dd.mm.yyyy')

ans =
   28.03.2005
```

Convert a nonstandard date form into a standard MATLAB date form by first converting to a date number and then to a string:

```
datestr(datenum('28.03.2005', 'dd.mm.yyyy'), 2)

ans =
   03/28/05
```

See Also

datenum, datevec, date, clock, now, datetick

Purpose

Date formatted tick labels

Syntax

```
datetick(tickaxis)
datetick(tickaxis,dateform)
datetick(...,'keeplimits')
datetick(...,'keepticks')
datetick(axes_handle,...)
```

Description

`datetick(tickaxis)` labels the tick lines of an axis using dates, replacing the default numeric labels. `tickaxis` is the string `'x'`, `'y'`, or `'z'`. The default is `'x'`. `datetick` selects a label format based on the minimum and maximum limits of the specified axis.

`datetick(tickaxis,dateform)` formats the labels according to the integer *`dateform`* (see table). To produce correct results, the data for the specified axis must be serial date numbers (as produced by `datenum`).

dateform (number)	*`dateform`* (string)	Example
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00

dateform (number)	*dateform* (string)	Example
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1 01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd.yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd.yyyy'	Mar.01.2000
23	'mm/dd/yyyy'	03/01/2000
24	'dd/mm/yyyy'	01/03/2000
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01
27	'QQ-YYYY'	Q1-2001
28	'mmmyyyy'	Mar2000

`datetick(...,'keeplimits')` changes the tick labels to date-based labels while preserving the axis limits.

`datetick(...,'keepticks')` changes the tick labels to date-based labels without changing their locations.

You can use both `keeplimits` and `keepticks` in the same call to `datetick`.

`datetick(axes_handle,...)` uses the axes specified by the handle `ax` instead of the current axes.

Remarks

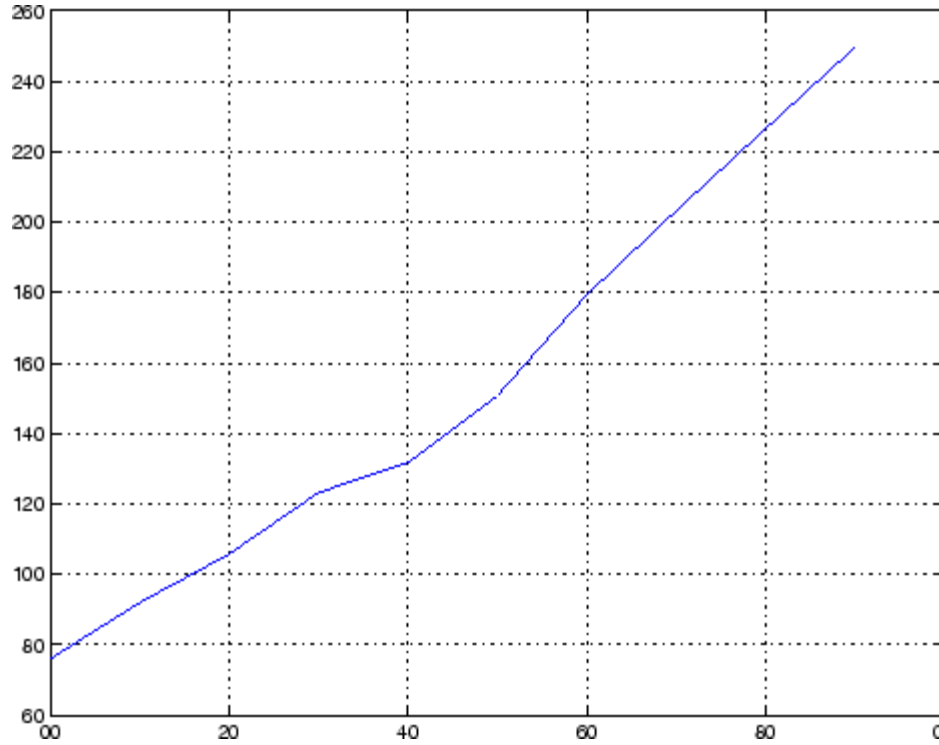
`datetick` calls `datestr` to convert date numbers to date strings.

To change the tick spacing and locations, set the appropriate axes property (i.e., `XTick`, `YTick`, or `ZTick`) before calling `datetick`.

Example

Consider graphing population data based on the 1990 U.S. census:

```
t = (1900:10:1990)';  % Time interval
p = [75.995 91.972 105.711 123.203 131.669 ...
 150.697 179.323 203.212 226.505 249.633]';  % Population
plot(datenum(t,1,1),p) % Convert years to date numbers and plot
grid on
datetick('x',11)   % Replace x-axis ticks with 2-digit year
labels
```



See Also

The axes properties `XTick`, `YTick`, and `ZTick`

`datenum`, `datestr`

"Annotating Plots" on page 1-87 for related functions

Purpose

Convert date and time to vector of components

Syntax

```
V = datevec(N)
V = datevec(S, F)
V = datevec(S, F, P)
V = datevec(S, P, F)
[Y, M, D, H, MN, S] = datevec(...)
V = datevec(S)
V = datevec(S, P)
```

Description

`datevec` is one of three conversion functions that enable you to express dates and times in any of three formats in MATLAB: a string (or *date string*), a vector of date and time components (or *date vector*), or as a numeric offset from a known date in time (or *serial date number*). Here is an example of a date and time expressed in the three MATLAB formats:

```
Date String:           '24-Oct-2003 12:45:07'
Date Vector:           [2003  10  24  12  45  07]
Serial Date Number:    7.3188e+005
```

A serial date number represents the whole and fractional number of days from 1-Jan-0000 to a specific date. The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time.

`V = datevec(N)` converts one or more date numbers `N` to date vectors `V`. Input argument `N` can be a scalar, vector, or multidimensional array of positive date numbers. `datevec` returns an `m`-by-6 matrix containing `m` date vectors, where `m` is the total number of date numbers in `N`.

`V = datevec(S, F)` converts one or more date strings `S` to date vectors `V` using format string `F` to interpret the date strings in `S`. Input argument `S` can be a cell array of strings or a character array where each row corresponds to one date string. All of the date strings in `S` must have the same format which must be composed of date format symbols according to the table “Free-Form Date Format Specifiers” in the `datestr` help.

Formats with 'Q' are not accepted by datevec. datevec returns an m-by-6 matrix of date vectors, where m is the number of date strings in S.

Certain formats may not contain enough information to compute a date vector. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. Date strings with two character years are interpreted to be within the 100 years centered around the current year.

V = datevec(S, F, P) converts the date string S to a date vector V using date format F and pivot year P. The pivot year is the starting year of the 100-year range in which a two-character year resides. The default pivot year is the current year minus 50 years.

V = datevec(S, P, F) is the same as the syntax shown above, except the order of the last two arguments are switched.

[Y, M, D, H, MN, S] = datevec(...) takes any of the two syntaxes shown above and returns the components of the date vector as individual variables. datevec does not return milliseconds in a separate output, but as a fractional part of the seconds (S) output.

V = datevec(S) converts date string S to date vector V. Input argument S must be in one of the date formats 0, 1, 2, 6, 13, 14, 15, 16, or 23 as defined in the reference page for the datestr function. This calling syntax is provided for backward compatibility, and is significantly slower than the syntax which specifies the format string. If the format is known, the V = datevec(S, F) syntax is recommended.

V = datevec(S, P) converts the date string S using pivot year P. If the format is known, the V = datevec(S, F, P) or V = datevec(S, P, F) syntax should be used.

Note If more than one input argument is used, the first argument must be a date string or array of date strings.

When creating your own date vector, you need not make the components integers. Any components that lie outside their conventional ranges

affect the next higher component (so that, for instance, the anomalous June 31 becomes July 1). A zeroth month, with zero days, is allowed.

Note The vectorized calling syntax can offer significant performance improvement for large arrays.

Examples

Obtain a date vector using a string as input:

```
format short g

datevec('March 28, 2005  3:37:07.952 PM')
ans =
     2005       3       28       15       37       7.952
```

Obtain a date vector using a serial date number as input:

```
t = datenum('March 28, 2005  3:37:07.952 PM')
t =
     7.324e+005

datevec(t)
ans =
     2005       3       28       15       37       7.952
```

Assign elements of the returned date vector:

```
[y, m, d, h, mn, s] = datevec('March 28, 2005  3:37:07.952 PM');
sprintf('Date: %d/%d/%d   Time: %d:%d:%2.3f\n', m, d, y, h, mn, s)

ans =
   Date: 3/28/2005   Time: 15:37:7.952
```

Use free-form date format `'dd.mm.yyyy'` to indicate how you want a nonstandard date string interpreted:

```
datevec('28.03.2005', 'dd.mm.yyyy')

ans = 2005    3    28    0   0   0
```

See Also

`datenum`, `datestr`, `date`, `clock`, `now`, `datetick`

dbclear

Purpose

Clear breakpoints

GUI Alternatives

In the Editor/Debugger, click  to clear a breakpoint, or  to clear all breakpoints. For details, see “Disabling and Clearing Breakpoints”.

Syntax

```
dbclear all
dbclear in mfile ...
dbclear if error ...
dbclear if warning ...
dbclear if naninf
dbclear if infnan
```

Description

`dbclear all` removes all breakpoints in all M-files, as well as breakpoints set for errors, caught errors, caught error identifiers, warnings, warning identifiers, and `naninf`/`infnan`.

`dbclear in mfile ...` formats are listed here:

Format	Action
`dbclear in mfile`	Removes all breakpoints in `mfile`.
`dbclear in mfile at lineno`	Removes the breakpoint set at line number `lineno` in `mfile`.
`dbclear in mfile at lineno@`	Removes the breakpoint set in the anonymous function at line number `lineno` in `mfile`.
`dbclear in mfile at lineno@n`	Removes the breakpoint set in the `n`the anonymous function at line number `lineno` in `mfile`.
`dbclear in mfile at subfun`	Removes all breakpoints in subfunction `subfun` in `mfile`.

`dbclear if error ...` formats are listed here:

Format	Action
`dbclear if error`	Removes the breakpoints set using the `dbstop if error` and `dbstop if error identifier` statements.
`dbclear if error identifier`	Removes the breakpoint set using `dbstop if error identifier` for the specified `identifier`. Running this produces an error if `dbstop if error` or `dbstop if error all` is set.
`dbclear if caught error`	Removes the breakpoints set using the `dbstop if caught error` and `dbstop if caught error identifier` statements.
`dbclear if caught error identifier`	Removes the breakpoints set using the `dbstop if caught error identifier` statement for the specified identifier. Running this produces an error if `dbstop if caught error` or `dbstop if caught error all` is set.

`dbclear if warning ...` formats are listed here:

`dbclear if warning`	Removes the breakpoints set using the `dbstop if warning` and `dbstop if warning identifier` statements.
`dbclear if warning identifier`	Removes the breakpoint set using `dbstop if warning identifier` for the specified `identifier`. Running this produces an error if `dbstop if warning` or `dbstop if warning all` is set.

`dbclear if naninf` removes the breakpoint set by `dbstop if naninf` or `dbstop if infnan`.

`dbclear if infnan` removes the breakpoint set by `dbstop if infnan` or `dbstop if naninf`.

Remarks

The **at** and **in** keywords are optional.

In the syntax, `mfile` can be an M-file, or the path to a function within a file. For example

```
dbclear in foo>myfun
```

clears the breakpoint at the `myfun` function in the file `foo.m`.

See Also

`dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`, `partialpath`

Purpose

Resume execution

GUI Alternatives

Select **Debug > Continue** from most desktop tools, or in the Editor/Debugger, click  .

Syntax

```
dbcont
```

Description

`dbcont` resumes execution of an M-file from a breakpoint. Execution continues until another breakpoint is encountered, a pause condition is met, an error occurs, or MATLAB returns to the base workspace prompt.

Note If you want to edit an M-file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the M-file. If you edit an M-file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

See Also

`dbclear`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`

dbdown

Purpose

Change local workspace context when in debug mode

GUI Alternatives

Use the **Stack** field 

in the Editor/Debugger or Workspace browser.

Syntax

```
dbdown
```

Description

`dbdown` changes the current workspace context to the workspace of the called M-file when a breakpoint is encountered. You must have issued the `dbup` function at least once before you issue this function. `dbdown` is the opposite of `dbup`.

Multiple `dbdown` functions change the workspace context to each successively executed M-file on the stack until the current workspace context is the current breakpoint. It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.

See Also

`dbclear`, `dbcont`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`

Purpose

Numerically evaluate double integral

Syntax

```
q = dblquad(fun,xmin,xmax,ymin,ymax)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)
```

Description

`q = dblquad(fun,xmin,xmax,ymin,ymax)` calls the `quad` function to evaluate the double integral `fun(x,y)` over the rectangle `xmin <= x <= xmax`, `ymin <= y <= ymax`. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. `fun(x,y)` must accept a vector `x` and a scalar `y` and return a vector of values of the integrand.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Example

Pass M-file function handle `@integrnd` to `dblquad`:

```
Q = dblquad(@integrnd,pi,2*pi,0,pi);
```

where the M-file `integrnd.m` is

```
function z = integrnd(x, y)
z = y*sin(x)+x*cos(y);
```

Pass anonymous function handle `F` to `dblquad`:

```
F = @(x,y)y*sin(x)+x*cos(y);
Q = dblquad(F,pi,2*pi,0,pi);
```

The `integrnd` function integrates `y*sin(x)+x*cos(y)` over the square `pi <= x <= 2*pi`, `0 <= y <= pi`. Note that the integrand can be evaluated with a vector `x` and a scalar `y`.

Nonsquare regions can be handled by setting the integrand to zero outside of the region. For example, the volume of a hemisphere is

```
dblquad(@(x,y)sqrt(max(1-(x.^2+y.^2),0)), -1, 1, -1, 1)
```

or

```
dblquad(@(x,y)sqrt(1-(x.^2+y.^2)).*(x.^2+y.^2<=1), -1, 1, -1, 1)
```

See Also

`quad`, `quadgk`, `quadl`, `triplequad`, `function_handle` (`@`), "Anonymous Functions"

Purpose

Enable MEX-file debugging

Syntax

```
dbmex on
dbmex off
dbmex stop
```

Description

`dbmex on` enables MEX-file debugging for UNIX platforms. It is not supported on the Sun Solaris platform. To use this option, first start MATLAB from within a debugger by typing `matlab -Ddebugger`, where `debugger` is the name of the debugger.

`dbmex off` disables MEX-file debugging.

`dbmex stop` returns to the debugger prompt.

Remarks

On Sun Solaris platforms, `dbmex` is not supported.
See the Technical Support solution 1-17Z0R at `http://www.mathworks.com/support/solutions/data/1-17Z0R.html` for an alternative method of debugging.

See Also

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`

dbquit

Purpose

Quit debug mode

GUI Alternative

From most desktop tools, select **Debug > Exit Debug Mode**, or in the Editor/Debugger, click .

Syntax

```
dbquit
dbquit('all')
dbquit all
```

Description

`dbquit` terminates debug mode. The Command Window then displays the standard prompt (`>>`). The M-file being processed is *not* completed and no results are returned. All breakpoints remain in effect. As an alternative to `dbquit`, press **Shift+F5**.

If you debug `file1` and step into `file2`, running `dbquit` terminates debugging for both files. However, if you debug `file3` and also debug `file4`, running `dbquit` terminates debugging for `file4`, but `file3` remains in debug mode until you run `dbquit` again.

`dbquit('all')` or the command form, `dbquit all`, ends debugging for all files at once.

Examples

This example illustrates the use of `dbquit` relative to `dbquit('all')`. Set breakpoints in and run `file1` and `file2`:

```
>> dbstop in file1
>> dbstop in file2
>> file1
K>> file2
K>> dbstack
```

MATLAB returns

```
K>> dbstack
  In file1 at 11
  In file2 at 22
```

If you use the `dbquit` syntax

```
K>> dbquit
```

MATLAB ends debugging for `file2` but `file1` is still in debug mode as shown here

```
K>> dbstack
    in file1 at 11
```

Run `dbquit` again to exit debug mode for `file1`.

Alternatively, `dbquit('all')` ends debugging for both files at once:

```
K>> dbstack
  In file1 at 11
  In file2 at 22
dbquit('all')
dbstack
```

returns no result.

See Also

`dbclear`, `dbcont`, `dbdown`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`

dbstack

Purpose

Function call stack

GUI Alternatives

Use the **Stack** field 

in the Editor/Debugger or Workspace browser.

Syntax

```
dbstack
dbstack(n)
dbstack('-completenames')
[ST,I] = dbstack
```

Description

`dbstack` displays the line numbers and M-file names of the function calls that led to the current breakpoint, listed in the order in which they were executed. The line number of the most recently executed function call (at which the current breakpoint occurred) is listed first, followed by its calling function, which is followed by its calling function, and so on, until the topmost M-file function is reached. Each line number is a hyperlink you can click to go directly to that line in the Editor/Debugger. The notation `functionname>subfunctionname` is used to describe the subfunction location.

`dbstack(n)` omits from the display the first `n` frames. This is useful when issuing a `dbstack` from within, say, an error handler.

`dbstack('-completenames')` outputs the "complete name" (the absolute file name and the entire sequence of functions that nests the function in the stack frame) of each function in the stack.

Either none, one, or both `n` and `'-completenames'` can appear. If both appear, the order is irrelevant.

`[ST,I] = dbstack` returns the stack trace information in an `m`-by-`1` structure `ST` with the fields

`file`	The file in which the function appears. This field will be the empty string if there is no file.
`name`	Function name within the file.
`line`	Function line number.

The current workspace index is returned in `I`.

If you step past the end of an M-file, then `dbstack` returns a negative line number value to identify that special case. For example, if the last line to be executed is line `15`, then the `dbstack` line number is `15` before you execute that line and `-15` afterwards.

Examples

```
dbstack

In /usr/local/matlab/toolbox/matlab/cond.m at line 13
In test1.m at line 2
In test.m at line 3
```

See Also

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`, `evalin`, `mfilename`, `whos`

MATLAB Desktop Tools and Development Environment Documentation

- “Editing and Debugging M-Files”
- “Examining Values”

dbstatus

Purpose

List all breakpoints

GUI Alternative

Breakpoint line numbers are displayed graphically via the breakpoint icons when the file is open in the Editor/Debugger.

Syntax

```
dbstatus
dbstatus mfile
dbstatus('-completenames')
s = dbstatus(...)
```

Description

`dbstatus` lists all the breakpoints in effect including errors, caught errors, warnings, and `naninfs`.

`dbstatus mfile` displays a list of the line numbers for which breakpoints are set in the specified M-file, where `mfile` is an M-file function name or a MATLAB relative partial pathname. Each line number is a hyperlink you can click to go directly to that line in the Editor/Debugger.

`dbstatus('-completenames')` displays, for each breakpoint, the absolute filename and the sequence of functions that nest the function containing the breakpoint.

`s = dbstatus(...)` returns breakpoint information in an `m`-by-`1` structure with the fields listed in the following table. Use this syntax to save breakpoint status and restore it at a later time using `dbstop(s)`—see `dbstop` for an example.

`name`	Function name.
`file`	Full pathname for file containing breakpoints.
`line`	Vector of breakpoint line numbers.
`anonymous`	Vector of integers representing the anonymous functions in the `line` field. For example, `2` means the second anonymous function in that line. A value of `0` means the breakpoint is at the start of the line, not in an anonymous function.

`expression`	Cell vector of breakpoint conditional expression strings corresponding to lines in the `line` field.
`cond`	Condition string (`'error'`, `'caught error'`, `'warning'`, or `'naninf'`).
`identifier`	When `cond` is `'error'`, `'caught error'`, or `'warning'`, a cell vector of MATLAB message identifier strings for which the particular `cond` state is set.

Use `dbstatus class/function`, `dbstatus private/function`, or `dbstatus class/private/function` to determine the status for methods, private functions, or private methods (for a class named `class`).

In all forms you can further qualify the function name with a subfunction name, as in `dbstatus function>subfunction`.

Remarks

In the syntax, `mfile` can be an M-file, or the path to a function within a file. For example

```
Breakpoint for foo>mfun is on line 9
```

means there is a breakpoint at the `myfun` subfunction, which is line 9 in the file `foo.m`.

See Also

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstep`, `dbstop`, `dbtype`, `dbup`, `error`, `partialpath`, `warning`

dbstep

Purpose

Execute one or more lines from current breakpoint

GUI Alternatives

As an alternative to `dbstep`, you can select **Debug > Step** or **Step In** in most desktop tools, or click the Step or Step In buttons on the Editor/Debugger toolbar.

Syntax

```
dbstep
dbstep nlines
dbstep in
dbstep out
```

Description

This function allows you to debug an M-file by following its execution from the current breakpoint. At a breakpoint, the `dbstep` function steps through execution of the current M-file one line at a time or at the rate specified by `nlines`.

`dbstep` executes the next executable line of the current M-file. `dbstep` steps over the current line, skipping any breakpoints set in functions called by that line.

`dbstep nlines` executes the specified number of executable lines.

`dbstep` **`in`** steps to the next executable line. If that line contains a call to another M-file function, execution will step to the first executable line of the called M-file function. If there is no call to an M-file on that line, `dbstep in` is the same as `dbstep`.

`dbstep` **`out`** runs the rest of the function and stops just after leaving the function.

For all forms, MATLAB also stops execution at any breakpoint it encounters.

Note If you want to edit an M-file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the M-file. If you edit an M-file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

See Also

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstop`, `dbtype`, `dbup`

dbstop

Purpose

Set breakpoints

GUI Alternative

Use the **Debug** menu in most desktop tools, or the context menu in Editor/Debugger. See details.

Syntax

```
dbstop in mfile ...
dbstop in nonmfile
dbstop if error ...
dbstop if warning ...
dbstop if naninf
dbstop if infnan
dbstop(s)
```

Description

`dbstop in mfile ...` formats are listed here:

Format	Action	Additional Information
`dbstop in mfile`	Temporarily stops execution of running `mfile` at the first executable line, putting MATLAB in debug mode. `mfile` must be in a directory that is on the search path, or in the current directory. `mfile` can be an M-file, or the path to a function (`subfun`) within the file, using the notation `mfile > subfun`. The **`in`** keyword is optional.	If you have graphical debugging enabled, the MATLAB Debugger opens with a breakpoint at the first executable line of `mfile`. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use `dbcont` or `dbstep` to resume execution of `mfile`. Use `dbquit` to exit from debug mode.

Format	Action	Additional Information
`dbstop in mfile at lineno`	Temporarily stops execution of running `mfile` just prior to execution of the line whose number is `lineno`, putting MATLAB in debug mode. If that line is not executable, execution stops and the breakpoint is set at the next executable line following `lineno`. `mfile` must be in a directory that is on the search path, or in the current directory. The **`at`** keyword is optional.	If you have graphical debugging enabled, MATLAB opens `mfile` with a breakpoint at line `lineno`. When execution stops, you can use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use `dbcont` or `dbstep` to resume execution of `mfile`. Use `dbquit` to exit from debug mode
`dbstop in mfile at lineno@`	Stops just after any call to the first anonymous function in the specified line number in `mfile`.	
`dbstop in mfile at lineno@n`	Stops just after any call to the `n`the anonymous function in the specified line number in `mfile`.	
`dbstop in mfile at subfun`	Temporarily stops execution of running `mfile` just prior to execution of the subfunction `subfun`, putting MATLAB in debug mode. `mfile` must be in a directory that is on the search path, or in the current directory.	If you have graphical debugging enabled, MATLAB opens `mfile` with a breakpoint at the subfunction `subfun`. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use `dbcont` or `dbstep` to resume execution of `mfile`. Use `dbquit` to exit from debug mode.

Format	Action	Additional Information
`dbstop in mfile at lineno if expression`	Temporarily stops execution of running `mfile`, just prior to execution of the line whose number is `lineno`, putting MATLAB in debug mode. Execution stops only if expression evaluates to `true`. `expression` is evaluated (as if by `eval`), in `mfile`'s workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (`1` or `0` for true or false). If that line is not executable, execution stops and the breakpoint is set at the next executable line following `lineno`. `mfile` must be in a directory that is on the search path, or in the current directory.	If you have graphical debugging enabled, MATLAB opens `mfile` with a breakpoint at line `lineno`. When execution stops, you can use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use `dbcont` or `dbstep` to resume execution of `mfile`. Use `dbquit` to exit from debug mode.
`dbstop in mfile at lineno@ if expression`	Stops just after any call to the first anonymous function in the specified line number in `mfile` if `expression` evaluates to logical `1` (true).	
`dbstop in mfile at lineno@n if expression`	Stops just after any call to the `n`the anonymous function in the specified line number in `mfile` if `expression` evaluates to logical `1` (true).	

Format	Action	Additional Information
`dbstop in mfile if expression`	Temporarily stops execution of running `mfile`, at the first executable line, putting MATLAB in debug mode. Execution stops only if `expression` evaluates to logical `1` (true). `expression` is evaluated (as if by `eval`), in `mfile`'s workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (`0` or `1` for true or false). `mfile` must be in a directory on the search path, or in the current directory	If you have graphical debugging enabled, MATLAB opens `mfile` with a breakpoint at the first executable line of `mfile`. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use `dbcont` or `dbstep` to resume execution of `mfile`. Use `dbquit` to exit from debug mode.
`dbstop in mfile at subfun if expression`	Temporarily stops execution of running `mfile`, just prior to execution of the subfunction `subfun`, putting MATLAB in debug mode. Execution stops only if `expression` evaluates to logical `1` (true). `expression` is evaluated (as if by `eval`), in `mfile`'s workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (`0` or `1` for true or false). `mfile` must be in a directory on the search path, or in the current directory	If you have graphical debugging enabled, MATLAB opens `mfile` with a breakpoint at the subfunction specified by `subfun`. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use `dbcont` or `dbstep` to resume execution of `mfile`. Use `dbquit` to exit from debug mode.

`dbstop in nonmfile` temporarily stops execution of the running M-file at the point where `nonmfile` is called. This puts MATLAB in debug mode, where `nonmfile` is, for example, a built-in or MDL-file. MATLAB issues a warning because it cannot actually stop *in* the file;

rather MATLAB stops prior to the file's execution. Once stopped, you can examine values and code around that point in the execution. Use `dbstop in nonmfile` with caution because the debugger stops in M-files it uses for running and debugging if they contain `nonmfile`. As a result, some debugging features do not operate as expected, such as typing `help functionname` at the `K>>` prompt.

`dbstop` **`if error ...`** formats are listed here:

Format	Action
`dbstop` **`if error`**	Stops execution when any M-file you subsequently run produces a run-time error, putting MATLAB in debug mode, paused at the line that generated the error. The errors that stop execution do not include run-time errors that are detected within a `try...catch` block. You cannot resume execution after an uncaught run-time error. Use `dbquit` to exit from debug mode.
`dbstop` **`if error`** `identifier`	Stops execution when any M-file you subsequently run produces a run-time error whose message identifier is `identifier`, putting MATLAB in debug mode, paused at the line that generated the error. The errors that stop execution do not include run-time errors that are detected within a `try...catch` block. You cannot resume execution after an uncaught run-time error. Use `dbquit` to exit from debug mode.
`dbstop` **`if caught error`**	Stops execution when any M-file you subsequently run produces a run-time error, putting MATLAB in debug mode, paused at the line in the `try` portion of the block that generated the error. The errors that stop execution are those detected within a `try...catch` block.
`dbstop` **`if caught error`** `identifier`	Stops execution when any M-file you subsequently run produces a run-time error whose message identifier is `identifier`, putting MATLAB in debug mode, paused at the line in the `try` portion of the block that generated the error. The errors that stop execution are those detected within a `try...catch` block.

`dbstop` **`if warning ...`** formats are listed here:

Format	Action
`dbstop if warning`	Stops execution when any M-file you subsequently run produces a run-time warning, putting MATLAB in debug mode, paused at the line that generated the warning. Use `dbcont` or `dbstep` to resume execution.
`dbstop if warning identifier`	Stops execution when any M-file you subsequently run produces a runtime warning whose message identifier is `identifier`, putting MATLAB in debug mode, paused at the line that generated the warning. Use `dbcont` or `dbstep` to resume execution.

`dbstop if naninf` or `dbstop if infnan` stops execution when any M-file you subsequently run produces an infinite value (`Inf`) or a value that is not a number (`NaN`) as a result of an operator, function call, or scalar assignment, putting MATLAB in debug mode, paused immediately after the line where `Inf` or `NaN` was encountered. For convenience, you can use either **`naninf`** or **`infnan`**—they perform in exactly the same manner. Use `dbcont` or `dbstep` to resume execution. Use `dbquit` to exit from debug mode.

`dbstop(s)` restores breakpoints previously saved to the structure `s` using `s=dbstatus`. The files for which the breakpoints have been saved need to be on the search path or in the current directory. In addition, because the breakpoints are assigned by line number, the lines in the file need to be the same as when the breakpoints were saved, or the results are unpredictable. See the example “Restore Saved Breakpoints” on page 2-802 and `dbstatus` for more information.

Remarks

Note that MATLAB could become nonresponsive if it stops at a breakpoint while displaying a modal dialog box or figure that your M-file creates. In that event, use **Ctrl+C** to go the MATLAB prompt.

To open the M-file in the Editor/Debugger when execution reaches a breakpoint, select **Debug > Open M-Files When Debugging**.

To stop at each pass through a for loop, do not set the breakpoint at the for statement. For example, in

```
for n = 1:10
    m = n+1;
end
```

MATLAB executes the `for` statement only once, which is efficient. Therefore, when you set a breakpoint at the `for` statement and step through the file, you only stop at the `for` statement once. Instead place the breakpoint at the next line, `m=n+1` to stop at each pass through the loop.

Examples

The file `buggy`, used in these examples, consists of three lines.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Stop at First Executable Line

The statements

```
dbstop in buggy
buggy(2:5)
```

stop execution at the first executable line in `buggy`:

```
n = length(x);
```

The function

```
dbstep
```

advances to the next line, at which point you can examine the value of `n`.

Stop if Error

Because `buggy` only works on vectors, it produces an error if the input `x` is a full matrix. The statements

```
dbstop if error
buggy(magic(3))
```

produce

```
??? Error using ==> ./
Matrix dimensions must agree.
Error in ==> c:\buggy.m
On line 3 ==> z = (1:n)./x;
K>>
```

and put MATLAB in debug mode.

Stop if InfNaN

In `buggy`, if any of the elements of the input `x` is zero, a division by zero occurs. The statements

```
dbstop if naninf
buggy(0:2)
```

produce

```
Warning: Divide by zero.
> In c:\buggy.m at line 3
K>>
```

and put MATLAB in debug mode.

Stop at Function in File

In this example, MATLAB stops at the `newTemp` function in the M-file `yearlyAvgs`:

```
dbstop in yearlyAvgs>newTemp
```

Stop at Non M-File

In this example, MATLAB stops at the built-in function `clear` when you run `myfile.m`.

```
dbstop in clear; myfile
```

MATLAB issues a warning, but permits the stop:

```
Warning: MATLAB debugger can only stop in M-files, and
"m_interpreter>clear" is not an M-file.
Instead, the debugger will stop at the point right before
"m_interpreter>clear" is called.
```

Execution stops in `myfile` at the point where the `clear` function is called.

Restore Saved Breakpoints

1. Set breakpoints in `myfile` as follows:

   ```
   dbstop at 12 in myfile
   dbstop if error
   ```

2. Running `dbstatus` shows

   ```
   Breakpoint for myfile is on line 12.
   Stop if error.
   ```

3. Save the breakpoints to the structure `s`, and then save `s` to the MAT-file `myfilebrkpnts`.

   ```
   s = dbstatus
   save myfilebrkpnts s
   ```

   Use `s=dbstatus('completenames')` to save absolute pathnames and the breakpoint function nesting sequence.

4. At this point, you can end the debugging session and clear all breakpoints, or even end the MATLAB session.

   When you want to restore the breakpoints, be sure all of the files containing the breakpoints are on the search path or in the current directory. Then load the MAT-file, which adds `s` to the workspace, and restore the breakpoints as follows:

   ```
   load myfilebrkpnts
   dbstop(s)
   ```

5 Verify the breakpoints by running `dbstatus`, which shows

```
dbstop at 12 in myfile
dbstop if error
```

If you made changes to `myfile` after saving the breakpoints, the results from restoring the breakpoints are not predictable. For example, if you added a new line prior to line 12 in `myfile`, the breakpoint will now be set at the new line 12.

See Also

`assignin`, `break`, `dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbtype`, `dbup`, `evalin`, `keyboard`, `partialpath`, `return`, `whos`

dbtype

Purpose

List M-file with line numbers

GUI Alternatives

As an alternative to the `dbtype` function, you can see an M-file with line numbers by opening it in the Editor/Debugger.

Syntax

```
dbtype mfilename
dbtype mfilename start:end
```

Description

The `dbtype` command is used to list an M-file with line numbers, which is helpful when setting breakpoints with `dbstop`.

`dbtype mfilename` displays the contents of the specified M-file, with the line number preceding each line. mfilename must be the full pathname of an M-file, or a MATLAB relative `partial pathname`.

`dbtype mfilename start:end` displays the portion of the M-file specified by a range of line numbers from `start` to `end`.

You cannot use `dbtype` for built-in functions.

Examples

To see only the input and output arguments for a function, that is, the first line of the M-file, use the syntax

```
dbtype mfilename 1
```

For example,

```
dbtype fileparts 1
```

returns

```
1     function [path, fname, extension,version] = fileparts(name)
```

See Also

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbup`, `partialpath`

Purpose

Change local workspace context

GUI Alternatives

As an alternative to the `dbup` function, you can select a different workspace from the **Stack** field in the Editor/Debugger toolbar.

Syntax

```
dbup
```

Description

This function allows you to examine the calling M-file to determine what led to the arguments' being passed to the called function.

`dbup` changes the current workspace context, while the user is in the debug mode, to the workspace of the calling M-file.

Multiple `dbup` functions change the workspace context to each previous calling M-file on the stack until the base workspace context is reached. (It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.)

See Also

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`

dde23

Purpose

Solve delay differential equations (DDEs) with constant delays

Syntax

```
sol = dde23(ddefun,lags,history,tspan)
sol = dde23(ddefun,lags,history,tspan,options)
```

Arguments

`ddefun`

Function handle that evaluates the right side of the differential equations $y'(t) = f(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$. The function must have the form

```
dydt = ddefun(t,y,Z)
```

where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(t-\tau_j)$ for delay τ_j = `lags(j)`. The output is a column vector corresponding to $f(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$.

`lags`

Vector of constant, positive delays $\tau_1, \ldots, \tau_k$.

`history`

Specify `history` in one of three ways:

- A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector
- A constant column vector, if $y(t)$ is constant
- The solution `sol` from a previous integration, if this call continues that integration

`tspan`	Interval of integration as a vector `[t0,tf]` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = dde23(ddefun,lags,history,tspan)` integrates the system of DDEs

$$y'(t) = f(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$$

on the interval $[t_0, t_f]$, where $\tau_1, \ldots, \tau_k$ are constant, positive delays and $t_0 < t_f$. `ddefun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `ddefun`, if necessary.

`dde23` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `dde23` has the following fields.

`sol.x`	Mesh selected by `dde23`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'dde23'`

`sol = dde23(ddefun,lags,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and "Initial Value Problems for DDEs" in the MATLAB documentation for details.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Jumps'` option to solve problems with discontinuities in the history or solution. Set this option to a vector that contains the locations of discontinuities in the solution prior to `t0` (the history) or in coefficients of the equations at known values of t after `t0`.

Use the `'Events'` option to specify a function that `dde23` calls to find where functions $g(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the `k`th event function in `events`:

- `value(k)` is the value of the `k`th event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and `0` otherwise.
- `direction(k) = 0` if you want `dde23` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the `'Events'` option and events are detected, the output structure `sol` also includes fields:

`sol.xe`	Row vector of locations of all events, i.e., times when an event function vanished
`sol.ye`	Matrix whose columns are the solution values corresponding to times in `sol.xe`
`sol.ie`	Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

Examples

This example solves a DDE on the interval [0, 5] with lags 1 and 0.2. The function `ddex1de` computes the delay differential equations, and `ddex1hist` computes the history for `t <= 0`.

Note The demo `ddex1` contains the complete code for this example. To see the code in an editor, click the example name, or type `edit ddex1` at the command line. To run the example type `ddex1` at the command line.

```
sol = dde23(@ddex1de,[1, 0.2],@ddex1hist,[0, 5]);
```

This code evaluates the solution at 100 equally spaced points in the interval `[0,5]`, then plots the result.

```
tint = linspace(0,5);
yint = deval(sol,tint);
plot(tint,yint);
```

`ddex1` shows how you can code this problem using subfunctions. For more examples see `ddex2`.

Algorithm

`dde23` tracks discontinuities and integrates with the explicit Runge-Kutta (2,3) pair and interpolant of `ode23`. It uses iteration to take steps longer than the lags.

See Also

`ddesd`, `ddeget`, `ddeset`, `deval`, `function_handle (@)`

References

[1] Shampine, L.F. and S. Thompson, "Solving DDEs in MATLAB, "*Applied Numerical Mathematics*, Vol. 37, 2001, pp. 441-458.

[2] Kierzenka, J., L.F. Shampine, and S. Thompson, "Solving Delay Differential Equations with DDE23," available at `www.mathworks.com/dde_tutorial`.

Purpose

Set up advisory link

Syntax

```
rc = ddeadv(channel,item,callback,upmtx,format,timeout)
```

Note Use COM, as described in COM Support in MATLAB. The ddeadv function will be removed in a future version of MATLAB.

Description

`rc = ddeadv(channel,item,callback,upmtx,format,timeout)` sets up an advisory link between MATLAB and a server application. When the data identified by the `item` argument changes, the string specified by the `callback` argument is passed to the `eval` function and evaluated. If the advisory link is a hot link, DDE modifies `upmtx`, the update matrix, to reflect the data in `item`.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddeadv` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.
`callback`	String specifying the callback that is evaluated on update notification. Changing the data identified by `item` at the server causes `callback` to get passed to the `eval` function to be evaluated.

upmtx (*optional*)	String specifying the name of a matrix that holds data sent with an update notification. If `upmtx` is included, changing `item` at the server causes `upmtx` to be updated with the revised data. Specifying `upmtx` creates a hot link. Omitting `upmtx` or specifying it as an empty string creates a warm link. If `upmtx` exists in the workspace, its contents are overwritten. If `upmtx` does not exist, it is created.
format (*optional*)	Two-element array specifying the format of the data to be sent on update. The first element specifies the Windows clipboard format to use for the data. The only currently supported format is `cf_text`, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are `numeric` (the default, which corresponds to a value of `0`) and `string` (which corresponds to a value of `1`). The default format array is `[1 0]`.
timeout (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). If advisory link is not established within `timeout` milliseconds, the function fails. The default value of `timeout` is three seconds.

Examples

Set up a hot link between a range of cells in Excel (Row 1, Column 1 through Row 5, Column 5) and the matrix `x`. If successful, display the matrix:

```
rc = ddeadv(channel, 'r1c1:r5c5', 'disp(x)', 'x');
```

Communication with Excel must have been established previously with a `ddeinit` command.

See Also

ddeexec, ddeinit, ddepoke, ddereq, ddeterm, ddeunadv

ddeexec

Purpose

Send string for execution

Syntax

```
rc = ddeexec(channel,command,item,timeout)
```

Note Use COM, as described in COM Support in MATLAB. The ddeexec function will be removed in a future version of MATLAB.

Description

`rc = ddeexec(channel,command,item,timeout)` sends a string for execution to another application via an established DDE conversation. Specify the string as the `command` argument.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddeexec` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`command`	String specifying the command to be executed.
`item` (*optional*)	String specifying the DDE item name for execution. This argument is not used for many applications. If your application requires this argument, it provides additional information for `command`. Consult your server documentation for more information.
timeout (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Given the channel assigned to a conversation, send a command to Excel:

```
rc = ddeexec(channel,'[formula.goto("r1c1")]')
```

Communication with Excel must have been established previously with a `ddeinit` command.

See Also

`ddeadv`, `ddeinit`, `ddepoke`, `ddereq`, `ddeterm`, `ddeunadv`

ddeget

Purpose

Extract properties from delay differential equations options structure

Syntax

```
val = ddeget(options,'name')
val = ddeget(options,'name',default)
```

Description

`val = ddeget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = ddeget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = ddeget(opts,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `opts`.

See Also

`dde23`, `ddesd`, `ddeset`

Purpose

Initiate Dynamic Data Exchange (DDE) conversation

Syntax

Note Use COM, as described in COM Support in MATLAB. The ddeinit function will be removed in a future version of MATLAB.

```
channel = ddeinit('service','topic')
```

Description

`channel = ddeinit('service','topic')` returns a channel handle assigned to the conversation, which is used with other MATLAB DDE functions. `'service'` is a string specifying the service or application name for the conversation. `'topic'` is a string specifying the topic for the conversation.

Examples

To initiate a conversation with Excel for the spreadsheet `'stocks.xls'`:

```
channel = ddeinit('excel','stocks.xls')

channel =
0.00
```

See Also

`ddeadv`, `ddeexec`, `ddepoke`, `ddereq`, `ddeterm`, `ddeunadv`

ddepoke

Purpose

Send data to application

Syntax

```
rc = ddepoke(channel,item,data,format,timeout)
```

Note Use COM, as described in COM Support in MATLAB. The ddepoke function will be removed in a future version of MATLAB.

Description

`rc = ddepoke(channel,item,data,format,timeout)` sends data to an application via an established DDE conversation. `ddepoke` formats the data matrix as follows before sending it to the server application:

- String matrices are converted, element by element, to characters and the resulting character buffer is sent.
- Numeric matrices are sent as tab-delimited columns and carriage-return, line-feed delimited rows of numbers. Only the real part of nonsparse matrices are sent.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddepoke` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the DDE item for the data sent. Item is the server data entity that is to contain the data sent in the `data` argument.
`data`	Matrix containing the data to send.

`format` (*optional*)	Scalar specifying the format of the data requested. The value indicates the Windows clipboard format to use for the data transfer. The only format currently supported is `cf_text`, which corresponds to a value of `1`.
`timeout` (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Assume that a conversation channel with Excel has previously been established with `ddeinit`. To send a 5-by-5 identity matrix to Excel, placing the data in Row 1, Column 1 through Row 5, Column 5:

```
rc = ddepoke(channel, 'r1c1:r5c5', eye(5));
```

See Also

`ddeadv`, `ddeexec`, `ddeinit`, `ddereq`, `ddeterm`, `ddeunadv`

ddereq

Purpose

Request data from application

Syntax

```
data = ddereq(channel,item,format,timeout)
```

Note Use COM, as described in COM Support in MATLAB. The ddereq function will be removed in a future version of MATLAB.

Description

`data = ddereq(channel,item,format,timeout)` requests data from a server application via an established DDE conversation. `ddereq` returns a matrix containing the requested data or an empty matrix if the function is unsuccessful.

If you omit optional arguments that are not at the end of the argument list, you must substitute the empty matrix for the missing argument(s).

If successful, `ddereq` returns a matrix containing the requested data in variable, `data`. Otherwise, it returns an empty matrix.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the server application's DDE item name for the data requested.

format (*optional*)	Two-element array specifying the format of the data requested. The first element specifies the Windows clipboard format to use. The only currently supported format is `cf_text`, which corresponds to a value of 1. The second element specifies the type of the resultant matrix. Valid types are `numeric` (the default, which corresponds to `0`) and `string` (which corresponds to a value of `1`). The default format array is `[1 0]`.
timeout (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Examples

Assume that you have an Excel spreadsheet `stocks.xls`. This spreadsheet contains the prices of three stocks in row 3 (columns 1 through 3) and the number of shares of these stocks in rows 6 through 8 (column 2). Initiate conversation with Excel with the command

```
channel = ddeinit('excel','stocks.xls')
```

DDE functions require the `rxcy` reference style for Excel worksheets. In Excel terminology the prices are in `r3c1:r3c3` and the shares in `r6c2:r8c2`.

Request the prices from Excel:

```
prices = ddereq(channel,'r3c1:r3c3')

prices =
42.50
15.00
78.88
```

Next, request the number of shares of each stock:

```
shares = ddereq(channel, 'r6c2:r8c2')

shares =
100.00
500.00
300.00
```

See Also

ddeadv, ddeexec, ddeinit, ddepoke, ddeterm, ddeunadv

Purpose

Solve delay differential equations (DDEs) with general delays

Syntax

```
sol = ddesd(ddefun,delays,history,tspan)
sol = ddesd(ddefun,delays,history,tspan,options)
```

Arguments

`ddefun`

Function handle that evaluates the right side of the differential equations $y'(t) = f(t, y(t), y(d(1)), \ldots, y(d(k)))$. The function must have the form

```
dydt = ddefun(t,y,Z)
```

where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(d(j))$ for delay $d(j)$ given as component j of `delays(t,y)`. The output is a column vector corresponding to $f(t, y(t), y(d(1)), \ldots, y(d(k)))$.

`delays`

Function handle that returns a column vector of delays $d(j)$. The delays can depend on both t and $y(t)$. `ddesd` imposes the requirement that $d(j) \le t$ by using `min(`$d(j)$`,` t`)`.

If all the delay functions have the form $d(j) = t - \tau_j$, you can set the argument `delays` to a constant vector `delays`$(j) = \tau_j$. With delay functions of this form, `ddesd` is used exactly like `dde23`.

`history`	Specify `history` in one of three ways: • A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector • A constant column vector, if $y(t)$ is constant • The solution `sol` from a previous integration, if this call continues that integration
`tspan`	Interval of integration as a vector `[t0,tf]` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = ddesd(ddefun,delays,history,tspan)` integrates the system of DDEs

$$y'(t) = f(t, y(t), y(d(1)), \ldots, y(d(k)))$$

on the interval $[t_0, t_f]$, where delays $d(j)$ can depend on both t and $y(t)$, and $t_0 < t_f$. Inputs `ddefun` and `delays` are function handles. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to provide additional parameters to the functions `ddefun`, `delays`, and `history`, if necessary.

`ddesd` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `ddesd` has the following fields.

`sol.x`	Mesh selected by `ddesd`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'ddesd'`

`sol = ddesd(ddefun,delays,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and "Initial Value Problems for DDEs" in the MATLAB documentation for details.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Events'` option to specify a function that `ddesd` calls to find where functions $g(t, y(t), y(d(1)), \ldots, y(d(k)))$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the `k`th event function in `events`:

- `value(k)` is the value of the `k`th event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and `0` otherwise.
- `direction(k) = 0` if you want `ddesd` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the 'Events' option and events are detected, the output structure sol also includes fields:

sol.xe	Row vector of locations of all events, i.e., times when an event function vanished
sol.ye	Matrix whose columns are the solution values corresponding to times in sol.xe
sol.ie	Vector containing indices that specify which event occurred at the corresponding time in sol.xe

Examples

The equation

```
sol = ddesd(@ddex1de,@ddex1delays,@ddex1hist,[0,5]);
```

solves a DDE on the interval [0,5] with delays specified by the function ddex1delays and differential equations computed by ddex1de. The history is evaluated for $t \le 0$ by the function ddex1hist. The solution is evaluated at 100 equally spaced points in [0,5]:

```
tint = linspace(0,5);
yint = deval(sol,tint);
```

and plotted with

```
plot(tint,yint);
```

This problem involves constant delays. The delay function has the form

```
function d = ddex1delays(t,y)
%DDEX1DELAYS  Delays for using with DDEX1DE.
d = [ t - 1
      t - 0.2];
```

The problem can also be solved with the syntax corresponding to constant delays

```
delays = [1, 0.2];
```

```
sol = ddesd(@ddex1de,delays,@ddex1hist,[0, 5]);
```

or using dde23:

```
sol = dde23(@ddex1de,delays,@ddex1hist,[0, 5]);
```

For more examples of solving delay differential equations see ddex2 and ddex3.

See Also

dde23, ddeget, ddeset, deval, function_handle (@)

References

[1] Shampine, L.F., "Solving ODEs and DDEs with Residual Control," *Applied Numerical Mathematics*, Vol. 52, 2005, pp. 113-127.

ddeset

Purpose

Create or alter delay differential equations options structure

Syntax

```
options = ddeset('name1',value1,'name2',value2,...)
options = ddeset(oldopts,'name1',value1,...)
options = ddeset(oldopts,newopts)
ddeset
```

Description

`options = ddeset('name1',value1,'name2',value2,...)` creates an integrator options structure `options` in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify the property. `ddeset` ignores case for property names.

`options = ddeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = ddeset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`ddeset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `ddeget` to query the `options` structure for the value of a specific property.

DDE Properties

The following sections describe the properties that you can set using `ddeset`. There are several categories of properties:

- Error control
- Solver output
- Step size
- Event location
- Discontinuities

Error Control Properties

At each step, solvers `dde23` and `ddesd` estimate an error `e`. `dde23` estimates the local truncation error, and `ddesd` estimates the residual. In either case, this error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

```
|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, `dde23` and `ddesd` deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For the absolute error tolerance, the scaling of the solution components is important: if `|y|` is somewhat smaller than `AbsTol`, the solver is not constrained to obtain any correct digits in `y`. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components

The following table describes the error control properties.

DDE Error Control Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the solution vector `y`. It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. The default, `1e-3`, corresponds to 0.1% accuracy. The estimated error in each integration step satisfies `|e(i)|max(RelTol*abs(y(i)), AbsTol(i))`.
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the individual components of the solution vector. `AbsTol(i)` is a threshold below which the value of the `i`th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components. If `AbsTol` is a vector, the length of `AbsTol` must be the same as the length of the solution vector `y`. If `AbsTol` is a scalar, the value applies to all components of `y`.
`NormControl`	`on` | {`off`}	Control error relative to norm of solution. Set this property `on` to request that the solvers control the error in each integration step with `norm(e)<= max(RelTol*norm(y),AbsTol)`. By default, solvers `dde23` and `ddesd` use a more stringent component-wise error control.

Solver Output Properties

You can use the solver output properties to control the output that the solvers generate.

DDE Solver Output Properties

Property	Value	Description

`OutputFcn`	Function handle {`@odeplot`}	The output function is a function that the solver calls after every successful integration step. To specify an output function, set `'OutputFcn'` to a function handle. For example,
		`options = ddeset('OutputFcn',...`
		`@myfun)`
		sets 'OutputFcn' to @myfun, a handle to the function `myfun`. See "Function Handles" in the MATLAB Programming documentation for more information.
		The output function must be of the form
		`status = myfun(t,y,flag)`
		"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to provide additional parameters to `myfun`, if necessary.
		The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:

DDE Solver Output Properties (Continued)

Property	Value	Description
		• `init` — The solver calls `myfun(tspan,y0,'init')` before beginning the integration to allow the output function to initialize. `tspan` is the input argument to solvers `dde23` and `ddesd`. `y0` is the initial value of the solution, either from `history(t0)` or specified in the `initialY` option. • {none} — The solver calls `status = myfun(t,y)` after each integration step on which output is requested. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points in `t`. If `t` is a vector, the `i`th column of `y` corresponds to the `i`th element of `t`. `myfun` must return a `status` output value of `0` or `1`. If literal > `status`, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button. • done — The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores.
		You can use these general purpose output functions or you can edit them to create your own. Type `help functionname` at the command line for more information. • `odeplot` – time series plotting (default when you call the solver with no output argument and you have not specified an output function) • `odephas2` – two-dimensional phase plane plotting • `odephas3` – three-dimensional phase plane plotting • `odeprint` – print solution as the solver computes it

DDE Solver Output Properties (Continued)

Property	Value	Description

`OutputSel`	Vector of indices	Vector of indices specifying which components of the solution vector the `dde23` or `ddesd` solver passes to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using
		`options = ddeset...`
		`('OutputFcn',@odeplot,...`
		`'OutputSel',[1 3]);`
		By default, the solver passes all components of the solution to the output function.

`Stats`	`on` | {`off`}	Specifies whether the solver should display statistics about its computations. By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays: • The number of successful steps • The number of failed attempts • The number of times the DDE function was called

Step Size Properties

The step size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step size properties.

DDE Step Size Properties

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size. `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the solver bases the initial step size on the slope of the solution at the initial time `tspan(1)`. The initial step size is limited by the shortest delay. If the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.

DDE Step Size Properties (Continued)

Property	Value	Description
`MaxStep`	Positive scalar {`0.1* abs(t0-tf)`}	Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep`: • When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. (See "Error Control Properties" on page 2-829 for a description of the error tolerance properties.)
		• To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver (`dde23` or `ddesd`) twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some DDE problems, the times of specific events are important. While solving a problem, the `dde23` and `ddesd` solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property.

DDE Events Property

String	Value	Description
`Events`	Function handle	Handle to a function that includes one or more event functions. See "Function Handles" in the MATLAB Programming documentation for more information. The function is of the form `[value,isterminal,direction] = events(t,y,Z)` `value`, `isterminal`, and `direction` are vectors for which the `i`th element corresponds to the `i`th event function:

DDE Events Property (Continued)

String	Value	Description
		• `value(i)` is the value of the `i`th event function. • `isterminal(i) = 1` if you want the integration to terminate at a zero of this event function, and `0` otherwise. • `direction(i) = 0` if you want the solver (`dde23` or `ddesd`) to locate all zeros (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing. If you specify an events function and events are detected, the solver returns three additional fields in the solution structure `sol`: • `sol.xe` is a row vector of times at which events occur. • `sol.ye` is a matrix whose columns are the solution values corresponding to times in `sol.xe`. • `sol.ie` is a vector containing indices that specify which event occurred at the corresponding time in `sol.xe`.
		For examples that use an event function while solving ordinary differential equation problems, see "Example: Simple Event Location" (`ballode`) and "Example: Advanced Event Location" (`orbitode`), in the MATLAB Mathematics documentation.

Discontinuity Properties

Solvers `dde23` and `ddesd` can solve problems with discontinuities in the history or in the coefficients of the equations. The following properties enable you to provide these solvers with a different initial value, and, for `dde23`, locations of known discontinuities. See "Discontinuities" in the MATLAB Mathematics documentation for more information.

The following table describes the discontinuity properties.

DDE Discontinuity Properties

String	Value	Description
`Jumps`	Vector	Location of discontinuities. Points t where the history or solution may have a jump discontinuity in a low-order derivative. This applies only to the `dde23` solver.
`InitialY`	Vector	Initial value of solution. By default the initial value of the solution is the value returned by `history` at the initial point. Supply a different initial value as the value of the `InitialY` property.

Example

To create an options structure that changes the relative error tolerance of the solver from the default value of `1e-3` to `1e-4`, enter

```
options = ddeset('RelTol', 1e-4);
```

To recover the value of `'RelTol'` from `options`, enter

```
ddeget(options, 'RelTol')

ans =

   1.0000e-004
```

See Also

`dde23`, `ddesd`, `ddeget`, `function_handle` (`@`)

Purpose

Terminate Dynamic Data Exchange (DDE) conversation

Syntax

Note Use COM, as described in COM Support in MATLAB. The ddeterm function will be removed in a future version of MATLAB.

```
rc = ddeterm(channel)
```

Description

`rc = ddeterm(channel)` accepts a channel handle returned by a previous call to `ddeinit` that established the DDE conversation. `ddeterm` terminates this conversation. `rc` is a return code where `0` indicates failure and `1` indicates success.

Examples

To close a conversation channel previously opened with `ddeinit`:

```
rc = ddeterm(channel)

rc =
1.00
```

See Also

`ddeadv`, `ddeexec`, `ddeinit`, `ddepoke`, `ddereq`, `ddeunadv`

ddeunadv

Purpose

Release advisory link

Syntax

```
rc = ddeunadv(channel,item,format,timeout)
```

Note Use COM, as described in COM Support in MATLAB. The ddeunadv function will be removed in a future version of MATLAB.

Description

`rc = ddeunadv(channel,item,format,timeout)` releases the advisory link between MATLAB and the server application established by an earlier `ddeadv` call. The `channel`, *`item`*, and `format` must be the same as those specified in the call to `ddeadv` that initiated the link. If you include the `timeout` argument but accept the default `format`, you must specify `format` as an empty matrix.

If successful, `ddeunadv` returns 1 in variable, `rc`. Otherwise it returns 0.

Arguments

`channel`	Conversation channel from `ddeinit`.
`item`	String specifying the DDE item name for the advisory link. Changing the data identified by `item` at the server triggers the advisory link.
format (*optional*)	Two-element array. This must be the same as the `format` argument for the corresponding `ddeadv` call.
timeout (*optional*)	Scalar specifying the time-out limit for this operation. `timeout` is specified in milliseconds. (1000 milliseconds = 1 second). The default value of `timeout` is three seconds.

Example

To release an advisory link established previously with `ddeadv`:

```
rc = ddeunadv(channel, 'r1c1:r5c5')
rc =
```

```
1.00
```

See Also

ddeadv, ddeexec, ddeinit, ddepoke, ddereq, ddeterm

deal

Purpose

Distribute inputs to outputs

Note As of MATLAB Version 7.0, you can access the contents of cell arrays and structure fields without using the `deal` function. See Example 3, below.

Syntax

```
[Y1, Y2, Y3, ...] = deal(X)
[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)
[S.field] = deal(X)
[X{:}] = deal(A.field)
[Y1, Y2, Y3, ...] = deal(X{:})
[Y1, Y2, Y3, ...] = deal(S.field)
```

Description

`[Y1, Y2, Y3, ...] = deal(X)` copies the single input to all the requested outputs. It is the same as `Y1 = X, Y2 = X, Y3 = X, ...`

`[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)` is the same as `Y1 = X1; Y2 = X2; Y3 = X3; ...`

Remarks

`deal` is most useful when used with cell arrays and structures via comma-separated list expansion. Here are some useful constructions:

`[S.field] = deal(X)` sets all the fields with the name `field` in the structure array `S` to the value `X`. If `S` doesn't exist, use `[S(1:m).field] = deal(X)`.

`[X{:}] = deal(A.field)` copies the values of the field with name `field` to the cell array `X`. If `X` doesn't exist, use `[X{1:m}] = deal(A.field)`.

`[Y1, Y2, Y3, ...] = deal(X{:})` copies the contents of the cell array `X` to the separate variables `Y1, Y2, Y3, ...`

`[Y1, Y2, Y3, ...] = deal(S.field)` copies the contents of the fields with the name `field` to separate variables `Y1, Y2, Y3, ...`

Examples

Example 1 — Assign Data From a Cell Array

Use `deal` to copy the contents of a 4-element cell array into four separate output variables.

```
C = {rand(3) ones(3,1) eye(3) zeros(3,1)};
[a,b,c,d] = deal(C{:})

a =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

b =
    1
    1
    1

c =
    1    0    0
    0    1    0
    0    0    1

d =
    0
    0
    0
```

Example 2 — Assign Data From Structure Fields

Use `deal` to obtain the contents of all the name fields in a structure array:

```
A.name = 'Pat'; A.number = 176554;
A(2).name = 'Tony'; A(2).number = 901325;
[name1,name2] = deal(A(:).name)

name1 =
   Pat
```

```
name2 =
   Tony
```

Example 3 — Doing the Same Without deal

As of MATLAB Version 7.0, you can, in most cases, access the contents of cell arrays and structure fields without using the `deal` function. The two commands shown below perform the same operation as those used in the previous two examples, except that these commands do not require `deal`.

```
[a,b,c,d] = C{:}
[name1,name2] = A(:).name
```

See Also

`cell`, `iscell`, `celldisp`, `struct`, `isstruct`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, `cell2struct`, `struct2cell`

Purpose

Strip trailing blanks from end of string

Syntax

```
str = deblank(str)
c = deblank(c)
```

Description

`str = deblank(str)` removes all trailing whitespace and null characters from the end of character string `str`. A whitespace is any character for which the `isspace` function returns logical 1 (`true`).

`c = deblank(c)` when `c` is a cell array of strings, applies `deblank` to each element of `c`.

The `deblank` function is useful for cleaning up the rows of a character array.

Examples

Example 1 – Removing Trailing Blanks From a String

Compose a string `str` that contains space, tab, and null characters:

```
NL = char(0);    TAB = char(9);
str = [NL 32 TAB NL 'AB' 32 NL 'CD' NL 32 TAB NL 32];
```

Display all characters of the string between | symbols:

```
['|' str '|']
ans =
    |    AB  CD       |
```

Remove trailing whitespace and null characters, and redisplay the string:

```
newstr = deblank(str);

['|' newstr '|']
ans =
    |    AB  CD|
```

Example 2– Removing Trailing Blanks From a Cell Array of Strings

```
A{1,1} = 'MATLAB    ';
A{1,2} = 'SIMULINK    ';
A{2,1} = 'Toolboxes    ';
A{2,2} = 'The MathWorks    ';
A =

    'MATLAB    '        'SIMULINK    '
    'Toolboxes    '    'The MathWorks    '


deblank(A)
ans =

    'MATLAB'            'SIMULINK'
    'Toolboxes'         'The MathWorks'
```

See Also

`strjust`, `strtrim`

Purpose

List M-file debugging functions

GUI Alternatives

Use the **Debug** menu in most desktop tools, or use the Editor/Debugger.

Syntax

```
debug
```

Description

`debug` lists M-file debugging functions.

Use debugging functions (listed in the See Also section) to help you identify problems in your M-files. Set breakpoints using `dbstop`. When MATLAB encounters a breakpoint during execution, it enters debug mode, the Editor/Debugger becomes active, and the prompt in the Command Window changes to a `K>>`. Any MATLAB command is allowed at the prompt. To resume execution, use `dbcont` or `dbstep`. To exit from debug mode, use `dbquit`.

To open the M-File in the Editor/Debugger when execution reaches a breakpoint, select **Debug > Open M-Files When Debugging**.

See Also

`dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`, `evalin`, `whos`

"Finding Errors, Debugging, and Correcting M-Files" in the MATLAB Desktop Tools and Development Environment documentation

dec2base

Purpose

Convert decimal to base N number in string

Syntax

```
str = dec2base(d, base)
str = dec2base(d, base, n)
```

Description

`str = dec2base(d, base)` converts the nonnegative integer `d` to the specified `base`. `d` must be a nonnegative integer smaller than 2^52, and `base` must be an integer between 2 and 36. The returned argument `str` is a string.

`str = dec2base(d, base, n)` produces a representation with at least `n` digits.

Examples

The expression `dec2base(23, 2)` converts 23_{10} to base 2, returning the string `'10111'`.

See Also

`base2dec`

Purpose

Convert decimal to binary number in string

Syntax

```
str = dec2bin(d)
str = dec2bin(d,n)
```

Description

returns the

`str = dec2bin(d)` binary representation of `d` as a string. `d` must be a nonnegative integer smaller than 2^52.

`str = dec2bin(d,n)` produces a binary representation with at least `n` bits.

Examples

Decimal 23 converts to binary 010111:

```
dec2bin(23)
ans =
    10111
```

See Also

`bin2dec`, `dec2hex`

dec2hex

Purpose

Convert decimal to hexadecimal number in string

Syntax

```
str = dec2hex(d)
str = dec2hex(d, n)
```

Description

`str = dec2hex(d)` converts the decimal integer `d` to its hexadecimal representation stored in a MATLAB string. `d` must be a nonnegative integer smaller than 2^52.

`str = dec2hex(d, n)` produces a hexadecimal representation with at least `n` digits.

Examples

To convert decimal 1023 to hexadecimal,

```
dec2hex(1023)

ans =
    3FF
```

See Also

`dec2bin`, `format`, `hex2dec`, `hex2num`

Purpose

Compute consistent initial conditions for `ode15i`

Syntax

```
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,
   options)
[y0mod,yp0mod,resnrm] = decic(odefun,t0,y0,fixed_y0,yp0,
   fixed_yp0...)
```

Description

`[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)` uses the inputs `y0` and `yp0` as initial guesses for an iteration to find output values that satisfy the requirement $f(t0, y0mod, yp0mod) = 0$, i.e., `y0mod` and `yp0mod` are consistent initial conditions. `odefun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. The function `decic` changes as few components of the guesses as possible. You can specify that `decic` holds certain components fixed by setting `fixed_y0(i) = 1` if no change is permitted in the guess for `y0(i)` and 0 otherwise. `decic` interprets `fixed_y0 = []` as allowing changes in all entries. `fixed_yp0` is handled similarly.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

You cannot fix more than `length(y0)` components. Depending on the problem, it may not be possible to fix this many. It also may not be possible to fix certain components of `y0` or `yp0`. It is recommended that you fix no more components than necessary.

`[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options)` computes as above with default tolerances for consistent initial conditions, `AbsTol` and `RelTol`, replaced by the values in options, a structure you create with the `odeset` function.

`[y0mod,yp0mod,resnrm] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0...)` returns the

norm of `odefun(t0,y0mod,yp0mod)` as `resnrm`. If the norm seems unduly large, use `options` to decrease `RelTol` (`1e-3` by default).

Examples

These demos provide examples of the use of `decic` in solving implicit ODEs: `ihb1dae`, `iburgersode`.

See Also

`ode15i`, `odeget`, `odeset`, `function_handle` (`@`)

Purpose

Deconvolution and polynomial division

Syntax

```
[q,r] = deconv(v,u)
```

Description

`[q,r] = deconv(v,u)` deconvolves vector `u` out of vector `v`, using long division. The quotient is returned in vector `q` and the remainder in vector `r` such that `v = conv(u,q)+r` .

If `u` and `v` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division. The result of dividing `v` by `u` is quotient `q` and remainder `r`.

Examples

If

```
u = [1   2   3   4]
v = [10   20   30]
```

the convolution is

```
c = conv(u,v)
c =
    10     40     100     160     170     120
```

Use deconvolution to recover `u`:

```
[q,r] = deconv(c,u)
q =
    10    20    30
r =
    0     0     0     0     0     0
```

This gives a quotient equal to `v` and a zero remainder.

Algorithm

`deconv` uses the `filter` primitive.

See Also

`conv`, `residue`

Purpose

Discrete Laplacian

Syntax

```
L = del2(U)
-L = del2(U)
L = del2(U,h)
L = del2(U,hx,hy)
L = del2(U,hx,hy,hz,...)
```

Definition

If the matrix `U` is regarded as a function $u(x, y)$ evaluated at the point on a square grid, then `4*del2(U)` is a finite difference approximation of Laplace's differential operator applied to u, that is:

$$l = \frac{\nabla^2 u}{4} = \frac{1}{4}\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right)$$

where:

$$l_{ij} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) - u_{i,}$$

in the interior. On the edges, the same formula is applied to a cubic extrapolation.

For functions of more variables $u(x, y, z, \ldots)$, `del2(U)` is an approximation,

$$l = \frac{\nabla^2 u}{2N} = \frac{1}{2N}\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} + \frac{d^2u}{dz^2} + \ldots\right)$$

where N is the number of variables in u.

Description

`L = del2(U)` where `U` is a rectangular array is a discrete approximation of

$$l = \frac{\nabla^2 u}{4} = \frac{1}{4}\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right)$$

The matrix `L` is the same size as `U` with each element equal to the difference between an element of `U` and the average of its four neighbors.

`-L = del2(U)` when `U` is an multidimensional array, returns an approximation of

$$\frac{\nabla^2 u}{2N}$$

where N is `ndims(u)`.

`L = del2(U,h)` where `H` is a scalar uses `H` as the spacing between points in each direction (`h=1` by default).

`L = del2(U,hx,hy)` when `U` is a rectangular array, uses the spacing specified by `hx` and `hy`. If `hx` is a scalar, it gives the spacing between points in the x-direction. If `hx` is a vector, it must be of length `size(u,2)` and specifies the x-coordinates of the points. Similarly, if `hy` is a scalar, it gives the spacing between points in the y-direction. If `hy` is a vector, it must be of length `size(u,1)` and specifies the y-coordinates of the points.

`L = del2(U,hx,hy,hz,...)` where `U` is multidimensional uses the spacing given by `hx`, `hy`, `hz`, ...

Remarks

MATLAB computes the boundaries of the grid by extrapolating the second differences from the interior. The algorithm used for this computation can be seen in the `del2` M-file code. To view this code, type

```
type del2
```

Examples

The function

$$u(x, y) = x^2 + y^2$$

has

$$\nabla^2 u = 4$$

For this function, `4*del2(U)` is also `4`.

```
[x,y] = meshgrid(-4:4,-3:3);
U = x.*x+y.*y
U =
    25    18    13    10     9    10    13    18    25
    20    13     8     5     4     5     8    13    20
    17    10     5     2     1     2     5    10    17
    16     9     4     1     0     1     4     9    16
    17    10     5     2     1     2     5    10    17
    20    13     8     5     4     5     8    13    20
    25    18    13    10     9    10    13    18    25

V = 4*del2(U)
V =
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
```

See Also

`diff`, `gradient`

Purpose

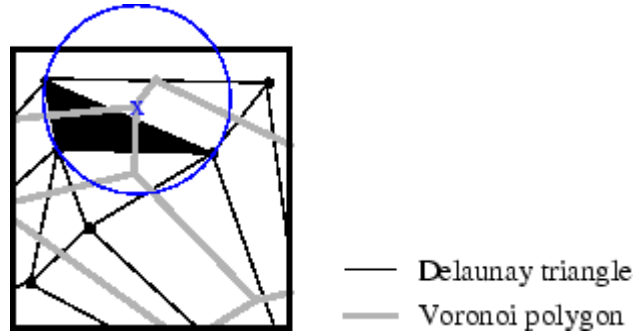
Delaunay triangulation

Syntax

```
TRI = delaunay(x,y)
TRI = delaunay(x,y,options)
```

Definition

Given a set of data points, the *Delaunay triangulation* is a set of lines connecting each point to its natural neighbors. The Delaunay triangulation is related to the Voronoi diagram — the circle circumscribed about a Delaunay triangle has its center at the vertex of a Voronoi polygon.



Description

`TRI = delaunay(x,y)` for the data points defined by vectors `x` and `y`, returns a set of triangles such that no data points are contained in any triangle's circumscribed circle. Each row of the `m`-by-3 matrix `TRI` defines one such triangle and contains indices into `x` and `y`. If the original data points are collinear or `x` is empty, the triangles cannot be computed and `delaunay` returns an empty matrix.

`delaunay` uses Qhull.

`TRI = delaunay(x,y,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`. The default options are `{'Qt','Qbb','Qc'}`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org`.

Remarks

The Delaunay triangulation is used by: `griddata` (to interpolate scattered data), `voronoi` (to compute the `voronoi` diagram), and is useful by itself to create a triangular grid for scattered data points.

The functions `dsearch` and `tsearch` search the triangulation to find nearest neighbor points or enclosing triangles, respectively.

Visualization

Use one of these functions to plot the output of `delaunay`:

`triplot` — Displays the triangles defined in the `m`-by-3 matrix `TRI`. See Example 1.

`trisurf` — Displays each triangle defined in the m-by-3 matrix TRI as a surface in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example

```
trisurf(TRI,x,y,zeros(size(x)))
```

See Example 2.

`trimesh` — Displays each triangle defined in the m-by-3 matrix TRI as a mesh in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example,

```
trimesh(TRI,x,y,zeros(size(x)))
```

produces almost the same result as `triplot`, except in 3-D space. See Example 2.

Examples

Example 1

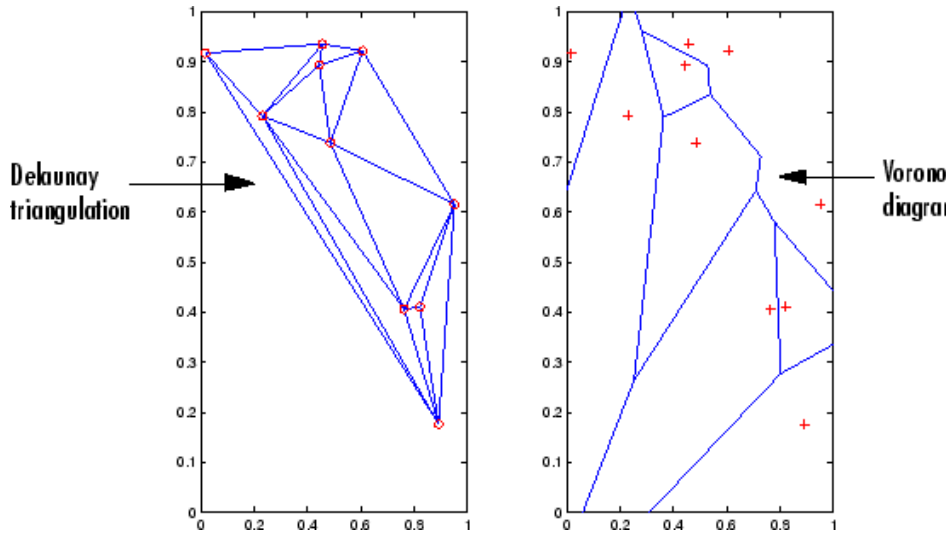
Plot the Delaunay triangulation for 10 randomly generated points.

```
rand('state',0);
x = rand(1,10);
y = rand(1,10);
```

```
TRI = delaunay(x,y);
subplot(1,2,1),...
triplot(TRI,x,y)
axis([0 1 0 1]);
hold on;
plot(x,y,'or');
hold off
```

Compare the Voronoi diagram of the same points:

```
[vx, vy] = voronoi(x,y,TRI);
subplot(1,2,2),...
plot(x,y,'r+',vx,vy,'b-'),...
axis([0 1 0 1])
```

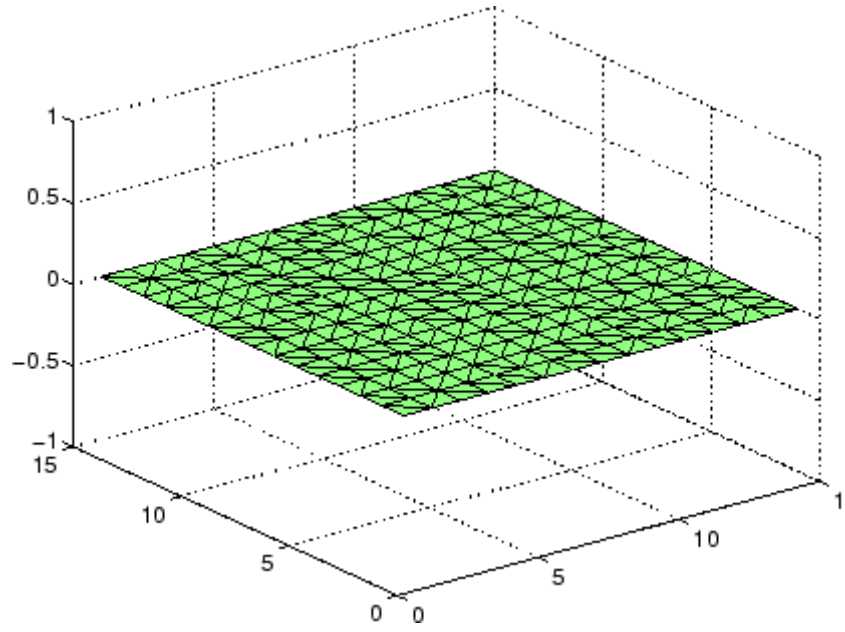


Example 2

Create a 2-D grid then use `trisurf` to plot its Delaunay triangulation in 3-D space by using `0`s for the third dimension.

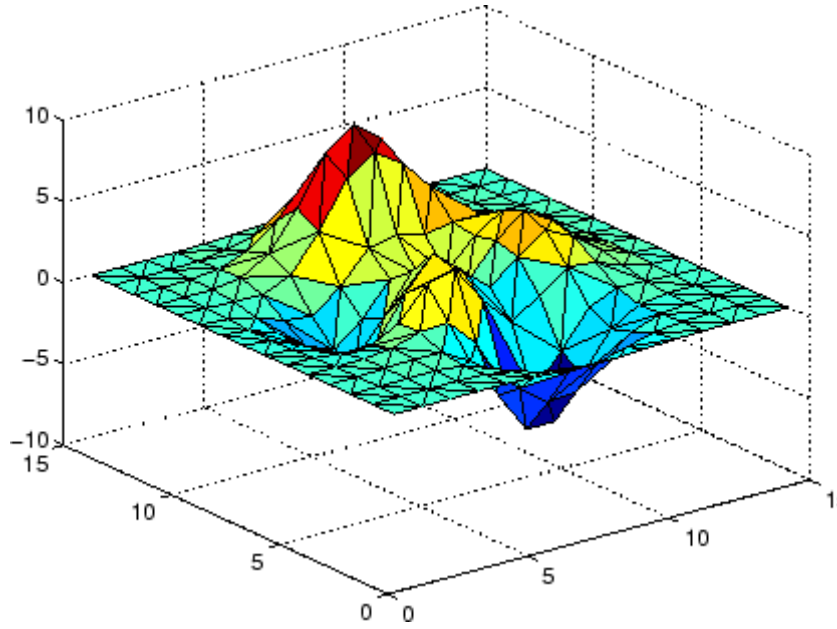
```
[x,y] = meshgrid(1:15,1:15);
```

```
tri = delaunay(x,y);
trisurf(tri,x,y,zeros(size(x)))
```



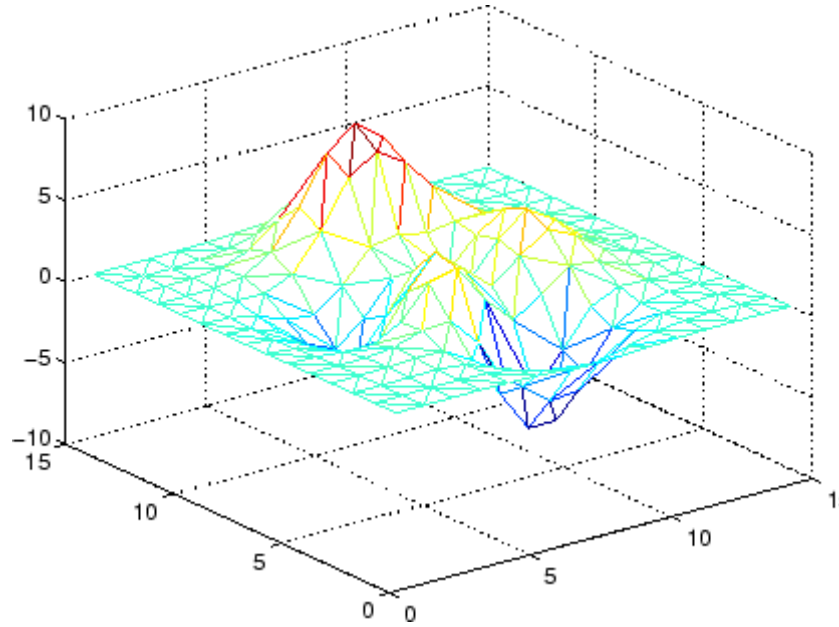
Next, generate `peaks` data as a 15-by-15 matrix, and use that data with the Delaunay triangulation to produce a surface in 3-D space.

```
z = peaks(15);
trisurf(tri,x,y,z)
```



You can use the same data with `trimesh` to produce a mesh in 3-D space.

```
trimesh(tri,x,y,z)
```



Example 3

The following example illustrates the `options` input for `delaunay`.

```
x = [-0.5 -0.5 0.5 0.5];
       y = [-0.5 0.5 0.5 -0.5];
```

The command

```
T = delaunay(X);
```

returns the following error message.

```
??? qhull input error: can not scale last coordinate. Input is
cocircular
   or cospherical. Use option 'Qz' to add a point at infinity.
```

The error message indicates that you should add `'Qz'` to the default Qhull options.

```
tri = delaunay(x,y,{'Qt','Qbb','Qc','Qz'})

tri =

     3     2     1
     3     4     1
```

Algorithm

`delaunay` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`delaunay3`, `delaunay`, `dsearch`, `griddata`, `plot`, `triplot`, `trimesh`, `trisurf`, `tsearch`, `voronoi`

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483.

delaunay3

Purpose

3-D Delaunay tessellation

Syntax

```
T = delaunay3(x,y,z)
T = delaunay3(x,y,z,options)
```

Description

`T = delaunay3(x,y,z)` returns an array `T`, each row of which contains the indices of the points in `(x,y,z)` that make up a tetrahedron in the tessellation of `(x,y,z)`. `T` is a `numtes`-by-`4` array where `numtes` is the number of facets in the tessellation. `x`, `y`, and `z` are vectors of equal length. If the original data points are collinear or `x`, `y`, and `z` define an insufficient number of points, the triangles cannot be computed and `delaunay3` returns an empty matrix.

`delaunay3` uses Qhull.

`T = delaunay3(x,y,z,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunay3`. The default options are `{'Qt','Qbb','Qc'}`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org`.

Visualization

Use `tetramesh` to plot `delaunay3` output. `tetramesh` displays the tetrahedrons defined in `T` as mesh. `tetramesh` uses the default transparency parameter value `'FaceAlpha' = 0.9`.

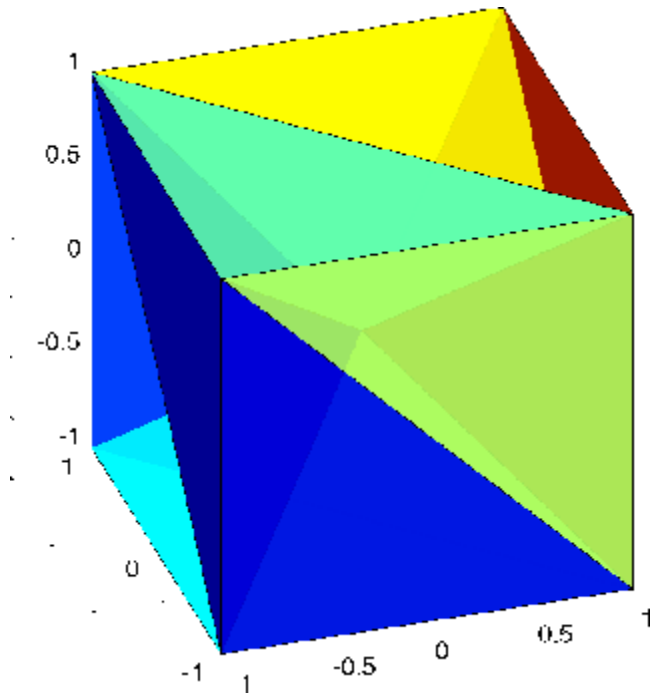
Examples

Example 1

This example generates a 3-dimensional Delaunay tessellation, then uses `tetramesh` to plot the tetrahedrons that form the corresponding simplex. `camorbit` rotates the camera position to provide a meaningful view of the figure.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);  % A cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
```

```
% [x,y,z] are corners of a cube plus the center.
Tes = delaunay3(x,y,z)

Tes =

     9   1   5   6
     3   9   1   5
     2   9   1   6
     2   3   9   4
     2   3   9   1
     7   9   5   6
     7   3   9   5
     8   7   9   6
     8   2   9   6
     8   2   9   4
     8   3   9   4
     8   7   3   9

X = [x(:) y(:) z(:)];
tetramesh(Tes,X);camorbit(20,0)
```



Example 2

The following example illustrates the `options` input for `delaunay3`.

```
X = [-0.5 -0.5 -0.5 -0.5 0.5 0.5 0.5 0.5];
Y = [-0.5 -0.5 0.5 0.5 -0.5 -0.5 0.5 0.5];
Z = [-0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5];
```

The command

```
T = delaunay3(X);
```

returns the following error message.

```
??? qhull input error: can not scale last coordinate. Input is
cocircular
```

```
    or cospherical. Use option 'Qz' to add a point at infinity.
```

The error message indicates that you should add `'Qz'` to the default Qhull options.

```
T = delaunay3( X, Y, Z, {'Qt', 'Qbb', 'Qc', 'Qz'} )

T =

     4     3     5     1
     4     2     5     1
     4     7     3     5
     4     7     8     5
     4     6     2     5
     4     6     8     5
```

Algorithm

`delaunay3` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`delaunay`, `delaunayn`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483.

delaunayn

Purpose

N-D Delaunay tessellation

Syntax

```
T = delaunayn(X)
T = delaunayn(X, options)
```

Description

`T = delaunayn(X)` computes a set of simplices such that no data points of `X` are contained in any circumspheres of the simplices. The set of simplices forms the Delaunay tessellation. `X` is an `m`-by-`n` array representing m points in n-dimensional space. `T` is a `numt`-by-`(n+1)` array where each row contains the indices into `X` of the vertices of the corresponding simplex.

`delaunayn` uses Qhull.

`T = delaunayn(X, options)` specifies a cell array of strings `options` to be used as options in Qhull. The default options are:

- `{'Qt','Qbb','Qc'}` for 2- and 3-dimensional input
- `{'Qt','Qbb','Qc','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options used. If `options` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org`.

Visualization

Plotting the output of `delaunayn` depends of the value of `n`:

- For `n = 2`, use `triplot`, `trisurf`, or `trimesh` as you would for `delaunay`.
- For `n = 3`, use `tetramesh` as you would for `delaunay3`.

  For more control over the color of the facets, use `patch` to plot the output. For an example, see “Tessellation and Interpolation of Scattered Data in Higher Dimensions” in the MATLAB documentation.
- You cannot plot `delaunayn` output for `n > 3`.

Examples

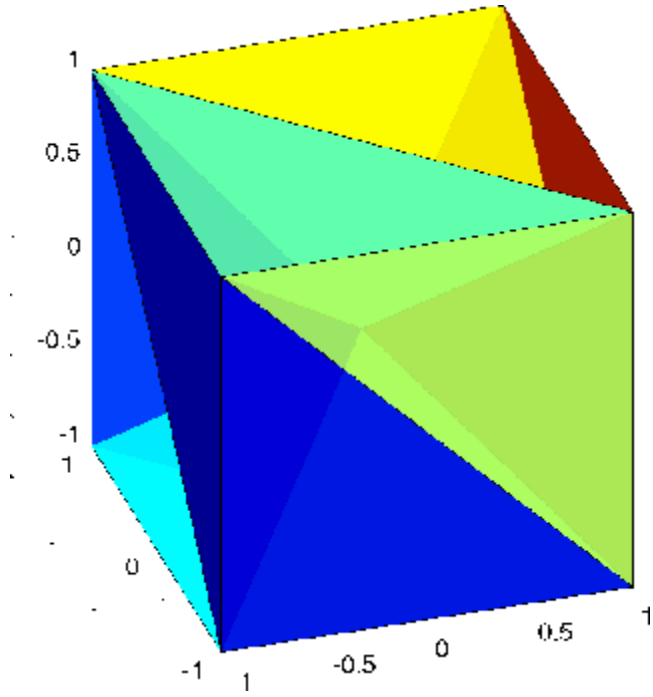
Example 1

This example generates an n-dimensional Delaunay tessellation, where `n = 3`.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);  % A cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
X = [x(:) y(:) z(:)];
Tes = delaunayn(X)

Tes =
   9  1  5  6
   3  9  1  5
   2  9  1  6
   2  3  9  4
   2  3  9  1
   7  9  5  6
   7  3  9  5
   8  7  9  6
   8  2  9  6
   8  2  9  4
   8  3  9  4
   8  7  3  9
```

You can use `tetramesh` to visualize the tetrahedrons that form the corresponding simplex. `camorbit` rotates the camera position to provide a meaningful view of the figure.

```
tetramesh(Tes,X);camorbit(20,0)
```



Example 2

The following example illustrates the `options` input for `delaunayn`.

```
X = [-0.5 -0.5  -0.5;...
            -0.5 -0.5   0.5;...
            -0.5  0.5  -0.5;...
            -0.5  0.5   0.5;...
             0.5 -0.5  -0.5;...
             0.5 -0.5   0.5;...
             0.5  0.5  -0.5;...
             0.5  0.5   0.5];
```

The command

```
T = delaunayn(X);
```

returns the following error message.

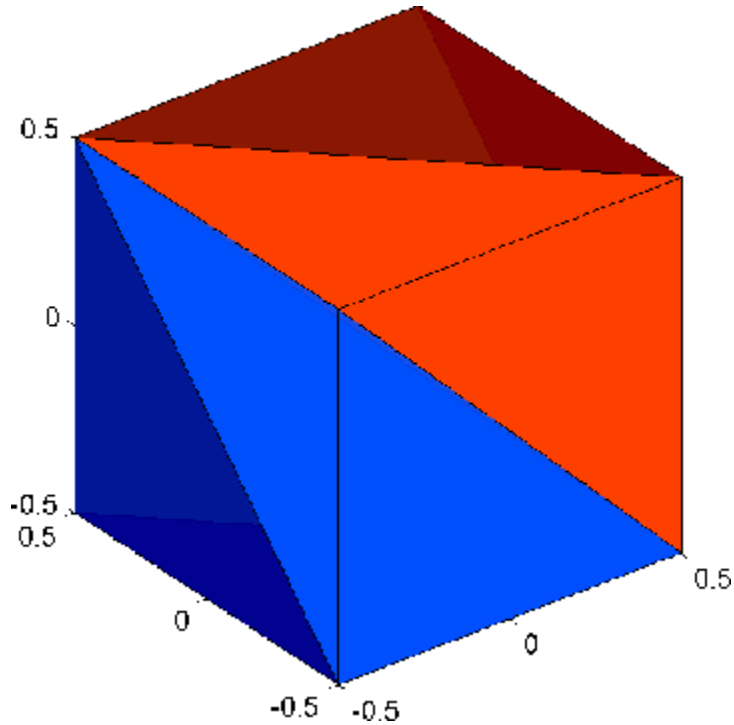
??? qhull input error: can not scale last coordinate. Input is cocircular or cospherical. Use option 'Qz' to add a point at infinity.

This suggests that you add `'Qz'` to the default options.

```
T = delaunayn(X,{'Qt','Qbb','Qc','Qz'});
```

To visualize this answer you can use the `tetramesh` function:

```
tetramesh(T,X)
```



Algorithm

`delaunayn` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`convhulln`, `delaunayn`, `delaunay3`, `tetramesh`, `voronoin`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483.

Purpose

Remove files or graphics objects

Graphical Interface

As an alternative to the `delete` function, you can delete files using the "Current Directory Browser", as described in the Desktop Tools and Development Environment documentation.

Syntax

```
delete filename
delete(h)
delete('filename')
```

Description

`delete filename` deletes the named file from the disk. The `filename` may include an absolute pathname or a pathname relative to the current directory. The `filename` may also include wildcards, (`*`).

`delete(h)` deletes the graphics object with handle `h`. The function deletes the object without requesting verification even if the object is a window.

`delete('filename')` is the function form of `delete`. Use this form when the filename is stored in a string.

Note MATLAB does not ask for confirmation when you enter the `delete` command. To avoid accidentally losing files or graphics objects that you need, make sure that you have accurately specified the items you want deleted.

Remarks

The action that the delete function takes on deleted files depends upon the setting of the MATLAB recycle state. If you set the recycle state to `on`, MATLAB moves deleted files to your recycle bin or temporary directory. With the recycle state set to `off` (the default), deleted files are permanently removed from the system.

To set the recycle state for all MATLAB sessions, use the **Preferences** dialog box. Open the **Preferences** dialog and select **General**. To enable or disable recycling, click **Move files to the recycle bin** or **Delete files permanently**. See "General Preferences for MATLAB"

in the Desktop Tools and Development Environment documentation for more information.

The `delete` function deletes files and handles to graphics objects only. Use the `rmdir` function to delete directories.

Examples

To delete all files with a `.mat` extension in the `../mytests/` directory, type

```
delete('../mytests/*.mat')
```

To delete a directory, use `rmdir` rather than `delete`:

```
rmdir mydirectory
```

See Also

`recycle`, `dir`, `edit`, `fileparts`, `mkdir`, `rmdir`, `type`

Purpose

Remove COM control or server

Syntax

```
h.delete
delete(h)
```

Description

`h.delete` releases all interfaces derived from the specified COM server or control, and then deletes the server or control itself. This is different from releasing an interface, which releases and invalidates only that interface.

`delete(h)` is an alternate syntax for the same operation.

Examples

Create a Microsoft Calender application. Then create a `TitleFont` interface and use it to change the appearance of the font of the calendar's title:

```
f = figure('position',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);

TFont = cal.TitleFont
TFont =
   Interface.Standard_OLE_Types.Font

TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

When you're finished working with the title font, release the `TitleFont` interface:

```
TFont.release;
```

Now create a `GridFont` interface and use it to modify the size of the calendar's date numerals:

```
GFont = cal.GridFont
GFont =
   Interface.Standard_OLE_Types.Font
```

```
GFont.Size = 16;
```

When you're done, delete the `cal` object and the figure window. Deleting the `cal` object also releases all interfaces to the object (e.g., `GFont`):

```
cal.delete;
delete(f);
clear f;
```

Note that, although the object and interfaces themselves have been destroyed, the variables assigned to them still reside in the MATLAB workspace until you remove them with `clear`:

```
whos
  Name        Size                    Bytes  Class

  GFont       1x1                         0  handle
  TFone       1x1                         0  handle
  cal         1x1                         0  handle

Grand total is 3 elements using 0 bytes
```

See Also

`release`, `save`, `load`, `actxcontrol`, `actxserver`

Purpose

Remove file on FTP server

Syntax

```
delete(f,'filename')
```

Description

`delete(f,'filename')` removes the file `filename` from the current directory of the FTP server `f`, where `f` was created using `ftp`.

Examples

Connect to server `testsite`.

```
test=ftp('ftp.testsite.com')
```

Change the current directory to `testdir` and view the contents.

```
cd(test,'testdir');
dir(test)
```

See Also

`ftp`

delete (serial)

Purpose

Remove serial port object from memory

Syntax

```
delete(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`delete(obj)` removes `obj` from memory.

Remarks

When you delete `obj`, it becomes an *invalid* object. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command. If multiple references to `obj` exist in the workspace, then deleting one reference invalidates the remaining references.

If `obj` is connected to the device, it has a `Status` property value of `open`. If you issue `delete` while `obj` is connected, then the connection is automatically broken. You can also disconnect `obj` from the device with the `fclose` function.

If you use the `help` command to display help for `delete`, then you need to supply the pathname shown below.

```
help serial/delete
```

Example

This example creates the serial port object `s`, connects `s` to the device, writes and reads text data, disconnects `s` from the device, removes `s` from memory using `delete`, and then removes `s` from the workspace using `clear`.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

See Also

Functions

`clear`, `fclose`, `isvalid`

Properties

`Status`

delete (timer)

Purpose

Remove timer object from memory

Syntax

```
delete(obj)
```

Description

`delete(obj)` removes the timer object, `obj`, from memory. If `obj` is an array of timer objects, `delete` removes all the objects from memory.

When you delete a timer object, it becomes invalid and cannot be reused. Use the `clear` command to remove invalid timer objects from the workspace.

If multiple references to a timer object exist in the workspace, deleting the timer object invalidates the remaining references. Use the `clear` command to remove the remaining references to the object from the workspace.

See Also

`clear`, `isvalid(timer)`, `timer`

Purpose

Remove custom property from object

Syntax

```
h.deleteproperty('propertyname')
deleteproperty(h, 'propertyname')
```

Description

`h.deleteproperty('propertyname')` deletes the property specified in the string `propertyname` from the custom properties belonging to object or interface, `h`.

`deleteproperty(h, 'propertyname')` is an alternate syntax for the same operation.

Note You can only delete properties that have been created with `addproperty`.

Examples

Create an `mwsamp` control and add a new property named `Position` to it. Assign an array value to the property:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.get
     Label: 'Label'
    Radius: 20

h.addproperty('Position');
h.Position = [200 120];
h.get
       Label: 'Label'
      Radius: 20
    Position: [200 120]
```

Delete the custom `Position` property:

```
h.deleteproperty('Position');
h.get
     Label: 'Label'
```

```
                    Radius: 20
```

See Also

addproperty, get, set, inspect

Purpose

Remove `tsdata.event` objects from `timeseries` object

Syntax

```
ts = delevent(ts,event)
ts = delevent(ts,events)
ts = delevent(ts,event,n)
```

Description

`ts = delevent(ts,event)` removes the `tsdata.event` object from the `ts.events` property, where `event` is an event name string.

`ts = delevent(ts,events)` removes the `tsdata.event` object from the `ts.events` property, where `events` is a cell array of event name strings.

`ts = delevent(ts,event,n)` removes the nth `tsdata.event` object from the `ts.events` property. `event` is the name of the `tsdata.event` object.

Examples

The following example shows how to remove an event from a `timeseries` object:

1. Create a time series.

   ```
   ts = timeseries(rand(5,4))
   ```

2. Create an event object called `'test'` such that the event occurs at time 3.

   ```
   e = tsdata.event('test',3)
   ```

3. Add the event object to the time series `ts`.

   ```
   ts = addevent(ts,e)
   ```

4. Remove the event object from the time series `ts`.

   ```
   ts = delevent(ts,'test')
   ```

See Also

`addevent`, `timeseries`, `tsdata.event`, `tsprops`

delsample

Purpose

Remove sample from `timeseries` object

Syntax

```
ts = delsample(ts,'Index',N)
ts = delsample(ts,'Value',Time)
```

Description

`ts = delsample(ts,'Index',N)` deletes samples from the `timeseries` object `ts`. `N` specifies the indices of the `ts` time vector that correspond to the samples you want to delete.

`ts = delsample(ts,'Value',Time)` deletes samples from the `timeseries` object `ts`. `Time` specifies the time values that correspond to the samples you want to delete.

See Also

`addsample`

Purpose

Remove sample from `tscollection` object

Syntax

```
tsc = delsamplefromcollection(tsc,'Index',N)
tsc = delsamplefromcollection(tsc,'Value',Time)
```

Description

`tsc = delsamplefromcollection(tsc,'Index',N)` deletes samples from the `tscollection` object tsc. N specifies the indices of the tsc time vector that correspond to the samples you want to delete.

`tsc = delsamplefromcollection(tsc,'Value',Time)` deletes samples from the `tscollection` object tsc. Time specifies the time values that correspond to the samples you want to delete.

See Also

`addsampletocollection`, `tscollection`

demo

Purpose

Access product demos via Help browser

GUI Alternatives

As an alternative to the `demo` function, you can select **Help > Demos** from any desktop tool, or click the **Demos** tab when the Help browser is open.

Syntax

```
demo
demo subtopic
demo subtopic category
demo('subtopic', 'category')
```

Description

`demo` opens the **Demos** pane in the Help browser, listing demos for all installed products. The Help browser product filter preference applies to the demos listed, so installed products (and their categories) appear only if selected in the product filter. In the left pane, expand the listing for a product area (for example, MATLAB). Within that product area, expand the listing for a product or product category (for example, MATLAB Programming). Select a specific demo from the list (for example, Square Wave from Sine Waves). In the right pane, view instructions for using the demo. For more information, see the topic “Demos in the Help Browser” in the MATLAB Desktop Tools and Development Environment documentation. To run a demo from the command line, type the demo name. To run an M-file demo, open it in the Editor/Debugger and run it using **Cell > Evaluate Current Cell and Advance**, or run `echodemo` followed by the demo name.

`demo` *`subtopic`* opens the **Demos** pane in the Help browser with the specified subtopic expanded. Subtopics are `matlab`, `toolbox`, `simulink`, `blockset`, and `links and targets`. If no products in `subtopic` are installed, or if none are selected in the Help browser product filter preference, an error page appears.

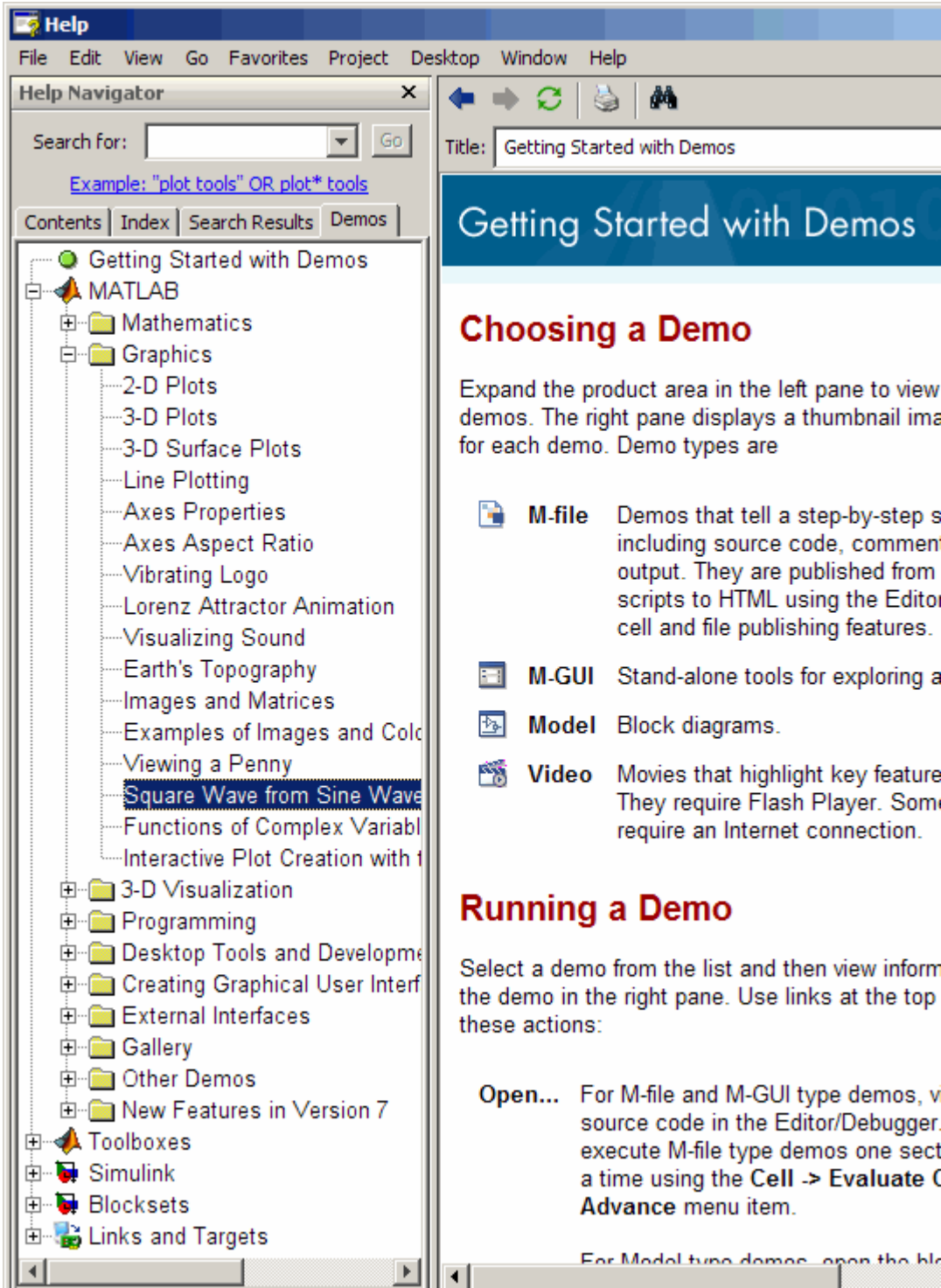
`demo` *`subtopic category`* opens the **Demos** pane in the Help browser to the specified product or category within the subtopic. The `demo` function uses the full name displayed in the **Demo** pane for `category`. If the product specified by `category` is not installed, or is not selected in the Help browser product filter preference, an error page appears.

`demo('subtopic', 'category')` is the function form of the syntax. Use this form when `subtopic` or `category` is more than one word.

This illustration shows the result of running

```
demo matlab graphics
```

and then selecting the Square Wave from Sine Waves example.


Help
File Edit View Go Favorites Project Desktop Window Help
Help Navigator
Search for:
Go
Example: "plot tools" OR plot* tools
Contents Index Search Results Demos
Getting Started with Demos
MATLAB
Mathematics
Graphics
2-D Plots
3-D Plots
3-D Surface Plots
Line Plotting
Axes Properties
Axes Aspect Ratio
Vibrating Logo
Lorenz Attractor Animation
Visualizing Sound
Earth's Topography
Images and Matrices
Viewing a Penny
Square Wave from Sine Wave
3-D Visualization
Programming
External Interfaces
Gallery
Other Demos
New Features in Version 7
Toolboxes
Simulink
Blocksets
Links and Targets
Title: Getting Started with Demos
Getting Started with Demos
Choosing a Demo
Expand the product area in the left pane to view
demos. The right pane displays a thumbnail ima
for each demo. Demo types are
M-file
Demos that tell a step-by-step s
including source code, comment
output. They are published from
scripts to HTML using the Editor
cell and file publishing features.
M-GUI
Stand-alone tools for exploring a
Model
Block diagrams.
Video
Movies that highlight key feature
They require Flash Player. Some
require an Internet connection.
Running a Demo
Select a demo from the list and then view inform
the demo in the right pane. Use links at the top
these actions:
Open...
For M-file and M-GUI type demos, v
source code in the Editor/Debugger.
execute M-file type demos one sect
a time using the Cell -> Evaluate C
Advance menu item.

Examples

Accessing Toolbox Demos

To find the demos relating to Communications Toolbox, type

```
demo toolbox communications
```

The Help browser opens to the **Demos** pane with the Toolbox subtopic expanded and with the Communications product highlighted and expanded to show the available demos.

Accessing Simulink Demos

To access the demos within Simulink, type

```
demo simulink automotive
```

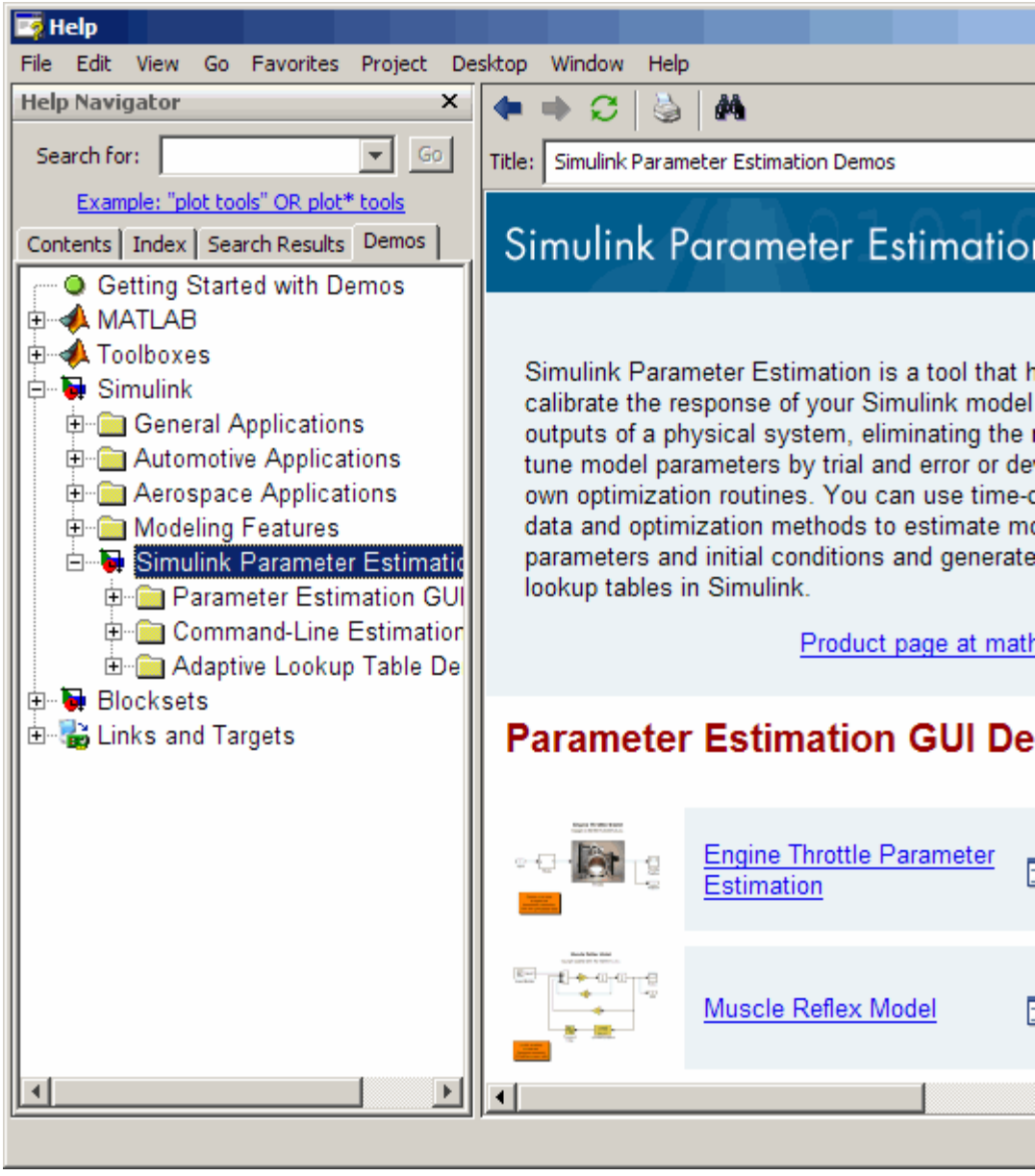
The **Demos** pane opens with the Simulink subtopic and Automotive category expanded.

Function Form of demo

To access the Simulink Parameter Estimation demos, run

```
demo('simulink', 'simulink parameter estimation')
```

which displays


Help
File Edit View Go Favorites Project Desktop Window Help
Help Navigator
Search for:
Go
Example: "plot tools" OR plot* tools
Contents Index Search Results Demos
Getting Started with Demos
MATLAB
Toolboxes
Simulink
General Applications
Automotive Applications
Aerospace Applications
Modeling Features
Simulink Parameter Estimatic
Parameter Estimation GUI
Command-Line Estimatior
Adaptive Lookup Table De
Blocksets
Links and Targets
Title: Simulink Parameter Estimation Demos
Simulink Parameter Estimatio
Simulink Parameter Estimation is a tool that h
calibrate the response of your Simulink model
outputs of a physical system, eliminating the
tune model parameters by trial and error or de
own optimization routines. You can use time-
data and optimization methods to estimate mo
parameters and initial conditions and generate
lookup tables in Simulink.
Product page at math
Parameter Estimation GUI De
Engine Throttle Parameter Estimation
Muscle Reflex Model

Running a Demo from the Command Line

Type

```
vibes
```

to run a visualization demonstration showing an animated L-shaped membrane.

Running an M-File Demo from the Command Line

Type

```
quake
```

to run an earthquake data demo. Not much appears to happen because `quake` is an M-file demo and executes from start to end without stopping. Verify this by viewing the M-file, `quake.m`, for example, by typing

```
edit quake
```

The first line, that is, the H1 line for `quake`, is

```
%% Loma Prieta Earthquake
```

The `%%` indicates that `quake` is an M-file demo. To step through the demo cell-by-cell, from the Editor/Debugger select **Cell > Evaluate Current Cell and Advance**.

Alternatively, run

```
echodemo quake
```

and the `quake` demo runs step-by-step in the Command Window.

See Also

`echodemo`, `grabcode`, `help`, `helpbrowser`, `helpwin`, `lookfor`

depdir

Purpose

List dependent directories of M-file or P-file

Syntax

```
list = depdir('file_name')
[list, prob_files, prob_sym,
   prob_strings] = depdir('file_name')
[...] = depdir('file_name1', 'file_name2',...)
```

Description

The `depdir` function lists the directories of all the functions that a specified M-file or P-file needs to operate. This function is useful for finding all the directories that need to be included with a run-time application and for determining the run-time path.

`list = depdir('file_name')` creates a cell array of strings containing the directories of all the M-files and P-files that `file_name.m` or `file_name.p` uses. This includes the second-level files that are called directly by `file_name`, as well as the third-level files that are called by the second-level files, and so on.

`[list, prob_files, prob_sym, prob_strings] = depdir('file_name')` creates three additional cell arrays containing information about any problems with the `depdir` search. `prob_files` contains filenames that `depdir` was unable to parse. `prob_sym` contains symbols that `depdir` was unable to find. `prob_strings` contains callback strings that `depdir` was unable to parse.

`[...] = depdir('file_name1', 'file_name2',...)` performs the same operation for multiple files. The dependent directories of all files are listed together in the output cell arrays.

Example

```
list = depdir('mesh')
```

See Also

`depfun`

Purpose

List dependencies of M-file or P-file

Syntax

```
list = depfun('fun')
[list, builtins, classes] = depfun('fun')
[list, builtins, classes, prob_files, prob_sym, eval_strings,
   ... called_from, java_classes] = depfun('fun')
[...] = depfun('fun1', 'fun2',...)
[...] = depfun({'fun1', 'fun2', ...})
[...] = depfun('fig_file')
[...] = depfun(..., options)
```

Description

The `depfun` function lists the paths of all files a specified M-file or P-file needs to operate.

Note It cannot be guaranteed that `depfun` will find every dependent file. Some dependent files can be hidden in callbacks, or can be constructed dynamically for evaluation, for example. Also note that the list of functions returned by `depfun` often includes extra files that would never be called if the specified function were actually evaluated.

`list = depfun('fun')` creates a cell array of strings containing the paths of all the files that function `fun` uses. This includes the second-level files that are called directly by `fun`, and the third-level files that are called by the second-level files, and so on.

Function `fun` must be on the MATLAB path, as determined by the `which` function. If the MATLAB path contains any relative directories, then files in those directories will also have a relative path.

Note If MATLAB returns a parse error for any of the input functions, or if the `prob_files` output below is nonempty, then the rest of the output of `depfun` might be incomplete. You should correct the problematic files and invoke `depfun` again.

`[list, builtins, classes] = depfun('fun')` creates three cell arrays containing information about dependent functions. `list` contains the paths of all the files that function `fun` and its subordinates use. `builtins` contains the built-in functions that `fun` and its subordinates use. `classes` contains the MATLAB classes that `fun` and its subordinates use.

`[list, builtins, classes, prob_files, prob_sym, eval_strings,... called_from, java_classes] = depfun('fun')` creates additional cell arrays or structure arrays containing information about any problems with the `depfun` search and about where the functions in `list` are invoked. The additional outputs are

- `prob_files` — Indicates which files `depfun` was unable to parse, find, or access. Parsing problems can arise from MATLAB syntax errors. `prob_files` is a structure array having these fields:
  - `name` (path to the file)
  - `listindex` (index of the file in `list`)
  - `errmsg` (problems encountered)
- *`unused`* — This is a placeholder for an output argument that is not fully implemented at this time. MATLAB returns an empty structure array for this output.
- `called_from` — Cell array of the same length as `list` that indicates which functions call other functions. This cell array is arranged so that the following statement returns all functions in function `fun` that invoke the function `list{`*`i`*`}`:

  ```
  list(called_from{i})
  ```

- `java_classes` — Cell array of Java class names used by `fun` and its subordinate functions.

`[...] = depfun('fun1', 'fun2',...)` performs the same operation for multiple functions. The dependent functions of all files are listed together in the output arrays.

`[...] = depfun({'fun1', 'fun2', ...})` performs the same operation, but on a cell array of functions. The dependent functions of all files are listed together in the output array.

`[...] = depfun('fig_file')` looks for dependent functions among the callback strings of the GUI elements that are defined in the figure file named `fig_file`.

`[...] = depfun(..., options)` modifies the `depfun` operation according to the *`options`* specified (see table below).

Option	Description
`'-all'`	Computes all possible left-side arguments and displays the results in the report(s). Only the specified arguments are returned.
`'-calltree'`	Returns a `call` list in place of a `called_from` list. This is derived from the `called_from` list as an extra step.
`'-expand'`	Includes both indices and full paths in the `call` or `called_from` list.
`'-print', 'file'`	Prints a full report to `file`.
`'-quiet'`	Displays only error and warning messages, and not a summary report.
`'-toponly'`	Examines *only* the files listed explicitly as input arguments. It does not examine the files on which they depend.
`'-verbose'`	Outputs additional internal messages.

Examples

```
list = depfun('mesh'); % Files mesh.m depends on
list = depfun('mesh','-toponly') % Files mesh.m depends on
directly
```

```
[list,builtins,classes] = depfun('gca');
```

See Also

depdir

Purpose

Matrix determinant

Syntax

```
d = det(X)
```

Description

`d = det(X)` returns the determinant of the square matrix `X`. If `X` contains only integer entries, the result `d` is also an integer.

Remarks

Using `det(X) == 0` as a test for matrix singularity is appropriate only for matrices of modest order with small integer entries. Testing singularity using `abs(det(X)) <= tolerance` is not recommended as it is difficult to choose the correct tolerance. The function `cond(X)` can check for singular and nearly singular matrices.

Algorithm

The determinant is computed from the triangular factors obtained by Gaussian elimination

```
[L,U] = lu(A)
s =  det(L)        % This is always +1 or -1
det(A) = s*prod(diag(U))
```

Examples

The statement `A = [1 2 3; 4 5 6; 7 8 9]`

produces

```
A =
     1     2     3
     4     5     6
     7     8     9
```

This happens to be a singular matrix, so `d = det(A)` produces `d = 0`. Changing `A(3,3)` with `A(3,3) = 0` turns `A` into a nonsingular matrix. Now `d = det(A)` produces `d = 27`.

See Also

`cond`, `condest`, `inv`, `lu`, `rref`

The arithmetic operators `\`, `/`

detrend

Purpose

Remove linear trends

Syntax

```
y = detrend(x)
y = detrend(x,'constant')
y = detrend(x,'linear',bp)
```

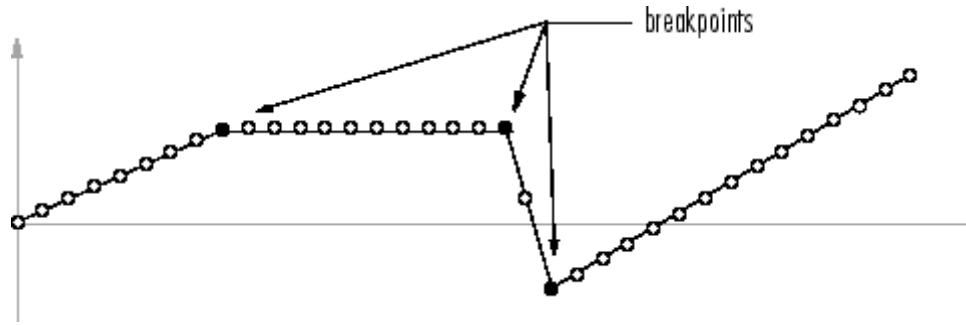
Description

`detrend` removes the mean value or linear trend from a vector or matrix, usually for FFT processing.

`y = detrend(x)` removes the best straight-line fit from vector `x` and returns it in `y`. If `x` is a matrix, `detrend` removes the trend from each column.

`y = detrend(x,'constant')` removes the mean value from vector `x` or, if `x` is a matrix, from each column of the matrix.

`y = detrend(x,'linear',bp)` removes a continuous, piecewise linear trend from vector `x` or, if `x` is a matrix, from each column of the matrix. Vector `bp` contains the indices of the breakpoints between adjacent linear segments. The breakpoint between two segments is defined as the data point that the two segments share.



`detrend(x,'linear')`, with no breakpoint vector specified, is the same as `detrend(x)`.

Example

```
sig = [0 1 -2 1 0 1 -2 1 0];      % signal with no linear trend
trend = [0 1 2 3 4 3 2 1 0];      % two-segment linear trend
```

```
x = sig+trend;                    % signal with added trend
y = detrend(x,'linear',5)         % breakpoint at 5th element

y =

   -0.0000
    1.0000
   -2.0000
    1.0000
    0.0000
    1.0000
   -2.0000
    1.0000
   -0.0000
```

Note that the breakpoint is specified to be the fifth element, which is the data point shared by the two segments.

Algorithm

`detrend` computes the least-squares fit of a straight line (or composite line for piecewise linear trends) to the data and subtracts the resulting function from the data. To obtain the equation of the straight-line fit, use `polyfit`.

See Also

`polyfit`

detrend (timeseries)

Purpose

Subtract mean or best-fit line and all NaNs from time series

Syntax

```
ts = detrend(ts1,method)
ts = detrend(ts1,Method,Index)
```

Description

`ts = detrend(ts1,method)` subtracts either a mean or a best-fit line from time-series data, usually for FFT processing. Method is a string that specifies the detrend method and has two possible values:

- `'constant'` — Subtracts the mean
- `'linear'` — Subtracts the best-fit line

`ts = detrend(ts1,Method,Index)` uses the optional `Index` integer array to specify the columns or rows to detrend. When `ts.IsTimeFirst` is `true`, `Index` specifies one or more data columns. When `ts.IsTimeFirst` is `false`, Index specifies one or more data rows.

Remarks

You cannot apply `detrend` to time-series data with more than two dimensions.

Purpose

Evaluate solution of differential equation problem

Syntax

```
sxint = deval(sol,xint)
sxint = deval(xint,sol)
sxint = deval(sol,xint,idx)
sxint = deval(xint,sol,idx)
[sxint, spxint] = deval(...)
```

Description

`sxint = deval(sol,xint)` and `sxint = deval(xint,sol)` evaluate the solution of a differential equation problem. `sol` is a structure returned by one of these solvers:

- An initial value problem solver (`ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode15i`)
- A delay differential equations solver (`dde23` or `ddesd`),
- The boundary value problem solver (`bvp4c`).

`xint` is a point or a vector of points at which you want the solution. The elements of `xint` must be in the interval `[sol.x(1),sol.x(end)]`. For each `i`, `sxint(:,i)` is the solution at `xint(i)`.

`sxint = deval(sol,xint,idx)` and `sxint = deval(xint,sol,idx)` evaluate as above but return only the solution components with indices listed in the vector `idx`.

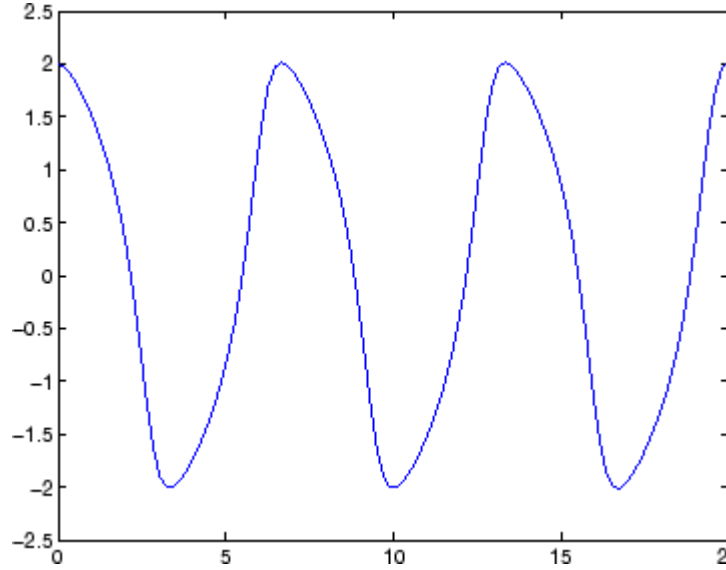
`[sxint, spxint] = deval(...)` also returns `spxint`, the value of the first derivative of the polynomial interpolating the solution.

Note For multipoint boundary value problems, the solution obtained by `bvp4c` might be discontinuous at the interfaces. For an interface point `xc`, `deval` returns the average of the limits from the left and right of `xc`. To get the limit values, set the `xint` argument of `deval` to be slightly smaller or slightly larger than `xc`.

Example

This example solves the system $y' = \mathrm{vdp1}(t, y)$ using `ode45`, and evaluates and plots the first component of the solution at 100 points in the interval `[0,20]`.

```
sol = ode45(@vdp1,[0 20],[2 0]);
x = linspace(0,20,100);
y = deval(sol,x,1);
plot(x,y);
```



See Also

ODE solvers: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode15i`

DDE solvers: `dde23`, `ddesd`

BVP solver: `bvp4c`

Purpose

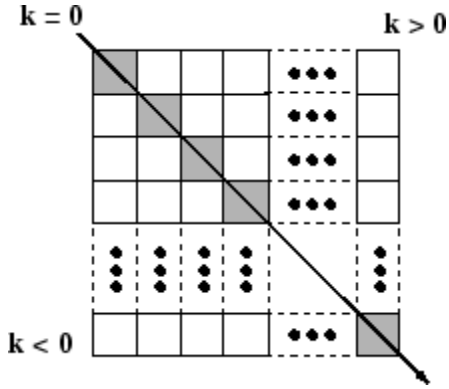
Diagonal matrices and diagonals of matrix

Syntax

```
X = diag(v,k)
X = diag(v)
v = diag(X,k)
v = diag(X)
```

Description

`X = diag(v,k)` when `v` is a vector of `n` components, returns a square matrix `X` of order `n+abs(k)`, with the elements of `v` on the `k`th diagonal. `k = 0` represents the main diagonal, `k > 0` above the main diagonal, and `k < 0` below the main diagonal.



`X = diag(v)` puts `v` on the main diagonal, same as above with `k = 0`.

`v = diag(X,k)` for matrix `X`, returns a column vector `v` formed from the elements of the `k`th diagonal of `X`.

`v = diag(X)` returns the main diagonal of `X`, same as above with `k = 0` .

Remarks

`(diag(X))` is a diagonal matrix.

`sum(diag(X))` is the trace of `X`.

`diag([])` generates an empty matrix, `([])`.

`diag(m-by-1,k)` generates a matrix of size `m+abs(k)`-by-`m+abs(k)`.

`diag(1-by-n,k)` generates a matrix of size `n+abs(k)`-by-`n+abs(k)`.

Examples

The statement

```
diag(-m:m)+diag(ones(2*m,1),1)+diag(ones(2*m,1),-1)
```

produces a tridiagonal matrix of order `2*m+1`.

See Also

`spdiags`, `tril`, `triu`, `blkdiag`

Purpose

Create and display dialog box

Syntax

```
h = dialog('PropertyName',PropertyValue,...)
```

Description

`h = dialog('PropertyName',PropertyValue,...)` returns a handle to a dialog box. This function creates a figure graphics object and sets the figure properties recommended for dialog boxes. You can specify any valid figure property value except `DockControls`, which is always `off`.

Note By default, the dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

See Also

`errordlg`, `helpdlg`, `inputdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `uiwait`, `uiresume`

"Predefined Dialog Boxes" on page 1-104 for related functions

diary

Purpose

Save session to file

Syntax

```
diary
diary('filename')
diary off
diary on
diary filename
```

Description

The `diary` function creates a log of keyboard input and the resulting text output, with some exceptions (see “Remarks” on page 2-906 for details). The output of `diary` is an ASCII file, suitable for searching in, printing, inclusion in most reports and other documents. If you do not specify `filename`, MATLAB creates a file named `diary` in the current directory.

`diary` toggles `diary` mode on and off. To see the status of `diary`, type `get(0,'Diary')`. MATLAB returns either `on` or `off` indicating the `diary` status.

`diary('filename')` writes a copy of all subsequent keyboard input and the resulting output (except it does not include graphics) to the named file, where `filename` is the full pathname or `filename` is in the current MATLAB directory. If the file already exists, output is appended to the end of the file. You cannot use a `filename` called `off` or `on`. To see the name of the `diary` file, use `get(0,'DiaryFile')`.

`diary off` suspends the diary.

`diary on` resumes diary mode using the current filename, or the default filename `diary` if none has yet been specified.

`diary filename` is the unquoted form of the syntax.

Remarks

Because the output of `diary` is plain text, the file does not exactly mirror input and output from the Command Window:

- Output does not include graphics (figure windows).
- Syntax highlighting and font preferences are not preserved.

- Hidden components of Command Window output such as hyperlink information generated with `matlab:` are shown in plain text. For example, if you enter the following statement

```
str = sprintf('%s%s', ...
   '<a href="matlab:magic(4)">', ...
   'Generate magic square</a>');
disp(str)
```

MATLAB displays

Generate magic square

However, the diary file, when viewed in a text editor, shows

```
str = sprintf('%s%s', ...
   '<a href="matlab:magic(4)">', ...
   'Generate magic square</a>');
disp(str)
<a href="matlab:magic(4)">Generate magic square</a>
```

If you view the output of diary in the Command Window, the Command Window interprets the `<a href ...>` statement and displays it as a hyperlink.

- Viewing the output of `diary` in a console window might produce different results compared to viewing `diary` output in the desktop Command Window. One example is using the `\r` option for the `fprintf` function; using the `\n` option might alleviate that problem.

See Also

`evalc`

"Command History Window" in the MATLAB Desktop Tools and Development Environment documentation

diff

Purpose

Differences and approximate derivatives

Syntax

```
Y = diff(X)
Y = diff(X,n)
Y = diff(X,n,dim)
```

Description

`Y = diff(X)` calculates differences between adjacent elements of `X`.

If `X` is a vector, then `diff(X)` returns a vector, one element shorter than `X`, of differences between adjacent elements:

```
[X(2)-X(1) X(3)-X(2) ... X(n)-X(n-1)]
```

If `X` is a matrix, then `diff(X)` returns a matrix of row differences:

```
[X(2:m,:)-X(1:m-1,:)]
```

In general, `diff(X)` returns the differences calculated along the first non-singleton (`size(X,dim) > 1`) dimension of `X`.

`Y = diff(X,n)` applies `diff` recursively `n` times, resulting in the nth difference. Thus, `diff(X,2)` is the same as `diff(diff(X))`.

`Y = diff(X,n,dim)` is the nth difference function calculated along the dimension specified by scalar `dim`. If order `n` equals or exceeds the length of dimension `dim`, `diff` returns an empty array.

Remarks

Since each iteration of `diff` reduces the length of `X` along dimension `dim`, it is possible to specify an order `n` sufficiently high to reduce `dim` to a singleton (`size(X,dim) = 1`) dimension. When this happens, `diff` continues calculating along the next nonsingleton dimension.

Examples

The quantity `diff(y)./diff(x)` is an approximate derivative.

```
x = [1 2 3 4 5];
y = diff(x)
y =
     1     1     1     1
```

```
z = diff(x,2)
z =
     0     0     0
```

Given,

```
A = rand(1,3,2,4);
```

`diff(A)` is the first-order difference along dimension `2`.

`diff(A,3,4)` is the third-order difference along dimension `4`.

See Also

`gradient`, `prod`, `sum`

diffuse

Purpose

Calculate diffuse reflectance

Syntax

```
R = diffuse(Nx,Ny,Nz,S)
```

Description

`R = diffuse(Nx,Ny,Nz,S)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` specifies the direction to the light source. You can specify these directions as three vectors`[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

Lambert's Law: `R = cos(PSI)` where `PSI` is the angle between the surface normal and light source.

See Also

`specular`, `surfnorm`, `surfl`

"Lighting as a Visualization Tool"

Purpose

Directory listing

Graphical Interface

As an alternative to the `dir` function, use the "Current Directory Browser".

Syntax

```
dir
dir name
files = dir('dirname')
```

Description

`dir` lists the files in the current working directory. Results are not sorted, but presented in the order returned by the operating system.

`dir name` lists the specified files. The `name` argument can be a pathname, filename, or can include both. You can use absolute and relative pathnames and wildcards (`*`).

`files = dir('dirname')` returns the list of files in the specified directory (or the current directory, if `dirname` is not specified) to an `m`-by-`1` structure with the fields.

Fieldname	Description	Data Type
`name`	Filename	char array
`date`	Modification date timestamp	char array
`bytes`	Number of bytes allocated to the file	double
`isdir`	`1` if name is a directory; `0` if not	logical
`datenum`	Modification date as serial date number	double

Remarks

Listing Drives

On Windows, obtain a list of drives available using the DOS `net use` command. In the Command Window, run

```
dos('net use')
```

Or run

```
[s,r] = dos('net use')
```

to return the results to the character array `r`.

DOS Filenames

The MATLAB `dir` function is consistent with the Microsoft Windows OS `dir` command in that both support short filenames generated by DOS. For example, both of the following commands are equivalent in both Windows and MATLAB:

```
dir long_matlab_mfile_name.m
   long_matlab_mfile_name.m

dir long_m~1.m
   long_matlab_m-file_name.m
```

Examples

List Directory Contents

To view the contents of the `matlab/audiovideo` directory, type

```
dir(fullfile(matlabroot, 'toolbox/matlab/audiovideo'))
```

Using Wildcard and File Extension

To view the MAT files in your current working directory that include the term `java`, type

```
dir *java*.mat
```

MATLAB returns all filenames that match this specification:

```
java_array.mat  javafrmobj.mat  testjava.mat
```

Using Relative Pathname

To view the M-files in the MATLAB `audiovideo` directory, type

```
dir(fullfile(matlabroot,'toolbox/matlab/audiovideo/*.m'))
```

MATLAB returns

```
Contents.m           aviinfo.m                render_uimgraudiotoolbar.m
audiodevinfo.m       aviread.m                sound.m
audioplayerreg.m     lin2mu.m                 soundsc.m
audiorecorderreg.m   mmcompinfo.m             wavfinfo.m
audiouniquename.m    mmfileinfo.m             wavplay.m
aufinfo.m            movie2avi.m              wavread.m
auread.m             mu2lin.m                 wavrecord.m
auwrite.m            prefspanel.m             wavwrite.m
avifinfo.m           render_fullaudiotoolbar.m
```

Returning File List to Structure

To return the list of files to the variable `av_files`, type

```
av_files = dir(fullfile(matlabroot, ...
               'toolbox/matlab/audiovideo/*.m'))
```

MATLAB returns the information in a structure array.

```
av_files =
24x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

Index into the structure to access a particular item. For example,

```
av_files(3).name
ans =
   audioplayerreg.m
```

See Also

`cd`, `copyfile`, `delete`, `fileattrib`, `filebrowser`, `fileparts`, `genpath`, `isdir`, `ls`, `matlabroot`, `mkdir`, `mfilename`, `movefile`, `rmdir`, `type`, `what`

dir (ftp)

Purpose

Directory contents on FTP server

Syntax

```
dir(f,'dirname')
d=dir(...)
```

Description

`dir(f,'dirname')` lists the files in the specified directory, `dirname`, on the FTP server `f`, where `f` was created using `ftp`. If `dirname` is unspecified, `dir` lists the files in the current directory of `f`.

`d=dir(...)` returns the results in an m-by-1 structure with the following fields for each file:

Fieldname	Description	Data Type
`name`	Filename	char array
`date`	Modification date timestamp	char array
`bytes`	Number of bytes allocated to the file	double
`isdir`	`1` if name is a directory; `0` if not	logical
`datenum`	Modification date as serial date number	char array

Examples

Connect to the MathWorks FTP server and view the contents.

```
tmw=ftp('ftp.mathworks.com');
dir(tmw)

README    incoming  matlab    outgoing  pub       pubs
```

Change to the directory `pub/pentium`.

```
cd(tmw,'pub/pentium')
```

View the contents of that directory.

```
dir(tmw)

.                     Intel_resp.txt        NYT_2.txt
..                    Intel_support.txt     NYT_Dec14.uu
Andy_Grove.txt        Intel_white.ps        New_York_Times.txt
Associated_Press.txt  MathWorks_press.txt   Nicely_1.txt
CNN.html              Mathisen.txt          Nicely_2.txt
Coe.txt               Moler_1.txt           Nicely_3.txt
Cygnus.txt            Moler_2.txt           Pratt.txt
EE_Times.txt          Moler_3.txt           README.txt
FAQ.txt               Moler_4.txt           SPSS.txt
IBM_study.txt         Moler_5.txt           Smith.txt
Intel_FAX.txt         Moler_6.ps            p87test.txt
Intel_fix.txt         Moler_7.txt           p87test.zip
Intel_replace.txt     Myths.txt             test
```

Or return the results to the structure `m`.

```
m=dir(tmw)

m =

37x1 struct array with fields:
    name
    date
    bytes
    isdir
    datanum
```

View element 17.

```
m(17)

ans =

     name: 'Moler_1.txt'
```

```
                date: '1995 Mar  27'
               bytes: 3427
               isdir: 0
             datenum: 728745
```

See Also

ftp, mkdir (ftp), rmdir (ftp)

Purpose

Display text or array

Syntax

```
disp(X)
```

Description

`disp(X)` displays an array, without printing the array name. If `X` contains a text string, the string is displayed.

Another way to display an array on the screen is to type its name, but this prints a leading `"X="` which is not always desirable.

Note that `disp` does not display empty arrays.

Examples

One use of `disp` in an M-file is to display a matrix with column labels:

```
disp('         Corn          Oats          Hay')
disp(rand(5,3))
```

which results in

```
     Corn         Oats         Hay
   0.2113       0.8474       0.2749
   0.0820       0.4524       0.8807
   0.7599       0.8075       0.6538
   0.0087       0.4832       0.4899
   0.8096       0.6135       0.7741
```

You can also use the `disp` command to display a hyperlink in the Command Window. Include the full hypertext string on a single line as input to `disp`.

```
disp('<a href = "http://www.mathworks.com">The MathWorks Web Site</a>')
```

generates this hyperlink in the Command Window:

```
The MathWorks Web Site
```

Click on this link to display The MathWorks home page in a MATLAB Web browser.

See Also format, int2str, matlabcolon, num2str, rats, sprintf

Purpose

Information about memmapfile object

Syntax

```
disp(obj)
```

Description

`disp(obj)` displays all properties and their values for `memmapfile` object `obj`.

MATLAB also displays this information when you construct a `memmapfile` object or set any of the object's property values, provided you do not terminate the command to do so with a semicolon.

Examples

Construct an object `m` of class `memmapfile`:

```
m = memmapfile('records.dat',                    ...
               'Offset', 2048,                   ...
               'Format', {                       ...
                  'int16'  [2 2] 'model';        ...
                  'uint32' [1 1] 'serialno';     ...
                  'single' [1 3] 'expenses'});
```

Use `disp` to display all the object's current properties:

```
disp(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 2048
      Format: {'int16' [2 2] 'model'
               'uint32' [1 1] 'serialno'
               'single' [1 3] 'expenses'}
      Repeat: Inf
        Data: 753x1 struct array with fields:
            model
         serialno
         expenses
```

See Also

`memmapfile`, `get(memmapfile)`

disp (MException)

Purpose

Display `MException` object

Syntax

```
disp(ME)
disp(ME.property)
```

Description

`disp(ME)` displays all properties (fields) of `MException` object `ME`.

`disp(ME.property)` displays the specified property of `MException` object `ME`.

Examples

Using the `surf` command without input arguments throws an exception. Use `disp` to display the `identifier`, `message`, `stack`, and `cause` properties of the `MException` object:

```
try
   surf
catch ME
   disp(ME)
end

 MException object with properties:

    identifier: 'MATLAB:nargchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         stack: [1x1 struct]
         cause: {}
```

Display only the `stack` property:

```
disp(ME.stack)
    file: 'X:\bat\Akernel\perfect\matlab\toolbox\matlab\
graph3d\surf.m'
    name: 'surf'
    line: 54
```

See Also

`try`, `catch`, `error`, `assert`, `MException`, `getReport(MException)`, `throw(MException)`, `rethrow(MException)`,

`throwAsCaller(MException)`, `addCause(MException)`, `isequal(MException)`, `eq(MException)`, `ne(MException)`, `last(MException)`,

disp (serial)

Purpose

Serial port object summary information

Syntax

```
obj
disp(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`obj` or `disp(obj)` displays summary information for `obj`.

Remarks

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when:

- Creating a serial port object
- Configuring property values using the dot notation

Use the display summary to quickly view the communication settings, communication state information, and information associated with read and write operations.

Example

The following commands display summary information for the serial port object `s`.

```
s = serial('COM1')
s.BaudRate = 300
s
```

Purpose

Information about timer object

Syntax

```
disp(obj)
obj
```

Description

`disp(obj)` displays summary information for the timer object, `obj`.

If `obj` is an array of timer objects, `disp` outputs a table of summary information about the timer objects in the array.

`obj`, that is, typing the object name alone, does the same as `disp(obj)`

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when

- Creating a timer object, using the `timer` function
- Configuring property values using the dot notation

Examples

The following commands display summary information for timer object `t`.

```
t = timer

Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: []
           ErrorFcn: []
           StartFcn: []
            StopFcn: []
```

This example shows the format of summary information displayed for an array of timer objects.

```
t2 = timer;
disp(timerfind)

Timer Object Array
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:     Name:
   1       singleShot      1        ''            timer-1
   2       singleShot      1        ''            timer-2
```

See Also

`timer`, `get(timer)`

Purpose

Display text or array (overloaded method)

Syntax

```
display(X)
```

Description

`display(X)` prints the value of a variable or expression, `X`. MATLAB calls `display(X)` when it interprets a variable or expression, `X`, that is not terminated by a semicolon. For example, `sin(A)` calls `display`, while `sin(A);` does not.

If `X` is an instance of a MATLAB class, then MATLAB calls the `display` method of that class, if such a method exists. If the class has no `display` method or if `X` is not an instance of a MATLAB class, then the MATLAB built-in `display` function is called.

Examples

A typical implementation of `display` calls `disp` to do most of the work and looks like this.

```
function display(X)
if isequal(get(0,'FormatSpacing'),'compact')
   disp([inputname(1) ' =']);
   disp(X)
else
   disp(' ')
   disp([inputname(1) ' =']);
   disp(' ');
   disp(X)
end
```

The expression `magic(3)`, with no terminating semicolon, calls this function as `display(magic(3))`.

```
magic(3)

ans =

     8     1     6
     3     5     7
     4     9     2
```

As an example of a class `display` method, the function below implements the `display` method for objects of the MATLAB class `polynom`.

```
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom
disp(' ');
disp([inputname(1),' = '])
disp(' ');
disp(['    ' char(p)])
disp(' ');
```

The statement

```
p = polynom([1 0 -2 -5])
```

creates a `polynom` object. Since the statement is not terminated with a semicolon, the MATLAB interpreter calls `display(p)`, resulting in the output

```
p =

    x^3 - 2*x - 5
```

See Also

`disp`, `ans`, `sprintf`, special characters

Purpose

Compute divergence of vector field

Syntax

```
div = divergence(X,Y,Z,U,V,W)
div = divergence(U,V,W)
div = divergence(X,Y,U,V)
div = divergence(U,V)
```

Description

`div = divergence(X,Y,Z,U,V,W)` computes the divergence of a 3-D vector field `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`).

`div = divergence(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`div = divergence(X,Y,U,V)` computes the divergence of a 2-D vector field `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`div = divergence(U,V)` assumes `X` and `Y` are determined by the expression

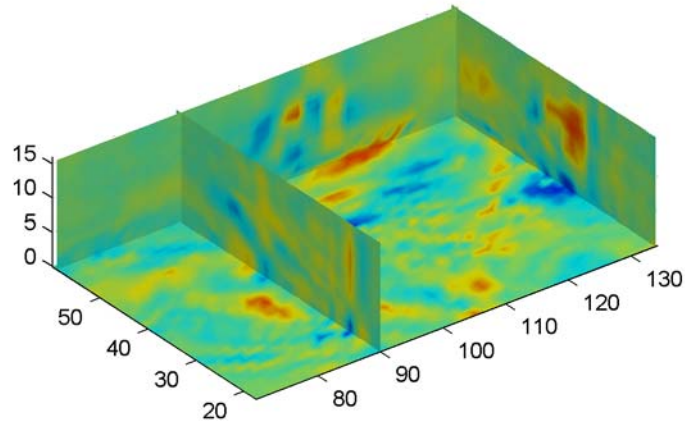
```
[X Y] = meshgrid(1:n,1:m)
```

where `[m,n] = size(U)`.

Examples

This example displays the divergence of vector volume data as slice planes, using color to indicate divergence.

```
load wind
div = divergence(x,y,z,u,v,w);
slice(x,y,z,div,[90 134],[59],[0]);
shading interp
daspect([1 1 1])
camlight
```



See Also

`streamtube`, `curl`, `isosurface`

"Volume Visualization" on page 1-102 for related functions

"Example — Displaying Divergence with Stream Tubes" for another example

Purpose

Read ASCII-delimited file of numeric data into matrix

Graphical Interface

As an alternative to `dlmread`, use the Import Wizard. To activate the Import Wizard, select **Import data** from the **File** menu.

Syntax

```
M = dlmread(filename)
M = dlmread(filename, delimiter)
M = dlmread(filename, delimiter, R, C)
M = dlmread(filename, delimiter, range)
```

Description

`M = dlmread(filename)` reads from the ASCII-delimited numeric data file `filename` to output matrix `M`. The `filename` input is a string enclosed in single quotes. The delimiter separating data elements is inferred from the formatting of the file. Comma (`,`) is the default delimiter.

`M = dlmread(filename, delimiter)` reads numeric data from the ASCII-delimited file `filename`, using the specified `delimiter`. Use `\t` to specify a tab delimiter.

Note When a delimiter is inferred from the formatting of the file, consecutive whitespaces are treated as a single delimiter. By contrast, if a delimiter is specified by the `delimiter` input, any repeated delimiter character is treated as a separate delimiter.

`M = dlmread(filename, delimiter, R, C)` reads numeric data from the ASCII-delimited file `filename`, using the specified `delimiter`. The values `R` and `C` specify the row and column where the upper left corner of the data lies in the file. `R` and `C` are zero based, so that `R=0`, `C=0` specifies the first value in the file, which is the upper left corner.

Note dlmread reads numeric data only. The file being read may contain nonnumeric data, but this nonnumeric data cannot be within the range being imported.

`M = dlmread(filename, delimiter, range)` reads the range specified by `range = [R1 C1 R2 C2]` where `(R1,C1)` is the upper left corner of the data to be read and `(R2,C2)` is the lower right corner. You can also specify the range using spreadsheet notation, as in `range = 'A1..B7'`.

Remarks

If you want to specify an `R`, `C`, or `range` input, but not a `delimiter`, set the `delimiter` argument to the empty string, (two consecutive single quotes with no spaces in between, `''`). For example,

```
M = dlmread('myfile.dat', '', 5, 2)
```

Using this syntax enables you to specify the starting row and column or range to read while having `dlmread` treat repeated whitespaces as a single delimiter.

`dlmread` fills empty delimited fields with zero. Data files having lines that end with a nonspace delimiter, such as a semicolon, produce a result that has an additional last column of zeros.

`dlmread` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. Valid forms for a complex number are

Form	Example
`–<real>–<imag>i|j`	`5.7-3.1i`
`–<imag>i|j`	`-7j`

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

Examples

Example 1

Export the 5-by-8 matrix `M` to a file, and read it with `dlmread`, first with no arguments other than the filename:

```
rand('state', 0);   M = rand(5,8);   M = floor(M * 100);
dlmwrite('myfile.txt', M, 'delimiter', '\t')

dlmread('myfile.txt')
ans =
    95    76    61    40     5    20     1    41
    23    45    79    93    35    19    74    84
    60     1    92    91    81    60    44    52
    48    82    73    41     0    27    93    20
    89    44    17    89    13    19    46    67
```

Now read a portion of the matrix by specifying the row and column of the upper left corner:

```
dlmread('myfile.txt', '\t', 2, 3)
ans =
    91    81    60    44    52
    41     0    27    93    20
    89    13    19    46    67
```

This time, read a different part of the matrix using a range specifier:

```
dlmread('myfile.txt', '\t', 'C1..G4')
ans =
    61    40     5    20     1
    79    93    35    19    74
    92    91    81    60    44
    73    41     0    27    93
```

Example 2

Export matrix `M` to a file, and then append an additional matrix to the file that is offset one row below the first:

```
M = magic(3);
```

```
dlmwrite('myfile.txt', [M*5 M/5], ' ')

dlmwrite('myfile.txt', rand(3), '-append', ...
   'roffset', 1, 'delimiter', ' ')

type myfile.txt

80 10 15 65 3.2 0.4 0.6 2.6
25 55 50 40 1 2.2 2 1.6
45 35 30 60 1.8 1.4 1.2 2.4
20 70 75 5 0.8 2.8 3 0.2

0.99008 0.49831 0.32004
0.78886 0.21396 0.9601
0.43866 0.64349 0.72663
```

When `dlmread` imports these two matrices from the file, it pads the smaller matrix with zeros:

```
dlmread('myfile.txt')
   40.0000    5.0000   30.0000   1.6000   0.2000   1.2000
   15.0000   25.0000   35.0000   0.6000   1.0000   1.4000
   20.0000   45.0000   10.0000   0.8000   1.8000   0.4000
    0.6038    0.0153    0.9318        0        0        0
    0.2722    0.7468    0.4660        0        0        0
    0.1988    0.4451    0.4187        0        0        0
```

See Also

`dlmwrite`, `textscan`, `csvread`, `csvwrite`, `wk1read`, `wk1write`

Purpose

Write matrix to ASCII-delimited file

Syntax

```
dlmwrite(filename, M)
dlmwrite(filename, M, 'D')
dlmwrite(filename, M, 'D', R, C)
dlmwrite(filename, M, 'attrib1', value1, 'attrib2', value2,
   ...)
dlmwrite(filename, M, '-append')
dlmwrite(filename, M, '-append', attribute-value list)
```

Description

`dlmwrite(filename, M)` writes matrix `M` into an ASCII format file using the default delimiter (`,`) to separate matrix elements. The data is written starting at the first column of the first row in the destination file, `filename`. The `filename` input is a string enclosed in single quotes.

`dlmwrite(filename, M, 'D')` writes matrix `M` into an ASCII format file, using delimiter `D` to separate matrix elements. The data is written starting at the first column of the first row in the destination file, `filename`. A comma (`,`) is the default delimiter. Use `\t` to produce tab-delimited files.

`dlmwrite(filename, M, 'D', R, C)` writes matrix `M` into an ASCII format file, using delimiter `D` to separate matrix elements. The data is written starting at row `R` and column `C` in the destination file, `filename`. `R` and `C` are zero based, so that `R=0`, `C=0` specifies the first value in the file, which is the upper left corner.

`dlmwrite(filename, M, 'attrib1', value1, 'attrib2', value2, ...)` is an alternate syntax to those shown above, in which you specify any number of attribute-value pairs in any order in the argument list. Each attribute must be immediately followed by a corresponding value (see the table below).

Attribute	Value
`delimiter`	Delimiter string to be used in separating matrix elements

Attribute	Value
newline	Character(s) to use in terminating each line (see table below)
roffset	Offset, in rows, from the top of the destination file to where matrix data is to be written. Offset is zero based.
coffset	Offset, in columns, from the left side of the destination file to where matrix data is to be written. Offset is zero based.
precision	Numeric precision to use in writing data to the file. Specify the number of significant digits or a C-style format string starting in %, such as `'%10.5f'`.

This table shows which values you can use when setting the **newline** attribute.

Line Terminator	Description
'pc'	PC terminator (implies carriage return/line feed (CR/LF))
'unix'	UNIX terminator (implies line feed (LF))

`dlmwrite(filename, M, '-append')` appends the matrix to the file. If you do not specify `'-append'`, `dlmwrite` overwrites any existing data in the file.

`dlmwrite(filename, M, '-append', attribute-value list)` is the same as the syntax shown above, but accepts a list of `attribute-value` pairs. You can place the `'-append'` flag in the argument list anywhere between `attribute-value` pairs, but not in between an `attribute` and its `value`.

Remarks

The resulting file is readable by spreadsheet programs.

Examples

Example 1

Export matrix `M` to a file delimited by the tab character and using a precision of six significant digits:

```
dlmwrite('myfile.txt', M, 'delimiter', '\t', ...
         'precision', 6)
type myfile.txt

0.893898      0.284409      0.582792      0.432907
0.199138      0.469224      0.423496      0.22595
0.298723      0.0647811     0.515512      0.579807
0.661443      0.988335      0.333951      0.760365
```

Example 2

Export matrix `M` to a file using a precision of six decimal places and the conventional line terminator for the PC platform:

```
dlmwrite('myfile.txt', m, 'precision', '%.6f', ...
         'newline', 'pc')
type myfile.txt

16.000000,2.000000,3.000000,13.000000
5.000000,11.000000,10.000000,8.000000
9.000000,7.000000,6.000000,12.000000
4.000000,14.000000,15.000000,1.000000
```

Example 3

Export matrix `M` to a file, and then append an additional matrix to the file that is offset one row below the first:

```
M = magic(3);
dlmwrite('myfile.txt', [M*5 M/5], ' ')

dlmwrite('myfile.txt', rand(3), '-append', ...
   'roffset', 1, 'delimiter', ' ')

type myfile.txt
```

```
80 10 15 65 3.2 0.4 0.6 2.6
25 55 50 40 1 2.2 2 1.6
45 35 30 60 1.8 1.4 1.2 2.4
20 70 75 5 0.8 2.8 3 0.2

0.99008 0.49831 0.32004
0.78886 0.21396 0.9601
0.43866 0.64349 0.72663
```

When `dlmread` imports these two matrices from the file, it pads the smaller matrix with zeros:

```
dlmread('myfile.txt')
   40.0000    5.0000   30.0000   1.6000   0.2000   1.2000
   15.0000   25.0000   35.0000   0.6000   1.0000   1.4000
   20.0000   45.0000   10.0000   0.8000   1.8000   0.4000
    0.6038    0.0153    0.9318        0        0        0
    0.2722    0.7468    0.4660        0        0        0
    0.1988    0.4451    0.4187        0        0        0
```

See Also

`dlmread`, `csvwrite`, `csvread`, `wk1write`, `wk1read`

Purpose

Dulmage-Mendelsohn decomposition

Syntax

```
p = dmperm(A)
[p,q,r,s,cc,rr] = dmperm(A)
```

Description

`p = dmperm(A)` finds a vector `p` such that `p(j) = i` if column `j` is matched to row `i`, or zero if column `j` is unmatched. If `A` is a square matrix with full structural rank, `p` is a maximum matching row permutation and `A(p,:)` has a zero-free diagonal. The structural rank of `A` is `sprank(A) = sum(p>0)`.

`[p,q,r,s,cc,rr] = dmperm(A)` where `A` need not be square or full structural rank, finds the Dulmage-Mendelsohn decomposition of `A`. `p` and `q` are row and column permutation vectors, respectively, such that `A(p,q)` has a block upper triangular form. `r` and `s` are index vectors indicating the block boundaries for the fine decomposition. `cc` and `rr` are vectors of length five indicating the block boundaries of the coarse decomposition.

`C = A(p,q)` is split into a 4-by-4 set of coarse blocks:

```
A11 A12 A13 A14
0    0  A23 A24
0    0   0  A34
0    0   0  A44
```

where `A12`, `A23`, and `A34` are square with zero-free diagonals. The columns of `A11` are the unmatched columns, and the rows of `A44` are the unmatched rows. Any of these blocks can be empty. In the coarse decomposition, the `(i,j)th` block is `C(rr(i):rr(i+1)-1,cc(j):cc(j+1)-1)`. For a linear system,

- `[A11 A12]` is the underdetermined part of the system—it is always rectangular and with more columns and rows, or 0-by-0,
- `A23` is the well-determined part of the system—it is always square, and

- `[A34 ; A44]` is the overdetermined part of the system—it is always rectangular with more rows than columns, or `0`-by-`0`.

The structural rank of `A` is `sprank(A) = rr(4)-1`, which is an upper bound on the numerical rank of `A`. `sprank(A) = rank(full(sprand(A)))` with probability 1 in exact arithmetic.

The `A23` submatrix is further subdivided into block upper triangular form via the fine decomposition (the strongly connected components of `A23`). If `A` is square and structurally nonsingular, `A23` is the entire matrix.

`C(r(i):r(i+1)-1,s(j):s(j+1)-1)` is the `(i,j)`th block of the fine decomposition. The `(1,1)` block is the rectangular block `[A11 A12]`, unless this block is `0`-by-`0`. The `(b,b)` block is the rectangular block `[A34 ; A44]`, unless this block is `0`-by-`0`, where `b = length(r)-1`. All other blocks of the form `C(r(i):r(i+1)-1,s(i):s(i+1)-1)` are diagonal blocks of `A23`, and are square with a zero-free diagonal.

Remarks

If `A` is a reducible matrix, the linear system $Ax=b$ can be solved by permuting `A` to a block upper triangular form, with irreducible diagonal blocks, and then performing block backsubstitution. Only the diagonal blocks of the permuted matrix need to be factored, saving fill and arithmetic in the blocks above the diagonal.

In graph theoretic terms, `dmperm` finds a maximum-size matching in the bipartite graph of `A`, and the diagonal blocks of `A(p,q)` correspond to the strong Hall components of that graph. The output of `dmperm` can also be used to find the connected or strongly connected components of an undirected or directed graph. For more information see Pothen and Fan [1].

`dmperm` uses CSparse [2].

References

[1] Pothen, Alex and Chin-Ju Fan “Computing the Block Triangular Form of a Sparse Matrix” *ACM Transactions on Mathematical Software* Vol 16, No. 4 Dec. 1990, pp. 303-324.

[2] T.A. Davis *Direct Methods for for Sparse Linear Systems*. SIAM, Philadelphia: 2006. Software available at:http://www.cise.ufl.edu/research/sparse/CSparse.

See Also

`sprank`

Purpose

Reference page in Help browser

GUI Alternatives

As an alternative to the `doc` function, use the Help browser **Search for** field. Type the function name and click **Go**.

Syntax

```
doc
doc functionname
doc toolboxname
doc toolboxname/functionname
doc classname.methodname
```

Description

`doc` opens the Help browser, if it is not already running, or brings the window to the top, displaying the **Contents** pane when the Help browser is already open.

`doc functionname` displays the reference page for the MATLAB function `functionname` in the Help browser. For example, you are looking at the reference page for the `doc` function. Here `functionname` can be a function, block, property, method, or object. If `functionname` is overloaded, that is, if `functionname` appears in multiple directories on the MATLAB search path, `doc` displays the reference page for the first `functionname` on the search path and displays a hyperlinked list of the other functions and their directories in the MATLAB Command Window. Overloaded functions within the same product are not listed — use the `overloaddirectory` form of the syntax. If a reference page for `functionname` does not exist, `doc` displays its M-file help in the Help browser. The `doc` function is intended only for help files supplied by The MathWorks, and is not supported for use with HTML files you create yourself.

`doc toolboxname` displays the roadmap page for `toolboxname` in the Help browser, which provides a summary of the most pertinent documentation for that product.

`doc toolboxname/functionname` displays the reference page for the `functionname` that belongs to the specified `toolboxname`, in the Help browser. This is useful for overloaded functions.

`doc classname.methodname` displays the reference page for the `methodname` that is a member of `classname`.

Note If there is a function called `name` as well as a toolbox called `name`, the roadmap page for the toolbox called `name` displays. To see the reference page for the function called `name`, use `doc toolboxname/name`, where `toolboxname` is the name of the toolbox in which the function `name` resides. For example, `doc matlab` displays the roadmap page for MATLAB (that is, the `matlab` toolbox), while `doc matlab/matlabunix` displays the reference page for the `matlab` startup function for UNIX, which is in MATLAB.

Examples

Type `doc abs` to display the reference page for the `abs` function. If Simulink and Signal Processing Toolbox are installed and on the search path, the Command Window lists hyperlinks for the `abs` function in those products:

```
doc signal/abs
doc simulink/abs
```

Type `doc signal/abs` to display the reference page for the `abs` function in Signal Processing Toolbox.

Type `doc signal` to display the roadmap page for Signal Processing Toolbox.

Type `doc serial.get` to display the reference page for the `get` method located in the `serial` directory of MATLAB. This syntax is required because there is at least one other `get` function in MATLAB.

See Also

`docopt`, `docsearch`, `help`, `helpbrowser`, `lookfor`, `type`, `web`

"Help for Using MATLAB" in the MATLAB Desktop Tools and Development Environment documentation.

docopt

Purpose

Web browser for UNIX platforms

Syntax

```
docopt
doccmd = docopt
```

Description

`docopt` displays the Web browser used with MATLAB on non-Macintosh UNIX platforms, with the default being `netscape` (for Netscape). For non-Macintosh UNIX platforms, you can modify the `docopt.m` file to specify the Web browser MATLAB uses. The Web browser is used with the `web` function and its `-browser` option. It is also used for links to external Web sites from the Help.

`doccmd = docopt` returns a string containing the command that `web -browser` uses to invoke a Web browser.

To change the browser, edit the `docopt.m` file and change line 51. For example,

```
50 elseif isunix                    % UNIX
51 %   doccmd = '';
```

Remove the comment symbol. In the quote, enter the command that starts your Web browser, and save the file. For example,

```
51     doccmd = 'mozilla';
```

specifies Mozilla as the Web browser MATLAB uses.

See Also

`doc`, `edit`, `helpbrowser`, `web`

Purpose

Open Help browser **Search** pane and search for specified term

GUI Alternatives

As an alternative to the `docsearch` function, select **Desktop > Help**, type in the **Search for** field, and click **Go**.

Syntax

```
docsearch
docsearch word
docsearch('word1 word2 ...')
docsearch('"word1 word2" ...')
docsearch('wo*rd ...')
docsearch('word1 word2 BOOLEANOP word3')
```

Description

`docsearch` opens the Help browser to the **Search Results** pane, or if the Help browser is already open to that pane, brings it to the top.

`docsearch word` executes a Help browser full-text search for `word`, displaying results in the Help browser **Search Results** pane. If `word` is a `functionname` or `blockname`, the first entry in **Search Results** is the reference page, or reference pages for overloaded functions.

`docsearch('word1 word2 ...')` executes a Help browser full-text search for pages containing `word1` and `word2` and any other specified words, displaying results in the Help browser **Search Results** pane.

`docsearch('"word1 word2" ...')` executes a Help browser full-text search for pages containing the exact phrase `word1 word2` and any other specified words, displaying results in the Help browser **Search Results** pane.

`docsearch('wo*rd ...')` executes a Help browser full-text search for pages containing words that begin with `wo` and end with `rd`, and any other specified words, displaying results in the Help browser **Search Results** pane. This is also called a wildcard or partial word search. You can use a wildcard symbol (`*`) multiple times within a word. You cannot use the wildcard symbol within an exact phrase. You must use at least two letters or digits with a wildcard symbol.

`docsearch('word1 word2 BOOLEANOP word3')` executes a Help browser full-text search for the term `word1 word2 BOOLEANOP word3`,

where BOOLEANOP is a Boolean operator (AND, NOT, OR) used to refine the search. docsearch evaluates NOTs first, then ORs, and finally ANDs. Results display in the Help browser **Search Results** pane.

Examples

`docsearch plot` finds all pages that contain the word `plot`.

`docsearch('plot tools')` finds all pages that contain the words `plot` and `tools` anywhere in the page.

`docsearch('"plot tools"')` finds all pages that contain the exact phrase `plot tools`.

`docsearch('plot* tools')` finds all pages that contain the word `tools` and the word `plot` or variations of `plot`, such as `plotting`, and `plots`.

`docsearch('"plot tools" NOT "time series"')` finds all pages that contain the exact phrase `plot tools`, but only if the pages do not contain the exact phrase `time series`.

See Also

`builddocsearchdb`, `doc`, `helpbrowser`

"Search Documentation and Demos with the Help Browser" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Execute DOS command and return result

Syntax

```
dos command
status = dos('command')
[status,result] = dos('command')
[status,result] = dos('command','-echo')
```

Description

`dos command` calls upon the shell to execute the given command for Windows systems.

`status = dos('command')` returns completion status to the `status` variable.

`[status,result] = dos('command')` in addition to completion status, returns the result of the command to the `result` variable.

`[status,result] = dos('command','-echo')` forces the output to the Command Window, even though it is also being assigned into a variable.

Both console (DOS) programs and Windows programs may be executed, but the syntax causes different results based on the type of programs. Console programs have `stdout` and their output is returned to the result variable. They are always run in an iconified DOS or Command Prompt Window except as noted below. Console programs never execute in the background. Also, MATLAB will always wait for the `stdout` pipe to close before continuing execution. Windows programs may be executed in the background as they have no `stdout`.

The ampersand, `&`, character has special meaning. For console programs this causes the console to open. Omitting this character will cause console programs to run iconically. For Windows programs, appending this character will cause the application to run in the background. MATLAB will continue processing.

Note Running `dos` with a `command` that relies upon the current directory will fail when the current directory is specified using a UNC pathname. This is because DOS does not support UNC pathnames. In that event, MATLAB returns this error: `??? Error using ==> dos DOS commands may not be executed when the current directory is a UNC pathname`. To work around this limitation, change the directory to a mapped drive prior to running `dos` or a function that calls `dos`.

Examples

The following example performs a directory listing, returning a zero (success) in `s` and the string containing the listing in `w`.

```
[s, w] = dos('dir');
```

To open the DOS 5.0 editor in a DOS window

```
dos('edit &')
```

To open the notepad editor and return control immediately to MATLAB

```
dos('notepad file.m &')
```

The next example returns a one in `s` and an error message in `w` because `foo` is not a valid shell command.

```
[s, w] = dos('foo')
```

This example echoes the results of the `dir` command to the Command Window as it executes as well as assigning the results to `w`.

```
[s, w] = dos('dir', '-echo');
```

See Also

`!` (exclamation point), `perl`, `system`, `unix`, `winopen`

"Running External Programs" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Vector dot product

Syntax

```
C = dot(A,B)
C = dot(A,B,dim)
```

Description

`C = dot(A,B)` returns the scalar product of the vectors `A` and `B`. `A` and `B` must be vectors of the same length. When `A` and `B` are both column vectors, `dot(A,B)` is the same as `A'*B`.

For multidimensional arrays `A` and `B`, `dot` returns the scalar product along the first non-singleton dimension of `A` and `B`. `A` and `B` must have the same size.

`C = dot(A,B,dim)` returns the scalar product of `A` and `B` in the dimension `dim`.

Examples

The dot product of two vectors is calculated as shown:

```
a = [1 2 3]; b = [4 5 6];
c = dot(a,b)

c =
      32
```

See Also

`cross`

double

Purpose

Convert to double precision

Syntax

```
double(x)
```

Description

`double(x)` returns the double-precision value for `X`. If `X` is already a double-precision array, `double` has no effect.

Remarks

`double` is called for the expressions in `for`, `if`, and `while` loops if the expression isn't already double-precision. `double` should be overloaded for any object when it makes sense to convert it to a double-precision value.

Purpose

Drag rectangles with mouse

Syntax

```
[finalrect] = dragrect(initialrect)
[finalrect] = dragrect(initialrect,stepsize)
```

Description

`[finalrect] = dragrect(initialrect)` tracks one or more rectangles anywhere on the screen. The n-by-4 matrix `initialrect` defines the rectangles. Each row of `initialrect` must contain the initial rectangle position as `[left bottom width height]` values. `dragrect` returns the final position of the rectangles in `finalrect`.

`[finalrect] = dragrect(initialrect,stepsize)` moves the rectangles in increments of `stepsize`. The lower left corner of the first rectangle is constrained to a grid of size equal to `stepsize` starting at the lower left corner of the figure, and all other rectangles maintain their original offset from the first rectangle.

`[finalrect] = dragrect(...)` returns the final positions of the rectangles when the mouse button is released. The default step size is `1`.

Remarks

`dragrect` returns immediately if a mouse button is not currently pressed. Use `dragrect` in a `ButtonDownFcn`, or from the command line in conjunction with `waitforbuttonpress`, to ensure that the mouse button is down when `dragrect` is called. `dragrect` returns when you release the mouse button.

If the drag ends over a figure window, the positions of the rectangles are returned in that figure's coordinate system. If the drag ends over a part of the screen not contained within a figure window, the rectangles are returned in the coordinate system of the figure over which the drag began.

Note You cannot use normalized figure units with `dragrect`.

Example

Drag a rectangle that is 50 pixels wide and 100 pixels in height.

```
waitforbuttonpress
point1 = get(gcf,'CurrentPoint') % button down detected
rect = [point1(1,1) point1(1,2) 50 100]
[r2] = dragrect(rect)
```

See Also

`rbbox`, `waitforbuttonpress`

"Selecting Region of Interest" on page 1-101 for related functions

Purpose

Flushes event queue and updates figure window

Syntax

```
drawnow
drawnow expose
drawnow update
```

Description

`drawnow` causes figure windows and their children to update and flushes the system event queue. Any callbacks generated by incoming events (e.g. mouse or key events) are dispatched before `drawnow` returns.

`drawnow expose` causes only graphics objects to refresh, if needed. It does not allow callbacks to execute and does not process other events in the queue.

`drawnow update` causes only non-graphics objects to refresh, if needed. It does not allow callbacks to execute and does not process other events in the queue.

You can combine the `expose` and `update` options to obtain both effects.

```
drawnow expose update
```

Other Events That Cause Event Queue Processing

Other events that cause MATLAB to flush the event queue and draw the figure includes:

- Returning to the MATLAB prompt
- Executing the following functions
  - `figure`
  - `getframe`
  - `input`
  - `keyboard`
  - `pause`
- Functions that wait for user input (i.e., `waitforbuttonpress`, `waitfor`, `ginput`)

Examples

Using drawnow in a loop causes the display to update while the loop executes:

```
t = 0:pi/20:2*pi;
y = exp(sin(t));
h = plot(t,y,'YDataSource','y');
for k = 1:.1:10
 y = exp(sin(t.*k));
 refreshdata(h,'caller') % Evaluate y in the function workspace
 drawnow; pause(.1)
end
```

See Also

waitfor, waitforbuttonpress

"Figure Windows" on page 1-95 for related functions

Purpose

Search Delaunay triangulation for nearest point

Syntax

```
K = dsearch(x,y,TRI,xi,yi)
K = dsearch(x,y,TRI,xi,yi,S)
```

Description

`K = dsearch(x,y,TRI,xi,yi)` returns the index into `x` and `y` of the nearest point to the point (`xi`,`yi`). `dsearch` requires a triangulation `TRI` of the points `x`,`y` obtained using `delaunay`. If `xi` and `yi` are vectors, `K` is a vector of the same size.

`K = dsearch(x,y,TRI,xi,yi,S)` uses the sparse matrix `S` instead of computing it each time:

```
S = sparse(TRI(:,[1 1 2 2 3 3]),TRI(:,[2 3 1 3 1 2]),1,nxy,nxy)
```

where `nxy = prod(size(x))`.

See Also

`delaunay`, `tsearch`, `voronoi`

dsearchn

Purpose

N-D nearest point search

Syntax

```
k = dsearchn(X,T,XI)
k = dsearchn(X,T,XI,outval)
k = dsearchn(X,XI)
[k,d] = dsearchn(X,...)
```

Description

`k = dsearchn(X,T,XI)` returns the indices `k` of the closest points in `X` for each point in `XI`. `X` is an `m`-by-`n` matrix representing `m` points in n-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in n-dimensional space. `T` is a `numt`-by-`n+1` matrix, a tessellation of the data `X` generated by `delaunayn`. The output `k` is a column vector of length `p`.

`k = dsearchn(X,T,XI,outval)` returns the indices `k` of the closest points in `X` for each point in `XI`, unless a point is outside the convex hull. If `XI(J,:)` is outside the convex hull, then `K(J)` is assigned `outval`, a scalar double. `Inf` is often used for `outval`. If `outval` is `[]`, then `k` is the same as in the case `k = dsearchn(X,T,XI)`.

`k = dsearchn(X,XI)` performs the search without using a tessellation. With large `X` and small `XI`, this approach is faster and uses much less memory.

`[k,d] = dsearchn(X,...)` also returns the distances `d` to the closest points. `d` is a column vector of length `p`.

Algorithm

`dsearchn` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`tsearch`, `dsearch`, `tsearchn`, `griddatan`, `delaunayn`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/`.

Purpose

Echo M-files during execution

Syntax

```
echo on
echo off
echo
echo fcnname on
echo fcnname off
echo fcnname
echo on all
echo off all
```

Description

The `echo` command controls the echoing of M-files during execution. Normally, the commands in M-files are not displayed on the screen during execution. Command echoing is useful for debugging or for demonstrations, allowing the commands to be viewed as they execute.

The `echo` command behaves in a slightly different manner for script files and function files. For script files, the use of `echo` is simple; echoing can be either `on` or `off`, in which case any script used is affected.

`echo on`	Turns on the echoing of commands in all script files
`echo off`	Turns off the echoing of commands in all script files
`echo`	Toggles the echo state

With function files, the use of `echo` is more complicated. If `echo` is enabled on a function file, the file is interpreted, rather than compiled. Each input line is then displayed as it is executed. Since this results in inefficient execution, use `echo` only for debugging.

`echo fcnname on`	Turns on echoing of the named function file
`echo fcnname off`	Turns off echoing of the named function file
`echo fcnname`	Toggles the echo state of the named function file

`echo on all`	Sets echoing on for all function files
`echo off all`	Sets echoing off for all function files

See Also

`function`

Purpose

Run M-file demo step-by-step in Command Window

GUI Alternatives

As an alternative to the `echodemo` function, select the demo in the Help browser **Demos** tab and click the **Run in the Command Window** link.

Syntax

```
echodemo filename
echodemo('filename', cellindex)
```

Description

`echodemo filename` runs the M-file demo `filename` step-by-step in the Command Window. At each step, follow links in the Command Window to proceed. Depending on the size of the Command Window, you might have to scroll up to see the links. The script `filename` was created in the Editor/Debugger using cells. (The associated HTML demo file for `filename` that appears in the Help browser **Demos** pane was created using the MATLAB cell publishing feature.) The link to `filename` also shows the current cell number, `n`, and the total number of cells, `m`, as `n/m`, and when clicked, opens `filename` in the Editor/Debugger. To end the demo, click the **Stop** link.

`echodemo('filename', cellindex)` runs the M-file type demo `filename`, starting with the cell number specified by `cellindex`. Because steps prior to `cellindex` are not run, this statement might produce an error or unexpected result, depending on the demo.

Note M-file demos run as scripts. Therefore, the variables are part of the base workspace, which could result in problems if you have any variables of the same name. For more information, see “Running Demos and Base Workspace Variables” in the Desktop Tools and Development Environment documentation.

Examples

`echodemo quake` runs the MATLAB Loma Prieta Earthquake demo.

`echodemo ('quake', 6)` runs the MATLAB Loma Prieta Earthquake demo, starting at cell 6.

`echodemo ('intro', 3)` produces an error because cell 3 of the MATLAB demo `intro` requires data created when cells 1 and 2 run.

See Also

`demo`, `helpbrowser`

Purpose

Edit or create M-file

GUI Alternatives

As an alternative to the `edit` function, select **File > New** or **Open** in the MATLAB desktop or any desktop tool.

Syntax

```
edit
edit fun.m
edit file.ext
edit fun1 fun2 fun3 ...
edit class/fun
edit private/fun
edit class/private/fun
edit('my file.m')
```

Description

`edit` opens a new editor window.

`edit fun.m` opens the M-file `fun.m` in the default editor. Note that `fun.m` can be a MATLAB `partialpath` or a complete path. If `fun.m` does not exist, a prompt appears asking if you want to create a new file titled `fun.m`. After you click **Yes**, the Editor/Debugger creates a blank file titled `fun.m`. If you do not want the prompt to appear in this situation, select that check box in the prompt. Then when you type `edit fun.m`, where `fun.m` did not previously exist, a new file called `fun.m` is automatically opened in the Editor/Debugger. To make the prompt appear, specify it in preferences for Prompt.

`edit file.ext` opens the specified file.

`edit fun1 fun2 fun3 ...` opens `fun1.m`, `fun2.m`, `fun3.m`, and so on, in the default editor.

`edit class/fun`, or `edit private/fun`, or `edit class/private/fun` edit a method, private function, or private method for the class named `class`.

`edit('my file.m')` opens the M-file `my file.m` in the default editor. This form of the `edit` function is useful when a filename contains a space, for example, because you cannot use the command form in such a case.

Remarks

To specify the default editor for MATLAB, select **Preferences** from the **File** menu. On the **Editor/Debugger** pane, select **MATLAB editor** or specify another.

UNIX Users

If you run MATLAB with the `-nodisplay` startup option, or run without the `DISPLAY` environment variable set, `edit` uses the `External Editor` command. It does not use the MATLAB Editor/Debugger, but instead uses the default editor defined for your system in *`matlabroot`*`/X11/app-defaults/Matlab`.

You can specify the editor that the `edit` function uses or specify editor options by adding the following line to your own `.Xdefaults` file, located in `~home`:

```
matlab*externalEditorCommand: $EDITOR -option $FILE
```

where

- `$EDITOR` is the name of your default editor, for example, `emacs`; leaving it as `$EDITOR` means your default system editor will be used.
- `-option` is a valid option flag you can include for the specified editor.
- **`$FILE`** means the filename you type with the `edit` command will open in the specified editor.

For example,

```
emacs $FILE
```

means that when you type `edit foo`, the file `foo` will open in the `emacs` editor.

After adding the line to your `.Xdefaults` file, you must run the following before starting MATLAB:

```
xrdb -merge ~home/.Xdefaults
```

See Also

`open`, `type`

Purpose

Find eigenvalues and eigenvectors

Syntax

```
d = eig(A)
d = eig(A,B)
[V,D] = eig(A)
[V,D] = eig(A,'nobalance')
[V,D] = eig(A,B)
[V,D] = eig(A,B,flag)
```

Description

`d = eig(A)` returns a vector of the eigenvalues of matrix `A`.

`d = eig(A,B)` returns a vector containing the generalized eigenvalues, if `A` and `B` are square matrices.

Note If `S` is sparse and symmetric, you can use `d = eig(S)` to returns the eigenvalues of `S`. If `S` is sparse but not symmetric, or if you want to return the eigenvectors of `S`, use the function `eigs` instead of `eig`.

`[V,D] = eig(A)` produces matrices of eigenvalues (`D`) and eigenvectors (`V`) of matrix `A`, so that `A*V = V*D`. Matrix `D` is the *canonical form* of `A` — a diagonal matrix with `A`'s eigenvalues on the main diagonal. Matrix `V` is the *modal matrix* — its columns are the eigenvectors of `A`.

If `W` is a matrix such that `W'*A = D*W'`, the columns of `W` are the *left eigenvectors* of `A`. Use `[W,D] = eig(A.'); W = conj(W)` to compute the left eigenvectors.

`[V,D] = eig(A,'nobalance')` finds eigenvalues and eigenvectors without a preliminary balancing step. This may give more accurate results for certain problems with unusual scaling. Ordinarily, balancing improves the conditioning of the input matrix, enabling more accurate computation of the eigenvectors and eigenvalues. However, if a matrix contains small elements that are really due to roundoff error, balancing may scale them up to make them as significant as the other elements of the original matrix, leading to incorrect eigenvectors. Use the

nobalance option in this event. See the balance function for more details.

[V,D] = eig(A,B) produces a diagonal matrix D of generalized eigenvalues and a full matrix V whose columns are the corresponding eigenvectors so that A*V = B*V*D .

[V,D] = eig(A,B,*flag*) specifies the algorithm used to compute eigenvalues and eigenvectors. *flag* can be:

'chol'	Computes the generalized eigenvalues of A and B using the Cholesky factorization of B. This is the default for symmetric (Hermitian) A and symmetric (Hermitian) positive definite B.
'qz'	Ignores the symmetry, if any, and uses the QZ algorithm as it would for nonsymmetric (non-Hermitian) A and B.

Note For eig(A), the eigenvectors are scaled so that the norm of each is 1.0. For eig(A,B), eig(A,'nobalance'), and eig(A,B,flag), the eigenvectors are not normalized.

Also note that if A is symmetric, eig(A,'nobalance') ignores the nobalance option since A is already balanced.

Remarks

The eigenvalue problem is to determine the nontrivial solutions of the equation

$$Ax = \lambda x$$

where A is an n-by-n matrix, x is a length n column vector, and λ is a scalar. The n values of λ that satisfy the equation are the *eigenvalues*, and the corresponding values of x are the *right eigenvectors*. In MATLAB, the function eig solves for the eigenvalues λ, and optionally the eigenvectors x.

The *generalized* eigenvalue problem is to determine the nontrivial solutions of the equation

$$Ax = \lambda Bx$$

where both A and B are n-by-n matrices and λ is a scalar. The values of λ that satisfy the equation are the *generalized eigenvalues* and the corresponding values of x are the *generalized right eigenvectors*.

If B is nonsingular, the problem could be solved by reducing it to a standard eigenvalue problem

$$B^{-1}Ax = \lambda x$$

Because B can be singular, an alternative algorithm, called the QZ method, is necessary.

When a matrix has no repeated eigenvalues, the eigenvectors are always independent and the eigenvector matrix `V` *diagonalizes* the original matrix `A` if applied as a similarity transformation. However, if a matrix has repeated eigenvalues, it is not similar to a diagonal matrix unless it has a full (independent) set of eigenvectors. If the eigenvectors are not independent then the original matrix is said to be *defective*. Even if a matrix is defective, the solution from `eig` satisfies `A*X = X*D`.

Examples

The matrix

```
B = [ 3     -2      -.9    2*eps
      -2     4       1    -eps
      -eps/4  eps/2  -1     0
      -.5    -.5      .1    1   ];
```

has elements on the order of roundoff error. It is an example for which the `nobalance` option is necessary to compute the eigenvectors correctly. Try the statements

```
[VB,DB] = eig(B)
B*VB - VB*DB
[VN,DN] = eig(B,'nobalance')
```

```
B*VN - VN*DN
```

Algorithm

Inputs of Type Double

For inputs of type `double`, MATLAB uses the following LAPACK routines to compute eigenvalues and eigenvectors.

Case	Routine
Real symmetric `A`	`DSYEV`
Real nonsymmetric `A`:	
• With preliminary balance step	`DGEEV` (with the scaling factor `SCLFAC = 2` in `DGEBAL`, instead of the LAPACK default value of `8`)
• `d = eig(A,'nobalance')`	`DGEHRD`, `DHSEQR`
• `[V,D] = eig(A,'nobalance')`	`DGEHRD`, `DORGHR`, `DHSEQR`, `DTREVC`
Hermitian `A`	`ZHEEV`
Non-Hermitian `A`:	
• With preliminary balance step	`ZGEEV` (with `SCLFAC = 2` instead of `8` in `ZGEBAL`)
• `d = eig(A,'nobalance')`	`ZGEHRD`, `ZHSEQR`
• `[V,D] = eig(A,'nobalance')`	`ZGEHRD`, `ZUNGHR`, `ZHSEQR`, `ZTREVC`
Real symmetric `A`, symmetric positive definite `B`.	`DSYGV`
Special case: `eig(A,B,'qz')` for real `A`, `B` (same as real nonsymmetric `A`, real general `B`)	`DGGEV`
Real nonsymmetric `A`, real general `B`	`DGGEV`
Complex Hermitian `A`, Hermitian positive definite `B`.	`ZHEGV`

Case	Routine
Special case: `eig(A,B,'qz')` for complex `A` or `B` (same as complex non-Hermitian `A`, complex `B`)	`ZGGEV`
Complex non-Hermitian `A`, complex `B`	`ZGGEV`

Inputs of Type Single

For inputs of type `single`, MATLAB uses the following LAPACK routines to compute eigenvalues and eigenvectors.

Case	Routine
Real symmetric `A`	`SSYEV`
Real nonsymmetric `A`:	
• With preliminary balance step	`SGEEV` (with the scaling factor `SCLFAC = 2` in `SGEBAL`, instead of the LAPACK default value of `8`)
• `d = eig(A,'nobalance')`	`SGEHRD`, `SHSEQR`
• `[V,D] = eig(A,'nobalance')`	`SGEHRD`, `SORGHR`, `SHSEQR`, `STREVC`
Hermitian `A`	`CHEEV`
Non-Hermitian `A`:	
• With preliminary balance step	`CGEEV`
• `d = eig(A,'nobalance')`	`CGEHRD`, `CHSEQR`
• `[V,D] = eig(A,'nobalance')`	`CGEHRD`, `CUNGHR`, `CHSEQR`, `CTREVC`
Real symmetric `A`, symmetric positive definite `B`.	`CSYGV`
Special case: `eig(A,B,'qz')` for real `A`, `B` (same as real nonsymmetric `A`, real general `B`)	`SGGEV`

Case	Routine
Real nonsymmetric `A`, real general `B`	`SGGEV`
Complex Hermitian `A`, Hermitian positive definite `B`.	`CHEGV`
Special case: `eig(A,B,'qz')` for complex `A` or `B` (same as complex non-Hermitian `A`, complex `B`)	`CGGEV`
Complex non-Hermitian `A`, complex `B`	`CGGEV`

See Also

`balance`, `condeig`, `eigs`, `hess`, `qz`, `schur`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Find largest eigenvalues and eigenvectors of sparse matrix

Syntax

```
d = eigs(A)
[V,D] = eigs(A)
[V,D,flag] = eigs(A)
eigs(A,B)
eigs(A,k)
eigs(A,B,k)
eigs(A,k,sigma)
eigs(A,B,k,sigma)
eigs(A,K,sigma,opts)
eigs(A,B,k,sigma,opts)
eigs(Afun,n,...)
```

Description

`d = eigs(A)` returns a vector of `A`'s six largest magnitude eigenvalues. `A` must be a square matrix, and should be large and sparse.

`[V,D] = eigs(A)` returns a diagonal matrix `D` of `A`'s six largest magnitude eigenvalues and a matrix `V` whose columns are the corresponding eigenvectors.

`[V,D,flag] = eigs(A)` also returns a convergence flag. If `flag` is `0` then all the eigenvalues converged; otherwise not all converged.

`eigs(A,B)` solves the generalized eigenvalue problem `A*V == B*V*D`. `B` must be symmetric (or Hermitian) positive definite and the same size as `A`. `eigs(A,[],...)` indicates the standard eigenvalue problem `A*V == V*D`.

`eigs(A,k)` and `eigs(A,B,k)` return the `k` largest magnitude eigenvalues.

`eigs(A,k,sigma)` and `eigs(A,B,k,sigma)` return `k` eigenvalues based on *`sigma`*, which can take any of the following values:

scalar (real or complex, including 0)	The eigenvalues closest to *sigma*. If A is a function, Afun must return Y = (A-*sigma**B)\x (i.e., Y = A\x when *sigma* = 0). Note, B need only be symmetric (Hermitian) positive semi-definite.
'lm'	Largest magnitude (default).
'sm'	Smallest magnitude. Same as *sigma* = 0. If A is a function, Afun must return Y = A\x. Note, B need only be symmetric (Hermitian) positive semi-definite.

For real symmetric problems, the following are also options:

'la'	Largest algebraic ('lr' in MATLAB 5)
'sa'	Smallest algebraic ('sr' in MATLAB 5)
'be'	Both ends (one more from high end if k is odd)

For nonsymmetric and complex problems, the following are also options:

'lr'	Largest real part
'sr'	Smallest real part
'li'	Largest imaginary part
'si'	Smallest imaginary part

Note The syntax eigs(A,k,...) is not valid when A is scalar. To pass a value for k, you must specify B as the second argument and k as the third (eigs(A,B,k,...)). If necessary, you can set B equal to [], the default.

---

eigs(A,K,*sigma*,opts) and eigs(A,B,k,*sigma*,opts) specify an options structure. Default values are shown in brackets ({}).

Parameter	Description	Values
`options.issym`	`1` if `A` or `A-sigma*B` represented by `Afun` is symmetric, `0` otherwise.	`[{0} | 1]`
`options.isreal`	`1` if `A` or `A-sigma*B` represented by `Afun` is real, `0` otherwise.	`[0 | {1}]`
`options.tol`	Convergence: Ritz estimate residual `<= tol*norm(A)`.	`[scalar | {eps}]`
`options.maxit`	Maximum number of iterations.	`[integer | {300}]`
`options.p`	Number of Lanczos basis vectors. `p >= 2k` (`p >= 2k+1` real nonsymmetric) advised. `p` must satisfy `k < p <= n` for real symmetric, `k+1 < p <= n` otherwise. Note: If you do not specify a `p` value, the default algorithm uses at least 20 Lanczos vectors.	`[integer | {2*k}]`
`options.v0`	Starting vector.	Randomly generated by ARPACK
`options.disp`	Diagnostic information display level.	`[0 | {1} | 2]`
`options.cholB`	`1` if `B` is really its Cholesky factor `chol(B)`, `0` otherwise.	`[{0} | 1]`
`options.permB`	Permutation vector `permB` if sparse `B` is really `chol(B(permB,permB))`.	`[permB | {1:n}]`

`eigs(Afun,n,...)` accepts the function handle `Afun` instead of the matrix `A`. See "Function Handles" in the MATLAB Programming documentation for more information. `Afun` must accept an input vector of size `n`.

`y = Afun(x)` should return:

`A*x`	if *`sigma`* is not specified, or is a string other than `'sm'`
`A\x`	if *`sigma`* is `0` or `'sm'`
`(A-`*`sigma`*`*I)\x`	if *`sigma`* is a nonzero scalar (standard eigenvalue problem). `I` is an identity matrix of the same size as `A`.
`(A-`*`sigma`*`*B)\x`	if *`sigma`* is a nonzero scalar (generalized eigenvalue problem)

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `Afun`, if necessary.

The matrix `A`, `A-`*`sigma`*`*I` or `A-`*`sigma`*`*B` represented by `Afun` is assumed to be real and nonsymmetric unless specified otherwise by `opts.isreal` and `opts.issym`. In all the `eigs` syntaxes, `eigs(A,...)` can be replaced by `eigs(Afun,n,...)`.

Remarks

`d = eigs(A,k)` is not a substitute for

```
d = eig(full(A))
d = sort(d)
d = d(end-k+1:end)
```

but is most appropriate for large sparse matrices. If the problem fits into memory, it may be quicker to use `eig(full(A))`.

Algorithm

`eigs` provides the reverse communication required by the Fortran library ARPACK, namely the routines `DSAUPD`, `DSEUPD`, `DNAUPD`, `DNEUPD`, `ZNAUPD`, and `ZNEUPD`.

Examples

Example 1

```
A = delsq(numgrid('C',15));
d1 = eigs(A,5,'sm')
```

returns

```
Iteration 1: a few Ritz values of the 20-by-20 matrix:
     0
     0
     0
     0
     0

Iteration 2: a few Ritz values of the 20-by-20 matrix:
    1.8117
    2.0889
    2.8827
    3.7374
    7.4954

Iteration 3: a few Ritz values of the 20-by-20 matrix:
    1.8117
    2.0889
    2.8827
    3.7374
    7.4954



d1 =

    0.5520
    0.4787
    0.3469
    0.2676
    0.1334
```

Example 2

This example replaces the matrix `A` in example 1 with a handle to a function `dnRk`. The example is contained in an M-file `run_eigs` that

- Calls `eigs` with the function handle `@dnRk` as its first argument.
- Contains `dnRk` as a nested function, so that all variables in `run_eigs` are available to `dnRk`.

The following shows the code for `run_eigs`:

```
function d2 = run_eigs
n = 139;
opts.issym = 1;
R = 'C';
k = 15;
d2 = eigs(@dnRk,n,5,'sm',opts);

    function y = dnRk(x)
        y = (delsq(numgrid(R,k))) \ x;
    end
end
```

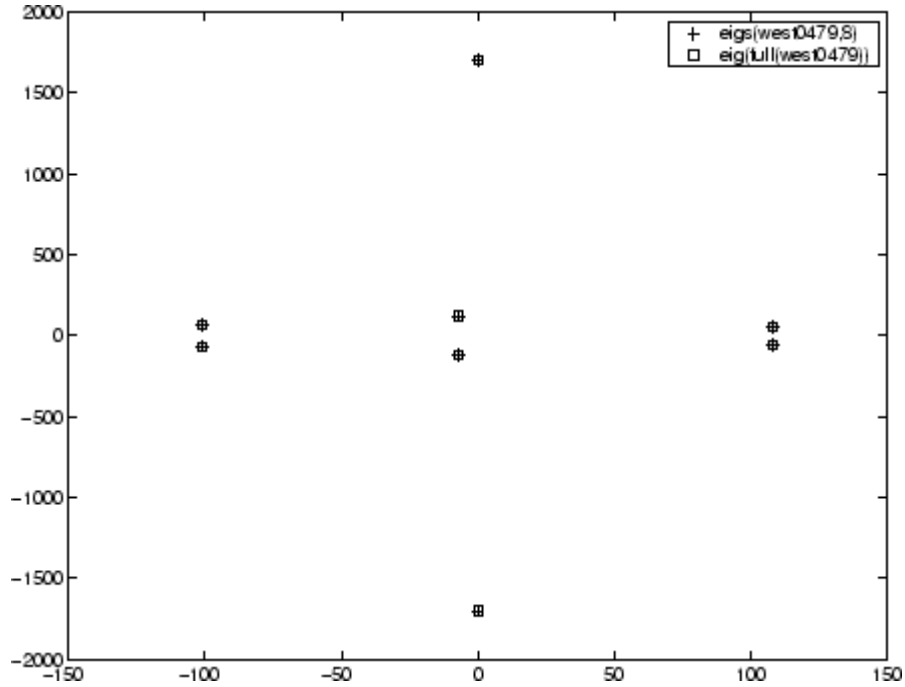
Example 3

`west0479` is a real 479-by-479 sparse matrix with both real and pairs of complex conjugate eigenvalues. `eig` computes all 479 eigenvalues. `eigs` easily picks out the largest magnitude eigenvalues.

This plot shows the 8 largest magnitude eigenvalues of `west0479` as computed by `eig` and `eigs`.

```
load west0479
d = eig(full(west0479))
dlm = eigs(west0479,8)
[dum,ind] = sort(abs(d));
plot(dlm,'k+')
hold on
plot(d(ind(end-7:end)),'ks')
```

```
hold off
legend('eigs(west0479,8)','eig(full(west0479))')
```

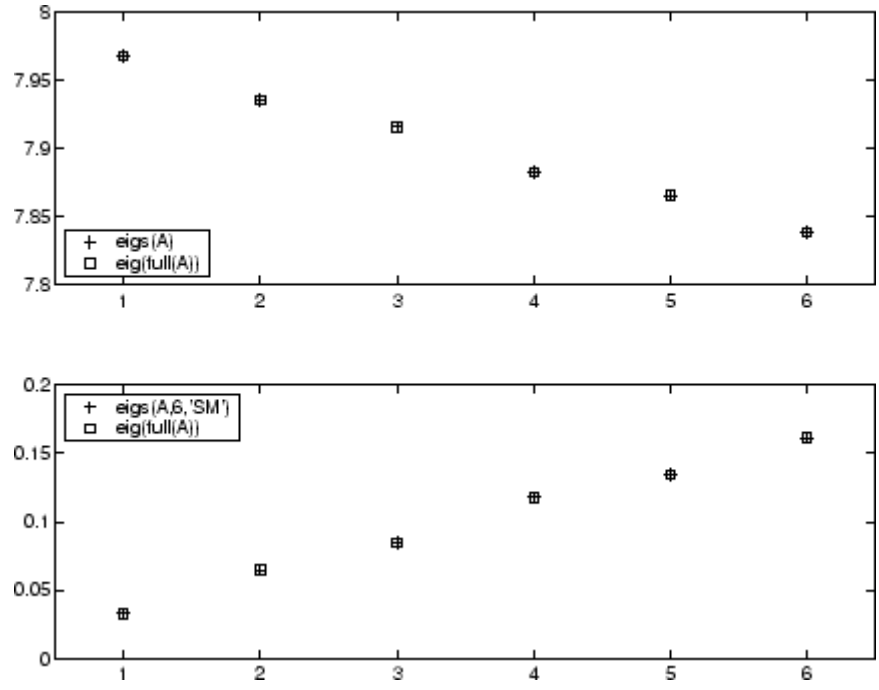


Example 4

`A = delsq(numgrid('C',30))` is a symmetric positive definite matrix of size 632 with eigenvalues reasonably well-distributed in the interval (0 8), but with 18 eigenvalues repeated at 4. The `eig` function computes all 632 eigenvalues. It computes and plots the six largest and smallest magnitude eigenvalues of `A` successfully with:

```
A = delsq(numgrid('C',30));
d = eig(full(A));
[dum,ind] = sort(abs(d));
dlm = eigs(A);
dsm = eigs(A,6,'sm');
```

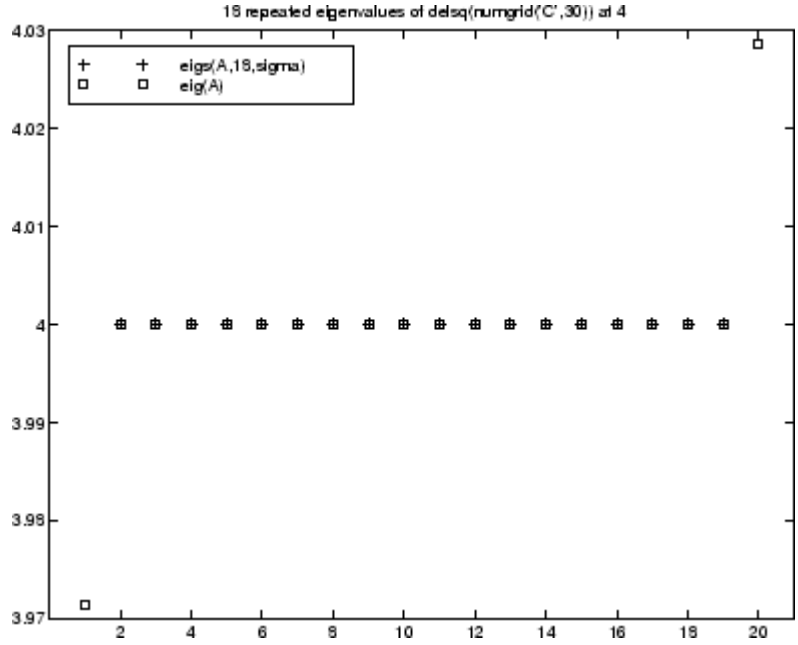
```
subplot(2,1,1)
plot(dlm,'k+')
hold on
plot(d(ind(end:-1:end-5)),'ks')
hold off
legend('eigs(A)','eig(full(A))',3)
set(gca,'XLim',[0.5 6.5])

subplot(2,1,2)
plot(dsm,'k+')
hold on
plot(d(ind(1:6)),'ks')
hold off
legend('eigs(A,6,''sm'')','eig(full(A))',2)
set(gca,'XLim',[0.5 6.5])
```



However, the repeated eigenvalue at 4 must be handled more carefully. The call `eigs(A,18,4.0)` to compute 18 eigenvalues near 4.0 tries to find eigenvalues of `A - 4.0*I`. This involves divisions of the form `1/(lambda - 4.0)`, where `lambda` is an estimate of an eigenvalue of `A`. As `lambda` gets closer to 4.0, `eigs` fails. We must use `sigma` near but not equal to 4 to find those 18 eigenvalues.

```
sigma = 4 - 1e-6
[V,D] = eigs(A,18,sigma)
```

The plot shows the 20 eigenvalues closest to 4 that were computed by `eig`, along with the 18 eigenvalues closest to `4 - 1e-6` that were computed by `eigs`.



See Also

`eig`, `svds`, `function_handle (@)`

References

[1] Lehoucq, R.B. and D.C. Sorensen, "Deflation Techniques for an Implicitly Re-Started Arnoldi Iteration," *SIAM J. Matrix Analysis and Applications*, Vol. 17, 1996, pp. 789-821.

[2] Lehoucq, R.B., D.C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, SIAM Publications, Philadelphia, 1998.

[3] Sorensen, D.C., "Implicit Application of Polynomial Filters in a k-Step Arnoldi Method," *SIAM J. Matrix Analysis and Applications*, Vol. 13, 1992, pp. 357-385.

Purpose

Jacobi elliptic functions

Syntax

```
[SN,CN,DN] = ellipj(U,M)
[SN,CN,DN] = ellipj(U,M,tol)
```

Definition

The Jacobi elliptic functions are defined in terms of the integral:

$$u = \int_0^{\phi} \frac{d\theta}{(1 - m\sin^2\theta)^{\frac{1}{2}}}$$

Then

$$sn(u) = \sin\phi,\ cn(u) = \cos\phi,\ dn(u) = (1 - m\sin^2\phi)^{\frac{1}{2}},\ am(u) = \phi$$

Some definitions of the elliptic functions use the modulus k instead of the parameter m. They are related by

$$k^2 = m = \sin^2\alpha$$

The Jacobi elliptic functions obey many mathematical identities; for a good sample, see [1].

Description

`[SN,CN,DN] = ellipj(U,M)` returns the Jacobi elliptic functions `SN`, `CN`, and `DN`, evaluated for corresponding elements of argument `U` and parameter `M`. Inputs `U` and `M` must be the same size (or either can be scalar).

`[SN,CN,DN] = ellipj(U,M,tol)` computes the Jacobi elliptic functions to accuracy `tol`. The default is `eps`; increase this for a less accurate but more quickly computed answer.

Algorithm

`ellipj` computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean [1]. It starts with the triplet of numbers:

$$a_0 = 1,\ b_0 = (1 - m)^{\frac{1}{2}},\ c_0 = (m)^{\frac{1}{2}}$$

`ellipj` computes successive iterates with

$$a_i = \frac{1}{2}(a_{i-1} + b_{i-1})$$
$$b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}}$$
$$c_i = \frac{1}{2}(a_{i-1} - b_{i-1})$$

Next, it calculates the amplitudes in radians using:

$$\sin(2\phi_{n-1} - \phi_n) = \frac{c_n}{a_n}\sin(\phi_n)$$

being careful to unwrap the phases correctly. The Jacobian elliptic functions are then simply:

$$sn(u) = \sin\phi_0$$
$$cn(u) = \cos\phi_0$$
$$dn(u) = (1 - m \cdot sn(u)^2)^{\frac{1}{2}}$$

Limitations

The `ellipj` function is limited to the input domain $0 \le m \le 1$. Map other values of `M` into this range using the transformations described in [1], equations 16.10 and 16.11. `U` is limited to real values.

See Also

`ellipke`

References

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

Purpose

Complete elliptic integrals of first and second kind

Syntax

```
K = ellipke(M)
[K,E] = ellipke(M)
[K,E] = ellipke(M,tol)
```

Definition

The *complete* elliptic integral of the first kind [1] is

$$K(m) = F(\pi/2|m)$$

where F, the elliptic integral of the first kind, is

$$K(m) = \int_0^1 [(1-t^2)(1-mt^2)]^{\frac{-1}{2}}\, dt = \int_0^{\frac{\pi}{2}} (1-m\sin^2\theta)^{\frac{-1}{2}}\, d\theta$$

The complete elliptic integral of the second kind

$$E(m) = E(K(m)) = E\langle\pi/2|m\rangle$$

is

$$E(m) = \int_0^1 (1-t^2)^{\frac{-1}{2}}(1-mt^2)^{\frac{1}{2}} dt = \int_0^{\frac{\pi}{2}} (1-m\sin^2\theta)^{\frac{1}{2}} d\theta$$

Some definitions of `K` and `E` use the modulus k instead of the parameter m. They are related by

$$k^2 = m = \sin^2\alpha$$

Description

`K = ellipke(M)` returns the complete elliptic integral of the first kind for the elements of `M`.

`[K,E] = ellipke(M)` returns the complete elliptic integral of the first and second kinds.

`[K,E] = ellipke(M,tol)` computes the complete elliptic integral to accuracy `tol`. The default is `eps`; increase this for a less accurate but more quickly computed answer.

Algorithm

`ellipke` computes the complete elliptic integral using the method of the arithmetic-geometric mean described in [1], section 17.6. It starts with the triplet of numbers

$$a_0 = 1,\ b_0 = (1-m)^{\frac{1}{2}},\ c_0 = (m)^{\frac{1}{2}}$$

`ellipke` computes successive iterations of a_i, b_i, and c_i with

$$a_i = \frac{1}{2}(a_{i-1} + b_{i-1})$$
$$b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}}$$
$$c_i = \frac{1}{2}(a_{i-1} - b_{i-1})$$

stopping at iteration n when $cn \approx 0$, within the tolerance specified by `eps`. The complete elliptic integral of the first kind is then

$$K(m) = \frac{\pi}{2a_n}$$

Limitations

`ellipke` is limited to the input domain $0 \le m \le 1$.

See Also

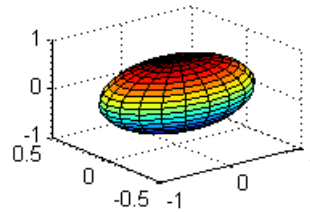
`ellipj`

References

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

Purpose

Generate ellipsoid



Syntax

```
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)
ellipsoid(axes_handle,...)
ellipsoid(...)
```

Description

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)` generates a surface mesh described by three `n+1`-by-`n+1` matrices, enabling `surf(x,y,z)` to plot an ellipsoid with center `(xc,yc,zc)` and semi-axis lengths `(xr,yr,zr)`.

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)` uses `n = 20`.

`ellipsoid(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`ellipsoid(...)` with no output arguments plots the ellipsoid as a surface.

Algorithm

`ellipsoid` generates the data using the following equation:

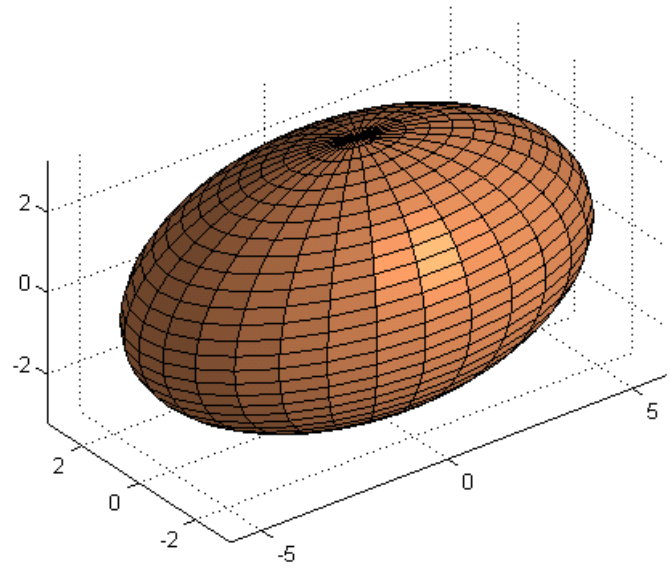
$$\frac{(x-xc)^2}{xr^2}+\frac{(y-yc)^2}{yr^2}+\frac{(z-zc)^2}{zr^2}$$

Note that `ellipsoid(0,0,0, .5,.5,.5)` is equivalent to a unit sphere.

Example

Generate ellipsoid with size and proportions of a standard U.S. football:

```
[x, y, z] = ellipsoid(0,0,0,5.9,3.25,3.25,30);
surfl(x, y, z)
colormap copper
axis equal
```



See Also

`cylinder`, `sphere`, `surf`

"Polygons and Surfaces" on page 1-90 for related functions

Purpose

Execute statements if condition is false

Syntax

```
if expression, statements1, else statements2, end
```

Description

`if` *`expression`*`,` *`statements1`*`, else` *`statements2`*`, end` evaluates *`expression`* and, if the evaluation yields logical `1` (`true`) or a nonzero result, executes one or more MATLAB commands denoted here as *`statements1`* or, if the evaluation yields logical `0` (`false`), executes the commands in *`statements2`*. `else` is used to delineate the alternate block of statements..

A `true` expression has either a logical `1` (`true`) or nonzero value. For nonscalar expressions, (for example, "if (matrix A is less than matrix B)"), `true` means that every element of the resulting matrix has a `true` or nonzero value.

Expressions usually involve relational operations such as `(count < limit)` or `isreal(A)`. Simple expressions can be combined by logical operators (`&`,`|`,`~`) into compound expressions such as `(count < limit) & ((height - offset) >= 0)`.

See "Program Control Statements" in the MATLAB Programming documentation for more information on controlling the flow of your program code.

Examples

In this example, if both of the conditions are not satisfied, then the student fails the course.

```
if ((attendance >= 0.90) & (grade_average >= 60))
   pass = 1;
else
   fail = 1;
end;
```

See Also

`if`, `elseif`, `end`, `for`, `while`, `switch`, `break`, `return`, relational operators, logical operators (elementwise and short-circuit)

elseif

Purpose

Execute statements if additional condition is true

Syntax

```
if expression1, statements1, elseif expression2,
statements2,
   end
```

Description

`if` *`expression1`*`, `*`statements1`*`, elseif` *`expression2`*`, `*`statements2`*`, end` evaluates *`expression1`* and, if the evaluation yields logical 1 (`true`) or a nonzero result, executes one or more MATLAB commands denoted here as *`statements1`*. If *`expression1`* is `false`, MATLAB evaluates the `elseif` expression, *`expression2`*. If *`expression2`* evaluates to `true` or a nonzero result, executes the commands in *`statements2`*.

A `true` expression has either a logical 1 (`true`) or nonzero value. For nonscalar expressions, (for example, is matrix A less then matrix B), `true` means that every element of the resulting matrix has a `true` or nonzero value.

Expressions usually involve relational operations such as `(count < limit)` or `isreal(A)`. Simple expressions can be combined by logical operators (`&`,`|`,`~`) into compound expressions such as `(count < limit) & ((height - offset) >= 0)`.

See "Program Control Statements" in the MATLAB Programming documentation for more information on controlling the flow of your program code.

Remarks

`elseif` , with a space between the `else` and the `if`, differs from `elseif`, with no space. The former introduces a new, nested `if`, which must have a matching `end`. The latter is used in a linear sequence of conditional statements with only one terminating `end`.

The two segments shown below produce identical results. Exactly one of the four assignments to `x` is executed, depending upon the values of the three logical expressions, `A`, `B`, and `C`.

```
if A                            if A
   x = a                            x = a
```

```
else                                elseif B
   if B                                 x = b
      x = b                         elseif C
   else                                 x = c
       if C                         else
          x = c                         x = d
       else                         end
           x = d
       end
   end
end
```

Examples

Here is an example showing `if`, `else`, and `elseif`.

```
for m = 1:k
    for n = 1:k
        if m == n
            a(m,n) = 2;
        elseif abs(m-n) == 2
            a(m,n) = 1;
        else
            a(m,n) = 0;
        end
    end
end
```

For `k=5` you get the matrix

```
a =

     2     0     1     0     0
     0     2     0     1     0
     1     0     2     0     1
     0     1     0     2     0
     0     0     1     0     2
```

See Also

`if`, `else`, `end`, `for`, `while`, `switch`, `break`, `return`, relational operators, logical operators (elementwise and short-circuit)

enableservice

Purpose

Enable, disable, or report status of Automation server

Syntax

```
state = enableservice('AutomationServer',enable)
state = enableservice('AutomationServer')
enableservice('DDEServer',enable)
```

Note Use COM, as described in COM Support in MATLAB. The `enableservice('DDEServer',enable)` syntax will be removed in a future version of MATLAB.

Description

`state = enableservice('AutomationServer',enable)` enables or disables the MATLAB Automation server.

If `enable` is logical `1` (`true`), `enableservice` converts an existing MATLAB session into an Automation server. If `enable` is logical `0` (`false`), `enableservice` disables the MATLAB Automation server.

`state` indicates the previous state of the Automation server. If `state = 1`, MATLAB was an Automation server. If `state` is logical `0` (`false`), MATLAB was not an Automation server.

`state = enableservice('AutomationServer')` returns the current state of the Automation server. If `state` is logical `1` (`true`), MATLAB is an Automation server.

`enableservice('DDEServer',enable)` enables the MATLAB DDE server. You cannot disable a DDE server once it has been enabled. Therefore, the only allowed value for `enable` is logical `1` (`true`).

Examples

Enable an Automation Server Example

Enable the Automation server in the current MATLAB session:

```
state = enableservice('AutomationServer',true);
```

Next, show the current state of the MATLAB session:

```
state = enableservice('AutomationServer')
```

MATLAB displays `state = 1 (true)`, showing that MATLAB is an Automation server.

Finally, enable the Automation server and show the previous state by typing

```
state = enableservice('AutomationServer',true)
```

MATLAB displays `state = 1 (true)`, showing that MATLAB previously was an Automation server.

Note the previous state may be the same as the current state. As seen in this case, `state = 1` shows MATLAB was, and still is, an Automation server.

end

Purpose

Terminate block of code, or indicate last array index

Syntax

```
end
```

Description

`end` is used to terminate `for`, `while`, `switch`, `try`, and `if` statements. Without an `end` statement, `for`, `while`, `switch`, `try`, and `if` wait for further input. Each `end` is paired with the closest previous unpaired `for`, `while`, `switch`, `try`, or `if` and serves to delimit its scope.

`end` also marks the termination of an M-file function, although in most cases, it is optional. `end` statements are required only in M-files that employ one or more nested functions. Within such an M-file, *every* function (including primary, nested, private, and subfunctions) must be terminated with an `end` statement. You can terminate any function type with `end`, but doing so is not required unless the M-file contains a nested function.

The `end` function also serves as the last index in an indexing expression. In that context, `end = (size(x,k))` when used as part of the `k`th index. Examples of this use are `X(3:end)` and `X(1,1:2:end-1)`. When using `end` to grow an array, as in `X(end+1)=5`, make sure `X` exists first.

You can overload the `end` statement for a user object by defining an `end` method for the object. The `end` method should have the calling sequence `end(obj,k,n)`, where `obj` is the user object, `k` is the index in the expression where the `end` syntax is used, and `n` is the total number of indices in the expression. For example, consider the expression

```
A(end-1,:)
```

MATLAB will call the `end` method defined for `A` using the syntax

```
end(A,1,2)
```

Examples

This example shows `end` used with the `for` and `if` statements.

```
for k = 1:n
 if a(k) == 0
  a(k) = a(k) + 2;
```

```
    end
end
```

In this example, `end` is used in an indexing expression.

```
A = magic(5)

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

B = A(end,2:end)

B =

    18    25     2     9
```

See Also

`break`, `for`, `if`, `return`, `switch`, `try`, `while`

eomday

Purpose

Last day of month

Syntax

```
E = eomday(Y, M)
```

Description

`E = eomday(Y, M)` returns the last day of the year and month given by corresponding elements of arrays `Y` and `M`.

Examples

Because 1996 is a leap year, the statement `eomday(1996,2)` returns `29`.

To show all the leap years in this century, try:

```
y = 1900:1999;
E = eomday(y, 2);
y(find(E == 29))

ans =
  Columns 1 through 6
     1904     1908     1912     1916     1920     1924

  Columns 7 through 12
     1928     1932     1936     1940     1944     1948

  Columns 13 through 18
     1952     1956     1960     1964     1968     1972

  Columns 19 through 24
     1976     1980     1984     1988     1992     1996
```

See Also

`datenum`, `datevec`, `weekday`

Purpose

Floating-point relative accuracy

Syntax

```
eps
d = eps(X)
eps('double')
eps('single')
```

Description

`eps` returns the distance from 1.0 to the next largest double-precision number, that is `eps = 2^(-52)`.

`d = eps(X)` is the positive distance from `abs(X)` to the next larger in magnitude floating point number of the same precision as `X`. `X` may be either double precision or single precision. For all `X`,

```
eps(X) = eps(-X) = eps(abs(X))
```

`eps('double')` is the same as `eps` or `eps(1.0)`.

`eps('single')` is the same as `eps(single(1.0))` or `single(2^-23)`.

Except for numbers whose absolute value is smaller than `realmin` , if `2^E <= abs(X) < 2^(E+1)`, then

```
eps(X) = 2^(E-23) if isa(X,'single')
eps(X) = 2^(E-52) if isa(X,'double')
```

For all `X` of class `double` such that `abs(X) <= realmin`, `eps(X) = 2^(-1074)`. Similarly, for all `X` of class `single` such that `abs(X) <= realmin('single')`, `eps(X) = 2^(-149)`.

Replace expressions of the form

```
if Y < eps * ABS(X)
```

with

```
if Y < eps(X)
```

Examples

```
double precision
eps(1/2) = 2^(-53)
```

```
eps(1) = 2^(-52)
eps(2) = 2^(-51)
eps(realmax) = 2^971
eps(0) = 2^(-1074)

if(abs(x)) <= realmin, eps(x) = 2^(-1074)
eps(realmin/2) = 2^(-1074)
eps(realmin/16) = 2^(-1074)
eps(Inf) = NaN
eps(NaN) = NaN

single precision
eps(single(1/2)) = 2^(-24)
eps(single(1)) = 2^(-23)
eps(single(2)) = 2^(-22)
eps(realmax('single')) = 2^104
eps(single(0)) = 2^(-149)
eps(realmin('single')/2) = 2^(-149)
eps(realmin('single')/16) = 2^(-149)
if(abs(x)) <= realmin('single'), eps(x) = 2^(-149)
eps(single(Inf)) = single(NaN)
eps(single(NaN)) = single(NaN)
```

See Also

`realmax`, `realmin`

Purpose

Test for equality

Syntax

```
A == B
eq(A, B)
```

Description

`A == B` compares each element of array `A` for equality with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` and `B` are equal, or logical 0 (`false`) where they are not equal. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then MATLAB returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`eq(A, B)` is called for the syntax `A == B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(magic(3), 2, 2);

A == B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     0     0
     1     0     0     0     0     0
```

```
     0     1     0     0     0     0
     1     0     0     0     0     0
```

See Also

`ne`, `le`, `ge`, `lt`, `gt`, relational operators

Purpose

Compare `MException` objects for equality

Syntax

```
eObj1 == eObj2
```

Description

`eObj1 == eObj2` tests scalar `MException` objects `eObj1` and `eObj2` for equality, returning logical 1 (`true`) if the two objects are identical, otherwise returning logical 0 (`false`).

See Also

`try`, `catch`, `error`, `assert`, `MException`, `isequal(MException)`, `ne(MException)`, `getReport(MException)`, `disp(MException)`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `last(MException)`

Purpose

Error functions

Syntax

```
Y = erf(X)
Y = erfc(X)
Y = erfcx(X)
X = erfinv(Y)
X = erfcinv(Y)
```

Definition

The error function `erf(X)` is twice the integral of the Gaussian distribution with 0 mean and variance of $1/2$.

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

The complementary error function `erfc(X)` is defined as

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1 - \mathrm{erf}(x)$$

The scaled complementary error function `erfcx(X)` is defined as

$$\mathrm{erfcx}(x) = e^{x^2} \mathrm{erfc}(x)$$

For large `X`, `erfcx(X)` is approximately $\left(\frac{1}{\sqrt{\pi}}\right)\frac{1}{x}$

Description

`Y = erf(X)` returns the value of the error function for each element of real array `X`.

`Y = erfc(X)` computes the value of the complementary error function.

`Y = erfcx(X)` computes the value of the scaled complementary error function.

`X = erfinv(Y)` returns the value of the inverse error function for each element of `Y`. Elements of `Y` must be in the interval `[-1 1]`. The function `erfinv` satisfies $y = \mathrm{erf}(x)$ for $-1 \le y \le 1$ and $-\infty \le x \le \infty$.

`X = erfcinv(Y)` returns the value of the inverse of the complementary error function for each element of `Y`. Elements of `Y` must be in the interval `[0 2]`. The function `erfcinv` satisfies $y = \text{erfc}(x)$ for $2 \geq y \geq 0$ and $-\infty \leq x \leq \infty$.

Remarks

The relationship between the complementary error function `erfc` and the standard normal probability distribution returned by the Statistics Toolbox function `normcdf` is

$$\text{normcdf}(x) = 0.5 * \text{erfc}(-x/\sqrt{2})$$

The relationship between the inverse complementary error function `erfcinv` and the inverse standard normal probability distribution returned by the Statistics Toolbox function `norminv` is

$$\text{norminv}(p) = -\sqrt{2} * \text{erfcinv}(2p)$$

Examples

`erfinv(1)` is `Inf`

`erfinv(-1)` is `-Inf`.

For `abs(Y) > 1`, `erfinv(Y)` is `NaN`.

Algorithms

For the error functions, the MATLAB code is a translation of a Fortran program by W. J. Cody, Argonne National Laboratory, NETLIB/SPECFUN, March 19, 1990. The main computation evaluates near-minimax rational approximations from [1].

For the inverse of the error function, rational approximations accurate to approximately six significant digits are used to generate an initial approximation, which is then improved to full accuracy by one step of Halley's method.

References

[1] Cody, W. J., "Rational Chebyshev Approximations for the Error Function," *Math. Comp.*, pgs. 631-638, 1969

Purpose

Display message and abort function

Syntax

```
error('message')
error('message', a1, a2, ...)
error('message_id', 'message')
error('message_id', 'message', a1, a2, ...)
error(message_struct)
```

Description

`error('message')` displays an error message and returns control to the keyboard. The error message contains the input string `message`.

The `error` command has no effect if `message` is an empty string.

`error('message', a1, a2, ...)` displays a message string that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `message` is converted to one of the values `a1, a2, ...` in the argument list.

Note MATLAB converts special characters (like `\n` and `%d`) in the error message string only when you specify more than one input argument with `error`. See Example 3 below.

`error('message_id', 'message')` attaches a unique message identifier, or `message_id`, to the error message. The identifier enables you to better identify the source of an error. See and in the MATLAB documentation for more information on the `message_id` argument and how to use it.

`error('message_id', 'message', a1, a2, ...)` includes formatting conversion characters in `message`, and the character translations `a1, a2, ...`.

`error(message_struct)` accepts a scalar error structure input `message_struct` with at least one of the fields `message`, `identifier`, and `stack`. (See the help for `lasterror` for more information on these fields.) If the `message_struct` input includes a `stack` field, then the `stack` field of the error will be set according to the contents of the `stack`

input. As a special case, if `message_struct` is an empty structure, no action is taken and `error` returns without exiting from the M-file.

Remarks

In addition to the `message_id` and `message`, the `error` function also determines where the error occurred, and provides this information using the `stack` field of the structure returned by `lasterror`. The `stack` field contains a structure array in the same format as the output of `dbstack`. This stack points to the line, function, and M-file in which the error occurred.

Examples

Example 1

The `error` function provides an error return from M-files:

```
function foo(x,y)
if nargin ~= 2
     error('Wrong number of input arguments')
end
```

The returned error message looks like this:

```
foo(pi)

??? Error using ==> foo
Wrong number of input arguments
```

Example 2

Specify a message identifier and error message string with `error`:

```
error('MyToolbox:angleTooLarge', ...
      'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterror` to determine the message identifier and error message string for the failing operation:

```
err = lasterror;

err.message
```

```
ans =
   The angle specified must be less than 90 degrees.

err.identifier
ans =
   MyToolbox:angleTooLarge
```

If this error is thrown from code in an M-file, you can find the M-file name, function, and line number using the `stack` field of the structure returned by `lasterror`:

```
err.stack
ans =
    file: 'd:\mytools\plotshape.m'
    name: 'check_angles'
    line: 26
```

Example 3

MATLAB converts special characters (like `\n` and `%d`) in the error message string only when you specify more than one input argument with `error`. In the single-argument case shown below, `\n` is taken to mean `backslash-n`. It is not converted to a newline character:

```
error('In this case, the newline \n is not converted.')
??? In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This holds true regardless of whether the additional argument supplies conversion values or is a message identifier:

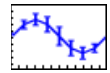
```
error('ErrorTests:convertTest', ...
      'In this case, the newline \n is converted.')
??? In this case, the newline
 is converted.
```

See Also

`lasterror`, `rethrow`, `errordlg`, `warning`, `lastwarn`, `warndlg`, `dbstop`, `disp`, `sprintf`

Purpose

Plot error bars along curve



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
errorbar(Y,E)
errorbar(X,Y,E)
errorbar(X,Y,L,U)
errorbar(...,LineSpec)
h = errorbar(...)
hlines = errorbar('v6',...)
```

Description

Error bars show the confidence level of data or the deviation along a curve.

`errorbar(Y,E)` plots `Y` and draws an error bar at each element of `Y`. The error bar is a distance of `E(i)` above and below the curve so that each bar is symmetric and `2*E(i)` long.

`errorbar(X,Y,E)` plots `Y` versus `X` with symmetric error bars `2*E(i)` long. `X`, `Y`, `E` must be the same size. When they are vectors, each error bar is a distance of `E(i)` above and below the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `E(i,j)` above and below the point defined by `(X(i,j),Y(i,j))`.

`errorbar(X,Y,L,U)` plots `X` versus `Y` with error bars `L(i)+U(i)` long specifying the lower and upper error bars. `X`, `Y`, `L`, and `U` must be the same size. When they are vectors, each error bar is a distance of `L(i)` below and `U(i)` above the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `L(i,j)` below and `U(i,j)` above the point defined by `(X(i,j),Y(i,j))`.

`errorbar(...,LineSpec)` uses the color and linestyle specified by the string `'LineSpec'`. The color is applied to the data line and error bars. The linestyle and marker are applied to the data line only. See `plot` for examples of styles.

`h = errorbar(...)` returns handles to the errorbarseries objects created. `errorbar` creates one object for vector input arguments and one object per column for matrix input arguments. See `errorbarseries properties` for more information.

Backward-Compatible Version

`hlines = errorbar('v6',...)` returns the handles of line objects instead of errorbarseries objects for compatibility with MATLAB 6.5 and earlier.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

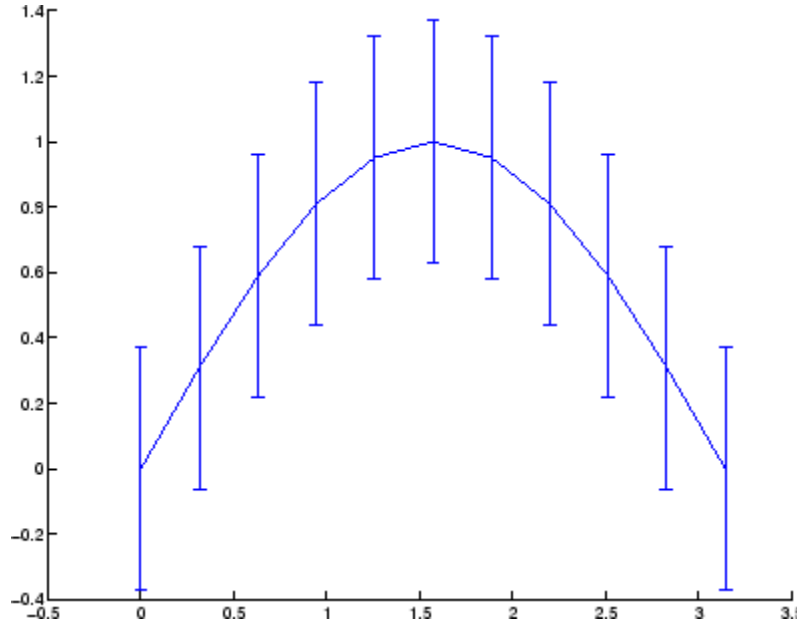
Remarks

When the arguments are all matrices, `errorbar` draws one line per matrix column. If `X` and `Y` are vectors, they specify one curve.

Examples

Draw symmetric error bars that are two standard deviation units in length.

```
X = 0:pi/10:pi;
Y = sin(X);
E = std(Y)*ones(size(X));
errorbar(X,Y,E)
```



See Also

`LineSpec`, `plot`, `std`, `corrcoef`

"Basic Plots and Graphs" on page 1-86 and ConfidenceBounds for related functions

See `Errorbarseries Properties` for property descriptions

Errorbarseries Properties

Purpose

Define errorbarseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property editor (`propertyeditor`).

Note that you cannot define default property values for errorbarseries objects. See “Plot Objects” for more information on errorbarseries objects.

Errorbarseries Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of errorbarseries objects in legends. The `Annotation` property enables you to specify whether this errorbarseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the errorbarseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the errorbarseries object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
`off`	Do not include the errorbarseries or its children in a legend (default)
`children`	Include only the children of the errorbarseries as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to

be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression

- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

See the ColorSpec reference page for more information on specifying color.

`CreateFcn`
string or function handle

Not available on errorbarseries objects.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this errorbarseries object. The `legend` function uses the string defined by the `DisplayName` property to label this errorbarseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this errorbarseries object's corresponding string and that string is used for the legend.

- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

`EraseMode`

```
{normal} | none | xor | background
```

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing

the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.

- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for

preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility`

settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`

`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When

HitTestArea is on, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LData`
array equal in size to `XData` and `YData`

Errorbar length below data point. The `errorbar` function uses this data to determine the length of the errorbar below each data point. Specify these values in data units. See also `UData`.

`LDataSource`
string (MATLAB variable)

Link `LData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `LData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `LData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

`LineStyle`

`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none`

specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this

property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the curve and error bars. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks.

For example, you might create an errorbarseries object and set the `Tag` property:

```
t = errorbar(Y,E,'Tag','errorbar1')
```

When you want to access the errorbarseries object, you can use `findobj` to find the errorbarseries object's handle.

The following statement changes the `MarkerFaceColor` property of the object whose `Tag` is `errorbar1`.

```
set(findobj('Tag','errorbar1'),'MarkerFaceColor','red')
```

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For errorbarseries objects, `Type` is 'hggroup'. The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UData`
array equal in size to `XData` and `YData`

Errorbar length above data point. The `errorbar` function uses this data to determine the length of the errorbar above each data point. Specify these values in data units.

`UDataSource`
string (MATLAB variable)

Link `UData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `UData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `UData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the errorbarseries object. Assign this property the handle of a uicontextmenu object created in the errorbarseries object's parent figure. Use the `uicontextmenu`

function to create the context menu. MATLAB displays the context menu whenever you right-click over the errorbarseries object.

`UserData`
array

User-specified data. This property can be any data you want to associate with the errorbarseries object (including cell arrays and structures). The errorbarseries object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of errorbarseries object and its children. By default, errorbarseries object visibility is `on`. This means all children of the errorbarseries object are visible unless the child object's `Visible` property is set to `off`. Setting an errorbarseries object's `Visible` property to `off` also makes its children invisible.

`XData`
array

X-coordinates of the curve. The `errorbar` function plots a curve using the *x*-axis coordinates in the `XData` array. `XData` must be the same size as `YData`.

If you do not specify `XData` (i.e., the input argument `x`), the `errorbar` function uses the indices of `YData` to create the curve. See the `XDataMode` property for related information.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the input argument `x`), the `errorbar` function sets this property to `manual`.

If you set `XDataMode` to `auto` after having specified `XData`, the `errorbar` function resets the *x* tick-mark labels to the indices of the `YData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
scalar, vector, or matrix

Data defining curve. `YData` contains the data defining the curve. If `YData` is a matrix, the `errorbar` function displays a curve with error bars for each column in the matrix.

The input argument `Y` in the `errorbar` function calling syntax assigns values to `YData`.

`YDataSource`
string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Purpose

Create and open error dialog box

Syntax

```
h = errordlg
h = errordlg(errorstring)
h = errordlg(errorstring,dlgname)
h = errordlg(errorstring,dlgname,createmode)
```

Description

`h = errordlg` creates and displays a dialog box with title `Error Dialog` that contains the string `This is the default error string.` The `errordlg` function returns the handle of the dialog box in `h`.

`h = errordlg(errorstring)` displays a dialog box with title `Error Dialog` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname)` displays a dialog box with title`dlgname` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname,createmode)` specifies whether the error dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `errorstring` and `dlgname`. The *`createmode`* argument can be a string or a structure.

If *`createmode`* is a string, it must be one of the values shown in the following table.

createmode Value	Description
`modal`	Replaces the error dialog box having the specified `Title`, that was last created or clicked on, with a modal error dialog box as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

createmode Value	Description
`non-modal` (default)	Creates a new nonmodal error dialog box with the specified parameters. Existing error dialog boxes with the same title are not deleted.
`replace`	Replaces the error dialog box having the specified `Title`, that was last created or clicked on, with a nonmodal error dialog boxbox as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the`uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in theFigure Properties.

If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings `'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

Remarks

MATLAB sizes the dialog box to fit the string `'errorstring'`. The error dialog box has an **OK** push button and remains on the screen until you press the **OK** button or the **Return** key. After pressing the button, the error dialog box disappears.

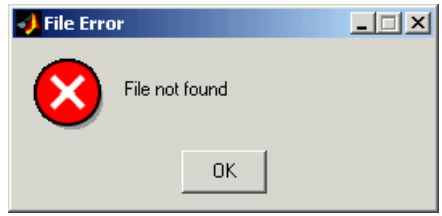
The appearance of the dialog box depends on the platform you use.

Examples

The function

```
errordlg('File not found','File Error');
```

displays this dialog box:



See Also

`dialog`, `helpdlg`, `inputdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-104 for related functions

Purpose

Time elapsed between date vectors

Syntax

```
e = etime(t2, t1)
```

Description

`e = etime(t2, t1)` returns the time in seconds between vectors `t1` and `t2`. The two vectors must be six elements long, in the format returned by `clock`:

```
T = [Year Month Day Hour Minute Second]
```

Remarks

When timing the duration of an event, use the `tic` and `toc` functions instead of `clock` or `etime`. These latter two functions are based on the system time which can be adjusted periodically by the operating system and thus might not be reliable in time comparison operations.

The `etime` function measures time elapsed between two points in time, and does not take into account differences in those points brought about by daylight savings time or changes in time zone.

Examples

Calculate how long a 2048-point real FFT takes.

```
x = rand(2048, 1);
t = clock;  fft(x);  etime(clock, t)
ans =
      0.4167
```

Limitations

As currently implemented, the `etime` function fails across month and year boundaries. Since `etime` is an M-file, you can modify the code to work across these boundaries if needed.

See Also

`clock`, `cputime`, `tic`, `toc`

Purpose

Elimination tree

Syntax

```
p = etree(A)
p = etree(A,'col')
p = etree(A,'sym')
[p,q] = etree(...)
```

Description

`p = etree(A)` returns an elimination tree for the square symmetric matrix whose upper triangle is that of `A`. `p(j)` is the parent of column `j` in the tree, or `0` if `j` is a root.

`p = etree(A,'col')` returns the elimination tree of `A'*A`.

`p = etree(A,'sym')` is the same as `p = etree(A)`.

`[p,q] = etree(...)` also returns a postorder permutation `q` of the tree.

See Also

`treelayout`, `treeplot`, `etreeplot`

Purpose

Plot elimination tree

Syntax

```
etreeplot(A)
etreeplot(A,nodeSpec,edgeSpec)
```

Description

`etreeplot(A)` plots the elimination tree of `A` (or `A+A'`, if non-symmetric).

`etreeplot(A,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

See Also

`etree`, `treeplot`, `treelayout`

Purpose

Execute string containing MATLAB expression

Syntax

```
eval(expression)
[a1, a2, a3, ...] = eval(function(b1, b2, b3, ...))
```

Description

`eval(expression)` executes `expression`, a string containing any valid MATLAB expression. You can construct `expression` by concatenating substrings and variables inside square brackets:

```
expression = [string1, int2str(var), string2, ...]
```

`[a1, a2, a3, ...] = eval(function(b1, b2, b3, ...))` executes `function` with arguments `b1, b2, b3, ...`, and returns the results in the specified output variables.

Remarks

Using the `eval` output argument list is recommended over including the output arguments in the expression string. The first syntax below avoids strict checking by the MATLAB parser and can produce untrapped errors and other unexpected behavior. Use the second syntax instead:

```
% Not recommended
  eval('[a1, a2, a3, ...] = function(var)')

% Recommended syntax
  [a1, a2, a3, ...] = eval('function(var)')
```

Examples

Example 1 – Working with a Series of Files

Load MAT-files `August1.mat` to `August10.mat` into the MATLAB workspace:

```
for d=1:10
   s = ['load August' int2str(d) '.mat']
   eval(s)
end
```

These are the strings being evaluated:

```
s =
   load August1.mat
s =
   load August2.mat
s =
   load August3.mat
      - etc. -
```

Example 2 – Assigning to Variables with Generated Names

Generate variable names that are unique in the MATLAB workspace and assign a value to each using `eval`:

```
for k = 1:5
   t = clock;
   pause(uint8(rand * 10));
   v = genvarname('time_elapsed', who);
   eval([v ' = etime(clock,t)'])
   end
```

As this code runs, `eval` creates a unique statement for each assignment:

```
time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010
time_elapsed3 =
    8.0010
time_elapsed4 =
    3.0040
```

Example 3 – Evaluating a Returned Function Name

The following command removes a figure by evaluating its `CloseRequestFcn` property as returned by `get`.

```
eval(get(h,'CloseRequestFcn'))
```

See Also

evalc, evalin, assignin, feval, catch, lasterror, try

Purpose

Evaluate MATLAB expression with capture

Syntax

```
T = evalc(S)
[T, X, Y, Z, ...] = evalc(S)
```

Description

`T = evalc(S)` is the same as `eval(S)` except that anything that would normally be written to the command window, except for error messages, is captured and returned in the character array `T` (lines in `T` are separated by `\n` characters).

`[T, X, Y, Z, ...] = evalc(S)` is the same as `[X, Y, Z, ...] = eval(S)` except that any output is captured into `T`.

Remark

When you are using `evalc`, `diary`, `more`, and `input` are disabled.

See Also

`eval`, `evalin`, `assignin`, `feval`, `diary`, `input`, `more`

evalin

Purpose

Execute MATLAB expression in specified workspace

Syntax

```
evalin(ws, expression)
[a1, a2, a3, ...] = evalin(ws, expression)
```

Description

`evalin(ws, expression)` executes *`expression`*, a string containing any valid MATLAB expression, in the context of the workspace `ws`. `ws` can have a value of `'base'` or `'caller'` to denote the MATLAB base workspace or the workspace of the caller function. You can construct *`expression`* by concatenating substrings and variables inside square brackets:

```
expression = [string1, int2str(var), string2,...]
```

`[a1, a2, a3, ...] = evalin(ws, expression)` executes *`expression`* and returns the results in the specified output variables. Using the `evalin` output argument list is recommended over including the output arguments in the expression string:

```
evalin(ws,'[a1, a2, a3, ...] = function(var)')
```

The above syntax avoids strict checking by the MATLAB parser and can produce untrapped errors and other unexpected behavior.

Remarks

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the M-file. Note, the base and caller workspaces are equivalent in the context of an M-file that is invoked from the MATLAB command line.

If you use `evalin('caller', ws)` in the MATLAB debugger after having changed your local workspace context with `dbup` or `dbdown`, MATLAB evaluates the expression in the context of the function that is one level up in the stack from your current workspace context.

Examples

This example extracts the value of the variable `var` in the MATLAB base workspace and captures the value in the local variable `v`:

```
v = evalin('base', 'var');
```

Limitation

`evalin` cannot be used recursively to evaluate an expression. For example, a sequence of the form `evalin('caller', 'evalin(''caller'', ''x'')')` doesn't work.

See Also

`assignin`, `eval`, `evalc`, `feval`, `catch`, `lasterror`, `try`

eventlisteners

Purpose

List of events attached to listeners

Syntax

```
C = h.eventlisteners
C = eventlisteners(h)
```

Description

`C = h.eventlisteners` lists any events, along with their event handler routines, that have been registered with control, `h`. The function returns cell array of strings `C`, with each row containing the name of a registered event and the handler routine for that event. If the control has no registered events, then `eventlisteners` returns an empty cell array.

Events and their event handler routines must be registered in order for the control to respond to them. You can register events either when you create the control, using `actxcontrol`, or at any time afterwards, using `registerevent`.

`C = eventlisteners(h)` is an alternate syntax for the same operation.

Examples

mwsamp Control Example

Create an `mwsamp` control, registering only the `Click` event. `eventlisteners` returns the name of the event and its event handler routine, `myclick`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ...
    [0 0 200 200], f, ...
    {'Click' 'myclick'});

h.eventlisteners
ans =
    'click'    'myclick'
```

Register two more events: `DblClick` and `MouseDown`. `eventlisteners` returns the names of the three registered events along with their respective handler routines:

```
h.registerevent({'DblClick', 'my2click'; ...
    'MouseDown' 'mymoused'});
```

```
h.eventlisteners
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
    'mousedown'    'mymoused'
```

Now unregister all events for the control. `eventlisteners` returns an empty cell array, indicating that no events have been registered for the control:

```
h.unregisterallevents

h.eventlisteners
ans =
     {}
```

Excel Workbook Example

```
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
wb.registerevent({'Activate' 'EvtActivateHandler'})
wb.eventlisteners

ans =

    'Activate'    'EvtActivateHandler'
```

See Also

`events`, `registerevent`, `unregisterevent`, `unregisterallevents`, `isevent`

Purpose

List of events control can trigger

Syntax

```
S = h.events
S = events(h)
```

Description

`S = h.events` returns structure array `S` containing all events, both registered and unregistered, known to the control, and the function prototype used when calling the event handler routine. For each array element, the structure field is the event name and the contents of that field is the function prototype for that event's handler.

`S = events(h)` is an alternate syntax for the same operation.

Examples

List Control Events Example

Create an `mwsamp` control and list all events:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

h.events
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
      Variant x, Variant y)
```

Assign the output to a variable and get one field of the returned structure:

```
ev = h.events;

ev.MouseDown
ans =
   void MouseDown(int16 Button, int16 Shift, ...
                  Variant x, Variant y)
```

List Excel Workbook Events Example

Open Excel and list all events for a Workbook object:

```
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
wb.events
```

MATLAB displays all events supported by the Workbook object.

```
 Open = void Open()
 Activate = void Activate()
 Deactivate = void Deactivate()
 BeforeClose = void BeforeClose(bool Cancel)
                 .
                 .
```

See Also

isevent, eventlisteners, registerevent, unregisterevent, unregisterallevents

Execute

Purpose

Execute MATLAB command in server

Syntax

MATLAB Client

```
result = h.Execute('command')
result = Execute(h, 'command')
result = invoke(h, 'Execute', 'command')
```

Method Signature

```
BSTR Execute([in] BSTR command)
```

Visual Basic Client

```
Execute(command As String) As String
```

Description

The `Execute` function executes the MATLAB statement specified by the string `command` in the MATLAB Automation server attached to handle `h`.

The server returns output from the command in the string, `result`. The `result` string also contains any warning or error messages that might have been issued by MATLAB as a result of the command.

Note that if you terminate the MATLAB command string with a semicolon and there are no warnings or error messages, `result` might be returned empty.

Remarks

If you want to be able to display output from `Execute` in the client window, you must specify an output variable (i.e., `result` in the above syntax statements).

Server function names, like `Execute`, are case sensitive when used with dot notation (the first syntax shown).

All three versions of the MATLAB client syntax perform the same operation.

Examples

Execute the MATLAB `version` function in the server and return the output to the MATLAB client.

MATLAB Client

```
h = actxserver('matlab.application');
server_version = h.Execute('version')
server_version =
ans =
   6.5.0.180913a (R13)
```

Visual Basic .NET Client

```
Dim Matlab As Object
Dim server_version As String
Matlab = CreateObject("matlab.application")
server_version = Matlab.Execute("version")
```

See Also

Feval, PutFullMatrix, GetFullMatrix, PutCharArray, GetCharArray

exifread

Purpose

Read EXIF information from JPEG and TIFF image files

Syntax

```
output = exifread(filename)
```

Description

`output = exifread(filename)` reads the Exchangeable Image File Format (EXIF) data from the file specified by the string `filename`. `filename` must specify a JPEG or TIFF image file. `output` is a structure containing metadata values about the image or images in `imagefile`.

Note `exifread` returns all EXIF tags and does not process them in any way.

EXIF is a standard used by digital camera manufacturers to store information in the image file, such as, the make and model of a camera, the time the picture was taken and digitized, the resolution of the image, exposure time, and focal length. For more information about EXIF and the meaning of metadata attributes, see `http://www.exif.org/`.

See Also

`imfinfo`, `imread`

Purpose

Check existence of variable, function, directory, or Java class

Graphical Interface

As an alternative to the `exist` function, use the Workspace Browser or the Current Directory Browser.

Syntax

```
exist name
exist name kind
A = exist('name','kind')
```

Description

`exist name` returns the status of `name`:

`0`	If `name` does not exist.
`1`	If `name` is a variable in the workspace.
`2`	If `name` is an M-file on your MATLAB search path. It also returns 2 when `name` is the full pathname to a file or the name of an ordinary file on your MATLAB search path.
`3`	If `name` is a MEX- or DLL-file on your MATLAB search path.
`4`	If `name` is an MDL-file on your MATLAB search path.
`5`	If `name` is a built-in MATLAB function.
`6`	If `name` is a P-file on your MATLAB search path.
`7`	If `name` is a directory.
`8`	If `name` is a Java class. (`exist` returns 0 if you start MATLAB with the `-nojvm` option.)

`exist name kind` returns the status of `name` for the specified *`kind`*. If `name` of type *`kind`* does not exist, it returns `0`. The *`kind`* argument may be one of the following:

`builtin`	Checks only for built-in functions.
`class`	Checks only for Java classes.
`dir`	Checks only for directories.

`file`	Checks only for files or directories.
`var`	Checks only for variables.

If `name` belongs to more than one category (e.g., if there are both an M-file and variable of the given name) and you do not specify a *`kind`* argument, `exist` returns one value according to the order of evaluation shown in the table below. For example, if `name` matches both a directory and M-file name, `exist` returns `7`, identifying it as a directory.

Order of Evaluation	Return Value	Type of Entity
1	1	Variable
2	5	Built-in
3	7	Directory
4	3	MEX or DLL-file
5	4	MDL-file
6	6	P-file
7	2	M-file
8	8	Java class

`A = exist('name','kind')` is the function form of the syntax.

Remarks

If `name` specifies a filename, that filename may include an extension to preclude conflicting with other similar filenames. For example, `exist('file.ext')`.

If `name` specifies a filename, MATLAB attempts to locate the file, examines the filename extension, and determines the value to return based on the extension alone. MATLAB does not examine the contents or internal structure of the file.

You can specify a partial path to a directory or file. A partial pathname is a pathname relative to the MATLAB path that contains only the trailing one or more components of the full pathname. For example,

both of the following commands return 2, identifying `mkdir.m` as an M-file. The first uses a partial pathname:

```
exist('matlab/general/mkdir.m')
exist([matlabroot '/toolbox/matlab/general/mkdir.m'])
```

If a file or directory is not on the search path, then `name` must specify either a full pathname, a partial pathname relative to `MATLABPATH`, a partial pathname relative to your current directory, or the file or directory must reside in your current working directory.

If `name` is a Java class, then `exist('name')` returns an `8`. However, if `name` is a Java class file, then `exist('name')` returns a `2`.

Remarks

To check for the existence of more than one variable, use the `ismember` function. For example,

```
a = 5.83;
c = 'teststring';
ismember({'a','b','c'},who)

ans =

     1     0     1
```

Examples

This example uses `exist` to check whether a MATLAB function is a built-in function or a file:

```
type = exist('plot')
type =
5
```

This indicates that `plot` is a built-in function.

In the next example, `exist` returns `8` on the Java class, `Welcome`, and returns `2` on the Java class file, `Welcome.class`:

```
exist Welcome
ans =
```

```
     8

exist javaclasses/Welcome.class
ans =
     2
```

indicates there is a Java class Welcome and a Java class file Welcome.class.

The following example indicates that `testresults` is both a variable in the workspace and a directory on the search path:

```
exist('testresults','var')
ans =
     1
exist('testresults','dir')
ans =
     7
```

See Also

`assignin`, `computer`, `dir`, `evalin`, `help`, `inmem`, `isfield`, `isempty`, `lookfor`, `mfilename`, `partialpath`, `what`, `which`, `who`

Purpose

Terminate MATLAB (same as `quit`)

GUI Alternatives

As an alternative to the `exit` function, select **File > Exit MATLAB** or click the Close box in the MATLAB desktop.

Syntax

```
exit
```

Description

`exit` terminates the current MATLAB session after running `finish.m`, if the file `finish.m` exists. It performs the same as `quit` and takes the same termination options, such as **`force`**. For more information, see `quit`.

See Also

`quit`, `finish`

Purpose

Exponential

Syntax

```
Y = exp(X)
```

Description

The `exp` function is an elementary function that operates element-wise on arrays. Its domain includes complex numbers.

`Y = exp(X)` returns the exponential for each element of `X`.
For complex $z = x + i*y$, it returns the complex exponential $e^z = e^x(\cos(y) + i\sin(y))$.

Remark

Use `expm` for matrix exponentials.

See Also

`expm`, `log`, `log10`, `expint`

Purpose

Exponential integral

Syntax

```
Y = expint(X)
```

Definitions

The exponential integral computed by this function is defined as

$$E_1(x) = \int_x^\infty \frac{e^{-t}}{t} dt$$

Another common definition of the exponential integral function is the Cauchy principal value integral

$$Ei(x) = \int_{-\infty}^x \frac{e^t}{t} dt$$

which, for real positive `x`, is related to `expint` as

$$E_1(-x) = -Ei(x) - i\pi$$

Description

`Y = expint(X)` evaluates the exponential integral for each element of `X`.

References

[1] Abramowitz, M. and I. A. Stegun. *Handbook of Mathematical Functions*. Chapter 5, New York: Dover Publications, 1965.

Purpose

Matrix exponential

Syntax

```
Y = expm(X)
```

Description

`Y = expm(X)` raises the constant e to the matrix power `X`.

Although it is not computed this way, if `X` has a full set of eigenvectors `V` with corresponding eigenvalues `D`, then

```
[V,D] = EIG(X) and EXPM(X) = V*diag(exp(diag(D)))/V
```

Use `exp` for the element-by-element exponential.

Algorithm

`expm` uses the Padé approximation with scaling and squaring. See reference [3], below.

Note The `expmdemo1`, `expmdemo2`, and `expmdemo3` demos illustrate the use of Padé approximation, Taylor series approximation, and eigenvalues and eigenvectors, respectively, to compute the matrix exponential. References [1] and [2] describe and compare many algorithms for computing a matrix exponential.

Examples

This example computes and compares the matrix exponential of `A` and the exponential of `A`.

```
A = [1         1         0
     0         0         2
     0         0        -1 ];

expm(A)
ans =
    2.7183    1.7183         1.0862
    0         1.0000         1.2642
    0              0         0.3679
```

```
exp(A)
ans =
    2.7183        2.7183        1.0000
    1.0000        1.0000        7.3891
    1.0000        1.0000        0.3679
```

Notice that the diagonal elements of the two results are equal. This would be true for any triangular matrix. But the off-diagonal elements, including those below the diagonal, are different.

See Also

`exp`, `expm1`, `funm`, `logm`, `eig`, `sqrtm`

References

[1] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, p. 384, Johns Hopkins University Press, 1983.

[2] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review 20*, 1978, pp. 801-836.

[3] Higham, N. J., "The Scaling and Squaring Method for the Matrix Exponential Revisited," *SIAM J. Matrix Anal. Appl.*, 26(4) (2005), pp. 1179-1193.

expm1

Purpose

Compute `exp(x)-1` accurately for small values of `x`

Syntax

```
y = expm1(x)
```

Description

`y = expm1(x)` computes `exp(x)-1`, compensating for the roundoff in `exp(x)`.

For small `x`, `expm1(x)` is approximately `x`, whereas `exp(x)-1` can be zero.

See Also

`exp`, `expm`, `log1p`

Purpose

Export variables to workspace

Syntax

```
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title,selected)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title,selected,helpfunction)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title,selected,helpfunction,functionlist)
hdialog = export2wsdlg(...)
[hdialog,ok_pressed] = export2wsdlg(...)
```

Description

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport)` creates a dialog with a series of check boxes and edit fields. `checkboxlabels` is a cell array of labels for the check boxes. `defaultvariablenames` is a cell array of strings that serve as a basis for variable names that appear in the edit fields. `itemstoexport` is a cell array of the values to be stored in the variables. If there is only one item to export, `export2wsdlg` creates a text control instead of a check box.

Note By default, the dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title)` creates the dialog with `title` as its title.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title,selected)` creates the dialog allowing the user to control which check boxes are checked. `selected` is a logical array whose length is the same as `checkboxlabels`. True indicates that the check box should initially be checked, false unchecked.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title,selected,helpfunction)` creates the dialog with a help button. `helpfunction` is a callback that displays help.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title,selected,helpfunction,functionlist)` creates a dialog that enables the user to pass in `functionlist`, a cell array of functions and optional arguments that calculate, then return the value to export. `functionlist` should be the same length as `checkboxlabels`.

`hdialog = export2wsdlg(...)` returns the handle of the dialog.

`[hdialog,ok_pressed] = export2wsdlg(...)` sets `ok_pressed` to true if the OK button is pressed, or false otherwise. If two return arguments are requested, `hdialog` is `[]` and the function does not return until the dialog is closed.

The user can edit the text fields to modify the default variable names. If the same name appears in multiple edit fields, `export2wsdlg` creates a structure using that name. It then uses the `defaultvariablenames` as fieldnames for that structure.

The lengths of `checkboxlabels`, `defaultvariablenames`, `itemstoexport` and `selected` must all be equal.

The strings in `defaultvariablenames` must be unique.

Examples

This example creates a dialog box that enables the user to save the variables `sumA` and/or `meanA` to the workspace. The dialog box title is Save Sums to Workspace.

```
A = randn(10,1);
checkLabels = {'Save sum of A to variable named:' ...
               'Save mean of A to variable named:'};
varNames = {'sumA','meanA'};
items = {sum(A),mean(A)};
export2wsdlg(checkLabels,varNames,items,...
             'Save Sums to Workspace');
```

Purpose

Identity matrix

Syntax

```
Y = eye(n)
Y = eye(m,n)
eye([m n])
Y = eye(size(A))
eye(m, n, classname)
eye([m,n],classname)
```

Description

`Y = eye(n)` returns the `n`-by-`n` identity matrix.

`Y = eye(m,n)` or `eye([m n])` returns an `m`-by-`n` matrix with 1's on the diagonal and 0's elsewhere.

Note The size inputs `m` and `n` should be nonnegative integers. Negative integers are treated as 0.

`Y = eye(size(A))` returns an identity matrix the same size as `A`.

`eye(m, n, classname)` or `eye([m,n],classname)` is an m-by-n matrix with 1's of class `classname` on the diagonal and zeros of class `classname` elsewhere. `classname` is a string specifying the data type of the output. `classname` can have the following values: `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Example:

```
x = eye(2,3,'int8');
```

Limitations

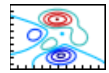
The identity matrix is not defined for higher-dimensional arrays. The assignment `y = eye([2,3,4])` results in an error.

See Also

`ones`, `rand`, `randn`, `zeros`

Purpose

Easy-to-use contour plotter



Syntax

```
ezcontour(fun)
ezcontour(fun,domain)
ezcontour(...,n)
ezcontour(axes_handle,...)
h = ezcontour(...)
```

Description

`ezcontour(fun)` plots the contour lines of `fun(x,y)` using the `contour` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and "Anonymous Functions") or a string (see Remarks).

`ezcontour(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]` (where `min` < x < `max`, `min` < y < `max`).

`ezcontour(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontour(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezcontour(...)` returns the handles to contour objects in `h`.

`ezcontour` automatically adds a title and axis labels.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezcontour`. For example, the MATLAB syntax for a contour plot of the expression

```
sqrt(x.^2 + y.^2)
```

is written as

```
ezcontour('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezcontour`.

If the function to be plotted is a function of the variables u and v (rather than x and y), the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontour('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontour`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontour(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezcontour` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then use an anonymous function to specify that parameter:

```
ezcontour(@(x,y)myfun(x,y,2))
```

Examples

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

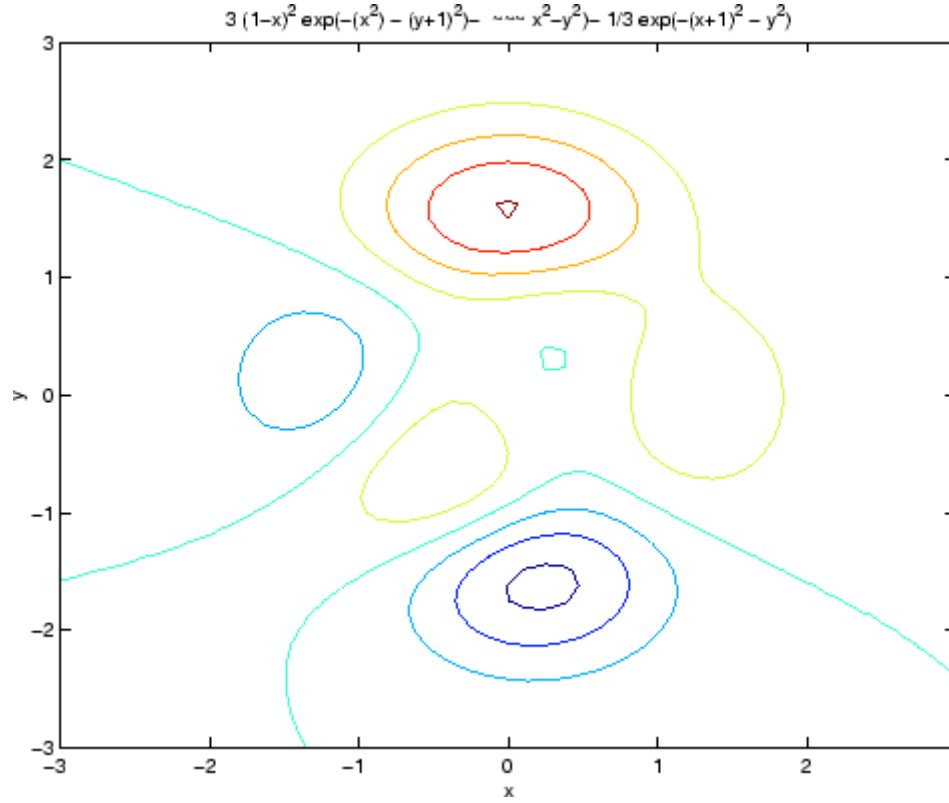
`ezcontour` requires a function handle argument that expresses this function using MATLAB syntax. This example uses an anonymous function, which you can define in the command window without creating an M-file.

```
f=@(x,y) 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

For convenience, this function is written on three lines. The MATLAB `peaks` function evaluates this expression for different sizes of grids.

Pass the function handle `f` to `ezcontour` along with a domain ranging from -3 to 3 in both *x* and *y* and specify a computational grid of 49-by-49:

```
ezcontour(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph, so MATLAB abbreviates the string.

See Also

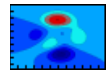
`contour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`, `function_handle`

“Contour Plots” on page 1-89 for related functions

ezcontourf

Purpose

Easy-to-use filled contour plotter



Syntax

```
ezcontourf(fun)
ezcontourf(fun,domain)
ezcontourf(...,n)
ezcontourf(axes_handle,...)
h = ezcontourf(...)
```

Description

`ezcontourf(fun)` plots the contour lines of `fun(x,y)` using the `contourf` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and Anonymous Functions) or a string (see Remarks).

`ezcontourf(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]`, where `min` $< x <$ `max`, `min` $< y <$ `max`).

`ezcontourf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontourf(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = ezcontourf(...)` returns the handles to contour objects in `h`.

`ezcontourf` automatically adds a title and axis labels.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezcontourf`. For example, the MATLAB syntax for a filled contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezcontourf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezcontourf`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontourf('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontourf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontourf(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^`, `.*`, `./`) since `ezcontourf` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezcontourf(@(x,y)myfun(x,y,2))
```

Examples

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

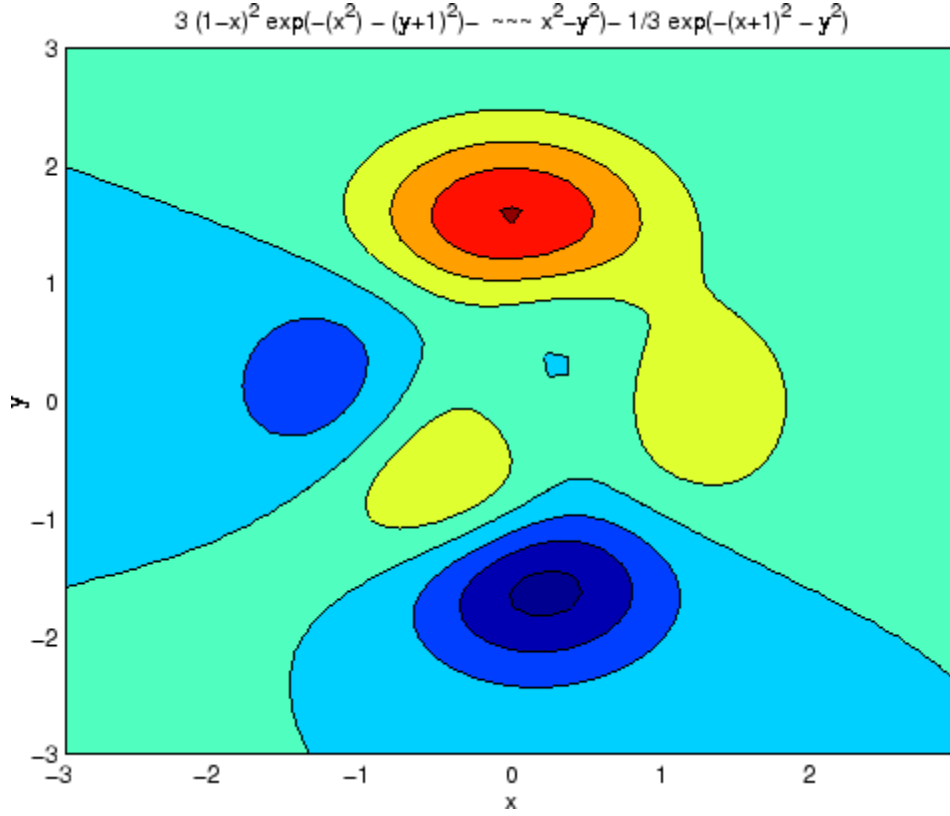
`ezcontourf` requires a string argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the string

```
f = ['3*(1-x)^2*exp(-(x^2)-(y+1)^2)',...
     '- 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2)',...
     '- 1/3*exp(-(x+1)^2 - y^2)'];
```

For convenience, this string is written on three lines and concatenated into one string using square brackets.

Pass the string variable `f` to `ezcontourf` along with a domain ranging from -3 to 3 and specify a grid of 49-by-49:

```
ezcontourf(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph, so MATLAB abbreviates the string.

See Also

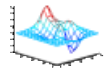
`contourf`, `ezcontour`, `ezmesh`, `ezmeshc`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`, `function_handle`

“Contour Plots” on page 1-89 for related functions

ezmesh

Purpose

Easy-to-use 3-D mesh plotter



Syntax

```
ezmesh(fun)
ezmesh(fun,domain)
ezmesh(funx,funy,funz)
ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])
ezmesh(funx,funy,funz,[min,max]
ezmesh(...,n)
ezmesh(...,'circ')
ezmesh(axes_handle,...)
h = ezmesh(...)
```

Description

`ezmesh(fun)` creates a graph of `fun(x,y)` using the `mesh` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and Anonymous Functions) or a string (see the Remarks section).

`ezmesh(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezmesh(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmesh(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmesh(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmesh(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezmesh(...)` returns the handle to a surface object in `h`.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezmesh`. For example, the MATLAB syntax for a mesh plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmesh('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezmesh`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmesh('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmesh`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmesh(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmesh` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmesh(@(x,y)myfun(x,y,2))
```

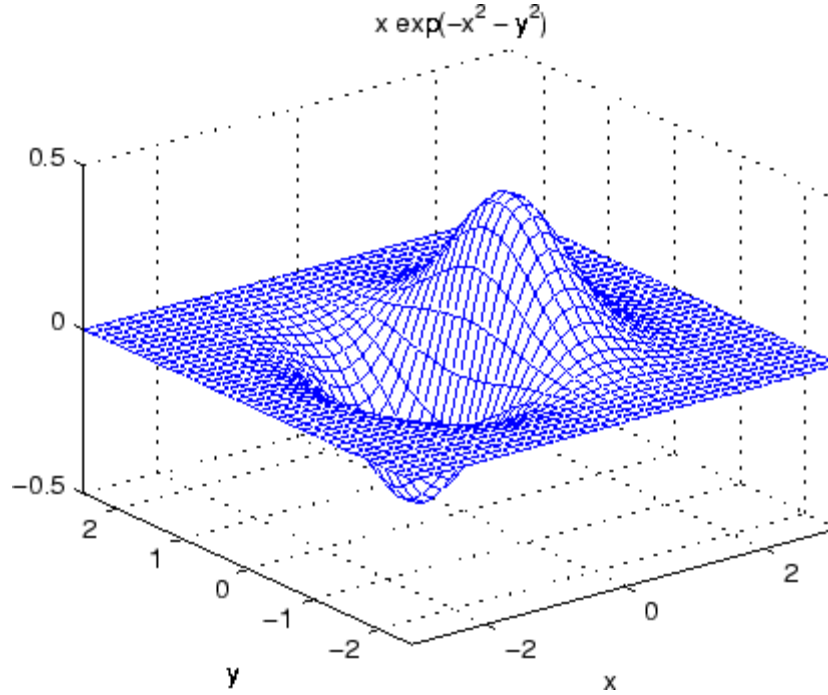
Examples

This example visualizes the function

$$f(x, y) = xe^{-x^2 - y^2}$$

with a mesh plot drawn on a 40-by-40 grid. The mesh lines are set to a uniform blue color by setting the colormap to a single color:

```
fh = @(x,y) x.*exp(-x.^2-y.^2);
ezmesh(fh,40)
colormap([0 0 1])
```



See Also

`ezmeshc`, `function_handle`, `mesh`

"Function Plots" on page 1-89 for related functions

ezmeshc

Purpose

Easy-to-use combination mesh/contour plotter

Syntax

```
ezmeshc(fun)
ezmeshc(fun,domain)
ezmeshc(funx,funy,funz)
ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])
ezmeshc(funx,funy,funz,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
ezmesh(axes_handle,...)
h = ezmeshc(...)
```

Description

`ezmeshc(fun)` creates a graph of `fun(x,y)` using the `meshc` function. `fun` is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and "Anonymous Functions") or a string (see the Remarks section).

`ezmeshc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` $< x <$ `max`, `min` $< y <$ `max`).

`ezmeshc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: $-2\pi <$ `s` $< 2\pi$, $-2\pi <$ `t` $< 2\pi$.

`ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmeshc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmeshc(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmeshc(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezmeshc(...)` returns the handle to a surface object in `h`.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezmeshc`. For example, the MATLAB syntax for a mesh/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmeshc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezmeshc`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmeshc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmeshc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmeshc(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmeshc` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmeshc(@(x,y)myfun(x,y,2))
```

Examples

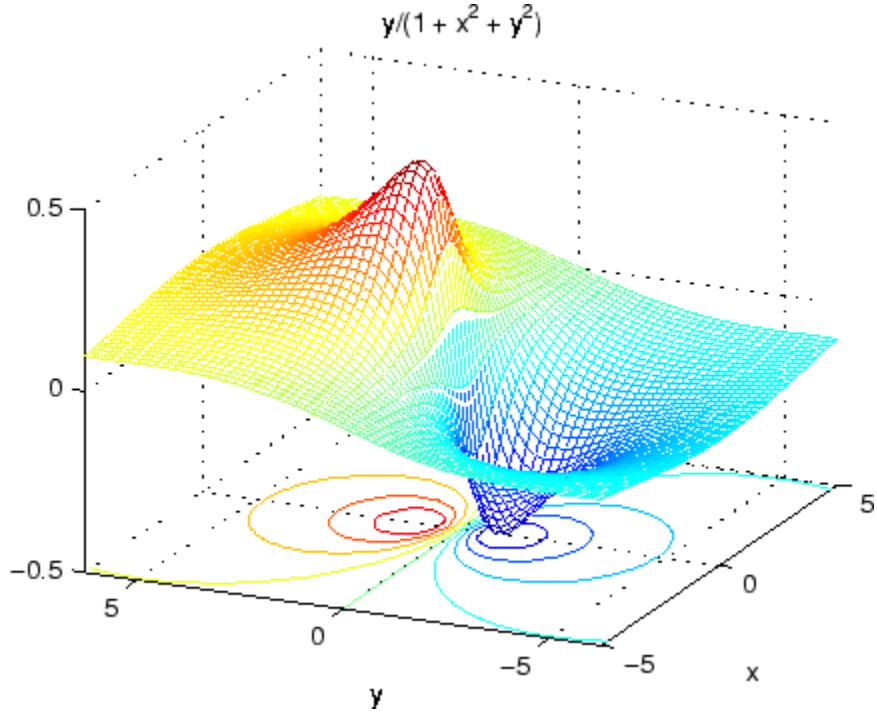
Create a mesh/contour graph of the expression

$$f(x, y) = \frac{y}{1 + x^2 + y^2}$$

over the domain -5 < x < 5, -2*pi < y < 2*pi:

```
ezmeshc('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi])
```

Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65.5 and elevation = 26)



See Also

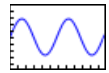
`ezmesh`, `ezsurfc`, `function_handle`, `meshc`

“Function Plots” on page 1-89 for related functions

ezplot

Purpose

Easy-to-use function plotter



Syntax

```
ezplot(fun)
ezplot(fun,[min,max])
ezplot(fun2)
ezplot(fun2,[xmin,xmax,ymin,ymax])
ezplot(fun2,[min,max])
ezplot(funx,funy)
ezplot(funx,funy,[tmin,tmax])
ezplot(...,figure_handle)
ezplot(axes_handle,...)
h = ezplot(...)
```

Description

`ezplot(fun)` plots the expression `fun(x)` over the default domain $-2\pi < x < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and Anonymous Functions) or a string (see the Remarks section).

`ezplot(fun,[min,max])` plots `fun(x)` over the domain: `min` $< x <$ `max`.

For implicitly defined functions, `fun2(x,y)`:

`ezplot(fun2)` plots `fun2(x,y) = 0` over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`ezplot(fun2,[xmin,xmax,ymin,ymax])` plots `fun2(x,y) = 0` over `xmin` < `x` < `xmax` and `ymin` < `y` < `ymax`.

`ezplot(fun2,[min,max])` plots `fun2(x,y) = 0` over `min` < `x` < `max` and `min` < `y` < `max`.

`ezplot(funx,funy)` plots the parametrically defined planar curve `funx(t)` and `funy(t)` over the default domain 0 < `t` < 2π.

`ezplot(funx,funy,[tmin,tmax])` plots `funx(t)` and `funy(t)` over `tmin < t < tmax`.

`ezplot(...,figure_handle)` plots the given function over the specified domain in the figure window identified by the handle `figure`.

`ezplot(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezplot(...)` returns the handle to a line objects in `h`.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot`. For example, the MATLAB syntax for a plot of the expression

```
x.^2 - y.^2
```

which represents an implicitly defined function, is written as

```
ezplot('x^2 - y^2')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezplot`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot`,

```
fh = @(x,y) sqrt(x.^2 + y.^2 - 1);
ezplot(fh)
axis equal
```

which plots a circle. Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezplot(@(x,y)myfun(x,y,2))
```

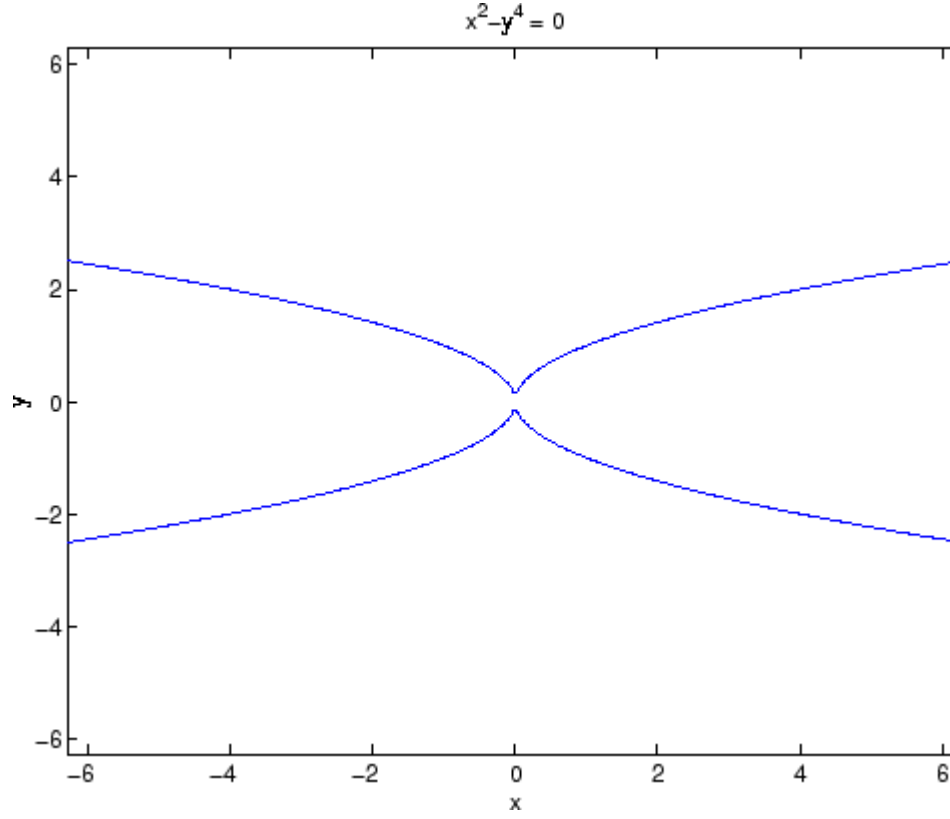
Examples

This example plots the implicitly defined function

$x^2 - y^4 = 0$

over the domain $[-2\pi, 2\pi]$:

```
ezplot('x^2-y^4')
```



See Also

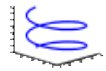
`ezplot3`, `ezpolar`, `function_handle`, `plot`

"Function Plots" on page 1-89 for related functions

ezplot3

Purpose

Easy-to-use 3-D parametric curve plotter



Syntax

```
ezplot3(funx,funy,funz)
ezplot3(funx,funy,funz,[tmin,tmax])
ezplot3(...,'animate')
ezplot3(axes_handle,...)
h = ezplot3(...)
```

Description

`ezplot3(funx,funy,funz)` plots the spatial curve `funx(t)`, `funy(t)`, and `funz(t)` over the default domain $0 < t < 2\pi$.

`funx`, `funy`, and `funz` can be function handles for M-file functions or an anonymous functions (see "Function Handles" and "Anonymous Functions") or strings (see the Remarks section).

`ezplot3(funx,funy,funz,[tmin,tmax])` plots the curve `funx(t)`, `funy(t)`, and `funz(t)` over the domain `tmin` < `t` < `tmax`.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

`ezplot3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezplot3(...)` returns the handle to the plotted objects in `h`.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot3`. For example, the MATLAB syntax for a plot of the expression

```
x = s./2, y = 2.*s, z = s.^2;
```

which represents a parametric function, is written as

```
ezplot3('s/2','2*s','s^2')
```

That is, `s/2` is interpreted as `s./2` in the string you pass to `ezplot3`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot3`.

```
fh1 = @(s) s./2; fh2 = @(s) 2.*s; fh3 = @(s) s.^2;
ezplot3(fh1,fh2,fh3)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfuntk`:

```
function s = myfuntk(t,k)
s = t.^k.*sin(t);
```

then you can use an anonymous function to specify that parameter:

```
ezplot3(@cos,@(t)myfuntk(t,1),@sqrt)
```

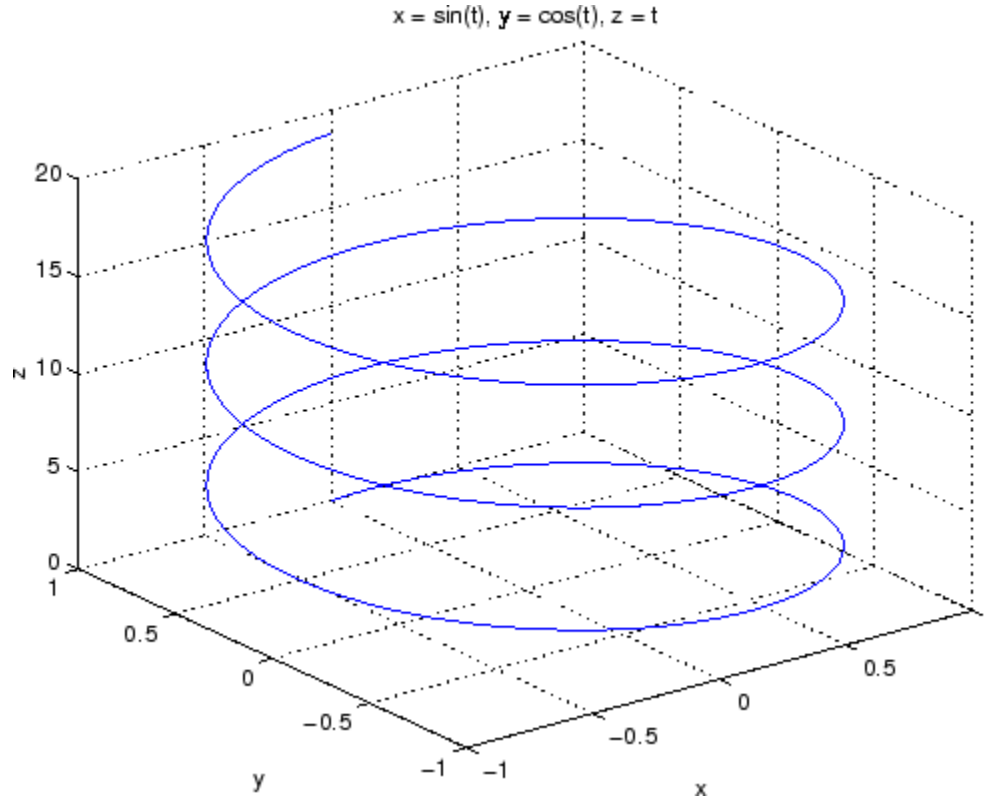
Examples

This example plots the parametric curve

$$x = \sin t, \quad y = \cos t, \quad z = t$$

over the domain [0,6π]:

```
ezplot3('sin(t)','cos(t)','t',[0,6*pi])
```



See Also

`ezplot`, `ezpolar`, `function_handle`, `plot3`

"Function Plots" on page 1-89 for related functions

Purpose

Easy-to-use polar coordinate plotter

Syntax

```
ezpolar(fun)
ezpolar(fun,[a,b])
ezpolar(axes_handle,...)
h = ezpolar(...)
```

Description

`ezpolar(fun)` plots the polar curve `rho = fun(theta)` over the default domain $0 <$ `theta` $< 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and "Function Handles") or a string (see the Remarks section).

`ezpolar(fun,[a,b])` plots `fun` for `a` < `theta` < `b`.

`ezpolar(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezpolar(...)` returns the handle to a line object in `h`.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezpolar`. For example, the MATLAB syntax for a plot of the expression

```
t.^2.*cos(t)
```

which represents an implicitly defined function, is written as

```
ezpolar('t^2*cos(t)')
```

That is, `t^2` is interpreted as `t.^2` in the string you pass to `ezpolar`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezpolar`.

```
fh = @(t) t.^2.*cos(t);
ezpolar(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^`, `.*`, `./`) since `ezpolar` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k1` and `k2` in `myfun`:

```
function s = myfun(t,k1,k2)
s = sin(k1*t).*cos(k2*t);
```

then you can use an anonymous function to specify the parameters:

```
ezpolar(@(t)myfun(t,2,3))
```

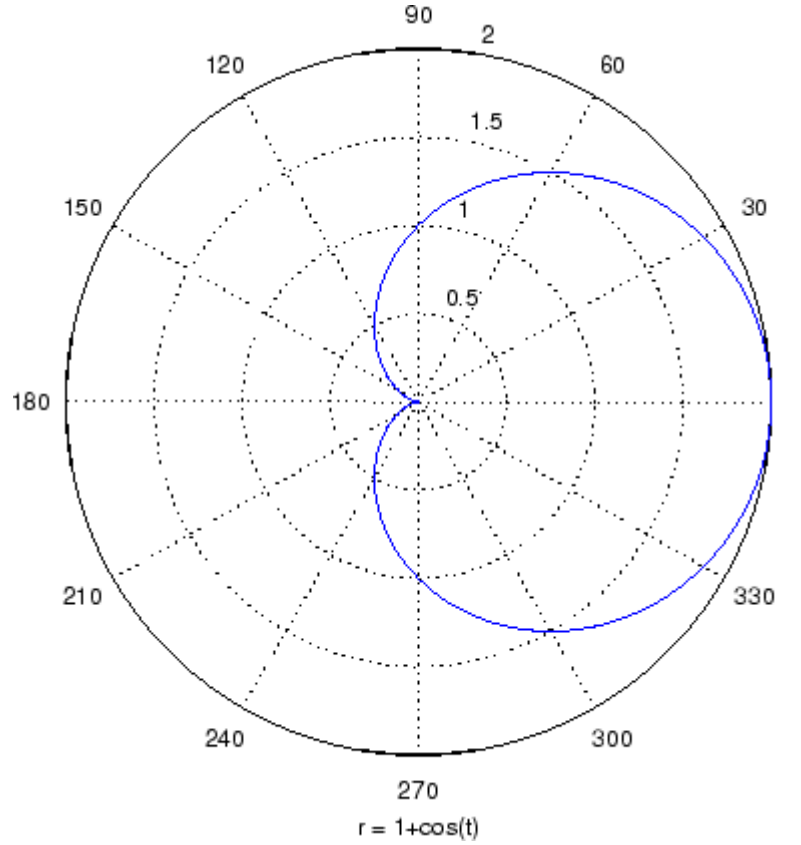
Examples

This example creates a polar plot of the function

1 + cos(t)

over the domain [0, 2π]:

```
ezpolar('1+cos(t)')
```



See Also

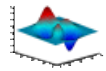
`ezplot`, `ezplot3`, `function_handle`, `plot`, `plot3`, `polar`

"Function Plots" on page 1-89 for related functions

ezsurf

Purpose

Easy-to-use 3-D colored surface plotter



Syntax

```
ezsurf(fun)
ezsurf(fun,domain)
ezsurf(funx,funy,funz)
ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurf(funx,funy,funz,[min,max]
ezsurf(...,n)
ezsurf(...,'circ')
ezsurf(axes_handle,...)
h = ezsurf(...)
```

Description

`ezsurf(fun)` creates a graph of `fun(x,y)` using the `surf` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and "Anonymous Functions") or a string (see the Remarks section).

`ezsurf(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezsurf(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurf(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurf(...,'circ')` plots `fun` over a disk centered on the domain.

`ezsurf(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezsurf(...)` returns the handle to a surface object in `h`.

Remarks

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezmesh`. For example, the MATLAB syntax for a surface plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezsurf`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurf('u^2 - v^3',[0,1],[3,6])` plots u^2 - v^3 over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurf` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurf(@(x,y)myfun(x,y,2,2,4))
```

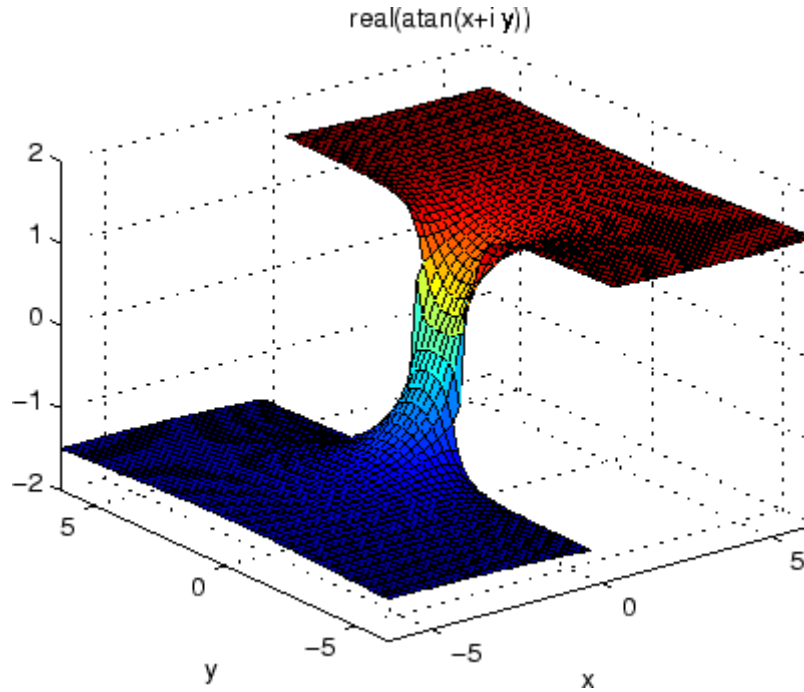
Examples

`ezsurf` does not graph points where the mathematical function is not defined (these data points are set to `NaN`s, which MATLAB does not plot). This example illustrates this filtering of singularities/discontinuous points by graphing the function

$$f(x, y) = real(atan(x + iy))$$

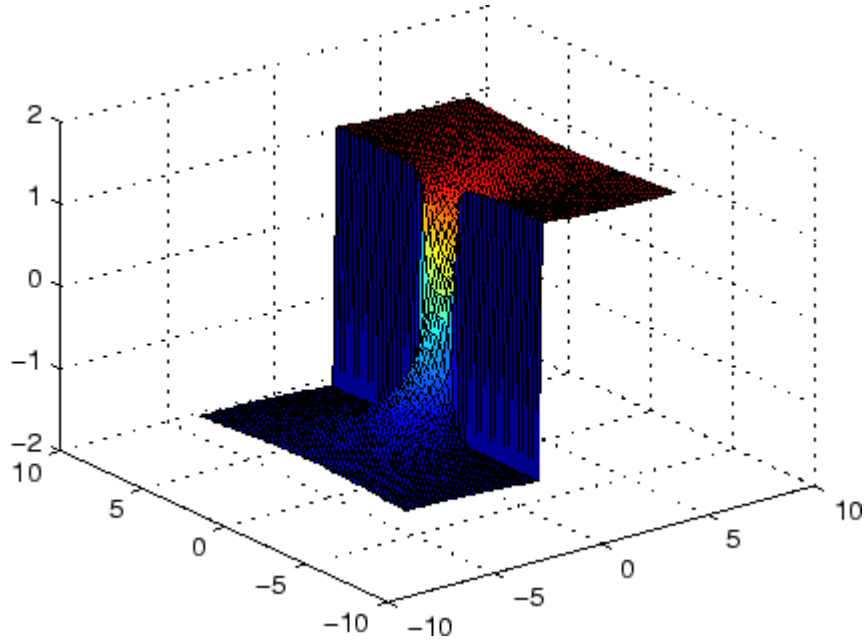
over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$:

```
ezsurf('real(atan(x+i*y))')
```



Using `surf` to plot the same data produces a graph without filtering of discontinuities (as well as requiring more steps):

```
[x,y] = meshgrid(linspace(-2*pi,2*pi,60));
z = real(atan(x+i.*y));
surf(x,y,z)
```



Note also that `ezsurf` creates graphs that have axis labels, a title, and extend to the axis limits.

See Also

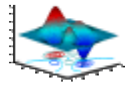
`ezmesh`, `ezsurfc`, `function_handle`, `surf`

"Function Plots" on page 1-89 for related functions

ezsurfc

Purpose

Easy-to-use combination surface/contour plotter



Syntax

```
ezsurfc(fun)
ezsurfc(fun,domain)
ezsurfc(funx,funy,funz)
ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurfc(funx,funy,funz,[min,max]
ezsurfc(...,n)
ezsurfc(...,'circ')
ezsurfc(axes_handle,...)
h = ezsurfc(...)
```

Description

`ezsurfc(fun)` creates a graph of `fun(x,y)` using the `surfc` function. The function `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for an M-file function or an anonymous function (see "Function Handles" and "Anonymous Functions") or a string (see the Remarks section).

`ezsurfc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where $min < x < max$, $min < y < max$).

`ezsurfc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurfc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurfc(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurfc(...,'circ')` plots *f* over a disk centered on the domain.

`ezsurfc(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezsurfc(...)` returns the handles to the graphics objects in `h`.

Remarks

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezsurfc`. For example, the MATLAB syntax for a surface/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurfc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezsurfc`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurfc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurfc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurfc` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurfc(@(x,y)myfun(x,y,2,2,4))
```

Examples

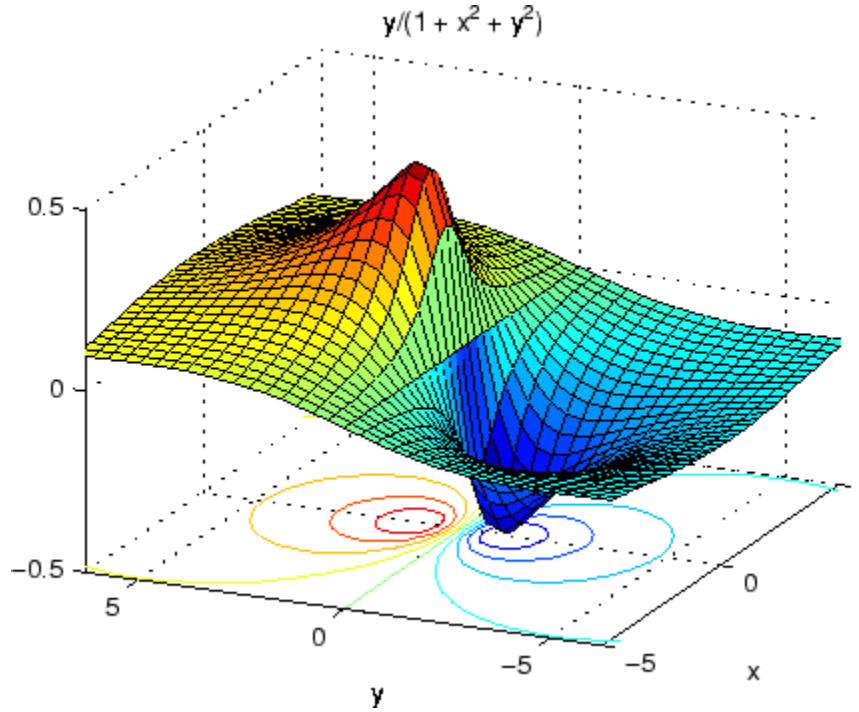
Create a surface/contour plot of the expression

$$f(x, y) = \frac{y}{1 + x^2 + y^2}$$

over the domain -5 < x < 5, -2*pi < y < 2*pi, with a computational grid of size 35-by-35:

```
ezsurfc('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
```

Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65.5 and elevation = 26).



See Also

`ezmesh`, `ezmeshc`, `ezsurf`, `function_handle`, `surfc`

"Function Plots" on page 1-89 for related functions

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

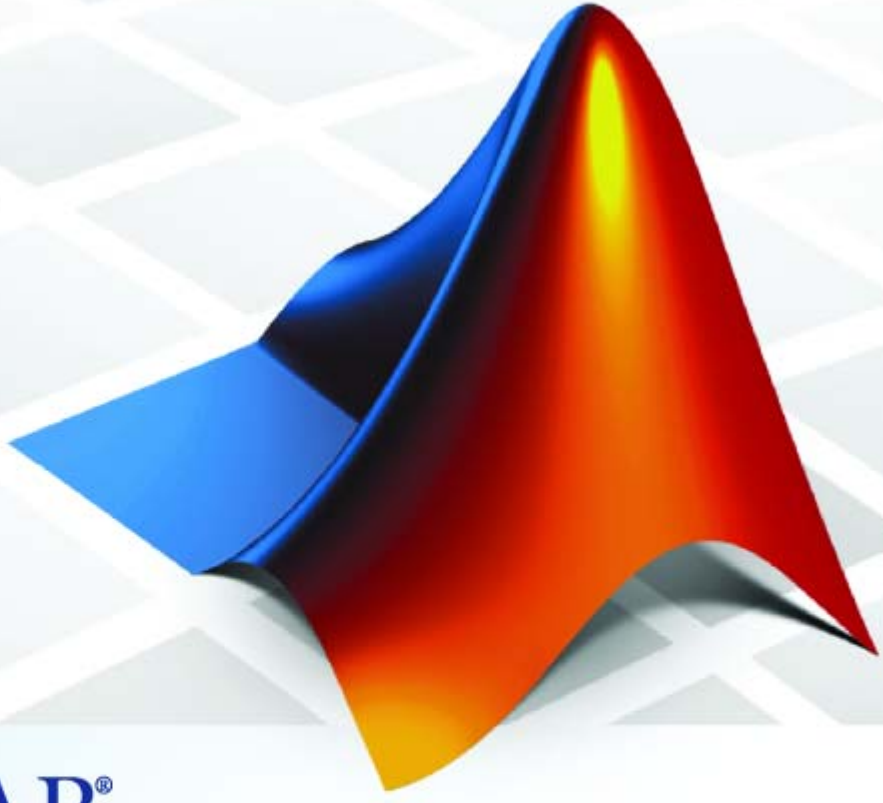
W

X

Y

Z

MATLAB® 7
Function Reference: Volume 2 (F-O)



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Function Reference

Revision History

December 1996	First printing	For MATLAB 5.0 (Release 8)
June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online only	Revised for MATLAB 5.3 (Release 11)
June 1999	Second printing	For MATLAB 5.3 (Release 11)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for 6.5 (Release 13)
June 2004	Online only	Revised for 7.0 (Release 14)
September 2006	Online only	Revised for 7.3 (Release 2006b)
September 2007	Online only	Revised for 7.5 (Release 2007b)

Contents

1 Functions — By Category

Functions — By Category

Creating Graphical User Interfaces (p. 1-104)	GUIDE, programming graphical user interfaces
External Interfaces (p. 1-109)	Interfaces to DLLs, Java, COM and ActiveX, Web services, and serial port devices, and C and Fortran routines

Desktop Tools and Development Environment

Startup and Shutdown

exit	Terminate MATLAB (same as `quit`)
finish	MATLAB termination M-file
matlab (UNIX)	Start MATLAB (UNIX systems)
matlab (Windows)	Start MATLAB (Windows systems)
matlabrc	MATLAB startup M-file for single-user systems or system administrators
prefdir	Directory containing preferences, history, and layout files
preferences	Open Preferences dialog box for MATLAB and related products

quit	Terminate MATLAB
startup	MATLAB startup M-file for user-defined options

Command Window and History

clc	Clear Command Window
commandhistory	Open Command History window, or select it if already open
commandwindow	Open Command Window, or select it if already open
diary	Save session to file
dos	Execute DOS command and return result
format	Set display format for output
home	Move cursor to upper-left corner of Command Window
matlabcolon (matlab:)	Run specified function via hyperlink
more	Control paged output for Command Window
perl	Call Perl script using appropriate operating system executable
system	Execute operating system command and return result
unix	Execute UNIX command and return result

Help for Using MATLAB

builddocsearchdb	Build searchable documentation database
demo	Access product demos via Help browser
doc	Reference page in Help browser
docopt	Web browser for UNIX platforms
docsearch	Open Help browser **Search** pane and search for specified term
echodemo	Run M-file demo step-by-step in Command Window
help	Help for MATLAB functions in Command Window
helpbrowser	Open Help browser to access all online documentation and demos
helpwin	Provide access to M-file help for all functions
info	Information about contacting The MathWorks
lookfor	Search for keyword in all help entries
playshow	Run M-file demo (deprecated; use `echodemo` instead)
support	Open MathWorks Technical Support Web page
web	Open Web site or file in Web browser or Help browser
whatsnew	Release Notes for MathWorks products

Workspace, Search Path, and File Operations

Workspace

assignin	Assign value to variable in specified workspace
clear	Remove items from workspace, freeing up system memory
evalin	Execute MATLAB expression in specified workspace
exist	Check existence of variable, function, directory, or Java class
openvar	Open workspace variable in Array Editor or other tool for graphical editing
pack	Consolidate workspace memory
uiimport	Open Import Wizard to import data
which	Locate functions and files
workspace	Open Workspace browser to manage workspace

Search Path

addpath	Add directories to MATLAB search path
genpath	Generate path string
partialpath	Partial pathname description

path	View or change MATLAB directory search path
path2rc	Save current MATLAB search path to `pathdef.m` file
pathdef	Directories in MATLAB search path
pathsep	Path separator for current platform
pathtool	Open Set Path dialog box to view and change MATLAB path
restoredefaultpath	Restore default MATLAB search path
rmpath	Remove directories from MATLAB search path
savepath	Save current MATLAB search path to `pathdef.m` file

File Operations

See also "File I/O" on page 1-76 functions.

cd	Change working directory
copyfile	Copy file or directory
delete	Remove files or graphics objects
dir	Directory listing
exist	Check existence of variable, function, directory, or Java class
fileattrib	Set or get attributes of file or directory
filebrowser	Current Directory browser
isdir	Determine whether input is a directory
lookfor	Search for keyword in all help entries

ls	Directory contents on UNIX system
matlabroot	Root directory of MATLAB installation
mkdir	Make new directory
movefile	Move file or directory
pwd	Identify current directory
recycle	Set option to move deleted files to recycle folder
rehash	Refresh function and file system path caches
rmdir	Remove directory
toolboxdir	Root directory for specified toolbox
type	Display contents of file
web	Open Web site or file in Web browser or Help browser
what	List MATLAB files in current directory
which	Locate functions and files

Programming Tools

Edit and Debug M-Files (p. 1-9)	Edit and debug M-files
Improve Performance and Tune M-Files (p. 1-9)	Improve performance and find potential problems in M-files
Source Control (p. 1-10)	Interface MATLAB with source control system
Publishing (p. 1-10)	Publish M-file code and results

Edit and Debug M-Files

clipboard	Copy and paste strings to and from system clipboard
datatipinfo	Produce short description of input variable
dbclear	Clear breakpoints
dbcont	Resume execution
dbdown	Change local workspace context when in debug mode
dbquit	Quit debug mode
dbstack	Function call stack
dbstatus	List all breakpoints
dbstep	Execute one or more lines from current breakpoint
dbstop	Set breakpoints
dbtype	List M-file with line numbers
dbup	Change local workspace context
debug	List M-file debugging functions
edit	Edit or create M-file
keyboard	Input from keyboard

Improve Performance and Tune M-Files

memory	Help for memory limitations
mlint	Check M-files for possible problems
mlintrpt	Run `mlint` for file or directory, reporting results in browser
pack	Consolidate workspace memory
profile	Profile execution time for function

profsave	Save profile report in HTML format
rehash	Refresh function and file system path caches
sparse	Create sparse matrix
zeros	Create array of all zeros

Source Control

checkin	Check files into source control system (UNIX)
checkout	Check files out of source control system (UNIX)
cmopts	Name of source control system
customverctrl	Allow custom source control system (UNIX)
undocheckout	Undo previous checkout from source control system (UNIX)
verctrl	Source control actions (Windows)

Publishing

grabcode	MATLAB code from M-files published to HTML
notebook	Open M-book in Microsoft Word (Windows)
publish	Publish M-file containing cells, saving output to file of specified type

System

Operating System Interface (p. 1-11)	Exchange operating system information and commands with MATLAB
MATLAB Version and License (p. 1-12)	Information about MATLAB version and license

Operating System Interface

clipboard	Copy and paste strings to and from system clipboard
computer	Information about computer on which MATLAB is running
dos	Execute DOS command and return result
getenv	Environment variable
hostid	MATLAB server host identification number
maxNumCompThreads	Controls maximum number of computational threads
perl	Call Perl script using appropriate operating system executable
setenv	Set environment variable
system	Execute operating system command and return result
unix	Execute UNIX command and return result
winqueryreg	Item from Microsoft Windows registry

MATLAB Version and License

ismac	Determine whether running Macintosh OS X versions of MATLAB
ispc	Determine whether PC (Windows) version of MATLAB
isstudent	Determine whether Student Version of MATLAB
isunix	Determine whether UNIX version of MATLAB
javachk	Generate error message based on Java feature support
license	Return license number or perform licensing task
prefdir	Directory containing preferences, history, and layout files
usejava	Determine whether Java feature is supported in MATLAB
ver	Version information for MathWorks products
verLessThan	Compare toolbox version to specified version string
version	Version number for MATLAB

Mathematics

Arrays and Matrices (p. 1-14)	Basic array operators and operations, creation of elementary and specialized arrays and matrices
Linear Algebra (p. 1-19)	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
Elementary Math (p. 1-23)	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
Polynomials (p. 1-28)	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
Interpolation and Computational Geometry (p. 1-28)	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
Cartesian Coordinate System Conversion (p. 1-31)	Conversions between Cartesian and polar or spherical coordinates
Nonlinear Numerical Methods (p. 1-31)	Differential equations, optimization, integration
Specialized Math (p. 1-35)	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
Sparse Matrices (p. 1-36)	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
Math Constants (p. 1-39)	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

Arrays and Matrices

Basic Information (p. 1-14)	Display array contents, get array information, determine array type
Operators (p. 1-15)	Arithmetic operators
Elementary Matrices and Arrays (p. 1-16)	Create elementary arrays of different types, generate arrays for plotting, array indexing, etc.
Array Operations (p. 1-17)	Operate on array content, apply function to each array element, find cumulative product or sum, etc.
Array Manipulation (p. 1-17)	Create, sort, rotate, permute, reshape, and shift array contents
Specialized Matrices (p. 1-18)	Create Hadamard, Companion, Hankel, Vandermonde, Pascal matrices, etc.

Basic Information

disp	Display text or array
display	Display text or array (overloaded method)
isempty	Determine whether array is empty
isequal	Test arrays for equality
isequalwithequalnans	Test arrays for equality, treating NaNs as equal
isfinite	Array elements that are finite
isfloat	Determine whether input is floating-point array
isinf	Array elements that are infinite
isinteger	Determine whether input is integer array

islogical	Determine whether input is logical array
isnan	Array elements that are NaN
isnumeric	Determine whether input is numeric array
isscalar	Determine whether input is scalar
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
length	Length of vector
max	Largest elements in array
min	Smallest elements in array
ndims	Number of array dimensions
numel	Number of elements in array or subscripted array expression
size	Array dimensions

Operators

+	Addition
+	Unary plus
-	Subtraction
-	Unary minus
*	Matrix multiplication
^	Matrix power
\	Backslash or left matrix divide
/	Slash or right matrix divide
’	Transpose
.’	Nonconjugated transpose
.*	Array multiplication (element-wise)

.^	Array power (element-wise)
.\	Left array divide (element-wise)
./	Right array divide (element-wise)

Elementary Matrices and Arrays

blkdiag	Construct block diagonal matrix from input arguments
diag	Diagonal matrices and diagonals of matrix
eye	Identity matrix
freqspace	Frequency spacing for frequency response
ind2sub	Subscripts from linear index
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
meshgrid	Generate `X` and `Y` arrays for 3-D plots
ndgrid	Generate arrays for N-D functions and interpolation
ones	Create array of all ones
rand	Uniformly distributed pseudorandom numbers
randn	Normally distributed random numbers
sub2ind	Single index from subscripts
zeros	Create array of all zeros

Array Operations

See “Linear Algebra” on page 1-19 and “Elementary Math” on page 1-23 for other array operations.

accumarray	Construct array with accumulation
arrayfun	Apply function to each element of array
bsxfun	Apply element-by-element binary operation to two arrays with singleton expansion enabled
cast	Cast variable to different data type
cross	Vector cross product
cumprod	Cumulative product
cumsum	Cumulative sum
dot	Vector dot product
idivide	Integer division with rounding option
kron	Kronecker tensor product
prod	Product of array elements
sum	Sum of array elements
tril	Lower triangular part of matrix
triu	Upper triangular part of matrix

Array Manipulation

blkdiag	Construct block diagonal matrix from input arguments
cat	Concatenate arrays along specified dimension
circshift	Shift array circularly

diag	Diagonal matrices and diagonals of matrix
end	Terminate block of code, or indicate last array index
flipdim	Flip array along specified dimension
fliplr	Flip matrix left to right
flipud	Flip matrix up to down
horzcat	Concatenate arrays horizontally
inline	Construct inline object
ipermute	Inverse permute dimensions of N-D array
permute	Rearrange dimensions of N-D array
repmat	Replicate and tile array
reshape	Reshape array
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
vectorize	Vectorize expression
vertcat	Concatenate arrays vertically

Specialized Matrices

compan	Companion matrix
gallery	Test matrices
hadamard	Hadamard matrix
hankel	Hankel matrix

hilb	Hilbert matrix
invhilb	Inverse of Hilbert matrix
magic	Magic square
pascal	Pascal matrix
rosser	Classic symmetric eigenvalue test problem
toeplitz	Toeplitz matrix
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix

Linear Algebra

Matrix Analysis (p. 1-19)	Compute norm, rank, determinant, condition number, etc.
Linear Equations (p. 1-20)	Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.
Eigenvalues and Singular Values (p. 1-21)	Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.
Matrix Logarithms and Exponentials (p. 1-22)	Matrix logarithms, exponentials, square root
Factorization (p. 1-22)	Cholesky, LU, and QR factorizations, diagonal forms, singular value decomposition

Matrix Analysis

cond	Condition number with respect to inversion
condeig	Condition number with respect to eigenvalues

det	Matrix determinant
norm	Vector and matrix norms
normest	2-norm estimate
null	Null space
orth	Range space of matrix
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
rref	Reduced row echelon form
subspace	Angle between two subspaces
trace	Sum of diagonal elements

Linear Equations

chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cond	Condition number with respect to inversion
condest	1-norm condition number estimate
funm	Evaluate general matrix function
ilu	Sparse incomplete LU factorization
inv	Matrix inverse
linsolve	Solve linear system of equations
lscov	Least-squares solution in presence of known covariance
lsqnonneg	Solve nonnegative least-squares constraints problem
lu	LU matrix factorization

luinc	Sparse incomplete LU factorization
pinv	Moore-Penrose pseudoinverse of matrix
qr	Orthogonal-triangular decomposition
rcond	Matrix reciprocal condition number estimate

Eigenvalues and Singular Values

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
condeig	Condition number with respect to eigenvalues
eig	Find eigenvalues and eigenvectors
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
gsvd	Generalized singular value decomposition
hess	Hessenberg form of matrix
ordeig	Eigenvalues of quasitriangular matrices
ordqz	Reorder eigenvalues in QZ factorization
ordschur	Reorder eigenvalues in Schur factorization
poly	Polynomial with specified roots
polyeig	Polynomial eigenvalue problem

rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtm	Matrix square root
ss2tf	Convert state-space filter parameters to transfer function form
svd	Singular value decomposition
svds	Find singular values and vectors

Matrix Logarithms and Exponentials

expm	Matrix exponential
logm	Matrix logarithm
sqrtm	Matrix square root

Factorization

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cholupdate	Rank 1 update to Cholesky factorization
gsvd	Generalized singular value decomposition
ilu	Sparse incomplete LU factorization
lu	LU matrix factorization

luinc	Sparse incomplete LU factorization
planerot	Givens plane rotation
qr	Orthogonal-triangular decomposition
qrdelete	Remove column or row from QR factorization
qrinsert	Insert column or row into QR factorization
qrupdate	
qz	QZ factorization for generalized eigenvalues
rsf2csf	Convert real Schur form to complex Schur form
svd	Singular value decomposition

Elementary Math

Trigonometric (p. 1-24)	Trigonometric functions with results in radians or degrees
Exponential (p. 1-25)	Exponential, logarithm, power, and root functions
Complex (p. 1-26)	Numbers with real and imaginary components, phase angles
Rounding and Remainder (p. 1-27)	Rounding, modulus, and remainder
Discrete Math (e.g., Prime Factors) (p. 1-27)	Prime factors, factorials, permutations, rational fractions, least common multiple, greatest common divisor

Trigonometric

acos	Inverse cosine; result in radians
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine; result in radians
asind	Inverse sine; result in degrees
asinh	Inverse hyperbolic sine
atan	Inverse tangent; result in radians
atan2	Four-quadrant inverse tangent
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent
cos	Cosine of argument in radians
cosd	Cosine ofo argument in degrees
cosh	Hyperbolic cosine
cot	Cotangent of argument in radians
cotd	Cotangent of argument in degrees
coth	Hyperbolic cotangent
csc	Cosecant of argument in radians

cscd	Cosecant of argument in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant of argument in radians
secd	Secant of argument in degrees
sech	Hyperbolic secant
sin	Sine of argument in radians
sind	Sine of argument in degrees
sinh	Hyperbolic sine of argument in radians
tan	Tangent of argument in radians
tand	Tangent of argument in degrees
tanh	Hyperbolic tangent

Exponential

exp	Exponential
expm1	Compute `exp(x)-1` accurately for small values of `x`
log	Natural logarithm
log10	Common (base 10) logarithm
log1p	Compute `log(1+x)` accurately for small values of `x`
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
nextpow2	Next higher power of 2
nthroot	Real `n`th root of real numbers
pow2	Base 2 power and scale floating-point numbers

reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays
sqrt	Square root

Complex

abs	Absolute value and complex magnitude
angle	Phase angle
complex	Construct complex data from real and imaginary components
conj	Complex conjugate
cplxpair	Sort complex numbers into complex conjugate pairs
i	Imaginary unit
imag	Imaginary part of complex number
isreal	Determine whether input is real array
j	Imaginary unit
real	Real part of complex number
sign	Signum function
unwrap	Correct phase angles to produce smoother phase plots

Rounding and Remainder

ceil	Round toward infinity
fix	Round toward zero
floor	Round toward minus infinity
idivide	Integer division with rounding option
mod	Modulus after division
rem	Remainder after division
round	Round to nearest integer

Discrete Math (e.g., Prime Factors)

factor	Prime factors
factorial	Factorial function
gcd	Greatest common divisor
isprime	Array elements that are prime numbers
lcm	Least common multiple
nchoosek	Binomial coefficient or all combinations
perms	All possible permutations
primes	Generate list of prime numbers
rat, rats	Rational fraction approximation

Polynomials

conv	Convolution and polynomial multiplication
deconv	Deconvolution and polynomial division
poly	Polynomial with specified roots
polyder	Polynomial derivative
polyeig	Polynomial eigenvalue problem
polyfit	Polynomial curve fitting
polyint	Integrate polynomial analytically
polyval	Polynomial evaluation
polyvalm	Matrix polynomial evaluation
residue	Convert between partial fraction expansion and polynomial coefficients
roots	Polynomial roots

Interpolation and Computational Geometry

Interpolation (p. 1-29)	Data interpolation, data gridding, polynomial evaluation, nearest point search
Delaunay Triangulation and Tessellation (p. 1-30)	Delaunay triangulation and tessellation, triangular surface and mesh plots
Convex Hull (p. 1-30)	Plot convex hull, plotting functions
Voronoi Diagrams (p. 1-30)	Plot Voronoi diagram, patch graphics object, plotting functions
Domain Generation (p. 1-31)	Generate arrays for 3-D plots, or for N-D functions and interpolation

Interpolation

dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
griddata	Data gridding
griddata3	Data gridding and hypersurface fitting for 3-D data
griddatan	Data gridding and hypersurface fitting (dimension >= 2)
interp1	1-D data interpolation (table lookup)
interp1q	Quick 1-D linear interpolation
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpft	1-D interpolation using FFT method
interpn	N-D data interpolation (table lookup)
meshgrid	Generate `X` and `Y` arrays for 3-D plots
mkpp	Make piecewise polynomial
ndgrid	Generate arrays for N-D functions and interpolation
padecoef	Padé approximation of time delays
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
ppval	Evaluate piecewise polynomial
spline	Cubic spline data interpolation
tsearchn	N-D closest simplex search
unmkpp	Piecewise polynomial details

Delaunay Triangulation and Tessellation

delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search

Convex Hull

convhull	Convex hull
convhulln	N-D convex hull
patch	Create patch graphics object
plot	2-D line plot
trisurf	Triangular surface plot

Voronoi Diagrams

dsearch	Search Delaunay triangulation for nearest point
patch	Create patch graphics object
plot	2-D line plot

voronoi	Voronoi diagram
voronoin	N-D Voronoi diagram

Domain Generation

meshgrid	Generate `X` and `Y` arrays for 3-D plots
ndgrid	Generate arrays for N-D functions and interpolation

Cartesian Coordinate System Conversion

cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical
pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Nonlinear Numerical Methods

Ordinary Differential Equations (IVP) (p. 1-32)	Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations (p. 1-33)	Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Boundary Value Problems (p. 1-33)	Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution

Partial Differential Equations (p. 1-34)	Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution
Optimization (p. 1-34)	Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature) (p. 1-34)	Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

Ordinary Differential Equations (IVP)

decic	Compute consistent initial conditions for `ode15i`
deval	Evaluate solution of differential equation problem
ode15i	Solve fully implicit differential equations, variable order method
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb	Solve initial value problems for ordinary differential equations
odefile	Define differential equation problem for ordinary differential equation solvers
odeget	Ordinary differential equation options parameters
odeset	Create or alter options structure for ordinary differential equation solvers
odextend	Extend solution of initial value problem for ordinary differential equation

Delay Differential Equations

dde23	Solve delay differential equations (DDEs) with constant delays
ddeget	Extract properties from delay differential equations options structure
ddesd	Solve delay differential equations (DDEs) with general delays
ddeset	Create or alter delay differential equations options structure
deval	Evaluate solution of differential equation problem

Boundary Value Problems

bvp4c	Solve boundary value problems for ordinary differential equations
bvp5c	Solve boundary value problems for ordinary differential equations
bvpget	Extract properties from options structure created with `bvpset`
bvpinit	Form initial guess for `bvp4c`
bvpset	Create or alter options structure of boundary value problem
bvpxtend	Form guess structure for extending boundary value solutions
deval	Evaluate solution of differential equation problem

Partial Differential Equations

pdepe	Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D
pdeval	Evaluate numerical solution of PDE using output of pdepe

Optimization

fminbnd	Find minimum of single-variable function on fixed interval
fminsearch	Find minimum of unconstrained multivariable function using derivative-free method
fzero	Find root of continuous function of one variable
lsqnonneg	Solve nonnegative least-squares constraints problem
optimget	Optimization options values
optimset	Create or edit optimization options structure

Numerical Integration (Quadrature)

dblquad	Numerically evaluate double integral
quad	Numerically evaluate integral, adaptive Simpson quadrature
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature
quadl	Numerically evaluate integral, adaptive Lobatto quadrature

quadv	Vectorized quadrature
triplequad	Numerically evaluate triple integral

Specialized Math

airy	Airy functions
besselh	Bessel function of third kind (Hankel function)
besseli	Modified Bessel function of first kind
besselj	Bessel function of first kind
besselk	Modified Bessel function of second kind
bessely	Bessel function of second kind
beta	Beta function
betainc	Incomplete beta function
betaln	Logarithm of beta function
ellipj	Jacobi elliptic functions
ellipke	Complete elliptic integrals of first and second kind
erf, erfc, erfcx, erfinv, erfcinv	Error functions
expint	Exponential integral
gamma, gammainc, gammaln	Gamma functions
legendre	Associated Legendre functions
psi	Psi (polygamma) function

Sparse Matrices

Elementary Sparse Matrices (p. 1-36)	Create random and nonrandom sparse matrices
Full to Sparse Conversion (p. 1-37)	Convert full matrix to sparse, sparse matrix to full
Working with Sparse Matrices (p. 1-37)	Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern.
Reordering Algorithms (p. 1-37)	Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
Linear Algebra (p. 1-38)	Compute norms, eigenvalues, factorizations, least squares, structural rank
Linear Equations (Iterative Methods) (p. 1-38)	Methods for conjugate and biconjugate gradients, residuals, lower quartile
Tree Operations (p. 1-39)	Elimination trees, tree plotting, factorization analysis

Elementary Sparse Matrices

spdiags	Extract and create sparse band and diagonal matrices
speye	Sparse identity matrix
sprand	Sparse uniformly distributed random matrix
sprandn	Sparse normally distributed random matrix
sprandsym	Sparse symmetric random matrix

Full to Sparse Conversion

find	Find indices and values of nonzero elements
full	Convert sparse matrix to full matrix
sparse	Create sparse matrix
spconvert	Import matrix from sparse matrix external format

Working with Sparse Matrices

issparse	Determine whether input is sparse
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
nzmax	Amount of storage allocated for nonzero matrix elements
spalloc	Allocate space for sparse matrix
spfun	Apply function to nonzero sparse matrix elements
spones	Replace nonzero sparse matrix elements with ones
spparms	Set parameters for sparse matrix routines
spy	Visualize sparsity pattern

Reordering Algorithms

amd	Approximate minimum degree permutation
colamd	Column approximate minimum degree permutation

colperm	Sparse column permutation based on nonzero count
dmperm	Dulmage-Mendelsohn decomposition
ldl	Block ldl' factorization for Hermitian indefinite matrices
randperm	Random permutation
symamd	Symmetric approximate minimum degree permutation
symrcm	Sparse reverse Cuthill-McKee ordering

Linear Algebra

cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
condest	1-norm condition number estimate
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
ilu	Sparse incomplete LU factorization
luinc	Sparse incomplete LU factorization
normest	2-norm estimate
spaugment	Form least squares augmented system
sprank	Structural rank
svds	Find singular values and vectors

Linear Equations (Iterative Methods)

bicg	Biconjugate gradients method
bicgstab	Biconjugate gradients stabilized method

cgs	Conjugate gradients squared method
gmres	Generalized minimum residual method (with restarts)
lsqr	LSQR method
minres	Minimum residual method
pcg	Preconditioned conjugate gradients method
qmr	Quasi-minimal residual method
symmlq	Symmetric LQ method

Tree Operations

etree	Elimination tree
etreeplot	Plot elimination tree
gplot	Plot nodes and links representing adjacency matrix
symbfact	Symbolic factorization analysis
treelayout	Lay out tree or forest
treeplot	Plot picture of tree

Math Constants

eps	Floating-point relative accuracy
i	Imaginary unit
Inf	Infinity
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
j	Imaginary unit

NaN	Not-a-Number
pi	Ratio of circle's circumference to its diameter, π
realmax	Largest positive floating-point number
realmin	Smallest positive normalized floating-point number

Data Analysis

Basic Operations

cumprod	Cumulative product
cumsum	Cumulative sum
prod	Product of array elements
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
sum	Sum of array elements

Descriptive Statistics

corrcoef	Correlation coefficients
cov	Covariance matrix
max	Largest elements in array
mean	Average or mean value of array
median	Median value of array

min	Smallest elements in array
mode	Most frequent values in array
std	Standard deviation
var	Variance

Filtering and Convolution

conv	Convolution and polynomial multiplication
conv2	2-D convolution
convn	N-D convolution
deconv	Deconvolution and polynomial division
detrend	Remove linear trends
filter	1-D digital filter
filter2	2-D digital filter

Interpolation and Regression

interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpn	N-D data interpolation (table lookup)
mldivide \, mrdivide /	Left or right matrix division
polyfit	Polynomial curve fitting
polyval	Polynomial evaluation

Fourier Transforms

abs	Absolute value and complex magnitude
angle	Phase angle
cplxpair	Sort complex numbers into complex conjugate pairs
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftn	N-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
fftw	Interface to FFTW library run-time algorithm tuning control
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform
ifftshift	Inverse FFT shift
nextpow2	Next higher power of 2
unwrap	Correct phase angles to produce smoother phase plots

Derivatives and Integrals

cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives

gradient	Numerical gradient
polyder	Polynomial derivative
polyint	Integrate polynomial analytically
trapz	Trapezoidal numerical integration

Time Series Objects

General Purpose (p. 1-44)	Combine `timeseries` objects, query and set `timeseries` object properties, plot `timeseries` objects
Data Manipulation (p. 1-45)	Add or delete data, manipulate `timeseries` objects
Event Data (p. 1-46)	Add or delete events, create new `timeseries` objects based on event data
Descriptive Statistics (p. 1-46)	Descriptive statistics for `timeseries` objects

General Purpose

get (timeseries)	Query `timeseries` object property values
getdatasamplesize	Size of data sample in `timeseries` object
getqualitydesc	Data quality descriptions
isempty (timeseries)	Determine whether `timeseries` object is empty
length (timeseries)	Length of time vector
plot (timeseries)	Plot time series
set (timeseries)	Set properties of `timeseries` object
size (timeseries)	Size of `timeseries` object

timeseries	Create `timeseries` object
tsdata.event	Construct event object for `timeseries` object
tsprops	Help on `timeseries` object properties
tstool	Open Time Series Tools GUI

Data Manipulation

addsample	Add data sample to `timeseries` object
ctranspose (timeseries)	Transpose `timeseries` object
delsample	Remove sample from `timeseries` object
detrend (timeseries)	Subtract mean or best-fit line and all `NaN`s from time series
filter (timeseries)	Shape frequency content of time series
getabstime (timeseries)	Extract date-string time vector into cell array
getinterpmethod	Interpolation method for `timeseries` object
getsampleusingtime (timeseries)	Extract data samples into new `timeseries` object
idealfilter (timeseries)	Apply ideal (noncausal) filter to `timeseries` object
resample (timeseries)	Select or interpolate `timeseries` data using new time vector
setabstime (timeseries)	Set times of `timeseries` object as date strings
setinterpmethod	Set default interpolation method for `timeseries` object

synchronize	Synchronize and resample two `timeseries` objects using common time vector
transpose (timeseries)	Transpose `timeseries` object
vertcat (timeseries)	Vertical concatenation of `timeseries` objects

Event Data

addevent	Add event to `timeseries` object
delevent	Remove `tsdata.event` objects from `timeseries` object
gettsafteratevent	New `timeseries` object with samples occurring at or after event
gettsafterevent	New `timeseries` object with samples occurring after event
gettsatevent	New `timeseries` object with samples occurring at event
gettsbeforeatevent	New `timeseries` object with samples occurring before or at event
gettsbeforeevent	New `timeseries` object with samples occurring before event
gettsbetweenevents	New `timeseries` object with samples occurring between events

Descriptive Statistics

iqr (timeseries)	Interquartile range of `timeseries` data
max (timeseries)	Maximum value of `timeseries` data
mean (timeseries)	Mean value of `timeseries` data
median (timeseries)	Median value of `timeseries` data

min (timeseries)	Minimum value of `timeseries` data
std (timeseries)	Standard deviation of `timeseries` data
sum (timeseries)	Sum of `timeseries` data
var (timeseries)	Variance of `timeseries` data

Time Series Collections

General Purpose (p. 1-47)	Query and set `tscollection` object properties, plot `tscollection` objects
Data Manipulation (p. 1-48)	Add or delete data, manipulate `tscollection` objects

General Purpose

get (tscollection)	Query `tscollection` object property values
isempty (tscollection)	Determine whether `tscollection` object is empty
length (tscollection)	Length of time vector
plot (timeseries)	Plot time series
set (tscollection)	Set properties of `tscollection` object
size (tscollection)	Size of `tscollection` object
tscollection	Create `tscollection` object
tstool	Open Time Series Tools GUI

Data Manipulation

addsampletocollection	Add sample to `tscollection` object
addts	Add `timeseries` object to `tscollection` object
delsamplefromcollection	Remove sample from `tscollection` object
getabstime (tscollection)	Extract date-string time vector into cell array
getsampleusingtime (tscollection)	Extract data samples into new `tscollection` object
gettimeseriesnames	Cell array of names of `timeseries` objects in `tscollection` object
horzcat (tscollection)	Horizontal concatenation for `tscollection` objects
removets	Remove `timeseries` objects from `tscollection` object
resample (tscollection)	Select or interpolate data in `tscollection` using new time vector
setabstime (tscollection)	Set times of `tscollection` object as date strings
settimeseriesnames	Change name of `timeseries` object in `tscollection`
vertcat (tscollection)	Vertical concatenation for `tscollection` objects

Programming and Data Types

Data Types

Cell Arrays (p. 1-53)	Data of varying types and sizes stored in cells of array
Function Handles (p. 1-54)	Invoke a function indirectly via handle
MATLAB Classes and Objects (p. 1-55)	MATLAB object-oriented class system
Java Classes and Objects (p. 1-55)	Access Java classes through MATLAB interface
Data Type Identification (p. 1-57)	Determine data type of a variable

Numeric Types

arrayfun	Apply function to each element of array
cast	Cast variable to different data type
cat	Concatenate arrays along specified dimension
class	Create object or return class of object
find	Find indices and values of nonzero elements
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
intwarning	Control state of integer warnings
ipermute	Inverse permute dimensions of N-D array
isa	Determine whether input is object of given class
isequal	Test arrays for equality

isequalwithequalnans	Test arrays for equality, treating NaNs as equal
isfinite	Array elements that are finite
isinf	Array elements that are infinite
isnan	Array elements that are NaN
isnumeric	Determine whether input is numeric array
isreal	Determine whether input is real array
isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
permute	Rearrange dimensions of N-D array
realmax	Largest positive floating-point number
realmin	Smallest positive normalized floating-point number
reshape	Reshape array
squeeze	Remove singleton dimensions
zeros	Create array of all zeros

Characters and Strings

See “String Functions” on page 1-63 for all string-related functions.

cellstr	Create cell array of strings from character array
char	Convert to character array (string)
eval	Execute string containing MATLAB expression
findstr	Find string within another, longer string

isstr	Determine whether input is character array
regexp, regexpi	Match regular expression
sprintf	Write formatted data to string
sscanf	Read formatted data from string
strcat	Concatenate strings horizontally
strcmp, strcmpi	Compare strings
strings	MATLAB string handling
strjust	Justify character array
strmatch	Find possible matches for string
strread	Read formatted data from string
strrep	Find and replace substring
strtrim	Remove leading and trailing white space from string
strvcat	Concatenate strings vertically

Structures

arrayfun	Apply function to each element of array
cell2struct	Convert cell array to structure array
class	Create object or return class of object
deal	Distribute inputs to outputs
fieldnames	Field names of structure, or public fields of object
getfield	Field of structure array
isa	Determine whether input is object of given class
isequal	Test arrays for equality

isfield	Determine whether input is structure array field
isscalar	Determine whether input is scalar
isstruct	Determine whether input is structure array
isvector	Determine whether input is vector
orderfields	Order fields of structure array
rmfield	Remove fields from structure
setfield	Set value of structure array field
struct	Create structure array
struct2cell	Convert structure to cell array
structfun	Apply function to each field of scalar structure

Cell Arrays

cell	Construct cell array
cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array
celldisp	Cell array contents
cellfun	Apply function to each cell in cell array
cellplot	Graphically display structure of cell array
cellstr	Create cell array of strings from character array
class	Create object or return class of object
deal	Distribute inputs to outputs

isa	Determine whether input is object of given class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
isequal	Test arrays for equality
isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
struct2cell	Convert structure to cell array

Function Handles

class	Create object or return class of object
feval	Evaluate function
func2str	Construct function name string from function handle
functions	Information about function handle
function_handle (@)	Handle used in calling functions indirectly
isa	Determine whether input is object of given class
isequal	Test arrays for equality
str2func	Construct function handle from function name string

MATLAB Classes and Objects

class	Create object or return class of object
fieldnames	Field names of structure, or public fields of object
inferiorto	Establish inferior class relationship
isa	Determine whether input is object of given class
isobject	Determine whether input is MATLAB OOPs object
loadobj	User-defined extension of `load` function for user objects
methods	Information on class methods
methodsview	Information on class methods in separate window
saveobj	User-defined extension of `save` function for user objects
subsasgn	Subscripted assignment for objects
subsindex	Subscripted indexing for objects
subsref	Subscripted reference for objects
substruct	Create structure argument for `subsasgn` or `subsref`
superiorto	Establish superior class relationship

Java Classes and Objects

cell	Construct cell array
class	Create object or return class of object
clear	Remove items from workspace, freeing up system memory
depfun	List dependencies of M-file or P-file

exist	Check existence of variable, function, directory, or Java class
fieldnames	Field names of structure, or public fields of object
im2java	Convert image to Java image
import	Add package or class to current Java import list
inmem	Names of M-files, MEX-files, Java classes in memory
isa	Determine whether input is object of given class
isjava	Determine whether input is Java object
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Set and get dynamic Java class path
javaMethod	Invoke Java method
javaObject	Construct Java object
javarmpath	Remove entries from dynamic Java class path
methods	Information on class methods
methodsview	Information on class methods in separate window
usejava	Determine whether Java feature is supported in MATLAB
which	Locate functions and files

Data Type Identification

is*	Detect state
isa	Determine whether input is object of given class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isfield	Determine whether input is structure array field
isfloat	Determine whether input is floating-point array
isinteger	Determine whether input is integer array
isjava	Determine whether input is Java object
islogical	Determine whether input is logical array
isnumeric	Determine whether input is numeric array
isobject	Determine whether input is MATLAB OOPs object
isreal	Determine whether input is real array
isstr	Determine whether input is character array
isstruct	Determine whether input is structure array

validateattributes	Check validity of array
who, whos	List variables in workspace

Data Type Conversion

Numeric (p. 1-58)	Convert data of one numeric type to another numeric type
String to Numeric (p. 1-58)	Convert characters to numeric equivalent
Numeric to String (p. 1-59)	Convert numeric to character equivalent
Other Conversions (p. 1-59)	Convert to structure, cell array, function handle, etc.

Numeric

cast	Cast variable to different data type
double	Convert to double precision
int8, int16, int32, int64	Convert to signed integer
single	Convert to single precision
typecast	Convert data types without changing underlying data
uint8, uint16, uint32, uint64	Convert to unsigned integer

String to Numeric

base2dec	Convert base N number string to decimal number
bin2dec	Convert binary number string to decimal number
cast	Cast variable to different data type

hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
str2double	Convert string to double-precision value
str2num	Convert string to number
unicode2native	Convert Unicode characters to numeric bytes

Numeric to String

cast	Cast variable to different data type
char	Convert to character array (string)
dec2base	Convert decimal to base N number in string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
int2str	Convert integer to string
mat2str	Convert matrix to string
native2unicode	Convert numeric bytes to Unicode characters
num2str	Convert number to string

Other Conversions

cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array

datestr	Convert date and time to string format
func2str	Construct function name string from function handle
logical	Convert numeric values to logical
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
num2hex	Convert singles and doubles to IEEE hexadecimal strings
str2func	Construct function handle from function name string
str2mat	Form blank-padded character matrix from strings
struct2cell	Convert structure to cell array

Operators and Special Characters

Arithmetic Operators (p. 1-60)	Plus, minus, power, left and right divide, transpose, etc.
Relational Operators (p. 1-61)	Equal to, greater than, less than or equal to, etc.
Logical Operators (p. 1-61)	Element-wise and short circuit and, or, not
Special Characters (p. 1-62)	Array constructors, line continuation, comments, etc.

Arithmetic Operators

+	Plus
-	Minus

.	Decimal point
=	Assignment
*	Matrix multiplication
/	Matrix right division
\	Matrix left division
^	Matrix power
'	Matrix transpose
.*	Array multiplication (element-wise)
./	Array right division (element-wise)
.\	Array left division (element-wise)
.^	Array power (element-wise)
.'	Array transpose

Relational Operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Logical Operators

See also "Logical Functions" on page 1-66 for functions like `xor`, `all`, `any`, etc.

&&	Logical AND
||	Logical OR
&	Logical AND for arrays

|	Logical OR for arrays
~	Logical NOT

Special Characters

:	Create vectors, subscript arrays, specify for-loop iterations
( )	Pass function arguments, prioritize operators
[ ]	Construct array, concatenate elements, specify multiple outputs from function
{ }	Construct cell array, index into cell array
.	Insert decimal point, define structure field, reference methods of object
.( )	Reference dynamic field of structure
..	Reference parent directory
...	Continue statement to next line
,	Separate rows of array, separate function input/output arguments, separate commands
;	Separate columns of array, suppress output from current command
%	Insert comment line into code
%{ %}	Insert block of comments into code
!	Issue command to operating system
' '	Construct character array
@	Construct function handle, reference class directory

String Functions

Description of Strings in MATLAB

strings	MATLAB string handling

String Creation

blanks	Create string of blank characters
cellstr	Create cell array of strings from character array
char	Convert to character array (string)
sprintf	Write formatted data to string
strcat	Concatenate strings horizontally
strvcat	Concatenate strings vertically

String Identification

class	Create object or return class of object
isa	Determine whether input is object of given class
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isletter	Array elements that are alphabetic letters
isscalar	Determine whether input is scalar
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
isvector	Determine whether input is vector
validatestring	Check validity of text string

String Manipulation

deblank	Strip trailing blanks from end of string
lower	Convert string to lowercase
strjust	Justify character array
strrep	Find and replace substring
strtrim	Remove leading and trailing white space from string
upper	Convert string to uppercase

String Parsing

findstr	Find string within another, longer string
regexp, regexpi	Match regular expression
regexprep	Replace string using regular expression
regexptranslate	Translate string into regular expression
sscanf	Read formatted data from string
strfind	Find one string within another
strread	Read formatted data from string
strtok	Selected parts of string

String Evaluation

eval	Execute string containing MATLAB expression
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace

String Comparison

strcmp, strcmpi	Compare strings
strmatch	Find possible matches for string
strncmp, strncmpi	Compare first n characters of strings

Bit-wise Functions

bitand	Bitwise AND
bitcmp	Bitwise complement
bitget	Bit at specified position
bitmax	Maximum double-precision floating-point integer
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR
swapbytes	Swap byte ordering

Logical Functions

all	Determine whether all array elements are nonzero
and	Find logical AND of array or scalar inputs
any	Determine whether any array elements are nonzero
false	Logical 0 (false)
find	Find indices and values of nonzero elements
isa	Determine whether input is object of given class
iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
logical	Convert numeric values to logical

not	Find logical NOT of array or scalar input
or	Find logical OR of array or scalar inputs
true	Logical 1 (true)
xor	Logical exclusive-OR

See “Operators and Special Characters” on page 1-60 for logical operators.

Relational Functions

eq	Test for equality
ge	Test for greater than or equal to
gt	Test for greater than
le	Test for less than or equal to
lt	Test for less than
ne	Test for inequality

See “Operators and Special Characters” on page 1-60 for relational operators.

Set Functions

intersect	Find set intersection of two vectors
ismember	Array elements that are members of set
issorted	Determine whether set elements are in sorted order
setdiff	Find set difference of two vectors
setxor	Find set exclusive OR of two vectors
union	Find set union of two vectors
unique	Find unique elements of vector

Date and Time Functions

addtodate	Modify date number by field
calendar	Calendar for specified month
clock	Current time as date vector
cputime	Elapsed CPU time
date	Current date string
datenum	Convert date and time to serial date number
datestr	Convert date and time to string format
datevec	Convert date and time to vector of components
eomday	Last day of month
etime	Time elapsed between date vectors
now	Current date and time
weekday	Day of week

Programming in MATLAB

M-File Functions and Scripts (p. 1-69)	Declare functions, handle arguments, identify dependencies, etc.
Evaluation of Expressions and Functions (p. 1-70)	Evaluate expression in string, apply function to array, run script file, etc.
Timer Functions (p. 1-71)	Schedule execution of MATLAB commands
Variables and Functions in Memory (p. 1-72)	List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches

Control Flow (p. 1-73)	if-then-else, for loops, switch-case, try-catch
Error Handling (p. 1-74)	Generate warnings and errors, test for and catch errors, retrieve most recent error message
MEX Programming (p. 1-75)	Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

M-File Functions and Scripts

addOptional (inputParser)	Add optional argument to `inputParser` schema
addParamValue (inputParser)	Add parameter-value argument to `inputParser` schema
addRequired (inputParser)	Add required argument to `inputParser` schema
createCopy (inputParser)	Create copy of `inputParser` object
depdir	List dependent directories of M-file or P-file
depfun	List dependencies of M-file or P-file
echo	Echo M-files during execution
end	Terminate block of code, or indicate last array index
function	Declare M-file function
input	Request user input
inputname	Variable name of function input
inputParser	Construct input parser object
mfilename	Name of currently running M-file
namelengthmax	Maximum identifier length
nargchk	Validate number of input arguments

nargin, nargout	Number of function arguments
nargoutchk	Validate number of output arguments
parse (inputParser)	Parse and validate named inputs
pcode	Create preparsed pseudocode file (P-file)
script	Script M-file description
syntax	Two ways to call MATLAB functions
varargin	Variable length input argument list
varargout	Variable length output argument list

Evaluation of Expressions and Functions

ans	Most recent answer
arrayfun	Apply function to each element of array
assert	Generate error when condition is violated
builtin	Execute built-in function from overloaded method
cellfun	Apply function to each cell in cell array
echo	Echo M-files during execution
eval	Execute string containing MATLAB expression
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace
feval	Evaluate function

iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
pause	Halt execution temporarily
run	Run script that is not on current path
script	Script M-file description
structfun	Apply function to each field of scalar structure
symvar	Determine symbolic variables in expression
tic, toc	Measure performance using stopwatch timer

Timer Functions

delete (timer)	Remove timer object from memory
disp (timer)	Information about timer object
get (timer)	Timer object properties
isvalid (timer)	Determine whether timer object is valid
set (timer)	Configure or display timer object properties
start	Start timer(s) running
startat	Start timer(s) running at specified time
stop	Stop timer(s)
timer	Construct timer object
timerfind	Find timer objects

timerfindall	Find timer objects, including invisible objects
wait	Wait until timer stops running

Variables and Functions in Memory

ans	Most recent answer
assignin	Assign value to variable in specified workspace
datatipinfo	Produce short description of input variable
genvarname	Construct valid variable name from string
global	Declare global variables
inmem	Names of M-files, MEX-files, Java classes in memory
isglobal	Determine whether input is global variable
mislocked	Determine whether M-file or MEX-file cannot be cleared from memory
mlock	Prevent clearing M-file or MEX-file from memory
munlock	Allow clearing M-file or MEX-file from memory
namelengthmax	Maximum identifier length
pack	Consolidate workspace memory
persistent	Define persistent variable
rehash	Refresh function and file system path caches

Control Flow

break	Terminate execution of `for` or `while` loop
case	Execute block of code if condition is `true`
catch	Specify how to respond to error in `try` statement
continue	Pass control to next iteration of `for` or `while` loop
else	Execute statements if condition is false
elseif	Execute statements if additional condition is true
end	Terminate block of code, or indicate last array index
error	Display message and abort function
for	Execute block of code specified number of times
if	Execute statements if condition is true
otherwise	Default part of switch statement
return	Return to invoking function
switch	Switch among several cases, based on expression
try	Attempt to execute block of code, and catch errors
while	Repeatedly execute statements while condition is true

Error Handling

addCause (MException)	Append `MException` objects
assert	Generate error when condition is violated
catch	Specify how to respond to error in `try` statement
disp (MException)	Display `MException` object
eq (MException)	Compare `MException` objects for equality
error	Display message and abort function
ferror	Query MATLAB about errors in file input or output
getReport (MException)	Get error message for exception
intwarning	Control state of integer warnings
isequal (MException)	Compare `MException` objects for equality
last (MException)	Last uncaught exception
lasterr	Last error message
lasterror	Last error message and related information
lastwarn	Last warning message
MException	Construct `MException` object
ne (MException)	Compare `MException` objects for inequality
rethrow	Reissue error
rethrow (MException)	Reissue existing exception
throw (MException)	Terminate function and issue exception

try	Attempt to execute block of code, and catch errors
warning	Warning message

MEX Programming

dbmex	Enable MEX-file debugging
inmem	Names of M-files, MEX-files, Java classes in memory
mex	Compile MEX-function from C, C++, or Fortran source code
mexext	MEX-filename extension

File I/O

File Name Construction (p. 1-76)	Get path, directory, filename information; construct filenames
Opening, Loading, Saving Files (p. 1-77)	Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-77)	Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-77)	Low-level operations that use a file identifier
Text Files (p. 1-78)	Delimited or formatted I/O to text files
XML Documents (p. 1-79)	Documents written in Extensible Markup Language
Spreadsheets (p. 1-79)	Excel and Lotus 1-2-3 files
Scientific Data (p. 1-80)	CDF, FITS, HDF formats
Audio and Audio/Video (p. 1-81)	General audio functions; SparcStation, WAVE, AVI files
Images (p. 1-83)	Graphics files
Internet Exchange (p. 1-84)	URL, FTP, zip, tar, and e-mail

To see a listing of file formats that are readable from MATLAB, go to file formats.

File Name Construction

filemarker	Character to separate file name and internal function name
fileparts	Parts of file name and path
filesep	Directory separator for current platform
fullfile	Build full filename from parts

tempdir	Name of system's temporary directory
tempname	Unique name for temporary file

Opening, Loading, Saving Files

daqread	Read Data Acquisition Toolbox (.daq) file
filehandle	Construct file handle object
importdata	Load data from disk file
load	Load workspace variables from disk
open	Open files based on extension
save	Save workspace variables to disk
uiimport	Open Import Wizard to import data
winopen	Open file in appropriate application (Windows)

Memory Mapping

disp (memmapfile)	Information about memmapfile object
get (memmapfile)	Memmapfile object properties
memmapfile	Construct memmapfile object

Low-Level File I/O

fclose	Close one or more open files
feof	Test for end-of-file
ferror	Query MATLAB about errors in file input or output

fgetl	Read line from file, discarding newline character
fgets	Read line from file, keeping newline character
fopen	Open file, or obtain information about open files
fprintf	Write formatted data to file
fread	Read binary data from file
frewind	Move file position indicator to beginning of open file
fscanf	Read formatted data from file
fseek	Set file position indicator
ftell	File position indicator
fwrite	Write binary data to file

Text Files

csvread	Read comma-separated value file
csvwrite	Write comma-separated value file
dlmread	Read ASCII-delimited file of numeric data into matrix
dlmwrite	Write matrix to ASCII-delimited file
textread	Read data from text file; write to multiple outputs
textscan	Read formatted data from text file or string

XML Documents

xmlread	Parse XML document and return Document Object Model node
xmlwrite	Serialize XML Document Object Model node
xslt	Transform XML document using XSLT engine

Spreadsheets

Microsoft Excel Functions (p. 1-79)	Read and write Microsoft Excel spreadsheet
Lotus 1-2-3 Functions (p. 1-79)	Read and write Lotus WK1 spreadsheet

Microsoft Excel Functions

xlsfinfo	Determine whether file contains Microsoft Excel (`.xls`) spreadsheet
xlsread	Read Microsoft Excel spreadsheet file (`.xls`)
xlswrite	Write Microsoft Excel spreadsheet file (`.xls`)

Lotus 1-2-3 Functions

wk1finfo	Determine whether file contains 1-2-3 WK1 worksheet
wk1read	Read Lotus 1-2-3 WK1 spreadsheet file into matrix
wk1write	Write matrix to Lotus 1-2-3 WK1 spreadsheet file

Scientific Data

Common Data Format (CDF) (p. 1-80)	Work with CDF files
Flexible Image Transport System (p. 1-80)	Work with FITS files
Hierarchical Data Format (HDF) (p. 1-81)	Work with HDF files
Band-Interleaved Data (p. 1-81)	Work with band-interleaved files

Common Data Format (CDF)

cdfepoch	Construct `cdfepoch` object for Common Data Format (CDF) export
cdfinfo	Information about Common Data Format (CDF) file
cdfread	Read data from Common Data Format (CDF) file
cdfwrite	Write data to Common Data Format (CDF) file
todatenum	Convert CDF epoch object to MATLAB datenum

Flexible Image Transport System

fitsinfo	Information about FITS file
fitsread	Read data from FITS file

Hierarchical Data Format (HDF)

hdf	Summary of MATLAB HDF4 capabilities
hdf5	Summary of MATLAB HDF5 capabilities
hdf5info	Information about HDF5 file
hdf5read	Read HDF5 file
hdf5write	Write data to file in HDF5 format
hdfinfo	Information about HDF4 or HDF-EOS file
hdfread	Read data from HDF4 or HDF-EOS file
hdftool	Browse and import data from HDF4 or HDF-EOS files

Band-Interleaved Data

multibandread	Read band-interleaved data from binary file
multibandwrite	Write band-interleaved data to file

Audio and Audio/Video

General (p. 1-82)	Create audio player object, obtain information about multimedia files, convert to/from audio signal
SPARCstation-Specific Sound Functions (p. 1-82)	Access NeXT/SUN (`.au`) sound files

Microsoft WAVE Sound Functions (p. 1-83)	Access Microsoft WAVE (`.wav`) sound files
Audio/Video Interleaved (AVI) Functions (p. 1-83)	Access Audio/Video interleaved (`.avi`) sound files

General

audioplayer	Create audio player object
audiorecorder	Create audio recorder object
beep	Produce beep sound
lin2mu	Convert linear audio signal to mu-law
mmfileinfo	Information about multimedia file
mmreader	Create multimedia reader object for reading video files
mu2lin	Convert mu-law audio signal to linear
read	Read video frame data from multimedia reader object
sound	Convert vector into sound
soundsc	Scale data and play as sound

SPARCstation-Specific Sound Functions

aufinfo	Information about NeXT/SUN (`.au`) sound file
auread	Read NeXT/SUN (`.au`) sound file
auwrite	Write NeXT/SUN (`.au`) sound file

Microsoft WAVE Sound Functions

wavfinfo	Information about Microsoft WAVE (`.wav`) sound file
wavplay	Play recorded sound on PC-based audio output device
wavread	Read Microsoft WAVE (`.wav`) sound file
wavrecord	Record sound using PC-based audio input device
wavwrite	Write Microsoft WAVE (`.wav`) sound file

Audio/Video Interleaved (AVI) Functions

addframe	Add frame to Audio/Video Interleaved (AVI) file
avifile	Create new Audio/Video Interleaved (AVI) file
aviinfo	Information about Audio/Video Interleaved (AVI) file
aviread	Read Audio/Video Interleaved (AVI) file
close (avifile)	Close Audio/Video Interleaved (AVI) file
movie2avi	Create Audio/Video Interleaved (AVI) movie from MATLAB movie

Images

exifread	Read EXIF information from JPEG and TIFF image files
im2java	Convert image to Java image

imfinfo	Information about graphics file
imread	Read image from graphics file
imwrite	Write image to graphics file

Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-84)	Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files
FTP Functions (p. 1-84)	Connect to FTP server, download from server, manage FTP files, close server connection

URL, Zip, Tar, E-Mail

gunzip	Uncompress GNU zip files
gzip	Compress files into GNU zip files
sendmail	Send e-mail message to address list
tar	Compress files into tar file
untar	Extract contents of tar file
unzip	Extract contents of zip file
urlread	Read content at URL
urlwrite	Save contents of URL to file
zip	Compress files into zip file

FTP Functions

ascii	Set FTP transfer type to ASCII
binary	Set FTP transfer type to binary

cd (ftp)	Change current directory on FTP server
close (ftp)	Close connection to FTP server
delete (ftp)	Remove file on FTP server
dir (ftp)	Directory contents on FTP server
ftp	Connect to FTP server, creating FTP object
mget	Download file from FTP server
mkdir (ftp)	Create new directory on FTP server
mput	Upload file or directory to FTP server
rename	Rename file on FTP server
rmdir (ftp)	Remove directory on FTP server

Graphics

Basic Plots and Graphs (p. 1-86)	Linear line plots, log and semilog plots
Plotting Tools (p. 1-87)	GUIs for interacting with plots
Annotating Plots (p. 1-87)	Functions for and properties of titles, axes labels, legends, mathematical symbols
Specialized Plotting (p. 1-88)	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images (p. 1-92)	Display image object, read and write graphics file, convert to movie frames
Printing (p. 1-92)	Printing and exporting figures to standard formats
Handle Graphics (p. 1-93)	Creating graphics objects, setting properties, finding handles

Basic Plots and Graphs

box	Axes border
errorbar	Plot error bars along curve
hold	Retain current graph in figure
LineSpec	Line specification string syntax
loglog	Log-log scale plot
plot	2-D line plot
plot3	3-D line plot
plotyy	2-D line plots with y-axes on both left and right side
polar	Polar coordinate plot

semilogx, semilogy	Semilogarithmic plots
subplot	Create axes in tiled positions

Plotting Tools

figurepalette	Show or hide figure palette
pan	Pan view of graph interactively
plotbrowser	Show or hide figure plot browser
plotedit	Interactively edit and annotate plots
plottools	Show or hide plot tools
propertyeditor	Show or hide property editor
rotate3d	Rotate 3-D view using mouse
showplottool	Show or hide figure plot tool
zoom	Turn zooming on or off or magnify by factor

Annotating Plots

annotation	Create annotation objects
clabel	Contour plot elevation labels
datacursormode	Enable or disable interactive data cursor mode
datetick	Date formatted tick labels
gtext	Mouse placement of text in 2-D view
legend	Graph legend for lines and patches
line	Create line object
rectangle	Create 2-D rectangle object
texlabel	Produce TeX format from character string

title	Add title to current axes
xlabel, ylabel, zlabel	Label *x*-, *y*-, and *z*-axis

Specialized Plotting

Area, Bar, and Pie Plots (p. 1-88)	1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-89)	Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-89)	Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-89)	Stair, step, and stem plots
Function Plots (p. 1-89)	Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-90)	Plots for showing distributions of data
Polygons and Surfaces (p. 1-90)	Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-91)	Plots of point distributions
Animation (p. 1-91)	Functions to create and play movies of plots

Area, Bar, and Pie Plots

area	Filled area 2-D plot
bar, barh	Plot bar graph (vertical and horizontal)
bar3, bar3h	Plot 3-D bar chart
pareto	Pareto chart
pie	Pie chart
pie3	3-D pie chart

Contour Plots

contour	Contour plot of matrix
contour3	3-D contour plot
contourc	Low-level contour plot computation
contourf	Filled 2-D contour plot
ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter

Direction and Velocity Plots

comet	2-D comet plot
comet3	3-D comet plot
compass	Plot arrows emanating from origin
feather	Plot velocity vectors
quiver	Quiver or velocity plot
quiver3	3-D quiver or velocity plot

Discrete Data Plots

stairs	Stairstep graph
stem	Plot discrete sequence data
stem3	Plot 3-D discrete sequence data

Function Plots

ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter
ezmesh	Easy-to-use 3-D mesh plotter

ezmeshc	Easy-to-use combination mesh/contour plotter
ezplot	Easy-to-use function plotter
ezplot3	Easy-to-use 3-D parametric curve plotter
ezpolar	Easy-to-use polar coordinate plotter
ezsurf	Easy-to-use 3-D colored surface plotter
ezsurfc	Easy-to-use combination surface/contour plotter
fplot	Plot function between specified limits

Histograms

hist	Histogram plot
histc	Histogram count
rose	Angle histogram plot

Polygons and Surfaces

convhull	Convex hull
cylinder	Generate cylinder
delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
ellipsoid	Generate ellipsoid

fill	Filled 2-D polygons
fill3	Filled 3-D polygons
inpolygon	Points inside polygonal region
pcolor	Pseudocolor (checkerboard) plot
polyarea	Area of polygon
rectint	Rectangle intersection area
ribbon	Ribbon plot
slice	Volumetric slice plot
sphere	Generate sphere
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search
voronoi	Voronoi diagram
waterfall	Waterfall plot

Scatter/Bubble Plots

plotmatrix	Scatter plot matrix
scatter	Scatter plot
scatter3	3-D scatter plot

Animation

frame2im	Convert movie frame to indexed image
getframe	Capture movie frame
im2frame	Convert image to movie frame

movie	Play recorded movie frames
noanimate	Change `EraseMode` of all objects to `normal`

Bit-Mapped Images

frame2im	Convert movie frame to indexed image
im2frame	Convert image to movie frame
im2java	Convert image to Java image
image	Display image object
imagesc	Scale data and display image object
imfinfo	Information about graphics file
imformats	Manage image file format registry
imread	Read image from graphics file
imwrite	Write image to graphics file
ind2rgb	Convert indexed image to RGB image

Printing

frameedit	Edit print frames for Simulink and Stateflow block diagrams
hgexport	Export figure
orient	Hardcopy paper orientation
print, printopt	Print figure or save to file and configure printer defaults
printdlg	Print dialog box

printpreview	Preview figure to print
saveas	Save figure or Simulink block diagram using specified format

Handle Graphics

Finding and Identifying Graphics Objects (p. 1-93)	Find and manipulate graphics objects via their handles
Object Creation Functions (p. 1-94)	Constructors for core graphics objects
Plot Objects (p. 1-94)	Property descriptions for plot objects
Figure Windows (p. 1-95)	Control and save figures
Axes Operations (p. 1-96)	Operate on axes objects
Operating on Object Properties (p. 1-96)	Query, set, and link object properties

Finding and Identifying Graphics Objects

allchild	Find all children of specified objects
ancestor	Ancestor of graphics object
copyobj	Copy graphics objects and their descendants
delete	Remove files or graphics objects
findall	Find all graphics objects
findfigs	Find visible offscreen figures
findobj	Locate graphics objects with specific properties
gca	Current axes handle
gcbf	Handle of figure containing object whose callback is executing

gcbo	Handle of object whose callback is executing
gco	Handle of current object
get	Query object properties
ishandle	Is object handle valid
propedit	Open Property Editor
set	Set object properties

Object Creation Functions

axes	Create axes graphics object
figure	Create figure graphics object
hggroup	Create hggroup object
hgtransform	Create hgtransform graphics object
image	Display image object
light	Create light object
line	Create line object
patch	Create patch graphics object
rectangle	Create 2-D rectangle object
root object	Root object properties
surface	Create surface object
text	Create text object in current axes
uicontextmenu	Create context menu

Plot Objects

Annotation Arrow Properties	Define annotation arrow properties
Annotation Doublearrow Properties	Define annotation doublearrow properties

Annotation Ellipse Properties	Define annotation ellipse properties
Annotation Line Properties	Define annotation line properties
Annotation Rectangle Properties	Define annotation rectangle properties
Annotation Textarrow Properties	Define annotation textarrow properties
Annotation Textbox Properties	Define annotation textbox properties
Areaseries Properties	Define areaseries properties
Barseries Properties	Define barseries properties
Contourgroup Properties	Define contourgroup properties
Errorbarseries Properties	Define errorbarseries properties
Image Properties	Define image properties
Lineseries Properties	Define lineseries properties
Quivergroup Properties	Define quivergroup properties
Scattergroup Properties	Define scattergroup properties
Stairseries Properties	Define stairseries properties
Stemseries Properties	Define stemseries properties
Surfaceplot Properties	Define surfaceplot properties

Figure Windows

clf	Clear current figure window
close	Remove specified figure
closereq	Default figure close request function
drawnow	Flushes event queue and updates figure window
gcf	Current figure handle
hgload	Load Handle Graphics object hierarchy from file

hgsave	Save Handle Graphics object hierarchy to file
newplot	Determine where to draw graphics objects
opengl	Control OpenGL rendering
refresh	Redraw current figure
saveas	Save figure or Simulink block diagram using specified format

Axes Operations

axis	Axis scaling and appearance
box	Axes border
cla	Clear current axes
gca	Current axes handle
grid	Grid lines for 2-D and 3-D plots
ishold	Current hold state
makehgtform	Create 4-by-4 transform matrix

Operating on Object Properties

get	Query object properties
linkaxes	Synchronize limits of specified 2-D axes
linkprop	Keep same value for corresponding properties
refreshdata	Refresh data in graph when data source is specified
set	Set object properties

3-D Visualization

Surface and Mesh Plots

Creating Surfaces and Meshes

hidden	Remove hidden lines from mesh plot
mesh, meshc, meshz	Mesh plots
peaks	Example function of two variables
surf, surfc	3-D shaded surface plot
surface	Create surface object
surfl	Surface plot with colormap-based lighting

tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot

Domain Generation

griddata	Data gridding
meshgrid	Generate `X` and `Y` arrays for 3-D plots

Color Operations

brighten	Brighten or darken colormap
caxis	Color axis scaling
colorbar	Colorbar showing color scale
colordef	Set default property values to display different color schemes
colormap	Set and get current colormap
colormapeditor	Start colormap editor
ColorSpec	Color specification
graymon	Set default figure properties for grayscale monitors
hsv2rgb	Convert HSV colormap to RGB colormap
rgb2hsv	Convert RGB colormap to HSV colormap
rgbplot	Plot colormap
shading	Set color shading properties
spinmap	Spin colormap

surfnorm	Compute and display 3-D surface normals
whitebg	Change axes background color

Colormaps

contrast	Grayscale colormap for contrast enhancement

View Control

Controlling the Camera Viewpoint (p. 1-99)	Orbiting, dollying, pointing, rotating camera positions and setting fields of view
Setting the Aspect Ratio and Axis Limits (p. 1-100)	Specifying what portions of axes to view and how to scale them
Object Manipulation (p. 1-100)	Panning, rotating, and zooming views
Selecting Region of Interest (p. 1-101)	Interactively identifying rectangular regions

Controlling the Camera Viewpoint

camdolly	Move camera position and target
cameratoolbar	Control camera toolbar programmatically
camlookat	Position camera to view object or group of objects
camorbit	Rotate camera position around camera target
campan	Rotate camera target around camera position

campos	Set or query camera position
camproj	Set or query projection type
camroll	Rotate camera about view axis
camtarget	Set or query location of camera target
camup	Set or query camera up vector
camva	Set or query camera view angle
camzoom	Zoom in and out on scene
makehgtform	Create 4-by-4 transform matrix
view	Viewpoint specification
viewmtx	View transformation matrices

Setting the Aspect Ratio and Axis Limits

daspect	Set or query axes data aspect ratio
pbaspect	Set or query plot box aspect ratio
xlim, ylim, zlim	Set or query axis limits

Object Manipulation

pan	Pan view of graph interactively
reset	Reset graphics object properties to their defaults
rotate	Rotate object in specified direction
rotate3d	Rotate 3-D view using mouse
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
zoom	Turn zooming on or off or magnify by factor

Selecting Region of Interest

dragrect	Drag rectangles with mouse
rbbox	Create rubberband box for area selection

Lighting

camlight	Create or move light object in camera coordinates
diffuse	Calculate diffuse reflectance
light	Create light object
lightangle	Create or position `light` object in spherical coordinates
lighting	Specify lighting algorithm
material	Control reflectance properties of surfaces and patches
specular	Calculate specular reflectance

Transparency

alim	Set or query axes alpha limits
alpha	Set transparency properties for objects in current axes
alphamap	Specify figure alphamap (transparency)

Volume Visualization

coneplot	Plot velocity vectors as cones in 3-D vector field
contourslice	Draw contours in volume slice planes
curl	Compute curl and angular velocity of vector field
divergence	Compute divergence of vector field
flow	Simple function of three variables
interpstreamspeed	Interpolate stream-line vertices from flow speed
isocaps	Compute isosurface end-cap geometry
isocolors	Calculate isosurface and patch colors
isonormals	Compute normals of isosurface vertices
isosurface	Extract isosurface data from volume data
reducepatch	Reduce number of patch faces
reducevolume	Reduce number of elements in volume data set
shrinkfaces	Reduce size of patch faces
slice	Volumetric slice plot
smooth3	Smooth 3-D data
stream2	Compute 2-D streamline data
stream3	Compute 3-D streamline data
streamline	Plot streamlines from 2-D or 3-D vector data
streamparticles	Plot stream particles
streamribbon	3-D stream ribbon plot from vector volume data

streamslice	Plot streamlines in slice planes
streamtube	Create 3-D stream tube plot
subvolume	Extract subset of volume data set
surf2patch	Convert surface data to patch data
volumebounds	Coordinate and color limits for volume data

Creating Graphical User Interfaces

Predefined Dialog Boxes

dialog	Create and display dialog box
errordlg	Create and open error dialog box
export2wsdlg	Export variables to workspace
helpdlg	Create and open help dialog box
inputdlg	Create and open input dialog box
listdlg	Create and open list-selection dialog box
msgbox	Create and open message box
printdlg	Print dialog box
printpreview	Preview figure to print
questdlg	Create and open question dialog box
uigetdir	Open standard dialog box for selecting a directory

uigetfile	Open standard dialog box for retrieving files
uigetpref	Open dialog box for retrieving preferences
uiopen	Open file selection dialog box with appropriate file filters
uiputfile	Open standard dialog box for saving files
uisave	Open standard dialog box for saving workspace variables
uisetcolor	Open standard dialog box for setting object's `ColorSpec`
uisetfont	Open standard dialog box for setting object's font characteristics
waitbar	Open waitbar
warndlg	Open warning dialog box

Deploying User Interfaces

guidata	Store or retrieve GUI data
guihandles	Create structure of handles
movegui	Move GUI figure to specified location on screen
openfig	Open new copy or raise existing copy of saved figure

Developing User Interfaces

addpref	Add preference
getappdata	Value of application-defined data
getpref	Preference

ginput	Graphical input from mouse or cursor
guidata	Store or retrieve GUI data
guide	Open GUI Layout Editor
inspect	Open Property Inspector
isappdata	True if application-defined data exists
ispref	Test for existence of preference
rmappdata	Remove application-defined data
rmpref	Remove preference
setappdata	Specify application-defined data
setpref	Set preference
uigetpref	Open dialog box for retrieving preferences
uisetpref	Manage preferences used in `uigetpref`
waitfor	Wait for condition before resuming execution
waitforbuttonpress	Wait for key press or mouse-button click

User Interface Objects

menu	Generate menu of choices for user input
uibuttongroup	Create container object to exclusively manage radio buttons and toggle buttons
uicontextmenu	Create context menu
uicontrol	Create user interface control object

uimenu	Create menus on figure windows
uipanel	Create panel container object
uipushtool	Create push button on toolbar
uitoggletool	Create toggle button on toolbar
uitoolbar	Create toolbar on figure

Finding Objects from Callbacks

findall	Find all graphics objects
findfigs	Find visible offscreen figures
findobj	Locate graphics objects with specific properties
gcbf	Handle of figure containing object whose callback is executing
gcbo	Handle of object whose callback is executing

GUI Utility Functions

align	Align user interface controls (uicontrols) and axes
getpixelposition	Get component position in pixels
listfonts	List available system fonts
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
setpixelposition	Set component position in pixels
textwrap	Wrapped string matrix for given uicontrol
uistack	Reorder visual stacking order of objects

Controlling Program Execution

uiresume, uiwait	Control program execution

External Interfaces

Dynamic Link Libraries (p. 1-109)	Access functions stored in external shared library (.dll) files
Java (p. 1-110)	Work with objects constructed from Java API and third-party class packages
Component Object Model and ActiveX (p. 1-111)	Integrate COM components into your application
Web Services (p. 1-113)	Communicate between applications over a network using SOAP and WSDL
Serial Port Devices (p. 1-113)	Read and write to devices connected to your computer's serial port

See also MATLAB C and Fortran API Reference for functions you can use in external routines that interact with MATLAB programs and the data in MATLAB workspaces.

Dynamic Link Libraries

calllib	Call function in external library
libfunctions	Information on functions in external library
libfunctionsview	Create window displaying information on functions in external library
libisloaded	Determine whether external library is loaded
libpointer	Create pointer object for use with external libraries
libstruct	Construct structure as defined in external library

loadlibrary	Load external library into MATLAB
unloadlibrary	Unload external library from memory

Java

class	Create object or return class of object
fieldnames	Field names of structure, or public fields of object
import	Add package or class to current Java import list
inspect	Open Property Inspector
isa	Determine whether input is object of given class
isjava	Determine whether input is Java object
ismethod	Determine whether input is object method
isprop	Determine whether input is object property
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Set and get dynamic Java class path
javaMethod	Invoke Java method
javaObject	Construct Java object
javarmpath	Remove entries from dynamic Java class path
methods	Information on class methods

methodsview	Information on class methods in separate window
usejava	Determine whether Java feature is supported in MATLAB

Component Object Model and ActiveX

actxcontrol	Create ActiveX control in figure window
actxcontrollist	List all currently installed ActiveX controls
actxcontrolselect	Open GUI to create ActiveX control
actxGetRunningServer	Get handle to running instance of Automation server
actxserver	Create COM server
addproperty	Add custom property to object
class	Create object or return class of object
delete (COM)	Remove COM control or server
deleteproperty	Remove custom property from object
enableservice	Enable, disable, or report status of Automation server
eventlisteners	List of events attached to listeners
events	List of events control can trigger
Execute	Execute MATLAB command in server
Feval (COM)	Evaluate MATLAB function in server
fieldnames	Field names of structure, or public fields of object
get (COM)	Get property value from interface, or display properties

GetCharArray	Get character array from server
GetFullMatrix	Get matrix from server
GetVariable	Get data from variable in server workspace
GetWorkspaceData	Get data from server workspace
inspect	Open Property Inspector
interfaces	List custom interfaces to COM server
invoke	Invoke method on object or interface, or display methods
isa	Determine whether input is object of given class
iscom	Is input COM object
isevent	Is input event
isinterface	Is input COM interface
ismethod	Determine whether input is object method
isprop	Determine whether input is object property
load (COM)	Initialize control object from file
MaximizeCommandWindow	Open server window on Windows desktop
methods	Information on class methods
methodsview	Information on class methods in separate window
MinimizeCommandWindow	Minimize size of server window
move	Move or resize control in parent window
propedit (COM)	Open built-in property page for control
PutCharArray	Store character array in server

PutFullMatrix	Store matrix in server
PutWorkspaceData	Store data in server workspace
Quit (COM)	Terminate MATLAB server
registerevent	Register event handler with control's event
release	Release interface
save (COM)	Serialize control object to file
set (COM)	Set object or interface property to specified value
unregisterallevents	Unregister all events for control
unregisterevent	Unregister event handler with control's event

Web Services

callSoapService	Send SOAP message off to endpoint
createClassFromWsdl	Create MATLAB object based on WSDL file
createSoapMessage	Create SOAP message to send to server
parseSoapResponse	Convert response string from SOAP server into MATLAB data types

Serial Port Devices

clear (serial)	Remove serial port object from MATLAB workspace
delete (serial)	Remove serial port object from memory
disp (serial)	Serial port object summary information

fclose (serial)	Disconnect serial port object from device
fgetl (serial)	Read line of text from device and discard terminator
fgets (serial)	Read line of text from device and include terminator
fopen (serial)	Connect serial port object to device
fprintf (serial)	Write text to device
fread (serial)	Read binary data from device
fscanf (serial)	Read data from device, and format as text
fwrite (serial)	Write binary data to device
get (serial)	Serial port object properties
instrcallback	Event information when event occurs
instrfind	Read serial port objects from memory to MATLAB workspace
instrfindall	Find visible and hidden serial port objects
isvalid (serial)	Determine whether serial port objects are valid
length (serial)	Length of serial port object array
load (serial)	Load serial port objects and variables into MATLAB workspace
readasync	Read data asynchronously from device
record	Record data and event information to file
save (serial)	Save serial port objects and variables to MAT-file
serial	Create serial port object

serialbreak	Send break to device connected to serial port
set (serial)	Configure or display serial port object properties
size (serial)	Size of serial port object array
stopasync	Stop asynchronous read and write operations

2

Functions — Alphabetical List

binary
bitand
bitcmp
bitget
bitmax
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
builddocsearchdb
builtin
bsxfun
bvp4c
bvp5c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit
campan
campos
camproj
camroll
camtarget
camup

camva
camzoom
cart2pol
cart2sph
case
cast
cat
catch
caxis
cd
cd (ftp)
cdf2rdf
cdfepoch
cdfinfo
cdfread
cdfwrite
ceil
cell
cell2mat
cell2struct
celldisp
cellfun
cellplot
cellstr
cgs
char
checkin
checkout
chol
cholinc
cholupdate
circshift
cla
clabel
class
clc
clear

clear (serial)
clf
clipboard
clock
close
close (avifile)
close (ftp)
closereq
cmopts
colamd
colmmd
colorbar
colordef
colormap
colormapeditor
ColorSpec
colperm
comet
comet3
commandhistory
commandwindow
compan
compass
complex
computer
cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contourgroup Properties
contourslice

contrast
conv
conv2
convhull
convhulln
convn
copyfile
copyobj
corrcoef
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
createClassFromWsdl
createCopy (inputParser)
createSoapMessage
cross
csc
cscd
csch
csvread
csvwrite
ctranspose (timeseries)
cumprod
cumsum
cumtrapz
curl
customverctrl
cylinder
daqread
daspect
datacursormode

datatipinfo
date
datenum
datestr
datetick
datevec
dbclear
dbcont
dbdown
dblquad
dbmex
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeadv
ddeexec
ddeget
ddeinit
ddepoke
ddereq
ddesd
ddeset
ddeterm
ddeunadv
deal
deblank
debug
dec2base
dec2bin
dec2hex
decic
deconv

del2
delaunay
delaunay3
delaunayn
delete
delete (COM)
delete (ftp)
delete (serial)
delete (timer)
deleteproperty
delevent
delsample
delsamplefromcollection
demo
depdir
depfun
det
detrend
detrend (timeseries)
deval
diag
dialog
diary
diff
diffuse
dir
dir (ftp)
disp
disp (memmapfile)
disp (MException)
disp (serial)
disp (timer)
display
divergence
dlmread
dlmwrite
dmperm

eventlisteners
events
Execute
exifread
exist
exit
exp
expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh
ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
factor
factorial
false
fclose
fclose (serial)
feather
feof
ferror
feval
Feval (COM)
fft
fft2
fftn
fftshift
fftw
fgetl

fgetl (serial)
fgets
fgets (serial)
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
File Formats
filemarker
fileparts
filehandle
filesep
fill
fill3
filter
filter (timeseries)
filter2
find
findall
findfigs
findobj
findstr
finish
fitsinfo
fitsread
fix
flipdim
fliplr
flipud
floor
flops
flow
fminbnd
fminsearch
fopen

fopen (serial)
for
format
fplot
fprintf
fprintf (serial)
frame2im
frameedit
fread
fread (serial)
freqspace
frewind
fscanf
fscanf (serial)
fseek
ftell
ftp
full
fullfile
func2str
function
function_handle (@)
functions
funm
fwrite
fwrite (serial)
fzero
gallery
gamma, gammainc, gammaln
gca
gcbf
gcbo
gcd
gcf
gco
ge
genpath

genvarname
get
get (COM)
get (memmapfile)
get (serial)
get (timer)
get (timeseries)
get (tscollection)
getabstime (timeseries)
getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getenv
getfield
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getReport (MException)
getsampleusingtime (timeseries)
getsampleusingtime (tscollection)
gettimeseriesnames
gettsafteratevent
gettsafterevent
gettsatevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents
GetVariable
GetWorkspaceData
ginput
global
gmres
gplot

grabcode
gradient
graymon
grid
griddata
griddata3
griddatan
gsvd
gt
gtext
guidata
guide
guihandles
gunzip
gzip
hadamard
hankel
hdf
hdf5
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
hex2dec
hex2num
hgexport
hggroup
Hggroup Properties
hgload

hgsave
hgtransform
Hgtransform Properties
hidden
hilb
hist
histc
hold
home
horzcat
horzcat (tscollection)
hostid
hsv2rgb
hypot
i
idealfilter (timeseries)
idivide
if
ifft
ifft2
ifftn
ifftshift
ilu
im2frame
im2java
imag
image
Image Properties
imagesc
imfinfo
imformats
import
importdata
imread
imwrite
ind2rgb
ind2sub

Inf
inferiorto
info
inline
inmem
inpolygon
input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8, int16, int32, int64
interfaces
interp1
interp1q
interp2
interp3
interpft
interpn
interpstreamspeed
intersect
intmax
intmin
intwarning
inv
invhilb
invoke
ipermute
iqr (timeseries)
is*
isa
isappdata
iscell

iscellstr
ischar
iscom
isdir
isempty
isempty (timeseries)
isempty (tscollection)
isequal
isequal (MException)
isequalwithequalnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishold
isinf
isinteger
isinterface
isjava
iskeyword
isletter
islogical
ismac
ismember
ismethod
isnan
isnumeric
isobject
isocaps
isocolors
isonormals
isosurface
ispc
ispref
isprime

isprop
isreal
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
isunix
isvalid (serial)
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
javaMethod
javaObject
javarmpath
keyboard
kron
last (MException)
lasterr
lasterror
lastwarn
lcm
ldl
ldivide, rdivide
le
legend
legendre
length
length (serial)

length (timeseries)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec
linkaxes
linkprop
linsolve
linspace
listdlg
listfonts
load
load (COM)
load (serial)
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor

lower
ls
lscov
lsqnonneg
lsqr
lt
lu
luinc
magic
makehgtform
mat2cell
mat2str
material
matlabcolon (matlab:)
matlabrc
matlabroot
matlab (UNIX)
matlab (Windows)
max
max (timeseries)
MaximizeCommandWindow
maxNumCompThreads
mean
mean (timeseries)
median
median (timeseries)
memmapfile
memory
MException
menu
mesh, meshc, meshz
meshgrid
methods
methodsview
mex
mexext
mfilename

mget
min
min (timeseries)
MinimizeCommandWindow
minres
mislocked
mkdir
mkdir (ftp)
mkpp
mldivide \, mrdivide /
mlint
mlintrpt
mlock
mmfileinfo
mmreader
mod
mode
more
move
movefile
movegui
movie
movie2avi
mput
msgbox
mtimes
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin, nargout
nargoutchk
native2unicode
nchoosek

ndgrid
ndims
ne
ne (MException)
newplot
nextpow2
nnz
noanimate
nonzeros
norm
normest
not
notebook
now
nthroot
null
num2cell
num2hex
num2str
numel
nzmax
ode15i
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
odefile
odeget
odeset
odextend
ones
open
openfig
opengl
openvar
optimget
optimset
or
ordeig
orderfields

ordqz
ordschur
orient
orth
otherwise
pack
padecoef
pagesetupdlg
pan
pareto
parse (inputParser)
parseSoapResponse
partialpath
pascal
patch
Patch Properties
path
path2rc
pathdef
pathsep
pathtool
pause
pbaspect
pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie
pie3

pinv
planerot
playshow
plot
plot (timeseries)
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval
polyvalm
pow2
power
ppval
prefdir
preferences
primes
print, printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
propertyeditor
psi

publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quadgk
quadl
quadv
questdlg
quit
Quit (COM)
quiver
quiver3
Quivergroup Properties
qz
rand
randn
randperm
rank
rat, rats
rbbox
rcond
read
readasync
real
reallog
realmax
realmin
realpow
realsqrt
record

rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp, regexpi
regexprep
regexptranslate
registerevent
rehash
release
rem
removets
rename
repmat
resample (timeseries)
resample (tscollection)
reset
reshape
residue
restoredefaultpath
rethrow
rethrow (MException)
return
rgb2hsv
rgbplot
ribbon
rmappdata
rmdir
rmdir (ftp)
rmfield
rmpath
rmpref
root object

Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rref
rsf2csf
run
save
save (COM)
save (serial)
saveas
saveobj
savepath
scatter
scatter3
Scattergroup Properties
schur
script
sec
secd
sech
selectmoveresize
semilogx, semilogy
sendmail
serial
serialbreak
set
set (COM)
set (serial)
set (timer)
set (timeseries)
set (tscollection)
setabstime (timeseries)

setabstime (tscollection)
setappdata
setdiff
setenv
setfield
setinterpmethod
setpixelposition
setpref
setstr
settimeseriesnames
setxor
shading
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size (serial)
size (timeseries)
size (tscollection)
slice
smooth3
sort
sortrows
sound
soundsc
spalloc
sparse
spaugment
spconvert
spdiags
specular
speye

spfun
sph2cart
sphere
spinmap
spline
spones
spparms
sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stairs
Stairseries Properties
start
startat
startup
std
std (timeseries)
stem
stem3
Stemseries Properties
stop
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp, strcmpi
stream2

stream3
streamline
streamparticles
streamribbon
streamslice
streamtube
strfind
strings
strjust
strmatch
strncmp, strncmpi
strread
strrep
strtok
strtrim
struct
struct2cell
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substruct
subvolume
sum
sum (timeseries)
superiorto
support
surf, surfc
surf2patch
surface
Surface Properties
Surfaceplot Properties
surfl

surfnorm
svd
svds
swapbytes
switch
symamd
symbfact
symmlq
symmmd
symrcm
symvar
synchronize
syntax
system
tan
tand
tanh
tar
tempdir
tempname
tetramesh
texlabel
text
Text Properties
textread
textscan
textwrap
throw (MException)
throwAsCaller (MException)
tic, toc
timer
timerfind
timerfindall
timeseries
title
todatenum
toeplitz

toolboxdir
trace
transpose (timeseries)
trapz
treelayout
treeplot
tril
trimesh
triplequad
triplot
trisurf
triu
true
try
tscollection
tsdata.event
tsearch
tsearchn
tsprops
tstool
type
typecast
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties
uigetdir
uigetfile
uigetpref
uiimport
uimenu
Uimenu Properties
uint8, uint16, uint32, uint64
uiopen
uipanel

vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (timeseries)
vertcat (tscollection)
view
viewmtx
volumebounds
voronoi
voronoin
wait
waitbar
waitfor
waitforbuttonpress
warndlg
warning
waterfall
wavfinfo
wavplay
wavread
wavrecord
wavwrite
web
weekday
what
whatsnew
which
while
whitebg
who, whos
wilkinson
winopen
winqueryreg
wk1finfo

wk1read
wk1write
workspace
xlabel, ylabel, zlabel
xlim, ylim, zlim
xlsfinfo
xlsread
xlswrite
xmlread
xmlwrite
xor
xslt
zeros
zip
zoom

factor

Purpose

Prime factors

Syntax

```
f = factor(n)
```

Description

`f = factor(n)` returns a row vector containing the prime factors of `n`.

Examples

```
f = factor(123)
f =
     3    41
```

See Also

`isprime`, `primes`

Purpose

Factorial function

Syntax

```
factorial(N)
```

Description

`factorial(N)`, for scalar `N`, is the product of all the integers from `1` to `N`, i.e. `prod(1:n)`. When `N` is an `N`-dimensional array, `factorial(N)` is the factorial for each element of `N`.

Since double precision numbers only have about 15 digits, the answer is only accurate for `n <= 21`. For larger `n`, the answer will have the right magnitude, and is accurate for the first 15 digits.

See Also

`prod`

false

Purpose

Logical 0 (false)

Syntax

```
false
false(n)
false(m, n)
false(m, n, p, ...)
false(size(A))
```

Description

`false` is shorthand for `logical(0)`.

`false(n)` is an `n`-by-`n` matrix of logical zeros.

`false(m, n)` or `false([m, n])` is an `m`-by-`n` matrix of logical zeros.

`false(m, n, p, ...)` or `false([m n p ...])` is an `m`-by-`n`-by-`p`-by-`...` array of logical zeros.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`false(size(A))` is an array of logical zeros that is the same size as array `A`.

Remarks

`false(n)` is much faster and more memory efficient than `logical(zeros(n))`.

See Also

`true`, `logical`

Purpose

Close one or more open files

Syntax

```
status = fclose(fid)
status = fclose('all')
```

Description

`status = fclose(fid)` closes the specified file if it is open, returning `0` if successful and `-1` if unsuccessful. Argument `fid` is a file identifier associated with an open file. (See `fopen` for a complete description of `fid`).

If `fid` does not represent an open file, or if it is equal to 0, 1, or 2, then `fclose` throws an error.

`status = fclose('all')` closes all open files (except standard input, output, and error), returning `0` if successful and `-1` if unsuccessful.

See Also

`ferror`, `fopen`, `fprintf`, `fread`, `frewind`, `fscanf`, `fseek`, `ftell`, `fwrite`

fclose (serial)

Purpose

Disconnect serial port object from device

Syntax

```
fclose(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`fclose(obj)` disconnects `obj` from the device.

Remarks

If `obj` was successfully disconnected, then the `Status` property is configured to `closed` and the `RecordStatus` property is configured to `off`. You can reconnect `obj` to the device using the `fopen` function.

An error is returned if you issue `fclose` while data is being written asynchronously. In this case, you should abort the write operation with the `stopasync` function, or wait for the write operation to complete.

If you use the `help` command to display help for `fclose`, then you need to supply the pathname shown below.

```
help serial/fclose
```

Example

This example creates the serial port object `s`, connects `s` to the device, writes and reads text data, and then disconnects `s` from the device using `fclose`.

```
s = serial('COM1');
fopen(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fclose(s)
```

At this point, the device is available to be connected to a serial port object. If you no longer need `s`, you should remove from memory with the `delete` function, and remove it from the workspace with the `clear` command.

See Also

Functions

clear, delete, fopen, stopasync

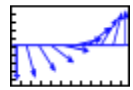
Properties

RecordStatus, Status

feather

Purpose

Plot velocity vectors



GUI Alternatives

Use the Plot Selector  to graph selected variables in the Workspace Browser and the Plot Catalog, accessed from the Figure Palette. Directly manipulate graphs in *plot edit* mode, and modify them using the Property Editor. For details, see "Working in Plot Edit Mode", and "The Figure Palette" in the MATLAB Graphics documentation, and also Creating Graphics from the Workspace Browser in the MATLAB Desktop documentation.

Syntax

```
feather(U,V)
feather(Z)
feather(...,LineSpec)
feather(axes_handle,...)
h = feather(...)
```

Description

A feather plot displays vectors emanating from equally spaced points along a horizontal axis. You express the vector components relative to the origin of the respective vector.

`feather(U,V)` displays the vectors specified by `U` and `V`, where `U` contains the x components as relative coordinates, and `V` contains the y components as relative coordinates.

`feather(Z)` displays the vectors specified by the complex numbers in `Z`. This is equivalent to `feather(real(Z),imag(Z))`.

`feather(...,LineSpec)` draws a feather plot using the line type, marker symbol, and color specified by `LineSpec`.

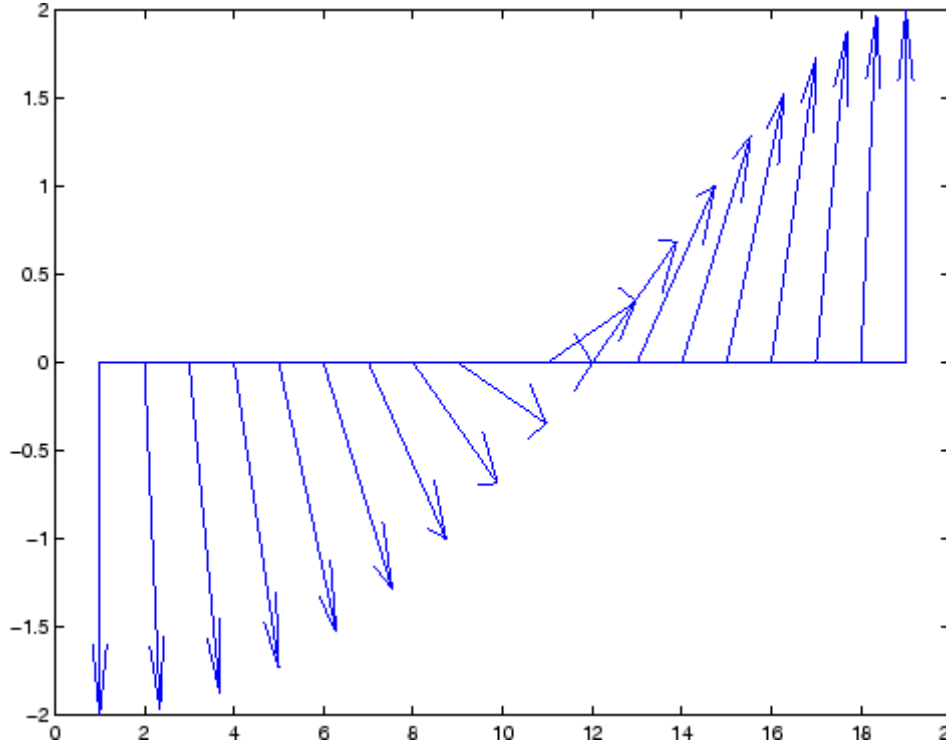
`feather(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = feather(...)` returns the handles to line objects in `h`.

Examples

Create a feather plot showing the direction of `theta`.

```
theta = (-90:10:90)*pi/180;
r = 2*ones(size(theta));
[u,v] = pol2cart(theta,r);
feather(u,v);
```



See Also

`compass`, `LineSpec`, `rose`

"Direction and Velocity Plots" on page 1-89 for related functions

feof

Purpose

Test for end-of-file

Syntax

```
eofstat = feof(fid)
```

Description

`eofstat = feof(fid)` returns `1` if the end-of-file indicator for the file `fid` has been set and `0` otherwise. (See `fopen` for a complete description of `fid`.)

The end-of-file indicator is set when there is no more input from the file.

See Also

`fopen`

Purpose

Query MATLAB about errors in file input or output

Syntax

```
message = ferror(fid)
message = ferror(fid, 'clear')
[message,errnum] = ferror(...)
```

Description

`message = ferror(fid)` returns the error string `message`. Argument `fid` is a file identifier associated with an open file (see `fopen` for a complete description of `fid`).

`message = ferror(fid, 'clear')` clears the error indicator for the specified file.

`[message,errnum] = ferror(...)` returns the error status number `errnum` of the most recent file I/O operation associated with the specified file.

If the most recent I/O operation performed on the specified file was successful, the value of `message` is empty and `ferror` returns an `errnum` value of `0`.

A nonzero `errnum` indicates that an error occurred in the most recent file I/O operation. The value of `message` is a string that can contain information about the nature of the error. If the message is not helpful, consult the C run-time library manual for your host operating system for further details.

See Also

`fclose`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`

feval

Purpose

Evaluate function

Syntax

```
[y1, y2, ...] = feval(fhandle, x1, ..., xn)
[y1, y2, ...] = feval(function, x1, ..., xn)
```

Description

`[y1, y2, ...] = feval(fhandle, x1, ..., xn)` evaluates the function handle, `fhandle`, using arguments `x1` through `xn`. If the function handle is bound to more than one built-in or M-file, (that is, it represents a set of overloaded functions), then the data type of the arguments `x1` through `xn` determines which function is dispatched to.

Note It is not necessary to use `feval` to call a function by means of a function handle. This is explained in "Calling a Function Using Its Handle" in the MATLAB Programming documentation.

`[y1, y2, ...] = feval(function, x1, ..., xn)`. If `function` is a quoted string containing the name of a function (usually defined by an M-file), then `feval(function, x1, ..., xn)` evaluates that function at the given arguments. The `function` parameter must be a simple function name; it cannot contain path information.

Remarks

The following two statements are equivalent.

```
[V,D] = eig(A)
[V,D] = feval(@eig, A)
```

Examples

The following example passes a function handle, `fhandle`, in a call to `fminbnd`. The `fhandle` argument is a handle to the `humps` function.

```
fhandle = @humps;
x = fminbnd(fhandle, 0.3, 1);
```

The `fminbnd` function uses `feval` to evaluate the function handle that was passed in.

```
function [xf, fval, exitflag, output] = ...
    fminbnd(funfcn, ax, bx, options, varargin)
            .
            .
            .
fx = feval(funfcn, x, varargin{:});
```

See Also

assignin, function_handle, functions, builtin, eval, evalin

Purpose

Evaluate MATLAB function in server

Syntax

MATLAB Client

```
result = h.Feval('functionname', numout, arg1, arg2, ...)
result = Feval(h, 'functionname', numout, arg1, arg2, ...)
result = invoke(h, 'Feval', 'functionname', numout, ...
arg1, arg2, ...)
```

Method Signatures

```
HRESULT Feval([in] BSTR functionname, [in] long nargout,
[out] VARIANT* result, [in, optional] VARIANT arg1, arg2, ...)
```

Visual Basic Client

```
Feval(String functionname, long numout,
arg1, arg2, ...) As Object
```

Description

`Feval` executes the MATLAB function specified by the string `functionname` in the Automation server attached to handle `h`.

Indicate the number of outputs to be returned by the function in a 1-by-1 `double` array, `numout`. The server returns output from the function in the cell array, `result`.

You can specify as many as 32 input arguments to be passed to the function. These arguments follow `numout` in the `Feval` argument list. There are four ways to pass an argument to the function being evaluated.

Passing Mechanism	Description
Pass the value itself	To pass any numeric or string value, specify the value in the `Feval` argument list: `a = h.Feval('sin', 1, -pi:0.01:pi);`

Passing Mechanism	Description

Pass a client variable	To pass an argument that is assigned to a variable in the client, specify the variable name alone:
	
	`x = -pi:0.01:pi;`
	`a = h.Feval('sin', 1, x);`

Reference a server variable	To reference a variable that is defined in the server, specify the variable name followed by an equals (=) sign:
	
	`h.PutWorkspaceData('x', 'base', -pi:0.01:pi);`
	`a = h.Feval('sin', 1, 'x=');`
	
	Note that the server variable is not reassigned.

Remarks

If you want output from `Feval` to be displayed at the client window, you must assign a returned value.

Server function names, like `Feval`, are case sensitive when using the first two syntaxes shown in the Syntax section.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Examples

Passing Arguments — MATLAB Client

This section contains a number of examples showing how to use `Feval` to execute MATLAB commands on a MATLAB Automation server.

- Concatenate two strings in the server by passing the input strings in a call to `strcat` through `Feval` (`strcat` deletes trailing spaces; use leading spaces):

```
h = actxserver('matlab.application');
a = h.Feval('strcat', 1, 'hello', ' world')
a =
    'hello world'
```

- Perform the same concatenation, passing a string and a local variable `clistr` that contains the second string:

```
clistr = ' world';
a = h.Feval('strcat', 1, 'hello', clistr)
a =
    'hello world'
```

- This next example is different in that the variable `srvstr` is defined in the server, not the client. Putting an equals sign after a variable name (e.g., `srvstr=`) indicates that it a server variable, and that MATLAB should not expect the variable to be defined on the client:

```
% Define the variable srvstr on the server.
h.PutCharArray('srvstr', 'base', ' world')

% Pass the name of the server variable using 'name=' syntax
a = h.Feval('strcat', 1, 'hello', 'srvstr=')
a =
    'hello world'
```

Visual Basic .NET Client

Here are the same examples shown above, but written for a Visual Basic .NET client. These examples return the same strings as shown above.

- Pass the two strings to the MATLAB function `strcat` on the server:

```
Dim Matlab As Object
Dim out As Object
Matlab = CreateObject("matlab.application")
out = Matlab.Feval("strcat", 1, "hello", " world")
```

- Define `clistr` locally and pass this variable:

```
Dim clistr As String
clistr = " world"
out = Matlab.Feval("strcat", 1, "hello", clistr)
```

- Pass the name of a variable defined on the server:

```
Matlab.PutCharArray("srvstr", "base", " world")
out = Matlab.Feval("strcat", 1, "hello", "srvstr=")
```

Feval Return Values — MATLAB Client. `Feval` returns data from the evaluated function in a cell array. The cell array has one row for every return value. You can control how many values are returned using the second input argument to `Feval`, as shown in this example.

The second argument in the following example specifies that `Feval` return three outputs from the `fileparts` function. As is the case here, you can request fewer than the maximum number of return values for a function (`fileparts` can return up to four):

```
a = h.Feval('fileparts', 3, 'd:\work\ConsoleApp.cpp')
a =
    'd:\work'
    'ConsoleApp'
    '.cpp'
```

Convert the returned values from the cell array `a` to char arrays:

```
a{:}
ans =
d:\work

ans =
ConsoleApp

ans =
.cpp
```

Feval Return Values — Visual Basic .NET Client

Here is the same example, but coded in Visual Basic. Define the argument returned by `Feval` as an `Object`.

```
Dim Matlab As Object
Dim out As Object
```

```
Matlab = CreateObject("matlab.application")
out = Matlab.Feval("fileparts", 3, "d:\work\ConsoleApp.cpp")
```

See Also

Execute, PutFullMatrix, GetFullMatrix, PutCharArray, GetCharArray

Purpose

Discrete Fourier transform

Syntax

```
Y = fft(X)
Y = fft(X,n)
Y = fft(X,[],dim)
Y = fft(X,n,dim)
```

Definition

The functions `X = fft(x)` and `x = ifft(X)` implement the transform and inverse transform pair given for vectors of length N by:

$$X(k) = \sum_{j=1}^{N} x(j)\omega_N^{(j-1)(k-1)}$$

$$x(j) = (1/N)\sum_{k=1}^{N} X(k)\omega_N^{-(j-1)(k-1)}$$

where

$$\omega_N = e^{(-2\pi i)/N}$$

is an Nth root of unity.

Description

`Y = fft(X)` returns the discrete Fourier transform (DFT) of vector `X`, computed with a fast Fourier transform (FFT) algorithm.

If `X` is a matrix, `fft` returns the Fourier transform of each column of the matrix.

If `X` is a multidimensional array, `fft` operates on the first nonsingleton dimension.

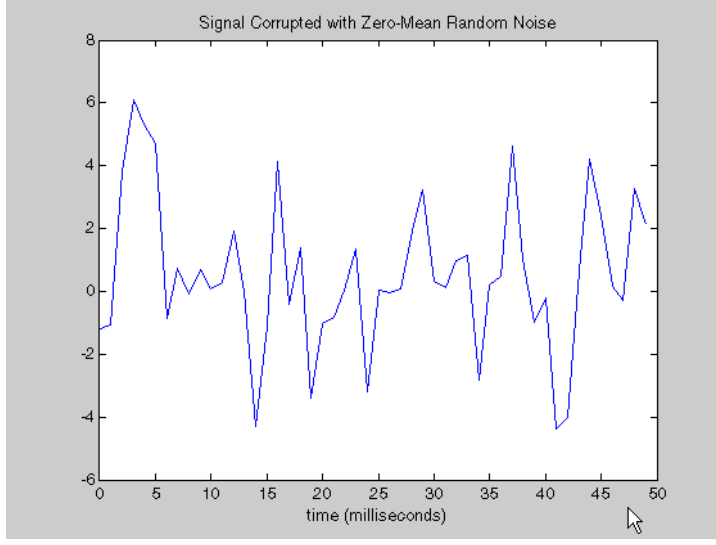
`Y = fft(X,n)` returns the `n`-point DFT. If the length of `X` is less than `n`, `X` is padded with trailing zeros to length `n`. If the length of `X` is greater than `n`, the sequence `X` is truncated. When `X` is a matrix, the length of the columns are adjusted in the same manner.

`Y = fft(X,[],dim)` and `Y = fft(X,n,dim)` applies the FFT operation across the dimension `dim`.

Examples

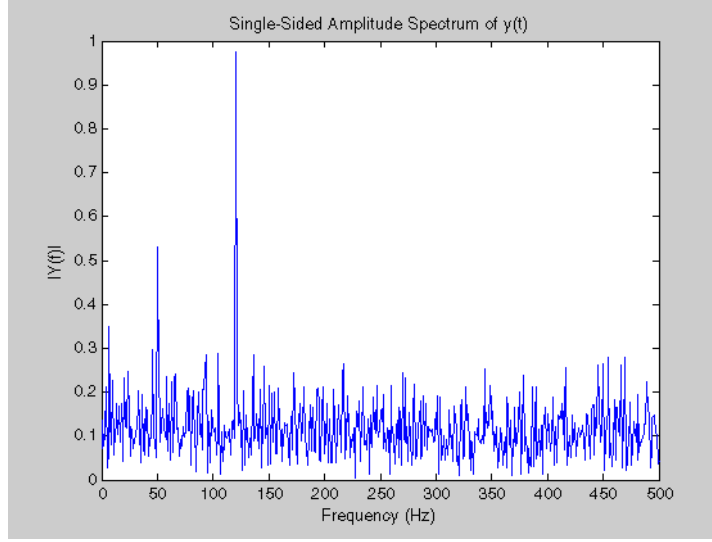
A common use of Fourier transforms is to find the frequency components of a signal buried in a noisy time domain signal. Consider data sampled at 1000 Hz. Form a signal containing a 50 Hz sinusoid of amplitude 0.7 and 120 Hz sinusoid of amplitude 1 and corrupt it with some zero-mean random noise:

```
Fs = 1000;                    % Sampling frequency
T = 1/Fs;                     % Sample time
L = 1000;                     % Length of signal
t = (0:L-1)*T;                % Time vector
% Sum of a 50 Hz sinusoid and a 120 Hz sinusoid
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(size(t));     % Sinusoids plus noise
plot(Fs*t(1:50),y(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (milliseconds)')
```



It is difficult to identify the frequency components by looking at the original signal. Converting to the frequency domain, the discrete Fourier transform of the noisy signal y is found by taking the fast Fourier transform (FFT):

```
NFFT = 2^nextpow2(L); % Next power of 2 from length of y
Y = fft(y,NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2);

% Plot single-sided amplitude spectrum.
plot(f,2*abs(Y(1:NFFT/2)))
title('Single-Sided Amplitude Spectrum of y(t)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
```



The main reason the amplitudes are not exactly at 0.7 and 1 is because of the noise. Several executions of this code (including recomputation of y) will produce different approximations to 0.7 and 1. The other reason is that you have a finite length signal. Increasing L from 1000 to

10000 in the example above will produce much better approximations on average.

Algorithm

The FFT functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`) are based on a library called FFTW [3],[4]. To compute an N-point DFT when N is composite (that is, when $N = N_1 N_2$), the FFTW library decomposes the problem using the Cooley-Tukey algorithm [1], which first computes N_1 transforms of size N_2, and then computes N_2 transforms of size N_1. The decomposition is applied recursively to both the N_1- and N_2-point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey [5], a prime factor algorithm [6], and a split-radix algorithm [2]. The particular factorization of N is chosen heuristically.

When N is a prime number, the FFTW library first decomposes an N-point problem into three $(N-1)$-point problems using Rader's algorithm [7]. It then uses the Cooley-Tukey decomposition described above to compute the $(N-1)$-point DFTs.

For most N, real-input DFTs require roughly half the computation time of complex-input DFTs. However, when N has large prime factors, there is little or no speed difference.

The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fft` using the utility function `fftw`, which controls how MATLAB optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`fft` supports inputs of data types `double` and `single`. If you call `fft` with the syntax `y = fft(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fft2`, `fftn`, `fftw`, `fftshift`, `ifft`

`dftmtx`, `filter`, and `freqz` in the Signal Processing Toolbox

References

[1] Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Computation of the Complex Fourier Series,"*Mathematics of Computation*, Vol. 19, April 1965, pp. 297-301.

[2] Duhamel, P. and M. Vetterli, "Fast Fourier Transforms: A Tutorial Review and a State of the Art," *Signal Processing*, Vol. 19, April 1990, pp. 259-299.

[3] FFTW (`http://www.fftw.org`)

[4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

[5] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 611.

[6] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 619.

[7] Rader, C. M., "Discrete Fourier Transforms when the Number of Data Samples Is Prime," *Proceedings of the IEEE*, Vol. 56, June 1968, pp. 1107-1108.

Purpose

2-D discrete Fourier transform

Syntax

```
Y = fft2(X)
Y = fft2(X,m,n)
```

Description

`Y = fft2(X)` returns the two-dimensional discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`Y = fft2(X,m,n)` truncates `X`, or pads `X` with zeros to create an `m`-by-`n` array before doing the transform. The result is `m`-by-`n`.

Algorithm

`fft2(X)` can be simply computed as

```
fft(fft(X).').'
```

This computes the one-dimensional DFT of each column `X`, then of each row of the result. The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fft2` using the utility function `fftw`, which controls how MATLAB optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`fft2` supports inputs of data types `double` and `single`. If you call `fft2` with the syntax `y = fft2(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fft`, `fftn`, `fftw`, `fftshift`, `ifft2`

Purpose

N-D discrete Fourier transform

Syntax

```
Y = fftn(X)
Y = fftn(X,siz)
```

Description

`Y = fftn(X)` returns the discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`Y = fftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the transform. The size of the result `Y` is `siz`.

Algorithm

`fftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = fft(Y,[],p);
end
```

This computes in-place the one-dimensional fast Fourier transform along each dimension of `X`. The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fftn` using the utility function `fftw`, which controls how MATLAB optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`fftn` supports inputs of data types `double` and `single`. If you call `fftn` with the syntax `y = fftn(X, ...)`, the output `y` has the same data type as the input `X`.

See Also fft, fft2, fftn, fftw, ifftn

Purpose

Shift zero-frequency component to center of spectrum

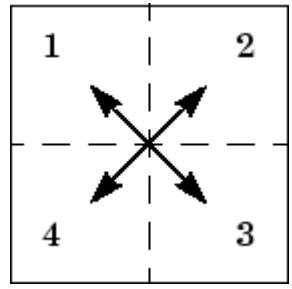
Syntax

```
Y = fftshift(X)
Y = fftshift(X,dim)
```

Description

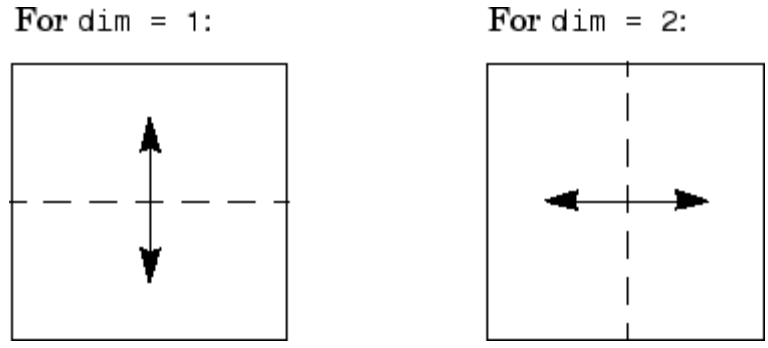
`Y = fftshift(X)` rearranges the outputs of `fft`, `fft2`, and `fftn` by moving the zero-frequency component to the center of the array. It is useful for visualizing a Fourier transform with the zero-frequency component in the middle of the spectrum.

For vectors, `fftshift(X)` swaps the left and right halves of `X`. For matrices, `fftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth.



For higher-dimensional arrays, `fftshift(X)` swaps “half-spaces” of `X` along each dimension.

`Y = fftshift(X,dim)` applies the `fftshift` operation along the dimension `dim`.



Note `ifftshift` will undo the results of `fftshift`. If the matrix `X` contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original `X`. Simply performing `fftshift(X)` twice will not produce `X`.

Examples

For any matrix `X`

```
Y = fft2(X)
```

has `Y(1,1) = sum(sum(X))`; the zero-frequency component of the signal is in the upper-left corner of the two-dimensional FFT. For

```
Z = fftshift(Y)
```

this zero-frequency component is near the center of the matrix.

See Also

`circshift`, `fft`, `fft2`, `fftn`, `ifftshift`

Purpose

Interface to FFTW library run-time algorithm tuning control

Syntax

```
fftw('planner', method)
method = fftw('planner')
str = fftw('dwisdom')
str = fftw('swisdom')
fftw('dwisdom', str)
fftw('swisdom', str)
```

Description

`fftw` enables you to optimize the speed of the MATLAB FFT functions `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, and `ifftn`. You can use `fftw` to set options for a tuning algorithm that experimentally determines the fastest algorithm for computing an FFT of a particular size and dimension at run time. MATLAB records the optimal algorithm in an internal data base and uses it to compute FFTs of the same size throughout the current session. The tuning algorithm is part of the FFTW library that MATLAB uses to compute FFTs.

`fftw('planner', method)` sets the method by which the tuning algorithm searches for a good FFT algorithm when the dimension of the FFT is not a power of 2. You can specify `method` to be one of the following. The default method is `estimate`:

- `'estimate'`
- `'measure'`
- `'patient'`
- `'exhaustive'`
- `'hybrid'`

When you call `fftw('planner', method)`, the next time you call one of the FFT functions, such as `fft`, the tuning algorithm uses the specified method to optimize the FFT computation. Because the tuning involves trying different algorithms, the first time you call an FFT function, it might run more slowly than if you did not call `fftw`. However,

subsequent calls to any of the FFT functions, for a problem of the same size, often run more quickly than they would without using `fftw`.

Note The FFT functions only use the optimal FFT algorithm during the current MATLAB session. “Reusing Optimal FFT Algorithms” on page 2-1118 explains how to reuse the optimal algorithm in a future MATLAB session.

If you set the method to `'estimate'`, the FFTW library does not use run-time tuning to select the algorithms. The resulting algorithms might not be optimal.

If you set the method to `'measure'`, the FFTW library experiments with many different algorithms to compute an FFT of a given size and chooses the fastest. Setting the method to `'patient'` or `'exhaustive'` has a similar result, but the library experiments with even more algorithms so that the tuning takes longer the first time you call an FFT function. However, subsequent calls to FFT functions are faster than with `'measure'`.

If you set `'planner'` to `'hybrid'`, MATLAB

- Sets method to `'measure'` method for FFT dimensions 8192 or smaller.
- Sets method to `'estimate'` for FFT dimensions greater than 8192.

`method = fftw('planner')` returns the current planner method.

`str = fftw('dwisdom')` returns the information in the FFTW library’s internal double-precision database as a string. The string can be saved and then later reused in a subsequent MATLAB session using the next syntax.

`str = fftw('swisdom')` returns the information in the FFTW library’s internal single-precision database as a string.

`fftw('dwisdom', str)` loads `fftw` wisdom represented by the string `str` into the FFTW library's internal double-precision wisdom database. `fftw('dwisdom','')` or `fftw('dwisdom',[])` clears the internal wisdom database.

`fftw('swisdom', str)` loads `fftw` wisdom represented by the string `str` into the FFTW library's internal single-precision wisdom database. `fftw('swisdom','')` or `fftw('swisdom',[])` clears the internal wisdom database.

Note on large powers of 2 For FFT dimensions that are powers of 2, between 2^{14} and 2^{22}, MATLAB uses special preloaded information in its internal database to optimize the FFT computation. No tuning is performed when the dimension of the FTT is a power of 2, unless you clear the database using the command `fftw('wisdom', [])`.

For more information about the FFTW library, see `http://www.fftw.org`.

Example

Comparison of Speed for Different Planner Methods

The following example illustrates the run times for different settings of `planner`. The example first creates some data and applies `fft` to it using the default method, `estimate`.

```
t=0:.001:5;
x = sin(2*pi*50*t)+sin(2*pi*120*t);
y = x + 2*randn(size(t));

tic; Y = fft(y,1458); toc
Elapsed time is 0.000400 seconds.
```

If you execute the commands

```
tic; Y = fft(y,1458); toc
```

a second time, MATLAB reports the elapsed time as essentially 0. To measure the elapsed time more accurately, you can execute the command `Y = fft(y,1458)` 1000 times in a loop.

```
tic; for k=1:1000
Y = fft(y,1458);
end; toc
Elapsed time is 0.098355 seconds.
```

This tells you that it takes approximately 1/1000 of a second to execute `fft(y, 1458)` a single time.

For comparison, set `planner` to `patient`. Since this `planner` explores possible algorithms more thoroughly than `hybrid`, the first time you run `fft`, it takes longer to compute the results.

```
fftw('planner','patient')
tic;Y = fft(y,1458);toc
Elapsed time is 0.000387 seconds.
```

However, the next time you call `fft`, it runs approximately 10 times faster than before you ran the method `patient`.

```
tic;for k=1:1000
Y=fft(y,1458);
end;toc
Elapsed time is 0.097793 seconds.
```

Reusing Optimal FFT Algorithms

In order to use the optimized FFT algorithm in a future MATLAB session, first save the “wisdom” using the command

```
str = fftw('wisdom')
```

You can save `str` for a future session using the command

```
save str
```

The next time you open MATLAB, load `str` using the command

```
load str
```

and then reload the “wisdom” into the FFTW database using the command

```
fftw('wisdom', str)
```

See Also

`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`, `fftshift`.

fgetl

Purpose

Read line from file, discarding newline character

Syntax

```
tline = fgetl(fid)
```

Description

`tline = fgetl(fid)` returns the next line of the file associated with the file identifier `fid`. If `fgetl` encounters the end-of-file indicator, it returns `-1`. (See `fopen` for a complete description of `fid`.) `fgetl` is intended for use with files that contain newline characters.

MATLAB reads characters using the encoding scheme associated with the file. See `fopen` for more information.

The returned string `tline` does not include the line terminator(s) with the text line. To obtain the line terminators, use `fgets`.

Examples

The example reads every line of the M-file `fgetl.m`.

```
fid=fopen('fgetl.m');
while 1
    tline = fgetl(fid);
    if ~ischar(tline),   break,   end
    disp(tline)
end
fclose(fid);
```

See Also

`fgets`

Purpose

Read line of text from device and discard terminator

Syntax

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
```

Arguments

`obj`	A serial port object.
`tline`	Text read from the instrument, excluding the terminator.
`count`	The number of values read, including the terminator.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`tline = fgetl(obj)` reads one line of text from the device connected to `obj`, and returns the data to `tline`. The returned data does not include the terminator with the text line. To include the terminator, use `fgets`.

`[tline,count] = fgetl(obj)` returns the number of values read to `count`.

`[tline,count,msg] = fgetl(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Remarks

Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgetl` is issued.

If you use the `help` command to display help for `fgetl`, then you need to supply the pathname shown below.

```
help serial/fgetl
```

Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Example

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    17
```

Use `fgetl` to read the data returned from the previous write operation, and discard the terminator.

```
settings = fgetl(s)
settings =
9600;0;0;NONE;LF
length(settings)
ans =
    16
```

Disconnect s from the scope, and remove s from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

fgets, fopen

Properties

BytesAvailable, InputBufferSize, ReadAsyncMode, Status, Terminator, Timeout, ValuesReceived

fgets

Purpose

Read line from file, keeping newline character

Syntax

```
tline = fgets(fid)
tline = fgets(fid, nchar)
```

Description

`tline = fgets(fid)` returns the next line of the file associated with file identifier `fid`. If `fgets` encounters the end-of-file indicator, it returns `-1`. (See `fopen` for a complete description of `fid`.) `fgets` is intended for use with files that contain newline characters.

MATLAB reads characters using the encoding scheme associated with the file. See `fopen` for more information.

The returned string `tline` includes the line terminators associated with the text line. To obtain the string without the line terminators, use `fgetl`.

`tline = fgets(fid, nchar)` returns at most `nchar` characters of the next line. No additional characters are read after the line terminators or an end-of-file.

See Also

`fgetl`

Purpose

Read line of text from device and include terminator

Syntax

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
```

Arguments

`obj`	A serial port object.
`tline`	Text read from the instrument, including the terminator.
`count`	The number of bytes read, including the terminator.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`tline = fgets(obj)` reads one line of text from the device connected to `obj`, and returns the data to `tline`. The returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Remarks

Before you can read text from the device, it must be connected to `obj` with the `fopen`function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgets` is issued.

If you use the `help` command to display help for `fgets`, then you need to supply the pathname shown below.

```
help serial/fgets
```

Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Example

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    17
```

Use `fgets` to read the data returned from the previous write operation, and include the terminator.

```
settings = fgets(s)
settings =
9600;0;0;NONE;LF
length(settings)
ans =
    17
```

Disconnect s from the scope, and remove s from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

fgetl, fopen

Properties

BytesAvailable, BytesAvailableFcn, InputBufferSize, Status, Terminator, Timeout, ValuesReceived

fieldnames

Purpose

Field names of structure, or public fields of object

Syntax

```
names = fieldnames(s)
names = fieldnames(obj)
names = fieldnames(obj, '-full')
```

Description

`names = fieldnames(s)` returns a cell array of strings containing the structure field names associated with the structure `s`.

`names = fieldnames(obj)` returns a cell array of strings containing the names of the public data fields associated with `obj`, which is a MATLAB, COM, or Java object.

`names = fieldnames(obj, '-full')` returns a cell array of strings containing the name, type, attributes, and inheritance of each field associated with `obj`, which is a MATLAB, COM, or Java object.

Examples

Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1
```

the command `n = fieldnames(mystr)` yields

```
n =
    'name'
    'ID'
```

In another example, if `f` is an object of Java class `java.awt.Frame`, the command `fieldnames(f)` lists the properties of `f`.

```
f = java.awt.Frame;

fieldnames(f)
ans =
    'WIDTH'
```

```
'HEIGHT'
'PROPERTIES'
'SOMEBITS'
'FRAMEBITS'
'ALLBITS'
    .
    .
```

See Also

`setfield`, `getfield`, `isfield`, `orderfields`, `rmfield`, "Using Dynamic Field Names"

figure

Purpose

Create figure graphics object

Syntax

```
figure
figure('PropertyName',propertyvalue,...)
figure(h)
h = figure(...)
```

Description

`figure` creates figure graphics objects. Figure objects are the individual windows on the screen in which MATLAB displays graphical output.

`figure` creates a new figure object using default property values.

`figure('PropertyName',propertyvalue,...)` creates a new figure object using the values of the properties specified. MATLAB uses default values for any properties that you do not explicitly define as arguments.

`figure(h)` does one of two things, depending on whether or not a figure with handle `h` exists. If `h` is the handle to an existing figure, `figure(h)` makes the figure identified by `h` the current figure, makes it visible, and raises it above all other figures on the screen. The current figure is the target for graphics output. If `h` is not the handle to an existing figure, but is an integer, `figure(h)` creates a figure and assigns it the handle `h`. `figure(h)` where `h` is not the handle to a figure, and is not an integer, is an error.

`h = figure(...)` returns the handle to the figure object.

Remarks

To create a figure object, MATLAB creates a new window whose characteristics are controlled by default figure properties (both factory installed and user defined) and properties specified as arguments. See the properties section for a description of these properties.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

Use `set` to modify the properties of an existing figure or `get` to query the current values of figure properties.

The gcf command returns the handle to the current figure and is useful as an argument to the set and get commands.

Figures can be docked in the desktop. The Dockable property determines whether you can dock the figure.

Making a Figure Current

The current figure is the target for graphics output. There are two ways to make a figure h the current figure.

- Make the figure h current, visible, and displayed on top of other figures:

```
figure(h)
```

- Make the figure h current, but do not change its visibility or stacking with respect to other figures:

```
set(0,'CurrentFigure',h)
```

Examples

Specifying Figure Size and Screen Location

To create a figure window that is one quarter the size of your screen and is positioned in the upper left corner, use the root object's ScreenSize property to determine the size. ScreenSize is a four-element vector: [left, bottom, width, height]:

```
scrsz = get(0,'ScreenSize');
figure('Position',[1 scrsz(4)/2 scrsz(3)/2 scrsz(4)/2])
```

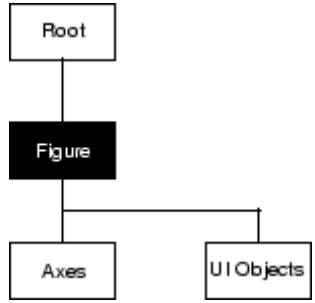
Specifying the Figure Window Title

You can add your own title to a figure by setting the Name property and you can turn off the figure number with the NumberTitle property:

```
figure('Name','Simulation Plot Window','NumberTitle','off')
```

See the Properties section for a description of all figure properties.

Object Hierarchy



Setting Default Properties

You can set default figure properties only on the root level.

```
set(0,'DefaultFigureProperty',PropertyValue...)
```

where *Property* is the name of the figure property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access figure properties.

See Also

`axes`, `uicontrol`, `uimenu`, `close`, `clf`, `gcf`, `rootobject`

"Object Creation Functions" on page 1-94 for related functions

`Figure Properties` descriptions of all figure properties

See "Figure Properties" in the MATLAB Graphics User Guide for more information on figures.

Purpose

Figure properties

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

Figure Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`Alphamap`

m-by-1 matrix of alpha values

Figure alphamap. This property is an m-by-1 array of non-NaN alpha values. MATLAB accesses alpha values by their row number. For example, an index of 1 specifies the first alpha value, an index of 2 specifies the second alpha value, and so on. Alphamaps can be any length. The default alphamap contains 64 values that progress linearly from 0 to 1.

Alphamaps affect the rendering of surface, image, and patch objects, but do not affect other graphics objects.

`BeingDeleted`

`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback function interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback functions. If there is a callback function executing, callback functions invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback function.
- `queue` — Queue the event that attempted to execute a second callback function until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is in the figure window, but not over a child object (i.e., uicontrol, uipanel, axes, or axes child). Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure)

See the figure's `SelectionType` property to determine whether modifier keys were also pressed.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

Using the ButtonDownFcn

This example, creates a figure and defines a function handle callback for the `ButtonDownFcn` property. When the user **Ctrl-**clicks the figure, the callback creates a new figure having the same callback.

Click to view in editor — This link opens the MATLAB editor with the following example.

Click to run example — **Ctrl-**click the figure to create a new figure.

```
fh_cb = @newfig; % Create function handle for newfig function
figure('ButtonDownFcn',fh_cb);

function newfig(src,evnt)
   if strcmp(get(src,'SelectionType'),'alt')
      figure('ButtonDownFcn',fh_cb)
   else
      disp('Use control-click to create a new figure')
   end
end
```

`Children`

vector of handles

Children of the figure. A vector containing the handles of all axes, user-interface objects displayed within the figure. You can change the order of the handles and thereby change the stacking of the objects on the display.

When an object's HandleVisibility property is set to off, it is not listed in its parent's Children property. See HandleVisibility for more information.

Clipping
{on} | off

This property has no effect on figures.

CloseRequestFcn
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Function executed on figure close. This property defines a function that MATLAB executes whenever you issue the close command (either a close(figure_handle) or a close all), when you close a figure window from the computer's window manager menu, or when you quit MATLAB.

The CloseRequestFcn provides a mechanism to intervene in the closing of a figure. It allows you to, for example, display a dialog box to ask a user to confirm or cancel the close operation or to prevent users from closing a figure that contains a GUI.

The basic mechanism is

- A user issues the close command from the command line, by closing the window from the computer's window manager menu, or by quitting MATLAB.
- The close operation executes the function defined by the figure CloseRequestFcn. The default function is named closereq and is predefined as

```
if isempty(gcbf)
   if length(dbstack) == 1
      warning('MATLAB:closereq', ...
          'Calling closereq from the command line is now obsolet
   end
```

```
    close force
else
    delete(gcbf);
end
```

These statements unconditionally delete the current figure, destroying the window. `closereq` takes advantage of the fact that the `close` command makes all figures specified as arguments the current figure before calling the respective close request function.

Note that `closereq` honors the user's `ShowHiddenHandles` setting during figure deletion. This means that hidden figures are not deleted.

Redefining the CloseRequestFcn

Define the `CloseRequestFcn` as a function handle. For example,

```
set(gcf,'CloseRequestFcn',@my_closefcn)
```

Where `@my_closefcn` is a function handle referencing function `my_closefcn`.

Unless the close request function calls `delete` or `close`, MATLAB never closes the figure. (Note that you can always call `delete(`*`figure_handle`*`)` from the command line if you have created a window with a nondestructive close request function.)

A useful application of the close request function is to display a question dialog box asking the user to confirm the close operation. The following function illustrates how to do this.

Click to view in editor — This link opens the MATLAB editor with the following example.

Click to run example — **Ctrl-**click the figure to create a new figure.

```
function my_closereq(src,evnt)
```

```
% User-defined close request function
% to display a question dialog box
   selection = questdlg('Close This Figure?',...
      'Close Request Function',...
      'Yes','No','Yes');
   switch selection,
      case 'Yes',
         delete(gcf)
      case 'No'
      return
   end
end
```

Now create a figure using the yourCloseRequestFcn:

```
figure('CloseRequestFcn',@my_closereq)
```

To make this function your default close request function, set a default value on the root level.

```
set(0,'DefaultFigureCloseRequestFcn',@my_closereq)
```

MATLAB then uses this setting for the `CloseRequestFcn` of all subsequently created figures.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`Color`

`ColorSpec`

Background color. This property controls the figure window background color. You can specify a color using a three-element vector of RGB values or one of the MATLAB predefined names. See `ColorSpec` for more information.

`Colormap`

m-by-3 matrix of RGB values

Figure colormap. This property is an m-by-3 array of red, green, and blue (RGB) intensity values that define m individual colors. MATLAB accesses colors by their row number. For example, an index of 1 specifies the first RGB triplet, an index of 2 specifies the second RGB triplet, and so on.

Number of Colors Allowed

Colormaps can be any length (up to 256 only on MS-Windows), but must be three columns wide. The default figure colormap contains 64 predefined colors.

Objects That Use Colormaps

Colormaps affect the rendering of `surface`, image, and `patch` objects, but generally do not affect other graphics objects. See `colormap` and `ColorSpec` for more information.

`CreateFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during figure creation. This property defines a callback function that executes when MATLAB creates a figure object. You must define this property as a default value on the root level. For example, the statement

```
set(0,'DefaultFigureCreateFcn',@fig_create)
```

defines a default value on the root level that causes all figures created to execute the setup function `fig_create`, which is defined below:

```
function fig_create(src,evnt)
set(src,'Color',[.2 .1 .5],...
   'IntegerHandle','off',...
   'MenuBar','none',...
   'ToolBar','none')
```

```
end
```

MATLAB executes the create function after setting all properties for the figure. Setting this property on an existing figure object has no effect.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`CurrentAxes`
handle of current axes

Target axes in this figure. MATLAB sets this property to the handle of the figure's current axes (i.e., the handle returned by the `gca` command when this figure is the current figure). In all figures for which axes children exist, there is always a current axes. The current axes does not have to be the topmost axes, and setting an axes to be the `CurrentAxes` does not restack it above all other axes.

You can make an axes current using the `axes` and `set` commands. For example, `axes(`*`axes_handle`*`)` and `set(gcf,'CurrentAxes',`*`axes_handle`*`)` both make the axes identified by the handle *`axes_handle`* the current axes. In addition, `axes(`*`axes_handle`*`)` restacks the axes above all other axes in the figure.

If a figure contains no axes, `get(gcf,'CurrentAxes')` returns the empty matrix. Note that the `gca` function actually creates an axes if one does not exist.

`CurrentCharacter`
single character

Last key pressed. MATLAB sets this property to the last key pressed in the figure window. `CurrentCharacter` is useful for obtaining user input.

`CurrentMenu`
(Obsolete)

This property produces a warning message when queried. It has been superseded by the root `CallbackObject` property.

`CurrentObject`
object handle

Handle of current object. MATLAB sets this property to the handle of the last object clicked on by the mouse. This object is the front-most object in the view. You can use this property to determine which object a user has selected. The function `gco` provides a convenient way to retrieve the `CurrentObject` of the `CurrentFigure`.

Note that the `HitTest` property controls whether an object can become the `CurrentObject`.

Hidden Handle Objects

Clicking on an object whose `HandleVisibility` property is set to `off` (such as axis labels and title) causes the `CurrentObject` property to be set to empty `[]`. To avoid returning an empty value when users click on hidden objects, set the hidden object's `HitTest` property to `off`.

Mouse Over

Note that cursor motion over objects does not update the `CurrentObject`; you must click on objects to update this property. See the `CurrentPoint` property for related information.

`CurrentPoint`

two-element vector: [*x*-coordinate, *y*-coordinate]

Location of last button click in this figure. MATLAB sets this property to the location of the pointer at the time of the most recent mouse button press. MATLAB updates this property whenever you press the mouse button while the pointer is in the figure window.

Note that if you select a point in the figure and then use the values returned by the `CurrentPoint` property to plot that point, there can be differences in the position due to round off errors.

CurrentPoint and Cursor Motion

In addition to the behavior described above, MATLAB updates `CurrentPoint` before executing callback routines defined for the figure `WindowButtonMotionFcn` and `WindowButtonUpFcn` properties. This enables you to query `CurrentPoint` from these callback routines. It behaves like this:

- If there is no callback routine defined for the `WindowButtonMotionFcn` or the `WindowButtonUpFcn`, then MATLAB updates the `CurrentPoint` only when the mouse button is pressed down within the figure window.
- If there is a callback routine defined for the `WindowButtonMotionFcn`, then MATLAB updates the `CurrentPoint` just before executing the callback. Note that the `WindowButtonMotionFcn` executes only within the figure window unless the mouse button is pressed down within the window and then held down while the pointer is moved around the screen. In this case, the routine executes (and the `CurrentPoint` is updated) anywhere on the screen until the mouse button is released.
- If there is a callback routine defined for the `WindowButtonUpFcn`, MATLAB updates the `CurrentPoint` just before executing

the callback. Note that the WindowButtonUpFcn executes only while the pointer is within the figure window unless the mouse button is pressed down initially within the window. In this case, releasing the button anywhere on the screen triggers callback execution, which is preceded by an update of the CurrentPoint.

The figure CurrentPoint is updated only when certain events occur, as previously described. In some situations, (such as when the WindowButtonMotionFcn takes a long time to execute and the pointer is moved very rapidly) the CurrentPoint may not reflect the actual location of the pointer, but rather the location at the time when the WindowButtonMotionFcn began execution.

The CurrentPoint is measured from the lower left corner of the figure window, in units determined by the Units property.

The root PointerLocation property contains the location of the pointer updated synchronously with pointer movement. However, the location is measured with respect to the screen, not a figure window.

See uicontrol for information on how this property is set when you click a uicontrol object.

DeleteFcn

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Delete figure callback function. A callback function that executes when the figure object is deleted (e.g., when you issue a delete or a close command). MATLAB executes the function before destroying the object's properties so these values are available to the callback routine.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

The handle of the object whose `DeleteFcn` is being executed is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See also the figure `CloseRequestFcn` property

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DockControls`

{on} | off

Displays controls used to dock figure. This property determines whether the figure enables the **Desktop** menu item and the dock figure button in the titlebar that allow you to dock the figure into the MATLAB desktop.

By default, the figure docking controls are visible. If you set this property to `off`, the **Desktop** menu item that enables you to dock the figure is disabled and the figure dock button is not displayed.

See also the `WindowStyle` property for more information on docking figure.

`DoubleBuffer`

{on} | off

Flash-free rendering for simple animations. Double buffering is the process of drawing to an off-screen pixel buffer and then blitting the buffer contents to the screen once the drawing is complete. Double buffering generally produces flash-free rendering for simple animations (such as those involving lines, as opposed to objects containing large numbers of polygons). Use double buffering with the animated objects' `EraseMode` property set to `normal`. Use the `set` command to disable double buffering.

```
set(figure_handle,'DoubleBuffer','off')
```

Double buffering works only when the figure `Renderer` property is set to `painters`.

`FileName`
String

GUI FIG-filename. GUIDE stores the name of the FIG-file used to save the GUI layout in this property.

`FixedColors`
m-by-3 matrix of RGB values (read only)

Noncolormap colors. Fixed colors define all colors appearing in a figure window that are not obtained from the figure colormap. These colors include axis lines and labels, the colors of `line`, `text`, `uicontrol`, and `uimenu` objects, and any colors that you explicitly define, for example, with a statement like

```
set(gcf,'Color',[0.3,0.7,0.9])
```

Fixed color definitions reside in the system color table and do not appear in the figure colormap. For this reason, fixed colors can limit the number of simultaneously displayed colors if the number of fixed colors plus the number of entries in the figure colormap exceed your system's maximum number of colors.

(See the root `ScreenDepth` property for information on determining the total number of colors supported on your system. See the `MinColorMap` property for information on how MATLAB shares colors between applications.)

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or

deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Callback Visibility

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Visibility Off

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

Visibility and Handles Returned by Other Functions

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure`property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Making All Handles Visible

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Validity of Hidden Handles

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines if the figure can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the figure. If `HitTest` is `off`, clicking the figure sets the `CurrentObject` to the empty matrix.

`IntegerHandle`

`{on} | off`

Figure handle mode. Figure object handles are integers by default. When creating a new figure, MATLAB uses the lowest integer that is not used by an existing figure. If you delete a figure, its integer handle can be reused.

If you set this property to `off`, MATLAB assigns nonreusable real-number handles (e.g., 67.0001221) instead of integers. This feature is designed for dialog boxes where removing the handle from integer values reduces the likelihood of inadvertently drawing into the dialog box.

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a figure callback function can be interrupted by subsequently invoked callbacks.

How callbacks are interrupted

MATLAB checks for queued events that can interrupt a callback function only when it encounters a call to `drawnow`, `figure`, `getframe`, or `pause` in the executing callback function. When one of these functions is executed, MATLAB processes all pending events, including executing all waiting callback functions. The interrupted callback then resumes execution.

What property callbacks are interruptible

Only callback functions defined for the `ButtonDownFcn`, `KeyPressFcn`, `KeyReleaseFcn`, `WindowButtonDownFcn`, `WindowButtonMotionFcn`, `WindowButtonUpFcn` , and `WindowScrollWheelFcn` are affected by the `Interruptible` property.

See the BusyAction property for related information.

`InvertHardcopy`

```
{on} | off
```

Change hardcopy to black objects on white background. This property affects only printed output. Printing a figure having a background color (`Color` property) that is not white results in poor contrast between graphics objects and the figure background and also consumes a lot of printer toner.

When `InvertHardCopy` is `on`, MATLAB eliminates this effect by changing the color of the figure and axes to white and the axis lines, tick marks, axis labels, etc., to black. `lines`, `text`, and the edges of `patches` and `surfaces` might be changed, depending on the `print` command options specified.

If you set `InvertHardCopy` to `off`, the printed output matches the colors displayed on the screen.

See `print` for more information on printing MATLAB figures.

`KeyPressFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Key press callback function. A callback function invoked by a key press that occurs while the figure window has focus. Define the `KeyPressFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure)

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

When there is no callback specified for this property (which is the default state), MATLAB passes any key presses to the command window. However, when you define a callback for this property, the figure retains focus with each key press and executes the specified callback with each key press.

KeyPressFcn Event Structure

When the callback is a function handle, MATLAB passes a structure to the callback function that contains the following fields.

Field	Contents
`Character`	The character displayed as a result of the key(s) pressed.

Field	Contents
`Modifier`	This field is a cell array that contains the names of one or more modifier keys that the user pressed (i.e., **control**, **alt**, **shift**). On Macintosh computers, MATLAB can also return **command**
`Key`	The key pressed (lower case label on key)

Some key combinations do not define a value for the `Character` field.

Using the KeyPressFcn

This example, creates a figure and defines a function handle callback for the `KeyPressFcn` property. When the "e" key is pressed, the callback exports the figure as an EPS file. When **Ctrl-t** is pressed, the callback exports the figure as a TIFF file.

```
function figure_keypress
   figure('KeyPressFcn',@printfig);

   function printfig(src,evnt)
      if evnt.Character == 'e'
         print ('-deps',['-f' num2str(src)])
      elseif length(evnt.Modifier) == 1 & strcmp(evnt.Modifier{:},'control') & evnt.Key == 't'
         print ('-dtiff','-r200',['-f' num2str(src)])
   end
end
```

`KeyReleaseFcn`

functional handle, or cell array containing function handle and additional arguments, string (not recommended)

Key release callback function. A callback function invoked by a key release that occurs while the figure window has focus. Define the `KeyReleaseFcn` as a function handle. The function must define at

least two input arguments (handle of figure associated with key release and an event structure)

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

KeyReleaseFcn Event Structure

When the callback is a function handle, MATLAB passes a structure as the second argument to the callback function that contains the following fields.

Field	Contents
`Character`	The character displayed as a result of the key(s) released.
`Modifier`	This field is a cell array that contains the names of one or more modifier keys that the user releases (i.e., **control**, **alt**, **shift**, or empty if no modifier keys were released). On Macintosh computers, MATLAB can also return **command**
`Key`	The lower case label on key that was released.

Some key combinations do not define a value for the `Character` field.

Properties Affected by the KeyReleaseFcn

When a callback is defined for the `KeyReleaseFcn` property, MATLAB updates the `CurrentCharacter`, `CurrentKey`, and `CurrentModifier` figure properties just before executing the callback.

Multiple Key Presses Events and a Single Key Release Event

Consider a figure having callbacks defined for both the `KeyPressFcn` and `KeyReleaseFcn`. In the case where a user presses multiple keys, one after another, MATLAB generates repeated `KeyPressFcn` events only for the last key pressed.

For example, suppose you press and hold down the **a** key, then press and hold down the **s** key. MATLAB generates repeated `KeyPressFcn` events for the **a** key until the **s** key is pressed, at which point MATLAB generates repeated `KeyPressFcn` events for the **s** key. If the **s** key is then released, a `KeyReleaseFcn` event is generated for the **s** key, but no new `KeyPressFcn` events are generated for the **a** key. When you then release the **a** key, the `KeyReleaseFcn` again executes.

The `KeyReleaseFcn` behavior is such that its callback is executed every time a key is released while the figure is in focus, regardless of what `KeyPressFcns` are generated.

Modifier Keys

When the user presses and releases a key and a modifier key, the modifier key is returned in the event structure `Modifier` field. If a modifier key is the only key pressed and released, it is not returned in the event structure of the `KeyReleaseFcn`, but is returned in the event structure of the `KeyPressFcn`.

Explore the Results

Click to view in editor — This link opens the MATLAB editor with the following example.

Click to run example — Press and release various key combinations while the figure has focus to see the data returned in the event structure.

The following code, creates a figure and defines a function handle callback for the `KeyReleaseFcn` property. The callback simply

displays the values returned by the event structure and enables you to explore the `KeyReleaseFcn` behavior when you release various key combinations.

```
function key_releaseFcn
   figure('KeyReleaseFcn',@cb)

   function cb(src,evnt)
      if ~isempty(evnt.Modifier)
         for ii = 1:length(evnt.Modifier)
            out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',evnt.Character,evnt.Mod
            disp(out)
         end
      else
         out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',evnt.Character,'No modifie
         disp(out)
      end
   end
end
```

`MenuBar`

`none | {figure}`

Enable-disable figure menu bar. This property enables you to display or hide the menu bar that MATLAB places at the top of a figure window. The default (`figure`) is to display the menu bar.

This property affects only built-in menus. Menus defined with the `uimenu` command are not affected by this property.

If you start MATLAB with the `nojvm` option, figures do not display the menu bar because most items require Java figures.

`MinColormap`

scalar (default = 64)

Minimum number of color table entries used. This property specifies the minimum number of system color table entries used

by MATLAB to store the colormap defined for the figure (see the `ColorMap` property). In certain situations, you may need to increase this value to ensure proper use of colors.

For example, suppose you are running color-intensive applications in addition to MATLAB and have defined a large figure colormap (e.g., 150 to 200 colors). MATLAB may select colors that are close but not exact from the existing colors in the system color table because there are not enough slots available to define all the colors you specified.

To ensure that MATLAB uses exactly the colors you define in the figure colormap, set `MinColorMap` equal to the length of the colormap.

```
set(gcf,'MinColormap',length(get(gcf,'ColorMap')))
```

Note that the larger the value of `MinColorMap`, the greater the likelihood that other windows (including other MATLAB figure windows) will be displayed in false colors.

`Name`

string

Figure window title. This property specifies the title displayed in the figure window. By default, `Name` is empty and the figure title is displayed as `Figure 1`, `Figure 2`, and so on. When you set this parameter to a string, the figure title becomes `Figure 1: <`*`string`*`>`. See the `NumberTitle` property.

`NextPlot`

`new | {add} | replace | replacechildren`

How to add next plot. `NextPlot` determines which figure MATLAB uses to display graphics output. If the value of the current figure is

- `new` — Create a new figure to display graphics (unless an existing parent is specified in the graphing function as a property/value pair).

- `add` — Use the current figure to display graphics (the default).
- `replace` — Reset all figure properties except `Position` to their defaults and delete all figure children before displaying graphics (equivalent to `clf reset`).
- `replacechildren` — Remove all child objects, but do not reset figure properties (equivalent to `clf`).

The `newplot` function provides an easy way to handle the `NextPlot` property. Also see the `NextPlot` axes property and "Controlling Graphics Output" for more information.

`NumberTitle`
`{on} | off` (GUIDE default `off`)

Figure window title number. This property determines whether the string `Figure No.  N` (where `N` is the figure number) is prefixed to the figure window title. See the `Name` property.

`PaperOrientation`
`{portrait} | landscape`

Horizontal or vertical paper orientation. This property determines how printed figures are oriented on the page. `portrait` orients the longest page dimension vertically; `landscape` orients the longest page dimension horizontally. See the `orient` command for more detail.

`PaperPosition`
four-element rect vector

Location on printed page. A rectangle that determines the location of the figure on the printed page. Specify this rectangle with a vector of the form

```
rect = [left, bottom, width, height]
```

where `left` specifies the distance from the left side of the paper to the left side of the rectangle and `bottom` specifies

the distance from the bottom of the page to the bottom of the rectangle. Together these distances define the lower left corner of the rectangle. `width` and `height` define the dimensions of the rectangle. The `PaperUnits` property specifies the units used to define this rectangle.

`PaperPositionMode`

`auto | {manual}`

WYSIWYG printing of figure. In `manual` mode, MATLAB honors the value specified by the `PaperPosition` property. In `auto` mode, MATLAB prints the figure the same size as it appears on the computer screen, centered on the page.

See the Pixels per Inch Solution for information on specifying a pixels per inch resolution setting for MATLAB figures. Doing so might be necessary to obtain a printed figure that is the same size as it is on screen.

`PaperSize`

`[width height]`

Paper size. This property contains the size of the current `PaperType`, measured in `PaperUnits`. See `PaperType` to select standard paper sizes.

`PaperType`

Select a value from the following table.

Selection of standard paper size. This property sets the `PaperSize` to one of the following standard sizes.

Property Value	Size (Width x Height)
`usletter (default)`	8.5-by-11 inches
`uslegal`	11-by-14 inches
`tabloid`	11-by-17 inches

Property Value	Size (Width x Height)
A0	841-by-1189mm
A1	594-by-841mm
A2	420-by-594mm
A3	297-by-420mm
A4	210-by-297mm
A5	148-by-210mm
B0	1029-by-1456mm
B1	728-by-1028mm
B2	514-by-728mm
B3	364-by-514mm
B4	257-by-364mm
B5	182-by-257mm
arch-A	9-by-12 inches
arch-B	12-by-18 inches
arch-C	18-by-24 inches
arch-D	24-by-36 inches
arch-E	36-by-48 inches
A	8.5-by-11 inches
B	11-by-17 inches
C	17-by-22 inches
D	22-by-34 inches
E	34-by-43 inches

Note that you may need to change the `PaperPosition` property in order to position the printed figure on the new paper size.

One solution is to use `normalized PaperUnits`, which enables MATLAB to automatically size the figure to occupy the same relative amount of the printed page, regardless of the paper size.

`PaperUnits`

```
normalized | {inches} | centimeters | points
```

Hardcopy measurement units. This property specifies the units used to define the `PaperPosition` and `PaperSize` properties. All units are measured from the lower left corner of the page. `normalized` units map the lower left corner of the page to (0, 0) and the upper right corner to (1.0, 1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).

If you change the value of `PaperUnits`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `PaperUnits` is set to the default value.

`Parent`

handle

Handle of figure's parent. The parent of a figure object is the root object. The handle to the root is always 0.

`Pointer`

```
crosshair | {arrow} | watch | topl |
topr | botl | botr | circle | cross |
fleur | left | right | top | bottom |
fullcrosshair | ibeam | custom
```

Pointer symbol selection. This property determines the symbol used to indicate the pointer (cursor) position in the figure window. Setting `Pointer` to `custom` allows you to define your own pointer symbol. See the `PointerShapeCData` property and "Specifying the Figure Pointer" for more information.

`PointerShapeCData`
16-by-16 matrix

User-defined pointer. This property defines the pointer that is used when you set the `Pointer` property to `custom`. It is a 16-by-16 element matrix defining the 16-by-16 pixel pointer using the following values:

- `1` — Color pixel black.
- `2` — Color pixel white.
- `NaN` — Make pixel transparent (underlying screen shows through).

Element (1,1) of the `PointerShapeCData` matrix corresponds to the upper left corner of the pointer. Setting the `Pointer` property to one of the predefined pointer symbols does not change the value of the `PointerShapeCData`. Computer systems supporting 32-by-32 pixel pointers fill only one quarter of the available pixmap.

`PointerShapeHotSpot`
two-element vector

Pointer active area. A two-element vector specifying the row and column indices in the `PointerShapeCData` matrix defining the pixel indicating the pointer location. The location is contained in the `CurrentPoint` property and the root object's `PointerLocation` property. The default value is element (1,1), which is the upper left corner.

`Position`
four-element vector

Figure position. This property specifies the size and location on the screen of the figure window. Specify the position rectangle with a four-element vector of the form:

```
rect = [left, bottom, width, height]
```

where `left` and `bottom` define the distance from the lower left corner of the screen to the lower left corner of the figure window. `width` and `height` define the dimensions of the window. See the `Units` property for information on the units used in this specification. The `left` and `bottom` elements can be negative on systems that have more than one monitor.

Position of Docked Figures

If the figure is docked in the MATLAB desktop, then the `Position` property is specified with respect to the figure group container instead of the screen.

Moving and Resizing Figures

You can use the `get` function to obtain this property and determine the position of the figure and you can use the `set` function to resize and move the figure to a new location. You cannot set the figure `Position` when it is docked.

Note that on MS-Windows systems, figure windows cannot be less than 104 pixels wide, regardless of the value of the `Position` property.

`Renderer`

```
painters | zbuffer | OpenGL
```

Rendering method used for screen and printing This property enables you to select the method used to render MATLAB graphics. The choices are

- `painters` — The original rendering method used by MATLAB is faster when the figure contains only simple or small graphics objects.
- `zbuffer` — MATLAB draws graphics objects faster and more accurately because objects are colored on a per-pixel basis and MATLAB renders only those pixels that are visible in the scene

(thus eliminating front-to-back sorting errors). Note that this method can consume a lot of system memory if MATLAB is displaying a complex scene.

- `OpenGL` — OpenGL is a renderer that is available on many computer systems. This renderer is generally faster than painters or zbuffer and in some cases enables MATLAB to access graphics hardware that is available on some systems.

Hardware vs. Software OpenGL Implementations

There are two kinds of OpenGL implementations — hardware and software.

The hardware implementation makes use of special graphics hardware to increase performance and is therefore significantly faster than the software version. Many computers have this special hardware available as an option or may come with this hardware right out of the box.

Software implementations of OpenGL are much like the ZBuffer renderer that is available on MATLAB Version 5.0 and later; however, OpenGL generally provides superior performance to ZBuffer.

OpenGL Availability

OpenGL is available on all computers that run MATLAB. MATLAB automatically finds hardware accelerated versions of OpenGl if such versions are available. If the hardware accelerated version is not available, then MATLAB uses the software version (except on Macintosh systems, which do not support software OpenGL).

The following software versions are available:

- On UNIX systems, MATLAB uses the software version of OpenGL that is included in the MATLAB distribution.

- On MS-Windows, OpenGL is available as part of the operating system. If you experience problems with OpenGL, contact your graphics driver vendor to obtain the latest qualified version of OpenGL.
- On Macintosh systems. software OpenGL is not available.

MATLAB issues a warning if it cannot find a usable OpenGL library.

Selecting Hardware Accelerated or Software OpenGL

MATLAB enables you to switch between hardware accelerated and software OpenGL. However, MS-Windows and Unix systems behave differently:

- On MS-Windows systems, you can toggle between software and hardware versions any time during the MATLAB session.
- On UNIX systems, you must set the OpenGL version before MATLAB initializes OpenGL. Therefore, you cannot issue the `opengl info` command or create graphs before you call `opengl software`. To re-enable hardware accelerated OpenGL, you must restart MATLAB.
- On Macintosh systems. software OpenGL is not available.

If you do not want to use hardware OpenGL, but do want to use object transparency, you can issue the following command.

```
opengl software
```

This command forces MATLAB to use software OpenGL. Software OpenGL is useful if your hardware accelerated version of OpenGL does not function correctly and you want to use image, patch, or surface transparency, which requires the OpenGL renderer. To reenable hardware OpenGL, use the command

```
opengl hardware
```

on MS-Windows systems or restart MATLAB on UNIX systems.

By default, MATLAB uses hardware accelerated OpenGL.

See the `opengl` reference page for additional information

Determining What Version You Are Using

To determine the version and vendor of the OpenGL library that MATLAB is using on your system, type the following command at the MATLAB prompt:

```
opengl info
```

The returned information contains a line that indicates if MATLAB is using software (`Software = true`) or hardware accelerated (`Software = false`) OpenGL.

This command also returns a string of extensions to the OpenGL specification that are available with the particular library MATLAB is using. This information is helpful to The MathWorks, so please include this information if you need to report bugs.

Note that issuing the `opengl info` command causes MATLAB to initialize OpenGL.

OpenGL vs. Other MATLAB Renderers

There are some differences between drawings created with OpenGL and those created with the other renderers. The OpenGL specific differences include

- OpenGL does not do colormap interpolation. If you create a surface or patch using indexed color and interpolated face or edge coloring, OpenGL interpolates the colors through the RGB color cube instead of through the colormap.

- OpenGL does not support the `phong` value for the `FaceLighting` and `EdgeLighting` properties of `surfaces` and `patches`.
- OpenGL does not support logarithmic-scale axes.
- OpenGL and Zbuffer renderers display objects sorted in front to back order, as seen on the monitor, and lines always draw in front of faces when at the same location on the plane of the monitor. Painters sorts by child order (order specified).

If You Are Having Problems

Consult the OpenGL Technical Note if you are having problems using OpenGL. This technical note contains a wealth of information on MATLAB renderers.

`RendererMode`

```
{auto} | manual
```

Automatic or user selection of renderer. This property enables you to specify whether MATLAB should choose the `Renderer` based on the contents of the figure window, or whether the `Renderer` should remain unchanged.

When the `RendererMode` property is set to `auto`, MATLAB selects the rendering method for printing as well as for screen display based on the size and complexity of the graphics objects in the figure.

For printing, MATLAB switches to `zbuffer` at a greater scene complexity than for screen rendering because printing from a Z-buffered figure can be considerably slower than one using the `painters` rendering method, and can result in large PostScript files. However, the output does always match what is on the screen. The same holds true for OpenGL: the output is the same as that produced by the ZBuffer renderer — a bitmap with a resolution determined by the `print` command's `-r` option.

Criteria for Autoselection of OpenGL Renderer

When the `RendererMode` property is set to `auto`, MATLAB uses the following criteria to determine whether to select the OpenGL renderer:

If the `opengl` autoselection mode is `autoselect`, MATLAB selects OpenGL if

- The host computer has OpenGL installed and is in True Color mode (OpenGL does not fully support 8-bit color mode).
- The figure contains no logarithmic axes (logarithmic axes are not supported in OpenGL).
- MATLAB would select `zbuffer` based on figure contents.
- Patch objects' faces have no more than three vertices (some OpenGL implementations of patch tessellation are unstable).
- The figure contains less than 10 uicontrols (OpenGL clipping around uicontrols is slow).
- No line objects use markers (drawing markers is slow).
- Phong lighting is not specified (OpenGL does not support Phong lighting; if you specify Phong lighting, MATLAB uses the ZBuffer renderer).

  Or

- Figure objects use transparency (OpenGL is the only MATLAB renderer that supports transparency).

When the `RendererMode` property is set to `manual`, MATLAB does not change the `Renderer`, regardless of changes to the figure contents.

`Resize`

`{on} | off`

Window resize mode. This property determines if you can resize the figure window with the mouse. `on` means you can resize the window, `off` means you cannot. When `Resize` is `off`, the figure window does not display any resizing controls (such as boxes at the corners), to indicate that it cannot be resized.

`ResizeFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Window resize callback function. MATLAB executes the specified callback function whenever you resize the figure window and also when the figure is created. You can query the figure's `Position` property to determine the new size and position of the figure. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

You can use `ResizeFcn` to maintain a GUI layout that is not directly supported by the MATLAB `Position/Units` paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following `ResizeFcn` accomplishes this; it keeps the `uicontrol` whose `Tag` is `'StatusBar'` 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the `Tag` property to retrieve the uicontrol handle, and the `gcbo` function to retrieve the figure handle. Also note the defensive programming regarding figure `Units`, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards.

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
```

```
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure `Position` from within the `ResizeFcn` callback; however, the `ResizeFcn` is not called again as a result.

Note that the `print` command can cause the `ResizeFcn` to be called if the `PaperPositionMode` property is set to `manual` and you have defined a resize function. If you do not want your resize function called by `print`, set the `PaperPositionMode` to `auto`.

See “Introduction” for an example of how to implement a resize function for a GUI.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`Selected`

`on | off`

Is object selected? This property indicates whether the figure is selected. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`

`{on} | off`

figures do not indicate selection.

`SelectionType`

`{normal} | extend | alt | open`

Mouse selection type. MATLAB maintains this property to provide information about the last mouse button press that occurred within the figure window. This information indicates the type of selection made. Selection types are actions that are generally associated with particular responses from the user interface

software (e.g., single-clicking a graphics object places it in move or resize mode; double-clicking a filename opens it, etc.).

The physical action required to make these selections varies on different platforms. However, all selection types exist on all platforms.

Selection Type	MS-Windows	X-Windows
`Normal`	Click left mouse button.	Click left mouse button.
`Extend`	**Shift** - click left mouse button or click both left and right mouse buttons.	**Shift** - click left mouse button or click middle mouse button.
`Alternate`	**Control** - click left mouse button or click right mouse button.	**Control** - click left mouse button or click right mouse button.
`Open`	Double-click any mouse button.	Double-click any mouse button.

Note that the `ListBox` style of `uicontrols` sets the figure `SelectionType` property to `normal` to indicate a single mouse click or to `open` to indicate a double mouse click. See `uicontrol` for information on how this property is set when you click a uicontrol object.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as

global variables or pass them as arguments between callback routines.

For example, suppose you want to direct all graphics output from an M-file to a particular figure, regardless of user actions that may have changed the current figure. To do this, identify the figure with a `Tag`.

```
figure('Tag','Plotting Figure')
```

Then make that figure the current figure before drawing by searching for the `Tag` with `findobj`.

```
figure(findobj('Tag','Plotting Figure'))
```

`Toolbar`

`none | {auto} | figure`

Control display of figure toolbar. The `Toolbar` property enables you to control whether MATLAB displays the default figure toolbar on figures. There are three possible values:

- `none` — do not display the figure toolbar
- `auto` — display the figure toolbar, but remove it if a uicontrol is added to the figure
- `figure` — display the figure toolbar

Note that this property affects only the figure toolbar; other toolbars (e.g., the Camera Toolbar or Plot Edit Toolbar) are not affected. Selecting **Figure Toolbar** from the figure **View** menu sets this property to `figure`.

If you start MATLAB with the `nojvm` option, figures do not display the toolbar because most tool require Java figures.

`Type`

`string (read only)`

Object class. This property identifies the kind of graphics object. For figures, `Type` is always the string `'figure'`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the figure. Assign this property the handle of a uicontextmenu object created in the figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the figure.

`Units`

```
{pixels} | normalized | inches |
centimeters | points | characters
```

Units of measurement. This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower left corner of the window.

- `normalized` units map the lower left corner of the figure window to (0,0) and the upper right corner to (1.0,1.0).
- `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).
- The size of a `pixel` depends on screen resolution.
- `characters` units are defined by characters from the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

This property affects the `CurrentPoint` and `Position` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

When specifying the units as property/value pairs during object creation, you must set the `Units` property before specifying the properties that you want to use these units.

`UserData`
matrix

User-specified data. You can specify `UserData` as any matrix you want to associate with the figure object. The object does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Object visibility. The `Visible` property determines whether an object is displayed on the screen. If the `Visible` property of a figure is `off`, the entire figure window is invisible.

A note about using the window button properties
Your window button callback functions might need to update the display by calling `drawnow` or `pause`, which causes MATLAB to process all events in the queue. Processing the event queue can cause your window button callback functions to be reentered. For example, a `drawnow` in the `WindowButtonDownFcn` might result in the `WindowButtonDownFcn` being called again before the first call has finished. You should design your code to handle reentrancy and you should not depend on global variables that might change state during reentrance.

You can use the `Interruptible` and `BusyAction` figure properties to control how events interact.

`WindowButtonDownFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. Use this property to define a callback that MATLAB executes whenever you press a mouse

button while the pointer is in the figure window. See the `WindowButtonMotionFcn` property for an example.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`WindowButtonMotionFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Mouse motion callback function. Use this property to define a callback that MATLAB executes whenever you move the pointer within the figure window. Define the `WindowButtonMotionFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

Example using all window button properties

Click to view in editor — This example enables you to use mouse motion to draw lines. It uses all three window button functions.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

Note On some computer systems, the `WindowButtonMotionFcn` is executed when a figure is created even though there has been no mouse motion within the figure.

`WindowButtonUpFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button release callback function. Use this property to define a callback that MATLAB executes whenever you release a mouse button. Define the `WindowButtonUpFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure)

The button up event is associated with the figure window in which the preceding button down event occurred. Therefore, the pointer need not be in the figure window when you release the button to generate the button up event.

If the callback routines defined by `WindowButtonDownFcn` or `WindowButtonMotionFcn` contain `drawnow` commands or call other functions that contain `drawnow` commands and the `Interruptible` property is set to `off`, the `WindowButtonUpFcn` might not be called. You can prevent this problem by setting `Interruptible` to `on`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`WindowScrollWheelFcn`

string, functional handle, or cell array containing function handle and additional arguments

Respond to mouse scroll wheel. Use this property to define a callback that MATLAB executes when the mouse wheel is scrolled while the figure has focus. MATLAB executes the callback with each single mouse wheel click.

Note that it is possible for another object to capture the event from MATLAB. For example, if the figure contains Java or ActiveX control objects that are listening for mouse scroll wheel

events, then these objects can consume the events and prevent the `WindowScrollWheelFcn` from executing.

There is no default callback defined for this property.

WindowScrollWheelFcn Event Structure

When the callback is a function handle, MATLAB passes a structure to the callback function that contains the following fields.

Field	Contents
`VerticalScrollCount`	A positive or negative integer that indicates the number of scroll wheel clicks. Positive values indicate clicks of the wheel scrolled in the down direction. Negative values indicate clicks of the wheel scrolled in the up direction.
`VerticalScrollAmount`	The current system setting for the number of lines that are scrolled for each click of the scroll wheel. If the mouse property setting for scrolling is set to `One screen at a time`, VerticalScrollAmount returns a value of 1.

Effects On Other Properties

- `CurrentObject` property — mouse scrolling does not update this figure property
- `CurrentPoint` property — if there is no callback defined for the `WindowScrollWheelFcn` property, then MATLAB does not update the `CurrentPoint` property as the scroll wheel is turned. However, if there is a callback defined for the `WindowScrollWheelFcn` property, then MATLAB updates the `CurrentPoint` property just before executing the callback. This enables you to determine the point at which the mouse scrolling occurred.
- `HitTest` property — the `WindowScrollWheelFcn` callback executes regardless of the setting of the figure `HitTest` property.
- `SelectionType` property — the `WindowScrollWheelFcn` callback has no effect on this property.

Values Returned by VerticalScrollCount

When a user moves the mouse scroll wheel by one click, MATLAB increments the count by +/- 1, depending on the direction of the scroll (scroll down being positive). When MATLAB calls the `WindowScrollWheelFcn` callback, the counter is reset. In most cases, this means that the absolute value of the returned value is 1. However, if the `WindowScrollWheelFcn` callback takes a long enough time to return and/or the user spins the scroll wheel very fast, then the returned value can have an absolute value greater than one.

The actual value returned by `VerticalScrollCount` is the algebraic sum of all scroll wheel clicks that occurred since last processed. This enables your callback to respond correctly to the user's action.

Example

Click to view in editor — This example creates a graph of a function and enables you to use the mouse scroll wheel to change the range over which a mathematical function is evaluated and update the graph to reflect the new limits as you turn the scroll wheel.

Click to run example — Mouse over the figure and scroll your mouse wheel.

Related Information

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`WindowStyle`

`{normal} | modal | docked`

Normal, modal, or dockable window behavior. When `WindowStyle` is set to `modal`:

- The figure window traps all keyboard and mouse events over all MATLAB windows as long as they are visible.
- Windows belonging to applications other than MATLAB are unaffected.
- Modal figures remain stacked above all normal figures and the MATLAB command window.
- When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to `WindowStyle normal` , or is deleted. At that time, focus reverts to the window that last had focus.

Use modal figures to create dialog boxes that force the user to respond without being able to interact with other windows. Typing **Control C** while the figure has focus causes all figures

with `WindowStyle modal` to revert to `WindowStyle normal` , allowing you to type at the command line.

Invisible Modal Figures

Figures with `WindowStyle modal` and `Visible off` do not behave modally until they are made visible, so it is acceptable to hide a `modal` window instead of destroying it when you want to reuse it.

Changing Modes

You can change the `WindowStyle` of a figure at any time, including when the figure is visible and contains children. However, on some systems this may cause the figure to flash or disappear and reappear, depending on the windowing system's implementation of normal and modal windows. For best visual results, you should set `WindowStyle` at creation time or when the figure is invisible.

Window Decorations on Modal Figures

Modal figures do not display `uimenu` children, built-in menus, or toolbars but it is not an error to create uimenus in a modal figure or to change `WindowStyle` to `modal` on a figure with uimenu children. The uimenu objects exist and their handles are retained by the figure. If you reset the figure's `WindowStyle` to `normal`, the uimenus are displayed.

Docked WindowStyle

When `WindowStyle` is set to `docked`, the figure is docked in the desktop or a document window. When you issue the following command,

```
set(figure_handle,'WindowStyle','docked')
```

MATLAB docks the figure identified by *`figure_handle`* and sets the `DockControls` property to `on`, if it was `off`.

Note that if `WindowStyle` is `docked`, you cannot set the `DockControls` property to `off`.

The value of the `WindowStyle` property is not changed by calling `reset` on a figure.

`WVisual`
identifier string (MS Windows only)

Specify pixel format for figure. MATLAB automatically selects a pixel format for figures based on your current display settings, the graphics hardware available on your system, and the graphical content of the figure.

Usually, MATLAB chooses the best pixel format to use in any given situation. However, in cases where graphics objects are not rendered correctly, you might be able select a different pixel format and improve results. See for more information.

Querying Available Pixel Formats on Window Systems

You can determine what pixel formats are available on your system for use with MATLAB using the following statement:

```
set(gcf,'WVisual')
```

MATLAB returns a list of the currently available pixel formats for the current figure. For example, the following are the first three entries from a typical list.

01 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, Opengl, GDI, Window)

02 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, Opengl, Double Buffered, Window)

03 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, Opengl, Double Buffered, Window)

Use the number at the beginning of the string to specify which pixel format to use. For example,

```
set(gcf,'WVisual','02')
```

specifies the second pixel format in the list above. Note that pixel formats might differ on your system.

Understanding the WVisual String

The string returned by querying the `WVisual` property provide information on the pixel format. For example,

- `RGB 16 bits(05 06 05 00)` – indicates true color with 16-bit resolution (5 bits for red, 6 bits for green, 5 bits for blue, and 0 for alpha (transparency). MATLAB requires true color.
- `zdepth 24` – indicates 24-bit resolution for sorting object's front to back position on the screen. Selecting pixel formats with higher (24 or 32) zdepth might solve sorting problems.
- `Hardware Accelerated` – some graphics functions may be performed by hardware for increased speed. If there are incompatibilities between your particular graphic hardware and MATLAB, select a pixel format in which the term `Generic` appears instead of `Hardware Accelerated`.
- `Opengl` – supports OpenGL. See for more information.
- `GDI` – supports for Windows 2-D graphics interface.
- `Double Buffered` – support for double buffering with the OpenGL renderer. Note that the figure `DoubleBuffer` property applies only to the painters renderer.
- `Bitmap` – support for rendering into a bitmap (as opposed to drawing in the window)
- `Window` – support for rendering into a window

Pixel Formats and OpenGL

If you are experiencing problems using hardware OpenGL on your system, you can try using generic OpenGL, which is implemented in software. To do this, first instruct MATLAB to use the software version of OpenGL with the following statement.

```
opengl software
```

Then allow MATLAB to select best pixel format to use.

See the `Renderer` property for more information on how MATLAB uses OpenGL.

`WVisualMode`
`auto` | `manual` (MS Windows only)

Auto or manual selection of pixel format. `VisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best pixel format to use based on your computer system and the graphical content of the figure. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `WVisual` property sets this property to `manual`.

`XDisplay`
display identifier (UNIX only)

Contains the display used for MATLAB. You can query this property to determine the name of the display that MATLAB is using. For example, if MATLAB is running on a system called `mycomputer`, querying `XDisplay` returns a string of the following form:

```
get(gcf,'XDisplay')
ans
mycomputer:0.0
```

Setting XDisplay on Motif

If your computer uses Motif-based figures, you can specify the display MATLAB uses for a figure by setting the value of the figure's `XDisplay` property. For example, to display the current figure on a system called `fred`, use the command

```
set(gcf,'XDisplay','fred:0.0')
```

`XVisual`
visual identifier (UNIX only)

Select visual used by MATLAB. You can select the visual used by MATLAB by setting the `XVisual` property to the desired visual ID. This can be useful if you want to test your application on an 8-bit or grayscale visual. To see what visuals are available on your system, use the UNIX `xdpyinfo` command. From MATLAB, type

```
!xdpyinfo
```

The information returned contains a line specifying the visual ID. For example,

```
visual id:    0x23
```

To use this visual with the current figure, set the `XVisual` property to the ID.

```
set(gcf,'XVisual','0x23')
```

To see which of the available visuals MATLAB can use, call `set` on the `XVisual` property:

```
set(gcf,'XVisual')
```

The following typical output shows the visual being used (in curly brackets) and other possible visuals. Note that MATLAB requires a TrueColor visual.

```
{ 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff) }
  0x24 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

```
  0x25 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x26 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x27 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x28 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x29 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x2a (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

You can also use the `glxinfo` Unix command to see what visuals are available for use with the OpenGL renderer. From MATLAB, type

```
!glxinfo
```

After providing information about the implementation of OpenGL on your system, `glxinfo` returns a table of visuals. The partial listing below shows typical output.

```
visual  x  bf lv rg d st colorbuffer ax dp st accumbuffer  ms  cav
 id dep cl sp sz l  ci b ro  r  g  b  a bf th cl  r  g  b  a ns b eat
-----------------------------------------------------------------
-
0x23 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0  0  0  0  0 0 None
0x24 24 tc  0 24  0 r  .  .  8  8  8  8  0  0  0  0  0  0  0 0 None
0x25 24 tc  0 24  0 r  y  .  8  8  8  8  0 24  8  0  0  0  0 0 None
0x26 24 tc  0 24  0 r  .  .  8  8  8  8  0 24  8  0  0  0  0 0 None
0x27 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0 16 16 16  0 0 Slow
```

The third column is the class of visual. `tc` means a true color visual. Note that some visuals may be labeled `Slow` under the caveat column. Such visuals should be avoided.

To determine which visual MATLAB will use by default with the OpenGL renderer, use the MATLAB `opengl info` command. The returned entry for the visual might look like the following.

```
Visual = 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

Experimenting with a different TrueColor visual may improve certain rendering problems.

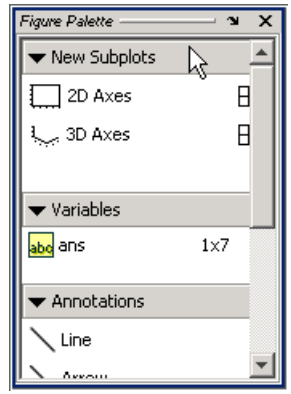
`XVisualMode`
`auto | manual`

Auto or manual selection of visual. `VisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best visual to use based on the number of colors, availability of the OpenGL extension, etc. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `XVisual` property sets this property to `manual`.

figurepalette

Purpose

Show or hide figure palette



GUI Alternatives

Click the larger **Plotting Tools** icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Open or close the **Figure Palette** tool from the figure's **View** menu. For details, see "The Figure Palette" in the MATLAB Graphics documentation.

Syntax

```
figurepalette('show')
figurepalette('hide')
figurepalette('toggle')
figurepalette(figure_handle,...)
```

Description

`figurepalette('show')` displays the palette on the current figure.

`figurepalette('hide')` hides the palette on the current figure.

`figurepalette('toggle')` or `figurepalette` toggles the visibility of the palette on the current figure.

`figurepalette(figure_handle,...)` shows or hides the palette on the figure specified by `figure_handle`.

See Also

plottools, plotbrowser, propertyeditor

fileattrib

Purpose

Set or get attributes of file or directory

Syntax

```
fileattrib
fileattrib('name')
fileattrib('name','attrib')
fileattrib('name','attrib','users')
fileattrib('name','attrib','users','s')
[status,message,messageid] =  fileattrib('name','attrib',
   'users','s')
```

Description

The `fileattrib` function is like the DOS `attrib` command or the UNIX `chmod` command.

`fileattrib` displays the attributes for the current directory. Values are as follows.

Value	Description
`0`	Attribute is off
`1`	Attribute is set (on)
`NaN`	Attribute does not apply

`fileattrib('name')` displays the attributes for `name`, where `name` is the absolute or relative pathname for a directory or file. Use the wildcard `*` at the end of `name` to view attributes for all matching files.

`fileattrib('name','attrib')` sets the attribute for `name`, where `name` is the absolute or relative pathname for a directory or file. Specify the `+` qualifier before the attribute to set it, and specify the `-` qualifier before the attribute to clear it. Use the wildcard `*` at the end of `name` to set attributes for all matching files. Values for *`attrib`* are as follows.

Value for attrib	Description
a	Archive (Windows only)
h	Hidden file (Windows only)
s	System file (Windows only)
w	Write access (Windows and UNIX)
x	Executable (UNIX only)

For example, `fileattrib('myfile.m','+w')` makes `myfile.m` a writable file.

`fileattrib('name','attrib','users')` sets the attribute for `name`, where `name` is the absolute or relative pathname for a directory or file, and defines which users are affected by *`attrib`*, where *`users`* is applicable only for UNIX systems. For more information about these attributes, see UNIX reference information for `chmod`. The default value for *`users`* is `u`. Values for *`users`* are

Value for users	Description
a	All users
g	Group of users
o	All other users
u	Current user

`fileattrib('name','attrib','users','s')` sets the attribute for `name`, where `name` is the absolute or relative pathname for a file or a directory and its contents, and defines which users are affected by *`attrib`*. Here the **s** specifies that *`attrib`* be applied to all contents of `name`, where `name` is a directory.

`[status,message,messageid] = fileattrib('name','attrib','users','s')` sets the attribute for `name`, returning the status, a message, and the

MATLAB error message ID (see `error` and `lasterror`). Here, `status` is `1` for success and is `0` for error. If `attrib`, *`users`*, and **`s`** are not specified, and `status` is `1`, `message` is a structure containing the file attributes and `messageid` is blank. If `status` is `0`, `messageid` contains the error. If you use a wildcard `*` at the end of `name`, `mess` will be a structure.

Examples

Get Attributes of File

To view the attributes of `myfile.m`, type

```
fileattrib('myfile.m')
```

MATLAB returns

```
             Name: 'd:/work/myfile.m'
          archive: 0
           system: 0
           hidden: 0
        directory: 0
         UserRead: 1
        UserWrite: 0
      UserExecute: 1
        GroupRead: NaN
       GroupWrite: NaN
     GroupExecute: NaN
        OtherRead: NaN
       OtherWrite: NaN
     OtherExecute: NaN
```

`UserWrite` is `0`, meaning `myfile.m` is read only. The `Group` and `Other` values are `NaN` because they do not apply to the current operating system, Windows.

Set File Attribute

To make `myfile.m` become writable, type

```
fileattrib('myfile.m','+w')
```

Running `fileattrib('myfile.m')` now shows `UserWrite` to be `1`.

Set Attributes for Specified Users

To make the directory `d:/work/results` be a read-only directory for all users, type

```
fileattrib('d:/work/results','-w','a')
```

The `-` preceding the write attribute, `w`, specifies that write status is removed.

Set Multiple Attributes for Directory and Its Contents

To make the directory `d:/work/results` and all its contents be read only and be hidden, on Windows, type

```
fileattrib('d:/work/results','+h-w','','s')
```

Because *`users`* is not applicable on Windows systems, its value is empty. Here, `s` applies the attribute to the contents of the specified directory.

Return Status and Structure of Attributes

To return the attributes for the directory `results` to a structure, type

```
[stat,mess]=fileattrib('results')
```

MATLAB returns

```
stat =
     1

mess =
            Name: 'd:\work\results'
         archive: 0
          system: 0
          hidden: 0
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
```

```
         GroupRead: NaN
        GroupWrite: NaN
      GroupExecute: NaN
         OtherRead: NaN
        OtherWrite: NaN
      OtherExecute: NaN
```

The operation was successful as indicated by the status, `stat`, being `1`. The structure `mess` contains the file attributes. Access the attribute values in the structure. For example, typing

```
mess.Name
```

returns the path for `results`

```
ans =
d:\work\results
```

Return Attributes with Wildcard for Name

Return the attributes for all files in the current directory whose names begin with `new`.

```
[stat,mess]=fileattrib('new*')
```

MATLAB returns

```
stat =
     1

mess =
1x3 struct array with fields:
    Name
    archive
    system
    hidden
    directory
    UserRead
    UserWrite
```

```
    UserExecute
    GroupRead
    GroupWrite
    GroupExecute
    OtherRead
    OtherWrite
    OtherExecute
```

The results indicate there are three matching files. To view the filenames, type

```
mess.Name
```

MATLAB returns

```
ans =
d:\work\results\newname.m

ans =
d:\work\results\newone.m

ans =
d:\work\results\newtest.m
```

To view just the first filename, type

```
mess(1).Name

ans =
d:\work\results\newname.m
```

See Also

`copyfile`, `cd`, `dir`, `filebrowser`, `fileparts`, `ls`, `mfilename`, `mkdir`, `movefile`, `rmdir`

filebrowser

Purpose

Current Directory browser

GUI Alternatives

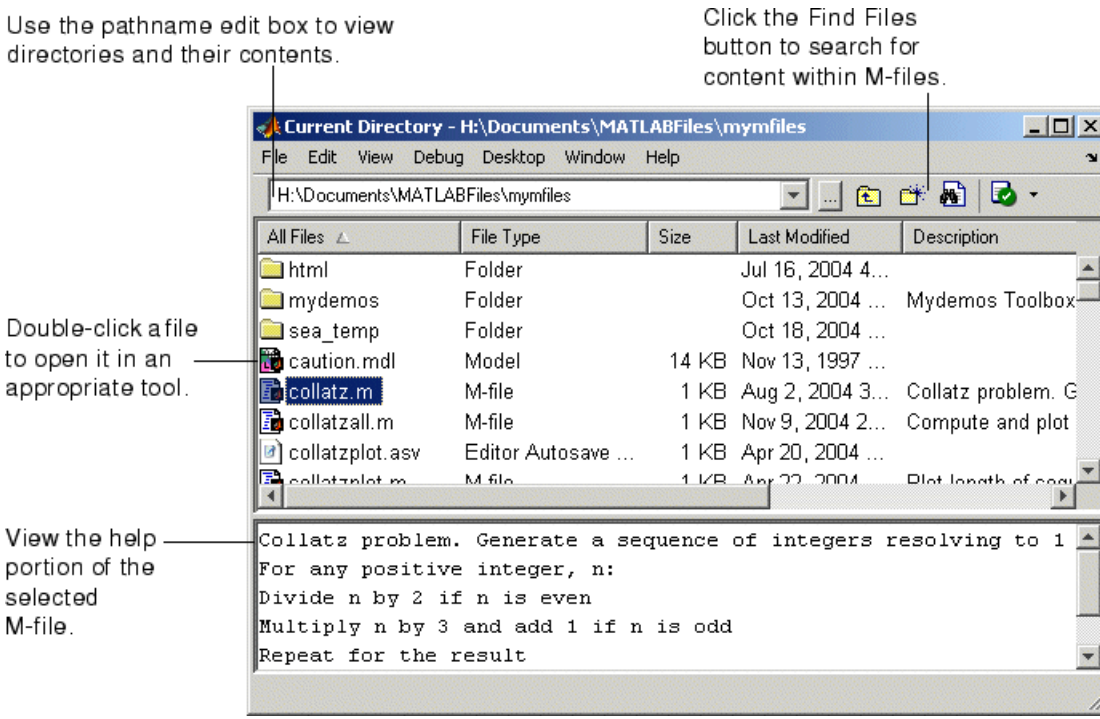
As an alternative to the `filebrowser` function, select **Desktop > Current Directory** in the MATLAB desktop.

Syntax

```
filebrowser
```

Description

`filebrowser` displays the “Current Directory Browser”.



See Also

`cd`, `copyfile`, `fileattrib`, `ls`, `mkdir`, `movefile`, `pwd`, `rmdir`

Purpose

Readable file formats

Description

This table shows the file formats that MATLAB is capable of reading.

File Format	Extension	File Content	Read Command	Returns
Text	MAT	Saved MATLAB workspace	`load`	Variables in the file
	CSV	Comma-separated numbers	`csvread`	Double array
	DLM	Delimited text	`dlmread`	Double array
	TAB	Tab-separated text	`dlmread`	Double array
Scientific Data	CDF	Data in Common Data Format	`cdfread`	Cell array of CDF records
	FITS	Flexible Image Transport System data	`fitsread`	Primary or extension table data
	HDF4	Data in Hierarchical Data Format, version 4	`hdfread`	HDF or HDF-EOS data set
	HDF5	Data in Hierarchical Data Format, version 5	`hdf5read`	HDF5 data set
Spreadsheet	XLS	Excel worksheet	`xlsread`	Double or cell array
	WK1	Lotus 123 worksheet	`wk1read`	Double or cell array

File Format	Extension	File Content	Read Command	Returns
Image	TIFF	TIFF image	`imread`	True color, grayscale, or indexed image(s)
	PNG	PNG image	`imread`	True color, grayscale, or indexed image
	HDF4	HDF4 image	`imread`	True color, grayscale, or indexed image(s)
	BMP	BMP image	`imread`	True color or indexed image
	JPEG	JPEG image	`imread`	True color or grayscale image
	GIF	GIF image	`imread`	Indexed image
	PCX	PCX image	`imread`	Indexed image
	XWD	XWD image	`imread`	Indexed image
	CUR	Cursor image	`imread`	Indexed image
	ICO	Icon image	`imread`	Indexed image
Audio file	AU	NeXT/SUN sound	`auread`	Sound data and sample rate
	WAV	Microsoft WAVE sound	`wavread`	Sound data and sample rate
Movie	AVI	Audio/video	`aviread`	MATLAB movie

See Also `fscanf`, `fread`, `textread`, `importdata`

Purpose

Character to separate file name and internal function name

Syntax

```
M = filemarker
```

Description

`M = filemarker` returns the character that separates a file and a within-file function name.

Examples

On Windows, for example, `filemarker` returns the '>' character:

```
filemarker
ans =
   >
```

You can use the following command on any platform to get the help text for subfunction `pdeodes` defined in M-file `pdepe.m`:

```
helptext = help(['pdepe' filemarker 'pdeodes'])

helptext =
 PDEODES  Assemble the difference equations and
    evaluate the time derivative for the ODE system.
```

See Also

`filesep`

fileparts

Purpose

Parts of file name and path

Syntax

```
[pathstr, name, ext, versn] = fileparts(filename)
```

Description

`[pathstr, name, ext, versn] = fileparts(filename)` returns the path, filename, extension, and version for the specified file. `filename` is a string enclosed in single quotes. The returned `ext` field contains a dot (`.`) before the file extension.

The `fileparts` function is platform dependent.

You can reconstruct the file from the parts using

```
fullfile(pathstr,[name ext versn])
```

Examples

Return the pieces of a file specification string to the separate string outputs `pathstr`, `name`, `ext`, and `versn`. The full file specification is

```
file = '\home\user4\matlab\classpath.txt';
```

Note that the character used to separate the segments of a pathname is dependent on the operating system you are currently running on. In this example, it is the backslash (`\`) character which is used as a separator on Windows systems. You can use the `filesep` function as shown below to insert the correct separator character:

```
sep = filesep;
file = ['' sep 'home' sep 'user4' sep 'matlab' sep ...
        'classpath.txt' ''];
```

Now use `fileparts` to return the path, filename, user name, and file version, if there is one:

```
[pathstr, name, ext, versn] = fileparts(file)

pathstr =
   \home\user4\matlab
```

```
name =
   classpath

ext =
   .txt

versn =
     ''
```

See Also

fullfile

filehandle

Purpose

Construct file handle object

Syntax

```
output = filehandle(arglist)
```

Description

`output = filehandle(arglist)` this file is a place-holder for now.

Example

See Also

`dialog`, `errordlg`, `helpdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `uiwait`, `uiresume`

"Predefined Dialog Boxes" on page 1-104 for related functions

Purpose

Directory separator for current platform

Syntax

```
f = filesep
```

Description

`f = filesep` returns the platform-specific file separator character. The file separator is the character that separates individual directory names in a path string.

Examples

On the PC,

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']

iofun_dir =

toolbox\matlab\iofun
```

On a UNIX system,

```
iodir = ['toolbox' filesep 'matlab' filesep 'iofun']

iodir =

toolbox/matlab/iofun
```

See Also

`fullfile`, `fileparts`, `pathsep`

fill

Purpose

Filled 2-D polygons

Syntax

```
fill(X,Y,C)
fill(X,Y,ColorSpec)
fill(X1,Y1,C1,X2,Y2,C2,...)
fill(...,'PropertyName',PropertyValue)
h = fill(...)
```

Description

The `fill` function creates colored polygons.

`fill(X,Y,C)` creates filled polygons from the data in `X` and `Y` with vertex color specified by `C`. `C` is a vector or matrix used as an index into the colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`. If necessary, `fill` closes the polygon by connecting the last vertex to the first.

`fill(X,Y,ColorSpec)` fills two-dimensional polygons specified by `X` and `Y` with the color specified by `ColorSpec`.

`fill(X1,Y1,C1,X2,Y2,C2,...)` specifies multiple two-dimensional filled areas.

`fill(...,'PropertyName',PropertyValue)` allows you to specify property names and values for a patch graphics object.

`h = fill(...)` returns a vector of handles to patch graphics objects, one handle per patch object.

Remarks

If `X` or `Y` is a matrix, and the other is a column vector with the same number of elements as rows in the matrix, `fill` replicates the column vector argument to produce a matrix of the required size. `fill` forms a vertex from corresponding elements in `X` and `Y` and creates one polygon from the data in each column.

The type of color shading depends on how you specify color in the argument list. If you specify color using `ColorSpec`, `fill` generates flat-shaded polygons by setting the patch object's `FaceColor` property to the corresponding RGB triple.

If you specify color using `C`, `fill` scales the elements of `C` by the values specified by the axes property `CLim`. After scaling `C`, `C` indexes the current colormap.

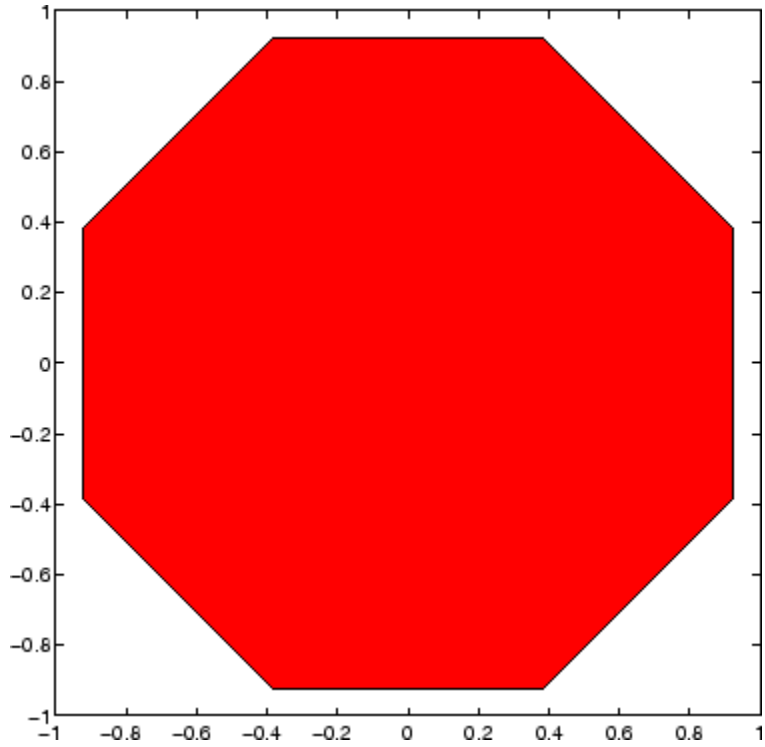
If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the `n`th patch object, where *n* is the corresponding column in `X` or `Y`.

If `C` is a column vector or a matrix, `fill` uses a linear interpolation of the vertex colors to generate polygons with interpolated colors. It sets the patch graphics object `FaceColor` property to `'interp'` and the elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill` replicates the column vector to produce the required sized matrix.

Examples

Create a red octagon.

```
t = (1/16:1/8:1)'*2*pi;
x = sin(t);
y = cos(t);
fill(x,y,'r')
axis square
```



See Also

`axis`, `caxis`, `colormap`, `ColorSpec`, `fill3`, `patch`

“Polygons and Surfaces” on page 1-90 for related functions

Purpose

Filled 3-D polygons

Syntax

```
fill3(X,Y,Z,C)
fill3(X,Y,Z,ColorSpec)
fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)
fill3(...,'PropertyName',PropertyValue)
h = fill3(...)
```

Description

The `fill3` function creates flat-shaded and Gouraud-shaded polygons.

`fill3(X,Y,Z,C)` fills three-dimensional polygons. `X`, `Y`, and `Z` triplets specify the polygon vertices. If `X`, `Y`, or `Z` is a matrix, `fill3` creates *n* polygons, where *n* is the number of columns in the matrix. `fill3` closes the polygons by connecting the last vertex to the first when necessary.

`C` specifies color, where `C` is a vector or matrix of indices into the current colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`.

`fill3(X,Y,Z,ColorSpec)` fills three-dimensional polygons defined by `X`, `Y`, and `Z` with color specified by `ColorSpec`.

`fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)` specifies multiple filled three-dimensional areas.

`fill3(...,'PropertyName',PropertyValue)` allows you to set values for specific patch properties.

`h = fill3(...)` returns a vector of handles to patch graphics objects, one handle per patch.

Algorithm

If `X`, `Y`, and `Z` are matrices of the same size, `fill3` forms a vertex from the corresponding elements of `X`, `Y`, and `Z` (all from the same matrix location), and creates one polygon from the data in each column.

If X, Y, or Z is a matrix, fill3 replicates any column vector argument to produce matrices of the required size.

If you specify color using ColorSpec, fill3 generates flat-shaded polygons and sets the patch object FaceColor property to an RGB triple.

If you specify color using C, fill3 scales the elements of C by the axes property CLim, which specifies the color axis scaling parameters, before indexing the current colormap.

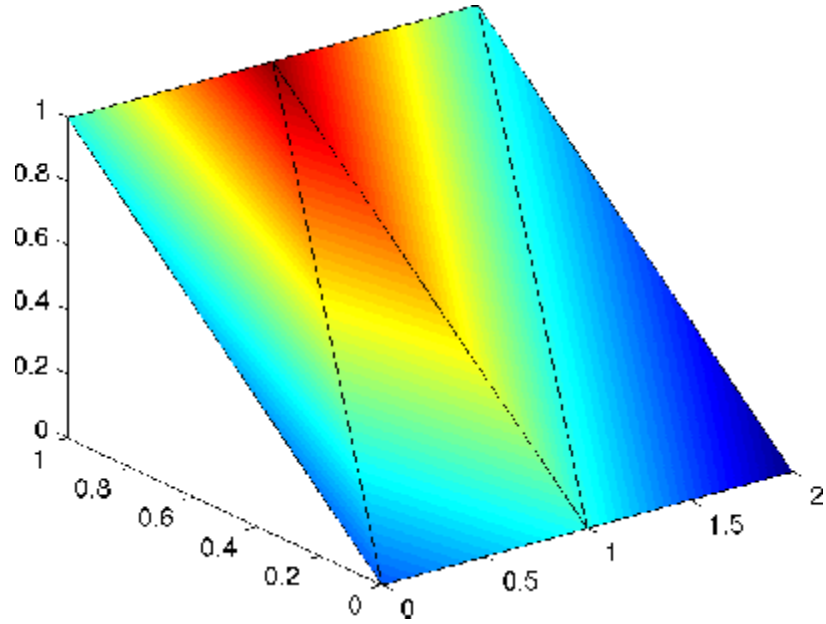
If C is a row vector, fill3 generates flat-shaded polygons and sets the FaceColor property of the patch objects to 'flat'. Each element becomes the CData property value for the respective patch object.

If C is a column vector or a matrix, fill3 generates polygons with interpolated colors and sets the patch object FaceColor property to 'interp'. fill3 uses a linear interpolation of the vertex colormap indices when generating polygons with interpolated colors. The elements in one column become the CData property value for the respective patch object. If C is a column vector, fill3 replicates the column vector to produce the required sized matrix.

Examples

Create four triangles with interpolated colors.

```
X = [0 1 1 2;1 1 2 2;0 0 1 1];
Y = [1 1 1 1;1 0 1 0;0 0 0 0];
Z = [1 1 1 1;1 0 1 0;0 0 0 0];
C = [0.5000 1.0000 1.0000 0.5000;
     1.0000 0.5000 0.5000 0.1667;
     0.3330 0.3330 0.5000 0.5000];
fill3(X,Y,Z,C)
```



See Also

`axis`, `caxis`, `colormap`, `ColorSpec`, `fill`, `patch`

"Polygons and Surfaces" on page 1-90 for related functions

Purpose

1-D digital filter

Syntax

```
y = filter(b,a,X)
[y,zf] = filter(b,a,X)
[y,zf] = filter(b,a,X,zi)
y = filter(b,a,X,zi,dim)
[...] = filter(b,a,X,[],dim)
```

Description

The `filter` function filters a data sequence using a digital filter which works for both real and complex inputs. The filter is a *direct form II transposed* implementation of the standard difference equation (see "Algorithm").

`y = filter(b,a,X)` filters the data in vector `X` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. If `a(1)` is not equal to `1`, `filter` normalizes the filter coefficients by `a(1)`. If `a(1)` equals `0`, `filter` returns an error.

If `X` is a matrix, `filter` operates on the columns of `X`. If `X` is a multidimensional array, `filter` operates on the first nonsingleton dimension.

`[y,zf] = filter(b,a,X)` returns the final conditions, `zf`, of the filter delays. If `X` is a row or column vector, output `zf` is a column vector of `max(length(a),length(b))-1`. If `X` is a matrix, `zf` is an array of such vectors, one for each column of `X`, and similarly for multidimensional arrays.

`[y,zf] = filter(b,a,X,zi)` accepts initial conditions, `zi`, and returns the final conditions, `zf`, of the filter delays. Input `zi` is a vector of length `max(length(a),length(b))-1`, or an array with the leading dimension of size `max(length(a),length(b))-1` and with remaining dimensions matching those of `X`.

`y = filter(b,a,X,zi,dim)` and `[...]  = filter(b,a,X,[],dim)` operate across the dimension `dim`.

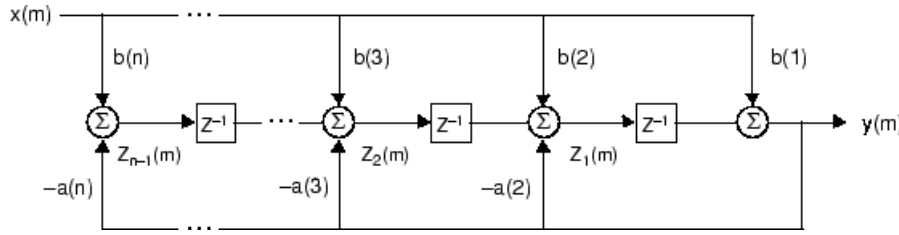
Example

You can use `filter` to find a running average without using a `for` loop. This example finds the running average of a 16-element vector, using a window size of 5.

```
data = [1:0.2:4]';
windowSize = 5;
filter(ones(1,windowSize)/windowSize,1,data)

ans =
    0.2000
    0.4400
    0.7200
    1.0400
    1.4000
    1.6000
    1.8000
    2.0000
    2.2000
    2.4000
    2.6000
    2.8000
    3.0000
    3.2000
    3.4000
    3.6000
```

Algorithm

The `filter` function is implemented as a direct form II transposed structure,



or

```
y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                 - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

where `n-1` is the filter order, which handles both FIR and IIR filters [1], `na` is the feedback filter order, and `nb` is the feedforward filter order.

The operation of `filter` at sample m is given by the time domain difference equations

$$
\begin{aligned}
y(m) &= b(1)x(m) + z_1(m-1) \\
z_1(m) &= b(2)x(m) + z_2(m-1) - a(2)y(m) \\
\vdots &= \quad \vdots \qquad\qquad \vdots \\
z_{n-2}(m) &= b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m) \\
z_{n-1}(m) &= b(n)x(m) - a(n)y(m)
\end{aligned}
$$

The input-output description of this filtering operation in the z-transform domain is a rational transfer function,

$$
Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \ldots + a(na+1)z^{-na}} X(z)
$$

See Also

`filter2`

`filtfilt`, `filtic` in the Signal Processing Toolbox

References

[1] Oppenheim, A. V. and R.W. Schafer. *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 311-312.

Purpose

Shape frequency content of time series

Syntax

```
ts2 = filter(ts1,b,a)
ts2 = filter(ts1,b,a,Index)
```

Description

`ts2 = filter(ts1,b,a)` applies the transfer function filter $b(z^{-1})/a(z^{-1})$ to the data in the `timeseries` object `ts1`.

`b` and `a` are the coefficient arrays of the transfer function numerator and denominator, respectively.

`ts2 = filter(ts1,b,a,Index)` uses the optional `Index` integer array to specify the columns or rows to filter. When `ts.IsTimeFirst` is `true`, `Index` specifies one or more data columns. When `ts.IsTimeFirst` is `false`, Index specifies one or more data rows.

Remarks

The time-series data must be uniformly sampled to use this filter.

The following function

```
y = filter(b,a,x)
```

creates filtered data `y` by processing the data in vector `x` with the filter described by vectors `a` and `b`.

The `filter` function is a general tapped delay-line filter, described by the difference equation

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(nb)x(n-nb+1) - a(2)y(n-1) - \ldots - a(N_a)y(n-N_b+1)$$

Here, n is the index of the current sample, N_a is the order of the polynomial described by vector `a`, and N_b is the order of the polynomial described by vector `b`. The output $y(n)$ is a linear combination of current and previous inputs, $x(n)$ $x(n\text{-}1)$..., and previous outputs, $y(n\text{-}1)$ $y(n\text{-}2)$... .

You use the discrete filter to shape the data by applying a transfer function to the input signal.

Depending on your objectives, the transfer function you choose might alter both the amplitude and the phase of the variations in the data at different frequencies to produce either a smoother or a rougher output.

In digital signal processing (DSP), it is customary to write transfer functions as rational expressions in z^{-1} and to order the numerator and denominator terms in ascending powers of z^{-1}.

Taking the z-transform of the difference equation

$$\begin{aligned} a(1)y(n) = \; & b(1)x(n) + b(2)x(n-1) + \ldots + b(nb)x(n-nb+1) \\ & - a(2)y(n-1) - \ldots - a(na)y(n-na+1) \end{aligned}$$

results in the transfer function

$$Y(z) = H(z^{-1})X(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(nb)z^{-nb+1}}{a(1) + a(2)z^{-1} + \ldots + a(na)z^{-na+1}}X(z)$$

where $Y(z)$ is the z-transform of the filtered output $y(n)$. The coefficients b and a are unchanged by the z-transform.

Examples

Consider the following transfer function:

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}$$

You will apply this transfer function to the data in `count.dat`.

1. Load the matrix `count` into the workspace.

```
load count.dat;
```

2. Create a time-series object based on this matrix.

```
count1=timeseries(count(:,1),[1:24]);
```

3 Enter the coefficients of the denominator ordered in ascending powers of z^{-1} to represent $1 + 0.2z^{-1}$.

```
a = [1 0.2];
```

4 Enter the coefficients of the numerator to represent $2 + 3z^{-1}$.

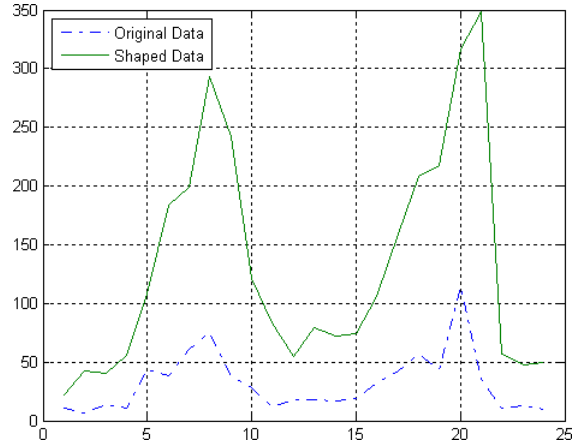
```
b = [2 3];
```

5 Call the filter function.

```
filter_count = filter(count1,b,a)
```

6 Compare the original data and the shaped data with an overlaid plot of the two curves:

```
plot(count1,'-.'), grid on, hold on
plot(filter_count,'-')
legend('Original Data','Shaped Data',2)
```



See Also

`idealfilter (timeseries)`, `timeseries`, `tsprops`

Purpose

2-D digital filter

Syntax

```
Y = filter2(h,X)
Y = filter2(h,X,shape)
```

Description

`Y = filter2(h,X)` filters the data in `X` with the two-dimensional FIR filter in the matrix `h`. It computes the result, `Y`, using two-dimensional correlation, and returns the central part of the correlation that is the same size as `X`.

`Y = filter2(h,X,shape)` returns the part of `Y` specified by the `shape` parameter. *`shape`* is a string with one of these values:

`'full'`	Returns the full two-dimensional correlation. In this case, `Y` is larger than `X`.
`'same'`	(default) Returns the central part of the correlation. In this case, `Y` is the same size as `X`.
`'valid'`	Returns only those parts of the correlation that are computed without zero-padded edges. In this case, `Y` is smaller than `X`.

Remarks

Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how `filter2` performs linear filtering.

Algorithm

Given a matrix `X` and a two-dimensional FIR filter `h`, `filter2` rotates your filter matrix 180 degrees to create a convolution kernel. It then calls `conv2`, the two-dimensional convolution function, to implement the filtering operation.

`filter2` uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` then extracts the central part of the convolution that is the same size as the input

matrix, and returns this as the result. If the `shape` parameter specifies an alternate part of the convolution for the result, `filter2` returns the appropriate part.

See Also

`conv2`, `filter`

find

Purpose

Find indices and values of nonzero elements

Syntax

```
ind = find(X)
ind = find(X, k)
ind = find(X, k, 'first')
ind = find(X, k, 'last')
[row,col] = find(X, ...)
[row,col,v] = find(X, ...)
```

Description

`ind = find(X)` locates all nonzero elements of array `X`, and returns the linear indices of those elements in vector `ind`. If `X` is a row vector, then `ind` is a row vector; otherwise, `ind` is a column vector. If `X` contains no nonzero elements or is an empty array, then `ind` is an empty array.

`ind = find(X, k)` or `ind = find(X, k, 'first')` returns at most the first `k` indices corresponding to the nonzero entries of `X`. `k` must be a positive integer, but it can be of any numeric data type.

`ind = find(X, k, 'last')` returns at most the last `k` indices corresponding to the nonzero entries of `X`.

`[row,col] = find(X, ...)` returns the row and column indices of the nonzero entries in the matrix `X`. This syntax is especially useful when working with sparse matrices. If `X` is an N-dimensional array with N > 2, `col` contains linear indices for the columns. For example, for a 5-by-7-by-3 array `X` with a nonzero element at `X(4,2,3)`, `find` returns 4 in `row` and 16 in `col`. That is, (7 columns in page 1) + (7 columns in page 2) + (2 columns in page 3) = 16.

`[row,col,v] = find(X, ...)` returns a column or row vector `v` of the nonzero entries in `X`, as well as row and column indices. If `X` is a logical expression, then `v` is a logical array. Output `v` contains the non-zero elements of the logical array obtained by evaluating the expression `X`. For example,

```
A= magic(4)
A =
    16     2     3    13
     5    11    10     8
```

```
     9     7     6    12
     4    14    15     1

[r,c,v]= find(A>10);

r', c', v'
ans =
     1     2     4     4     1     3
ans =
     1     2     2     3     4     4
ans =
     1     1     1     1     1     1
```

Here the returned vector `v` is a logical array that contains the nonzero elements of `N` where

```
N=(A>10)
```

Examples

Example 1

```
X = [1 0 4 -3 0 0 0 8 6];
indices = find(X)
```

returns linear indices for the nonzero entries of `X`.

```
indices =
     1     3     4     8     9
```

Example 2

You can use a logical expression to define `X`. For example,

```
find(X > 2)
```

returns linear indices corresponding to the entries of `X` that are greater than 2.

```
ans =
     3     8     9
```

Example 3

The following `find` command

```
X = [3 2 0; -5 0 7; 0 0 1];
[r,c,v] = find(X)
```

returns a vector of row indices of the nonzero entries of `X`

```
r =
     1
     2
     1
     2
     3
```

a vector of column indices of the nonzero entries of `X`

```
c =
     1
     1
     2
     3
     3
```

and a vector containing the nonzero entries of `X`.

```
v =
     3
    -5
     2
     7
     1
```

Example 4

The expression

```
[r,c,v] = find(X>2)
```

returns a vector of row indices of the nonzero entries of `X`

```
r =
     1
     2
```

a vector of column indices of the nonzero entries of `X`

```
c =
     1
     3
```

and a logical array that contains the non zero elements of `N` where `N=(X>2)`.

```
v =
     1
     1
```

Recall that when you use `find` on a logical expression, the output vector `v` does not contain the nonzero entries of the input array. Instead, it contains the nonzero values returned after evaluating the logical expression.

Example 5

Some operations on a vector

```
x = [11  0  33  0  55]';

find(x)
ans =
     1
     3
     5

find(x == 0)
ans =
     2
```

```
     4

find(0 < x & x < 10*pi)
ans =
     1
```

Example 6

For the matrix

```
M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2

find(M > 3, 4)
```

returns the indices of the first four entries of `M` that are greater than 3.

```
ans =
     1
     3
     5
     6
```

Example 7

If `X` is a vector of all zeros, `find(X)` returns an empty matrix. For example,

```
indices = find([0;0;0])
indices =
   Empty matrix: 0-by-1
```

See Also

`nonzeros`, `sparse`, `colon`, logical operators (elementwise and short-circuit), relational operators, `ind2sub`

Purpose

Find all graphics objects

Syntax

```
object_handles = findall(handle_list)
object_handles = findall(handle_list,'property','value',...)
```

Description

`object_handles = findall(handle_list)` returns the handles, including hidden handles, of all objects in the hierarchy under the objects identified in `handle_list`.

`object_handles = findall(handle_list,'property','value',...)` returns the handles of all objects in the hierarchy under the objects identified in `handle_list` that have the specified properties set to the specified values.

Remarks

`findall` is similar to `findobj`, except that it finds objects even if their `HandleVisibility` is set to `off`.

Examples

```
plot(1:10)
xlabel xlab
a = findall(gcf)
b = findobj(gcf)
c = findall(b,'Type','text') % return the xlabel handle twice
d = findobj(b,'Type','text') % can't find the xlabel handle
```

See Also

`allchild`, `findobj`

findfigs

Purpose

Find visible offscreen figures

Syntax

```
findfigs
```

Description

`findfigs` finds all visible figure windows whose display area is off the screen and positions them on the screen.

A window appears to MATLAB to be offscreen when its display area (the area not covered by the window's title bar, menu bar, and toolbar) does not appear on the screen.

This function is useful when you are bringing an application from a larger monitor to a smaller one (or one with lower resolution). Windows visible on the larger monitor may appear offscreen on a smaller monitor. Using `findfigs` ensures that all windows appear on the screen.

See Also

"Finding and Identifying Graphics Objects" on page 1-93 for related functions.

Purpose

Locate graphics objects with specific properties

Syntax

```
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h =
findobj('PropertyName',PropertyValue,'-logicaloperator',
   PropertyName',PropertyValue,...)
h = findobj('-regexp','PropertyName','regexp',...)
h = findobj('-property','PropertyName')
h = findobj(objhandles,...)
h = findobj(objhandles,'-depth',d,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,
   ...)
```

Description

`findobj` locates graphics objects and returns their handles. You can limit the search to objects with particular property values and along specific branches of the hierarchy.

`h = findobj` returns the handles of the root object and all its descendants.

`h = findobj('PropertyName',PropertyValue,...)` returns the handles of all graphics objects having the property *`PropertyName`*, set to the value `PropertyValue`. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h = findobj('PropertyName',PropertyValue,'-logicaloperator', PropertyName',PropertyValue,...)` applies the logical operator to the property value matching. Possible values for *`-logicaloperator`* are:

- `-and`
- `-or`
- `-xor`
- `-not`

See the Examples section for examples of how to use these operators. See “Logical Operators” for an explanation of logical operators.

`h = findobj('-regexp','PropertyName','regexp',...)` matches objects using regular expressions as if the value of the property `PropertyName` was passed to the `regexp` function as

```
regexp(PropertyValue,'regexp')
```

If a match occurs, `findobj` returns the object's handle. See the `regexp` function for information on how MATLAB uses regular expressions.

`h = findobj('-property','PropertyName')` finds all objects having the specified property.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendants.

`h = findobj(objhandles,'-depth',d,...)` specified the depth of the search. The depth argument `d` controls how many levels under the handles in `objhandles` are traversed. Specifying `d` as `inf` to get the default behavior of all levels. Specify `d` as `0` to get the same behavior as using the `flat` argument.

`h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendants.

Remarks

`findobj` returns an error if a handle refers to a nonexistent graphics object.

`findobj` correctly matches any legal property value. For example,

```
findobj('Color','r')
```

finds all objects having a `Color` property set to `red`, `r`, or `[1 0 0]`.

When a graphics object is a descendant of more than one object identified in `objhandles`, MATLAB searches the object each time

findobj encounters its handle. Therefore, implicit references to a graphics object can result in its handle being returned multiple times.

Examples

Find all line objects in the current axes:

```
h = findobj(gca,'Type','line')
```

Find all objects having a Label set to 'foo' and a String set to 'bar':

```
h = findobj('Label','foo','-and','String','bar');
```

Find all objects whose String is not 'foo' and is not 'bar':

```
h = findobj('-not','String','foo','-not','String','bar');
```

Find all objects having a String set to 'foo' and a Tag set to 'button one' and whose Color is not 'red' or 'blue':

```
h = findobj('String','foo','-and','Tag','button one',...
 '-and','-not',{'Color','red','-or','Color','blue'})
```

Find all objects for which you have assigned a value to the Tag property (that is, the value is not the empty string ''):

```
h = findobj('-regexp','Tag','[^'']')
```

Find all children of the current figure that have their BackgroundColor property set to a certain shade of gray ([.7 .7 .7]). Note that this statement also searches the current figure for the matching property value pair.

```
h = findobj(gcf,'-depth',1,'BackgroundColor',[.7 .7 .7])
```

See Also

copyobj, gcf, gca, gcbo, gco, get, regexp, set

See "Example — Using Logical Operators and Regular Expression" for more examples.

"Finding and Identifying Graphics Objects" on page 1-93 for related functions

findstr

Purpose

Find string within another, longer string

Syntax

```
k = findstr(str1, str2)
```

Description

`k = findstr(str1, str2)` searches the longer of the two input strings for any occurrences of the shorter string, returning the starting index of each such occurrence in the double array `k`. If no occurrences are found, then `findstr` returns the empty array, `[]`.

The search performed by `findstr` is case sensitive. Any leading and trailing blanks in either input string are explicitly included in the comparison.

Unlike the `strfind` function, the order of the input arguments to `findstr` is not important. This can be useful if you are not certain which of the two input strings is the longer one.

Examples

```
s = 'Find the starting indices of the shorter string.';

findstr(s, 'the')
ans =
     6    30

findstr('the', s)
ans =
     6    30
```

See Also

`strfind`, `strmatch`, `strtok`, `strcmp`, `strncmp`, `strcmpi`, `strncmpi`, `regexp`, `regexpi`, `regexprep`

Purpose

MATLAB termination M-file

Description

When MATLAB quits, it runs a script called `finish.m`, if the script exists and is on the MATLAB search path or in the current directory. This is a file you create yourself that instructs MATLAB to perform any final tasks just prior to terminating. For example, you might want to save the data in your workspace to a MAT-file before MATLAB exits.

`finish.m` is invoked whenever you do one of the following:

- Click the Close box  in the MATLAB desktop on Windows or the UNIX equivalent
- Select **Exit MATLAB** from the desktop **File** menu
- Type `quit` or `exit` at the Command Window prompt

Remarks

When using Handle Graphics in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

Examples

Two sample `finish.m` files are provided with MATLAB in *`matlabroot`*`/toolbox/local`. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` and add it to the path to use it:

- `finishsav.m` — Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m` — Displays a dialog allowing you to cancel quitting and saves the workspace. See also the MATLAB general preference for confirmation dialogs for quitting.

See Also

`quit`, `exit`, `startup`

"Quitting MATLAB" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Information about FITS file

Syntax

```
info = fitsinfo(filename)
```

Description

`info = fitsinfo(filename)` returns the structure, `info`, with fields that contain information about the contents of a Flexible Image Transport System (FITS) file. `filename` is a string enclosed in single quotes that specifies the name of the FITS file.

The `info` structure contains the following fields, listed in the order they appear in the structure. In addition, the info structure can also contain information about any number of optional file components, called *extensions* in FITS terminology. For more information, see "FITS File Extensions" on page 2-1227.

Field Name	Description	Return Type
`Filename`	Name of the file	String
`FileModDate`	File modification date	String
`FileSize`	Size of the file in bytes	Double
`Contents`	List of extensions in the file in the order that they occur	Cell array of strings
`PrimaryData`	Information about the primary data in the FITS file	Structure array

PrimaryData

The `PrimaryData` field is a structure that describes the primary data in the file. The following table lists the fields in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Array containing the size of each dimension	Double array

Field Name	Description	Return Type
`DataSize`	Size of the primary data in bytes	Double
`MissingDataValue`	Value used to represent undefined data	Double
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Offset`	Number of bytes from beginning of the file to the location of the first data value	Double
`Keywords`	A number-of-keywords-by-3 cell array containing keywords, values, and comments of the header in each column	Cell array of strings

FITS File Extensions

A FITS file can also include optional extensions. If the file contains any of these extensions, the `info` structure can contain these additional fields.

- AsciiTable — Numeric information in tabular format, stored as ASCII characters

- BinaryTable — Numeric information in tabular format, stored in binary representation
- Image — A multidimensional array of pixels
- Unknown — Nonstandard extension

AsciiTable Extension

The `AsciiTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	Double
`RowSize`	Number of characters in each row	Double
`NFields`	Number of fields in each row	Double array
`FieldFormat`	A 1-by-`NFields` cell containing formats in which each field is encoded. The formats are FORTRAN-77 format codes.	Cell array of strings
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of strings
`FieldWidth`	A 1-by-`NFields` array containing the number of characters in each field	Double array
`FieldPos`	A 1-by-`NFields` array of numbers representing the starting column for each field	Double array
`DataSize`	Size of the data in the table in bytes	Double
`MissingDataValue`	A 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of strings

Field Name	Description	Return Type
`Intercept`	A 1-by-`NFields` array of numbers used along with `Slope` to calculate actual data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	Double array
`Slope`	A 1-by-`NFields` array of numbers used with `Intercept` to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	Double array
`Offset`	Number of bytes from beginning of the file to the location of the first data value in the table	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, Values and Comments in the ASCII table header	Cell array of strings

BinaryTable Extension

The `BinaryTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	Double
`RowSize`	Number of bytes in each row	Double
`NFields`	Number of fields in each row	Double

Field Name	Description	Return Type
`FieldFormat`	A 1-by-`NFields` cell array containing the data type of the data in each field. The data type is represented by a FITS binary table format code.	Cell array of strings
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of strings
`FieldSize`	A 1-by-`NFields` array, where each element contains the number of values in the Nth field	Double array
`DataSize`	Size of the data in the Binary Table, in bytes. Includes any data past the main table.	Double
`MissingDataValue`	An 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of double
`Intercept`	A 1-by-`NFields` array of numbers used along with Slope to calculate actual data values from the array data values using the equation: `actual_value = slope*array_value+Intercept`	Double array
`Slope`	A 1-by-`NFields` array of numbers used with Intercept to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	Double array

Field Name	Description	Return Type
`Offset`	Number of bytes from beginning of the file to the location of the first data value	Double
`ExtensionSize`	Size of any data past the main table, in bytes	Double
`ExtensionOffset`	Number of bytes from the beginning of the file to any data past the main table	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Image Extension

The `Image` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Array containing sizes of each dimension	Double array
`DataSize`	Size of the data in the Image extension in bytes	Double
`Offset`	Number of bytes from the beginning of the file to the first data value	Double
`MissingDataValue`	Value used to represent undefined data	Double

Field Name	Description	Return Type
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Unknown Structure

The `Unknown` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Sizes of each dimension	Double array
`DataSize`	Size of the data in nonstandard extensions, in bytes	Double
`Offset`	Number of bytes from beginning of the file to the first data value	Double

Field Name	Description	Return Type
`MissingDataValue`	Representation of undefined data	Double
`Intercept`	Value, used with `Slope`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Example

Use `fitsinfo` to obtain information about the FITS file `tst0012.fits`. In addition to its primary data, the file also contains an example of the extensions `BinaryTable`, `Unknown`, `Image`, and `AsciiTable`.

```
S = fitsinfo('tst0012.fits');
S =
          Filename: [1x71 char]
       FileModDate: '12-Mar-2001 18:37:46'
          FileSize: 109440
          Contents: {'Primary' 'Binary Table' 'Unknown'
'Image' 'ASCII Table'}
       PrimaryData: [1x1 struct]
       BinaryTable: [1x1 struct]
```

```
           Unknown: [1x1 struct]
             Image: [1x1 struct]
        AsciiTable: [1x1 struct]
```

The `PrimaryData` field describes the data in the file. For example, the `Size` field indicates the data is a 102-by-109 matrix.

```
S.PrimaryData
          DataType: 'single'
              Size: [102 109]
          DataSize: 44472
 MissingDataValue: []
         Intercept: 0
             Slope: 1
            Offset: 2880
          Keywords: {25x3 cell}
```

The `AsciiTable` field describes the `AsciiTable` extension. For example, using the `FieldWidth` and `FieldPos` fields you can determine the length and location of each field within a row.

```
S.AsciiTable
ans =
              Rows: 53
           RowSize: 59
           NFields: 8
       FieldFormat: {'A9' 'F6.2' 'I3' 'E10.4' 'D20.15' 'A5' 'A1' 'I4'}
    FieldPrecision: {1x8 cell}
        FieldWidth: [9 6.2000 3 10.4000 20.1500 5 1 4]
          FieldPos: [1 11 18 22 33 54 54 55]
          DataSize: 3127
 MissingDataValue: {'*' '---.--' '  *' [] '*' '*' '*' ''}
         Intercept: [0 0 -70.2000 0 0 0 0 0]
             Slope: [1 1 2.1000 1 1 1 1 1]
            Offset: 103680
          Keywords: {65x3 cell}
```

See Also

`fitsread`

Purpose

Read data from FITS file

Syntax

```
data = fitsread(filename)
data = fitsread(filename, extname)
data = fitsread(filename, extname, index)
data = fitsread(filename, 'raw')
```

Description

`data = fitsread(filename)` reads the primary data of the Flexible Image Transport System (FITS) file specified by `filename`. Undefined data values are replaced by `NaN`. Numeric data are scaled by the slope and intercept values and are always returned in double precision. The `filename` argument is a string enclosed in single quotes.

`data = fitsread(filename, extname)` reads data from a FITS file according to the data array or extension specified in `extname`. You can specify only one `extname`. The valid choices for `extname` are shown in the following table.

Data Arrays or Extensions

extname	Description
'primary'	Read data from the primary data array.
'table'	Read data from the ASCII Table extension.
'bintable'	Read data from the Binary Table extension.
'image'	Read data from the Image extension.
'unknown'	Read data from the Unknown extension.

`data = fitsread(filename, extname, index)` is the same as the above syntax, except that if there is more than one of the specified extension type `extname` in the file, then only the one at the specified `index` is read.

`data = fitsread(filename, 'raw')` reads the primary or extension data of the FITS file, but, unlike the above syntaxes, does not replace

undefined data values with `NaN` and does not scale the data. The `data` returned has the same class as the data stored in the file.

Example

Read FITS file `tst0012.fits` into a 109-by-102 matrix called `data`.

```
data = fitsread('tst0012.fits');

whos data
  Name       Size            Bytes  Class

  data     109x102           88944  double array
```

Here is the beginning of the data read from the file.

```
data(1:5,1:6)
ans =
   135.200  134.9436  134.1752  132.8980  131.1165  128.8378
   137.568  134.9436  134.1752  132.8989  131.1167  126.3343
  135.9946  134.9437  134.1752  132.8989  131.1185  128.1711
  134.0093  134.9440  134.1749  132.8983  131.1201  126.3349
  131.5855  134.9439  134.1749  132.8989  131.1204  126.3356
```

Read only the Binary Table extension from the file.

```
data = fitsread('tst0012.fits', 'bintable')

data =
  Columns 1 through 4
    {11x1 cell}  [11x1 int16]  [11x3 uint8]  [11x2 double]
  Columns 5 through 9
    [11x3 cell]  {11x1 cell}  [11x1 int8]  {11x1 cell}  [11x3 int32]
  Columns 10 through 13
    [11x2 int32]  [11x2 single]  [11x1 double]  [11x1 uint8]
```

See Also

`fitsinfo`

Purpose

Round toward zero

Syntax

```
B = fix(A)
```

Description

`B = fix(A)` rounds the elements of `A` toward zero, resulting in an array of integers. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000        -0.2000        3.4000        5.6000

  Columns 5 through 6
   7.0000        2.4000 + 3.6000i

fix(a)

ans =
  Columns 1 through 4
  -1.0000        0             3.0000        5.0000

  Columns 5 through 6
   7.0000        2.0000 + 3.0000i
```

See Also

`ceil`, `floor`, `round`

flipdim

Purpose

Flip array along specified dimension

Syntax

```
B = flipdim(A,dim)
```

Description

`B = flipdim(A,dim)` returns `A` with dimension `dim` flipped.

When the value of `dim` is `1`, the array is flipped row-wise down. When `dim` is `2`, the array is flipped columnwise left to right. `flipdim(A,1)` is the same as `flipud(A)`, and `flipdim(A,2)` is the same as `fliplr(A)`.

Examples

`flipdim(A,1)` where

```
A =

     1     4
     2     5
     3     6
```

produces

```
     3     6
     2     5
     1     4
```

See Also

`fliplr`, `flipud`, `permute`, `rot90`

Purpose

Flip matrix left to right

Syntax

```
B = fliplr(A)
```

Description

`B = fliplr(A)` returns `A` with columns flipped in the left-right direction, that is, about a vertical axis.

If `A` is a row vector, then `fliplr(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a column vector, then `fliplr(A)` simply returns `A`.

Examples

If `A` is the 3-by-2 matrix,

```
A =
     1     4
     2     5
     3     6
```

`then fliplr(A)` produces

```
     4     1
     5     2
     6     3
```

If `A` is a row vector,

```
A =
     1     3     5     7     9
```

then `fliplr(A)` produces

```
     9     7     5     3     1
```

Limitations

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

See Also

`flipdim`, `flipud`, `rot90`

flipud

Purpose

Flip matrix up to down

Syntax

```
B = flipud(A)
```

Description

`B = flipud(A)` returns `A` with rows flipped in the up-down direction, that is, about a horizontal axis.

If `A` is a column vector, then `flipud(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a row vector, then `flipud(A)` simply returns `A`.

Examples

If `A` is the 3-by-2 matrix,

```
A =
     1     4
     2     5
     3     6
```

`then flipud(A)` produces

```
     3     6
     2     5
     1     4
```

If `A` is a column vector,

```
A =
     3
     5
     7
```

then `flipud(A)` produces

```
A =
     7
     5
     3
```

Limitations

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

See Also

`flipdim`, `fliplr`, `rot90`

floor

Purpose

Round toward minus infinity

Syntax

```
B = floor(A)
```

Description

`B = floor(A)` rounds the elements of `A` to the nearest integers less than or equal to `A`. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000            -0.2000             3.4000             5.6000

  Columns 5 through 6
   7.0000             2.4000 + 3.6000i

floor(a)

ans =
  Columns 1 through 4
  -2.0000            -1.0000             3.0000             5.0000

  Columns 5 through 6
   7.0000             2.0000 + 3.0000i
```

See Also

`ceil`, `fix`, `round`

Purpose

Count floating-point operations

Description

This is an obsolete function. With the incorporation of LAPACK in MATLAB version 6, counting floating-point operations is no longer practical.

flow

Purpose

Simple function of three variables

Syntax

```
v = flow
v = flow(n)
v = flow(x,y,z)
[x,y,z,v] = flow(...)
```

Description

`flow`, a function of three variables, generates fluid-flow data that is useful for demonstrating `slice`, `interp3`, and other functions that visualize scalar volume data.

`v = flow` produces a 50-by-25-by-25 array.

`v = flow(n)` produces a 2`n`-by-`n`-by-`n` array.

`v = flow(x,y,z)` evaluates the speed profile at the points `x`, `y`, and `z`.

`[x,y,z,v] = flow(...)` returns the coordinates as well as the volume data.

See Also

`slice`, `interp3`

"Volume Visualization" on page 1-102 for related functions

See "Example — Slicing Fluid Flow Data" for an example that uses `flow`.

Purpose

Find minimum of single-variable function on fixed interval

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
[x,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

Description

`fminbnd` finds the minimum of a function of one variable within a fixed interval.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the function that is described in `fun` in the interval `x1 < x < x2`. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminbnd` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.

`OutputFcn`	User-defined function that is called at each iteration. See "Output Function" in the Optimization Toolbox for more information.
`PlotFcns`	User-defined plot function that is called at each iteration. See "Plot Functions" in the Optimization Toolbox for more information.
`TolX`	Termination tolerance on `x`.

`[x,fval] = fminbnd(...)` returns the value of the objective function computed in `fun` at `x`.

`[x,fval,exitflag] = fminbnd(...)` returns a value `exitflag` that describes the exit condition of `fminbnd`:

`1`	`fminbnd` converged to a solution `x` based on `options.TolX`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.
`-2`	Bounds are inconsistent (`x1 > x2`).

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization:

`output.algorithm`	Algorithm used
`output.funcCount`	Number of function evaluations
`output.iterations`	Number of iterations
`output.message`	Exit message

Arguments

`fun` is the function to be minimized. `fun` accepts a scalar `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminbnd(@myfun,x1,x2);
```

where `myfun.m` is an M-file function such as

```
function f = myfun(x)
f = ...         % Compute function value at x.
```

or as a function handle for an anonymous function:

```
x = fminbnd(@(x) sin(x*x),x1,x2);
```

Other arguments are described in the syntax descriptions above.

Examples

`x = fminbnd(@cos,3,4)` computes π to a few decimal places and gives a message on termination.

```
[x,fval,exitflag] = ...
   fminbnd(@cos,3,4,optimset('TolX',1e-12,'Display','off'))
```

computes π to about 12 decimal places, suppresses output, returns the function value at `x`, and returns an `exitflag` of 1.

The argument `fun` can also be a function handle for an anonymous function. For example, to find the minimum of the function $f(x) = x^3 - 2x - 5$ on the interval `(0,2)`, create an anonymous function `f`

```
f = @(x)x.^3-2*x-5;
```

Then invoke `fminbnd` with

```
x = fminbnd(f, 0, 2)
```

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)

y =
    -6.0887
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminbind`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminbnd` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminbnd(@(x) myfun(x,a),0,1)
   ```

Algorithm

`fminbnd` is an M-file. The algorithm is based on golden section search and parabolic interpolation. Unless the left endpoint x_1 is very close to the right endpoint x_2, `fminbnd` never evaluates `fun` at the endpoints, so `fun` need only be defined for x in the interval $x_1 < x < x_2$. If the minimum actually occurs at x_1 or x_2, `fminbnd` returns an interior point at a distance of no more than `2*TolX` from x_1 or x_2, where `TolX` is the termination tolerance. See [1] or [2] for details about the algorithm.

Limitations

The function to be minimized must be continuous. `fminbnd` may only give local solutions.

`fminbnd` often exhibits slow convergence when the solution is on a boundary of the interval.

`fminbnd` only handles real variables.

See Also

`fminsearch`, `fzero`, `optimset`, `function_handle` (`@`), anonymous function

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

[2] Brent, Richard. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973

Purpose

Find minimum of unconstrained multivariable function using derivative-free method

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

Description

`fminsearch` finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminsearch(fun,x0)` starts at the point `x0` and finds a local minimum `x` of the function described in `fun`. `x0` can be a scalar, vector, or matrix. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`. See also "Example 2" on page 2-1351 and "Example 3" on page 2-1351 below.

`x = fminsearch(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminsearch` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex`, `Inf` or `NaN`. `'off'` (the default) displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed

`MaxIter`	Maximum number of iterations allowed
`OutputFcn`	User-defined function that is called at each iteration. See "Output Function" in the Optimization Toolbox for more information.
`PlotFcns`	User-defined plot function that is called at each iteration. See "Plot Functions" in the Optimization Toolbox for more information.
`TolFun`	Termination tolerance on the function value
`TolX`	Termination tolerance on `x`

`[x,fval] = fminsearch(...)` returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch(...)` returns a value `exitflag` that describes the exit condition of `fminsearch`:

`1`	`fminsearch` converged to a solution `x`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.

`[x,fval,exitflag,output] = fminsearch(...)` returns a structure `output` that contains information about the optimization:

`output.algorithm`	Algorithm used
`output.funcCount`	Number of function evaluations
`output.iterations`	Number of iterations
`output.message`	Exit message

Arguments

`fun` is the function to be minimized. It accepts an input `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fminsearch(@myfun, x0)
```

where `myfun` is an M-file function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

or as a function handle for an anonymous function, such as

```
x = fminsearch(@(x)sin(x^2), x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Example 1

A classic test example for multidimensional minimization is the Rosenbrock banana function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

The minimum is at `(1,1)` and has the value `0`. The traditional starting point is `(-1.2,1)`. The anonymous function shown here defines the function and returns a function handle called `banana`:

```
banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to `fminsearch`:

```
[x,fval] = fminsearch(banana,[-1.2, 1])
```

This produces

```
x =

    1.0000    1.0000

fval =

    8.1777e-010
```

This indicates that the minimizer was found to at least four decimal places with a value near zero.

Example 2

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = x(1)^2 + a*x(2)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminsearch`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminsearch` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminsearch(@(x) myfun(x,a),[0,1])
   ```

Example 3

You can modify the first example by adding a parameter a to the second term of the banana function:

$$f(x) = 100(x_2 - x_1^2)^2 + (a - x_1)^2$$

This changes the location of the minimum to the point `[a,a^2]`. To minimize this function for a specific value of `a`, for example `a = sqrt(2)`, create a one-argument anonymous function that captures the value of `a`.

```
a = sqrt(2);
banana = @(x)100*(x(2)-x(1)^2)^2+(a-x(1))^2;
```

Then the statement

```
[x,fval] = fminsearch(banana, [-1.2, 1], ...
   optimset('TolX',1e-8));
```

seeks the minimum `[sqrt(2), 2]` to an accuracy higher than the default on `x`.

Algorithm

fminsearch uses the simplex search method of [1]. This is a direct search method that does not use numerical or analytic gradients.

If `n` is the length of `x`, a simplex in `n`-dimensional space is characterized by the `n+1` distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

Limitations

`fminsearch` can often handle discontinuity, particularly if it does not occur near the solution. fminsearch may only give local solutions.

`fminsearch` only minimizes over the real numbers, that is, x must only consist of real numbers and $f(x)$ must only return real numbers. When x has complex variables, they must be split into real and imaginary parts.

See Also

`fminbnd`, `optimset`, `function_handle` (@), anonymous function

References

[1] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, Vol. 9 Number 1, pp. 112-147, 1998.

Purpose

Open file, or obtain information about open files

Syntax

```
fid = fopen(filename)
fid = fopen(filename, permission)
fid = fopen(filename, permission_tmode)
[fid, message] = fopen(filename, permission)
[fid, message] = fopen(filename, permission, machineformat)
[fid, message] = fopen(filename, permission, machineformat,
   encoding)
fids = fopen('all')
[filename, permission, machineformat, encoding] = fopen(fid)
```

Description

`fid = fopen(filename)` opens the file `filename` for read access. (On Windows systems, `fopen` opens files for binary read access.) The `filename` argument is a string enclosed in single quotes. It can be a `MATLABPATH` relative partial pathname if the file is opened for reading only. A relative path is always searched for first with respect to the current directory. If it is not found, and reading only is specified or implied, then `fopen` does an additional search of the `MATLABPATH`.

`fid` is a scalar MATLAB integer, called a file identifier. You use the `fid` as the first argument to other file input/output routines. If `fopen` cannot open the file, it returns `-1`. Two file identifiers are automatically available and need not be opened. They are `fid=1` (standard output) and `fid=2` (standard error).

`fid = fopen(filename, permission)` opens the file `filename` in the specified `permission`. The `permission` argument can be any of the following:

Permission Specifiers

Permission	Description
'r'	Open file for reading (default).
'w'	Open file, or create new file, for writing; discard existing contents, if any.

Permission Specifiers (Continued)

Permission	Description
'a'	Open file, or create new file, for writing; append data to the end of the file.
'r+'	Open file for reading and writing.
'w+'	Open file, or create new file, for reading and writing; discard existing contents, if any.
'a+'	Open file, or create new file, for reading and writing; append data to the end of the file.
'A'	Append without automatic flushing; used with tape drives.
'W'	Write without automatic flushing; used with tape drives.

Note If the file is opened in update mode (`'+'`), an input command like `fread`, `fscanf`, `fgets`, or `fgetl` cannot be immediately followed by an output command like `fwrite` or `fprintf` without an intervening `fseek` or `frewind`. The reverse is also true: that is, an output command like `fwrite` or `fprintf` cannot be immediately followed by an input command like `fread`, `fscanf`, `fgets`, or `fgetl` without an intervening `fseek` or `frewind`.

`fid = fopen(filename, permission_tmode)` on Windows systems, opens the file in text mode instead of binary mode (the default). The `permission_tmode` argument consists of any of the specifiers shown in the Permission Specifiers on page 2-1255 table above, followed by the letter `t`, for example `'rt'` or `'wt+`. On UNIX, text and binary mode are the same.

Binary and Text Modes

Mode	Behavior
Binary	No characters are given special treatment.
Text	On a read operation, whenever MATLAB encounters a carriage return followed by a newline character, it removes the carriage return from the input. On a write or append operation, MATLAB inserts a carriage return before any newline character.

`[fid, message] = fopen(filename, permission)` opens a file as above. If it cannot open the file, `fid` equals `-1` and `message` contains a system-dependent error message. If `fopen` successfully opens a file, the value of `message` is empty.

`[fid, message] = fopen(filename, permission, machineformat)` opens the file with the specified `permission` and treats data read using `fread` or data written using `fwrite` as having a format given by `machineformat`. `machineformat` is one of the following strings:

Full Precision Support

`'ieee be'` or `'b'`	IEEE floating point with big-endian byte ordering
`'ieee le'` or `'l'`	IEEE floating point with little-endian byte ordering
`'ieee-be.l64'` or `'s'`	IEEE floating point with big-endian byte ordering and 64-bit long data type
`'ieee-le.l64'` or `'a'`	IEEE floating point with little-endian byte ordering and 64-bit long data type
`'native'` or `'n'`	Numeric format of the machine on which MATLAB is running (the default)

Full Precision Support (Continued)

'vaxd' or 'd'	VAX D floating point and VAX ordering
'vaxg' or 'g'	VAX G floating point and VAX ordering

Limited Precision Support: (double or equivalent)

'cray' or 'c'	Cray floating point with big-endian byte ordering

`[fid, message] = fopen(filename, permission, machineformat, encoding)` opens the specified file using the specified `permission` and `machineformat`. `encoding` is a string that specifies the character encoding scheme associated with the file. It must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. For common names and aliases, see the Web site `http://www.iana.org/assignments/character-sets`. If `encoding` is unspecified or is the empty string (`''`), MATLAB's default encoding scheme is used.

`fids = fopen('all')` returns a row vector containing the file identifiers of all open files, not including `1` and `2` (standard output and standard error). The number of elements in the vector is equal to the number of open files.

`[filename, permission, machineformat, encoding] = fopen(fid)` returns the `filename`, `permission`, `machineformat`, and `encoding` values used by MATLAB when it opened the file associated with identifier `fid`. MATLAB does not determine these output values by reading information from the opened file. For any of these parameters that were not specified when the file was opened, MATLAB returns its default value. The `encoding` string is a standard character encoding scheme name that may not be the same as the `encoding` argument used in the call to `fopen` that opened the file. An invalid `fid` returns empty strings for all output arguments.

The 'W' and 'A' modes do not automatically perform a flush of the current output buffer after output operations.

Examples

The example uses fopen to open a file and then passes the fid returned by fopen to other file I/O functions to read data from the file and then close the file.

```
fid = fopen('fgetl.m');
while 1
    tline = fgetl(fid);
    if ~ischar(tline),   break,   end
    disp(tline)
end
fclose(fid);
```

See Also

fclose, ferror, fprintf, fread, fscanf, fseek, ftell, fwrite

fopen (serial)

Purpose

Connect serial port object to device

Syntax

```
fopen(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`fopen(obj)` connects `obj` to the device.

Remarks

Before you can perform a read or write operation, `obj` must be connected to the device with the `fopen` function. When `obj` is connected to the device:

- Data remaining in the input buffer or the output buffer is flushed.
- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the device. You can connect only one serial port object to a given device.

Some properties are read-only while the serial port object is open (connected), and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the device. If any of these properties are incorrectly configured, then an error is returned when `fopen` is issued and `obj` is not connected to the device. Properties of this type include `BaudRate`, and are associated with device settings.

If you use the `help` command to display help for `fopen`, then you need to supply the pathname shown below.

```
help serial/fopen
```

Example

This example creates the serial port object `s`, connects `s` to the device using `fopen`, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
```

See Also

Functions

`fclose`

Properties

`BytesAvailable`, `BytesToOutput`, `Status`, `ValuesReceived`, `ValuesSent`

Purpose

Execute block of code specified number of times

Syntax

```
for x=initval:endval, statements, end
for x=initval:stepval:endval, statements, end
```

Description

`for x=initval:endval, statements, end` repeatedly executes one or more MATLAB *`statements`* in a loop. Loop counter variable `x` is initialized to value `initval` at the start of the first pass through the loop, and automatically increments by 1 each time through the loop. The program makes repeated passes through *`statements`* until either `x` has incremented to the value *`endval`*, or MATLAB encounters a `break`, or `return` instruction, thus forcing an immediately exit of the loop. If MATLAB encounters a `continue` statement in the loop code, it immediately exits the current pass at the location of the `continue` statement, skipping any remaining code in that pass, and begins another pass at the start of the loop *`statements`* with the value of the loop counter incremented by 1.

The values `initval` and `endval` must be real numbers or arrays of real numbers, or can also be calls to functions that return the same. The value assigned to `x` is often used in the code within the loop, however it is recommended that you do not assign to `x` in the loop code.

`for x=initval:stepval:endval, statements, end` is the same as the above syntax, except that loop counter `x` is incremented (or decremented when *`stepval`* is negative) by the value *`stepval`* on each iteration through the loop. The value `stepval` must be a real number or can also be a call to a function that returns a real number.

The general format is

```
for variable = initval:endval
    statement
    ...
    statement
end
```

The scope of the `for` statement is always terminated with a matching `end`.

See "Program Control Statements" in the MATLAB Programming documentation for more information on controlling the flow of your program code.

Remarks

It is recommended that you do not assign to the loop control variable while in the body of a loop. If you do assign to a variable that has the same name as the loop control variable (see `k` in the example below), then the value of that variable alternates between the value assigned by the `for` statement at the start of each loop iteration and the value explicitly assigned to it in the loop code:

```
for k=1:2
   disp(sprintf('   At the start of the loop, k = %d', k))
   k = 10;
   disp(sprintf('   Following the assignment, k = %d\n', k))
end

   At the start of the loop, k = 1
   Following the assignment, k = 10

   At the start of the loop, k = 2
   Following the assignment, k = 10
```

Examples

Assume `k` has already been assigned a value. Create the Hilbert matrix, using `zeros` to preallocate the matrix to conserve memory:

```
a = zeros(k,k)  % Preallocate matrix
for m = 1:k
    for n = 1:k
        a(m,n) = 1/(m+n -1);
    end
end
```

Step `s` with increments of `-0.1`:

```
for s = 1.0: -0.1: 0.0,..., end
```

Step `s` with values `1`, `5`, `8`, and `17`:

```
for s = [1,5,8,17], ..., end
```

Successively set `e` to the unit `n`-vectors:

```
for e = eye(n), ..., end
```

The line

```
for V = A, ..., end
```

has the same effect as

```
for k = 1:n, V = A(:,k); ..., end
```

except `k` is also set here.

See Also

`end`, `while`, `break`, `continue`, `return`, `if`, `switch`, `colon`

Purpose

Set display format for output

Graphical Interface

As an alternative to `format`, use preferences. Select **Preferences** from the **File** menu in the MATLAB desktop and use **Command Window** preferences.

Syntax

```
format
format type
format('type')
```

Description

Use the `format` function to control the output format of numeric values displayed in the Command Window.

Note The `format` function affects only how numbers are displayed, not how MATLAB computes or saves them.

`format` by itself, changes the output format to the default appropriate for the class of the variable currently being used. For floating-point variables, for example, the default is `format short` (i.e., 5-digit scaled, fixed-point values).

`format type` changes the format to the specified *`type`*. The tables shown below list the allowable values for *`type`*.

`format('type')` is the function form of the syntax.

The tables below show the allowable values for *`type`*, and provides an example for each type using `pi`.

Use these format types to switch between different output display formats for floating-point variables.

Type	Result
`short`	Scaled fixed point format, with 4 digits after the decimal point. For example, `3.1416`
`long`	Scaled fixed point format with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.14159265358979`
`short e`	Floating point format, with 4 digits after the decimal point. For example, `3.1416e+000`
`long e`	Floating point format, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.141592653589793e+000`
`short g`	Best of fixed or floating point, with 4 digits after the decimal point. For example, `3.1416`
`long g`	Best of fixed or floating point, with 14 to 15 digits after the decimal point for double; and 7 digits after the decimal point for single. For example, `3.14159265358979`
`short eng`	Engineering format that has 4 digits after the decimal point, and a power that is a multiple of three. For example, `3.1416e+000`
`long eng`	Engineering format that has exactly 16 significant digits and a power that is a multiple of three. For example, `3.14159265358979e+000`

Use these format types to switch between different output display formats for all numeric variables.

Value for type	Result
`+`	+, -, blank
`bank`	Fixed dollars and cents. For example, `3.14`

Value for type	Result
`hex`	Hexadecimal (hexadecimal representation of a binary double-precision number). For example, `400921fb54442d18`
`rat`	Ratio of small integers. For example, `355/113`

Use these format types to affect the spacing in the display of all variables.

Value for type	Result	Example
`compact`	Suppresses excess line feeds to show more output in a single screen. Contrast with `loose`.	`theta = pi/2 theta = 1.5708`
`loose`	Adds linefeeds to make output more readable. Contrast with `compact`.	`theta = pi/2 theta = 1.5708`

Remarks

Computations on floating-point variables, namely `single` or `double`, are done in appropriate floating-point precision, no matter how those variables are displayed. Computations on integer variables are done natively in integer.

MATLAB always displays integer variables to the appropriate number of digits for the class. For example, MATLAB uses three digits to display numbers of type `int8` (i.e., -128:127). Setting `format` to `short` or `long` does not affect the display of integer variables.

The specified format applies only to the current MATLAB session. To maintain a format across sessions, use MATLAB preferences.

To see which `type` is currently in use, type

```
get(0,'Format')
```

To see if `compact` or `loose` formatting is currently selected, type

```
get(0,'FormatSpacing').
```

Examples

Example 1

Change the format to `long` by typing

```
format long
```

View the result for the value of `pi` by typing

```
pi
ans =
   3.14159265358979
```

View the current format by typing

```
get(0,'format')
ans =
   long
```

Set the format to `short e` by typing

```
format short e
```

or use the function form of the syntax

```
format('short','e')
```

Example 2

When the format is set to `short`, both `pi` and `single(pi)` display as 5-digit values:

```
format short

pi
ans =
    3.1416

single(pi)
ans =
    3.1416
```

Now set format to `long`, and `pi` displays a 15-digit value while `single(pi)` display an 8-digit value:

```
format long

pi
ans =
    3.14159265358979

single(pi)
ans =
    3.1415927
```

Example 3

Set the format to its default, and display the maximum values for integers and real numbers in MATLAB:

```
format

intmax('uint64')
ans =
   18446744073709551615

realmax
ans =
  1.7977e+308
```

Now change the format to hexadecimal, and display these same values:

```
format hex

intmax('uint64')
ans =
   ffffffffffffffff

realmax
ans =
   7fefffffffffffff
```

The hexadecimal display corresponds to the internal representation of the value. It is not the same as the hexadecimal notation in the C programming language.

Example 4

This example illustrates the `short eng` and `long eng` formats. The value assigned to variable `A` increases by a multiple of 10 each time through the `for` loop.

```
A = 5.123456789;

for k=1:10
   disp(A)
   A = A * 10;
end
```

The values displayed for `A` are shown here. The power of 10 is always a multiple of 3. The value itself is expressed in 5 or more digits for the `short eng` format, and in exactly 15 digits for `long eng`:

```
   format short eng                format long eng

     5.1235e+000                  5.12345678900000e+000
    51.2346e+000                 51.2345678900000e+000
   512.3457e+000                512.345678900000e+000
     5.1235e+003                  5.12345678900000e+003
    51.2346e+003                 51.2345678900000e+003
   512.3457e+003                512.345678900000e+003
     5.1235e+006                  5.12345678900000e+006
    51.2346e+006                 51.2345678900000e+006
   512.3457e+006                512.345678900000e+006
     5.1235e+009                  5.12345678900000e+009
```

Algorithms

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats. The function `format +` displays +, -, and blank characters for positive, negative, and zero elements. `format hex` displays the hexadecimal

representation of a binary double-precision number. `format rat` uses a continued fraction algorithm to approximate floating-point values by ratios of small integers. See `rat.m` for the complete code.

See Also

`disp`, `display`, `isnumeric`, `isfloat`, `isinteger`, `floor`, `sprintf`, `fprintf`, `num2str`, `rat`, `spy`

Purpose

Plot function between specified limits

Syntax

```
fplot(fun,limits)
fplot(fun,limits,LineSpec)
fplot(fun,limits,tol)
fplot(fun,limits,tol,LineSpec)
fplot(fun,limits,n)
fplot(fun,lims,...)
fplot(axes_handle,...)
[X,Y] = fplot(fun,limits,...)
```

Description

`fplot` plots a function between specified limits. The function must be of the form $y = f(x)$, where x is a vector whose range specifies the limits, and `y` is a vector the same size as `x` and contains the function's value at the points in x (see the first example). If the function returns more than one value for a given x, then y is a matrix whose columns contain each component of $f(x)$ (see the second example).

`fplot(fun,limits)` plots `fun` between the limits specified by `limits`. `limits` is a vector specifying the x-axis limits (`[xmin xmax]`), or the x- and y-axes limits, (`[xmin xmax ymin ymax]`).

`fun` must be

- The name of an M-file function
- A string with variable `x` that may be passed to `eval`, such as `'sin(x)'`, `'diric(x,10)'`, or `'[sin(x),cos(x)]'`
- A function handle for an M-file function or an anonymous function (see "Function Handles" and "Anonymous Functions" for more information)

The function $f(x)$ must return a row vector for each element of vector x. For example, if $f(x)$ returns `[f1(x),f2(x),f3(x)]` then for input `[x1;x2]` the function should return the matrix

```
f1(x1) f2(x1) f3(x1)
f1(x2) f2(x2) f3(x2)
```

`fplot(fun,limits,LineSpec)` plots `fun` using the line specification `LineSpec`.

`fplot(fun,limits,tol)` plots `fun` using the relative error tolerance `tol` (the default is `2e-3`, i.e., 0.2 percent accuracy).

`fplot(fun,limits,tol,LineSpec)` plots `fun` using the relative error tolerance `tol` and a line specification that determines line type, marker symbol, and color. See `LineSpec` for more information.

`fplot(fun,limits,n)` with `n >= 1` plots the function with a minimum of `n+1` points. The default `n` is `1`. The maximum step size is restricted to be `(1/n)*(xmax-xmin)`.

`fplot(fun,lims,...)` accepts combinations of the optional arguments `tol`, `n`, and `LineSpec`, in any order.

`fplot(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`[X,Y] = fplot(fun,limits,...)` returns the abscissas and ordinates for `fun` in `X` and `Y`. No plot is drawn on the screen; however, you can plot the function using `plot(X,Y)`.

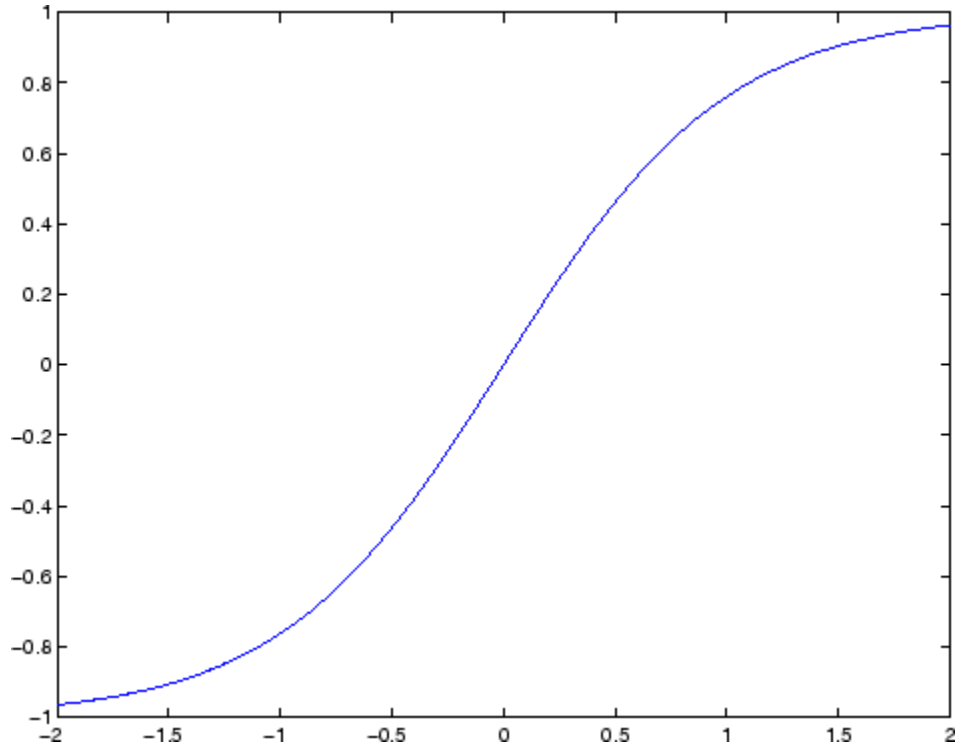
Remarks

`fplot` uses adaptive step control to produce a representative graph, concentrating its evaluation in regions where the function's rate of change is the greatest.

Examples

Plot the hyperbolic tangent function from -2 to 2:

```
fnch = @tanh;
fplot(fnch,[-2 2])
```



Create an M-file, `myfun`, that returns a two-column matrix:

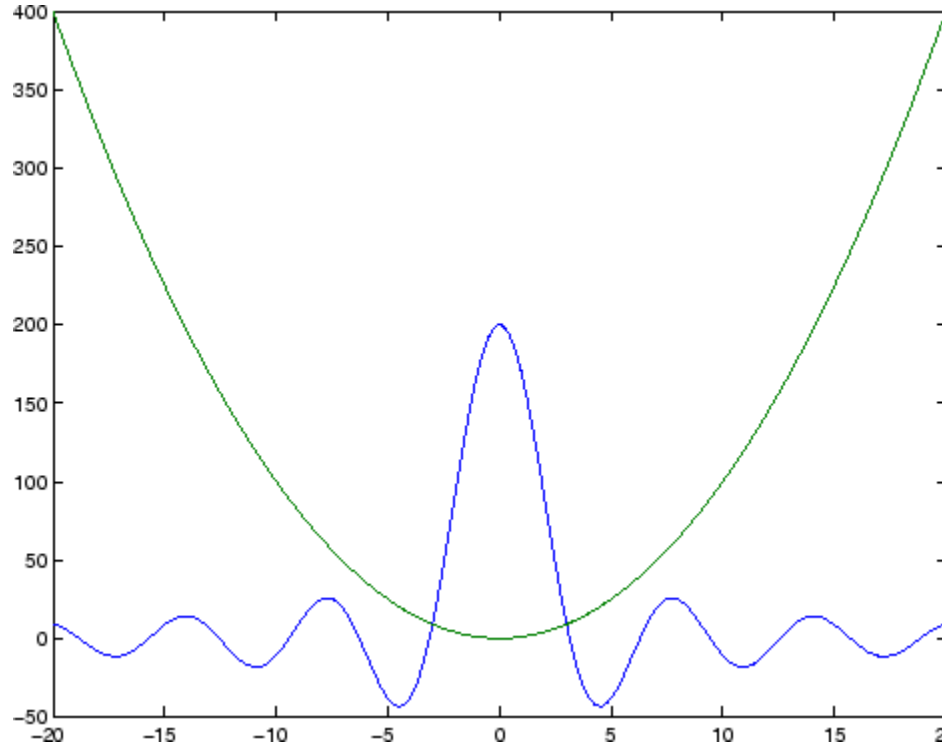
```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Create a function handle pointing to `myfun`:

```
fh = @myfun;
```

Plot the function with the statement

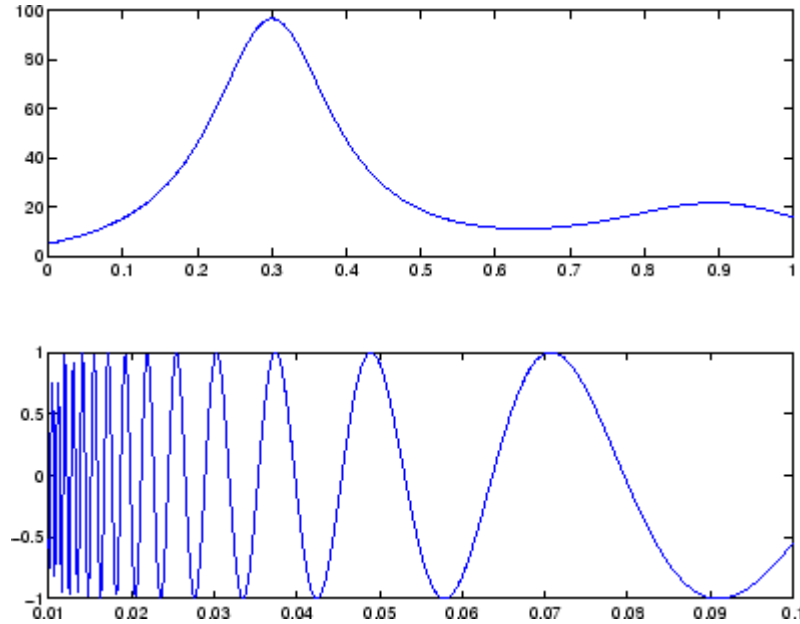
```
fplot(fh,[-20 20])
```



Additional Example

This example passes function handles to `fplot`, one created from a MATLAB function and the other created from an anonymous function.

```
hmp = @humps;
subplot(2,1,1);fplot(hmp,[0 1])
sn = @(x) sin(1./x);
subplot(2,1,2);fplot(sn,[.01 .1])
```



See Also

`eval`, `ezplot`, `feval`, `LineSpec`, `plot`

"Function Plots" on page 1-89 for related functions

"Plotting Mathematical Functions" for more examples

Purpose

Write formatted data to file

Syntax

```
count = fprintf(fid, format, A, ...)
```

Description

`count = fprintf(fid, format, A, ...)` formats the data in the real part of matrix `A` (and in any additional matrix arguments) under control of the specified `format` string, and writes it to the file associated with file identifier `fid`. `fprintf` returns a count of the number of bytes written.

Argument `fid` is an integer file identifier obtained from `fopen`. (It can also be `1` for standard output (the screen) or `2` for standard error. See `fopen` for more information.) Omitting `fid` causes output to appear on the screen.

See "Formatting Strings" in the MATLAB Programming documentation for more detailed information on using string formatting commands.

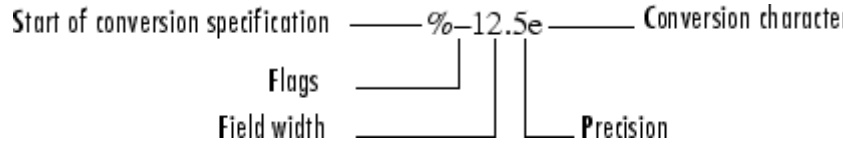
Format String

The `format` argument is a string containing ordinary characters and/or C language conversion specifications. A conversion specification controls the notation, alignment, significant digits, field width, and other aspects of output format. The format string can contain escape characters to represent nonprinting characters such as newline characters and tabs.

Conversion specifications begin with the `%` character and contain these optional and required elements:

- Flags (optional)
- Width and precision fields (optional)
- A subtype specifier (optional)
- Conversion character (required)

You specify these elements in the following order:



Flags

You can control the alignment of the output using any of these optional flags.

Character	Description	Example
Minus sign ( )	Left-justifies the converted argument in its field	`%-5.2d`
Plus sign (+)	Always prints a sign character (+ or)	`%+5.2d`
Space character	Inserts a space before the value	`% 5.2d`
Zero (0)	Pads with zeros rather than spaces	`%05.2d`

Field Width and Precision Specifications

You can control the width and precision of the output by including these options in the format string.

Character	Description	Example
Field width	A digit string specifying the minimum number of digits to be printed	`%6f`
Precision	A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point	`%6.2f`

Conversion Characters

Conversion characters specify the notation of the output.

Specifier	Description
%c	Single character
%d	Decimal notation (signed)
%e	Exponential notation (using a lowercase e as in 3.1415e+00)
%E	Exponential notation (using an uppercase E as in 3.1415E+00)
%f	Fixed-point notation
%g	The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.
%G	Same as %g, but using an uppercase E
%i	Decimal notation (signed)
%o	Octal notation (unsigned)
%s	String of characters
%u	Decimal notation (unsigned)
%x	Hexadecimal notation (using lowercase letters a–f)
%X	Hexadecimal notation (using uppercase letters A–F)

Conversion characters %o, %u, %x, and %X support subtype specifiers. See Remarks for more information.

Escape Characters

This table lists the escape character sequences you use to specify nonprinting characters in a format specification.

Character	Description
\b	Backspace
\f	Form feed
\n	New line

Character	Description
`\r`	Carriage return
`\t`	Horizontal tab
`\\`	Backslash
`\'' or ''` (two single quotes)	Single quotation mark
`%%`	Percent character

Remarks

When writing text to a file on Windows, it is recommended that you open the file in write-text mode (e.g., `fopen(file_id, 'wt')`). This ensures that lines in the file are terminated in such a way as to be compatible with all applications that might use the file.

MATLAB writes characters using the encoding scheme associated with the file. See `fopen` for more information.

The `fprintf` function behaves like its ANSI C language namesake with these exceptions and extensions:

- If you use `fprintf` to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the `fix`, `floor`, `ceil`, or `round` function to change the value in the double into a value that can be represented as an integer before passing it to `sprintf`.
- The following nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

`b` The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like `'%bx'`.

`t` The underlying C data type is a float rather than an unsigned integer.

For example, to print a double value in hexadecimal, use the format `'%bx'`.

- The `fprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of `A` (columnwise) until all the elements are used up. The function then continues in a similar manner through any additional matrix arguments.

Note `fprintf` displays negative zero (`-0`) differently on some platforms, as shown in the following table.

	Conversion Character		
Platform	**%e or %E**	**%f**	**%g or %G**
PC	0.000000e+000	0.000000	0
Others	-0.000000e+00	-0.000000	-0

Examples

Example 1

Create a text file called `exp.txt` containing a short table of the exponential function. (On Windows platforms, it is recommended that you use `fopen` with the mode set to `'wt'` to open a text file for writing.)

```
x = 0:.1:1;
y = [x; exp(x)];
fid = fopen('exp.txt', 'wt');
fprintf(fid, '%6.2f %12.8f\n', y);
```

```
fclose(fid)
```

Now examine the contents of `exp.txt`:

```
type exp.txt
   0.00    1.00000000
   0.10    1.10517092
       ...
   1.00    2.71828183
```

Example 2

The command

```
fprintf( ...
  'A unit circle has circumference %g radians.\n', 2*pi)
```

displays a line on the screen:

```
A unit circle has circumference 6.283186 radians.
```

Example 3

To insert a single quotation mark in a string, use two single quotation marks together. For example,

```
fprintf(1,'It''s Friday.\n')
```

displays on the screen

```
It's Friday.
```

Example 4

Use `fprintf` to display a hyperlink on the screen. For example,

```
site = '"http://www.mathworks.com"';
title = 'The MathWorks Web Site';
fprintf(['<a href = ' site '>' title '</a>'])
```

creates the hyperlink

```
The Mathworks Web Site
```

in the Command Window. Click on this link to display The MathWorks home page in a MATLAB Web browser.

Example 5

The commands

```
B = [8.8  7.7; 8800  7700]
fprintf(1, 'X is %6.2f meters or %8.3f mm\n', 9.9, 9900, B)
```

display the lines

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

Example 6

Explicitly convert MATLAB double-precision variables to integer values for use with an integer conversion specifier. For instance, to convert signed 32-bit data to hexadecimal format,

```
a = [6 10 14 44];
fprintf('%9X\n', a + (a<0)*2^32)
        6
        A
        E
       2C
```

See Also

`disp`, `fclose`, `ferror`, `fopen`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`

References

[1] Kernighan, B.W., and D.M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: “Programming Language C,” ANSI, 1430 Broadway, New York, NY 10018.

Purpose

Write text to device

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Arguments

`obj`	A serial port object.
`'cmd'`	The string written to the device.
`'format'`	C language conversion specification.
`'mode'`	Specifies whether data is written synchronously or asynchronously.

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the device connected to `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command line until execution is complete.

`fprintf(obj,'format','cmd')` writes the string using the format specified by *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sprintf` file I/O format specifications or a C manual for more information.

`fprintf(obj,'cmd','mode')` writes the string with command line access specified by *`mode`*. If *`mode`* is `sync`, `cmd` is written synchronously and the command line is blocked. If *`mode`* is `async`, `cmd` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fprintf(obj,'format','cmd','mode')` writes the string using the specified format. If *`mode`* is `sync`, `cmd` is written synchronously. If *`mode`* is `async`, `cmd` is written asynchronously.

Remarks

Before you can write text to the device, it must be connected to `obj` with the `fclose` function. A connected serial port object has a `Status`

property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the device.

The `ValuesSent` property value is increased by the number of values written each time `fprintf` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

If you use the `help` command to display help for `fprintf`, then you need to supply the pathname shown below.

```
help serial/fprintf
```

Synchronous Versus Asynchronous Write Operations

By default, text is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in Controlling Access to the MATLAB Command Line.

Rules for Completing a Write Operation with fprintf

A synchronous or asynchronous write operation using `fprintf` completes when:

- The specified data is written.

- The time specified by the Timeout property passes.

Additionally, you can stop an asynchronous write operation with the stopasync function.

Rules for Writing the Terminator

All occurrences of \n in cmd are replaced with the Terminator property value. Therefore, when using the default format %s\n, all commands written to the device will end with this property value. The terminator required by your device will be described in its documentation.

Example

Create the serial port object s, connect s to a Tektronix TDS 210 oscilloscope, and write the RS232? command with the fprintf function. RS232? instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default format for fprintf is %s\n, the terminator specified by the Terminator property was automatically written. However, in some cases you might want to suppress writing the terminator. To do so, you must explicitly specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
fprintf(s,'%s','RS232?')
```

See Also

Functions

fopen, fwrite, stopasync

Properties

BytesToOutput, OutputBufferSize, OutputEmptyFcn, Status,TransferStatus, ValuesSent

frame2im

Purpose

Convert movie frame to indexed image

Syntax

```
[X,Map] = frame2im(F)
```

Description

`[X,Map] = frame2im(F)` converts the single movie frame `F` into the indexed image `X` and associated colormap `Map`. The functions `getframe` and `im2frame` create a movie frame. If the frame contains true-color data, then `Map` is empty.

See Also

`getframe`, `im2frame`, `movie`

"Bit-Mapped Images" on page 1-92 for related functions

Purpose

Edit print frames for Simulink and Stateflow block diagrams

Syntax

```
frameedit
frameedit filename
```

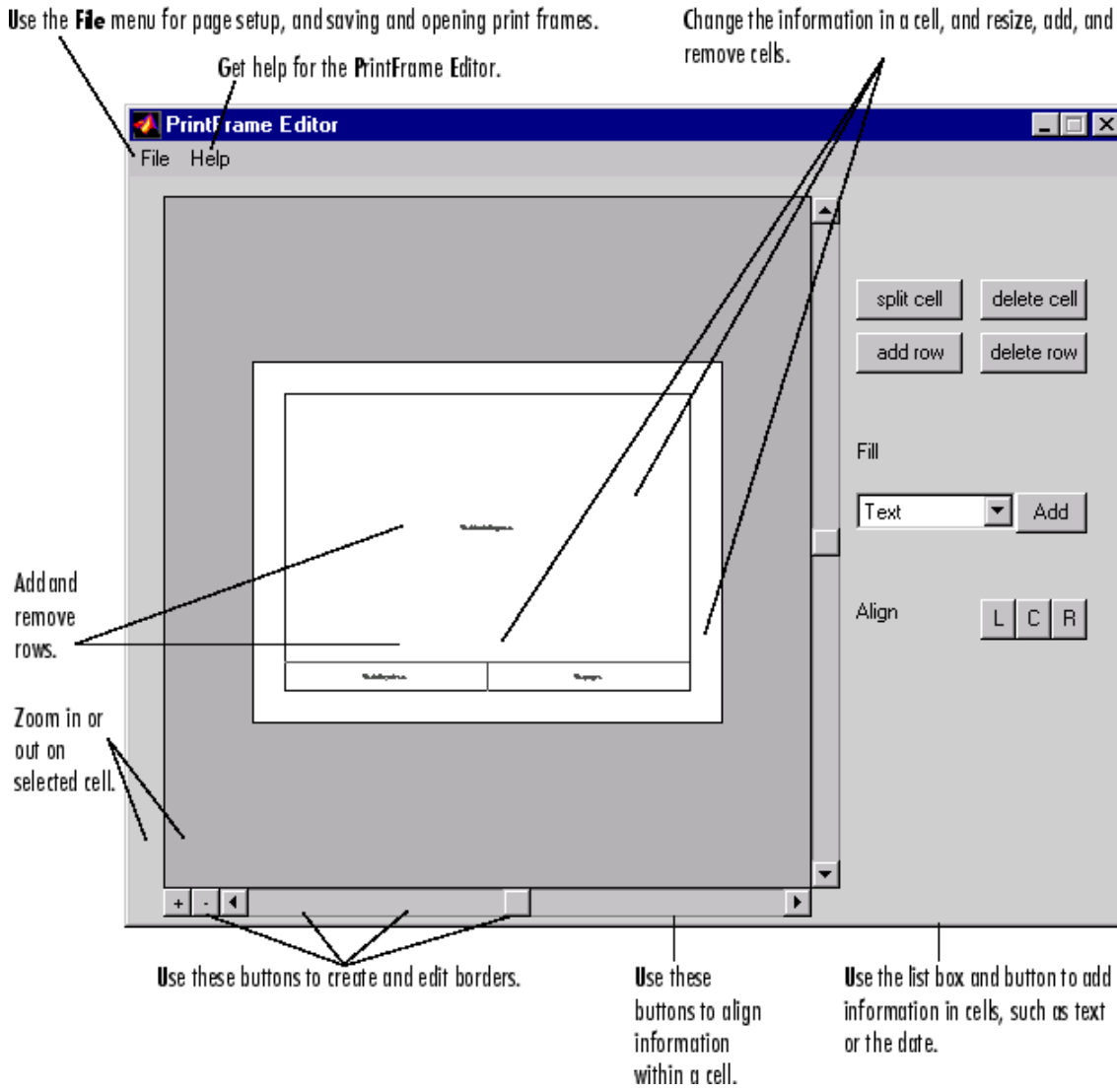
Description

`frameedit` starts the PrintFrame Editor, a graphical user interface you use to create borders for Simulink and Stateflow block diagrams. With no argument, `frameedit` opens the **PrintFrame Editor** window with a new file.

`frameedit filename` opens the **PrintFrame Editor** window with the specified filename, where `filename` is a figure file (`.fig`) previously created and saved using `frameedit`.

Remarks

This illustrates the main features of the PrintFrame Editor.



Closing the PrintFrame Editor

To close the **PrintFrame Editor** window, click the close box in the upper right corner, or select **Close** from the **File** menu.

Printing Simulink Block Diagrams with Print Frames

Select **Print** from the Simulink **File** menu. Check the **Frame** box and supply the filename for the print frame you want to use. Click **OK** in the **Print** dialog box.

Getting Help for the PrintFrame Editor

For further instructions on using the PrintFrame Editor, select **PrintFrame Editor Help** from the **Help** menu in the PrintFrame Editor.

fread

Purpose

Read binary data from file

Syntax

```
A = fread(fid)
A = fread(fid, count)
A = fread(fid, count, precision)
A = fread(fid, count, precision, skip)
A = fread(fid, count, precision, skip, machineformat)
[A, count] = fread(...)
```

Description

`A = fread(fid)` reads data in binary format from the file specified by `fid` into matrix `A`. Open the file using `fopen` before calling `fread`. The `fid` argument is the integer file identifier obtained from the `fopen` operation. MATLAB reads the file from beginning to end, and then positions the file pointer at the end of the file (see `feof` for details).

Note `fread` is intended primarily for binary data. When reading text files, use the `fgetl` function.

`A = fread(fid, count)` reads the number of elements specified by count. At the end of the `fread`, MATLAB sets the file pointer to the next byte to be read. A subsequent fread will begin at the location of the file pointer. See “Specifying the Number of Elements” on page 2-1293, below.

Note In the following syntaxes, the `count` and `skip` arguments are optional. For example, `fread(fid, precision)` is a valid syntax.

`A = fread(fid, count, precision)` reads the file according to the data format specified by the string `precision`. This argument commonly contains a data type specifier such as `int` or `float`, followed by an integer giving the size in bits. See “Specifying precision” on page 2-1293 and “Specifying Output Format” on page 2-1295, below.

`A = fread(fid, count, precision, skip)` includes an optional `skip` argument that specifies the number of bytes to skip after each `precision` value is read. If `precision` specifies a bit format like `'bitN'` or `'ubitN'`, the `skip` argument is interpreted as the number of bits to skip. See “Specifying a Skip Value” on page 2-1296, below.

`A = fread(fid, count, precision, skip, machineformat)` treats the data read as having a format given by `machineformat`. You can obtain the `machineformat` argument from the output of the `fopen` function. See `fopen` for possible values for `machineformat`.

`[A, count] = fread(...)` returns the data read from the file in `A`, and the number of elements successfully read in `count`.

Specifying the Number of Elements

Valid options for count are

`n`	Reads `n` elements into a column vector.
`inf`	Reads to the end of the file, resulting in a column vector containing the same number of elements as are in the file. If using `inf` results in an "out of memory" error, specify a numeric count value.
`[m,n]`	Reads enough elements to fill an `m`-by-`n` matrix, filling in elements in column order, padding with zeros if the file is too small to fill the matrix. `n` can be specified as `inf`, but `m` cannot.

Specifying precision

Any of the strings in the following table, either the MATLAB version or their C or Fortran equivalent, can be used for `precision`. If `precision` is not specified, MATLAB uses the default, which is `'uint8'`.

MATLAB	C or Fortran	Interpretation
’schar’	’signed char’	Signed integer; 8 bits
’uchar’	’unsigned char’	Unsigned integer; 8 bits

MATLAB	C or Fortran	Interpretation
'int8'	'integer*1'	Integer; 8 bits
'int16'	'integer*2'	Integer; 16 bits
'int32'	'integer*4'	Integer; 32 bits
'int64'	'integer*8'	Integer; 64 bits
'uint8'	'integer*1'	Unsigned integer; 8 bits
'uint16'	'integer*2'	Unsigned integer; 16 bits
'uint32'	'integer*4'	Unsigned integer; 32 bits
'uint64'	'integer*8'	Unsigned integer; 64 bits
'float32'	'real*4'	Floating-point; 32 bits
'float64'	'real*8'	Floating-point; 64 bits
'double'	'real*8'	Floating-point; 64 bits

The following platform-dependent formats are also supported, but they are not guaranteed to be the same size on all platforms.

MATLAB	C or Fortran	Interpretation
'char'	'char*1'	Character
'short'	'short'	Integer; 16 bits
'int'	'int'	Integer; 32 bits
'long'	'long'	Integer; 32 or 64 bits
'ushort'	'unsigned short'	Unsigned integer; 16 bits
'uint'	'unsigned int'	Unsigned integer; 32 bits
'ulong'	'unsigned long'	Unsigned integer; 32 or 64 bits
'float'	'float'	Floating-point; 32 bits

Note If the format is `'char'` or `'char*1'`, MATLAB reads characters using the encoding scheme associated with the file. See `fopen` for more information.

The following formats map to an input stream of bits rather than bytes.

MATLAB	C or Fortran	Interpretation
'bitN'	-	Signed integer; N bits ($1 \le N \le 64$)
'ubitN'	-	Unsigned integer; N bits ($1 \le N \le 64$)

Specifying Output Format

By default, numeric and character values are returned in class `double` arrays. To return these values stored in classes other than `double`, create your `format` argument by first specifying your source format, then following it with the characters "=>," and finally specifying your destination format. You are not required to use the exact name of a MATLAB class type for destination. (See `class` for details). `fread` translates the name to the most appropriate MATLAB class type. If the source and destination formats are the same, the following shorthand notation can be used.

```
*source
```

which means

```
source=>source
```

For example, `'*uint16'` is the same as `'uint16=>uint16'`.

Note You can also use the `*source` notation with an input stream that is specified as a number of bits (e.g., `bit4` or `ubit18`). MATLAB translates this into an output type that is a signed or unsigned integer (depending on the input type), and that is large enough to hold all of the bits in the source format. For example, `*ubit18` does not translate to `ubit18=>ubit18`, but instead to `ubit18=>uint32`.

This table shows some example precision format strings.

`'uint8=>uint8'`	Read in unsigned 8-bit integers and save them in an unsigned 8-bit integer array.
`'*uint8'`	Shorthand version of the above.
`'bit4=>int8'`	Read in signed 4-bit integers packed in bytes and save them in a signed 8-bit array. Each 4-bit integer becomes an 8-bit integer.
`'double=>real*4'`	Read in doubles, convert, and save as a 32-bit floating-point array.

Specifying a Skip Value

When `skip` is used, the `precision` string can contain a positive integer repetition factor of the form `'N*'`, which prefixes the source format specification, such as `'40*uchar'`.

Note Do not confuse the asterisk (*) used in the repetition factor with the asterisk used as precision format shorthand. The format string `'40*uchar'` is equivalent to `'40*uchar=>double'`, not `'40*uchar=>uchar'`.

When `skip` is specified, `fread` reads in, at most, a repetition factor number of values (default is 1), skips the amount of input specified by the `skip` argument, reads in another block of values, again skips

input, and so on, until `count` number of values have been read. If a `skip` argument is not specified, the repetition factor is ignored. Use the repetition factor with the `skip` argument to extract data in noncontiguous fields from fixed-length records.

Remarks

If the input stream is bytes and `fread` reaches the end of file (see `feof`) in the middle of reading the number of bytes required for an element, the partial result is ignored. However, if the input stream is bits, then the partial result is returned as the last value. If an error occurs before reaching the end of file, only full elements read up to that point are used.

Examples

Example 1

The file `alphabet.txt` contains the 26 letters of the English alphabet, all capitalized. Open the file for read access with `fopen`, and read the first five elements into output `c`. Because a precision has not been specified, MATLAB uses the default precision of `uint8`, and the output is numeric:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 5)'
c =
    65    66    67    68    69
fclose(fid);
```

This time, specify that you want each element read as an unsigned 8-bit integer and output as a character. (Using a precision of `'char=>char'` or `'*char'` will produce the same result):

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, 5, 'uint8=>char')'
c =
    ABCDE
fclose(fid);
```

When you leave out the optional count argument, MATLAB reads the file to the end, `A` through `Z`:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, '*char')'
c =
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
fclose(fid);
```

The `fopen` function positions the file pointer at the start of the file. So the first `fread` in this example reads the first five elements in the file, and then repositions the file pointer at the beginning of the next element. For this reason, the next `fread` picks up where the previous `fread` left off, at the character `F`.

```
fid = fopen('alphabet.txt', 'r');
c1 = fread(fid, 5, '*char');
c2 = fread(fid, 8, '*char');
c3 = fread(fid, 5, '*char');
fclose(fid);

sprintf('%c', c1, ' * ', c2, ' * ', c3)
ans =
   ABCDE * FGHIJKLM * NOPQR
```

Skip two elements between each read by specifying a `skip` argument of 2:

```
fid = fopen('alphabet.txt', 'r');
c = fread(fid, '*char', 2);    % Skip 2 bytes per read
fclose(fid);

sprintf('%c', c)
ans =
   ADGJMPSVY
```

Example 2

This command displays the complete M-file containing this `fread` help entry:

```
type fread.m
```

To simulate this command using `fread`, enter the following:

```
fid = fopen('fread.m', 'r');
F = fread(fid, '*char')';
fclose(fid);
```

In the example, the `fread` command assumes the default size, `'inf'`, and precision `'*char'` (the same as `'char=>char'`). `fread` reads the entire file. To display the result as readable text, the column vector is transposed to a row vector.

Example 3

As another example,

```
s = fread(fid, 120, '40*uchar=>uchar', 8);
```

reads in 120 bytes in blocks of 40, each separated by 8 bytes. Note that the class type of `s` is `'uint8'` since it is the appropriate class corresponding to the destination format `'uchar'`. Also, since 40 evenly divides 120, the last block read is a full block, which means that a final skip is done before the command is finished. If the last block read is not a full block, then `fread` does not finish with a skip.

See `fopen` for information about reading big and little-endian files.

Example 4

Invoke the `fopen` function with just an `fid` input argument to obtain the machine format for the file. You can see that this file was written in IEEE floating point with little-endian byte ordering (`'ieee-le'`) format:

```
fid = fopen('A1.dat', 'r');

[fname, mode, mformat] = fopen(fid);
mformat
mformat =
   ieee-le
```

Use the MATLAB format function (not related to the machine format type) to have MATLAB display output using hexadecimal:

```
format hex
```

Now use the `machineformat` input with `fread` to read the data from the file using the same format:

```
x = fread(fid, 6, 'uint64', 'ieee-le')
x =
   4260800000002000
   0000000000000000
   4282000000180000
   0000000000000000
   42ca5e0000258000
   42f0000464d45200
fclose(fid);
```

Change the machine format to IEEE floating point with big-endian byte ordering (`'ieee-be'`) and verify that you get different results:

```
fid = fopen('A1.dat', 'r');
x = fread(fid, 6, 'uint64', 'ieee-be')
x =
   4370000008400000
   0000000000000000
   4308000200100000
   0000000000000000
   4352c0002f0d0000
   43c022a6a3000000
fclose(fid);
```

Example 5

This example reads some Japanese text from a file that uses the Shift-JIS character encoding scheme. It creates a string of Unicode characters, `str`, and displays the string. Note that the computer must be configured to display Japanese (e.g., a Japanese Windows machine) for the output of `disp(str)` to be correct.

```
fid = fopen('japanese.txt', 'r', 'n', 'Shift_JIS');
str = fread(fid, '*char')';
```

```
fclose(fid);
disp(str);
```

See Also

fgetl, fscanf, fwrite, fprintf, fopen, fclose, fseek, ftell, feof

fread (serial)

Purpose

Read binary data from device

Syntax

```
A = fread(obj)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

Arguments

`obj`	A serial port object.
`size`	The number of values to read.
`'precision'`	The number of bits read for each value, and the interpretation of the bits as character, integer, or floating-point values.
`A`	Binary data returned from the device.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = fread(obj)` and A = fread(obj,size) read binary data from the device connected to `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *`precision`* (see below).

`A = fread(obj,size,'precision')` reads binary data with precision specified by *`precision`*.

`precision` controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *`precision`* are listed below in Remarks.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.

Remarks

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

If you use the `help` command to display help for `fread`, then you need to supply the pathname shown below.

```
help serial/fread
```

Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB command line until:

- The specified number of values are read.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used for binary read operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer

Data Type	Precision	Interpretation
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

Functions

`fgetl`, `fgets`, `fopen`, `fscanf`

Properties

`BytesAvailable`, `BytesAvailableFcn`, `InputBufferSize`, `Status`, `Terminator`, `ValuesReceived`

freqspace

Purpose

Frequency spacing for frequency response

Syntax

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for various one- and two-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`f = freqspace(N)` returns the one-dimensional frequency vector `f` assuming `N` evenly spaced points around the unit circle. For `N` even or odd, `f` is `(0:2/N:1)`. For `N` even, `freqspace` therefore returns `(N+2)/2` points. For `N` odd, it returns `(N+1)/2` points.

`f = freqspace(N,'whole')` returns `N` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/N:2*(N-1)/N`.

See Also

`meshgrid`

Purpose

Move file position indicator to beginning of open file

Syntax

```
frewind(fid)
```

Description

`frewind(fid)` sets the file position indicator to the beginning of the file specified by `fid`, an integer file identifier obtained from `fopen`.

Remarks

Rewinding a `fid` associated with a tape device might not work even though `frewind` does not generate an error message.

See Also

`fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`, `fwrite`

fscanf

Purpose

Read formatted data from file

Syntax

```
A = fscanf(fid, format)
[A,count] = fscanf(fid, format, size)
```

Description

`A = fscanf(fid, format)` reads data from the file specified by `fid`, converts it according to the specified `format` string, and returns it in matrix `A`. Argument `fid` is an integer file identifier obtained from `fopen`. `format` is a string specifying the format of the data to be read. See "Remarks" for details.

`[A,count] = fscanf(fid, format, size)` reads the amount of data specified by `size`, converts it according to the specified `format` string, and returns it along with a `count` of values successfully read. `size` is an argument that determines how much data is read. Valid options are

`n`	Read at most `n` numbers, characters, or strings.
`inf`	Read to the end of the file.
`[m,n]`	Read at most `(m*n)` numbers, characters, or strings. Fill a matrix of at most `m` rows in column order. `n` can be `inf`, but `m` cannot.

Characteristics of the output matrix `A` depend on the values read from the file and on the `size` argument. If fscanf reads only numbers, and if `size` is not of the form `[m,n]`, matrix `A` is a column vector of numbers. If fscanf reads only characters or strings, and if `size` is not of the form `[m,n]`, matrix `A` is a row vector of characters. See the Remarks section for more information.

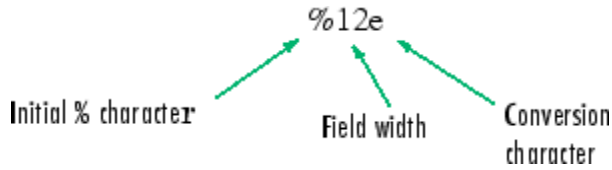
`fscanf` differs from its C language namesake `fscanf()` in an important respect — it is *vectorized* to return a matrix argument. The `format` string is cycled through the file until the first of these conditions occurs:

- The `format` string fails to match the data in the file
- The amount of data specified by `size` is read
- The end of the file is reached

Remarks

When MATLAB reads a specified file, it attempts to match the data in the file to the `format` string. If a match occurs, the data is written into the output matrix. If a partial match occurs, only the matching data is written to the matrix, and the read operation stops.

The `format` string consists of ordinary characters and/or conversion specifications. Conversion specifications indicate the type of data to be matched and involve the character `%`, optional width fields, and conversion characters, organized as shown below.



Add one or more of these characters between the `%` and the conversion character:

An asterisk (`*`)	Skip over the matched value. If `%*d`, then the value that matches `d` is ignored and is not stored.
A digit string	Maximum field width. For example, `%10d`.
A letter	The size of the receiving object, for example, `h` for short, as in `%hd` for a short integer, or `l` for long, as in `%ld` for a long integer, or `%lg` for a double floating-point number.

Valid conversion characters are

`%c`	Sequence of characters; number specified by field width
`%d`	Base 10 integers
`%e`, `%f`, `%g`	Floating-point numbers
`%i`	Defaults to base 10 integers. Data starting with `0` is read as base 8. Data starting with `0x` or `0X` is read as base 16.

`%o`	Signed octal integer
`%s`	A series of non-white-space characters
`%u`	Signed decimal integer
`%x`	Signed hexadecimal integer
`[...]`	Sequence of characters (`scanlist`)

Format specifiers `%e`, `%f`, and `%g` accept the text `'inf'`, `'-inf'`, `'nan'`, and `'-nan'`. This text is not case sensitive. The `fscanf` function converts these to the numeric representation of `Inf`, `-Inf`, `NaN`, and `-NaN`.

Use `%c` to read space characters or `%s` to skip all white space. MATLAB skips over any ordinary characters that are used in the format specifier (see Example 2 below).

MATLAB reads characters using the encoding scheme associated with the file. See `fopen` for more information. If the format string contains ordinary characters, MATLAB matches each of those characters with a character read from the file after converting both to the MATLAB internal representation of characters.

For more information about format strings, refer to the `scanf()` and `fscanf()` routines in a C language reference manual.

Output Characteristics: Only Numeric Values Read

Format characters that cause `fscanf` to read numbers from the file are `%d`, `%e`, `%f`, `%g`, `%i`, `%o`, `%u`, and `%x`. When `fscanf` reads only numbers from the file, the elements of the output matrix `A` are numbers.

When there is no `size` argument or the `size` argument is `inf`, `fscanf` reads to the end of the file. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a scalar `n`, fscanf reads at most `n` numbers from the file. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a matrix `[m,n]`, `fscanf` reads at most `(m*n)` numbers from the file. The output matrix contains at most `m` rows and `n` columns. `fscanf` fills the output matrix in column order, using as many columns as it needs to contain all the numbers read from the input. Any unfilled elements in the final column contain zeros.

Output Characteristics: Only Character Values Read

The format characters that cause `fscanf` to read characters and strings from the file are `%c` and `%s`. When `fscanf` reads only characters and strings from the file, the elements of the output matrix `A` are characters. When fscanf reads a string from the input, the output matrix includes one element for each character in the string.

When there is no `size` argument or the `size` argument is `inf`, `fscanf` reads to the end of the file. The output matrix is a row vector with one element for each character read from the input.

When the `size` argument is a scalar `n`, `fscanf` reads at most `n` character or string values from the file. The output matrix is a row vector with one element for each character read from the input. When string values are read from the input, the output matrix can contain more than `n` columns.

When the `size` argument is a matrix `[m,n]`, `fscanf` reads at most `(m*n)` character or string values from the file. The output matrix contains at most `m` rows. `fscanf` fills the output matrix in column order, using as many columns as it needs to contain all the characters read from the input. When string values are read from the input, the output matrix can contain more than `n` columns. Any unfilled elements in the final column contain `char(0)`.

Output Characteristics: Both Numeric and Character Values Read

When `fscanf` reads a combination of numbers and either characters or strings from the file, the elements of the output matrix `A` are numbers. This is true even when a format specifier such as `'%*d %s'` tells MATLAB to ignore numbers in the input string and output only characters or strings. When `fscanf` reads a string from the input, the

output matrix includes one element for each character in the string. All characters are converted to their numeric equivalents in the output matrix.

When there is no `size` argument or the `size` argument is `inf`, `fscanf` reads to the end of the file. The output matrix is a column vector with one element for each character read from the input.

When the `size` argument is a scalar `n`, `fscanf` reads at most `n` number, character, or string values from the file. The output matrix contains at most `n` rows. `fscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than one column. Any unfilled elements in the final column contain zeros.

When the `size` argument is a matrix `[m,n]`, `fscanf` reads at most `(m*n)` number, character, or string values from the file. The output matrix contains at most `m` rows. `fscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than `n` columns. Any unfilled elements in the final column contain zeros.

Note This section applies only when `fscanf` actually reads a combination of numbers and either characters or strings from the file. Even if the format string has both format characters that would result in numbers (such as `%d`) and format characters that would result in characters or strings (such as `%s`), `fscanf` might actually read only numbers or only characters or strings. If `fscanf` reads only numbers, see "Output Characteristics: Only Numeric Values Read" on page 2-1310. If `fscanf` reads only characters or strings, see "Output Characteristics: Only Character Values Read" on page 2-1311.

Examples

Example 1

An example in `fprintf` generates a text file called `exp.txt` that looks like this:

```
0.00    1.00000000
0.10    1.10517092
...
1.00    2.71828183
```

Read this file back into a two-column MATLAB matrix:

```
fid = fopen('exp.txt', 'r');
a = fscanf(fid, '%g %g', [2 inf])    % It has two rows now.
a = a';
fclose(fid)
```

Example 2

Start with a file `temp.dat` that contains temperature readings:

```
78 F 72 F 64 F 66 F 49 F
```

Open the file using `fopen` and read it with `fscanf`. If you include ordinary characters (such as the degree (°) and Farrenheit (`F`) symbols used here) in the conversion string, `fscanf` skips over those characters when reading the string:

```
fid = fopen('temps.dat', 'r');

degrees = char(176)
degrees =

fscanf(fid, ['%d' degrees 'F'])
ans =
    78
    72
    64
    66
    49
```

See Also

fgetl, fgets, fread, fprintf, fscanf, input, sscanf, textread

Purpose

Read data from device, and format as text

Syntax

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

Arguments

`obj`	A serial port object.
`'format'`	C language conversion specification.
`size`	The number of values to read.
`A`	Data read from the device and formatted as text.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = fscanf(obj)` reads data from the device connected to `obj`, and returns it to `A`. The data is converted to text using the `%c` format.

`A = fscanf(obj,'format')` reads data and converts it according to *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'format',size)` reads the number of values specified by `size`. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

size cannot be inf, and an error is returned if the specified number of values cannot be stored in the input buffer. If size is not of the form [m,n], and a character conversion is specified, then A is returned as a row vector. You specify the size, in bytes, of the input buffer with the InputBufferSize property. An ASCII value is one byte.

[A,count] = fscanf(...) returns the number of values read to count.

[A,count,msg] = fscanf(...) returns a warning message to msg if the read operation did not complete successfully.

Remarks

Before you can read data from the device, it must be connected to obj with the fopen function. A connected serial port object has a Status property value of open. An error is returned if you attempt to perform a read operation while obj is not connected to the device.

If msg is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The ValuesReceived property value is increased by the number of values read – including the terminator – each time fscanf is issued.

If you use the help command to display help for fscanf, then you need to supply the pathname shown below.

```
help serial/fscanf
```

Rules for Completing a Read Operation with fscanf

A read operation with fscanf blocks access to the MATLAB command line until:

- The terminator specified by the Terminator property is read.
- The time specified by the Timeout property passes.
- The number of values specified by size is read.
- The input buffer is filled (unless size is specified)

Example

Create the serial port object `s` and connect `s` to a Tektronix TDS 210 oscilloscope, which is displaying sine wave.

```
s = serial('COM1');
fopen(s)
```

Use the `fprintf` function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VALUE?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    21
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
meas =
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',14)
pk2pk =
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

fgetl, fgets, fopen, fread, strread

Properties

BytesAvailable, BytesAvailableFcn, InputBufferSize, Status, Terminator, Timeout

Purpose

Set file position indicator

Syntax

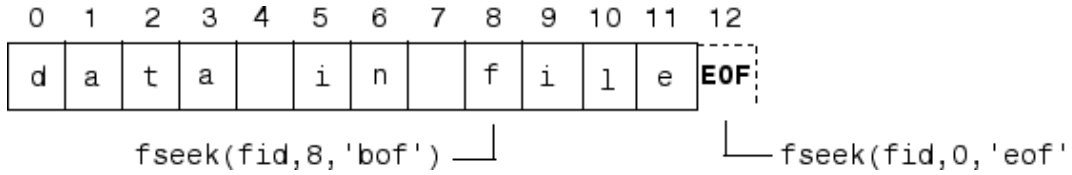
```
status = fseek(fid, offset, origin)
```

Description

`status = fseek(fid, offset, origin)` repositions the file position indicator in the file with the given `fid` to the byte with the specified `offset` relative to `origin`.

For a file having `n` bytes, the bytes are numbered from `0` to `n-1`. The position immediately following the last byte is the end-of-file, or `eof`, position. You would seek to the `eof` position if you wanted to add data to the end of a file.

This figure represents a file having 12 bytes, numbered 0 through 11. The first command shown seeks to the ninth byte of data in the file. The second command seeks just past the end of the file data, to the `eof` position.



`fseek` does not seek beyond the end of file `eof` position. If you attempt to seek beyond `eof`, MATLAB returns an error status.

Arguments

`fid`	An integer file identifier obtained from `fopen`	
`offset`	A value that is interpreted as follows,	
	`offset > 0`	Move position indicator `offset` bytes toward the end of the file.
	`offset = 0`	Do not change position.

	`offset < 0`	Move position indicator `offset` bytes toward the beginning of the file.
`origin`	A string whose legal values are	
	`'bof'`	`-1`: Beginning of file
	`'cof'`	`0`: Current position in file
	`'eof'`	`1`: End of file
`status`	A returned value that is `0` if the `fseek` operation is successful and `-1` if it fails. If an error occurs, use the function `ferror` to get more information.	

Examples

This example opens the file `test1.dat`, seeks to the 20th byte, reads fifty 32-bit unsigned integers into variable `A`, and closes the file. It then opens a second file, `test2.dat`, seeks to the end-of-file position, appends the data in `A` to the end of this file, and closes the file.

```
fid = fopen('test1.dat', 'r');
fseek(fid, 19, 'bof');
A = fread(fid, 50, 'uint32');
fclose(fid);

fid = fopen('test2.dat', 'r+');
fseek(fid, 0, 'eof');
fwrite(fid, A, 'uint32');
fclose(fid);
```

See Also

`fopen`, `fclose`, `ferror`, `fprintf`, `fread`, `fscanf`, `ftell`, `fwrite`

Purpose

File position indicator

Syntax

```
position = ftell(fid)
```

Description

`position = ftell(fid)` returns the location of the file position indicator for the file specified by `fid`, an integer file identifier obtained from `fopen`. The `position` is a nonnegative, zero-based integer specified in bytes from the beginning of the file. A returned value of `-1` for `position` indicates that the query was unsuccessful; use `ferror` to determine the nature of the error.

Remarks

`ftell` is likely to return an invalid `position` when *all* of the following are true. This is due to the way in which the Microsoft Windows C library currently handles its `ftell` and `fgetpos` commands:

- The file you are currently operating on is an ASCII text file.
- The file was written on a UNIX-based system, or uses the UNIX-style line terminator: a line feed (with no carriage return) at the end of each line of text. (This is the default output format for MATLAB functions `dlmwrite` and `csvwrite`.)
- You are reading the file on a Windows system.
- You opened the file with the `fopen` function with `mode` set to `'rt'`.
- The `ftell` command is directly preceded by an `fgets` or `fgetl` command.

Note that this does not affect the ability to accurately read from and write to this type of file from MATLAB.

See Also

`fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `fwrite`

Purpose

Connect to FTP server, creating FTP object

Syntax

```
f = ftp('host','username','password')
```

Description

`f = ftp('host','username','password')` connects to the FTP server, `host`, creating the FTP object, `f`. If a username and password are not required for an anonymous connection, only use the `host` argument. Specify an alternate port by separating it from `host` using a colon (:). After running `ftp`, perform file operation functions on the FTP object, `f`, using methods such as `cd` and others listed under "See Also." When you're finished using the server, run `close (ftp)` to close the connection.

FTP is not a secure protocol; others can see your username and password.

The `ftp` function is based on code from the Apache Jakarta Project.

Examples

Connect Without Username

Connect to `ftp.mathworks.com`, which does not require a username or password. Assign the resulting FTP object to `tmw`. You can access this FTP site to experiment with the FTP functions.

```
tmw=ftp('ftp.mathworks.com')
```

MATLAB returns

```
tmw =
 FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: binary
```

Connect to Specified Port

To connect to port 34, type

```
tmw=ftp('ftp.mathworks.com:34')
```

Connect with Username

Connect to `ftp.testsite.com` and assign the resulting FTP object to `test`.

```
test=ftp('ftp.testsite.com','myname','mypassword')
```

MATLAB returns

```
test =
 FTP Object
  host: ftp.testsite.com
  user: myname
   dir: /
  mode: binary
  myname@ftp.testsite.com
  /
```

See Also

`ascii`, `binary`, `cd (ftp)`, `close (ftp)`, `delete (ftp)`, `dir (ftp)`, `mget`, `mkdir (ftp)`, `mput`, `rename`, `rmdir (ftp)`

full

Purpose

Convert sparse matrix to full matrix

Syntax

```
A = full(S)
```

Description

`A = full(S)` converts a sparse matrix `S` to full storage organization. If `S` is a full matrix, it is left unchanged. If `A` is full, `issparse(A)` is `0`.

Remarks

Let `X` be an `m`-by-`n` matrix with `nz = nnz(X)` nonzero entries. Then `full(X)` requires space to store `m*n` real numbers while `sparse(X)` requires space to store `nz` real numbers and `(nz+n)` integers.

On most computers, a real number requires twice as much storage as an integer. On such computers, `sparse(X)` requires less storage than `full(X)` if the density, `nnz/prod(size(X))`, is less than one third. Operations on sparse matrices, however, require more execution time per element than those on full matrices, so density should be considerably less than two-thirds before sparse storage is used.

Examples

Here is an example of a sparse matrix with a density of about two-thirds. `sparse(S)` and `full(S)` require about the same number of bytes of storage.

```
S = sparse(+(rand(200,200) < 2/3));
A = full(S);
whos
Name    Size    Bytes     Class
  A    200X200 320000 double array
  S    200X200 318432 double array (sparse)
```

See Also

`issparse`, `sparse`

Purpose

Build full filename from parts

Syntax

```
f = fullfile(dir1, dir2, ..., filename)
```

Description

`f = fullfile(dir1, dir2, ..., filename)` builds a full file specification `f` from the directories and filename specified. Input arguments `dir1`, `dir2`, etc. and `filename` are each a string enclosed in single quotes. The output of the `fullfile` command is conceptually equivalent to

```
f = [dir1 filesep dir2 filesep ... filesep filename]
```

except that care is taken to handle the cases when the directories begin or end with a directory separator.

Examples

To create the full filename from a disk name, directories, and filename,

```
f = fullfile('C:', 'Applications', 'matlab', 'myfun.m')
f =
C:\Applications\matlab\myfun.m
```

The following examples both produce the same result on UNIX, but only the second one works on all platforms.

```
fullfile(matlabroot, 'toolbox/matlab/general/Contents.m')
fullfile(matlabroot, 'toolbox', 'matlab', 'general', ...
         'Contents.m')
```

See Also

`fileparts`, `filesep`, `path`, `pathsep`, `genpath`

Purpose

Construct function name string from function handle

Syntax

```
func2str(fhandle)
```

Description

`func2str(fhandle)` constructs a string `s` that holds the name of the function to which the function handle `fhandle` belongs.

When you need to perform a string operation, such as compare or display, on a function handle, you can use `func2str` to construct a string bearing the function name.

The `func2str` command does not operate on nonscalar function handles. Passing a nonscalar function handle to `func2str` results in an error.

Examples

Example 1

Convert a `sin` function handle to a string:

```
fhandle = @sin;

func2str(fhandle)
ans =
    sin
```

Example 2

The `catcherr` function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, `catcherr` uses `sprintf` to display an error message giving the name of the failing function. The function name must be a string for `sprintf` to display it. The code derives the function name from the function handle using `func2str`:

```
function catcherr(func, data)
try
    ans = func(data);
    disp('Answer is:');
    ans
catch
```

```
        disp(sprintf('Error executing function ''%s''\n', ...
            func2str(func)))
    end
```

The first call to `catcherr` passes a handle to the `round` function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, `round` fails, causing `catcherr` to display an error message that includes the failing function name:

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

See Also

`function_handle`, `str2func`, `functions`

function

Purpose

Declare M-file function

Syntax

```
function [out1, out2, ...] = funname(in1, in2, ...)
```

Description

`function [out1, out2, ...] = funname(in1, in2, ...)` defines function `funname` that accepts inputs `in1`, `in2`, etc. and returns outputs `out1`, `out2`, etc.

You add new functions to the MATLAB vocabulary by expressing them in terms of existing functions. The existing commands and functions that compose the new function reside in a text file called an *M-file*.

M-files can be either *scripts* or *functions*. Scripts are simply files containing a sequence of MATLAB statements. Functions make use of their own local variables and accept input arguments.

The name of an M-file begins with an alphabetic character and has a filename extension of `.m`. The M-file name, less its extension, is what MATLAB searches for when you try to use the script or function.

A line at the top of a function M-file contains the syntax definition. The name of a function, as defined in the first line of the M-file, should be the same as the name of the file without the `.m` extension.

The variables within the body of the function are all local variables.

A *subfunction*,visible only to the other functions in the same file, is created by defining a new function with the `function` keyword after the body of the preceding function or subfunction. Subfunctions are not visible outside the file where they are defined.

You can terminate any function with an `end` statement but, in most cases, this is optional. `end` statements are required only in M-files that employ one or more nested functions. Within such an M-file, *every* function (including primary, nested, private, and subfunctions) must be terminated with an `end` statement. You can terminate any function type with `end`, but doing so is not required unless the M-file contains a nested function.

Functions normally return when the end of the function is reached. Use a `return` statement to force an early return.

When MATLAB does not recognize a function by name, it searches for a file of the same name on disk. If the function is found, MATLAB compiles it into memory for subsequent use. The section "Determining Which Function Is Called" in the MATLAB Programming documentation explains how MATLAB interprets variable and function names that you enter, and also covers the precedence used in function dispatching.

When you call an M-file function from the command line or from within another M-file, MATLAB parses the function and stores it in memory. The parsed function remains in memory until cleared with the `clear` command or you `quit` MATLAB. The `pcode` command performs the parsing step and stores the result on the disk as a P-file to be loaded later.

Examples

Example 1

The existence of a file on disk called `stat.m` containing this code defines a new function called `stat` that calculates the mean and standard deviation of a vector:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/n));
```

Example 2

`avg` is a subfunction within the file `stat.m`:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = avg(x,n);
stdev = sqrt(sum((x-avg(x,n)).^2)/n);

function mean = avg(x,n)
mean = sum(x)/n;
```

See Also

`nargin`, `nargout`, `pcode`, `varargin`, `varargout`, `what`

function_handle (@)

Purpose

Handle used in calling functions indirectly

Syntax

```
handle = @functionname
handle = @(arglist)anonymous_function
```

Description

`handle = @functionname` returns a handle to the specified MATLAB function.

A function handle is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks). A function handle is one of the standard MATLAB data types.

At the time you create a function handle, the function you specify must be on the MATLAB path and in the current scope. This condition does not apply when you evaluate the function handle. You can, for example, execute a subfunction from a separate (out-of-scope) M-file using a function handle as long as the handle was created within the subfunction's M-file (in-scope).

`handle = @(arglist)anonymous_function` constructs an anonymous function and returns a `handle` to that function. The body of the function, to the right of the parentheses, is a single MATLAB statement or command. `arglist` is a comma-separated list of input arguments. Execute the function by calling it by means of the function handle, `handle`.

Remarks

The function handle is a standard MATLAB data type. As such, you can manipulate and operate on function handles in the same manner as on other MATLAB data types. This includes using function handles in structures and cell arrays:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
C = {@sin, @cos, @tan};
```

However, standard matrices or arrays of function handles are not supported:

```
A = [@sin, @cos, @tan];        % This is not supported
```

For nonoverloaded functions, subfunctions, and private functions, a function handle references just the one function specified in the `@functionname` syntax. When you evaluate an overloaded function by means of its handle, the arguments the handle is evaluated with determine the actual function that MATLAB dispatches to.

Use `isa(h, 'function_handle')` to see if variable `h` is a function handle.

Examples

Example 1 — Constructing a Handle to a Named Function

The following example creates a function handle for the `humps` function and assigns it to the variable `fhandle`.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to `fminbnd`, which then minimizes over the interval `[0.3, 1]`.

```
x = fminbnd(fhandle, 0.3, 1)
x =
    0.6370
```

The `fminbnd` function evaluates the `@humps` function handle. A small portion of the `fminbnd` M-file is shown below. In line 1, the `funfcn` input parameter receives the function handle `@humps` that was passed in. The statement, in line 113, evaluates the handle.

```
1    function [xf,fval,exitflag,output] = ...
         fminbnd(funfcn,ax,bx,options,varargin)
             .
             .
             .
```

```
113  fx = funfcn(x,varargin{:});
```

Example 2 — Constructing a Handle to an Anonymous Function

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `quad` function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
   0.3333
```

See Also

`str2func`, `func2str`, `functions`, `isa`

Purpose

Information about function handle

Syntax

```
S = functions(funhandle)
```

Description

`S = functions(funhandle)` returns, in MATLAB structure `S`, the function name, type, filename, and other information for the function handle stored in the variable `funhandle`.

`functions` does not operate on nonscalar function handles. Passing a nonscalar function handle to `functions` results in an error.

Caution The `functions` function is provided for querying and debugging purposes. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

This table lists the standard fields of the return structure.

Field Name	Field Description
`function`	Function name
`type`	Function type (e.g., simple, overloaded)
`file`	The file to be executed when the function handle is evaluated with a nonoverloaded data type

Remarks

For handles to functions that overload one of the standard MATLAB data types, like `double` or `char`, the structure returned by `functions` contains an additional field named `methods`. The `methods` field is a substructure containing one field name for each MATLAB class that overloads the function. The value of each field is the path and name of the file that defines the method.

Examples

Example 1

To obtain information on a function handle for the `poly` function, type

```
f = functions(@poly)
f =
    function: 'poly'
        type: 'simple'
        file:  '$matlabroot\toolbox\matlab\polyfun\poly.m'
```

(The term `$matlabroot` used in this example stands for the file specification of the directory in which MATLAB software is installed for your system. Your output will display this file specification.)

Access individual fields of the returned structure using dot selection notation:

```
f.type
ans =
   simple
```

Example 2

The function `get_handles` returns function handles for a subfunction and private function in output arguments `s` and `p` respectively:

```
function [s, p] = get_handles
s = @mysubfun;
p = @myprivatefun;
%
function mysubfun
disp 'Executing subfunction mysubfun'
```

Call `get_handles` to obtain the two function handles, and then pass each to the `functions` function. MATLAB returns information in a structure having the fields `function`, `type`, `file`, and `parentage`. The `file` field contains the file specification for the subfunction or private function:

```
[fsub fprv] = get_handles;

functions(fsub)
ans =
```

```
     function: 'mysubfun'
         type: 'scopedfunction'
         file: 'c:\matlab\get_handles.m'
    parentage: {'mysubfun'  'get_handles'}

functions(fprv)
ans =
     function: 'myprivatefun'
         type: 'scopedfunction'
         file: 'c:\matlab\private\myprivatefun.m'
    parentage: {'myprivatefun'}
```

Example 3

In this example, the function `get_handles_nested.m` contains a nested function `nestfun`. This function has a single output which is a function handle to the nested function:

```
function handle = get_handles_nested(A)
nestfun(A);

   function y = nestfun(x)
   y = x + 1;
   end

handle = @nestfun;
end
```

Call this function to get the handle to the nested function. Use this handle as the input to `functions` to return the information shown here. Note that the `function` field of the return structure contains the names of the nested function and the function in which it is nested in the format. Also note that `functions` returns a `workspace` field containing the variables that are in context at the time you call this function by its handle:

```
fh = get_handles_nested(5);

fhinfo = functions(fh)
```

```
fhinfo =
     function: 'get_handles_nested/nestfun'
         type: 'nested'
         file: 'c:\matlab\get_handles_nested.m'
    workspace: [1x1 struct]

fhinfo.workspace
ans =
    handle: @get_handles_nested/nestfun
         A: 5
```

See Also

function_handle

Purpose

Evaluate general matrix function

Syntax

```
F = funm(A,fun)
F = funm(A, fun, options)
[F, exitflag] = funm(...)
[F, exitflag, output] = funm(...)
```

Description

`F = funm(A,fun)` evaluates the user-defined function `fun` at the square matrix argument `A`. `F = fun(x, k)` must accept a vector `x` and an integer `k`, and return a vector `f` of the same size of `x`, where `f(i)` is the `k`th derivative of the function `fun` evaluated at `x(i)`. The function represented by fun must have a Taylor series with an infinite radius of convergence, except for `fun = @log`, which is treated as a special case.

You can also use `funm` to evaluate the special functions listed in the following table at the matrix `A`.

Function	Syntax for Evaluating Function at Matrix A
exp	`funm(A, @exp)`
log	`funm(A, @log)`
sin	`funm(A, @sin)`
cos	`funm(A, @cos)`
sinh	`funm(A, @sinh)`
cosh	`funm(A, @cosh)`

For matrix square roots, use `sqrtm(A)` instead. For matrix exponentials, which of `expm(A)` or `funm(A, @exp)` is the more accurate depends on the matrix `A`.

"Parameterizing Functions Called by Function Functions", in the online MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`F = funm(A, fun, options)` sets the algorithm's parameters to the values in the structure `options`. The following table lists the fields of `options`.

Field	Description	Values
`options.TolBlk`	Level of display	`'off'` (default), `'on'`, `'verbose'`
`options.TolTay`	Tolerance for blocking Schur form	Positive scalar. The default is `eps`.
`options.MaxTerms`	Maximum number of Tayor series terms	Positive integer. The default is `250`.
`options.MaxSqrt`	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is `100`.
`options.Ord`	Specifies the ordering of the Schur form `T`.	A vector of length `length(A)`. `options.Ord(i)` is the index of the block into which `T(i,i)` is placed. The default is `[]`.

`[F, exitflag] = funm(...)` returns a scalar `exitflag` that describes the exit condition of `funm`. `exitflag` can have the following values:

- `0` — The algorithm was successful.
- `1` — One or more Taylor series evaluations did not converge. However, the computed value of `F` might still be accurate.

`[F, exitflag, output] = funm(...)` returns a structure `output` with the following fields:

Field	Description
`output.terms`	Vector for which `output.terms(i)` is the number of Taylor series terms used when evaluating the ith block, or, in the case of the logarithm, the number of square roots.
`output.ind`	Cell array for which the `(i,j)` block of the reordered Schur factor `T` is `T(output.ind{i}, output.ind{j})`.
`output.ord`	Ordering of the Schur form, as passed to `ordschur`
`output.T`	Reordered Schur form

If the Schur form is diagonal then `output = struct('terms',ones(n,1),'ind',{1:n})`.

Examples

Example 1

The following command computes the matrix sine of the 3-by-3 magic matrix.

```
F=funm(magic(3), @sin)

F =

   -0.3850    1.0191    0.0162
    0.6179    0.2168   -0.1844
    0.4173   -0.5856    0.8185
```

Example 2

The statements

```
S = funm(X,@sin);
C = funm(X,@cos);
```

produce the same results to within roundoff error as

```
E = expm(i*X);
C = real(E);
S = imag(E);
```

In either case, the results satisfy `S*S+C*C = I`, where `I = eye(size(X))`.

Example 3

To compute the function `exp(x) + cos(x)` at `A` with one call to `funm`, use

```
F = funm(A,@fun_expcos)
```

where `fun_expcos` is the following M-file function.

```
function f = fun_expcos(x, k)
% Return kth derivative of exp + cos at X.
        g = mod(ceil(k/2),2);
        if mod(k,2)
           f = exp(x) + sin(x)*(-1)^g;
        else
           f = exp(x) + cos(x)*(-1)^g;
        end
```

Algorithm

The algorithm `funm` uses is described in [1].

See Also

`expm`, `logm`, `sqrtm`, `function_handle` (`@`)

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Third Edition, Johns Hopkins University Press, 1996, p. 384.

[3] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later" *SIAM Review 20*, Vol. 45, Number 1, pp. 1-47, 2003.

Purpose

Write binary data to file

Syntax

```
count = fwrite(fid, A)
count = fwrite(fid, A, precision)
count = fwrite(fid, A, precision, skip)
count = fwrite(fid, A, precision, skip, machineformat)
```

Description

`count = fwrite(fid, A)` writes the elements of matrix `A` to the specified file. The data is written to the file in column order, and a `count` is kept of the number of elements written successfully.

`fid` is an integer file identifier obtained from `fopen`, or `1` for standard output or `2` for standard error.

`count = fwrite(fid, A, precision)` writes the elements of matrix `A` to the specified file, translating MATLAB values to the specified `precision`.

`precision` controls the form and size of the result. See `fread` for a list of allowed precisions. If `precision` is not specified, MATLAB uses the default, which is `'uint8'`. For `'bitN'` or `'ubitN'` precisions, `fwrite` sets all bits in `A` when the value is out of range. If the precision is `'char'` or `'char*1'`, MATLAB writes characters using the encoding scheme associated with the file. See `fopen` for more information.

`count = fwrite(fid, A, precision, skip)` includes an optional `skip` argument that specifies the number of bytes to skip before each `precision` value is written. With the `skip` argument present, `fwrite` skips and writes one value, skips and writes another value, etc., until all of `A` is written. If `precision` is a bit format like `'bitN'` or `'ubitN'`, `skip` is specified in bits. This is useful for inserting data into noncontiguous fields in fixed-length records.

`count = fwrite(fid, A, precision, skip, machineformat)` treats the data written as having a format given by `machineformat`. You can obtain the `machineformat` argument from the output of the `fopen` function. See `fopen` for possible values for `machineformat`.

Remarks

You cannot view or `type` the contents of the file you are writing with `fwrite` until you close the file with the `fclose` function.

Examples

Example 1

This example creates a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, stored as 4-byte integers:

```
fid = fopen('magic5.bin', 'wb');
fwrite(fid, magic(5), 'integer*4')
```

Example 2

This example takes a string of Unicode characters, `str`, which contains Japanese text, and writes the string into a file using the Shift-JIS character encoding scheme:

```
fid = fopen('japanese_out.txt', 'w', 'n', 'Shift_JIS');
fwrite(fid, str, 'char');
fclose(fid);
```

See Also

`fclose`, `ferror`, `fopen`, `fprintf`, `fread`, `fscanf`, `fseek`, `ftell`

fwrite (serial)

Purpose

Write binary data to device

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Arguments

`obj`	A serial port object.
`A`	The binary data written to the device.
`'precision'`	The number of bits written for each value, and the interpretation of the bits as character, integer, or floating-point values.
`'mode'`	Specifies whether data is written synchronously or asynchronously.

Description

`fwrite(obj,A)` writes the binary data `A` to the device connected to `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *`precision`*.

`precision` controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. The supported values for *`precision`* are listed below in Remarks.

`fwrite(obj,A,'mode')` writes binary data with command line access specified by *`mode`*. If *`mode`* is `sync`, `A` is written synchronously and the command line is blocked. If *`mode`* is `async`, `A` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fwrite(obj,A,'precision','mode')` writes binary data with precision specified by *`precision`* and command line access specified by *`mode`*.

Remarks

Before you can write data to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the device.

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

If you use the `help` command to display help for `fwrite`, then you need to supply the pathname shown below.

```
help serial/fwrite
```

Synchronous Versus Asynchronous Write Operations

By default, data is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The M-file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in Writing Data.

Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when:

- The specified data is written.

- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used with binary write operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer

Data Type	Precision	Interpretation
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

See Also

Functions

`fopen`, `fprintf`

Properties

`BytesToOutput`, `OutputBufferSize`, `OutputEmptyFcn`, `Status`, `Timeout`, `TransferStatus`, `ValuesSent`

Purpose

Find root of continuous function of one variable

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
[x,fval] = fzero(...)
[x,fval,exitflag] = fzero(...)
[x,fval,exitflag,output] = fzero(...)
```

Description

`x = fzero(fun,x0)` tries to find a zero of `fun` near `x0`, if `x0` is a scalar. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. The value `x` returned by `fzero` is near a point where `fun` changes sign, or `NaN` if the search fails. In this case, the search terminates when the search interval is expanded until an `Inf`, `NaN`, or complex value is found.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`. See also "Example 2" on page 2-1351 and "Example 3" on page 2-1351 below.

If `x0` is a vector of length two, `fzero` assumes `x0` is an interval where the sign of `fun(x0(1))` differs from the sign of `fun(x0(2))`. An error occurs if this is not true. Calling `fzero` with such an interval guarantees `fzero` will return a value near a point where `fun` changes sign.

`x = fzero(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fzero` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
FunValCheck	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` (the default) displays no error.
OutputFcn	User-defined function that is called at each iteration. See "Output Function" in the Optimization Toolbox for more information.
PlotFcns	User-defined plot function that is called at each iteration. See "Plot Functions" in the Optimization Toolbox for more information.
`TolX`	Termination tolerance on `x`

`[x,fval] = fzero(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fzero(...)` returns a value `exitflag` that describes the exit condition of `fzero`:

- `1` Function converged to a solution `x`.
- `-1` Algorithm was terminated by the output function.
- `-3` `NaN` or `Inf` function value was encountered during search for an interval containing a sign change.
- `-4` Complex function value was encountered during search for an interval containing a sign change.
- `-5` `fzero` might have converged to a singular point.
- `-6` `fzero` can not detect a change in sign of the function.

`[x,fval,exitflag,output] = fzero(...)` returns a structure `output` that contains information about the optimization:

`output.algorithm`	Algorithm used
`output.funcCount`	Number of function evaluations
`output.intervaliterations`	Number of iterations taken to find an interval
`output.iterations`	Number of zero-finding iterations
`output.message`	Exit message

Note For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the *x*-axis.

Arguments

`fun` is the function whose zero is to be computed. It accepts a vector `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for an M-file function

```
x = fzero(@myfun,x0);
```

where `myfun` is an M-file function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

or as a function handle for an anonymous function:

```
x = fzero(@(x)sin(x*x),x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Example 1

Calculate π by finding the zero of the sine function near 3.

```
x = fzero(@sin,3)
x =
    3.1416
```

Example 2

To find the zero of cosine between 1 and 2

```
x = fzero(@cos,[1 2])
x =
    1.5708
```

Note that `cos(1)` and `cos(2)` differ in sign.

Example 3

To find a zero of the function $f(x) = x^3 - 2x - 5$

write an anonymous function `f`:

```
f = @(x)x.^3-2*x-5;
```

Then find the zero near 2:

```
z = fzero(f,2)
z =
    2.0946
```

Because this function is a polynomial, the statement `roots([1 0 -2 -5])` finds the same real zero, and a complex conjugate pair of zeros.

```
    2.0946
   -1.0473 + 1.1359i
   -1.0473 - 1.1359i
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following M-file function.

```
function f = myfun(x,a)
f = cos(a*x);
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fzero`. To optimize for a specific value of `a`, such as `a = 2`.

1. Assign the value to `a`.

```
a = 2; % define parameter first
```

2. Call `fzero` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

```
x = fzero(@(x) myfun(x,a),0.1)
```

Algorithm

The `fzero` command is an M-file. The algorithm, which was originated by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which the `fzero` M-file is based, is in [2].

Limitations

The `fzero` command finds a point where the function changes sign. If the function is *continuous*, this is also a point where the function has a value near zero. If the function is not continuous, `fzero` may return values that are discontinuous points instead of zeros. For example, `fzero(@tan,1)` returns `1.5708`, a discontinuous point in `tan`.

Furthermore, the `fzero` command defines a *zero* as a point where the function crosses the *x*-axis. Points where the function touches, but does not cross, the *x*-axis are not valid zeros. For example, `y = x.^2` is a parabola that touches the *x*-axis at 0. Because the function never crosses the *x*-axis, however, no zero is found. For functions with no valid zeros, `fzero` executes until `Inf`, `NaN`, or a complex value is detected.

See Also

`roots`, `fminbnd`, `optimset`, `function_handle` (`@`), "Anonymous Functions"

References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

Purpose

Test matrices

Syntax

```
[A,B,C,...] = gallery(matname,P1,P2,...)
[A,B,C,...] = gallery(matname,P1,P2,...,classname)
gallery(3)
gallery(5)
```

Description

`[A,B,C,...] = gallery(matname,P1,P2,...)` returns the test matrices specified by the quoted string `matname`. The `matname` input is the name of a matrix family selected from the table below. `P1,P2,...` are input parameters required by the individual matrix family. The number of optional parameters `P1,P2,...` used in the calling syntax varies from matrix to matrix. The exact calling syntaxes are detailed in the individual matrix descriptions below.

`[A,B,C,...] = gallery(matname,P1,P2,...,classname)` produces a matrix of class `classname`. The `classname` input is a quoted string that must be either `'single'` or `'double'`. If `classname` is not specified, then the class of the matrix is determined from those arguments among `P1,P2,...` that do not specify dimensions or select an option. If any of these arguments is of class `single` then the matrix is `single`; otherwise the matrix is `double`.

`gallery(3)` is a badly conditioned 3-by-3 matrix and `gallery(5)` is an interesting eigenvalue problem.

The gallery holds over fifty different test matrix functions useful for testing algorithms and other purposes.

Test Matrices			
binomial	cauchy	chebspec	chebvand
chow	circul	clement	compar
condex	cycol	dorr	dramadah
fiedler	forsythe	frank	gearmat
gcdmat	grcar	hanowa	house

Test Matrices			
invhess	invol	ipjfact	jordbloc
kahan	kms	krylov	lauchli
lehmer	leslie	lesp	lotkin
minij	moler	neumann	orthog
parter	pei	poisson	prolate
randcolu	randcorr	randhess	randjorth
rando	randsvd	redheff	riemann
ris	smoke	toeppd	tridiag
triw	wathen	wilk	

binomial — Multiple of involutory matrix

`A = gallery('binomial',n)` returns an `n`-by-`n` matrix,with integer entries such that `A^2 = 2^(n-1)*eye(n)`.

Thus, `B = A*2^((1-n)/2)` is involutory, that is, `B^2 = eye(n)`.

cauchy — Cauchy matrix

`C = gallery('cauchy',x,y)` returns an `n`-by-`n` matrix, `C(i,j) = 1/(x(i)+y(j))`. Arguments `x` and `y` are vectors of length `n`. If you pass in scalars for `x` and `y`, they are interpreted as vectors `1:x` and `1:y`.

`C = gallery('cauchy',x)` returns the same as above with `y = x`. That is, the command returns `C(i,j) = 1/(x(i)+x(j))`.

Explicit formulas are known for the inverse and determinant of a Cauchy matrix. The determinant `det(C)` is nonzero if `x` and `y` both have distinct elements. `C` is totally positive if `0 < x(1) <... < x(n)` and `0 < y(1) < ... < y(n)`.

chebspec — Chebyshev spectral differentiation matrix

`C = gallery('chebspec',n,switch)` returns a Chebyshev spectral differentiation matrix of order `n`. Argument `switch` is a variable that determines the character of the output matrix. By default, `switch = 0`.

For `switch = 0` ("no boundary conditions"), `C` is nilpotent (`C^n = 0`) and has the null vector `ones(n,1)`. The matrix `C` is similar to a Jordan block of size `n` with eigenvalue zero.

For `switch = 1`, `C` is nonsingular and well-conditioned, and its eigenvalues have negative real parts.

The eigenvector matrix of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand — Vandermonde-like matrix for the Chebyshev polynomials

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where `m` is scalar, produces a rectangular version of the above, with `m` rows.

If `p` is a vector, then $C(i,j) = T_{i-1}(p(j))$ where T_{i-1} is the Chebyshev polynomial of degree i-1. If `p` is a scalar, then `p` equally spaced points on the interval `[0,1]` are used to calculate `C`.

chow — Singular Toeplitz lower Hessenberg matrix

`A = gallery('chow',n,alpha,delta)` returns `A` such that `A = H(alpha) + delta*eye(n)`, where $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$ and argument `n` is the order of the Chow matrix. Default value for scalars `alpha` and `delta` are `1` and `0`, respectively.

`H(alpha)` has `p = floor(n/2)` eigenvalues that are equal to zero. The rest of the eigenvalues are equal to `4*alpha*cos(k*pi/(n+2))^2, k=1:n-p`.

circul — Circulant matrix

`C = gallery('circul',v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals "wrap around."

If `v` is a scalar, then `C = gallery('circul',1:v)`.

The eigensystem of `C` (`n`-by-`n`) is known explicitly: If `t` is an `n`th root of unity, then the inner product of `v` and $w = [1\ t\ t^2 \ldots t^{(n-1)}]$ is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

clement — Tridiagonal matrix with zero diagonal entries

`A = gallery('clement',n,sym)` returns an `n`-by-`n` tridiagonal matrix with zeros on its main diagonal and known eigenvalues. It is singular if order `n` is odd. About 64 percent of the entries of the inverse are zero. The eigenvalues include plus and minus the numbers `n-1`, `n-3`, `n-5`, `...`, as well as (for odd `n`) a final eigenvalue of `1` or `0`.

Argument `sym` determines whether the Clement matrix is symmetric. For `sym = 0` (the default) the matrix is nonsymmetric, while for `sym = 1`, it is symmetric.

compar — Comparison matrices

`A = gallery('compar',A,1)` returns `A` with each diagonal element replaced by its absolute value, and each off-diagonal element replaced by minus the absolute value of the largest element in absolute value in its row. However, if `A` is triangular `compar(A,1)` is too.

`gallery('compar',A)` is `diag(B) - tril(B,-1) - triu(B,1)`, where `B = abs(A)`. `compar(A)` is often denoted by M(A) in the literature.

`gallery('compar',A,0)` is the same as `gallery('compar',A)`.

condex — Counter-examples to matrix condition number estimators

`A = gallery('condex',n,k,theta)` returns a "counter-example" matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k`:

`k = 1`	4-by-4	LINPACK
`k = 2`	3-by-3	LINPACK
`k = 3`	arbitrary	LINPACK (`rcond`) (independent of `theta`)
`k = 4`	`n >= 4`	LAPACK (RCOND) (default). It is the inverse of this matrix that is a counter-example.

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

cycol — Matrix whose columns repeat cyclically

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one "cycle" consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`, and `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol',[n n],k)`.

dorr — Diagonally dominant, ill-conditioned, tridiagonal matrix

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining an `n`-by-`n`, row diagonally dominant, tridiagonal matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is `0.01`. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

dramadah — Matrix of zeros and ones whose inverse has large integer entries

`A = gallery('dramadah',n,k)` returns an `n`-by-`n` matrix of `0`'s and `1`'s for which `mu(A) = norm(inv(A),'fro')` is relatively large, although not necessarily maximal. An anti-Hadamard matrix `A` is a matrix with elements `0` or `1` for which `mu(A)` is maximal.

`n` and `k` must both be scalars. Argument `k` determines the character of the output matrix:

k = 1	Default. `A` is Toeplitz, with `abs(det(A)) = 1`, and `mu(A) > c(1.75)^n`, where `c` is a constant. The inverse of `A` has integer entries.
k = 2	`A` is upper triangular and Toeplitz. The inverse of `A` has integer entries.
k = 3	`A` has maximal determinant among lower Hessenberg (0,1) matrices. `det(A)` = the `n`th Fibonacci number. `A` is Toeplitz. The eigenvalues have an interesting distribution in the complex plane.

fiedler — Symmetric matrix

`A = gallery('fiedler',c)`, where `c` is a length `n` vector, returns the `n`-by-`n` symmetric matrix with elements `abs(n(i)-n(j))`. For scalar `c`, `A = gallery('fiedler',1:c)`.

Matrix `A` has a dominant positive eigenvalue and all the other eigenvalues are negative.

Explicit formulas for `inv(A)` and `det(A)` are given in [Todd, J., *Basic Numerical Mathematics*, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159] and attributed to Fiedler. These indicate that `inv(A)` is tridiagonal except for nonzero `(1,n)` and `(n,1)` elements.

forsythe — Perturbed Jordan block

`A = gallery('forsythe',n,alpha,lambda)` returns the `n`-by-`n` matrix equal to the Jordan block with eigenvalue `lambda`, excepting that `A(n,1) = alpha`. The default values of scalars `alpha` and `lambda` are `sqrt(eps)` and `0`, respectively.

The characteristic polynomial of `A` is given by:

```
det(A-t*I) = (lambda-t)^N - alpha*(-1)^n.
```

frank — Matrix with ill-conditioned eigenvalues

`F = gallery('frank',n,k)` returns the Frank matrix of order `n`. It is upper Hessenberg with determinant `1`. If `k = 1`, the elements are reflected about the anti-diagonal `(1,n)` — `(n,1)`. The eigenvalues of `F` may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if `n` is odd, `1` is an eigenvalue. `F` has `floor(n/2)` ill-conditioned eigenvalues — the smaller ones.

gcdmat — Greatest common divisor matrix

`A = gallery('gcdmat',n)` returns the `n`-by-`n` matrix with `(i,j)` entry `gcd(i,j)`. `Matrix` A is symmetric positive definite, and A.^r is symmetric positive semidefinite for all nonnegative `r`.

gearmat — Gear matrix

`A = gallery('gearmat',n,i,j)` returns the `n`-by-`n` matrix with ones on the sub- and super-diagonals, `sign(i)` in the `(1,abs(i))` position, `sign(j)` in the `(n,n+1-abs(j))` position, and zeros everywhere else. Arguments `i` and `j` default to `n` and `-n`, respectively.

Matrix `A` is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form `2*cos(a)` and the eigenvectors are of the form `[sin(w+a), sin(w+2*a), ..., sin(w+n*a)]`, where `a` and `w` are given in Gear, C. W., "A Simple Set of Test Matrices for Eigenvalue Programs," *Math. Comp.*, Vol. 23 (1969), pp. 119-125.

grcar — Toeplitz matrix with sensitive eigenvalues

`A = gallery('grcar',n,k)` returns an `n`-by-`n` Toeplitz matrix with `-1`s on the subdiagonal, `1`s on the diagonal, and `k` superdiagonals of `1`s. The default is `k = 3`. The eigenvalues are sensitive.

hanowa — Matrix whose eigenvalues lie on a vertical line in the complex plane

`A = gallery('hanowa',n,d)` returns an `n`-by-`n` block 2-by-2 matrix of the form:

```
[d*eye(m)   -diag(1:m)
diag(1:m)   d*eye(m)]
```

Argument `n` is an even integer `n=2*m`. Matrix `A` has complex eigenvalues of the form `d ± k*i`, for `1 <= k <= m`. The default value of `d` is `-1`.

house — Householder matrix

`[v,beta,s] = gallery('house',x,k)` takes `x`, an `n`-element column vector, and returns `V` and `beta` such that `H*x = s*e1`. In this expression, `e1` is the first column of `eye(n)`, `abs(s) = norm(x)`, and `H = eye(n) - beta*V*V'` is a Householder matrix.

`k` determines the sign of `s`:

`k = 0`	`sign(s) = -sign(x(1))` (default)
`k = 1`	`sign(s) = sign(x(1))`
`k = 2`	`sign(s) = 1` (`x` must be real)

If `x` is complex, then `sign(x) = x./abs(x)` when `x` is nonzero.

If `x = 0`, or if `x = alpha*e1` (`alpha >= 0`) and either `k = 1` or `k = 2`, then `V = 0`, `beta = 1`, and `s = x(1)`. In this case, `H` is the identity matrix, which is not strictly a Householder matrix.

`[v, beta] = gallery('house',x)` takes `x`, a scalar or `n`-element column vector, and returns `v` and `beta` such that `eye(n,n) -`

`beta*v*v'` is a Householder matrix. A Householder matrix `H` satisfies the relationship

```
H*x = -sign(x(1))*norm(x)*e1
```

where `e1` is the first column of `eye(n,n)`. Note that if `x` is complex, then `sign(x) exp(i*arg(x))` (which equals `x./abs(x)` when `x` is nonzero).

If `x = 0`, then `v = 0` and `beta = 1`.

invhess — Inverse of an upper Hessenberg matrix

`A = gallery('invhess',x,y)`, where `x` is a length `n` vector and `y` is a length `n-1` vector, returns the matrix whose lower triangle agrees with that of `ones(n,1)*x'` and whose strict upper triangle agrees with that of `[1 y]*ones(1,n)`.

The matrix is nonsingular if `x(1) ~= 0` and `x(i+1) ~= y(i)` for all `i`, and its inverse is an upper Hessenberg matrix. Argument `y` defaults to `-x(1:n-1)`.

If `x` is a scalar, `invhess(x)` is the same as `invhess(1:x)`.

invol — Involutory matrix

`A = gallery('invol',n)` returns an `n`-by-`n` involutory (`A*A = eye(n)`) and ill-conditioned matrix. It is a diagonally scaled version of `hilb(n)`.

`B = (eye(n)-A)/2` and `B = (eye(n)+A)/2` are idempotent (`B*B = B`).

ipjfact — Hankel matrix with factorial elements

`[A,d] = gallery('ipjfact',n,k)` returns `A`, an `n`-by-`n` Hankel matrix, and `d`, the determinant of `A`, which is known explicitly. If `k = 0` (the default), then the elements of `A` are `A(i,j) = (i+j)!` If `k = 1`, then the elements of `A` are `A(i,j) 1/(i+j)`.

Note that the inverse of `A` is also known explicitly.

jordbloc — Jordan block

`A = gallery('jordbloc',n,lambda)` returns the `n`-by-`n` Jordan block with eigenvalue `lambda`. The default value for `lambda` is `1`.

kahan — Upper trapezoidal matrix

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is `n`-by-`n`. The useful range of `theta` is `0 < theta < pi`, with a default value of `1.2`.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is `25`, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

kms — Kac-Murdock-Szego Toeplitz matrix

`A = gallery('kms',n,rho)` returns the `n`-by-`n` Kac-Murdock-Szego Toeplitz matrix such that `A(i,j) = rho^(abs(i-j))`, for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L inv(gallery('triw',n,-rho,1))'`, and `D(i,i) (1-abs(rho)^2)*eye(n)`, except `D(1,1) = 1`.
- Positive definite if and only if `0 < abs(rho) < 1`.
- The inverse `inv(A)` is tridiagonal.

krylov — Krylov matrix

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

```
[x, Ax, A^2x, ..., A^(j-1)x]
```

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',(randn(n))`.

lauchli — Rectangular matrix

`A = gallery('lauchli',n,mu)` returns the `(n+1)`-by-`n` matrix

```
[ones(1,n); mu*eye(n)]
```

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming `A'*A`. Argument `mu` defaults to `sqrt(eps)`.

lehmer — Symmetric positive definite matrix

`A = gallery('lehmer',n)` returns the symmetric positive definite `n`-by-`n` matrix such that `A(i,j) = i/j` for `j >= i`.

The Lehmer matrix `A` has these properties:

- `A` is totally nonnegative.
- The inverse `inv(A)` is tridiagonal and explicitly known.
- The order `n <= cond(A) <= 4*n*n`.

leslie —

`L = gallery('leslie',a,b)` is the `n`-by-`n` matrix from the Leslie population model with average birth numbers `a(1:n)` and survival rates `b(1:n-1)`. It is zero, apart from the first row (which contains the `a(i)`) and the first subdiagonal (which contains the `b(i)`). For a valid model, the `a(i)` are nonnegative and the `b(i)` are positive and bounded by 1, i.e., `0 < b(i) <= 1`.

`L = gallery('leslie',n)` generates the Leslie matrix with `a = ones(n,1)`, `b = ones(n-1,1)`.

lesp — Tridiagonal matrix with real, sensitive eigenvalues

`A = gallery('lesp',n)` returns an `n`-by-`n` matrix whose eigenvalues are real and smoothly distributed in the interval approximately `[-2*N-3.5, -4.5]`.

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with `D = diag(1!,2!,...,n!)`.

lotkin — Lotkin matrix

`A = gallery('lotkin',n)` returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix `A` is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

minij — Symmetric positive definite matrix

`A = gallery('minij',n)` returns the `n`-by-`n` symmetric positive definite matrix with `A(i,j) = min(i,j)`.

The `minij` matrix has these properties:

- The inverse `inv(A)` is tridiagonal and equal to `-1` times the second difference matrix, except its `(n,n)` element is `1`.
- Givens' matrix, `2*A-ones(size(A))`, has tridiagonal inverse and eigenvalues `0.5*sec((2*r-1)*pi/(4*n))^2`, where `r=1:n`.
- `(n+1)*ones(size(A))-A` has elements that are `max(i,j)` and a tridiagonal inverse.

moler — Symmetric positive definite matrix

`A = gallery('moler',n,alpha)` returns the symmetric positive definite `n`-by-`n` matrix `U'*U`, where `U = gallery('triw',n,alpha)`.

For the default `alpha = -1`, `A(i,j) = min(i,j)-2`, and `A(i,i) = i`. One of the eigenvalues of `A` is small.

neumann — Singular matrix from the discrete Neumann problem (sparse)

`C = gallery('neumann',n)` returns the sparse `n`-by-`n` singular, row diagonally dominant matrix resulting from discretizing the Neumann problem with the usual five-point operator on a regular mesh.

Argument `n` is a perfect square integer $n = m^2$ or a two-element vector. `C` is sparse and has a one-dimensional null space with null vector `ones(n,1)`.

orthog — Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the kth type of matrix of order `n`, where `k > 0` selects exactly orthogonal matrices, and `k < 0` selects diagonal scalings of orthogonal matrices. Available types are:

`k = 1` `Q(i,j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))`

Symmetric eigenvector matrix for second difference matrix. This is the default.

`k = 2` `Q(i,j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))`

Symmetric.

`k = 3` `Q(r,s) = exp(2*pi*i*(r-1)*(s-1)/n) / sqrt(n)`

Unitary, the Fourier matrix. Q^4 is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)!`

`k = 4` Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.

`k = 5` `Q(i,j) = sin(2*pi*(i-1)*(j-1)/n) + cos(2*pi*(i-1)*(j-1)/n)`

Symmetric matrix arising in the Hartley transform.

`k = 6` `Q(i,j) = sqrt(2/n)*cos((i-1/2)*(j-1/2)*pi/n)`

Symmetric matrix arising as a discrete cosine transform.

`k = -1`	`Q(i,j) = cos((i-1)*(j-1)*pi/(n-1))` Chebyshev Vandermonde-like matrix, based on extrema of `T(n-1)`.
`k = -2`	`Q(i,j) = cos((i-1)*(j-1/2)*pi/n))` Chebyshev Vandermonde-like matrix, based on zeros of `T(n)`.

parter — Toeplitz matrix with singular values near pi

`C = gallery('parter',n)` returns the matrix `C` such that `C(i,j) = 1/(i-j+0.5)`.

`C` is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of `C` are very close to `pi`.

pei — Pei matrix

`A = gallery('pei',n,alpha)`, where `alpha` is a scalar, returns the symmetric matrix `alpha*eye(n) + ones(n)`. The default for `alpha` is 1. The matrix is singular for `alpha` equal to either `0` or `-n`.

poisson — Block tridiagonal matrix from Poisson's equation (sparse)

`A = gallery('poisson',n)` returns the block tridiagonal (sparse) matrix of order `n^2` resulting from discretizing Poisson's equation with the 5-point operator on an `n`-by-`n` mesh.

prolate — Symmetric, ill-conditioned Toeplitz matrix

`A = gallery('prolate',n,w)` returns the `n`-by-`n` prolate matrix with parameter `w`. It is a symmetric Toeplitz matrix.

If `0 < w < 0.5` then `A` is positive definite

- The eigenvalues of `A` are distinct, lie in `(0,1)`, and tend to cluster around `0` and `1`.
- The default value of `w` is 0.25.

randcolu — Random matrix with normalized cols and specified singular values

`A = gallery('randcolu',n)` is a random `n`-by-`n` matrix with columns of unit 2-norm, with random singular values whose squares are from a uniform distribution.

`A'*A` is a correlation matrix of the form produced by `gallery('randcorr',n)`.

`gallery('randcolu',x)` where `x` is an `n`-vector (`n > 1`), produces a random `n`-by-`n` matrix having singular values given by the vector `x`. The vector `x` must have nonnegative elements whose sum of squares is `n`.

`gallery('randcolu',x,m)` where `m >= n`, produces an `m`-by-`n` matrix.

`gallery('randcolu',x,m,k)` provides a further option:

`k = 0`	`diag(x)` is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have zero entries.

For more information, see:

[1] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

randcorr — Random correlation matrix with specified eigenvalues

`gallery('randcorr',n)` is a random `n`-by-`n` correlation matrix with random eigenvalues from a uniform distribution. A correlation matrix is a symmetric positive semidefinite matrix with `1`s on the diagonal (see `corrcoef`).

`gallery('randcorr',x)` produces a random correlation matrix having eigenvalues given by the vector `x`, where `length(x) > 1`. The vector `x` must have nonnegative elements summing to `length(x)`.

`gallery('randcorr',x,k)` provides a further option:

`k = 0`	The diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have some zero entries.

For more information, see:

[1] Bendel, R. B. and M. R. Mickey, "Population Correlation Matrices for Sampling Experiments," *Commun. Statist. Simulation Comput.*, B7, 1978, pp. 163-182.

[2] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

randhess — Random, orthogonal upper Hessenberg matrix

`H = gallery('randhess',n)` returns an `n`-by-`n` real, random, orthogonal upper Hessenberg matrix.

`H = gallery('randhess',x)` if `x` is an arbitrary, real, length `n` vector with `n > 1`, constructs `H` nonrandomly using the elements of `x` as parameters.

Matrix `H` is constructed via a product of `n-1` Givens rotations.

randjorth — Random J-orthogonal matrix

`A = gallery('randjorth', n)`, for a positive integer `n`, produces a random `n`-by-`n` J-orthogonal matrix `A`, where

- `J = blkdiag(eye(ceil(n/2)),-eye(floor(n/2)))`
- `cond(A) = sqrt(1/eps)`

J-orthogonality means that A'*J*A = J. Such matrices are sometimes called *hyperbolic*.

`A = gallery('randjorth', n, m)`, for positive integers `n` and `m`, produces a random (`n+m`)-by-(`n+m`) J-orthogonal matrix `A`, where

- `J = blkdiag(eye(n),-eye(m))`
- `cond(A) = sqrt(1/eps)`

`A = gallery('randjorth',n,m,c,symm,method)`

uses the following optional input arguments:

- `c` — Specifies `cond(A)` to be the scalar `c`.
- `symm` — Enforces symmetry if the scalar `symm` is nonzero.
- `method` — calls `qr` to perform the underlying orthogonal transformations if the scalar `method` is nonzero. A call to `qr` is much faster than the default method for large dimensions

rando — Random matrix composed of elements -1, 0 or 1

`A = gallery('rando',n,k)` returns a random `n`-by-`n` matrix with elements from one of the following discrete distributions:

`k = 1`	`A(i,j)` = 0 or 1 with equal probability (default).
`k = 2`	`A(i,j)` = -1 or 1 with equal probability.
`k = 3`	`A(i,j)` = -1, 0 or 1 with equal probability.

Argument `n` may be a two-element vector, in which case the matrix is `n(1)`-by-`n(2)`.

randsvd — Random matrix with preassigned singular values

`A = gallery('randsvd',n,kappa,mode,kl,ku)` returns a banded (multidiagonal) random matrix of order `n` with `cond(A) = kappa` and singular values from the distribution mode. If `n` is a two-element vector, `A` is `n(1)`-by-`n(2)`.

Arguments `kl` and `ku` specify the number of lower and upper off-diagonals, respectively, in `A`. If they are omitted, a full matrix is produced. If only `kl` is present, `ku` defaults to `kl`.

Distribution `mode` can be:

`1`	One large singular value.
`2`	One small singular value.
`3`	Geometrically distributed singular values (default).
`4`	Arithmetically distributed singular values.
`5`	Random singular values with uniformly distributed logarithm.
`< 0`	If `mode` is `-1`, `-2`, `-3`, `-4`, or `-5`, then `randsvd` treats `mode` as `abs(mode)`, except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number `kappa` defaults to `sqrt(1/eps)`. In the special case where `kappa < 0`, `A` is a random, full, symmetric, positive definite matrix with `cond(A) = -kappa` and eigenvalues distributed according to `mode`. Arguments `kl` and `ku`, if present, are ignored.

`A = gallery('randsvd',n,kappa,mode,kl,ku,method)` specifies how the computations are carried out. `method = 0` is the default, while `method = 1` uses an alternative method that is much faster for large dimensions, even though it uses more flops.

redheff — Redheffer's matrix of 1s and 0s

`A = gallery('redheff',n)` returns an `n`-by-`n` matrix of 0's and 1's defined by `A(i,j) = 1`, if `j = 1` or if `i` divides `j`, and `A(i,j) = 0` otherwise.

The Redheffer matrix has these properties:

- `(n-floor(log2(n)))-1` eigenvalues equal to `1`
- A real eigenvalue (the spectral radius) approximately `sqrt(n)`
- A negative eigenvalue approximately `-sqrt(n)`
- The remaining eigenvalues are provably “small.”
- The Riemann hypothesis is true if and only if $\det(A) = O(n^{\frac{1}{2}+\varepsilon})$ for every epsilon `> 0`.

Barrett and Jarvis conjecture that “the small eigenvalues all lie inside the unit circle `abs(Z) = 1`,” and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as `n` tends to infinity, would yield a new proof of the prime number theorem.

riemann — Matrix associated with the Riemann hypothesis

`A = gallery('riemann',n)` returns an `n`-by-`n` matrix for which the Riemann hypothesis is true if and only if

$$\det(A) = O(n!n^{-\frac{1}{2}+\varepsilon})$$

for every $\varepsilon > 0$.

The Riemann matrix is defined by:

```
A = B(2:n+1,2:n+1)
```

where `B(i,j) = i-1` if `i` divides `j`, and `B(i,j) = -1` otherwise.

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `abs(e(i)) <= m-1/m`, where `m = n+1`.
- `i <= e(i) <= i+1` with at most `m-sqrt(m)` exceptions.
- All integers in the interval `(m/3, m/2]` are eigenvalues.

ris — Symmetric Hankel matrix

`A = gallery('ris',n)` returns a symmetric `n`-by-`n` Hankel matrix with elements

```
A(i,j) = 0.5/(n-i-j+1.5)
```

The eigenvalues of `A` cluster around $\pi/2$ and $-\pi/2$. This matrix was invented by F.N. Ris.

smoke — Complex matrix with a 'smoke ring' pseudospectrum

`A = gallery('smoke',n)` returns an `n`-by-`n` matrix with 1's on the superdiagonal, `1` in the `(n,1)` position, and powers of roots of unity along the diagonal.

`A = gallery('smoke',n,1)` returns the same except that element `A(n,1)` is zero.

The eigenvalues of `gallery('smoke',n,1)` are the `n`th roots of unity; those of `gallery('smoke',n)` are the `n`th roots of unity times `2^(1/n)`.

toeppd — Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd',n,m,w,theta)` returns an `n`-by-`n` symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of `m` rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

```
T = w(1)*T(theta(1)) + ... + w(m)*T(theta(m))
```

where `T(theta(k))` has `(i,j)` element `cos(2*pi*theta(k)*(i-j))`.

By default: `m = n`, `w = rand(m,1)`, and `theta = rand(m,1)`.

toeppen — Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen',n,a,b,c,d,e)` returns the `n`-by-`n` sparse, pentadiagonal Toeplitz matrix with the diagonals: `P(3,1) = a`, `P(2,1) = b`, `P(1,1) = c`, `P(1,2) = d`, and `P(1,3) = e`, where `a`, `b`, `c`, `d`, and `e` are scalars.

By default, `(a,b,c,d,e) = (1,-10,0,10,1)`, yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment `2*cos(2*t) + 20*i*sin(t)`.

tridiag — Tridiagonal matrix (sparse)

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)-1`.

`A = gallery('tridiag',n,c,d,e)`, where `c`, `d`, and `e` are all scalars, yields the Toeplitz tridiagonal matrix of order `n` with subdiagonal elements `c`, diagonal elements `d`, and superdiagonal elements `e`. This matrix has eigenvalues

```
d + 2*sqrt(c*e)*cos(k*pi/(n+1))
```

where `k = 1:n`. (see [1].)

`A = gallery('tridiag',n)` is the same as `A = gallery('tridiag',n,-1,2,-1)`, which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

triw — Upper triangular matrix discussed by Wilkinson and others

`A = gallery('triw',n,alpha,k)` returns the upper triangular matrix with ones on the diagonal and `alpha`s on the first `k >= 0` superdiagonals.

Order `n` may be a 2-element vector, in which case the matrix is `n(1)`-by-`n(2)` and upper trapezoidal.

Ostrowski ["On the Spectrum of a One-parametric Family of Matrices," *J. Reine Angew. Math.,* 1954] shows that

```
cond(gallery('triw',n,2)) = cot(pi/(4*n))^2,
```

and, for large `abs(alpha)`, `cond(gallery('triw',n,alpha))` is approximately `abs(alpha)^n*sin(pi/(4*n-2))`.

Adding `-2^(2-n)` to the `(n,1)` element makes `triw(n)` singular, as does adding `-2^(1-n)` to all the elements in the first column.

wathen — Finite element matrix (sparse, random entries)

`A = gallery('wathen',nx,ny)` returns a sparse, random, `n`-by-`n` finite element matrix where `n = 3*nx*ny + 2*nx + 2*ny + 1`.

Matrix `A` is precisely the "consistent mass matrix" for a regular `nx`-by-`ny` grid of 8-node (serendipity) elements in two dimensions. `A` is symmetric, positive definite for any (positive) values of the "density," `rho(nx,ny)`, which is chosen randomly in this routine.

`A = gallery('wathen',nx,ny,1)` returns a diagonally scaled matrix such that

```
0.25 <= eig(inv(D)*A) <= 4.5
```

where `D = diag(diag(A))` for any positive integers `nx` and `ny` and any densities `rho(nx,ny)`.

wilk — Various matrices devised or discussed by Wilkinson

`[A,b] = gallery('wilk',n)` returns a different matrix or linear system depending on the value of `n`.

`n = 3`	Upper triangular system `Ux=b` illustrating inaccurate solution.
`n = 4`	Lower triangular system `Lx=b`, ill-conditioned.
`n = 5`	`hilb(6)(1:5,2:6)*1.8144`. A symmetric positive definite matrix.
`n = 21`	`W21+`, a tridiagonal matrix. eigenvalue problem. For more detail, see [2].

See Also

`hadamard`, `hilb`, `invhilb`, `magic`, `wilkinson`

References

[1] The MATLAB gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics,

University of Manchester, Manchester, England. Additional detail on these matrices is documented in *The Test Matrix Toolbox for MATLAB* by N. J. Higham, September, 1995. This report is available via anonymous ftp from The MathWorks at `http://www.mathworks.com/access/pub/testmatrix.ps` or on the Web at `ftp://ftp.ma.man.ac.uk/pub/narep`. Further background can be found in the book *Accuracy and Stability of Numerical Algorithms,* Nicholas J. Higham, SIAM, 1996.

[2] Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965, p.308.

Purpose

Gamma functions

Syntax

```
Y = gamma(A)
Y = gammainc(X,A)
Y = gammainc(X,A,tail)
Y = gammaln(A)
```

Definition

The gamma function is defined by the integral:

$$\Gamma(a) = \int_0^\infty e^{-t}\, t^{a-1} dt$$

The gamma function interpolates the `factorial` function. For integer `n`:

```
gamma(n+1) = n! = prod(1:n)
```

The incomplete gamma function is:

$$P(x,a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

For any `a>=0`, `gammainc(x,a)` approaches 1 as `x` approaches `infinity`. For small `x` and `a`, `gammainc(x,a)` is approximately equal to `x^a`, so `gammainc(0,0) = 1`.

Description

`Y = gamma(A)` returns the gamma function at the elements of `A`. `A` must be real.

`Y = gammainc(X,A)` returns the incomplete gamma function of corresponding elements of `X` and `A`. Arguments `X` and `A` must be real and the same size (or either can be scalar).

`Y = gammainc(X,A,tail)` specifies the tail of the incomplete gamma function when `X` is non-negative. The choices for `tail` are `'lower'` (the default) and `'upper'`. The upper incomplete gamma function is defined as

```
1 - gammainc(x,a)
```

Note When `X` is negative, `Y` can be inaccurate for `abs(X)>A+1`.

`Y = gammaln(A)` returns the logarithm of the `gamma` function, `gammaln(A) = log(gamma(A))`. The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using `log(gamma(A))`.

Algorithm

The computations of `gamma` and `gammaln` are based on algorithms outlined in [1]. Several different minimax rational approximations are used depending upon the value of `A`. Computation of the incomplete `gamma` function is based on the algorithm in [2].

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

Purpose

Current axes handle

Syntax

```
h = gca
```

Description

`h = gca` returns the handle to the current axes for the current figure. If no axes exists, MATLAB creates one and returns its handle. You can use the statement

```
get(gcf,'CurrentAxes')
```

if you do not want MATLAB to create an axes if one does not already exist.

Current Axes

The current axes is the target for graphics output when you create axes children. The current axes is typically the last axes used for plotting or the last axes clicked on by the mouse. Graphics commands such as `plot`, `text`, and `surf` draw their results in the current axes. Changing the current figure also changes the current axes.

See Also

`axes`, `cla`, `gcf`, `findobj`

figure `CurrentAxes` property

"Finding and Identifying Graphics Objects" on page 1-93 for related functions

gcbf

Purpose

Handle of figure containing object whose callback is executing

Syntax

```
fig = gcbf
```

Description

`fig = gcbf` returns the handle of the figure that contains the object whose callback is currently executing. This object can be the figure itself, in which case, `gcbf` returns the figure's handle.

When no callback is executing, `gcbf` returns the empty matrix, `[]`.

The value returned by `gcbf` is identical to the `figure` output argument returned by `gcbo`.

See Also

`gcbo`, `gco`, `gcf`, `gca`

Purpose

Handle of object whose callback is executing

Syntax

```
h = gcbo
[h,figure] = gcbo
```

Description

`h = gcbo` returns the handle of the graphics object whose callback is executing.

`[h,figure] = gcbo` returns the handle of the current callback object and the handle of the figure containing this object.

Remarks

MATLAB stores the handle of the object whose callback is executing in the root `CallbackObject` property. If a callback interrupts another callback, MATLAB replaces the `CallbackObject` value with the handle of the object whose callback is interrupting. When that callback completes, MATLAB restores the handle of the object whose callback was interrupted.

The root `CallbackObject` property is read only, so its value is always valid at any time during callback execution. The root `CurrentFigure` property, and the figure `CurrentAxes` and `CurrentObject` properties (returned by `gcf`, `gca`, and `gco`, respectively) are user settable, so they can change during the execution of a callback, especially if that callback is interrupted by another callback. Therefore, those functions are not reliable indicators of which object's callback is executing.

When you write callback routines for the `CreateFcn` and `DeleteFcn` of any object and the figure `ResizeFcn`, you must use `gcbo` since those callbacks do not update the root's `CurrentFigure` property, or the figure's `CurrentObject` or `CurrentAxes` properties; they only update the root's `CallbackObject` property.

When no callbacks are executing, `gcbo` returns `[]` (an empty matrix).

See Also

`gca`, `gcf`, `gco`, `rootobject`

"Finding and Identifying Graphics Objects" on page 1-93 for related functions.

Purpose

Greatest common divisor

Syntax

```
G = gcd(A,B)
[G,C,D] = gcd(A,B)
```

Description

`G = gcd(A,B)` returns an array containing the greatest common divisors of the corresponding elements of integer arrays `A` and `B`. By convention, `gcd(0,0)` returns a value of `0`; all other inputs return positive integers for `G`.

`[G,C,D] = gcd(A,B)` returns both the greatest common divisor array `G`, and the arrays `C` and `D`, which satisfy the equation: `A(i).*C(i) + B(i).*D(i) = G(i)`. These are useful for solving Diophantine equations and computing elementary Hermite transformations.

Examples

The first example involves elementary Hermite transformations.

For any two integers `a` and `b` there is a `2`-by-`2` matrix `E` with integer entries and determinant = `1` (a *unimodular* matrix) such that:

```
E * [a;b] = [g,0],
```

where `g` is the greatest common divisor of `a` and `b` as returned by the command `[g,c,d] = gcd(a,b)`.

The matrix `E` equals:

```
c       d
-b/g    a/g
```

In the case where `a = 2` and `b = 4`:

```
[g,c,d] = gcd(2,4)
g =
     2
c =
     1
d =
     0
```

So that

```
E =
     1     0
    -2     1
```

In the next example, we solve for x and y in the Diophantine equation 30x + 56y = 8.

```
[g,c,d] = gcd(30,56)
g =
     2
c =
    -13
d =
     7
```

By the definition, for scalars c and d:

```
30(-13) + 56(7) = 2,
```

Multiplying through by 8/2:

```
30(-13*4) + 56(7*4) = 8
```

Comparing this to the original equation, a solution can be read by inspection:

```
x = (-13*4) = -52; y = (7*4) = 28
```

See Also

lcm

References

[1] Knuth, Donald, *The Art of Computer Programming*, Vol. 2, Addison-Wesley: Reading MA, 1973. Section 4.5.2, Algorithm X.

gcf

Purpose

Current figure handle

Syntax

```
h = gcf
```

Description

`h = gcf` returns the handle of the current figure. The current figure is the figure window in which graphics commands such as `plot`, `title`, and `surf` draw their results. If no figure exists, MATLAB creates one and returns its handle. You can use the statement

```
get(0,'CurrentFigure')
```

if you do not want MATLAB to create a figure if one does not already exist.

See Also

`clf`, `figure`, `gca`

Root `CurrentFigure` property

"Finding and Identifying Graphics Objects" on page 1-93 for related functions

Purpose

Handle of current object

Syntax

```
h = gco
h = gco(figure_handle)
```

Description

`h = gco` *returns* the handle of the current object.

`h = gco(figure_handle)` returns the value of the current object for the figure specified by `figure_handle`.

Remarks

The current object is the last object clicked on, excluding `uimenus`. If the mouse click did not occur over a figure child object, the figure becomes the current object. MATLAB stores the handle of the current object in the figure's `CurrentObject` property.

The `CurrentObject` of the `CurrentFigure` does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the `CurrentObject` or even the `CurrentFigure`. Some callbacks, such as `CreateFcn` and `DeleteFcn`, and uimenu `Callback`, intentionally do not update `CurrentFigure` or `CurrentObject`.

`gcbo` provides the only completely reliable way to retrieve the handle to the object whose callback is executing, at any point in the `callback` function, regardless of the type of callback or of any previous interruptions.

Examples

This statement returns the handle to the current object in figure window 2:

```
h = gco(2)
```

See Also

`gca`, `gcbo`, `gcf`

The `root` object description

"Finding and Identifying Graphics Objects" on page 1-93 for related functions

Purpose

Test for greater than or equal to

Syntax

```
A >= B
ge(A, B)
```

Description

`A >= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is greater than or equal to `B`, or set to logical 0 (`false`) where `A` is less than `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then MATLAB returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`ge(A, B)` is called for the syntax `A >= B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are greater than or equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A >= B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     1     1
     0     1     1     0     1     0
```

```
     1     0     1     1     0     0
     0     1     1     1     0     1
```

See Also

`gt`, `eq`, `le`, `lt`, `ne`, “Relational Operators”

genpath

Purpose

Generate path string

Syntax

```
genpath
genpath directory
p = genpath('directory')
```

Description

`genpath` returns a path string formed by recursively adding all the directories below `matlabroot/toolbox`.

`genpath directory` returns a path string formed by recursively adding all the directories below `directory`. This path string does not include directories named private or directories that begin with the character `@`.

`p = genpath('directory')` returns the path string to variable, `p`.

Examples

You generate a path that includes `matlabroot/toolbox/images` and all directories below that with the following command:

```
p = genpath(fullfile(matlabroot,'toolbox','images'))
```

```
p =

   matlabroot\toolbox\images;matlabroot\toolbox\images\
   images;matlabroot\toolbox\images\images\ja;
   matlabroot\toolbox\images\imdemos;matlabroot\
   toolbox\images\imdemos\ja;
```

You can also use `genpath` in conjunction with `addpath` to add subdirectories to the path from the command line. The following example adds the `/control` directory and its subdirectories to the current path.

```
%  Display the current path
path

                MATLABPATH

      K:\toolbox\matlab\general
      K:\toolbox\matlab\ops
```

```
      K:\toolbox\matlab\lang
      K:\toolbox\matlab\elmat
      K:\toolbox\matlab\elfun
                :
                :
                :

%  Use GENPATH to add /control and its subdirectories
addpath(genpath('K:/toolbox/control'))


%  Display the new path
path

             MATLABPATH

      K:\toolbox\control
      K:\toolbox\control\ctrlutil
      K:\toolbox\control\control
      K:\toolbox\control\ctrlguis
      K:\toolbox\control\ctrldemos
      K:\toolbox\matlab\general
      K:\toolbox\matlab\ops
      K:\toolbox\matlab\lang
      K:\toolbox\matlab\elmat
      K:\toolbox\matlab\elfun
                :
                :
                :
```

See Also

`addpath`, `path`, `pathdef`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`

"Search Path" in the MATLAB Desktop Tools and Development Environment documentation

genvarname

Purpose

Construct valid variable name from string

Syntax

```
varname = genvarname(str)
varname = genvarname(str, exclusions)
```

Description

`varname = genvarname(str)` constructs a string `varname` that is similar to or the same as the `str` input, and can be used as a valid variable name. `str` can be a single character array or a cell array of strings. If `str` is a cell array of strings, `genvarname` returns a cell array of strings in `varname`. The strings in a cell array returned by `genvarname` are guaranteed to be different from each other.

`varname = genvarname(str, exclusions)` returns a valid variable name that is different from any name listed in the `exclusions` input. The `exclusions` input can be a single character array or a cell array of strings. Specify the function `who` in the `exclusions` character array to create a variable name that will be unique in the current MATLAB workapace (see “Example 4” on page 2-1392, below).

Note `genvarname` returns a string that can be used as a variable name. It does not create a variable in the MATLAB workspace. You cannot, therefore, assign a value to the output of `genvarname`.

Remarks

A valid MATLAB variable name is a character string of letters, digits, and underscores, such that the first character is a letter, and the length of the string is less than or equal to the value returned by the `namelengthmax` function. Any string that exceeds `namelengthmax` is truncated in the `varname` output. See “Example 6” on page 2-1393, below.

The variable name returned by `genvarname` is not guaranteed to be different from other variable names currently in the MATLAB workspace unless you use the `exclusions` input in the manner shown in “Example 4” on page 2-1392, below.

If you use genvarname to generate a field name for a structure, MATLAB does create a variable for the structure and field in the MATLAB workspace. See “Example 3” on page 2-1391, below.

If the str input contains any whitespace characters, genvarname removes then and capitalizes the next alphabetic character in str. If str contains any nonalphanumeric characters, genvarname translates these characters into their hexadecimal value.

Examples

Example 1

Create four similar variable name strings that do not conflict with each other:

```
v = genvarname({'A', 'A', 'A', 'A'})
v =
    'A'    'A1'    'A2'    'A3'
```

Example 2

Read a column header hdr from worksheet trial2 in Excel spreadsheet myproj_apr23:

```
[data hdr] = xlsread('myproj_apr23.xls', 'trial2');
```

Make a variable name from the text of the column header that will not conflict with other names:

```
v = genvarname(['Column ' hdr{1,3}]);
```

Assign data taken from the spreadsheet to the variable in the MATLAB workspace:

```
eval([v '= data(1:7, 3);']);
```

Example 3

Collect readings from an instrument once every minute over the period of an hour into different fields of a structure. genvarname not only generates unique fieldname strings, but also creates the structure and fields in the MATLAB workspace:

```
for k = 1:60
record.(genvarname(['reading' datestr(clock, 'HHMMSS')])) = takeReading;
pause(60)
end
```

After the program ends, display the recorded data from the workspace:

```
record
record =
    reading090446: 27.3960
    reading090546: 23.4890
    reading090646: 21.1140
    reading090746: 23.0730
    reading090846: 28.5650
          .
          .
          .
```

Example 4

Generate variable names that are unique in the MATLAB workspace by putting the output from the `who` function in the `exclusions` list.

```
for k = 1:5
   t = clock;
   pause(uint8(rand * 10));
   v = genvarname('time_elapsed', who);
   eval([v ' = etime(clock,t)'])
   end
```

As this code runs, you can see that the variables created by `genvarname` are unique in the workspace:

```
time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010
```

```
time_elapsed3 =
    8.0010
time_elapsed4 =
    3.0040
```

After the program completes, use the `who` function to view the workspace variables:

```
who

k           time_elapsed   time_elapsed2  time_elapsed4
t           time_elapsed1  time_elapsed3  v
```

Example 5

If you try to make a variable name from a MATLAB keyword, `genvarname` creates a variable name string that capitalizes the keyword and precedes it with the letter `x`:

```
v = genvarname('global')
v =
   xGlobal
```

Example 6

If you enter a string that is longer than the value returned by the `namelengthmax` function, `genvarname` truncates the resulting variable name string:

```
namelengthmax
ans =
    63

vstr = genvarname(sprintf('%s%s', ...
   'This name truncates because it contains ', ...
   'more than the maximum number of characters'))
vstr =
ThisNameTruncatesBecauseItContainsMoreThanTheMaximumNumberOfCha
```

See Also

`isvarname`, `iskeyword`, `isletter`, `namelengthmax`, `who`, `regexp`

Purpose

Query object properties

Syntax

```
get(h)
get(h,'PropertyName')
<m-by-n value cell array> = get(H,pn)
a = get(h)
a = get(0,'Factory')
a = get(0,'FactoryObjectTypePropertyName')
a = get(h,'Default')
a = get(h,'DefaultObjectTypePropertyName')
```

Description

`get(h)` returns all properties of the graphics object identified by the handle `h` and their current values.

`get(h,'PropertyName')` returns the value of the property `'PropertyName'` of the graphics object identified by `h`.

`<m-by-n value cell array> = get(H,pn)` returns n property values for m graphics objects in the m-by-n cell array, where `m = length(H)` and n is equal to the number of property names contained in `pn`.

`a = get(h)` returns a structure whose field names are the object's property names and whose values are the current values of the corresponding properties. `h` must be a scalar. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'Factory')` returns the factory-defined values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'FactoryObjectTypePropertyName')` returns the factory-defined value of the named property for the specified object type. The argument *`FactoryObjectTypePropertyName`* is the word `Factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

`FactoryFigureColor` `a = get(h,'Default')` returns all default values currently defined on object `h`. `a` is a structure array whose field names

are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(h,'Default`*`ObjectTypePropertyName`*`')` returns the factory-defined value of the named property for the specified object type. The argument *`DefaultObjectTypePropertyName`* is the word `Default` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

```
DefaultFigureColor
```

Examples

You can obtain the default value of the `LineWidth` property for line graphics objects defined on the root level with the statement

```
get(0,'DefaultLineLineWidth')
ans =
  0.5000
```

To query a set of properties on all axes children, define a cell array of property names:

```
props = {'HandleVisibility', 'Interruptible';
  'SelectionHighlight', 'Type'};
output = get(get(gca,'Children'),props);
```

The variable `output` is a cell array of dimension `length(get(gca,'Children')`-by-4.

For example, type

```
patch;surface;text;line
output = get(get(gca,'Children'),props)
output =
 'on'    'on'    'on'    'line'
 'on'    'off'   'on'    'text'
 'on'    'on'    'on'    'surface'
 'on'    'on'    'on'    'patch'
```

See Also

`findobj`, `gca`, `gcf`, `gco`, `set`

Handle Graphics Properties

“Finding and Identifying Graphics Objects” on page 1-93 for related functions

Purpose

Get property value from interface, or display properties

Syntax

```
V = h.get
V = h.get('propertyname')
V = get(h, ...)
```

Description

`V = h.get` returns a list of all properties and their values for the object or interface, `h`.

If `V` is empty, either there are no properties in the object, or MATLAB cannot read the object's type library. Refer to the COM vendor's documentation. For Automation objects, if the vendor provides documentation for a specific property, use the `V = get(h, ...)` syntax to call it.

`V = h.get('propertyname')` returns the value of the property specified in the string, `propertyname`.

`V = get(h, ...)` is an alternate syntax for the same operation.

Remarks

The meaning and type of the return value is dependent upon the specific property being retrieved. The object's documentation should describe the specific meaning of the return value. MATLAB may convert the data type of the return value. See "Handling COM Data in MATLAB" in the External Interfaces documentation for a description of how MATLAB converts COM data types.

Examples

Create a COM server running Microsoft Excel:

```
e = actxserver ('Excel.Application');
```

Retrieve a single property value:

```
e.Path
ans =
   D:\Applications\MSOffice\Office
```

Retrieve a list of all properties for the `CommandBars` interface:

```
c = e.CommandBars.get
ans =
                Application: [1x1
Interface.excel.application.CommandBars.Application]
                    Creator: 1.4808e+009
              ActionControl: []
              ActiveMenuBar: [1x1
Interface.excel.application.CommandBars.ActiveMenuBar]
                      Count: 94
            DisplayTooltips: 1
      DisplayKeysInTooltips: 0
               LargeButtons: 0
         MenuAnimationStyle: 'msoMenuAnimationNone'
                     Parent: [1x1
Interface.excel.application.CommandBars.Parent]
              AdaptiveMenus: 0
               DisplayFonts: 1
```

See Also

set, inspect, isprop, addproperty, deleteproperty

Purpose

Memmapfile object properties

Syntax

```
s = get(obj)
val = get(obj, prop)
```

Description

`s = get(obj)` returns the values of all properties of the `memmapfile` object `obj` in structure array `s`. Each property retrieved from the object is represented by a field in the output structure. The name and contents of each field are the same as the name and value of the property it represents.

Note Although property names of a `memmapfile` object are not case sensitive, field names of the output structure returned by `get` (named the same as the properties they represent) are case sensitive.

`val = get(obj, prop)` returns the value(s) of one or more properties specified by `prop`. The `prop` input can be a quoted string or a cell array of quoted strings, each containing a property name. If the latter is true, `get` returns the property values in a cell array.

Examples

You can use the `get` method of the `memmapfile` class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a `memmapfile` object. Start by constructing the object:

```
m = memmapfile('records.dat',                  ...
               'Offset', 2048,                 ...
               'Format', {                     ...
                  'int16'  [2 2] 'model';      ...
                  'uint32' [1 1] 'serialno';   ...
                  'single' [1 3] 'expenses'});
```

Use the `get` method to return the specified property values in a 1-by-3 cell array `m_props`:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

Another way to return the same information is to use the `objname.property` syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with `get`, pass just the object name:

```
s = get(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: 0
      Offset: 2048
      Format: {3x3 cell}
      Repeat: Inf
        Data: [753 1]
```

To see just the `Format` field of the returned structure, type

```
s.Format
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

See Also

`memmapfile`, `disp(memmapfile)`

get (serial)

Purpose

Serial port object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`'PropertyName'`	A property name or a cell array of property names.
`out`	A single property value, a structure of property values, or a cell array of property values.

Description

`get(obj)` returns all property names and their current values to the command line for `obj`.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *`PropertyName`* for `obj`. If *`PropertyName`* is replaced by a 1-by-n or n-by-1 cell array of strings containing property names, then `get` returns a 1-by-n cell array of values to `out`. If `obj` is an array of serial port objects, then `out` will be a m-by-n cell array of property values where m is equal to the length of `obj` and n is equal to the number of properties specified.

Remarks

Refer to Displaying Property Names and Property Values for a list of serial port object properties that you can return with `get`.

When you specify a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then these commands are all valid.

```
out = get(s,'BaudRate');
```

```
out = get(s,'baudrate');
out = get(s,'BAUD');
```

If you use the `help` command to display help for `get`, then you need to supply the pathname shown below.

```
help serial/get
```

Example

This example illustrates some of the ways you can use `get` to return property values for the serial port object `s`.

```
s = serial('COM1');
out1 = get(s);
out2 = get(s,{'BaudRate','DataBits'});
get(s,'Parity')
ans =
none
```

See Also

Functions

`set`

get (timer)

Purpose

Timer object properties

Syntax

```
get(obj)
V = get(obj)
V = get(obj,'PropertyName')
```

Description

`get(obj)` displays all property names and their current values for the timer object `obj`. `obj` must be a single timer object.

`V = get(obj)` returns a structure, `V`, where each field name is the name of a property of `obj` and each field contains the value of that property. If `obj` is an M-by-1 vector of timer objects, V is an M-by-1 array of structures.

`V = get(obj,'PropertyName')` returns the value, `V`, of the timer object property specified in *`PropertyName`*.

If *`PropertyName`* is a 1-by-N or N-by-1 cell array of strings containing property names, `V` is a 1-by-N cell array of values. If `obj` is a vector of timer objects, `V` is an M-by-N cell array of property values where M is equal to the length of `obj` and N is equal to the number of properties specified.

Examples

```
t = timer;
get(t)
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-1'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 1
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
```

```
          TasksExecuted: 0
         TasksToExecute: Inf
                TimerFcn: ''
                    Type: 'timer'
                UserData: []
get(t, {'StartDelay','Period'})
ans =

    [0]    [1]
```

See Also

`timer`, `set(timer)`

get (timeseries)

Purpose

Query `timeseries` object property values

Syntax

```
value = get(ts,'PropertyName')
get(ts)
```

Description

`value = get(ts,'PropertyName')` returns the value of the specified property of the `timeseries` object. The following syntax is equivalent:

```
value = ts.PropertyName
```

`get(ts)` displays all properties and values of the time series `ts`.

See Also

`set (timeseries)`, `timeseries`, `tsprops`

Purpose

Query `tscollection` object property values

Syntax

```
value = get(tsc,'PropertyName')
```

Description

`value = get(tsc,'PropertyName')` returns the value of the specified property of the `tscollection` object tsc. The following syntax is equivalent:

```
value = tsc.PropertyName
```

`get(tsc)` displays all properties and values of the `tscollection` object tsc.

See Also

`set (tscollection)`, `tscollection`

getabstime (timeseries)

Purpose

Extract date-string time vector into cell array

Syntax

```
getabstime(ts)
```

Description

`getabstime(ts)` extracts the time vector from the `timeseries` object `ts` as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the `timeseries` object. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Examples

The following example shows how to extract a time vector as a cell array of date strings from a `timeseries` object.

1. Create a `timeseries` object.

   ```
   ts = timeseries([3 6 8 0 10]);
   ```

   The default time vector for `ts` is [0 1 2 3 4], which starts at 0 and increases in 1-second increments. The length of the time vector is equal to the length of the data.

2. Set the `StartDate` property.

   ```
   ts.TimeInfo.StartDate = '10/27/2005 07:05:36';
   ```

3. Extract the time vector.

   ```
   getabstime(ts)

   ans =

       '27-Oct-2005 07:05:36'
       '27-Oct-2005 07:05:37'
       '27-Oct-2005 07:05:38'
       '27-Oct-2005 07:05:39'
       '27-Oct-2005 07:05:40'
   ```

4 Change the date-string format of the time vector.

```
ts.TimeInfo.Format = 'mm/dd/yy'
```

5 Extract the time vector with the new date-string format.

```
getabstime(ts)

ans =

    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
```

See Also

`setabstime (timeseries)`, `timeseries`, `tsprops`

getabstime (tscollection)

Purpose

Extract date-string time vector into cell array

Syntax

```
getabstime(tsc)
```

Description

`getabstime(tsc)` extracts the time vector from the `tscollection` object tsc as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the time-series collection. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Examples

1 Create a `tscollection` object.

```
tsc = tscollection(timeseries([3 6 8 0 10]));
```

2 Set the `StartDate` property.

```
tsc.TimeInfo.StartDate = '10/27/2005 07:05:36';
```

3 Extract a vector of absolute time values.

```
getabstime(tsc)

ans =

    '27-Oct-2005 07:05:36'
    '27-Oct-2005 07:05:37'
    '27-Oct-2005 07:05:38'
    '27-Oct-2005 07:05:39'
    '27-Oct-2005 07:05:40'
```

4 Change the date-string format of the time vector.

```
tsc.TimeInfo.Format = 'mm/dd/yy';
```

5 Extract the time vector with the new date-string format.

```
getabstime(tsc)
```

```
ans =

    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
```

See Also

datestr, setabstime (tscollection), tscollection

getappdata

Purpose

Value of application-defined data

Syntax

```
value = getappdata(h,name)
values = getappdata(h)
```

Description

`value = getappdata(h,name)` gets the value of the application-defined data with the name specified by `name`, in the object with handle `h`. If the application-defined data does not exist, MATLAB returns an empty matrix in `value`.

`values = getappdata(h)` returns all application-defined data for the object with handle `h`.

See Also

`setappdata`, `rmappdata`, `isappdata`

Purpose

Get character array from server

Syntax

MATLAB Client

```
string = h.GetCharArray('varname', 'workspace')
string = GetCharArray(h, 'varname', 'workspace')
string = invoke(h, 'GetCharArray', 'varname', 'workspace')
```

Method Signature

```
HRESULT GetCharArray ([in] BSTR varName, [in] BSTR Workspace,
[out, retval] BSTR *mlString)
```

Visual Basic Client

```
GetCharArray(varname As String, workspace As String) As String
```

Description

`GetCharArray` gets the character array stored in the variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in `string`. The *`workspace`* argument can be either `base` or `global`.

Remarks

If you want output from `GetCharArray` to be displayed at the client window, you must specify an output variable (e.g., `string`).

Server function names, like `GetCharArray`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Examples

Assign a string to variable `str` in the base workspace of the server using `PutCharArray`. Read it back in the client with `GetCharArray`.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutCharArray('str', 'base', ...
   'He jests at scars that never felt a wound.');
S = h.GetCharArray('str', 'base')
S =
  He jests at scars that never felt a wound.
```

Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab As Object
Dim S As String
Matlab = CreateObject("matlab.application")
MsgBox("In MATLAB, type" & vbCrLf _
    & "str='new string';")
```

Open the MATLAB window, then type

```
str='new string';
```

Click **Ok**.

```
Try
    S = Matlab.GetCharArray("str", "base")
    MsgBox("str = " & S)
Catch ex As Exception
    MsgBox("You did not set 'str' in MATLAB")
End Try
```

The Visual Basic `MsgBox` displays what you typed in MATLAB.

See Also

`PutCharArray`, `GetWorkspaceData`, `PutWorkspaceData`, `GetVariable`, `Execute`

Purpose

Size of data sample in `timeseries` object

Syntax

```
getdatasamplesize(ts)
```

Description

`getdatasamplesize(ts)` returns the size of each data sample in a `timeseries` object.

Remarks

A time-series *data sample* consists of one or more scalar values recorded at a specific time. The number of data samples in is the same as the length of the time vector.

Examples

The following example shows how to get the size of a data sample in a `timeseries` object.

1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','VehicleCount')
```

3 Get the size of the data sample for this `timeseries` object.

```
getdatasamplesize(count_ts)

ans =

     1     3
```

The size of each data sample in `count_ts` is 1-by-3, which means that each data sample is stored as a row with three values.

See Also

`addsample`, `size (timeseries)`, `tsprops`

getenv

Purpose

Environment variable

Syntax

```
getenv 'name'
N = getenv('name')
```

Description

`getenv 'name'` searches the underlying operating system's environment list for a string of the form `name=value`, where `name` is the input string. If found, MATLAB returns the string `value`. If the specified name cannot be found, an empty matrix is returned.

`N = getenv('name')` returns `value` to the variable `N`.

Examples

```
os = getenv('OS')

os =
Windows_NT
```

See Also

`setenv`, `computer`, `pwd`, `ver`, `path`

Purpose

Field of structure array

Syntax

```
f = getfield(s,'field')
f = getfield(s, {i,j}, 'field', {k})
```

Description

`f = getfield(s,'field')`, where `s` is a 1-by-1 structure, returns the contents of the specified field. This is equivalent to the syntax `f = s.field`.

If `s` is a structure having dimensions greater than 1-by-1, `getfield` returns the first of all output values requested in the call. That is, for structure array `s(m,n)`, `getfield` returns `f = s(1,1).field`.

`f = getfield(s, {i,j}, 'field', {k})` returns the contents of the specified field. This is equivalent to the syntax `f = s(i,j).field(k)`. All subscripts must be passed as cell arrays — that is, they must be enclosed in curly braces (similar to`{i,j}` and `{k}` above). Pass field references as strings.

Remarks

In many cases, you can use dynamic field names in place of the `getfield` and `setfield` functions. Dynamic field names express structure fields as variable expressions that MATLAB evaluates at run-time. See Solution 1-19QWG for information about using dynamic field names versus the `getfield` and `setfield` functions.

Examples

Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1
```

Then the command `f = getfield(mystr, {2,1}, 'name')` yields

```
f =
   gertrude
```

To list the contents of all name (or other) fields, embed `getfield` in a loop.

```
for k = 1:2
    name{k} = getfield(mystr, {k,1}, 'name');
end
name

name =

    'alice'    'gertrude'
```

The following example starts out by creating a structure using the standard structure syntax. It then reads the fields of the structure, using `getfield` with variable and quoted field names and additional subscripting arguments.

```
class = 5;     student = 'John_Doe';
grades(class).John_Doe.Math(10,21:30) = ...
             [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
```

Use `getfield` to access the structure fields.

```
getfield(grades, {class}, student, 'Math', {10,21:30})

ans =
   85   89   76   93   85   91   68   84   95   73
```

See Also

`setfield`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, "Using Dynamic Field Names"

Purpose

Capture movie frame

Syntax

```
getframe
F = getframe
F = getframe(h)
F = getframe(h,rect)
```

Description

`getframe` returns a movie frame. The frame is a snapshot (pixmap) of the current axes or figure.

`F = getframe` gets a frame from the current axes.

`F = getframe(h)` gets a frame from the figure or axes identified by handle `h`.

`F = getframe(h,rect)` specifies a rectangular area from which to copy the pixmap. `rect` is relative to the lower left corner of the figure or axes `h`, in pixel units. `rect` is a four-element vector in the form `[left bottom width height]`, where `width` and `height` define the dimensions of the rectangle.

`getframe` returns a movie frame, which is a structure having two fields:

- `cdata` — The image data stored as a matrix of `uint8` values. The dimensions of `F.cdata` are height-by-width-by-3.
- `colormap` — The colormap stored as an n-by-3 matrix of doubles. `F.colormap` is empty on true color systems.

To capture an image, use this approach:

```
F = getframe(gcf);
image(F.cdata)
colormap(F.colormap)
```

Remarks

`getframe` is usually used in a `for` loop to assemble an array of movie frames for playback using `movie`. For example,

```
for j = 1:n plotting commands
 F(j) = getframe;
```

```
end
movie(F)
```

If you are capturing frames of a plot that takes a long time to generate or are repeatedly calling `getframe` in a loop, make sure that your computer's screen saver does not activate and that your monitor does not turn off for the duration of the capture; otherwise one or more of the captured frames can contain graphics from your screen saver or nothing at all.

Note In situations where MATLAB is running on a virtual desktop that is not currently visible on your monitor, calls to `getframe` will complete, but will capture a region on your monitor that corresponds to the position occupied by the figure or axes on the hidden desktop. Therefore, make sure that the window to be captured by `getframe` exists on the currently active desktop.

Capture Regions

Note that `F = getframe` returns the contents of the current axes, exclusive of the axis labels, title, or tick labels. `F = getframe(gcf)` captures the entire interior of the current figure window. To capture the figure window menu, use the form `F = getframe(h,rect)` with a rectangle sized to include the menu.

Resolution of Captured Frames

The resolution of the framed image depends on the size of the axes in pixels when `getframe` is called. As the `getframe` command takes a snapshot of the screen, if the axes is small in size (e.g., because you have restricted the view to a window within the axes), `getframe` will capture fewer screen pixels, and the captured image might have poor resolution if enlarged for display.

Examples

Make the `peaks` function vibrate.

```
Z = peaks; surf(Z)
```

```
axis tight
set(gca,'nextplot','replacechildren');
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
movie(F,20) % Play the movie twenty times
```

See Also

`frame2im`, `image`, `im2frame`, `movie`

"Bit-Mapped Images" on page 1-92 for related functions

GetFullMatrix

Purpose

Get matrix from server

Syntax

MATLAB Client

```
[xreal ximag] = h.GetFullMatrix('varname', 'workspace',
zreal, zimag)
[xreal ximag] = GetFullMatrix(h, 'varname', 'workspace',
zreal, zimag)
[xreal ximag] = invoke(h, 'GetFullMatrix', 'varname', 'workspace',
zreal, zimag)
```

Method Signature

```
GetFullMatrix([in] BSTR varname,
[in] BSTR workspace, [in, out] SAFEARRAY(double) *pr,
[in, out] SAFEARRAY(double) *pi)
```

Visual Basic Client

```
GetFullMatrix(varname As String, workspace As String,
[out] XReal As Double, [out] XImag As Double
```

Note `GetFullMatrix` works only with values of type `double`. Use `GetVariable` or `GetWorkspaceData` for other types.

Description

`GetFullMatrix` gets the matrix stored in the variable `varname` from the specified `workspace` of the server attached to handle `h` and returns the real part in `xreal` and the imaginary part in `ximag`. The *`workspace`* argument can be either `base` or `global`.

The `zreal` and `zimag` arguments are matrices of the same size as the real and imaginary matrices (`xreal` and `ximag`) being returned from the server. The `zreal` and `zimag` matrices are commonly set to zero (see example below).

Remarks

If you want output from `GetFullMatrix` to be displayed at the client window, you must specify one or both output variables (e.g., `xreal` and/or `ximag`).

Server function names, like `GetFullMatrix`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of `safearray`, which is not supported by VBScript.

Examples

Assign a 5-by-5 real matrix to the variable `M` in the base workspace of the server, and then read it back with `GetFullMatrix`.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutFullMatrix('M','base',rand(5),zeros(5));

MReal = h.GetFullMatrix('M','base',zeros(5),zeros(5))
MReal =
    0.9501    0.7621    0.6154    0.4057    0.0579
    0.2311    0.4565    0.7919    0.9355    0.3529
    0.6068    0.0185    0.9218    0.9169    0.8132
    0.4860    0.8214    0.7382    0.4103    0.0099
    0.8913    0.4447    0.1763    0.8936    0.1389
```

Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim MatLab As Object
Dim Result As String
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim i, j As Integer

MatLab = CreateObject("matlab.application")
Result = MatLab.Execute("M = rand(5);")
```

```
MsgBox("In MATLAB, type" & vbCrLf _
  & "M(3,4)")
```

Open the MATLAB window and type

```
M(3,4)
```

Click **Ok**.

```
MatLab.GetFullMatrix("M", "base", XReal, XImag)
i = 2   %0-based array
j = 3

MsgBox("XReal(" & i + 1 & "," & j + 1 & ")" & _
       " = " & XReal(i, j))
```

Click **Ok** to close and terminate MATLAB.

See Also

`PutFullMatrix`, `GetWorkspaceData`, `PutWorkspaceData`, `GetVariable`, `Execute`

Purpose

Interpolation method for `timeseries` object

Syntax

```
getinterpmethod(ts)
```

Description

`getinterpmethod(ts)` returns the interpolation method as a string that is used by the `timeseries` object `ts`. Predefined interpolation methods are `'zoh'` (zero-order hold) and `'linear'` (linear interpolation). The method strings are case sensitive.

Examples

1 Create a `timeseries` object.

```
ts = timeseries(rand(5));
```

2 Get the interpolation method for this object.

```
getinterpmethod(ts)

ans =

     linear
```

See Also

`setinterpmethod`, `timeseries`, `tsprops`

getpixelposition

Purpose

Get component position in pixels

Syntax

```
position = getpixelposition(handle)
position = getpixelposition(handle,recursive)
```

Description

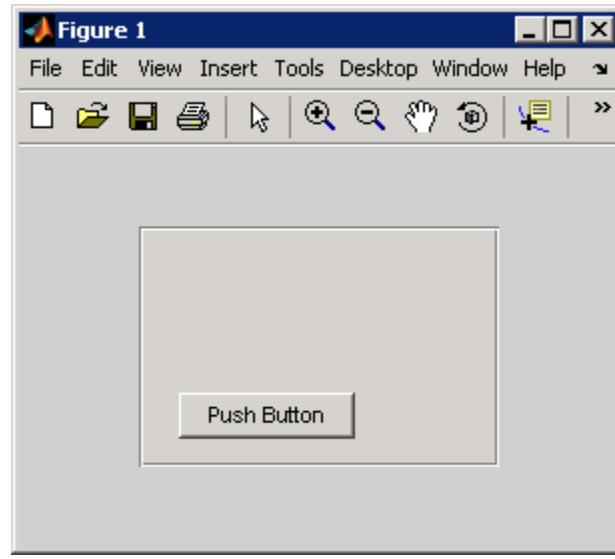
`position = getpixelposition(handle)` gets the position, in pixel units, of the component with handle `handle`. The `position` is returned as a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

`position = getpixelposition(handle,recursive)` gets the position as above. If `recursive` is true, the returned position is relative to the parent figure of `handle`.

Example

This example creates a push button within a panel, and then retrieves its position, in pixels, relative to the panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6];
h1 = uicontrol(p,'Style','PushButton','Units','Normalized',...
               'String','Push Button','Position',[.1 .1 .5 .2]);
pos1 = getpixelposition(h1)

pos1 =
     18.6000  12.6000  88.0000  23.2000
```



The following statement retrieves the position of the push button, in pixels, relative to the figure.

```
pos1 = getpixelposition(h1,true)

pos1 =
      79.6000   53.6000   88.0000   23.2000
```

See Also

`setpixelposition`, `uicontrol`, `uipanel`

getpref

Purpose

Preference

Syntax

```
getpref('group','pref')
getpref('group','pref',default)
getpref('group',{'pref1','pref2',...'prefn'})
getpref('group',{'pref1',...'prefn'},{default1,...defaultn})
getpref('group')
getpref
```

Description

`getpref('group','pref')` returns the value for the preference specified by `group` and `pref`. It is an error to get a preference that does not exist.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g. `'ApplicationOnePrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`getpref('group','pref',default)` returns the current value if the preference specified by `group` and `pref` exists. Otherwise creates the preference with the specified default value and returns that value.

`getpref('group',{'pref1','pref2',...'prefn'})` returns a cell array containing the values for the preferences specified by `group` and the cell array of preference names. The return value is the same size as the input cell array. It is an error if any of the preferences do not exist.

`getpref('group',{'pref1',...'prefn'},{default1,...defaultn})` returns a cell array with the current values of the preferences specified by `group` and the cell array of preference names. Any preference that does not exist is created with the specified default value and returned.

`getpref('group')` returns the names and values of all preferences in the `group` as a structure.

`getpref` returns all groups and preferences as a structure.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

Example 1

```
addpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')

ans =
     1.0
```

Example 2

```
rmpref('mytoolbox','version')
getpref('mytoolbox','version','1.0');
getpref('mytoolbox','version')

ans =
     1.0
```

See Also

addpref, ispref, rmpref, setpref, uigetpref, uisetpref

getqualitydesc

Purpose

Data quality descriptions

Syntax

```
getqualitydesc(ts)
```

Description

`getqualitydesc(ts)` returns a cell array of data quality descriptions based on the `Quality` values you assigned to a `timeseries` object `ts`.

Examples

1 Create a `timeseries` object with `Data`, `Time`, and `Quality` values, respectively.

```
ts = timeseries([3; 4.2; 5; 6.1; 8], 1:5, [1; 0; 1; 0; 1]);
```

2 Set the `QualityInfo` property, consisting of `Code` and `Description`.

```
ts.QualityInfo.Code = [0 1];
ts.QualityInfo.Description = {'good' 'bad'};
```

3 Get the data quality description strings for `ts`.

```
getqualitydesc(ts)

ans =

    'bad'
    'good'
    'bad'
    'good'
    'bad'
```

See Also

`tsprops`

Purpose

Get error message for exception

Syntax

```
Report = getReport(ME)
```

Description

`Report = getReport(ME)` returns a formatted message string based on the current exception (represented by `MException` object `ME`) and that uses the same format as errors thrown by internal MATLAB code. The message string returned by `getReport` is the same as the error message displayed by MATLAB when it throws the exception.

Examples

Using the `surf` command without input arguments throws an exception. The `catch` function captures the exception in `MException` object `ME`. Use `getReport` to obtain the error text in the same format that MATLAB uses:

```
try
   surf
catch ME
   getReport(ME)
end

ans =
   ??? Error using ==> surf at 54
   Not enough input arguments.
```

See Also

`try`, `catch`, `error`, `assert`, `MException`, `disp(MException)`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `isequal(MException)`, `eq(MException)`, `ne(MException)`, `last(MException)`,

getsampleusingtime (timeseries)

Purpose

Extract data samples into new `timeseries` object

Syntax

```
ts2 = getsampleusingtime(ts1,Time)
ts2 = getsampleusingtime(ts1,StartTime,EndTime)
```

Description

`ts2 = getsampleusingtime(ts1,Time)` returns a new `timeseries` object `ts2` with a single sample corresponding to the time Time in ts1.

`ts2 = getsampleusingtime(ts1,StartTime,EndTime)` returns a new `timeseries` object `ts2` with samples between the times `StartTime` and `EndTime` in ts1.

Remarks

When the time vector in ts1 is numeric, `StartTime` and `EndTime` must also be numeric. When the times in ts1 are date strings and the `StartTime` and `EndTime` values are numeric, then the `StartTime` and `EndTime` values are treated as `datenum` values.

See Also

`timeseries`

Purpose

Extract data samples into new `tscollection` object

Syntax

```
tsc2 = getsampleusingtime(tsc1,Time)
tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)
```

Description

`tsc2 = getsampleusingtime(tsc1,Time)` returns a new `tscollection` tsc2 with a single sample corresponding to Time in tsc1.

`tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)` returns a new `tscollection` tsc2 with samples between the times StartTime and EndTime in tsc1.

Remarks

When the time vector in ts1 is numeric, `StartTime` and `EndTime` must also be numeric. When the times in ts1 are date strings and the `StartTime` and `EndTime` values are numeric, then the `StartTime` and `EndTime` values are treated as `datenum` values.

See Also

`tscollection`

gettimeseriesnames

Purpose

Cell array of names of `timeseries` objects in `tscollection` object

Syntax

```
names = gettimeseriesnames(tsc)
```

Description

`names = gettimeseriesnames(tsc)` returns `names` of `timeseries` objects in a `tscollection` object `tsc`. `names` is a cell array of strings.

Examples

1 Create `timeseries` objects `a` and `b`.

```
a = timeseries(rand(1000,1),'name','position');
b = timeseries(rand(1000,1),'name','response');
```

2 Create a `tscollection` object that includes these two time series.

```
tsc = tscollection({a,b});
```

3 Get the names of the `timeseries` objects in `tsc`.

```
names = gettimeseriesnames(tsc)

names =

    'position'    'response'
```

See Also

`timeseries`, `tscollection`, `tsprops`

Purpose

New `timeseries` object with samples occurring at or after event

Syntax

```
ts1 = gettsafteratevent(ts,event)
ts1 = gettsafteratevent(ts,event,n)
```

Description

`ts1 = gettsafteratevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring at and after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of the time series ts that matches the event name specifies the time.

`ts1 = gettsafteratevent(ts,event,n)` returns a new `timeseries` object ts1 with samples at and after an event in ts, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

gettsafterevent

Purpose

New `timeseries` object with samples occurring after event

Syntax

```
ts1 = gettsafterevent(ts,event)
ts1 = ttsafterevent(ts,event,n)
```

Description

`ts1 = gettsafterevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = ttsafterevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring after an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafteratevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

Purpose

New `timeseries` object with samples occurring at event

Syntax

```
ts1 = gettsatevent(ts,event)
ts1 = gettsatevent(ts,event,n)
```

Description

`ts1 = gettsatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsatevent(ts,event,n)` returns a new time series `ts1` with samples occurring at an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in the `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsafteratevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

gettsbeforeatevent

Purpose

New `timeseries` object with samples occurring before or at event

Syntax

```
ts1 = gettsbeforeatevent(ts,event)
ts1 = gettsbeforeatevent(ts,event,n)
```

Description

`ts1 = gettsbeforeatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeatevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

Purpose

New `timeseries` object with samples occurring before event

Syntax

```
ts1 = gettsbeforeevent(ts,event)
ts1 = gettsbeforeevent(ts,event,n)
```

Description

`ts1 = gettsbeforeevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where n is the number of the event occurrence with a matching event name.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeatevent`, `gettsbetweenevents`, `tsdata.event`, `tsprops`

gettsbetweenevents

Purpose

New `timeseries` object with samples occurring between events

Syntax

```
ts1 = gettsbetweenevents(ts,event1,event2)
ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)
```

Description

`ts1 = gettsbetweenevents(ts,event1,event2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `event1` and `event2` can be either a `tsdata.event` object or a string. When `event1` and `event2` are `tsdata.event` objects, the time defined by the events is used. When `event1` and `event2` are strings, the first `tsdata.event` object in the `Events` property of `ts` that matches the event names specifies the time.

`ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `n1` and `n2` are the nth occurrences of the events with matching event names.

Remarks

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent`, `gettsbeforeevent`, `tsdata.event`, `tsprops`

Purpose

Get data from variable in server workspace

Syntax

MATLAB Client

```
D = h.GetVariable('varname', 'workspace')
D = GetVariable(h, 'varname', 'workspace')
D = invoke(h, 'GetVariable', 'varname', 'workspace')
```

Method Signature

```
HRESULT GetVariable([in] BSTR varname, [in] BSTR workspace,
[out, retval] VARIANT* pdata)
```

Visual Basic Client

```
GetVariable(varname As String, workspace As String) As Object
```

Description

`GetVariable` returns the data stored in the specified variable from the specified workspace of the server. Each syntax in the MATLAB Client section produce the same result. Note that the dot notation (`h.GetVariable`) is case sensitive.

`varname` from the specified `workspace` of the server that is attached to handle `h`. The *`workspace`* argument can be either `base` or `global`.

`varname` — the name of the variable whose data is returned

`workspace` — the workspace containing the variable can be either:

- `base` is the base workspace of the server
- `global` is the global workspace of the server (see `global` for more information about how to access variables in the global workspace).

Note `GetVariable` works on all MATLAB data types except sparse arrays, structures, and function handles.

Remarks

You can use `GetVariable` in place of `GetWorkspaceData`, `GetFullMatrix` and `GetCharArray` to get data stored in workspace variables when you

need a result returned explicitly (which might be required by some scripting languages).

Examples

This example assigns a cell array to the variable `C1` in the base workspace of the server, and then read it back with `GetVariable`, assigning it to a new variable `C2`.

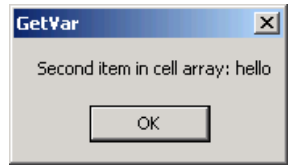
MATLAB Client

```
h = actxserver('matlab.application');
h.PutWorkspaceData('C1', 'base', {25.72, 'hello', rand(4)});
C2 = h.GetVariable('C1','base')
C2 =
    [25.7200]    'hello'    [4x4 double]
```

Visual Basic .NET Client

```
Dim Matlab As Object
Dim Result As String
Dim C2 As Object
Matlab = CreateObject("matlab.application")
Result = Matlab.Execute("C1 = {25.72, 'hello', rand(4)};")
C2 = Matlab.GetVariable("C1", "base")
MsgBox("Second item in cell array: " & C2(0, 1))
```

The Visual Basic Client example creates a message box displaying the second element in the cell array, which is the string `hello`.



See Also

`GetWorkspaceData`, `PutWorkspaceData`, `GetFullMatrix`, `PutFullMatrix`, `GetCharArray`, `PutCharArray`, `Execute`

Purpose

Get data from server workspace

Syntax

MATLAB Client

```
D = h.GetWorkspaceData('varname', 'workspace')
D = GetWorkspaceData(h, 'varname', 'workspace')
D = invoke(h, 'GetWorkspaceData', 'varname', 'workspace')
```

Method Signature

```
HRESULT GetWorkspaceData([in] BSTR varname, [in] BSTR workspace,
[out] VARIANT* pdata)
```

Visual Basic Client

```
GetWorkspaceData(varname As String, workspace As String) As Object
```

Description

`GetWorkspaceData` gets the data stored in the variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The *`workspace`* argument can be either `base` or `global`.

Note `GetWorkspaceData` works on all MATLAB data types except sparse arrays, structures, and function handles.

Remarks

You can use `GetWorkspaceData` in place of `GetFullMatrix` and `GetCharArray` to get numeric and character array data respectively.

If you want output from `GetWorkspaceData` to be displayed at the client window, you must specify an output variable.

Server function names, like `GetWorkspaceData`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

The `GetWorkspaceData` and `PutWorkspaceData` functions pass numeric data as a `variant` data type. These functions are especially useful for

VBScript clients as VBScript does not support the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`.

Examples

Assign a cell array to variable `C1` in the base workspace of the server, and then read it back with `GetWorkspaceData`.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutWorkspaceData('C1', 'base', ...
                   {25.72, 'hello', rand(4)});
C2 = h.GetWorkspaceData('C1', 'base')

C2 =
    [25.7200]    'hello'    [4x4 double]
```

Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab, C2 As Object
Dim Result As String
Matlab = CreateObject("matlab.application")
Result = MatLab.Execute("C1 = {25.72, 'hello', rand(4)};")
MsgBox("In MATLAB, type" & vbCrLf & "C1")
Matlab.GetWorkspaceData("C1", "base", C2)
MsgBox("second value of C1 = " & C2(0, 1))
```

See Also

`PutWorkspaceData`, `GetFullMatrix`, `PutFullMatrix`, `GetCharArray`, `PutCharArray`, `GetVariable`, `Execute`

Purpose

Graphical input from mouse or cursor

Syntax

```
[x,y] = ginput(n)
[x,y] = ginput
[x,y,button] = ginput(...)
```

Description

`ginput` enables you to select points from the figure using the mouse for cursor positioning. The figure must have focus before `ginput` receives input.

`[x,y] = ginput(n)` enables you to select `n` points from the current axes and returns the *x*- and *y*-coordinates in the column vectors `x` and `y`, respectively. Press the **Return** key to terminate the input before entering `n` points.

`[x,y] = ginput` gathers an unlimited number of points until you press the **Return** key.

Note Clicking an axes makes that axes the current axes. Although you may set the current axes before calling `ginput`, whichever axes the user clicks becomes the current axes and `ginput` returns points relative to that axes. For example, if a user selects points from multiple axes, the results returned are relative to the different axes' coordinate systems.

`[x,y,button] = ginput(...)` returns the *x*-coordinates, the *y*-coordinates, and the button or key designation. `button` is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right), or ASCII numbers indicating which keys on the keyboard you pressed.

Examples

Pick 10 two-dimensional points from the figure window.

```
[x,y] = ginput(10)
```

Position the cursor with the mouse. Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 10 points, press the **Return** key.

See Also

`gtext`

"Interactive Plotting" for an example

"Developing User Interfaces" on page 1-105 for related functions

Purpose

Declare global variables

Syntax

```
global X Y Z
```

Description

`global X Y Z` defines `X`, `Y`, and `Z` as global in scope.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace. However, if several functions, and possibly the base workspace, all declare a particular name as global, they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the functions declaring it global.

If the global variable does not exist the first time you issue the `global` statement, it is initialized to the empty matrix.

If a variable with the same name as the global variable already exists in the current workspace, MATLAB issues a warning and changes the value of that variable to match the global.

Remarks

Use `clear global` *`variable`* to clear a global variable from the global workspace. Use `clear` *`variable`* to clear the global link from the current workspace without affecting the value of the global.

To use a global within a callback, declare the global, use it, then clear the global link from the workspace. This avoids declaring the global after it has been referenced. For example,

```
cbstr = sprintf('%s, %s, %s, %s, %s', ...
   'global MY_GLOBAL',                ...
   'MY_GLOBAL = 100',                 ...
   'disp(MY_GLOBAL)',                 ...
   'MY_GLOBAL = MY_GLOBAL+1',         ...
   'clear MY_GLOBAL');

uicontrol('style', 'pushbutton', 'CallBack', cbstr, ...
   'string', 'count')
```

There is no function form of the global command (i.e., you cannot use parentheses and quote the variable names).

Examples

Here is the code for the functions `tic` and `toc` (some comments abridged). These functions manipulate a stopwatch-like timer. The global variable `TICTOC` is shared by the two functions, but it is invisible in the base workspace or in any other functions that do not declare it.

```
function tic
%    TIC Start a stopwatch timer.
%        TIC; any stuff; TOC
%    prints the time required.
%    See also: TOC, CLOCK.
global TICTOC
TICTOC = clock;

function t = toc
%    TOC Read the stopwatch timer.
%    TOC prints the elapsed time since TIC was used.
%    t = TOC; saves elapsed time in t, does not print.
%    See also: TIC, ETIME.
global TICTOC
if nargout < 1
    elapsed_time = etime(clock, TICTOC)
else
    t = etime(clock, TICTOC);
end
```

See Also

`clear`, `isglobal`, `who`

Purpose

Generalized minimum residual method (with restarts)

Syntax

```
x = gmres(A,b)
gmres(A,b,restart)
gmres(A,b,restart,tol)
gmres(A,b,restart,tol,maxit)
gmres(A,b,restart,tol,maxit,M)
gmres(A,b,restart,tol,maxit,M1,M2)
gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag] = gmres(A,b,...)
[x,flag,relres] = gmres(A,b,...)
[x,flag,relres,iter] = gmres(A,b,...)
[x,flag,relres,iter,resvec] = gmres(A,b,...)
```

Description

`x = gmres(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See "Function Handles" in the MATLAB Programming documentation for more information. For this syntax, `gmres` does not restart; the maximum number of iterations is `min(n,10)`.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `gmres` converges, a message to that effect is displayed. If `gmres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`gmres(A,b,restart)` restarts the method every `restart` inner iterations. The maximum number of outer iterations is `min(n/restart,10)`. The maximum number of total iterations is `restart*min(n/restart,10)`. If `restart` is `n` or `[]`, then `gmres` does not restart and the maximum number of total iterations is `min(n,10)`.

`gmres(A,b,restart,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `gmres` uses the default, `1e-6`.

`gmres(A,b,restart,tol,maxit)` specifies the maximum number of outer iterations, i.e., the total number of iterations does not exceed `restart*maxit`. If `maxit` is `[]` then `gmres` uses the default, `min(n/restart,10)`. If `restart` is `n` or `[]`, then the maximum number of total iterations is `maxit` (instead of `restart*maxit`).

`gmres(A,b,restart,tol,maxit,M)` and `gmres(A,b,restart,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `gmres` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`gmres(A,b,restart,tol,maxit,M1,M2,x0)` specifies the first initial guess. If `x0` is `[]`, then `gmres` uses the default, an all-zero vector.

`[x,flag] = gmres(A,b,...)` also returns a convergence flag:

`flag = 0`	`gmres` converged to the desired tolerance `tol` within `maxit` outer iterations.
`flag = 1`	`gmres` iterated `maxit` times but did not converge.
`flag = 2`	Preconditioner `M` was ill-conditioned.
`flag = 3`	`gmres` stagnated. (Two consecutive iterates were the same.)

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = gmres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = gmres(A,b,...)` also returns both the outer and inner iteration numbers at which `x` was computed, where `0 <= iter(1) <= maxit` and `0 <= iter(2) <= restart`.

`[x,flag,relres,iter,resvec] = gmres(A,b,...)` also returns a vector of the residual norms at each inner iteration, including `norm(b-A*x0)`.

Examples

Example 1

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = gmres(A,b,10,tol,maxit,M1);
```

displays the following message:

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in an M-file `run_gmres` that

- Calls `gmres` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_gmres` are available to `afun` and `mfun`.

The following shows the code for `run_gmres`:

```
function x1 = run_gmres
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;  maxit = 15;
x1 = gmres(@afun,b,10,tol,maxit,@mfun);
```

```
    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_gmres;
```

MATLAB displays the message

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

Example 3

```
load west0479
A = west0479
b = sum(A,2)
[x,flag] = gmres(A,b,5)
```

`flag` is `1` because `gmres` does not converge to the default tolerance `1e-6` within the default 10 outer iterations.

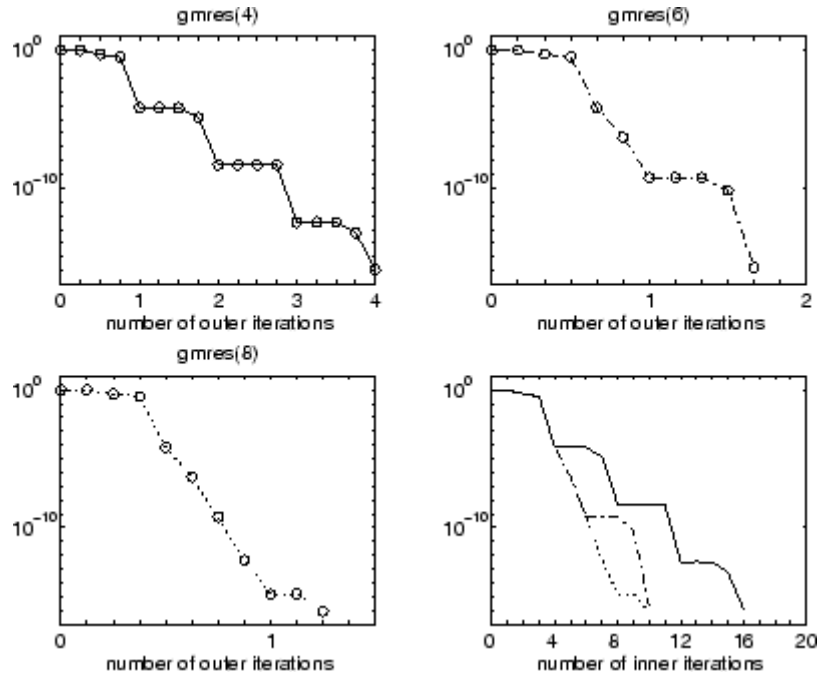
```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = gmres(A,b,5,1e-6,5,L1,U1);
```

`flag1` is `2` because the upper triangular `U1` has a zero on its diagonal, and `gmres` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` using backslash.

```
[L2,U2] = luinc(A,1e-6);
```

```
tol = 1e-15;
[x4,flag4,relres4,iter4,resvec4] = gmres(A,b,4,tol,5,L2,U2);
[x6,flag6,relres6,iter6,resvec6] = gmres(A,b,6,tol,3,L2,U2);
[x8,flag8,relres8,iter8,resvec8] = gmres(A,b,8,tol,3,L2,U2);
```

`flag4`, `flag6`, and `flag8` are all `0` because `gmres` converged when restarted at iterations 4, 6, and 8 while preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. This is verified by the plots of outer iteration number against relative residual. A combined plot of all three clearly shows the restarting at iterations 4 and 6. The total number of iterations computed may be more for lower values of `restart`, but the number of length `n` vectors stored is fewer, and the amount of work done in the method decreases proportionally.



See Also

`bicg`, `bicgstab`, `cgs`, `lsqr`, `ilu`, `luinc`, `minres`, `pcg`, `qmr`, `symmlq`

`function_handle` (`@`), `mldivide` (`\`)

References

Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Saad, Youcef and Martin H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, July 1986, Vol. 7, No. 3, pp. 856-869.

Purpose

Plot nodes and links representing adjacency matrix

Syntax

```
gplot(A,Coordinates)
gplot(A,Coordinates,LineSpec)
```

Description

The `gplot` function graphs a set of coordinates using an adjacency matrix.

`gplot(A,Coordinates)` plots a graph of the nodes defined in `Coordinates` according to the *n*-by-*n* adjacency matrix `A`, where *n* is the number of nodes. `Coordinates` is an *n*-by-2 matrix, where *n* is the number of nodes and each coordinate pair represents one node.

`gplot(A,Coordinates,LineSpec)` plots the nodes using the line type, marker symbol, and color specified by *`LineSpec`*.

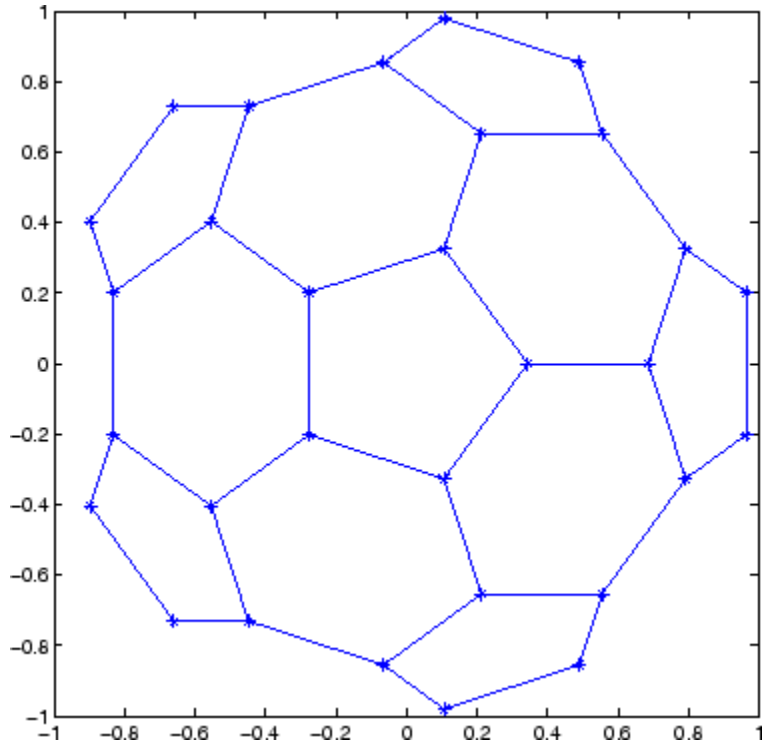
Remarks

For two-dimensional data, `Coordinates(i,:) = [x(i) y(i)]` denotes node `i`, and `Coordinates(j,:) = [x(j)y(j)]` denotes node `j`. If node `i` and node `j` are connected, `A(i,j)` or `A(j,i)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

Examples

To draw half of a Bucky ball with asterisks at each node,

```
k = 1:30;
[B,XY] = bucky;
gplot(B(k,k),XY(k,:),'-*')
axis square
```



See Also

`LineSpec`, `sparse`, `spy`

"Tree Operations" on page 1-39 for related functions

Purpose

MATLAB code from M-files published to HTML

Syntax

```
grabcode('name.html')
grabcode('urlname')
codeString = grabcode('name.html')
```

Description

`grabcode('name.html')` copies MATLAB code from the file `name.html` and pastes it into an untitled document in the Editor/Debugger. Use `grabcode` to get MATLAB code from demos or other published M-files when the M-file source code is not readily available. The file `name.html` was created by publishing `name.m`, an M-file containing cells. The MATLAB code from `name.m` is included at the end of `name.html` as HTML comments.

`grabcode('urlname')` copies MATLAB code from the `urlname` location and pastes it into an untitled document in the Editor/Debugger.

`codeString = grabcode('name.html')` get MATLAB code from the file `name.html` and assigns it the variable `codeString`.

Examples

Run

```
sineWaveString = grabcode('d:/mymfiles/sine_wave_.html')
```

and MATLAB displays

```
sineWaveString =

%% Simple Sine Wave Plot

%% Part One: Calculate Sine Wave
% Define the range |x|.
% Calculate the sine |y| over that range.
x = 0:.01:6*pi;
y = sin(x);

%% Part Two: Plot Sine Wave
% Graph the result.
```

```
plot(x,y)
```

See Also

demo, publish

Purpose

Numerical gradient

Syntax

```
FX = gradient(F)
[FX,FY] = gradient(F)
[FX,FY,FZ,...] = gradient(F)
[...] = gradient(F,h)
[...] = gradient(F,h1,h2,...)
```

Definition

The *gradient* of a function of two variables, $F(x, y)$, is defined as

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j}$$

and can be thought of as a collection of vectors pointing in the direction of increasing values of F. In MATLAB, numerical gradients (differences) can be computed for functions with any number of variables. For a function of N variables, $F(x, y, z, \ldots)$,

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} + \frac{\partial F}{\partial z}\hat{k} + \ldots$$

Description

`FX = gradient(F)` where `F` is a vector returns the one-dimensional numerical gradient of `F`. `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction.

`[FX,FY] = gradient(F)` where `F` is a matrix returns the x and y components of the two-dimensional numerical gradient. `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction. `FY` corresponds to $\partial F/\partial y$, the differences in the y (vertical) direction. The spacing between points in each direction is assumed to be one.

`[FX,FY,FZ,...] = gradient(F)` where `F` has `N dimensions` returns the `N` components of the gradient of `F`. There are two ways to control the spacing between values in `F`:

- A single spacing value, `h`, specifies the spacing between points in every direction.

- `N` spacing values (`h1,h2,...`) specifies the spacing for each dimension of `F`. Scalar spacing parameters specify a constant spacing for each dimension. Vector parameters specify the coordinates of the values along corresponding dimensions of `F`. In this case, the length of the vector must match the size of the corresponding dimension.

Note The first output `FX` is always the gradient along the 2nd dimension of `F`, going across columns. The second output `FY` is always the gradient along the 1st dimension of `F`, going across rows. For the third output `FZ` and the outputs that follow, the Nth output is the gradient along the Nth dimension of `F`.

`[...] = gradient(F,h)` where `h` is a scalar uses `h` as the spacing between points in each direction.

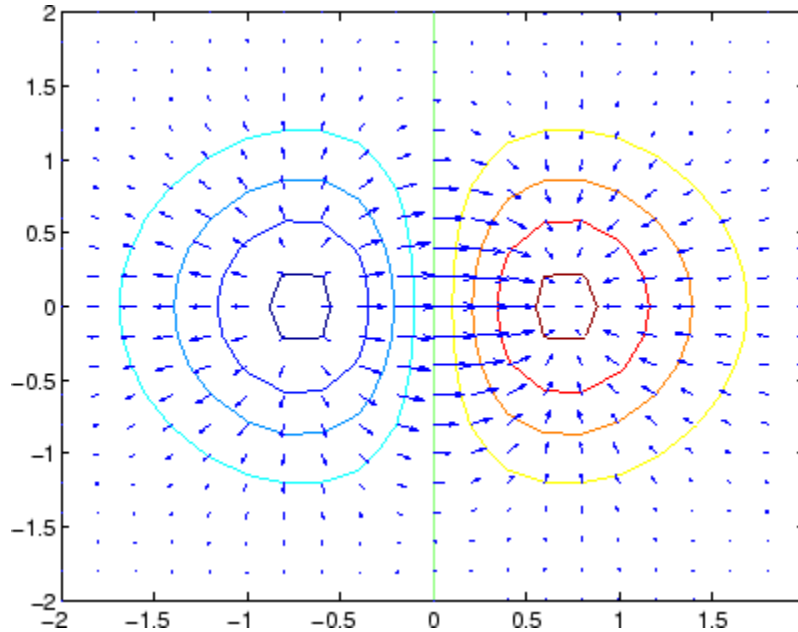
`[...] = gradient(F,h1,h2,...)` with `N` spacing parameters specifies the spacing for each dimension of `F`.

Examples

The statements

```
v = -2:0.2:2;
[x,y] = meshgrid(v);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,.2,.2);
contour(v,v,z), hold on, quiver(v,v,px,py), hold off
```

produce



Given,

```
F(:,:,1) = magic(3); F(:,:,2) = pascal(3);
gradient(F)
```

takes `dx = dy = dz = 1`.

```
[PX,PY,PZ] = gradient(F,0.2,0.1,0.2)
```

takes `dx = 0.2`, `dy = 0.1`, and `dz = 0.2`.

See Also

`del2`, `diff`

graymon

Purpose

Set default figure properties for grayscale monitors

Syntax

```
graymon
```

Description

`graymon` sets defaults for graphics properties to produce more legible displays for grayscale monitors.

See Also

`axes`, `figure`

"Color Operations" on page 1-98 for related functions

Purpose

Grid lines for 2-D and 3-D plots

GUI Alternative

To control the presence and appearance of grid lines on a graph, use the Property Editor, one of the plotting tools  . For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
grid on
grid off
grid
grid(axes_handle,...)
grid minor
```

Description

The `grid` function turns the current axes' grid lines on and off.

`grid on` adds major grid lines to the current axes.

`grid off` removes major and minor grid lines from the current axes.

`grid` toggles the major grid visibility state.

`grid(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

Algorithm

`grid` sets the `XGrid`, `YGrid`, and `ZGrid` properties of the axes.

`grid minor` sets the `XMinorGrid`, `YMinorGrid`, and `ZMinorGrid` properties of the axes.

You can set the grid lines for just one axis using the `set` command and the individual property. For example,

```
set(axes_handle,'XGrid','on')
```

turns on only *x*-axis grid lines.

You can set grid line width with the axes `LineWidth` property.

See Also

`box`, `axes`, `set`

The properties of axes objects

“Axes Operations” on page 1-96 for related functions

Purpose

Data gridding

Syntax

```
ZI = griddata(x,y,z,XI,YI)
[XI,YI,ZI] = griddata(x,y,z,XI,YI)
[...] = griddata(...,method)
[...] = griddata(...,method,options)
```

Description

`ZI = griddata(x,y,z,XI,YI)` fits a surface of the form `z = f(x,y)` to the data in the (usually) nonuniformly spaced vectors `(x,y,z)`. `griddata` interpolates this surface at the points specified by `(XI,YI)` to produce `ZI`. The surface always passes through the data points. `XI` and `YI` usually form a uniform grid (as produced by `meshgrid`).

`XI` can be a row vector, in which case it specifies a matrix with constant columns. Similarly, `YI` can be a column vector, and it specifies a matrix with constant rows.

`[XI,YI,ZI] = griddata(x,y,z,XI,YI)` returns the interpolated matrix `ZI` as above, and also returns the matrices `XI` and `YI` formed from row vector `XI` and column vector `yi`. These latter are the same as the matrices returned by `meshgrid`.

`[...] = griddata(...,method)` uses the specified interpolation method:

'linear'	Triangle-based linear interpolation (default)
'cubic'	Triangle-based cubic interpolation
'nearest'	Nearest neighbor interpolation
'v4'	MATLAB 4 `griddata` method

The `method` defines the type of surface fit to the data. The `'cubic'` and `'v4'` methods produce smooth surfaces while `'linear'` and `'nearest'` have discontinuities in the first and zero'th derivatives, respectively. All the methods except `'v4'` are based on a Delaunay triangulation of the data. If `method` is `[]`, then the default `'linear'` method is used.

`[...] = griddata(...,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`. If `options` is `[]`, the default `delaunayn` options are used. If `options` is `{''}`, no options are used, not even the default.

Occasionally, `griddata` might return points on or very near the convex hull of the data as `NaN`s. This is because roundoff in the computations sometimes makes it difficult to determine if a point near the boundary is in the convex hull.

Remarks

`XI` and `YI` can be matrices, in which case `griddata` returns the values for the corresponding points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `griddata` interprets these vectors as if they were matrices produced by the command `meshgrid(xi,yi)`.

Examples

Sample a function at 100 random points between ±2.0:

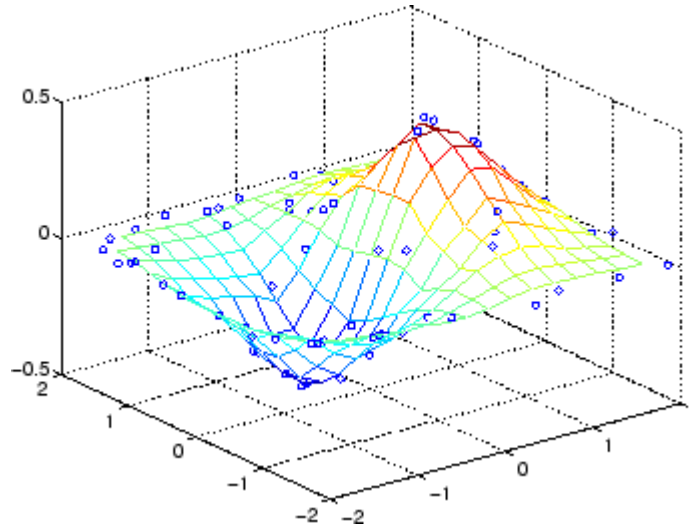
```
rand('seed',0)
x = rand(100,1)*4-2;  y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

`x`, `y`, and `z` are now vectors containing nonuniformly sampled data. Define a regular grid, and grid the data to it:

```
ti = -2:.25:2;
[XI,YI] = meshgrid(ti,ti);
ZI = griddata(x,y,z,XI,YI);
```

Plot the gridded data along with the nonuniform data points used to generate it:

```
mesh(XI,YI,ZI), hold
plot3(x,y,z,'o'), hold off
```



Algorithm

The `griddata(...,'v4')` command uses the method documented in [2]. The other `griddata` methods are based on a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`delaunay`, `griddata3`, `griddatan`, `interp2`, `meshgrid`

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/.

[2] Sandwell, David T., "Biharmonic Spline Interpolation of GEOS-3 and SEASAT Altimeter Data", *Geophysical Research Letters*, 14, 2, 139-142,1987.

[3] Watson, David E., *Contouring: A Guide to the Analysis and Display of Spatial Data*, Tarrytown, NY: Pergamon (Elsevier Science, Inc.): 1992.

Purpose

Data gridding and hypersurface fitting for 3-D data

Syntax

```
w = griddata3(x,y,z,v,xi,yi,zi)
w = griddata3(x,y,z,v,xi,yi,zi,method)
w = griddata3(x,y,z,v,xi,yi,zi,method,options)
```

Description

`w = griddata3(x,y,z,v,xi,yi,zi)` fits a hypersurface of the form $w = f(x, y, z)$ to the data in the (usually) nonuniformly spaced vectors (`x`, `y`, `z`, `v`). `griddata3` interpolates this hypersurface at the points specified by (`xi`,`yi`,`zi`) to produce `w`. `w` is the same size as `xi`, `yi`, and `zi`.

(`xi`,`yi`,`zi`) is usually a uniform grid (as produced by `meshgrid`) and is where `griddata3` gets its name.

`w = griddata3(x,y,z,v,xi,yi,zi,method)` defines the type of surface that is fit to the data, where `method` is either:

`'linear'`	Tesselation-based linear interpolation (default)
`'nearest'`	Nearest neighbor interpolation

If `method` is `[]`, the default `'linear'` method is used.

`w = griddata3(x,y,z,v,xi,yi,zi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default.

Examples

Create vectors `x`, `y`, and `z` containing nonuniformly sampled data:

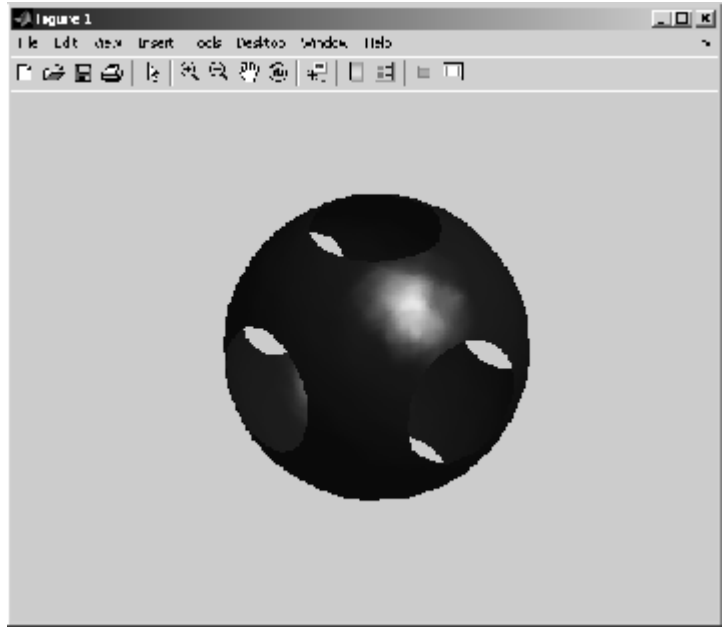
```
rand('state',0);
x = 2*rand(5000,1)-1;
y = 2*rand(5000,1)-1;
z = 2*rand(5000,1)-1;
v = x.^2 + y.^2 + z.^2;
```

Define a regular grid, and grid the data to it:

```
d = -0.8:0.05:0.8;
[xi,yi,zi] = meshgrid(d,d,d);
w = griddata3(x,y,z,v,xi,yi,zi);
```

Since it is difficult to visualize 4D data sets, use isosurface at 0.8:

```
p = patch(isosurface(xi,yi,zi,w,0.8));
isonormals(xi,yi,zi,w,p);
set(p,'FaceColor','blue','EdgeColor','none');
view(3), axis equal, axis off, camlight, lighting phong
```



Algorithm

The `griddata3` methods are based on a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

delaunayn, griddata, griddatan, meshgrid

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/.

griddatan

Purpose

Data gridding and hypersurface fitting (dimension >= 2)

Syntax

```
yi = griddatan(X,y,xi)
yi = griddatan(x,y,z,v,xi,yi,zi,method)
```

Description

`yi = griddatan(X,y,xi)` fits a hyper-surface of the form $y = f(X)$ to the data in the (usually) nonuniformly-spaced vectors (`X`, `y`). `griddatan` interpolates this hyper-surface at the points specified by `xi` to produce `yi`. `xi` can be nonuniform.

`X` is of dimension `m`-by-`n`, representing `m` points in n-dimensional space. `y` is of dimension `m`-by-`1`, representing `m` values of the hyper-surface f(`X`). `xi` is a vector of size `p`-by-`n`, representing `p` points in the n-dimensional space whose surface value is to be fitted. `yi` is a vector of length `p` approximating the values f(`xi`). The hypersurface always goes through the data points (`X`,`y`). `xi` is usually a uniform grid (as produced by `meshgrid`).

`yi = griddatan(x,y,z,v,xi,yi,zi,method)` defines the type of surface fit to the data, where `'method'` is one of:

'linear'	Tessellation-based linear interpolation (default)
'nearest'	Nearest neighbor interpolation

All the methods are based on a Delaunay tessellation of the data.

If `method` is `[]`, the default `'linear'` method is used.

`yi = griddatan(x,y,z,v,xi,yi,zi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default.

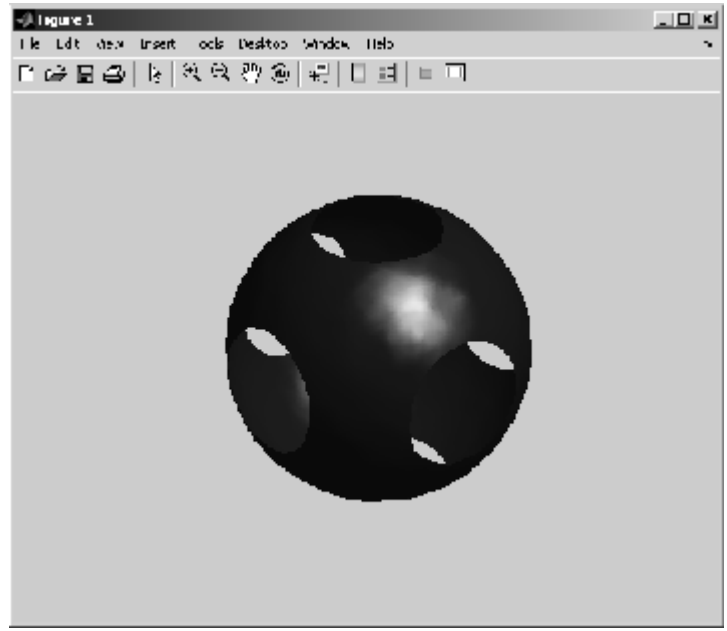
Examples

```
rand('state',0)
X = 2*rand(5000,3)-1;
Y = sum(X.^2,2);
d = -0.8:0.05:0.8;
```

```
[yO,xO,zO] = ndgrid(d,d,d);
XI = [xO(:) yO(:) zO(:)];
YI = griddatan(X,Y,XI);
```

Since it is difficult to visualize 4D data sets, use isosurface at 0.8:

```
YI = reshape(YI, size(xO));
p = patch(isosurface(xO,yO,zO,YI,0.8));
isonormals(xO,yO,zO,YI,p);
set(p,'FaceColor','blue','EdgeColor','none');
view(3), axis equal, axis off, camlight, lighting phong
```



Algorithm

The `griddatan` methods are based on a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

delaunayn, griddata, griddata3, meshgrid

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/.

Purpose

Generalized singular value decomposition

Syntax

```
[U,V,X,C,S] = gsvd(A,B)
sigma = gsvd(A,B)
```

Description

`[U,V,X,C,S] = gsvd(A,B)` returns unitary matrices `U` and `V`, a (usually) square matrix `X`, and nonnegative diagonal matrices `C` and `S` so that

```
A = U*C*X'
B = V*S*X'
C'*C + S'*S = I
```

`A` and `B` must have the same number of columns, but may have different numbers of rows. If `A` is `m`-by-`p` and `B` is `n`-by-`p`, then `U` is `m`-by-`m`, `V` is `n`-by-`n` and `X` is `p`-by-`q` where `q = min(m+n,p)`.

`sigma = gsvd(A,B)` returns the vector of generalized singular values, `sqrt(diag(C'*C)./diag(S'*S))`.

The nonzero elements of `S` are always on its main diagonal. If `m >= p` the nonzero elements of `C` are also on its main diagonal. But if `m < p`, the nonzero diagonal of `C` is `diag(C,p-m)`. This allows the diagonal elements to be ordered so that the generalized singular values are nondecreasing.

`gsvd(A,B,0)`, with three input arguments and either `m` or `n >= p`, produces the "economy-sized"decomposition where the resulting `U` and `V` have at most `p` columns, and `C` and `S` have at most `p` rows. The generalized singular values are `diag(C)./diag(S)`.

When `B` is square and nonsingular, the generalized singular values, `gsvd(A,B)`, are equal to the ordinary singular values, `svd(A/B)`, but they are sorted in the opposite order. Their reciprocals are `gsvd(B,A)`.

In this formulation of the `gsvd`, no assumptions are made about the individual ranks of `A` or `B`. The matrix `X` has full rank if and only if the matrix `[A;B]` has full rank. In fact, `svd(X)` and `cond(X)` are equal to `svd([A;B])` and `cond([A;B])`. Other formulations, eg. G. Golub and

C. Van Loan [1], require that `null(A)` and `null(B)` do not overlap and replace `X` by `inv(X)` or `inv(X')`.

Note, however, that when `null(A)` and `null(B)` do overlap, the nonzero elements of `C` and `S` are not uniquely determined.

Examples

Example 1

The matrices have at least as many rows as columns.

```
A = reshape(1:15,5,3)
B = magic(3)
A =
          1     6    11
          2     7    12
          3     8    13
          4     9    14
          5    10    15
B =
          8     1     6
          3     5     7
          4     9     2
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 5-by-5 orthogonal `U`, a 3-by-3 orthogonal `V`, a 3-by-3 nonsingular `X`,

```
X =
         2.8284   -9.3761   -6.9346
        -5.6569   -8.3071  -18.3301
         2.8284   -7.2381  -29.7256
```

and

```
C =
         0.0000         0         0
```

```
         0    0.3155         0
         0         0    0.9807
         0         0         0
         0         0         0
S =
    1.0000         0         0
         0    0.9489         0
         0         0    0.1957
```

Since `A` is rank deficient, the first diagonal element of `C` is zero.

The economy sized decomposition,

```
[U,V,X,C,S] = gsvd(A,B,0)
```

produces a 5-by-3 matrix `U` and a 3-by-3 matrix `C`.

```
U =
    0.5700   -0.6457   -0.4279
   -0.7455   -0.3296   -0.4375
   -0.1702   -0.0135   -0.4470
    0.2966    0.3026   -0.4566
    0.0490    0.6187   -0.4661

C =
    0.0000         0         0
         0    0.3155         0
         0         0    0.9807
```

The other three matrices, `V`, `X`, and `S` are the same as those obtained with the full decomposition.

The generalized singular values are the ratios of the diagonal elements of `C` and `S`.

```
sigma = gsvd(A,B)
sigma =
    0.0000
    0.3325
```

```
        5.0123
```

These values are a reordering of the ordinary singular values

```
svd(A/B)
ans =
        5.0123
        0.3325
        0.0000
```

Example 2

The matrices have at least as many columns as rows.

```
A = reshape(1:15,3,5)
B = magic(5)
A =

     1     4     7    10    13
     2     5     8    11    14
     3     6     9    12    15
B =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 3-by-3 orthogonal `U`, a 5-by-5 orthogonal `V`, a 5-by-5 nonsingular `X` and

```
C =
             0         0    0.0000         0         0
             0         0         0    0.0439         0
             0         0         0         0    0.7432
```

```
S =
        1.0000         0         0         0         0
             0    1.0000         0         0         0
             0         0    1.0000         0         0
             0         0         0    0.9990         0
             0         0         0         0    0.6690
```

In this situation, the nonzero diagonal of `C` is `diag(C,2)`. The generalized singular values include three zeros.

```
sigma = gsvd(A,B)
sigma =
             0
             0
        0.0000
        0.0439
        1.1109
```

Reversing the roles of `A` and `B` reciprocates these values, producing two infinities.

```
gsvd(B,A)
ans =
      1.0e+016 *

        0.0000
        0.0000
        4.4126
           Inf
           Inf
```

Algorithm

The generalized singular value decomposition uses the C-S decomposition described in [1], as well as the built-in `svd` and `qr` functions. The C-S decomposition is implemented in a subfunction in the `gsvd` M-file.

Diagnostics

The only warning or error message produced by `gsvd` itself occurs when the two input arguments do not have the same number of columns.

See Also

`qr`, `svd`

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

Purpose

Test for greater than

Syntax

```
A > B
gt(A, B)
```

Description

`A > B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is greater than `B`, or set to logical 0 (`false`) where `A` is less than or equal to `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then MATLAB returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`gt(A, B)` is called for the syntax `A>B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are greater than the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A > B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     0     1
     0     1     1     0     1     0
```

```
     1     0     1     1     0     0
     0     1     1     1     0     1
```

See Also

`lt`, `ge`, `le`, `ne`, `eq`, "Relational Operators"

Purpose

Mouse placement of text in 2-D view

Syntax

```
gtext('string')
gtext({'string1','string2','string3',...})
gtext({'string1';'string2';'string3';...})
h = gtext(...)
```

Description

`gtext` displays a text string in the current figure window after you select a location with the mouse.

`gtext('string')` waits for you to press a mouse button or keyboard key while the pointer is within a figure window. Pressing a mouse button or any key places `'string'` on the plot at the selected location.

`gtext({'string1','string2','string3',...})` places all strings with one click, each on a separate line.

`gtext({'string1';'string2';'string3';...})` places one string per click, in the sequence specified.

`h = gtext(...)` returns the handle to a text graphics object that is placed on the plot at the location you select.

Remarks

As you move the pointer into a figure window, the pointer becomes crosshairs to indicate that `gtext` is waiting for you to select a location. `gtext` uses the functions `ginput` and `text`.

Examples

Place a label on the current plot:

```
gtext('Note this divergence!')
```

See Also

`ginput`, `text`

"Annotating Plots" on page 1-87 for related functions

guidata

Purpose

Store or retrieve GUI data

Syntax

```
guidata(object_handle,data)
data = guidata(object_handle)
```

Description

`guidata(object_handle,data)` stores the variable `data` as GUI data. If `object_handle` is not a figure handle, then the object's parent figure is used. `data` can be any MATLAB variable, but is typically a structure, which enables you to add new fields as required.

`guidata` can manage only one variable at any time. Subsequent calls to `guidata(object_handle,data)` overwrite the previously created version of GUI `data`.

Note for GUIDE Users GUIDE uses `guidata` to store and maintain the `handles` structure. From a GUIDE-generated GUI M-file, do not use `guidata` to store any data other than `handles`. If you do, you may overwrite the `handles` structure and your GUI will not work. If you need to store other data with your GUI, you can add it to the `handles` structure. See GUI Data in the MATLAB documentation.

`data = guidata(object_handle)` returns previously stored data, or an empty matrix if nothing has been stored.

To change the data managed by `guidata`:

1. Get a copy of the data with the command `data = guidata(object_handle)`.
2. Make the desired changes to `data`.
3. Save the changed version of `data` with the command `guidata(object_handle,data)`.

`guidata` provides application developers with a convenient interface to a figure's application data:

- You do not need to create and maintain a hard-coded property name for the application data throughout your source code.
- You can access the data from within a subfunction callback routine using the component's handle (which is returned by `gcbo`), without needing to find the figure's handle.

If you are not using GUIDE, `guidata` is particularly useful in conjunction with `guihandles`, which creates a structure containing the handles of all the components in a GUI.

Examples

In this example, `guidata` is used to save a structure on a GUI figure's application data from within the initialization section of the application M-file. This structure is initially created by `guihandles` and then used to save additional data as well.

```
% create structure of handles
myhandles = guihandles(figure_handle);
% add some additional data
myhandles.numberOfErrors = 0;
% save the structure
guidata(figure_handle,myhandles)
```

You can recall the data from within a subfunction callback routine and then save the structure again:

```
% get the structure in the subfunction
myhandles = guidata(gcbo);
myhandles.numberOfErrors = myhandles.numberOfErrors + 1;
% save the changes to the structure
guidata(gcbo,myhandles)
```

See Also

`guide`, `guihandles`, `getappdata`, `setappdata`

guide

Purpose

Open GUI Layout Editor

Syntax

```
guide
guide('filename.fig')
guide('fullpath')
guide(HandleList)
```

Description

`guide` initiates the GUI design environment (GUIDE) tools that allow you to create or edit GUIs interactively.

`guide` opens the GUIDE Quick Start dialog where you can choose to open a previously created GUI or create a new one using one of the provided templates.

`guide('filename.fig')` opens the FIG-file named `filename.fig` for editing if it is on the MATLAB path.

`guide('fullpath')` opens the FIG-file at `fullpath` even if it is not on the MATLAB path.

`guide(HandleList)` opens the content of each of the figures in `HandleList` in a separate copy of the GUIDE design environment.

See Also

`inspect`

Creating GUIs

Purpose

Create structure of handles

Syntax

```
handles = guihandles(object_handle)
handles = guihandles
```

Description

`handles = guihandles(object_handle)` returns a structure containing the handles of the objects in a figure, using the value of their `Tag` properties as the fieldnames, with the following caveats:

- Objects are excluded if their `Tag` properties are empty, or are not legal variable names.
- If several objects have the same `Tag`, that field in the structure contains a vector of handles.
- Objects with hidden handles are included in the structure.

`handles = guihandles` returns a structure of handles for the current figure.

See Also

`guidata`, `guide`, `getappdata`, `setappdata`

gunzip

Purpose

Uncompress GNU zip files

Syntax

```
gunzip(files)
gunzip(files,outputdir)
gunzip(url, ...)
filenames = gunzip(...)
```

Description

`gunzip(files)` uncompresses GNU zip files from the list of files specified in `files`. Directories recursively gunzip all of their content. The output files have the same name, excluding the extension `.gz`, and are written to the same directory as the input files.

`files` is a string or cell array of strings containing a list of files or directories. Individual files that are on the MATLAB path can be specified as partial pathnames. Otherwise an individual file can be specified relative to the current directory or with an absolute path. Directories must be specified relative to the current directory or with absolute paths. On UNIX systems, directories can also start with `~/` or `~username/`, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a filename or partial pathname.

`gunzip(files,outputdir)` writes the gunzipped file into the directory `outputdir`. `outputdir` is created if it does not exist.

`gunzip(url, ...)` extracts the GNU zip contents from an Internet universal resource locator (URL). The URL must include the protocol type (e.g., `'http://'`). The URL is downloaded to the temp directory and deleted.

`filenames = gunzip(...)` gunzips the files and returns the relative pathnames of the gunzipped files in the string cell array `filenames`.

Examples

To gunzip all `.gz` files in the current directory,

```
gunzip('*.gz');
```

To gunzip Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`:

```
url ='http://www.mathworks.com/moler/ncm.tar.gz';
gunzip(url,'ncm')
untar('ncm/ncm.tar','ncm')
```

See Also

`gzip`, `tar`, `untar`, `unzip`, `zip`

gzip

Purpose

Compress files into GNU zip files

Syntax

```
gzip(files)
gzip(files,outputdir)
filenames = gzip(...)
```

Description

`gzip(files)` creates GNU zip files from the list of files specified in `files`. Directories recursively `gzip` all their contents. Each output gzipped file is written to the same directory as the input file and with the file extension `.gz`.

`files` is a string or cell array of strings containing a list of files or directories to `gzip`. Individual files that are on the MATLAB path can be specified as partial pathnames. Otherwise an individual file can be specified relative to the current directory or with an absolute path. Directories must be specified relative to the current directory or with absolute paths. On UNIX systems, directories can also start with `~/` or *`~username/`*, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a filename or partial pathname.

`gzip(files,outputdir)` writes the gzipped files into the directory `outputdir`. `outputdir` is created if it does not exist.

`filenames = gzip(...)` gzips the files and returns the relative pathnames of all gzipped files in the string cell array `filenames`.

Example

To `gzip` all `.m` and `.mat` files in the current directory and store the results in the directory `archive`,

```
gzip({'*.m','*.mat'},'archive');
```

See Also

`gunzip`, `tar`, `untar`, `unzip`, `zip`

Purpose

Hadamard matrix

Syntax

```
H = hadamard(n)
```

Description

`H = hadamard(n)` returns the Hadamard matrix of order `n`.

Definition

Hadamard matrices are matrices of 1's and `-1`'s whose columns are orthogonal,

```
H'*H = n*I
```

where `[n n]=size(H)` and `I = eye(n,n)` ,.

They have applications in several different areas, including combinatorics, signal processing, and numerical analysis, [1], [2].

An `n`-by-`n` Hadamard matrix with `n > 2` exists only if `rem(n,4) = 0`. This function handles only the cases where `n`, `n/12`, or `n/20` is a power of 2.

Examples

The command `hadamard(4)` produces the 4-by-4 matrix:

```
   1     1     1     1
   1    -1     1    -1
   1     1    -1    -1
   1    -1    -1     1
```

See Also

`compan`, `hankel`, `toeplitz`

References

[1] Ryser, H. J., *Combinatorial Mathematics*, John Wiley and Sons, 1963.

[2] Pratt, W. K., *Digital Signal Processing*, John Wiley and Sons, 1978.

hankel

Purpose

Hankel matrix

Syntax

```
H = hankel(c)
H = hankel(c,r)
```

Description

`H = hankel(c)` returns the square Hankel matrix whose first column is `c` and whose elements are zero below the first anti-diagonal.

`H = hankel(c,r)` returns a Hankel matrix whose first column is `c` and whose last row is `r`. If the last element of `c` differs from the first element of `r`, the last element of `c` prevails.

Definition

A Hankel matrix is a matrix that is symmetric and constant across the anti-diagonals, and has elements `h(i,j) = p(i+j-1)`, where vector `p = [c r(2:end)]` completely determines the Hankel matrix.

Examples

A Hankel matrix with anti-diagonal disagreement is

```
c = 1:3; r = 7:10;
h = hankel(c,r)
h =
     1     2     3     8
     2     3     8     9
     3     8     9    10

p = [1 2 3 8 9 10]
```

See Also

`hadamard`, `toeplitz`, `kron`

Purpose

Summary of MATLAB HDF4 capabilities

Description

MATLAB provides a set of low-level functions that enable you to access the HDF4 library developed by the National Center for Supercomputing Applications (NCSA). For information about HDF4, go to the HDF Web page at `http://www.hdfgroup.org`.

Note For information about MATLAB HDF5 capabilities, which is a completely separate, incompatible format, see `hdf5`.

The following table lists all the HDF4 application programming interfaces (APIs) supported by MATLAB with the name of the MATLAB function used to access the API. To use these functions, you must be familiar with the HDF library. For more information about using these MATLAB functions, see Working with Scientific Data Formats.

Application Programming Interface	Description	MATLAB Function
Annotations	Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.	`hdfan`
General Raster Images	Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes. Note: Use the MATLAB functions `imread` and `imwrite` with HDF raster image formats.	`hdfdf24`, `hdfdfr8`

Application Programming Interface	Description	MATLAB Function
HDF-EOS	Provides functions to read HDF-EOS grid (GD), point (PT), and swath (SW) data.	`hdfgd`, `hdfpt`, `hdfsw`
HDF Utilities	Provides functions to open and close HDF files and handle errors.	`hdfh`, `hdfhd`, `hdfhe`
MATLAB HDF Utilitie	Provides utility functions that help you work with HDF files in the MATLAB environment.	`hdfml`
Scientific Data	Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.	`hdfsd`
V Groups	Creates and retrieves groups of other HDF data objects, such as raster images or V data.	`hdfv`
V Data	Stores, manages, and retrieves multivariate data stored as records in a table.	`hdfvf`, `hdfvh`, `hdfvs`

See Also

`hdfinfo`,`hdfread`, `hdftool`, `imread`

Purpose

Summary of MATLAB HDF5 capabilities

Description

MATLAB provides both high-level and low-level access to HDF5 files. The high-level access functions make it easy to read a data set from an HDF5 file or write a variable from the MATLAB workspace into an HDF5 file. The MATLAB low-level interface provides direct access to the more than 200 functions in the HDF5 library. MATLAB currently supports version HDF5-1.6.5 of the library.

Note For information about MATLAB HDF4 capabilities, which is a completely separate, incompatible format, see `hdf`.

The following sections provide an overview of both this high- and low-level access. To use these MATLAB functions, you must be familiar with HDF5 programming concepts and, when using the low-level functions, details about the functions in the library. To get this information, go to the HDF Web page at `http://www.hdfgroup.org`.

High-level Access

MATLAB includes three functions that provide high-level access to HDF5 files:

- `hdf5info`
- `hdf5read`
- `hdf5write`

Using these functions you can read data and metadata from an HDF5 file and write data from the MATLAB workspace to a file in HDF5 format. For more information about these functions, see their individual reference pages.

Low-level Access

MATLAB provides direct access to the over 200 functions in the HDF5 Library. Using these functions, you can read and write complex

datatypes, utilize HDF5 data subsetting capabilities, and take advantage of other features present in the HDF5 library.

The HDF5 library organizes the routines in the library into interfaces. MATLAB organizes the corresponding MATLAB functions into class directories that match these HDF5 library interfaces. For example, the MATLAB functions for the HDF5 Attribute Interface are in the @H5A class directory.

The following table lists all the HDF5 library interfaces in alphabetical order by name. The table includes the name of the associated MATLAB class directory.

HDF5 Library Interface	MATLAB Class Directory	Description
Attribute	`@H5A`	Manipulate metadata associated with data sets or groups
Dataset	`@H5D`	Manipulate multidimensional arrays of data elements, together with supporting metadata
Dataspace	`@H5S`	Define and work with data spaces, which describe the the dimensionality of a data set
Datatype	`@H5T`	Define the type of variable that is stored in a data set
Error	`@H5E`	Handle errors
File	`@H5F`	Access files
Filters and Compression	`@H5Z`	Create inline data filters and data compression
Group	`@H5G`	Organize objects in a file; analogous to a directory structure
Identifier	`@H5I`	Manipulate HDF5 object identifiers

HDF5 Library Interface	MATLAB Class Directory	Description
Library	`@H5`	General-purpose functions for use with the entire HDF5 library, such as initialization
MATLAB	`@H5ML`	MATLAB utility functions that are not part of the HDF5 library itself.
Property	`@H5P`	Manipulate object property lists
Reference	`@H5R`	Manipulate HDF5 references, which are like UNIX links or Windows shortcuts

In most cases, the syntax of the MATLAB function is identical to the syntax of the HDF5 library function. To get detailed information about the MATLAB syntax of an HDF5 library function, view the help for the individual MATLAB function, as follows:

```
help @H5F/open
```

To view a list of all the MATLAB HDF5 functions in a particular interface, type:

```
help imagesci/@H5F
```

See Also

`hdf`, `hdf5info`, `hdf5read`, `hdf5write`

hdf5info

Purpose

Information about HDF5 file

Syntax

```
fileinfo = hdf5info(filename)
fileinfo = hdf5info(...,'ReadAttributes',BOOL)
[...] = hdf5info(..., 'V71Dimensions', BOOL)
```

Description

`fileinfo = hdf5info(filename)` returns a structure `fileinfo` whose fields contain information about the contents of the HDF5 file `filename`. `filename` is a string that specifies the name of the HDF5 file.

`fileinfo = hdf5info(...,'ReadAttributes',BOOL)` specifies whether `hdf5info` returns the values of the attributes or just information describing the attributes. By default, `hdf5info` reads in attribute values (`BOOL = true`).

`[...] = hdf5info(..., 'V71Dimensions', BOOL)` specifies whether to report the dimensions of data sets and attributes as they were returned in previous versions of `hdf5info` (MATLAB 7.1 [R14SP3] and earlier). If `BOOL` is true, `hdf5info` swaps the first two dimensions of the data set. This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimenions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, swapping these dimensions may not correctly reflect the intent of the data in the file and may invalidate metadata. When `BOOL` is false (the default), `hdf5info` returns data dimensions that correctly reflect the data ordering as it is written in the file—each dimension in the output variable matches the same dimension in the file.

Note If you use the `'V71Dimensions'` parameter and intend on passing the `fileinfo` structure returned to the `hdf5read` function, you should also specify the `'V71Dimensions'` parameters with `hdf5read`. If you do not, `hdf5read` uses the new behavior when reading the data set and certain metadata returned by `hdf5info` does not match the actual data returned by `hdf5read`.

Examples

```
fileinfo = hdf5info('example.h5')

fileinfo =

          Filename: 'example.h5'
        LibVersion: '1.4.5'
            Offset: 0
          FileSize: 8172
    GroupHierarchy: [1x1 struct]
```

To get more information about the contents of the HDF5 file, look at the `GroupHierarchy` field in the `fileinfo` structure returned by `hdf5info`.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: [1x64 char]
          Name: '/'
        Groups: [1x2 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [1x2 struct]
```

To probe further into the file hierarchy, keep examining the `Groups` field.

See also

`hdf5read`, `hdf5write`

hdf5read

Purpose

Read HDF5 file

Syntax

```
data = hdf5read(filename,datasetname)
attr = hdf5read(filename,attributename)
[data, attr] = hdf5read(...,'ReadAttributes',BOOL)
data = hdf5read(hinfo)
[...] = hdf5read(..., 'V71Dimensions', BOOL)
```

Description

`data = hdf5read(filename,datasetname)` reads all the data in the data set `datasetname` that is stored in the HDF5 file `filename` and returns it in the variable `data`. To determine the names of data sets in an HDF5 file, use the `hdf5info` function.

The return value, `data`, is a multidimensional array. `hdf5read` maps HDF5 data types to native MATLAB data types, whenever possible. If it cannot represent the data using MATLAB data types, `hdf5read` uses one of the HDF5 data type objects. For example, if an HDF5 file contains a data set made up of an enumerated data type, `hdf5read` uses the `hdf5.h5enum` object to represent the data in the MATLAB workspace. The `hdf5.h5enum` object has data members that store the enumerations (names), their corresponding values, and the enumerated data. For more information about the HDF5 data type objects, see the `hdf5` reference page.

`attr = hdf5read(filename,attributename)` reads all the metadata in the attribute `attributename`, stored in the HDF5 file `filename`, and returns it in the variable `attr`. To determine the names of attributes in an HDF5 file, use the `hdf5info` function.

`[data, attr] = hdf5read(...,'ReadAttributes',BOOL)` reads all the data, as well as all of the associated attribute information contained within that data set. By default, `BOOL` is false.

`data = hdf5read(hinfo)` reads all of the data in the data set specified in the structure `hinfo` and returns it in the variable `data`. The `hinfo` structure is extracted from the output returned by `hdf5info`, which specifies an HDF5 file and a specific data set.

`[...] = hdf5read(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets read from the file. If `BOOL` is true, `hdf5read` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data dimensions correctly reflect the data ordering as it is written in the file — each dimension in the output variable matches the same dimension in the file.

Examples

Use `hdf5info` to get information about an HDF5 file and then use `hdf5read` to read a data set, using the information structure (`hinfo`) returned by `hdf5info` to specify the data set.

```
hinfo = hdf5info('example.h5');
dset = hdf5read(hinfo.GroupHierarchy.Groups(2).Datasets(1));
```

See Also

`hdf5`, `hdf5info`, `hdf5write`

Purpose

Write data to file in HDF5 format

Syntax

```
hdf5write(filename,location,dataset)
hdf5write(filename,details,dataset)
hdf5write(filename,details,attribute)
hdf5write(filename, details1, dataset1, details2, dataset2,
   ...)
hdf5write(filename,...,'WriteMode',mode,...)
hdf5write(..., 'V71Dimensions', BOOL)
```

Description

`hdf5write(filename,location,dataset)` writes the data `dataset` to the HDF5 file, `filename`. If `filename` does not exist, `hdf5write` creates it. If `filename` exists, `hdf5write` overwrites the existing file, by default, but you can also append data to an existing file using an optional syntax.

`location` defines where to write the data set in the file. HDF5 files are organized in a hierarchical structure similar to a UNIX directory structure. `location` is a string that resembles a UNIX path.

`hdf5write` maps the data in `dataset` to HDF5 data types according to rules outlined below.

`hdf5write(filename,details,dataset)` writes `dataset` to `filename` using the values in the `details` structure. For a data set, the `details` structure can contain the following fields.

Field Name	Description	Data Type
`Location`	Location of the data set in the file	Character array
`Name`	Name to attach to the data set	Character array

`hdf5write(filename,details,attribute)` writes the metadata `attribute` to `filename` using the values in the `details` structure. For an attribute, the `details` structure can contain following fields.

Field Name	Description	Data Type
`AttachedTo`	Location of the object this attribute modifies	Structure array
`AttachType`	Identifies what kind of object this attribute modifies; possible values are `'group'` and `'dataset'`	Character array
`Name`	Name to attach to the data set	Character array

`hdf5write(filename, details1, dataset1, details2, dataset2,...)` writes multiple data sets and associated attributes to `filename` in one operation. Each data set and attribute must have an associated `details` structure.

`hdf5write(filename,...,'WriteMode',mode,...)` specifies whether `hdf5write` overwrites the existing file (the default) or appends data sets and attributes to the file. Possible values for mode are `'overwrite'` and `'append'`.

`hdf5write(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets written to the file. If `BOOL` is true, `hdf5write` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data written to the file correctly reflects the data ordering of the data sets — each dimension in the file's data sets matches the same dimension in the corresponding MATLAB variable.

Data Type Mappings

The following table lists how `hdf5write` maps the data type from the workspace into an HDF5 file. If the data in the workspace that is being written to the file is a MATLAB data type, `hdf5write` uses the following rules when translating MATLAB data into HDF5 data objects.

MATLAB Data Type	HDF5 Data Set or Attribute
Numeric	Corresponding HDF5 native data type. For example, if the workspace data type is `uint8`, the `hdf5write` function writes the data to the file as 8-bit integers. The size of the HDF5 dataspace is the same size as the MATLAB array.
String	Single, null-terminated string
Cell array of strings	Multiple, null-terminated strings, each the same length. Length is determined by the length of the longest string in the cell array. The size of the HDF5 dataspace is the same size as the cell array.
Cell array of numeric data	Numeric array, the same dimensions as the cell array. The elements of the array must all have the same size and type. The data type is determined by the first element in the cell array.
Structure array	HDF5 compound type. Individual fields in the structure employ the same data translation rules for individual data types. For example, a cell array of strings becomes a multiple, null-terminated strings.
HDF5 objects	If the data being written to the file is composed of HDF5 objects, `hdf5write` uses the same data type when writing to the file. For all HDF5 objects, except `HDF5.h5enum` objects, the dataspace has the same dimensions as the array of HDF5 objects passed to the function. For `HDF5.h5enum` objects, the size and dimensions of the data set in the HDF5 file is the same as the object's Data field.

Examples

Write a 5-by-5 data set of `uint8` values to the root group.

```
hdf5write('myfile.h5', '/dataset1', uint8(magic(5)))
```

Write a 2-by-2 string data set in a subgroup.

```
dataset = {'north', 'south'; 'east', 'west'};
hdf5write('myfile2.h5', '/group1/dataset1.1', dataset);
```

Write a data set and attribute to an existing group.

```
dset = single(rand(10,10));
dset_details.Location = '/group1/dataset1.2';
dset_details.Name = 'Random';

attr = 'Some random data';
attr_details.Name = 'Description';
attr_details.AttachedTo = '/group1/dataset1.2/Random';
attr_details.AttachType = 'dataset';

hdf5write('myfile2.h5', dset_details, dset, ...
           attr_details, attr, 'WriteMode', 'append');
```

Write a data set using objects.

```
dset = hdf5.h5array(magic(5));
hdf5write('myfile3.h5', '/g1/objects', dset);
```

See Also

hdf5, hdf5read, hdf5info

hdfinfo

Purpose

Information about HDF4 or HDF-EOS file

Syntax

```
S = hdfinfo(filename)
S = hdfinfo(filename,mode)
```

Description

`S = hdfinfo(filename)` returns a structure `S` whose fields contain information about the contents of an HDF4 or HDF-EOS file. `filename` is a string that specifies the name of the HDF4 file.

`S = hdfinfo(filename,mode)` reads the file as an HDF4 file, if `mode` is `'hdf'`, or as an HDF-EOS file, if `mode` is `'eos'`. If `mode` is `'eos'`, only HDF-EOS data objects are queried. To retrieve information on the entire contents of a file containing both HDF4 and HDF-EOS objects, `mode` must be `'hdf'`.

Note `hdfinfo` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To get information about an HDF5 file, use `hdf5info`.

The set of fields in the returned structure `S` depends on the individual file. Fields that can be present in the `S` structure are shown in the following table.

Mode	Field Name	Description	Return Type
`HDF`	`Attributes`	Attributes of the data set	Structure array
	`Description`	Annotation description	Cell array
	`Filename`	Name of the file	String
	`Label`	Annotation label	Cell array
	`Raster8`	Description of 8-bit raster images	Structure array

Mode	Field Name	Description	Return Type
	`Raster24`	Description of 24-bit raster images	Structure array
	`SDS`	Description of scientific data sets	Structure array
	`Vdata`	Description of Vdata sets	Structure array
	`Vgroup`	Description of Vgroups	Structure array
`EOS`	`Filename`	Name of the file	String
	`Grid`	Grid data	Structure array
	`Point`	Point data	Structure array
	`Swath`	Swath data	Structure array

Those fields in the table above that contain structure arrays are further described in the tables shown below.

Fields Common to Returned Structure Arrays

Structure arrays returned by `hdfinfo` contain some common fields. These are shown in the table below. Not all structure arrays will contain all of these fields.

Field Name	Description	Data Type
`Attributes`	Data set attributes. Contains fields `Name` and `Value`.	Structure array
`Description`	Annotation description	Cell array
`Filename`	Name of the file	String
`Label`	Annotation label	Cell array

Field Name	Description	Data Type
`Name`	Name of the data set	String
`Rank`	Number of dimensions of the data set	Double
`Ref`	Data set reference number	Double
`Type`	Type of HDF or HDF-EOS object	String

Fields Specific to Certain Structures

Structure arrays returned by `hdfinfo` also contain fields that are unique to each structure. These are shown in the tables below.

Fields of the Attribute Structure

Field Name	Description	Data Type
`Name`	Attribute name	String
`Value`	Attribute value or description	Numeric or string

Fields of the Raster8 and Raster24 Structures

Field Name	Description	Data Type
`HasPalette`	`1` (`true`) if the image has an associated palette, otherwise `0` (`false`) (8-bit only)	Logical
`Height`	Height of the image, in pixels	Number
`Interlace`	Interlace mode of the image (24-bit only)	String

Fields of the Raster8 and Raster24 Structures (Continued)

Field Name	Description	Data Type
`Name`	Name of the image	String
`Width`	Width of the image, in pixels	Number

Fields of the SDS Structure

Field Name	Description	Data Type
`DataType`	Data precision	String
`Dims`	Dimensions of the data set. Contains fields `Name`, `DataType`, `Size`, `Scale`, and `Attributes`. `Scale` is an array of numbers to place along the dimension and demarcate intervals in the data set.	Structure array
`Index`	Index of the SDS	Number

Fields of the Vdata Structure

Field Name	Description	Data Type
`DataAttributes`	Attributes of the entire data set. Contains fields `Name` and `Value`.	Structure array
`Class`	Class name of the data set	String
`Fields`	Fields of the Vdata. Contains fields `Name` and `Attributes`.	Structure array

Fields of the Vdata Structure (Continued)

Field Name	Description	Data Type
`NumRecords`	Number of data set records	Double
`IsAttribute`	`1` (`true`) if Vdata is an attribute, otherwise `0` (`false`)	Logical

Fields of the Vgroup Structure

Field Name	Description	Data Type
`Class`	Class name of the data set	String
`Raster8`	Description of the 8-bit raster image	Structure array
`Raster24`	Description of the 24-bit raster image	Structure array
`SDS`	Description of the Scientific Data sets	Structure array
`Tag`	Tag of this Vgroup	Number
`Vdata`	Description of the Vdata sets	Structure array
`Vgroup`	Description of the Vgroups	Structure array

Fields of the Grid Structure

Field Name	Description	Data Type
`Columns`	Number of columns in the grid	Number

Fields of the Grid Structure (Continued)

Field Name	Description	Data Type
`DataFields`	Description of the data fields in each `Grid` field of the grid. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, `FillValue`, and `TileDims`.	Structure array
`LowerRight`	Lower right corner location, in meters	Number
`Origin Code`	Origin code for the grid	Number
`PixRegCode`	Pixel registration code	Number
`Projection`	Projection code, zone code, sphere code, and projection parameters of the grid. Contains fields `ProjCode`, `ZoneCode`, `SphereCode`, and `ProjParam`.	Structure
`Rows`	Number of rows in the grid	Number
`UpperLeft`	Upper left corner location, in meters	Number

Fields of the Point Structure

Field Name	Description	Data Type
`Level`	Description of each level of the point. Contains fields `Name`, `NumRecords`, `FieldNames`, `DataType`, and `Index`.	Structure

Fields of the Swath Structure

Field Name	Description	Data Type
`DataFields`	Data fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`GeolocationFields`	Geolocation fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`IdxMapInfo`	Relationship between indexed elements of the geolocation mapping. Contains fields `Map` and `Size`.	Structure
`MapInfo`	Relationship between data and geolocation fields. Contains fields `Map`, `Offset`, and `Increment`.	Structure

Examples

To retrieve information about the file `example.hdf`,

```
fileinfo = hdfinfo('example.hdf')

fileinfo =
    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

And to retrieve information from this about the scientific data set in `example.hdf`,

```
sds_info = fileinfo.SDS
```

```
sds_info =
       Filename: 'example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
           Dims: [2x1 struct]
          Label: {}
    Description: {}
          Index: 0
```

See Also

hdfread, hdf

Purpose

Read data from HDF4 or HDF-EOS file

Syntax

```
data = hdfread(filename, datasetname)
data = hdfread(hinfo.fieldname)
data = hdfread(...,param1,value1,param2,value2,...)
[data,map] = hdfread(...)
```

Description

`data = hdfread(filename, datasetname)` returns all the data in the data set specified by `datasetname` from the HDF4 or HDF-EOS file specified by `filename`. To determine the name of a data set in an HDF4 file, use the `hdfinfo` function.

Note `hdfread` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To read data from and HDF5 file, use `hdf5read`.

`data = hdfread(hinfo.fieldname)` returns all the data in the data set specified by `hinfo.fieldname`, where `hinfo` is the structure returned by the `hdfinfo` function and *`fieldname`* is the name of a field in the structure that relates to a particular type of data set. For example, to read an HDF scientific data set, specify the `SDS` field, as in `hinfo.SDS`. To read HDF V data, specify the `Vdata` field, as in `hinfo.Vdata`. `hdfread` can get the name of the HDF file from these structures.

`data = hdfread(...,param1,value1,param2,value2,...)` returns subsets of the data according to the specified parameter and value pairs. See the tables below to find the valid parameters and values for different types of data sets.

`[data,map] = hdfread(...)` returns the image `data` and the colormap `map` for an 8-bit raster image.

Subsetting Parameters

The following tables show the subsetting parameters that can be used with the `hdfread` function for certain types of HDF4 data. These data types are

- HDF Scientific Data (SD)
- HDF Vdata (V)
- HDF-EOS Grid Data
- HDF-EOS Point Data
- HDF-EOS Swath Data

Note the following:

- If a parameter requires multiple values, the values must be stored in a cell array. For example, the `'Index'` parameter requires three values: `start`, `stride`, and `edge`. Enclose these values in curly braces as a cell array.

  ```
  hdfread(dataset_name, 'Index', {start,stride,edge})
  ```

- All values that are indices are 1-based.

Subsetting Parameters for HDF Scientific Data (SD) Data Sets

When you are working with HDF SD files, `hdfread` supports the parameters listed in this table.

Parameter	Description
'Index'	Three-element cell array, {start,stride,edge}, specifying the location, range, and values to be read from the data set • start — A 1-based array specifying the position in the file to begin reading Default: 1, start at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set. • stride — A 1-based array specifying the interval between the values to read Default: 1, read every element of the data set. • edge — A 1-based array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions

For example, this code reads the data set Example SDS from the HDF file example.hdf. The 'Index' parameter specifies that hdfread start reading data at the beginning of each dimension, read until the end of each dimension, but only read every other data value in the first dimension.

```
hdfread('example.hdf','Example SDS', ...
                    'Index', {[], [2 1], []})
```

Subsetting Parameters for HDF Vdata Sets

When you are working with HDF Vdata files, hdfread supports these parameters.

Parameter	Description
'Fields'	Text string specifying the name of the data set field to be read from. When specifying multiple field names, use a comma-separated list.
'FirstRecord'	1-based number specifying the record from which to begin reading
'NumRecords'	Number specifying the total number of records to read

For example, this code reads the Vdata set `Example Vdata` from the HDF file `example.hdf`.

```
hdfread('example.hdf', 'Example Vdata', 'FirstRecord', 400,...
        'NumRecords', 50)
```

Subsetting Parameters for HDF-EOS Grid Data

When you are working with HDF-EOS grid data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive parameters — You can only specify one of these parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
'Fields'	String naming the data set field to be read. You can specify only one field name for a Grid data set.
Mutually Exclusive Optional Parameters	

Parameter	Description
'Index'	Three-element cell array, {start,stride,edge}, specifying the location, range, and values to be read from the data set start — An array specifying the position in the file to begin reading Default: 1, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. stride — An array specifying the interval between the values to read Default: 1, read every element of the data set. edge — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
'Interpolate'	Two-element cell array, {longitude,latitude}, specifying the longitude and latitude points that define a region for bilinear interpolation. Each element is an N-length vector specifying longitude and latitude coordinates.
'Pixels'	Two-element cell array, {longitude,latitude}, specifying the longitude and latitude coordinates that define a region. Each element is an N-length vector specifying longitude and latitude coordinates. This region is converted into pixel rows and columns with the origin in the upper left corner of the grid. Note: This is the pixel equivalent of reading a 'Box' region.
'Tile'	Vector specifying the coordinates of the tile to read, for HDF-EOS Grid files that support tiles
Optional Parameters	
'Box'	Two-element cell array, {longitude,latitude}, specifying the longitude and latitude coordinates that define a region. longitude and latitude are each two-element vectors specifying longitude and latitude coordinates.

Parameter	Description
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and end-point for a period of time
`'Vertical'`	Two-element cell array, `{dimension, range}` `dimension` — String specifying the name of the data set field to be read from. You can specify only one field name for a Grid data set. `range` — Two-element array specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
hdfread(grid_dataset, 'Fields', fieldname, ...
                'Vertical', {dimension, [min, max]})
```

Subsetting Parameters for HDF-EOS Point Data

When you are working with HDF-EOS Point data, `hdfread` has two required parameters and three optional parameters.

Parameter	Description
Required Parameters	
`'Fields'`	String naming the data set field to be read. For multiple field names, use a comma-separated list.
`'Level'`	1-based number specifying which level to read from in an HDF-EOS Point data set
Optional Parameters	

Parameter	Description
'Box'	Two-element cell array, {longitude,latitude}, specifying the longitude and latitude coordinates that define a region. longitude and latitude are each two-element vectors specifying longitude and latitude coordinates.
'RecordNumbers'	Vector specifying the record numbers to read
'Time'	Two-element cell array, [start stop], where start and stop are numbers that specify the start and endpoint for a period of time

For example,

```
hdfread(point_dataset, 'Fields', {field1, field2}, ...
         'Level', level, 'RecordNumbers', [1:50, 200:250])
```

Subsetting Parameters for HDF-EOS Swath Data

When you are working with HDF-EOS Swath data, hdfread supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive

You can only use one of the mutually exclusive parameters in a call to hdfread, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
'Fields'	String naming the data set field to be read. You can specify only one field name for a Swath data set.
Mutually Exclusive Optional Parameters	

Parameter	Description
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. • `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Time'`	Three-element cell array, `{start, stop, mode}`, where `start` and `stop` specify the beginning and the endpoint for a period of time, and `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`). • Any point is within the box (`mode='anypoint'`).
Optional Parameters	

Parameter	Description
`'Box'`	Three-element cell array, `{longitude, latitude, mode}` specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are two-element vectors that specify longitude and latitude coordinates. `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`). • Any point is within the box (`mode='anypoint'`).
`'ExtMode'`	String specifying whether geolocation fields and data fields must be in the same swath (`mode='internal'`), or can be in different swaths (`mode='external'`) Note: `mode` is only used when extracting a time period or a region.
`'Vertical'`	Two-element cell array, `{dimension, range}` • `dimension` is a string specifying either a dimension name or field name to subset the data by. • `range` is a two-element vector specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
hdfread('example.hdf', swath_dataset, 'Fields', fieldname, ...
```

```
'Time', {start, stop, 'midpoint'})
```

Examples

Example 1

Specify the name of the HDF file and the name of the data set. This example reads a data set named `'Example SDS'` from a sample HDF file.

```
data = hdfread('example.hdf', 'Example SDS')
```

Example 2

Use data returned by `hdfinfo` to specify the data set to read.

1. Call `hdfinfo` to retrieve information about the contents of the HDF file.

```
fileinfo = hdfinfo('example.hdf')
fileinfo =

    Filename: 'N:\toolbox\matlab\demos\example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

2. Extract the structure containing information about the particular data set you want to import from the data returned by `hdfinfo`. The example uses the structure in the `SDS` field to retrieve a scientific data set.

```
sds_info = fileinfo.SDS
sds_info =

       Filename: 'N:\toolbox\matlab\demos\example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
           Dims: [2x1 struct]
          Label: {}
```

```
        Description: {}
              Index: 0
```

3 You can pass this structure to `hdfread` to import the data in the data set.

```
data = hdfread(sds_info)
```

Example 3

You can use the information returned by `hdfinfo` to check the size of the data set.

```
sds_info.Dims.Size
ans =
    16
ans =
     5
```

Using the `'index'` parameter with `hdfread`, you can read a subset of the data in the data set. This example specifies a starting index of `[3 3]`, an interval of 1 between values (`[]` meaning the default value of 1), and a length of 10 rows and 2 columns.

```
data = hdfread(sds_info, 'Index', {[3 3],[],[10 2]});

data(:,1)
ans =
     7
     8
     9
    10
    11
    12
    13
    14
    15
    16
```

```
data(:,2)
ans =
     8
     9
    10
    11
    12
    13
    14
    15
    16
    17
```

Example 4

This example uses the `Vdata` field from the information returned by `hdfinfo` to read two fields of the data, `Idx` and `Temp`.

```
info = hdfinfo('example.hdf');

data = hdfread(info.Vdata,...
   'Fields',{'Idx','Temp'})

data =
    [1x10 int16]
    [1x10 int16]

index = data{1,1};
temp = data{2,1};

temp(1:6)
ans =
     0    12     3     5    10    -1
```

See Also

`hdfinfo`, `hdf`

hdftool

Purpose

Browse and import data from HDF4 or HDF-EOS files

Syntax

```
hdftool
hdftool(filename)
h = hdftool(...)
```

Description

`hdftool` starts the HDF Import Tool, a graphical user interface used to browse the contents of HDF4 and HDF-EOS files and import data and subsets of data from these files. To open an HDF4 or HDF-EOS file, select **Open** from the **File** menu. You can open multiple files in the HDF Import Tool by selecting **Open** from the **File** menu.

`hdftool(filename)` opens the HDF4 or HDF-EOS file specified by `filename` in the HDF Import Tool.

`h = hdftool(...)` returns a handle `h` to the HDF Import Tool. To close the tool from the command line, use `close(h)`.

Example

```
hdftool('example.hdf');
```

See Also

`hdf`, `hdfinfo`, `hdfread`, `uiimport`

Purpose

Help for MATLAB functions in Command Window

GUI Alternatives

Use the Help browser **Contents** for a product to view **Functions — Alphabetical List** or **Functions — By Category**, or run `doc functionname` to view more extensive help for a function in the Help browser.

Syntax

```
help
help /
help functionname
help modelname.mdl
help toolboxname
help toolboxname/functionname
help classname.methodname
help classname
help syntax
t = help('topic')
```

Description

`help` lists all primary help topics in the Command Window. Each main help topic corresponds to a directory name on the MATLAB search path.

`help /` lists all operators and special characters, along with their descriptions.

`help functionname` displays M-file help, which is a brief description and the syntax for `functionname`, in the Command Window. The output includes a link to `doc functionname`, which displays the reference page in the Help browser, often providing additional information. Output also includes see also links, which display help in the Command Window for related functions. If `functionname` is overloaded, that is, appears in multiple directories on the search path, `help` displays the M-file help for the first `functionname` found on the search path, and displays a hyperlinked list of the overloaded functions and their directories. If `functionname` is also the name of a toolbox, `help` also displays a list of subdirectories and hyperlinked list of functions in the toolbox, as defined in the `Contents.m` file for the toolbox.

`help modelname.mdl` displays the complete description for the MDL-file `modelname` as defined in **Model Properties > Description**. If Simulink is installed, you do not need to specify the `.mdl` extension.

`help toolboxname` displays the `Contents.m` file for the specified directory named `toolboxname`, where `Contents.m` contains a list and corresponding description of M-files in `toolboxname` — see the Remarks topic, "Creating Contents Files for Your Own M-File Directories" on page 2-1530. It is not necessary to give the full pathname of the directory; the last component, or the last several components, are sufficient. If `toolboxname` is also a function name, help also displays the M-file help for the function `toolboxname`.

`help toolboxname/functionname` displays the M-file help for the `functionname` that resides in the `toolboxname` directory. Use this form to get direct help for an overloaded function.

`help classname.methodname` displays help for the method `methodname` of the fully qualified class `classname`. If you do not know the fully qualified class for the method, use `class(obj)`, where `methodname` is of the same class as the object `obj`.

`help classname` displays help for the fully qualified class `classname`.

`help` **`syntax`** displays M-file help describing the syntax used in MATLAB commands and functions.

`t = help('topic')` returns the help text for `topic` as a string, with each line separated by `/n`, where `topic` is any allowable argument for `help`.

Note M-file help displayed in the Command Window uses all uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, use lowercase characters. Some functions for interfacing to Java do use mixed case; the M-file help accurately reflects that and you should use mixed case when typing them. For example, the `javaObject` function uses mixed case.

Remarks

To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`, and then enter the `help` statement.

Creating Online Help for Your Own M-Files

The MATLAB help system, like MATLAB itself, is highly extensible. You can write help descriptions for your own M-files and toolboxes using the same self-documenting method that MATLAB M-files and toolboxes use.

The `help` function lists all help topics by displaying the first line (the H1 line) of the contents files in each directory on the MATLAB search path. The contents files are the M-files named `Contents.m` within each directory.

Typing `helptopic` , where `topic` is a directory name, displays the comment lines in the `Contents.m` file located in that directory. If a contents file does not exist, `help` displays the H1 lines of all the files in the directory.

Typing `help topic`, where `topic` is a function name, displays help for the function by listing the first contiguous comment lines in the M-file `topic.m`.

Create self-documenting online help for your own M-files by entering text on one or more contiguous comment lines, beginning with the second line of the file (first line if it is a script). For example, the function `soundspeed.m` begins with

```
function c=soundspeed(s,t,p)
% soundspeed computes the speed of sound in water
% where c is the speed of sound in water in m/s

t = 0:.1:35;
```

When you execute `help soundspeed`, MATLAB displays

```
soundspeed computes the speed of sound in water
where c is the speed of sound in water in m/s
```

These lines are the first block of contiguous comment lines. After the first contiguous comment lines, enter an executable statement or blank line, which effectively ends the help section. Any later comments in the M-file do not appear when you type `help` for the function.

The first comment line in any M-file (the H1 line) is special. It should contain the function name and a brief description of the function. The `lookfor` function searches and displays this line, and `help` displays these lines in directories that do not contain a `Contents.m` file. For the `soundspeed` example, the H1 line is

```
% soundspeed computes speed of sound in water
```

Use the "Help Report" to help you create and manage M-file help for your own files.

Creating Contents Files for Your Own M-File Directories

A `Contents.m` file is provided for each M-file directory included with the MATLAB software. If you create directories in which to store your own M-files, it is a good practice to create `Contents.m` files for them, too. Use the "Contents Report" to help you create and maintain your own `Contents.m` files.

Examples

`help close` displays help for the `close` function.

`help database/close` displays help for the `close` function in Database Toolbox.

`help datafeed` displays help for Datafeed Toolbox.

`help database` lists the functions in Database Toolbox and displays help for the `database` function, because there are a function and a toolbox called `database`.

`help general` lists all functions in the directory *`matlabroot`*`/toolbox/matlab/general`. This illustrates how to specify a relative partial pathname rather than a full pathname.

`help f14_dap` displays the description of the Simulink `f14_dap.mdl` model file (Simulink must be installed.).

`t = help('close')` gets help for the function `close` and stores it as a string in `t`.

See Also

`class`, `doc`, `docsearch`, `helpbrowser`, `helpwin`, `lookfor`, `more`, `partialpath`, `path`, `what`, `which`, `whos`

helpbrowser

Purpose

Open Help browser to access all online documentation and demos

GUI Alternatives

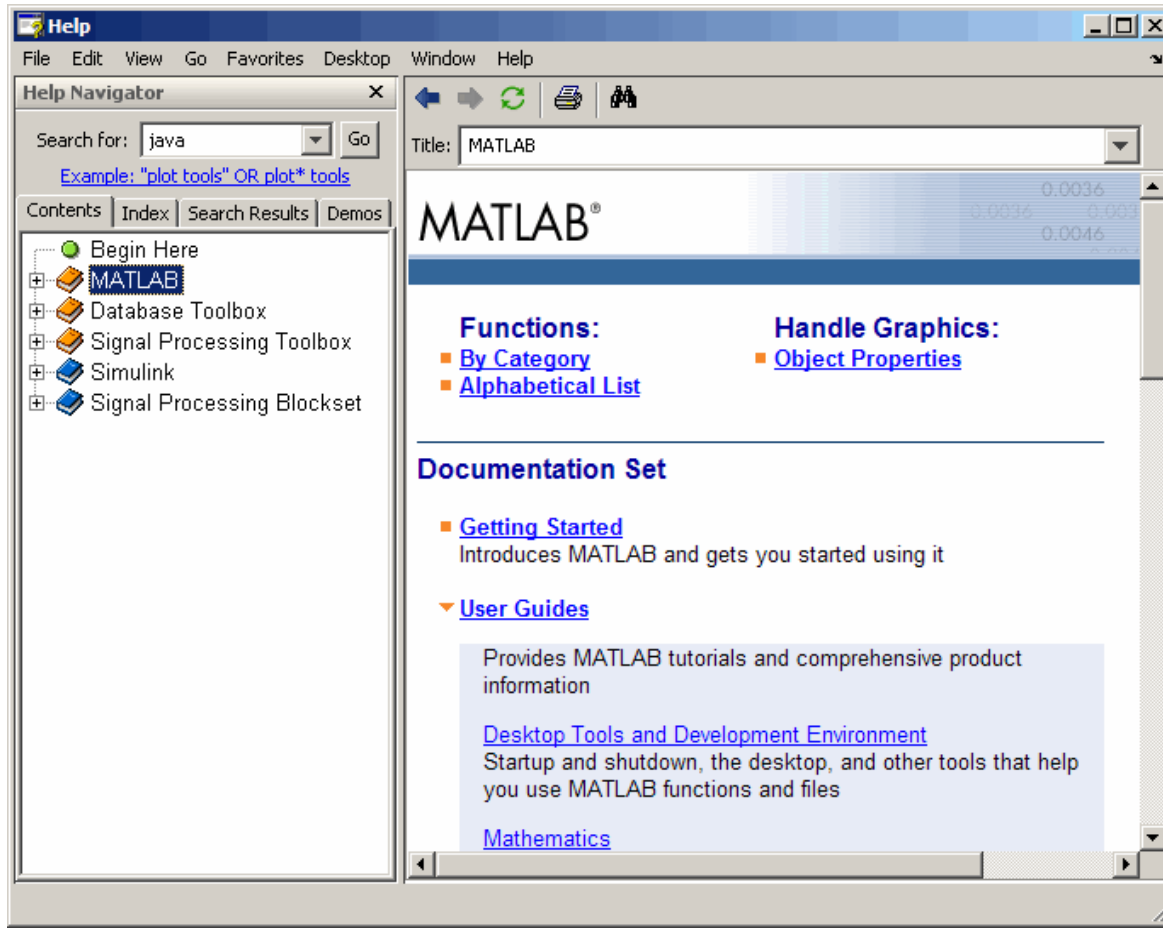
As an alternative to the `helpbrowser` function, select **Desktop > Help** or click the Help button  on the toolbar in the MATLAB desktop.

Syntax

```
helpbrowser
```

Description

`helpbrowser` displays the Help browser, providing direct access to a comprehensive library of online documentation, including reference pages and user guides. If the Help browser was previously opened in the current session, `helpbrowser` shows the last page viewed; otherwise it shows the **Begin Here** page. For details, see the "Help Browser Overview" topic in the MATLAB Desktop Tools and Development Environment documentation.



See Also

`builddocsearchdb`, `doc`, `docopt`, `docsearch`, `help`, `helpdesk`, `helpwin`, `lookfor`, `web`

helpdesk

Purpose

Open Help browser

Syntax

```
helpdesk
```

Description

`helpdesk` displays the Help browser and shows the “Begin Here” page. In previous releases, `helpdesk` displayed the Help Desk, which was the precursor to the Help browser. In a future release, the `helpdesk` function will be phased out — use the `doc` or `helpbrowser` function instead.

See Also

`doc`, `helpbrowser`

Purpose

Create and open help dialog box

Syntax

```
helpdlg
helpdlg('helpstring')
helpdlg('helpstring','dlgname')
h = helpdlg(...)
```

Description

`helpdlg` creates a nonmodal help dialog box or brings the named help dialog box to the front.

Note A nonmodal dialog box enables the user to interact with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`helpdlg` displays a dialog box named `'Help Dialog'` containing the string `'This is the default help string.'`

`helpdlg('helpstring')` displays a dialog box named `'Help Dialog'` containing the string specified by `'helpstring'`.

`helpdlg('helpstring','dlgname')` displays a dialog box named `'dlgname'` containing the string `'helpstring'`.

`h = helpdlg(...)` returns the handle of the dialog box.

Remarks

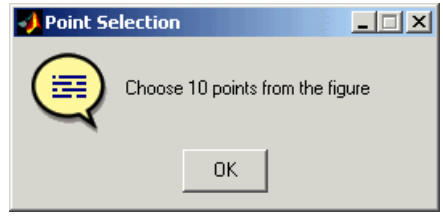
MATLAB wraps the text in `'helpstring'` to fit the width of the dialog box. The dialog box remains on your screen until you press the **OK** button or the **Enter** key. After either of these actions, the help dialog box disappears.

Examples

The statement

```
helpdlg('Choose 10 points from the figure','Point Selection');
```

displays this dialog box:



See Also

`dialog`, `errordlg`, `inputdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-104 for related functions

Purpose

Provide access to M-file help for all functions

Syntax

```
helpwin
helpwin topic
```

Description

`helpwin` lists topics for groups of functions in the Help browser. It shows brief descriptions of the topics and provides links to display M-file help for the functions in the Help browser. You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

`helpwin topic` displays help information for the topic in the Help browser. If `topic` is a directory, it displays all functions in the directory. If `topic` is a function, `helpwin` displays M-file help for that function in the Help browser. From the page, you can access a list of directories (**Default Topics** link) as well as the reference page help for the function (**Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

Examples

Typing

```
helpwin datafun
```

displays the functions in the `datafun` directory and a brief description of each.

Typing

```
helpwin fft
```

displays the M-file help for the `fft` function in the Help browser.

See Also

`doc`, `docopt`, `help`, `helpbrowser`, `lookfor`, `web`

hess

Purpose

Hessenberg form of matrix

Syntax

```
H = hess(A)
[P,H] = hess(A)
[AA,BB,Q,Z] = HESS(A,B)
```

Description

`H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.

`[P,H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P*H*P'` and `P'*P = eye(size(A))` .

`[AA,BB,Q,Z] = HESS(A,B)` for square matrices `A` and `B`, produces an upper Hessenberg matrix `AA`, an upper triangular matrix `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA` and `Q*B*Z = BB`.

Definition

A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H =
   -149    -50   -154
    537    180    546
    -27     -9    -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =
   -149.0000    42.2037   -156.3165
   -537.6783   152.5511   -554.9272
           0     0.0728      2.4489
```

Algorithm

Inputs of Type Double

For inputs of type double, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

Matrix A	Routine
Real symmetric	DSYTRD DSYTRD, DORGTR, (with output P)
Real nonsymmetric	DGEHRD DGEHRD, DORGHR (with output P)
Complex Hermitian	ZHETRD ZHETRD, ZUNGTR (with output P)
Complex non-Hermitian	ZGEHRD ZGEHRD, ZUNGHR (with output P)

Inputs of Type Single

For inputs of type `single`, `hess` uses the following LAPACK routines to compute the Hessenberg form of a matrix:

Matrix A	Routine
Real symmetric	SSYTRD SSYTRD, DORGTR, (with output P)
Real nonsymmetric	SGEHRD SGEHRD, SORGHR (with output P)
Complex Hermitian	CHETRD CHETRD, CUNGTR (with output P)
Complex non-Hermitian	CGEHRD CGEHRD, CUNGHR (with output P)

See Also

`eig`, `qz`, `schur`

References

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling,

A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Convert hexadecimal number string to decimal number

Syntax

```
d = hex2dec('hex_value')
```

Description

`d = hex2dec('hex_value')` converts *`hex_value`* to its floating-point integer representation. The argument *`hex_value`* is a hexadecimal integer stored in a MATLAB string. The value of *`hex_value`* must be smaller than hexadecimal `10,000,000,000,000`.

If *`hex_value`* is a character array, each row is interpreted as a hexadecimal string.

Examples

```
hex2dec('3ff')

ans =

        1023
```

For a character array `S`,

```
S =
0FF
2DE
123
hex2dec(S)

ans =

255
734
291
```

See Also

`dec2hex`, `format`, `hex2num`, `sprintf`

hex2num

Purpose

Convert hexadecimal number string to double-precision number

Syntax

```
n = hex2num(S)
```

Description

`n = hex2num(S)`, where `S` is a 16 character string representing a hexadecimal number, returns the IEEE double-precision floating-point number `n` that it represents. Fewer than 16 characters are padded on the right with zeros. If `S` is a character array, each row is interpreted as a double-precision number.

`NaN`s, infinities and denorms are handled correctly.

Example

```
hex2num('400921fb54442d18')
```

returns `Pi`.

```
hex2num('bff')
```

returns

```
ans =

    -1
```

See Also

`num2hex`, `hex2dec`, `sprintf`, `format`

Purpose

Export figure

GUI Alternative

Use the **File —> Saveas** on the figure window menu to access the **Export Setup** GUI. Use **Edit —> Copy Figure** to copy the figure's contents to your system's clipboard. For details, see How to Print or Export in the MATLAB Graphics documentation.

Syntax

```
hgexport(h,filename)
hgexport(h,'-clipboard')
```

Description

`hgexport(h,filename)` writes figure `h` to the file `filename`.

`hgexport(h,'-clipboard')` writes figure `h` to the Windows clipboard.

The format in which the figure is exported is determined by which renderer you use. The Painters renderer generates a metafile. The ZBuffer and OpenGL renderers generate a bitmap.

See Also

`print`

hggroup

Purpose

Create hggroup object

Syntax

Description

An hggroup object can be the parent of any axes children except light objects, as well as other hggroup objects. You can use hggroup objects to form a group of objects that can be treated as a single object with respect to the following cases:

- Visible — Setting the hggroup object's `Visible` property also sets each child object's `Visible` property to the same value.
- Selectable — Setting each hggroup child object's `HitTest` property to `off` enables you to select all children by clicking any child object.
- Current object — Setting each hggroup child object's `HitTest` property to `off` enables the hggroup object to become the current object when any child object is picked. See the next section for an example.

Examples

This example defines a callback for the `ButtonDownFcn` property of an hggroup object. In order for the hggroup to receive the mouse button down event that executes the `ButtonDownFcn` callback, the `HitTest` properties of all the line objects must be set to `off`. The event is then passed up the hierarchy to the hggroup.

The following function creates a random set of lines that are parented to an hggroup object. The subfunction `set_lines` defines a callback that executes when the mouse button is pressed over any of the lines. The callback simply increases the widths of all the lines by 1 with each button press.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

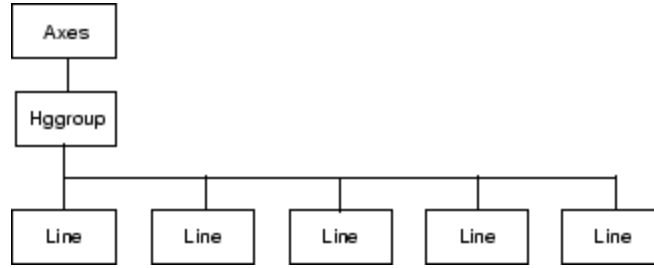
```
function doc_hggroup
```

```
hg = hggroup('ButtonDownFcn',@set_lines);
hl  = line(randn(5),randn(5),'HitTest','off','Parent',hg);

function set_lines(cb,eventdata)
hl = get(cb,'Children');% cb is handle of hggroup object
lw = get(hl,'LineWidth');% get current line widths
set(hl,{'LineWidth'},num2cell([lw{:}]+1,[5,1])')
```

Note that selecting any one of the lines selects all the lines. (To select an object, enable plot edit mode by selecting **Plot Edit** from the **Tools** menu.)

Instance Diagram for This Example

The following diagram shows the object hierarchy created by this example.



Hggroup Properties

Setting Default Properties

You can set default hggroup properties on the axes, figure, and root levels.

```
set(0,'DefaultHggroup<i>Property</i>',PropertyValue...)
set(gcf,'DefaultHggroup<i>Property</i>',PropertyValue...)
set(gca,'DefaultHggroup<i>Property</i>',PropertyValue...)
```

where *`Property`* is the name of the `hggroup` property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the hggroup properties.

See Also

`hgtransform`

“Group Objects” for more information and examples.

“Function Handle Callbacks” for information on how to use function handles to define callbacks.

`Hggroup Properties` for property descriptions

Purpose

Hggroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

To change the default values of properties, see "Setting Default Property Values".

See "Group Objects" for general information on this type of object.

Hggroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of hggroup objects in legends. The `Annotation` property enables you to specify whether this hggroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the hggroup object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the hggroup object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
`off`	Do not include the hggroup or its children in a legend (default)
`children`	Include only the children of the hggroup as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine whether objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the `delete` function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore can check the object’s `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the children of the hggroup object. Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure).

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

`Children`

array of graphics object handles

Children of the hggroup object. An array containing the handles of all objects parented to the hggroup object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not appear in the hggroup `Children` property unless you set the Root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips stairs plots to the axes plot box by default. If you set `Clipping` to `off`, lines might be displayed outside the axes plot box.

`CreateFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback executed during object creation. This property defines a callback function that executes when MATLAB creates an hggroup object. You must define this property as a default value for hggroup objects or in a call to the `hggroup` function to create a new hggroup object. For example, the statement

```
set(0,'DefaulthggroupCreateFcn',@myCreateFcn)
```

defines a default value on the root level that applies to every hggroup object created in that MATLAB session. Whenever you create an hggroup object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hggroup object's properties. Setting the `CreateFcn` property on an existing hggroup object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback executed during object deletion. A callback function that executes when the hggroup object is deleted (e.g., this might happen when you issue a `delete` command on the hggroup object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

String used by legend for this hggroup object. The `legend` function uses the string defined by the `DisplayName` property to label this hggroup object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this hggroup object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase hggroup child objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing

with `EraseMode` `none`, you cannot print these objects because MATLAB stores no information about their former locations.

- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

```
HandleVisibility
     {on} | callback | off
```

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hggroup object.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root ShowHiddenHandles property to on to make all handles visible regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties). See also findall.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HitTest

{on} | off

Pickable by mouse click. HitTest determines whether the hggroup object can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click on the hggroup child objects. Note that to pick the hggroup object, its children must have their HitTest property set to off.

If the hggroup object's HitTest is off, clicking it picks the object behind it.

Interruptible

{on} | off

Callback routine interruption mode. The Interruptible property controls whether an hggroup object callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from an hggroup property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`Parent`
axes handle

Parent of hggroup object. This property contains the handle of the hggroup object's parent object. The parent of an hggroup object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints of hggroup child objects if the `SelectionHighlight` property is also `on` (the default).

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the hggroup child objects. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics

programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks.

For example, you might create an hggroup object and set the `Tag` property:

```
t = hggroup('Tag','group1')
```

When you want to access the object, you can use `findobj` to find its handle. For example,

```
h = findobj('Tag','group1');
```

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For hggroup objects, `Type` is `'hggroup'`. The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`

handle of a uicontextmenu object

Associate a context menu with the hggroup object. Assign this property the handle of a uicontextmenu object created in the hggroup object's figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click the hggroup object.

`UserData`

array

User-specified data. This property can be any data you want to associate with the hggroup object (including cell arrays and structures). The hggroup object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`

`{on} | off`

Visibility of hggroup object and its children. By default, hggroup object visibility is `on`. This means all children of the hggroup are visible unless the child object's `Visible` property is set to `off`. Setting an hggroup object's `Visible` property to `off` also makes its children invisible.

Purpose

Load Handle Graphics object hierarchy from file

GUI Alternative

Use the **File —> Open** on the figure window menu to access figure files with the **Open** dialog.

Syntax

```
h = hgload('filename')
[h,old_prop_values] = hgload(...,property_structure)
hgload(...,'all')
```

Description

`h = hgload('filename')` loads Handle Graphics objects and its children if any from the FIG-file specified by `filename` and returns handles to the top-level objects. If `filename` contains no extension, then MATLAB adds the `.fig` extension.

`[h,old_prop_values] = hgload(...,property_structure)` overrides the properties on the top-level objects stored in the FIG-file with the values in `property_structure`, and returns their previous values in `old_prop_values`.

`property_structure` must be a structure having field names that correspond to property names and values that are the new property values.

`old_prop_values` is a cell array equal in length to `h`, containing the old values of the overridden properties for each object. Each cell contains a structure having field names that are property names, each of which contains the original value of each property that has been changed. Any property specified in `property_structure` that is not a property of a top-level object in the FIG-file is not included in `old_prop_values`.

`hgload(...,'all')` overrides the default behavior, which does not reload nonserializable objects saved in the file. These objects include the default toolbars and default menus.

Nonserializable objects (such as the default toolbars and the default menus) are normally not reloaded because they are loaded from different files at figure creation time. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files.

Passing the string `all` to `hgload` ensures that any nonserializable objects contained in the file are also reloaded.

Note that, by default, `hgsave` excludes nonserializable objects from the FIG-file unless you use the `all` flag.

See Also

`hgsave`, `open`

"Figure Windows" on page 1-95 for related functions

Purpose

Save Handle Graphics object hierarchy to file

GUI Alternative

Use the **File —> Saveas** on the figure window menu to access the Export Setup GUI. For details, see How to Print or Export in the MATLAB Graphics documentation.

Syntax

```
hgsave('filename')
hgsave(h,'filename')
hgsave(...,'all')
hgsave(...,'-v6')
```

Description

`hgsave('filename')` saves the current figure to a file named `filename`.

`hgsave(h,'filename')` saves the objects identified by the array of handles `h` to a file named `filename`. If you do not specify an extension for `filename`, then MATLAB adds the extension `.fig`. If `h` is a vector, none of the handles in `h` may be ancestors or descendents of any other handles in `h`.

`hgsave(...,'all')` overrides the default behavior, which does not save nonserializable objects. Nonserializable objects include the default toolbars and default menus. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files and also reduces the size of FIG-files. Passing the string `all` to `hgsave` ensures that nonserializable objects are also saved.

Note: the default behavior of `hgload` is to ignore nonserializable objects in the file at load time. This behavior can be overwritten using the `all` argument with `hgload`.

`hgsave(...,'-v6')` saves the FIG-file in a format that can be loaded by versions prior to MATLAB 7.

Full Backward Compatibility

When creating a figure you want to save and use in a MATLAB version prior to MATLAB 7, use the `'v6'` option with the plotting function and the `'-v6'` option for `hgsave`. Check the reference page for the plotting function you are using for more information.

See “Plot Objects and Backward Compatibility” for more information.

See Also

`hgload`, `open`, `save`

“Figure Windows” on page 1-95 for related functions

Purpose

Create hgtransform graphics object

Syntax

```
h = hgtransform
h = hgtransform('PropertyName',propertyvalue,...)
```

Description

`h = hgtransform` creates an hgtransform object and returns its handle.

`h = hgtransform('PropertyName',propertyvalue,...)` creates an hgtransform object with the property value settings specified in the argument list.

Hgtransform objects can contain other objects and thereby enable you to treat the hgtransform and its children as a single entity with respect to visibility, size, orientation, etc. You can group objects together by parenting them to a single hgtransform object (i.e., setting the object's `Parent` property to the hgtransform object's handle). For example,

```
h = hgtransform;
surface('Parent',h,...)
```

The primary advantage of parenting objects to an hgtransform object is that it provides the ability to perform *transforms* (e.g., translation, scaling, rotation, etc.) on the child objects in unison.

The parent of an hgtransform object is either an axes object or another hgtransform.

Although you cannot see an hgtransform object, setting its `Visible` property to `off` makes all its children invisible as well.

Exceptions and Limitations

- An hgtransform object can be the parent of any number axes children objects belonging to the same axes, with the exception of light objects.
- hgtransform objects can never be the parent of axes objects and therefore can contain objects only from a single axes.
- hgtransform objects can be the parent of other hgtransform objects within the same axes.

- You cannot transform image objects because images are not true 3-D objects. Texture mapping the image data to a surface `CData` enables you to produce the effect of transforming an image in 3-D space.

Note Many plotting functions clear the axes (i.e., remove axes children) before drawing the graph. Clearing the axes also deletes any hgtransform objects in the axes.

More Information

- The references in the "See Also" on page 2-1568 section for information on types of transforms
- The "Examples" on page 2-1564 section provides examples that illustrate the use of transforms.

Examples

Transforming a Group of Objects

This example shows how to create a 3-D star with a group of surface objects parented to a single hgtransform object. The hgtransform object is then rotated about the *z*-axis while its size is scaled.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

---

1 Create an axes and adjust the view. Set the axes limits to prevent auto limit selection during scaling.

```
ax = axes('XLim',[-1.5 1.5],'YLim',[-1.5 1.5],...
 'ZLim',[-1.5 1.5]);
view(3); grid on; axis equal
```

2 Create the objects you want to parent to the hgtransform object.

```
[x y z] = cylinder([.2 0]);
```

```
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

3 Create an hgtransform object and parent the surface objects to it.

```
t = hgtransform('Parent',ax);
set(h,'Parent',t)
```

4 Select a renderer and show the objects.

```
set(gcf,'Renderer','opengl')
drawnow
```

5 Initialize the rotation and scaling matrix to the identity matrix (`eye`).

```
Rz = eye(4);
Sxy = Rz;
```

6 Form the *z*-axis rotation matrix and the scaling matrix. Rotate 360 degrees (2*pi radians) and scale by using the increasing values of r.

```
for r = 1:.1:2*pi
 % Z-axis rotation matrix
 Rz = makehgtform('zrotate',r);
 % Scaling matrix
 Sxy = makehgtform('scale',r/4);
 % Concatenate the transforms and
 % set the hgtransform Matrix property
    set(t,'Matrix',Rz*Sxy)
    drawnow
end
pause(1)
```

7 Reset to the original orientation and size using the identity matrix.

```
set(t,'Matrix',eye(4))
```

Transforming Objects Independently

This example creates two hgtransform objects to illustrate how each can be transformed independently within the same axes. One of the hgtransform objects has been moved (by translation) away from the origin.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

---

1. Create and set up the axes object that will be the parent of both hgtransform objects. Set the limits to accommodate the translated object.

```
ax = axes('XLim',[-2 1],'YLim',[-2 1],'ZLim',[-1 1]);
view(3); grid on; axis equal
```

2. Create the surface objects to group.

```
[x y z] = cylinder([.3 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

3. Create the hgtransform objects and parent them to the same axes.

```
t1 = hgtransform('Parent',ax);
t2 = hgtransform('Parent',ax);
```

4. Set the renderer to use OpenGL.

```
set(gcf,'Renderer','opengl')
```

5 Parent the surfaces to hgtransform `t1`, then copy the surface objects and parent the copies to hgtransform `t2`.

```
set(h,'Parent',t1)
h2 = copyobj(h,t2);
```

6 Translate the second hgtransform object away from the first hgtransform object and display the result.

```
Txy = makehgtform('translate',[-1.5 -1.5 0]);
set(t2,'Matrix',Txy)
drawnow
```

7 Rotate both hgtransform objects in opposite directions. Hgtransform `t2` has already been translated away from the origin, so to rotate it about its *z*-axis you must first translate it to its original position. You can do this with the identity matrix (`eye`).

```
% rotate 10 times (2pi radians = 1 rotation)
for r = 1:.1:20*pi
 % Form z-axis rotation matrix
 Rz = makehgtform('zrotate',r);
 % Set transforms for both hgtransform objects
 set(t1,'Matrix',Rz)
 set(t2,'Matrix',Txy*inv(Rz)*I)
 drawnow
end
```

Setting Default Properties

You can set default hgtransform properties on the axes, figure, and root levels:

```
set(0,'DefaultHgtransform*PropertyName*',propertyvalue,...)
set(gcf,'DefaultHgtransform*PropertyName*',propertyvalue,...)
set(gca,'DefaultHgtransform*PropertyName*',propertyvalue,...)
```

where *`PropertyName`* is the name of the hgtransform property and `propertyvalue` is the value you are specifying. Use `set` and `get` to access hgtransform properties.

See Also

`hggroup`, `makehgtform`

For more information about transforms, see Tomas Moller and Eric Haines, *Real-Time Rendering*, A K Peters, Ltd., 1999.

"Group Objects" for more information and examples.

`Hgtransform Properties` for property descriptions

Purpose

Hgtransform properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

To change the default values of properties, see “Setting Default Property Values”.

See “Group Objects” for general information on this type of object.

Hgtransform Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of hgtransform objects in legends. The `Annotation` property enables you to specify whether this hgtransform object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the hgtransform object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the hgtransform object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
`off`	Do not include the hgtransform or its children in a legend (default)
`children`	Include only the children of the hgtransform as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine whether objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback functions. If there is a callback executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is within the extent of the hgtransform object, but not over another graphics object. The extent of an hgtransform object is the smallest rectangle that encloses all the children. Note that you cannot execute the hgtransform object's button down function if it has no children.

Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure).

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of the hgtransform object. An array containing the handles of all graphics objects parented to the hgtransform object (whether visible or not).

The graphics objects that can be children of an hgtransform are images, lights, lines, patches, rectangles, surfaces, and text. You can change the order of the handles and thereby change the stacking of the objects on the display.

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in the hgtransform `Children` property unless you set the Root `ShowHiddenHandles` property to `on`.

`Clipping`
`{on} | off`

This property has no effect on hgtransform objects.

`CreateFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback executed during object creation. This property defines a callback function that executes when MATLAB creates an hgtransform object. You must define this property as a default value for hgtransform objects. For example, the statement

```
set(0,'DefaultHgtransformCreateFcn',@myCreateFcn)
```

defines a default value on the root level that applies to every hgtransform object created in a MATLAB session. Whenever you

create an hgtransform object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hgtransform object's properties. Setting the `CreateFcn` property on an existing hgtransform object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DeleteFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback executed during object deletion. A callback function that executes when the hgtransform object is deleted (e.g., this might happen when you issue a `delete` command on the hgtransform object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

String used by legend for this hgtransform object. The `legend` function uses the string defined by the `DisplayName` property to label this hgtransform object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this hgtransform object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase hgtransform child objects (light objects have no erase mode). Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster,

but do not perform a complete redraw and are therefore less accurate.

- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor`— Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

  Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR operation on a pixel color and the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hgtransform object.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property,

figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

`HitTest`

```
{on} | off
```

Pickable by mouse click. `HitTest` determines whether the hgtransform object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click within the limits of the hgtransform object. If `HitTest` is `off`, clicking the hgtransform picks the object behind it.

`Interruptible`

```
{on} | off
```

Callback routine interruption mode. The `Interruptible` property controls whether an hgtransform object callback can be interrupted by callbacks invoked subsequently. Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`,

getframe, or pause command in the routine. See the BusyAction property for related information.

Setting Interruptible to on allows any graphics object's callback to interrupt callback routines originating from an hgtransform property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the gca or gcf command) when an interruption occurs.

Matrix
4-by-4 matrix

Transformation matrix applied to hgtransform object and its children. The hgtransform object applies the transformation matrix to all its children.

See "Group Objects" for more information and examples.

Parent
figure handle

Parent of hgtransform object. This property contains the handle of the hgtransform object's parent object. The parent of an hgtransform object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

Selected
on | {off}

Is object selected? When you set this property to on, MATLAB displays selection handles on all child objects of the hgtransform if the SelectionHighlight property is also on (the default).

SelectionHighlight
{on} | off

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the objects parented to the hgtransform. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks.

For example, you might create an hgtransform object and set the `Tag` property:

```
t = hgtransform('Tag','subgroup1')
```

When you want to access the hgtransform object to add another object, you can use `findobj` to find the hgtransform object's handle. The following statement adds a line to `subgroup1` (assuming `x` and `y` are defined).

```
line('XData',x,'YData',y,'Parent',findobj('Tag','subgroup1'))
```

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For hgtransform objects, `Type` is set to '`hgtransform`'. The following statement finds all the hgtransform objects in the current axes.

```
t = findobj(gca,'Type','hgtransform');
```

`UIContextMenu`

handle of a uicontextmenu object

Associate a context menu with the hgtransform object. Assign this property the handle of a uicontextmenu object created in the hgtransform object's figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the extent of the hgtransform object.

`UserData`
array

User-specified data. This property can be any data you want to associate with the hgtransform object (including cell arrays and structures). The hgtransform object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of hgtransform object and its children. By default, hgtransform object visibility is `on`. This means all children of the hgtransform are visible unless the child object's `Visible` property is set to `off`. Setting an hgtransform object's `Visible` property to `off` also makes its children invisible.

Purpose

Remove hidden lines from mesh plot

Syntax

```
hidden on
hidden off
hidden
```

Description

Hidden line removal draws only those lines that are not obscured by other objects in the field of view.

`hidden on` turns on hidden line removal for the current graph so lines in the back of a mesh are hidden by those in front. This is the default behavior.

`hidden off` turns off hidden line removal for the current graph.

`hidden` toggles the hidden line removal state.

Algorithm

`hidden on` sets the `FaceColor` property of a surface graphics object to the background `Color` of the axes (or of the figure if axes `Color` is `none`).

Examples

Set hidden line removal `off` and `on` while displaying the `peaks` function.

```
mesh(peaks)
hidden off
hidden on
```

See Also

`shading`, `mesh`

The surface properties `FaceColor` and `EdgeColor`

"Creating Surfaces and Meshes" on page 1-97 for related functions

hilb

Purpose

Hilbert matrix

Syntax

```
H = hilb(n)
```

Description

`H = hilb(n)` returns the Hilbert matrix of order `n`.

Definition

The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i,j) = 1/(i+j-1)$.

Examples

Even the fourth-order Hilbert matrix shows signs of poor conditioning.

```
cond(hilb(4)) =
    1.5514e+04
```

Note See the M-file for a good example of efficient MATLAB programming where conventional `for` loops are replaced by vectorized statements.

See Also

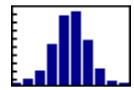
`invhilb`

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

Purpose

Histogram plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
n = hist(Y)
n = hist(Y,x)
n = hist(Y,nbins)
[n,xout] = hist(...)
hist(...)
hist(axes_handle,...)
```

Description

A histogram shows the distribution of data values.

`n = hist(Y)` bins the elements in vector `Y` into 10 equally spaced containers and returns the number of elements in each container as a row vector. If `Y` is an `m`-by-`p` matrix, `hist` treats the columns of `Y` as vectors and returns a 10-by-`p` matrix `n`. Each column of `n` contains the results for the corresponding column of `Y`. No elements of `Y` can be complex.

`n = hist(Y,x)` where `x` is a vector, returns the distribution of `Y` among `length(x)` bins with centers specified by `x`. For example, if `x` is a 5-element vector, `hist` distributes the elements of `Y` into five bins centered on the *x*-axis at the elements in `x`, none of which can be complex. Note: use `histc` if it is more natural to specify bin edges instead of centers.

`n = hist(Y,nbins)` where `nbins` is a scalar, uses `nbins` number of bins.

`[n,xout] = hist(...)` returns vectors `n` and `xout` containing the frequency counts and the bin locations. You can use `bar(xout,n)` to plot the histogram.

`hist(...)` without output arguments produces a histogram plot of the output described above. `hist` distributes the bins along the *x*-axis between the minimum and maximum values of `Y`.

`hist(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

Remarks

All elements in vector `Y` or in one column of matrix `Y` are grouped according to their numeric range. Each group is shown as one bin.

The histogram's *x*-axis reflects the range of values in `Y`. The histogram's *y*-axis shows the number of elements that fall within the groups; therefore, the *y*-axis ranges from `0` to the greatest number of elements deposited in any bin. The *x*-range of the leftmost and rightmost bins extends to include the entire data range in the case when the user-specified range does not cover the data range. If you want a plot in which this does not happen (that is, all bins have equal width), you can create a histogram-like display using the `bar` command.

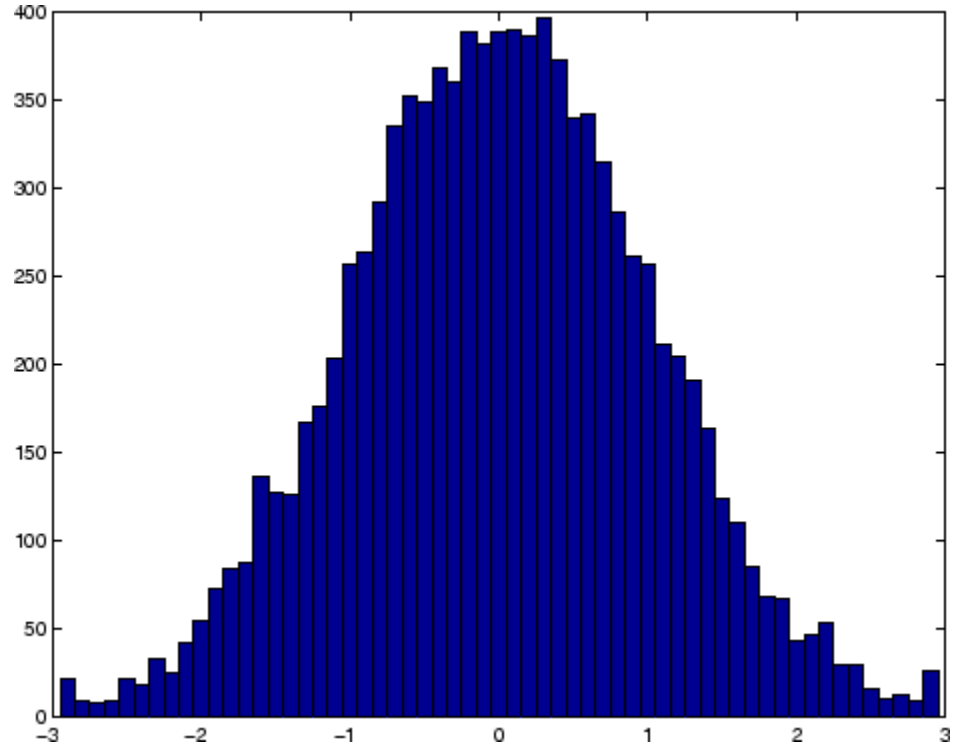
The `hist` function does not work with data that contain `inf` values.

The histogram is created with a patch graphics object. If you want to change the color of the graph, you can set patch properties. See the examples for more information. By default, the graph color is controlled by the current colormap, which maps the bin color to the first color in the colormap.

Examples

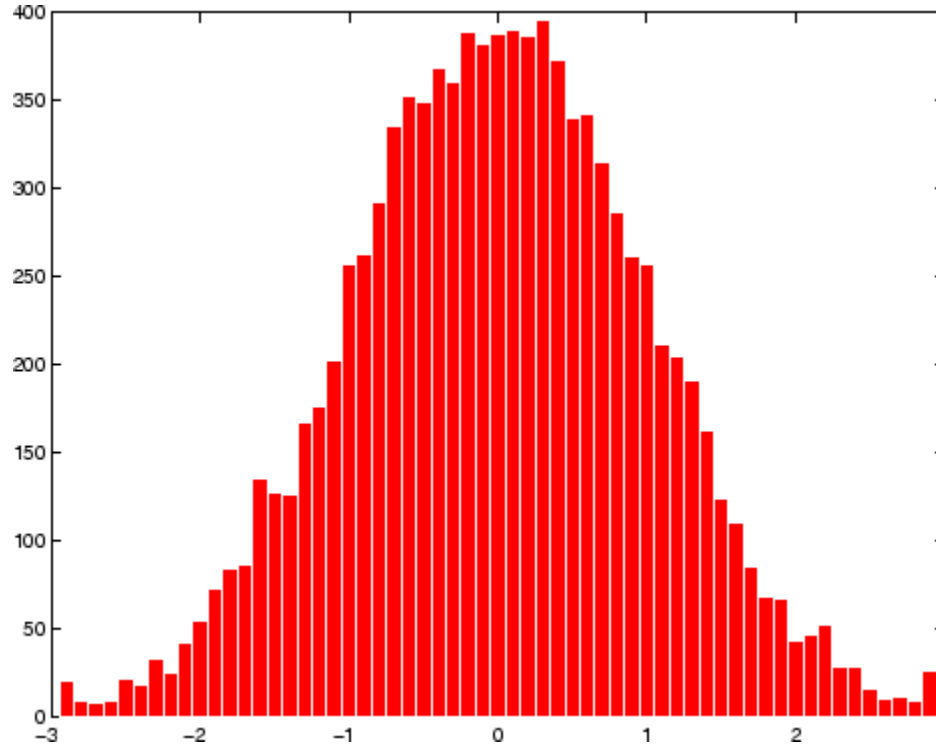
Generate a bell-curve histogram from Gaussian data.

```
x = -2.9:0.1:2.9;
y = randn(10000,1);
hist(y,x)
```



Change the color of the graph so that the bins are red and the edges of the bins are white.

```
h = findobj(gca,'Type','patch');
set(h,'FaceColor','r','EdgeColor','w')
```



See Also

`bar`, `ColorSpec`, `histc`, `mode`, `patch`, `rose`, `stairs`

"Specialized Plotting" on page 1-88 for related functions

"Histograms" for examples

Purpose

Histogram count

Syntax

```
n = histc(x,edges)
n = histc(x,edges,dim)
[n,bin] = histc(...)
```

Description

`n = histc(x,edges)` counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically nondecreasing values). `n` is a `length(edges)` vector containing these counts. No elements of `x` can be complex.

`n(k)` counts the value `x(i)` if `edges(k) <= x(i) < edges(k+1)`. The last bin counts any values of `x` that match `edges(end)`. Values outside the values in `edges` are not counted. Use `-inf` and `inf` in `edges` to include all non-`NaN` values.

For matrices, `histc(x,edges)` returns a matrix of column histogram counts. For N-D arrays, `histc(x,edges)` operates along the first nonsingleton dimension.

`n = histc(x,edges,dim)` operates along the dimension `dim`.

`[n,bin] = histc(...)` also returns an index matrix `bin`. If `x` is a vector, `n(k) = sum(bin==k)`. `bin` is zero for out of range values. If `x` is an `M`-by-`N` matrix, then

```
for j=1:N,
n(k,j) = sum(bin(:,j)==k);
end
```

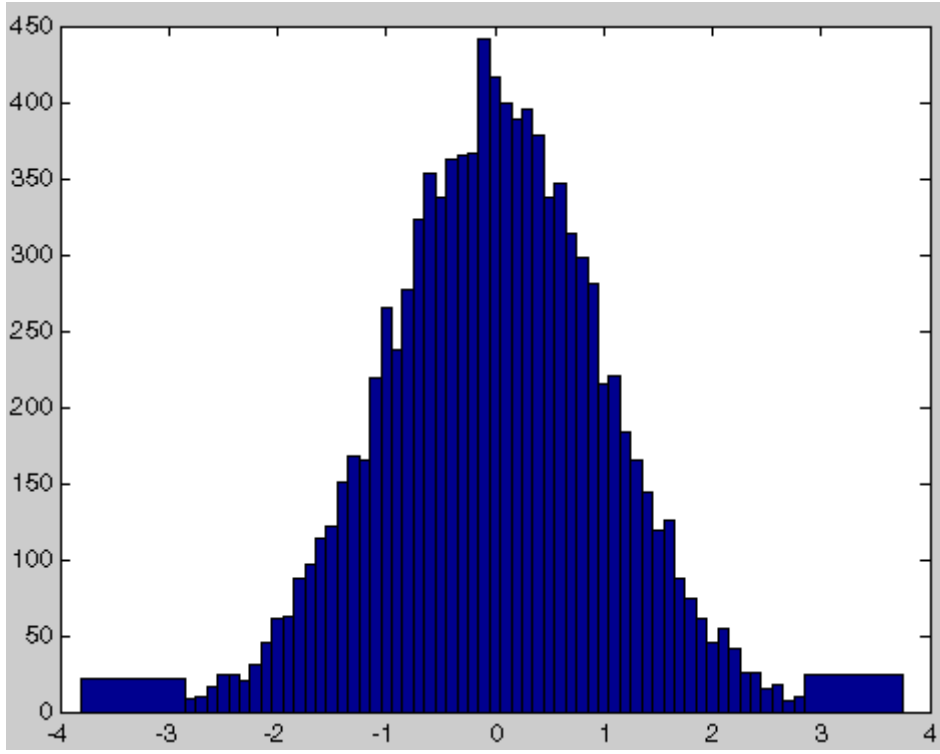
To plot the histogram, use the `bar` command.

Examples

Generate a cumulative histogram of a distribution.

Consider the following distribution:

```
x = -2.9:0.1:2.9;
y = randn(10000,1);
figure(1), hist(y,x)
```



Calculate number of elements in each bin

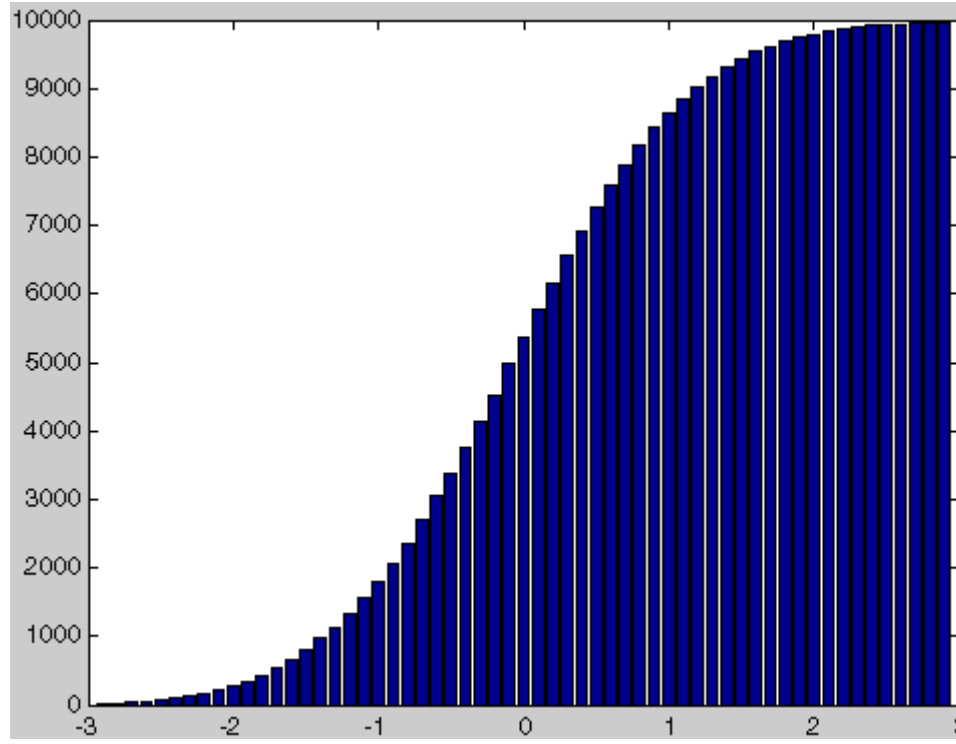
```
n_elements = histc(y,x);
```

Calculate the cumulative sum of these elements using `cumsum`

```
c_elements = cumsum(n_elements)
```

Plot the cumulative histogram

```
figure(2),bar(x,c_elements)
```



See Also

`hist`, `mode`

"Specialized Plotting" on page 1-88 for related functions

hold

Purpose

Retain current graph in figure

Syntax

```
hold on
hold off
hold all
hold
hold(axes_handle,...)
```

Description

The `hold` function determines whether new graphics objects are added to the graph or replace objects in the graph.

`hold on` retains the current plot and certain axes properties so that subsequent graphing commands add to the existing graph.

`hold off` resets axes properties to their defaults before drawing new plots. `hold off` is the default.

`hold all` holds the plot and the current line color and line style so that subsequent plotting commands do not reset the `ColorOrder` and `LineStyleOrder` property values to the beginning of the list. Plotting commands continue cycling through the predefined colors and linestyles from where the last plot stopped in the list.

`hold` toggles the hold state between adding to the graph and replacing the graph.

`hold(axes_handle,...)` applies the hold to the axes identified by the handle `axes_handle`.

Remarks

Test the hold state using the `ishold` function.

Although the hold state is `on`, some axes properties change to accommodate additional graphics objects. For example, the axes' limits increase when the data requires them to do so.

The `hold` function sets the `NextPlot` property of the current figure and the current axes. If several axes objects exist in a figure window, each axes has its own hold state. `hold` also creates an axes if one does not exist.

`hold on` sets the `NextPlot` property of the current figure and axes to `add`.

`hold off` sets the `NextPlot` property of the current axes to `replace`.

`hold` toggles the `NextPlot` property between the `add` and `replace` states.

See Also

`axis`, `cla`, `ishold`, `newplot`

The `NextPlot` property of axes and figure graphics objects

"Basic Plots and Graphs" on page 1-86 for related functions

home

Purpose

Move cursor to upper-left corner of Command Window

Syntax

```
home
```

Description

`home` moves the cursor to the upper-left corner of the Command Window. You can use the scroll bar to see the history of previous functions.

Examples

Use `home` in an M-file to return the cursor to the upper-left corner of the screen.

See Also

`clc`

Purpose

Concatenate arrays horizontally

Syntax

```
C = horzcat(A1, A2, ...)
```

Description

`C = horzcat(A1, A2, ...)` horizontally concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of rows.

`horzcat` concatenates N-dimensional arrays along the second dimension. The first and remaining dimensions must match.

MATLAB calls `C = horzcat(A1, A2,...)` for the syntax `C = [A1 A2 ...]` when any of `A1`, `A2`, etc., is an object.

Examples

Create a 3-by-5 matrix, `A`, and a 3-by-3 matrix, `B`. Then horizontally concatenate `A` and `B`.

```
A = magic(5);            % Create 3-by-5 matrix, A
A(4:5,:) = []

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22


B = magic(3)*100       % Create 3-by-3 matrix, B

B =

   800   100   600
   300   500   700
   400   900   200


C = horzcat(A, B)      % Horizontally concatenate A and B
```

```
C =

    17    24     1     8    15   800   100   600
    23     5     7    14    16   300   500   700
     4     6    13    20    22   400   900   200
```

See Also

vertcat, cat

Purpose

Horizontal concatenation for `tscollection` objects

Syntax

```
tsc = horzcat(tsc1,tsc2,...)
```

Description

`tsc = horzcat(tsc1,tsc2,...)` performs horizontal concatenation for `tscollection` objects:

```
tsc = [tsc1 tsc2 ...]
```

This operation combines multiple `tscollection` objects, which must have the same time vectors, into one `tscollection` containing `timeseries` objects from all concatenated collections.

See Also

`tscollection`, `vertcat (tscollection)`

hostid

Purpose

MATLAB server host identification number

Syntax

```
id = hostid
```

Description

`id = hostid` usually returns a single element cell array containing the identifier as a string. UNIX systems may have more than one identifier. In this case, `hostid` returns a cell array with an identifier in each cell.

Purpose

Convert HSV colormap to RGB colormap

Syntax

```
M = hsv2rgb(H)
rgb_image = hsv2rgb(hsv_image)
```

Description

`M = hsv2rgb(H)` converts a hue-saturation-value (HSV) colormap to a red-green-blue (RGB) colormap. `H` is an *m*-by-3 matrix, where *m* is the number of colors in the colormap. The columns of `H` represent hue, saturation, and value, respectively. `M` is an *m*-by-3 matrix. Its columns are intensities of red, green, and blue, respectively.

`rgb_image = hsv2rgb(hsv_image)` converts the HSV image to the equivalent RGB image. HSV is an *m*-by-*n*-by-3 image array whose three planes contain the hue, saturation, and value components for the image. RGB is returned as an *m*-by-*n*-by-3 image array whose three planes contain the red, green, and blue components for the image.

Remarks

As `H(:,1)` varies from 0 to 1, the resulting color varies from red through yellow, green, cyan, blue, and magenta, and returns to red. When `H(:,2)` is 0, the colors are unsaturated (i.e., shades of gray). When `H(:,2)` is 1, the colors are fully saturated (i.e., they contain no white component). As `H(:,3)` varies from 0 to 1, the brightness increases.

The MATLAB `hsv` colormap uses `hsv2rgb([huesaturationvalue])` where `hue` is a linear ramp from 0 to 1, and `saturation` and `value` are all 1's.

See Also

`brighten`, `colormap`, `rgb2hsv`

"Color Operations" on page 1-98 for related functions

hypot

Purpose

Square root of sum of squares

Syntax

```
c = hypot(a,b)
```

Description

`c = hypot(a,b)` returns the element-wise result of the following equation, computed to avoid underflow and overflow:

```
c = sqrt(abs(a).^2 + abs(b).^2)
```

Inputs `a` and `b` must follow these rules:

- Both `a` and `b` must be single- or double-precision, floating-point arrays.
- The sizes of the `a` and `b` arrays must either be equal, or one a scalar and the other nonscalar. In the latter case, `hypot` expands the scalar input to match the size of the nonscalar input.
- If `a` or `b` is an empty array (0-by-N or N-by-0), the other must be the same size or a scalar. The result `c` is an empty array having the same size as the empty input(s).

`hypot` returns the following in output `c`, depending upon the types of inputs:

- If the inputs to `hypot` are complex (`w+xi` and `y+zi`), then the statement `c = hypot(w+xi,y+zi)` returns the *positive real* result

  ```
  c = sqrt(abs(w).^2+abs(x).^2+abs(y).^2+abs(z).^2)
  ```

- If `a` or `b` is `–Inf`, `hypot` returns `Inf`.
- If neither `a` nor `b` is `Inf`, but one or both inputs is `NaN`, `hypot` returns `NaN`.
- If all inputs are finite, the result is finite. The one exception is when both inputs are very near the value of the MATLAB constant `realmax`. The reason for this is that the equation `c =`

`hypot(realmax,realmax)` is theoretically `sqrt(2)*realmax`, which overflows to `Inf`.

Examples

Example 1

To illustrate the difference between using the `hypot` function and coding the basic `hypot` equation in M-code, create an anonymous function that performs the same function as `hypot`, but without the consideration to underflow and overflow that `hypot` offers:

```
myhypot = @(a,b)sqrt(abs(a).^2+abs(b).^2);
```

Find the upper limit at which your coded function returns a useful value. You can see that this test function reaches its maximum at about `1e154`, returning an infinite result at that point:

```
myhypot(1e153,1e153)
ans =
    1.4142e+153

myhypot(1e154,1e154)
ans =
   Inf
```

Do the same using the `hypot` function, and observe that `hypot` operates on values up to about `1e308`, which is approximately equal to the value for `realmax` on your computer (the largest double-precision floating-point number you can represent on a particular computer):

```
hypot(1e308,1e308)
ans =
  1.4142e+308

hypot(1e309,1e309)
ans =
   Inf
```

Example 2

`hypot(a,a)` theoretically returns `sqrt(2)*abs(a)`, as shown in this example:

```
x = 1.271161e308;

y = x * sqrt(2)
y =
  1.7977e+308

y = hypot(x,x)
y =
  1.7977e+308
```

Algorithm

`hypot` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`sqrt`, `abs`, `norm`

Purpose

Imaginary unit

Syntax

```
i
a+bi
x+i*y
```

Description

As the basic imaginary unit `sqrt(-1)`, `i` is used to enter complex numbers. Since `i` is a function, it can be overridden and used as a variable. This permits you to use `i` as an index in `for` loops, etc.

If desired, use the character `i` without a multiplication sign as a suffix in forming a complex numerical constant.

You can also use the character `j` as the imaginary unit.

Examples

```
Z = 2+3i
Z = x+i*y
Z = r*exp(i*theta)
```

See Also

`conj`, `imag`, `j`, `real`

idealfilter (timeseries)

Purpose

Apply ideal (noncausal) filter to `timeseries` object

Syntax

```
ts2 = idealfilter(ts1,Interval,FilterType)
ts2 = idealfilter(ts1,Interval,FilterType,Index)
```

Description

`ts2 = idealfilter(ts1,Interval,FilterType)` applies an ideal filter of `FilterType` `'pass'` or `'notch'` to one or more frequency intervals specified by `Interval` for the `timeseries` object `ts1`. You specify several frequency intervals as an `n`-by-2 array of start and end frequencies, where `n` represents the number of intervals.

`ts2 = idealfilter(ts1,Interval,FilterType,Index)` applies an ideal filter and uses the optional `Index` integer array to specify the columns or rows to filter. When `ts.IsTimeFirst` is set to `true`, `Index` specifies one or more data columns. When `ts.IsTimeFirst` is set to `false`, Index specifies one or more data rows.

Remarks

When to Use the Ideal Filter

You use the ideal *notch* filter when you want to remove variations in a specific frequency range. Alternatively, you use the ideal *pass* filter to allow only the variations in a specific frequency range.

These filters are ideal in the sense that they are not realizable; an ideal filter is noncausal and the ends of the filter amplitude are perfectly flat in the frequency domain.

Requirement for Uniform Samples in Time

If the time-series data is sampled nonuniformly, filtering resamples this data on a uniform time vector.

Interpolation of NaN Values

All `NaN`s in the time series are interpolated before filtering using the interpolation method you assigned to the `timeseries` object.

Examples

You will apply an ideal notch filter to the data in `count.dat`.

1. Load the matrix `count` into the workspace.

```
load count.dat;
```

2 Create a `timeseries` object based on this matrix. The time vector ranges from 1 to 24 seconds in 1-second intervals.

```
count1=timeseries(count(:,1),1:24);
```

3 Enter the frequency interval in hertz.

```
interval=[0.08 0.2];
```

4 Call the filter function:

```
idealfilter_count = idealfilter(count1,interval,'notch')
```

5 Compare the original data and the shaped data with an overlaid plot of the two curves.

```
plot(count1,'-.'), grid on, hold on
plot(filter_count,'-')
legend('Original Data','Shaped Data',2)
```

See Also

`filter (timeseries)`, `timeseries`

Purpose

Integer division with rounding option

Syntax

```
C = idivide(A, B, opt)
C = idivide(A, B)
C = idivide(A, B, 'fix')
C = idivide(A, B, 'round')
C = idivide(A, B, 'floor')
C = idivide(A, B, 'ceil')
```

Description

`C = idivide(A, B, opt)` is the same as `A./B` for integer classes except that fractional quotients are rounded to integers using the optional rounding mode specified by `opt`. The default rounding mode is `'fix'`. Inputs `A` and `B` must be real and must have the same dimensions unless one is a scalar. At least one of the arguments `A` and `B` must belong to an integer class, and the other must belong to the same integer class or be a scalar double. The result `C` belongs to the integer class.

`C = idivide(A, B)` is the same as `A./B` except that fractional quotients are rounded toward zero to the nearest integers.

`C = idivide(A, B, 'fix')` is the same as the syntax shown immediately above.

`C = idivide(A, B, 'round')` is the same as `A./B` for integer classes. Fractional quotients are rounded to the nearest integers.

`C = idivide(A, B, 'floor')` is the same as `A./B` except that fractional quotients are rounded toward negative infinity to the nearest integers.

`C = idivide(A, B, 'ceil')` is the same as `A./B` except that the fractional quotients are rounded toward infinity to the nearest integers.

Examples

```
a = int32([-2 2]);
b = int32(3);

idivide(a,b)             % Returns [0 0]
idivide(a,b,'floor')     % Returns [-1 0]
idivide(a,b,'ceil')      % Returns [0 1]
```

```
idivide(a,b,'round')     % Returns [-1 1]
```

See Also

ldivide, rdivide, mldivide, mrdivide

Purpose

Execute statements if condition is true

Syntax

```
if expression, statements, end
```

Description

`if expression, statements, end` evaluates *`expression`* and, if the evaluation yields logical `1` (`true`) or a nonzero result, executes one or more MATLAB commands denoted here as *`statements`*.

`expression` is a MATLAB expression, usually consisting of variables or smaller expressions joined by relational operators (e.g., `count < limit`), or logical functions (e.g., `isreal(A)`). Simple expressions can be combined by logical operators (`&&`, `||`, `~`) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

```
(count < limit) && ((height - offset) >= 0)
```

Nested `if` statements must each be paired with a matching `end`.

The `if` function can be used alone or with the `else` and `elseif` functions. When using `elseif` and/or `else` within an `if` statement, the general form of the statement is

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

See “Program Control Statements” in the MATLAB Programming documentation for more information on controlling the flow of your program code.

Remarks

Nonscalar Expressions

If the evaluated `expression` yields a nonscalar value, then every element of this value must be true or nonzero for the entire expression to be considered true. For example, the statement `if (A < B)` is true only if each element of matrix `A` is less than its corresponding element in matrix `B`. See Example 2, below.

Partial Evaluation of the expression Argument

Within the context of an `if` or `while` expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is true or false through only partial evaluation.

For example, if `A` equals zero in statement 1 below, then the expression evaluates to `false`, regardless of the value of `B`. In this case, there is no need to evaluate `B` and MATLAB does not do so. In statement 2, if `A` is nonzero, then the expression is true, regardless of `B`. Again, MATLAB does not evaluate the latter part of the expression.

```
1)   if (A && B)              2)   if (A || B)
```

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

```
while (b ~= 0) && (a/b > 18.5)
if exist('myfun.m') && (myfun(x) >= y)
if iscell(A) && all(cellfun('isreal', A))
```

Empty Arrays

In most cases, using `if` on an empty array treats the array as `false`. There are some conditions however under which `if` evaluates as `true` on an empty array. Two examples of this, where `A` is equal to `[]`, are

```
if all(A), do_something, end
if 1|A, do_something, end
```

The latter expression is `true` because of short-circuiting, which causes MATLAB to ignore the right side operand of an OR statement whenever the left side evaluates to `true`.

Short-Circuiting Behavior

When used in the context of an `if` or `while` expression, and only in this context, the element-wise `|` and `&` operators use short-circuiting in evaluating their expressions. That is, `A|B` and `A&B` ignore the second operand, `B`, if the first operand, `A`, is sufficient to determine the result.

See "Short-Circuiting in Elementwise Operators" for more information on this.

Examples

Example 1 - Simple if Statement

In this example, if both of the conditions are satisfied, then the student passes the course.

```
if ((attendance >= 0.90) && (grade_average >= 60))
   pass = 1;
end;
```

Example 2 - Nonscalar Expression

Given matrices `A` and `B`,

```
A =                 B =
     1     0             1     1
     2     3             3     4
```

Expression	Evaluates As	Because
`A < B`	`false`	A(1,1) is not less than B(1,1).
`A < (B + 1)`	`true`	Every element of A is less than that same element of B with 1 added.

Expression	Evaluates As	Because
`A & B`	`false`	A(1,2) is false, and B is ignored due to short-circuiting.
`B < 5`	`true`	Every element of B is less than 5.

See Also

`else`, `elseif`, `end`, `for`, `while`, `switch`, `break`, `return`, `relational operators`, logical operators (`elementwise` and `short-circuit`),

Purpose

Inverse discrete Fourier transform

Syntax

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
y = ifft(..., 'symmetric')
y = ifft(..., 'nonsymmetric')
```

Description

`y = ifft(X)` returns the inverse discrete Fourier transform (DFT) of vector `X`, computed with a fast Fourier transform (FFT) algorithm. If `X` is a matrix, `ifft` returns the inverse DFT of each column of the matrix.

`ifft` tests `X` to see whether vectors in `X` along the active dimension are *conjugate symmetric*. If so, the computation is faster and the output is real. An `N`-element vector `x` is conjugate symmetric if `x(i) = conj(x(mod(N-i+1,N)+1))` for each element of `x`.

If `X` is a multidimensional array, `ifft` operates on the first non-singleton dimension.

`y = ifft(X,n)` returns the `n`-point inverse DFT of vector `X`.

`y = ifft(X,[],dim)` and `y = ifft(X,n,dim)` return the inverse DFT of `X` across the dimension `dim`.

`y = ifft(..., 'symmetric')` causes `ifft` to treat `X` as conjugate symmetric along the active dimension. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft(..., 'nonsymmetric')` is the same as calling `ifft(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft(fft(X))` equals `X` to within roundoff error.

Algorithm

The algorithm for `ifft(X)` is the same as the algorithm for `fft(X)`, except for a sign change and a scale factor of `n = length(X)`. As for `fft`, the execution time for `ifft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have

only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft` using the utility function `fftw`, which controls how MATLAB optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifft` supports inputs of data types `double` and `single`. If you call `ifft` with the syntax `y = ifft(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fft`, `fft2`, `ifft2`, `ifftn`, `ifftshift`, `fftw`, `ifft2`, `ifftn`

`dftmtx` and `freqz`, in the Signal Processing Toolbox

Purpose

2-D inverse discrete Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
y = ifft2(..., 'symmetric')
y = ifft2(..., 'nonsymmetric')
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifft2` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `M`-by-`N` matrix `X` is conjugate symmetric if `X(i,j) = conj(X(mod(M-i+1, M) + 1, mod(N-j+1, N) + 1))` for each element of `X`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

`y = ifft2(..., 'symmetric')` causes `ifft2` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft2(..., 'nonsymmetric')` is the same as calling `ifft2(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error.

Algorithm

The algorithm for `ifft2(X)` is the same as the algorithm for `fft2(X)`, except for a sign change and scale factors of `[m,n] = size(X)`. The execution time for `ifft2` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft2` using the utility function `fftw`, which controls how MATLAB optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifft2` supports inputs of data types `double` and `single`. If you call `ifft2` with the syntax `y = ifft2(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`dftmtx` and `freqz` in the Signal Processing Toolbox, and:

`fft2`, `fftw`, `fftshift`, `ifft`, `ifftn`, `ifftshift`

Purpose

N-D inverse discrete Fourier transform

Syntax

```
Y = ifftn(X)
Y = ifftn(X,siz)
y = ifftn(..., 'symmetric')
y = ifftn(..., 'nonsymmetric')
```

Description

`Y = ifftn(X)` returns the n-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifftn` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `N1-by-N2-by- ... Nk` array `X` is conjugate symmetric if

```
X(i1,i2, ...,ik) = conj(X(mod(N1-i1+1,N1)+1, mod(N2-i2+1,N2)+1,
... mod(Nk-ik+1,Nk)+1))
```

for each element of `X`.

`Y = ifftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the inverse transform. The size of the result `Y` is `siz`.

`y = ifftn(..., 'symmetric')` causes `ifftn` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifftn(..., 'nonsymmetric')` is the same as calling `ifftn(...)` without the argument `'nonsymmetric'`.

Remarks

For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error.

Algorithm

`ifftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = ifft(Y,[],p);
end
```

This computes in-place the one-dimensional inverse DFT along each dimension of `X`.

The execution time for `ifftn` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifftn` using the utility function `fftw`, which controls how MATLAB optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifftn` supports inputs of data types `double` and `single`. If you call `ifftn` with the syntax `y = ifftn(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fftn`, `fftw`, `ifft`, `ifft2`, `ifftshift`

Purpose

Inverse FFT shift

Syntax

```
ifftshift(X)
ifftshift(X,dim)
```

Description

`ifftshift(X)` swaps the left and right halves of the vector `X`. For matrices, `ifftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth. If `X` is a multidimensional array, `ifftshift(X)` swaps "half-spaces" of `X` along each dimension.

`ifftshift(X,dim)` applies the `ifftshift` operation along the dimension `dim`.

Note `ifftshift` undoes the results of `fftshift`. If the matrix `X` contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original `X`. Simply performing `fftshift(X)` twice will not produce `X`.

See Also

`fft`, `fft2`, `fftn`, `fftshift`

Purpose

Sparse incomplete LU factorization

Syntax

```
ilu(A,setup)
[L,U] = ilu(A,setup)
[L,U,P] = ilu(A,setup)
```

Description

`ilu` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`ilu(A,setup)` computes the incomplete LU factorization of `A`. `setup` is an input structure with up to five setup options. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`type`	Type of factorization. Values for `type` include: • `'nofill'`—Performs ILU factorization with `0` level of fill in, known as ILU(0). With `type` set to `'nofill'`, only the `milu` setup option is used; all other fields are ignored. • `'crout'`—Performs the Crout version of ILU factorization, known as ILUC. With `type` set to `'crout'`, only the `droptol` and `milu` setup options are used; all other fields are ignored. • `'ilutp'` (default)—Performs ILU factorization with threshold and pivoting. If `type` is not specified, the ILU factorization with pivoting ILUTP is performed. Pivoting is never performed with `type` set to `'nofill'` or `'crout'`.

Field Name	Description

`droptol`	Drop tolerance of the incomplete LU factorization. `droptol` is a non-negative scalar. The default value is `0`, which produces the complete LU factorization.
	
	The nonzero entries of `U` satisfy
	
	`abs(U(i,j)) >= droptol*norm((A:,j)),`
	
	with the exception of the diagonal entries, which are retained regardless of satisfying the criterion. The entries of `L` are tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`
	
	`abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).`

`milu`	Modified incomplete LU factorization. Values for `milu` include: • `'row'`—Produces the row-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `A*e = L*U*e`, where `e` is the vector of ones. • `'col'`—Produces the column-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `e'*A = e'*L*U`. • `'off'` (default)—No modified incomplete LU factorization is produced.

Field Name	Description
udiag	If udiag is 1, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is 0.
thresh	Pivot threshold between 0 (forces diagonal pivoting) and 1, the default, which always chooses the maximum magnitude entry in the column to be the pivot.

`ilu(A,setup)` returns `L+U-speye(size(A))`, where `L` is a unit lower triangular matrix and `U` is an upper triangular matrix.

`[L,U] = ilu(A,setup)` returns a unit lower triangular matrix in `L` and an upper triangular matrix in `U`.

`[L,U,P] = ilu(A,setup)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

Remarks

These incomplete factorizations may be useful as preconditioners for a system of linear equations being solved by iterative methods such as BICG (BiConjugate Gradients), GMRES (Generalized Minimum Residual Method).

Limitations

`ilu` works on sparse square matrices only.

Examples

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'crout';
setup.milu = 'row';
setup.droptol = 0.1;
[L,U] = ilu(A,setup);
e = ones(size(A,2),1);
norm(A*e-L*U*e)

ans =
```

```
    1.4251e-014
```

This shows that `A` and `L*U`, where `L` and `U` are given by the modified Crout `ILU`, have the same row-sum.

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'nofill';
nnz(A)
ans =

        7840

nnz(lu(A))
ans =

      126478

nnz(ilu(A,setup))
ans =

        7840
```

This shows that `A` has `7840` nonzeros, the complete LU factorization has `126478` nonzeros, and the incomplete LU factorization, with `0` level of fill-in, has `7840` nonzeros, the same amount as `A`.

See Also

`bicg`, `cholinc`,`gmres`,`luinc`

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

im2frame

Purpose

Convert image to movie frame

Syntax

```
f = im2frame(X,map)
f = im2frame(X)
```

Description

`f = im2frame(X,map)` converts the indexed image `X` and associated colormap `map` into a movie frame `f`. If `X` is a truecolor (m-by-n-by-3) image, then `map` is optional and has no effect.

Typical usage:

```
M(1) = im2frame(X1,map);
M(2) = im2frame(X2,map);
  ...
M(n) = im2frame(Xn,map);
movie(M)
```

`f = im2frame(X)` converts the indexed image `X` into a movie frame `f` using the current colormap if `X` contains an indexed image.

See Also

`frame2im`, `movie`

"Bit-Mapped Images" on page 1-92 for related functions

Purpose

Convert image to Java image

Syntax

```
jimage = im2java(I)
jimage = im2java(X,MAP)
jimage = im2java(RGB)
```

Description

To work with a MATLAB image in the Java environment, you must convert the image from its MATLAB representation into an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(I)` converts the intensity image `I` to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(X,MAP)` converts the indexed image `X`, with colormap `MAP`, to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(RGB)` converts the RGB image `RGB` to an instance of the Java image class, `java.awt.Image`.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`.

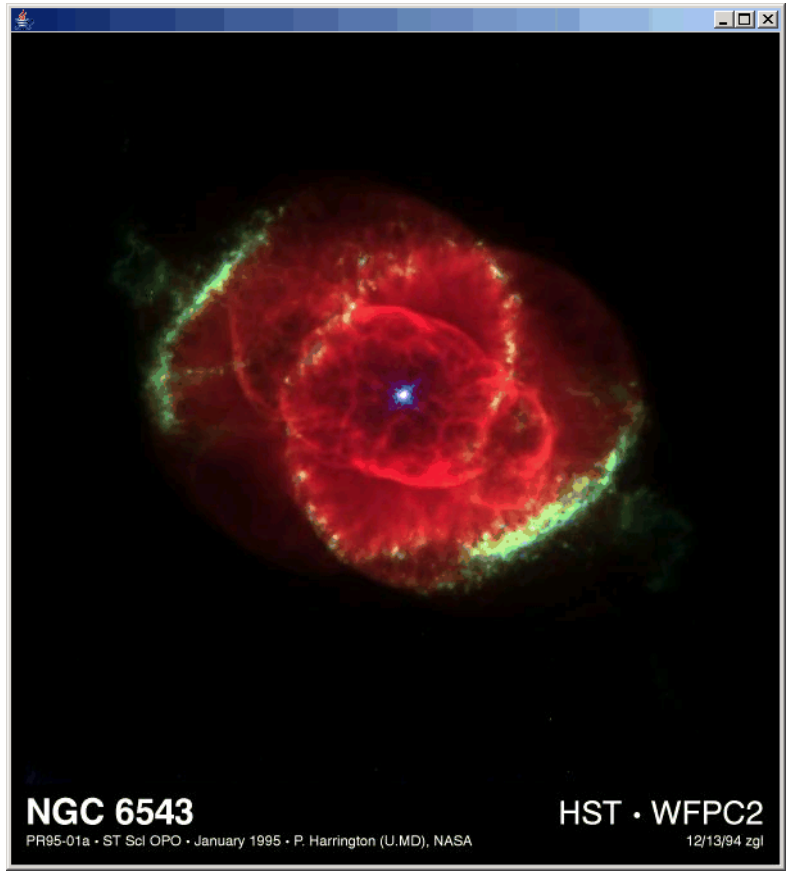
Note Java requires `uint8` data to create an instance of the Java image class, `java.awt.Image`. If the input image is of class `uint8`, `jimage` contains the same `uint8` data. If the input image is of class `double` or `uint16`, `im2java` makes an equivalent image of class `uint8`, rescaling or offsetting the data as necessary, and then converts this `uint8` representation to an instance of the Java image class, `java.awt.Image`.

Example

This example reads an image into the MATLAB workspace and then uses `im2java` to convert it into an instance of the Java image class.

```
I = imread('ngc6543a.jpg');
javaImage = im2java(I);
frame = javax.swing.JFrame;
icon = javax.swing.ImageIcon(javaImage);
label = javax.swing.JLabel(icon);
```

```
frame.getContentPane.add(label);
frame.pack
frame.show
```



See Also

"Bit-Mapped Images" on page 1-92 for related functions

Purpose

Imaginary part of complex number

Syntax

```
Y = imag(Z)
```

Description

`Y = imag(Z)` returns the imaginary part of the elements of array `Z`.

Examples

```
imag(2+3i)

ans =

    3
```

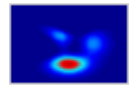
See Also

`conj`, `i`, `j`, `real`

image

Purpose

Display image object



GUI Alternatives

To plot a selected matrix as an image use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate image characteristics in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
image(C)
image(x,y,C)
image(x,y,C,'PropertyName',PropertyValue,...)
image('PropertyName',PropertyValue,...)
handle = image(...)
```

Description

`image` creates an image graphics object by interpreting each element in a matrix as an index into the figure's colormap or directly as RGB values, depending on the data specified.

The `image` function has two forms:

- A high-level function that calls `newplot` to determine where to draw the graphics objects and sets the following axes properties:
  - `XLim` and `YLim` to enclose the image
  - `Layer` to `top` to place the image in front of the tick marks and grid lines
  - `YDir` to `reverse`
  - `View` to `[0 90]`

- A low-level function that adds the image to the current axes without calling `newplot`. The low-level function argument list can contain only property name/property value pairs.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see set and get for examples of how to specify these data types).

`image(C)` displays matrix `C` as an image. Each element of `C` specifies the color of a rectangular segment in the image.

`image(x,y,C)`, where `x` and `y` are two-element vectors, specifies the range of the *x*- and *y*-axis labels, but produces the same image as `image(C)`. This can be useful, for example, if you want the axis tick labels to correspond to real physical dimensions represented by the image.

`image(x,y,C,'`*`PropertyName`*`',PropertyValue,...)` is a high-level function that also specifies property name/property value pairs. This syntax calls `newplot` before drawing the image.

`image('`*`PropertyName`*`',PropertyValue,...)` is the low-level syntax of the `image` function. It specifies only property name/property value pairs as input arguments.

`handle = image(...)` returns the handle of the image object it creates. You can obtain the handle with all forms of the `image` function.

Remarks

Image data can be either indexed or true color. An indexed image stores colors as an array of indices into the figure colormap. A true color image does not use a colormap; instead, the color values for each pixel are stored directly as RGB triplets. In MATLAB, the `CData` property of a truecolor image object is a three-dimensional (`m`-by-`n`-by-3) array. This array consists of three `m`-by-`n` matrices (representing the red, green, and blue color planes) concatenated along the third dimension.

The `imread` function reads image data into MATLAB arrays from graphics files in various standard formats, such as TIFF. You can write MATLAB image data to graphics files using the `imwrite` function.

`imread` and `imwrite` both support a variety of graphics file formats and compression schemes.

When you read image data into MATLAB using `imread`, the data is usually stored as an array of 8-bit integers. However, `imread` also supports reading 16-bit-per-pixel data from TIFF and PNG files. These are more efficient storage methods than the double-precision (64-bit) floating-point numbers that MATLAB typically uses. However, it is necessary for MATLAB to interpret 8-bit and 16-bit image data differently from 64-bit data. This table summarizes these differences.

You cannot interactively pan or zoom outside the *x*-limits or *y*-limits of an image, unless the axes limits are already been set outside the bounds of the image, in which case there is no such restriction. If other objects (such as lineseries) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Image Type	Double-Precision Data (double Array)	8-Bit Data (uint8 Array) 16-Bit Data (uint16 Array)
Indexed (colormap)	Image is stored as a two-dimensional (`m`-by-`n`) array of integers in the range [1, `length(colormap)`]; `colormap` is an `m`-by-3 array of floating-point values in the range [0, 1].	Image is stored as a two-dimensional (`m`-by-`n`) array of integers in the range [0, 255] (`uint8`) or [0, 65535] (`uint16`); `colormap` is an `m`-by-3 array of floating-point values in the range [0, 1].
True color (RGB)	Image is stored as a three-dimensional (`m`-by-`n`-by-3) array of floating-point values in the range [0, 1].	Image is stored as a three-dimensional (`m`-by-`n`-by-3) array of integers in the range [0, 255] (`uint8`) or [0, 65535] (`uint16`).

Indexed Images

In an indexed image of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. In a `uint8` or `uint16` indexed image, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

If you want to convert a `uint8` or `uint16` indexed image to `double`, you need to add 1 to the result. For example,

```
X64 = double(X8) + 1;
```

or

```
X64 = double(X16) + 1;
```

To convert from `double` to `uint8` or `uint16`, you need to first subtract 1, and then use `round` to ensure all the values are integers.

```
X8 = uint8(round(X64 - 1));
```

or

```
X16 = uint16(round(X64 - 1));
```

When you write an indexed image using `imwrite`, MATLAB automatically converts the values if necessary.

Colormaps

Colormaps in MATLAB are always `m`-by-3 arrays of double-precision floating-point numbers in the range [0, 1]. In most graphics file formats, colormaps are stored as integers, but MATLAB does not support colormaps with integer values. `imread` and `imwrite` automatically convert colormap values when reading and writing files.

True Color Images

In a true color image of class `double`, the data values are floating-point numbers in the range [0, 1]. In a true color image of class `uint8`, the data values are integers in the range [0, 255], and for true color images of class `uint16` the data values are integers in the range [0, 65535].

If you want to convert a true color image from one data type to the other, you must rescale the data. For example, this statement converts a `uint8` true color image to `double`.

```
RGB64 = double(RGB8)/255;
```

or for `uint16` images,

```
RGB64 = double(RGB16)/65535;
```

This statement converts a `double` true color image to `uint8`:

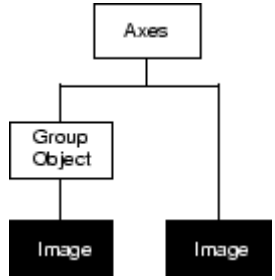
```
RGB8 = uint8(round(RGB64*255));
```

or to obtain `uint16` images, type

```
RGB16 = uint16(round(RGB64*65535));
```

When you write a true color image using `imwrite`, MATLAB automatically converts the values if necessary.

Object Hierarchy



Setting Default Properties

You can set default image properties on the axes, figure, and root levels:

```
set(0,'DefaultImageProperty',PropertyValue...)
set(gcf,'DefaultImageProperty',PropertyValue...)
set(gca,'DefaultImageProperty',PropertyValue...)
```

where *Property* is the name of the image property and PropertyValue is the value you are specifying. Use set and get to access image properties.

Example

Example 1

Load a mat-file containing a photograph of a colorful primate. Display the indexed image using its associated colormap.

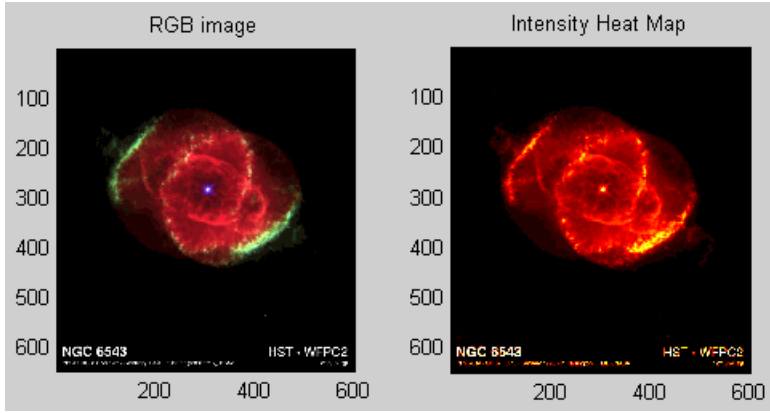
```
load mandrill
figure('color','k')
image(X)
colormap(map)
axis off          % Remove axis ticks and numbers
axis image        % Set aspect ratio to obtain square pixels
```

Example 2

Load a JPEG image file of the Cat's Eye Nebula from the Hubble Space Telescope (image courtesy NASA). Display the original image using its RGB color values (left) as a subplot. Create a linked subplot (same

size and scale) to display the transformed intensity image as a heat map (right).

```
figure
ax(1) = subplot(1,2,1);
rgb = imread('ngc6543a.jpg');
image(rgb); title('RGB image')
ax(2) = subplot(122);
im = mean(rgb,3);
image(im); title('Intensity Heat Map')
colormap(hot(256))
linkaxes(ax,'xy')
axis(ax,'image')
```



See Also

`imagesc`, `imfinfo`, `imread`, `imwrite`, `colormap`, `pcolor`, `newplot`, `surface`

"Displaying Bit-Mapped Images"

"Bit-Mapped Images" on page 1-92 for related functions

Image Properties for property descriptions

Purpose

Define image properties

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Image Properties

This section lists property names along with the types of values each property accepts.

`AlphaData`

m-by-n matrix of `double` or `uint8`

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class double or uint8.

MATLAB determines the transparency in one of three ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)
- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`
`{none} | direct| scaled`

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. It can be any of the following:

- `none` — The transparency values of `AlphaData` are between 0 and 1 or are clamped to this range (the default).
- `scaled` — Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` — Use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`Annotation`
`hg.Annotation` object Read Only

Control the display of image objects in legends. The `Annotation` property enables you to specify whether this image object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the image object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this image object in a legend (default)
`off`	Do not include this image object in a legend
`children`	Same as `on` because image objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object’s `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

`CData`

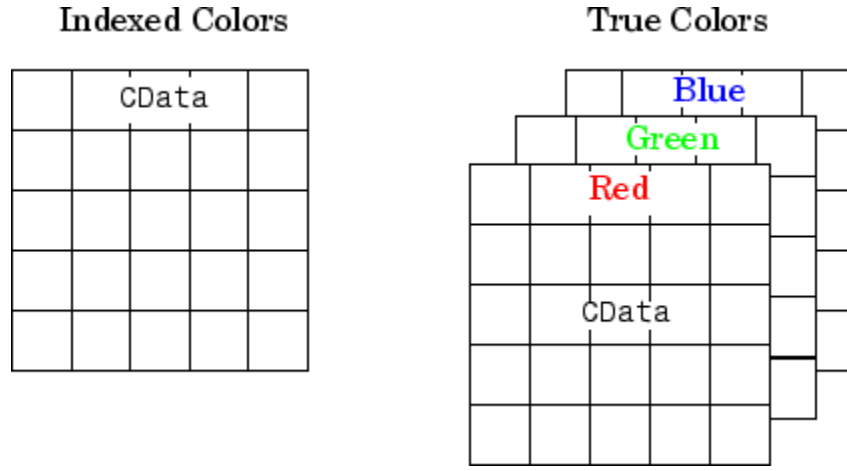
matrix or m-by-n-by-3 array

The image data. A matrix or 3-D array of values specifying the color of each rectangular area defining the image. `image(C)` assigns the values of `C` to `CData`. MATLAB determines the coloring of the image in one of three ways:

- Using the elements of `CData` as indices into the current colormap (the default) (`CDataMapping` set to `direct`)
- Scaling the elements of `CData` to range between the values `min(get(gca,'CLim'))` and `max(get(gca,'CLim'))` (`CDataMapping` set to `scaled`)
- Interpreting the elements of `CData` directly as RGB values (true color specification)

Note that the behavior of `NaN`s in image `CData` is not defined. See the image `AlphaData` property for information on using transparency with images.

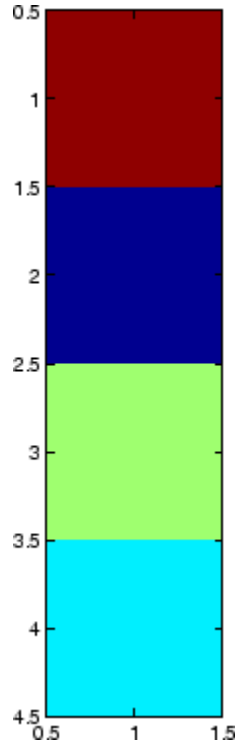
A true color specification for `CData` requires an m-by-n-by-3 array of RGB values. The first page contains the red component, the second page the green component, and the third page the blue component of each element in the image. RGB values range from 0 to 1. The following picture illustrates the relative dimensions of `CData` for the two color models.



If `CData` has only one row or column, the height or width respectively is always one data unit and is centered about the first `YData` or `XData` element respectively. For example, using a 4-by-1 matrix of random data,

```
C = rand(4,1);
image(C,'CDataMapping','scaled')
axis image
```

produces



`CDataMapping`
`scaled | {direct}`

Direct or scaled indexed colors. This property determines whether MATLAB interprets the values in `CData` as indices into the figure colormap (the default) or scales the values according to the values of the axes `CLim` property.

When `CDataMapping` is `direct`, the values of `CData` should be in the range 1 to `length(get(gcf,'Colormap'))`. If you use true color specification for `CData`, this property has no effect.

`Children`
handles

The empty matrix; image objects have no children.

`Clipping`
`on | off`

Clipping mode. By default, MATLAB clips images to the axes rectangle. If you set `Clipping` to `off`, the image can be displayed outside the axes rectangle. For example, if you create an image, set `hold` to `on`, freeze axis scaling (with `axis manual`), and then create a larger image, it extends beyond the axis limits.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates an image object. You must define this property as a default value for images or in a call to the `image` function to create a new image object. For example, the statement

```
set(0,'DefaultImageCreateFcn','axis image')
```

defines a default value on the root level that sets the aspect ratio and the axis limits so the image has square pixels. MATLAB executes this routine after setting all image properties. Setting this property on an existing image object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue

a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this image object. The `legend` function uses the string defined by the `DisplayName` property to label this image object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this image object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

```
{on} | callback | off
```

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`Parent`

handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For image objects, `Type` is always '`image`'.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
`[1 size(CData,2)]` by default

Control placement of image along x-axis. A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

The width of each `CData` element is determined by the expression

```
(XData(2)-XData(1))/(size(CData,2)-1)
```

You can also specify a single value for `XData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.

`YData`

`[1 size(CData,1)]` by default

Control placement of image along y-axis. A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

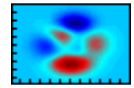
The height of each `CData` element is determined by the expression

```
(YData(2)-YData(1))/(size(CData,1)-1)
```

You can also specify a single value for `YData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.

Purpose

Scale data and display image object



GUI Alternatives

To plot a selected matrix as an image use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate image characteristics in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
imagesc(C)
imagesc(x,y,C)
imagesc(...,clims)
h = imagesc(...)
```

Description

The `imagesc` function scales image data to the full range of the current colormap and displays the image. (See “Examples” on page 2-1650 for an illustration.)

`imagesc(C)` displays `C` as an image. Each element of `C` corresponds to a rectangular area in the image. The values of the elements of `C` are indices into the current colormap that determine the color of each patch.

`imagesc(x,y,C)` displays `C` as an image and specifies the bounds of the *x*- and *y*-axis with vectors `x` and `y`.

`imagesc(...,clims)` normalizes the values in `C` to the range specified by `clims` and displays `C` as an image. `clims` is a two-element vector that limits the range of data values in `C`. These values map to the full range of values in the current colormap.

`h = imagesc(...)` returns the handle for an image graphics object.

Remarks

`x` and `y` do not affect the elements in `C`; they only affect the annotation of the axes. If `length(x) > 2` or `length(y) > 2`, `imagesc` ignores all except the first and last elements of the respective vector.

`imagesc` creates an image with `CDataMapping` set to scaled, and sets the axes `CLim` property to the value passed in `clims`.

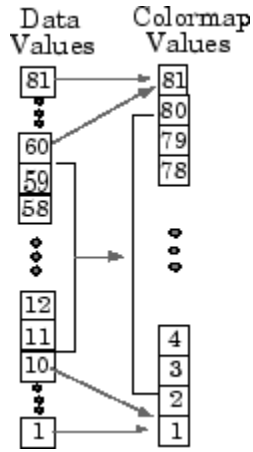
You cannot interactively pan or zoom outside the *x*-limits or *y*-limits of an image.

Examples

You can expand midrange color resolution by mapping low values to the first color and high values to the last color in the colormap by specifying color value limits (`clims`). If the size of the current colormap is 81-by-3, the statements

```
clims = [ 10 60 ]
imagesc(C,clims)
```

map the data values in C to the colormap as shown in this illustration and the code that follows:

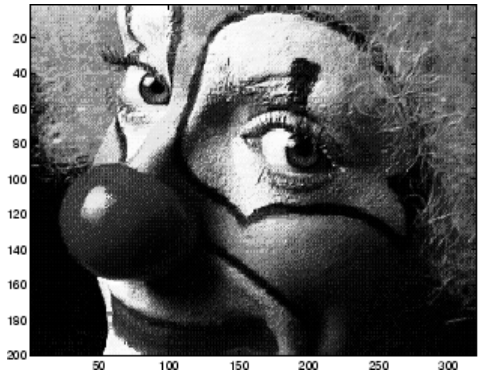


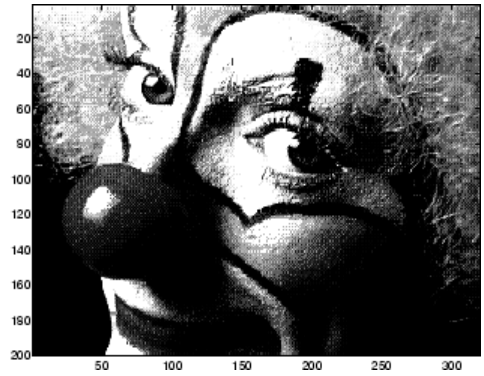
In this example, the left image maps to the `gray` colormap using the statements

```
load clown
imagesc(X)
colormap(gray)
```

The right image has values between 10 and 60 scaled to the full range of the gray colormap using the statements

```
load clown
clims = [10 60];
imagesc(X,clims)
colormap(gray)
```





See Also

image, imfinfo, imread, imwrite, colorbar, colormap, pcolor, surface, surf

"Bit-Mapped Images" on page 1-92 for related functions

Purpose

Information about graphics file

Syntax

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
info = imfino(URL,...)
```

Description

`info = imfinfo(filename,fmt)` returns a structure, `info`, whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and *`fmt`* is a string that specifies the format of the file. The file must be in the current directory or in a directory on the MATLAB path. If `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.`*`fmt`*. The possible values for *`fmt`* are contained in the MATLAB file format registry. To view of list of these formats, run the `imformats` command.

If `filename` is a TIFF, HDF, ICO, GIF, or CUR file containing more than one image, `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file.

`info = imfinfo(filename)` attempts to infer the format of the file from its contents.

`info = imfino(URL,...)` reads the image from the specified Internet `URL`. The URL must include the protocol type (e.g., `http://`)

Information Returned

The set of fields in `info` depends on the individual file and its format. However, the first nine fields are always the same. This table lists these common fields, in the order they appear in the structure, and describes their values.

Field	Value
`Filename`	A string containing the name of the file; if the file is not in the current directory, the string contains the full pathname of the file.

Field	Value
`FileModDate`	A string containing the date when the file was last modified
`FileSize`	An integer indicating the size of the file in bytes
`Format`	A string containing the file format, as specified by *`fmt`*; for JPEG and TIFF files, the three-letter variant is returned.
`FormatVersion`	A string or number describing the version of the format
`Width`	An integer indicating the width of the image in pixels
`Height`	An integer indicating the height of the image in pixels
`BitDepth`	An integer indicating the number of bits per pixel
`ColorType`	A string indicating the type of image; either `'truecolor'` for a truecolor RGB image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image

Example

```
info = imfinfo('canoe.tif')

info =

                    Filename: [1x76 char]
                FileModDate: '04-Dec-2000 13:57:55'
                    FileSize: 69708
                      Format: 'tif'
                FormatVersion: []
                       Width: 346
                      Height: 207
                    BitDepth: 8
                   ColorType: 'indexed'
```

```
                FormatSignature: [73 73 42 0]
                      ByteOrder: 'little-endian'
                 NewSubFileType: 0
                  BitsPerSample: 8
                    Compression: 'PackBits'
PhotometricInterpretation: 'RGB Palette'
                   StripOffsets: [9x1 double]
               SamplesPerPixel: 1
                  RowsPerStrip: 23
               StripByteCounts: [9x1 double]
                   XResolution: 72
                   YResolution: 72
                ResolutionUnit: 'Inch'
                      Colormap: [256x3 double]
      PlanarConfiguration: 'Chunky'
                 TileWidth: []
                TileLength: []
                TileOffsets: []
             TileByteCounts: []
                Orientation: 1
                  FillOrder: 1
            GrayResponseUnit: 0.0100
              MaxSampleValue: 255
              MinSampleValue: 0
                Thresholding: 1
```

See Also

`imformats`, `imread`, `imwrite`

“Bit-Mapped Images” on page 1-92 for related functions

Purpose

Manage image file format registry

Syntax

```
imformats
formats = imformats
formats = imformats('fmt')
formats = imformats(format_struct)
formats = imformats('factory')
```

Description

`imformats` displays a table of information listing all the values in the MATLAB file format registry. This registry determines which file formats are supported by the `imfinfo`, `imread`, and `imwrite` functions.

`formats = imformats` returns a structure containing all the values in the MATLAB file format registry. The following tables lists the fields in the order they appear in the structure.

Field	Value
`ext`	A cell array of strings that specify filename extensions that are valid for this format
`isa`	A string specifying the name of the function that determines if a file is a certain format. This can also be a function handle.
`info`	A string specifying the name of the function that reads information about a file. This can also be a function handle.
`read`	A string specifying the name of the function that reads image data in a file. This can also be a function handle.
`write`	A string specifying the name of the function that writes MATLAB data to a file. This can also be a function handle.
`alpha`	Returns `1` if the format has an alpha channel, `0` otherwise
`description`	A text description of the file format

Note The values for the `isa`, `info`, `read`, and `write` fields must be functions on the MATLAB search path or function handles.

`formats = imformats('fmt')` searches the known formats in the MATLAB file format registry for the format associated with the filename extension `'fmt'`. If found, `imformats` returns a structure containing the characteristics and function names associated with the format. Otherwise, it returns an empty structure.

`formats = imformats(format_struct)` sets the MATLAB file format registry to the values in `format_struct`. The output structure, `formats`, contains the new registry settings.

Caution Using `imformats` to specify values in the MATLAB file format registry can result in the inability to load any image files. To return the file format registry to a working state, use `imformats` with the `'factory'` setting.

`formats = imformats('factory')` resets the MATLAB file format registry to the default format registry values. This removes any user-specified settings.

Changes to the format registry do not persist between MATLAB sessions. To have a format always available when you start MATLAB, add the appropriate `imformats` command to the MATLAB startup file, `startup.m`, located in `$MATLAB/toolbox/local` on UNIX systems, or `$MATLAB\toolbox\local` on Windows systems.

Example

```
formats = imformats;
formats(1)

ans =

            ext: {'bmp'}
```

```
          isa: @isbmp
         info: @imbmpinfo
         read: @readbmp
        write: @writebmp
        alpha: 0
  description: 'Windows Bitmap (BMP)'
```

See Also

`fileformats`, `imfinfo`, `imread`, `imwrite`, `path`

"Bit-Mapped Images" on page 1-92 for related functions

import

Purpose

Add package or class to current Java import list

Syntax

```
import package_name.*
import class_name
import cls_or_pkg_name1 cls_or_pkg_name2...
import
L = import
```

Description

`import package_name.*` adds all the classes in *`package_name`* to the current import list. Note that *`package_name`* must be followed by `.*`.

`import class_name` adds a single class to the current import list. Note that *`class_name`* must be fully qualified (that is, it must include the package name).

`import cls_or_pkg_name1 cls_or_pkg_name2...` adds all named classes and packages to the current import list. Note that each class name must be fully qualified, and each package name must be followed by `.*`.

`import` with no input arguments displays the current import list, without adding to it.

`L = import` with no input arguments returns a cell array of strings containing the current import list, without adding to it.

The `import` command operates exclusively on the import list of the function from which it is invoked. When invoked at the command prompt, `import` uses the import list for the MATLAB command environment. If `import` is used in a script invoked from a function, it affects the import list of the function. If `import` is used in a script that is invoked from the command prompt, it affects the import list for the command environment.

The import list of a function is persistent across calls to that function and is only cleared when the function is cleared.

To clear the current import list, use the following command.

```
clear import
```

This command may only be invoked at the command prompt. Attempting to use `clear import` within a function results in an error.

Remarks

The only reason for using `import` is to allow your code to refer to each imported class with the immediate class name only, rather than with the fully qualified class name. `import` is particularly useful in streamlining calls to constructors, where most references to Java classes occur.

Examples

This example shows importing and using the single class, `java.lang.String`, and two complete packages, `java.util` and `java.awt`.

```
import java.lang.String
import java.util.* java.awt.*
f = Frame;             % Create java.awt.Frame object
s = String('hello');   % Create java.lang.String object
methods Enumeration    % List java.util.Enumeration methods
```

See Also

`clear`, `importdata`

importdata

Purpose

Load data from disk file

Syntax

```
importdata(filename)
A = importdata(filename)
A = importdata(filename,delimiter)
A = importdata(filename,delimiter,headerline)
[A D] = importdata(...)
[A D H] = importdata(...)
[...] = importdata('-pastespecial', ...)
```

Description

`importdata(filename)` loads data from `filename` into the workspace. The `filename` input is a string enclosed in single quotes.

`A = importdata(filename)` loads data from `filename` into structure `A`.

`A = importdata(filename,delimiter)` loads data from `filename` using `delimiter` as the column separator. The `delimiter` argument must be a string enclosed in single quotes. Use `'\t'` for tab. When importing from an ASCII file, `delimiter` only separates numeric data.

`A = importdata(filename,delimiter,headerline)` where `headerline` is a number that indicates on which line of the file the header text is located, loads data from line `headerline+1` to the end of the file.

`[A D] = importdata(...)` returns the output structure in `A`, and the delimiter character in `D`.

`[A D H] = importdata(...)` returns the output structure in `A`, the delimiter character in `D`, and the line number of the header in `H`.

`[...] = importdata('-pastespecial', ...)` loads data from your computer's paste buffer rather than from a file.

Remarks

`importdata` looks at the file extension to determine which helper function to use. If it can recognize the file extension, `importdata` calls the appropriate helper function, specifying the maximum number of output arguments. If it cannot recognize the file extension, `importdata` calls `finfo` to determine which helper function to use. If no helper

function is defined for this file extension, `importdata` treats the file as delimited text. `importdata` removes from the result empty outputs returned from the helper function.

Examples

Example 1 — A Simple Import

Import data from file `ding.wav`:

```
s = importdata('ding.wav')
s =

    data: [11554x1 double]
      fs: 22050
```

Example 2 — Importing with Delimiter and Header

Use `importdata` to read in a text file. The third input argument is `colheaders`, which is the number of lines that belong to the header:

```
type 'myfile.txt'

 Day1  Day2  Day3  Day4  Day5  Day6  Day7
95.01 76.21 61.54 40.57  5.79 20.28  1.53
23.11 45.65 79.19 93.55 35.29 19.87 74.68
60.68  1.85 92.18 91.69 81.32 60.38 44.51
48.60 82.14 73.82 41.03  0.99 27.22 93.18
89.13 44.47 17.63 89.36 13.89 19.88 46.60
```

Import from the file, specifying the space character as the delimiter and 1 row for the column header. Assign the output to variable `M`:

```
M = importdata('myfile.txt', ' ', 1);
```

Print out columns 3 and 5, including the header for those columns:

```
for k=3:2:5
   M.colheaders(1,k)
   M.data(:,k)
   disp ' '
end
```

```
ans =
    'Day3'
ans =
   61.5400
   79.1900
   92.1800
   73.8200
   17.6300

ans =
    'Day5'
ans =
    5.7900
   35.2900
   81.3200
    0.9900
   13.8900
```

See Also

load

Purpose

Read image from graphics file

Syntax

```
A = imread(filename, fmt)
[X, map] = imread(...)
[...]  = imread(filename)
[...]  = imread(URL,...)
[...]  = imread(..., idx)  CUR or ICO
[A, map, alpha] = imread(...)  CUR or ICO
[...]  = imread(..., idx)  GIF
[...]  = imread(..., 'frames', idx)  GIF
[...]  = imread(..., ref)  HDF4
[...]  = imread(...,'BackgroundColor',BG) PNG
[A, map, alpha] = imread(...)  PNG
[...]  = imread(..., idx)  TIFF
[...]  = imread(..., 'PixelRegion', {ROWS, COLS})  TIFF
```

Description

`A = imread(filename, fmt)` reads a grayscale or color image from the file specified by the string `filename`. If the file is not in the current directory, or in a directory on the MATLAB path, specify the full pathname.

The text string *`fmt`* specifies the format of the file by its standard file extension. For example, specify `'gif'` for Graphics Interchange Format files. To see a list of supported formats, with their file extensions, use the `imformats` function. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

The return value `A` is an array containing the image data. If the file contains a grayscale image, `A` is an M-by-N array. If the file contains a truecolor image, `A` is an M-by-N-by-3 array. For TIFF files containing color images that use the CMYK color space, `A` is an M-by-N-by-4 array. See TIFF in the Format-Specific Information section for more information.

The class of `A` depends on the bits-per-sample of the image data, rounded to the next byte boundary. For example, `imread` returns 24-bit color data as an array of `uint8` data because the sample size for

each color component is 8 bits. See "Remarks" on page 2-1664 for a discussion of bitdepths, and see "Format-Specific Information" on page 2-1664 for more detail about supported bitdepths and sample sizes for a particular format.

`[X, map] = imread(...)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. Colormap values in the image file are automatically rescaled into the range `[0,1]`.

`[...] = imread(filename)` attempts to infer the format of the file from its content.

`[...] = imread(URL,...)` reads the image from an Internet URL. The `URL` must include the protocol type (e.g., `http://`).

See the format-specific sections for additional syntaxes.

Remarks

Bitdepth is the number of bits used to represent each image pixel. Bitdepth is calculated by multiplying the bits-per-sample with the samples-per-pixel. Thus, a format that uses 8-bits for each color component (or sample) and three samples per pixel has a bitdepth of 24. Sometimes the sample size associated with a bitdepth can be ambiguous: does a 48-bit bitdepth represent six 8-bit samples, four 12-bit samples, or three 16-bit samples? The following format-specific sections provide sample size information to avoid this ambiguity.

Format-Specific Information

The following sections provide information about the support for specific formats, listed in alphabetical order by format name. These sections include information about format-specific syntaxes, if they exist. The following is a list of links to the various sections.

BMP — Windows Bitmap

The following table lists the supported bitdepths, compression, and output classes for BMP data.

Supported Bitdepths	No Compression	RLE Compression	Output Class	Notes
1-bit	x	–	logical	
4-bit	x	x	`uint8`	
8-bit	x	x	`uint8`	
16-bit	x	–	`uint8`	1 sample/pixel
24-bit	x	–	`uint8`	3 samples/pixel
32-bit	x	–	`uint8`	3 samples/pixel (1 byte padding)

CUR — Cursor File

The following table lists the supported bitdepths, compression, and output classes for Cursor files and Icon files.

Supported Bitdepths	No Compression	Compression	Output Class
1-bit	x	–	logical
4-bit	x	–	`uint8`
8-bit	x	–	`uint8`

The following are format-specific syntaxes for Cursor files and Icon files.

`[...] = imread(..., idx)` `CUR or ICO` reads in one image from a multi-image icon or cursor file. `idx` is an integer value that specifies the order that the image appears in the file. For example, if `idx` is `3`, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[A, map, alpha] = imread(...)` `CUR or ICO` returns the AND mask for the resource, which can be used to determine the transparency information. For cursor files, this mask may contain the only useful data.

Note By default, Microsoft Windows cursors are 32-by-32 pixels. MATLAB pointers must be 16-by-16. You will probably need to scale your image. If you have Image Processing Toolbox, you can use the `imresize` function.

GIF — Graphics Interchange Format

The following table lists the supported bitdepths, compression, and output classes for GIF files.

Supported Bitdepths	No Compression	Compression	Output Class
1-bit	x	–	logical
2-bit to 8-bit	x	–	`uint8`

The following are format-specific syntaxes for GIF files.

`[...] = imread(..., idx)` `GIF` reads in one or more frames from a multiframe (i.e., animated) GIF file. `idx` must be an integer scalar or vector of integer values. For example, if `idx` is `3`, `imread` reads the third image in the file. If `idx` is `1:5`, `imread` returns only the first five frames.

`[...] = imread(..., 'frames', idx)` `GIF` is the same as the syntax above except that `idx` can be `'all'`. In this case, all the frames are read and returned in the order that they appear in the file.

Note Because of the way that GIF files are structured, all the frames must be read when a particular frame is requested. Consequently, it is much faster to specify a vector of frames or `'all'` for `idx` than to call `imread` in a loop when reading multiple frames from the same GIF file.

HDF4 — Hierarchical Data Format

The following table lists the supported bitdepths, compression, and output classes for HDF4 files.

Supported Bitdepths	Raster Image with colormap	Raster image without colormap	Output Class	Notes
8-bit	x	x	`uint8`	
24-bit	–	–	`uint8`	3 samples/pixel

The following are format-specific syntaxes for HDF4 files.

`[...] = imread(..., ref)` `HDF4` reads in one image from a multi-image HDF4 file. `ref` is an integer value that specifies the reference number used to identify the image. For example, if `ref` is `12`, `imread` reads the image whose reference number is 12. (Note that in an HDF4 file the reference numbers do not necessarily correspond to the order of the images in the file. You can use `imfinfo` to match image

order with reference number.) If you omit this argument, `imread` reads the first image in the file.

ICO — Icon File

See CUR – Cursor File

JPEG — Joint Photographic Experts Group

`imread` can read any baseline JPEG image as well as JPEG images with some commonly used extensions. The following table lists the supported bitdepths, compression, and output classes for JPEG files.

Supported Bitdepths	Lossy Compression	Lossless Compression	Output Class	Notes
8-bit	x	x	`uint8`	Grayscale or RGB
12-bit	x	x	`uint16`	Grayscale
16-bit	–	x	`uint16`	Grayscale
36-bit	x	x	`uint16`	RGB Three 12-bit samples/pixel

PBM — Portable Bitmap

The following table lists the supported bitdepths, compression, and output classes for PBM files.

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
1-bit	x	x	logical

PCX — Windows Paintbrush

The following table lists the supported bitdepths, compression, and output classes for PCX files.

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Grayscale only
8-bit	`uint8`	Grayscale or indexed
24-bit	`uint8`	RGB Three 8-bit samples/pixel

PGM — Portable Graymap

The following table lists the supported bitdepths, compression, and output classes for PGM files.

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16-bit	x	–	`uint8`
Arbitrary	–	x	

PNG — Portable Network Graphics

The following table lists the supported bitdepths, compression, and output classes for PNG data.

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Grayscale
2-bit	`uint8`	Grayscale
4-bit	`uint8`	Grayscale
8-bit	`uint8`	Grayscale or Indexed
16-bit	`uint16`	Grayscale or Indexed

Supported Bitdepths	Output Class	Notes
24-bit	`uint8`	RGB Three 8-bit samples/pixel.
48-bit	`uint16`	RGB Three 16-bit samples/pixel.

The following are format-specific syntaxes for PNG files.

`[...] = imread(...,'BackgroundColor',BG)` `PNG` composites any transparent pixels in the input image against the color specified in `BG`. If `BG` is `'none'`, then no compositing is performed. If the input image is indexed, `BG` must be an integer in the range `[1,P]` where `P` is the colormap length. If the input image is grayscale, `BG` should be an integer in the range `[0,1]`. If the input image is RGB, `BG` should be a three-element vector whose values are in the range `[0,1]`. The string `'BackgroundColor'` may be abbreviated.

`[A, map, alpha] = imread(...)` `PNG` returns the alpha channel if one is present; otherwise `alpha` is `[]`. Note that `map` may be empty if the file contains a grayscale or truecolor image.

If the `alpha` output argument is specified, `BG` defaults to `'none'`, if not specified by the user. Otherwise, if the PNG file contains a background color chunk, that color is used as the default value for `BG`. If `alpha` is not used and the file does not contain a background color chunk, then the default value for `BG` is `1` for indexed images; `0` for grayscale images; and `[0 0 0]` for truecolor images.

PPM — Portable Pixmap

The following table lists the supported bitdepths, compression, and output classes for PPM files.

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16-bit	x	–	`uint8`
Arbitrary	–	x	

RAS — Sun Raster

The following table lists the supported bitdepths, compression, and output classes for RAS files.

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Bitmap
8-bit	`uint8`	Indexed
24-bit	`uint8`	RGB Three 8-bit samples/pixel
32-bit	`uint8`	RGB with Alpha Four 8-bit samples/pixel

TIFF — Tagged Image File Format

`imread` supports the following TIFF capabilities:

- Any number of samples-per-pixel
- CCITT group 3 and 4 FAX, Packbits, JPEG, LZW, Deflate, ThunderScan compression, and uncompressed images
- Logical, grayscale, indexed color, truecolor and hyperspectral images
- RGB, CMYK, CIELAB, ICCLAB color spaces
- Data organized into tiles or scanlines

The following table lists the supported bit/sample and corresponding output classes for TIFF files.

Bits-per-Sample	Sample Format	Output Class
1	integer	logical
2 – 8	integer	uint8
9 – 16	integer	uint16
17 – 32	integer	uint32
32	float	single
33 – 64	integer	uint64
64	float	double

The following are format-specific syntaxes for TIFF files.

`A = imread(...)` returns color data that uses the RGB, CIELAB, ICCLAB, or CMYK color spaces. If the color image uses the CMYK color space, A is an M-by-N-by-4 array.

`[...] = imread(..., idx)` reads in one image from a multi-image TIFF file. `idx` is an integer value that specifies the order in which the image appears in the file. For example, if `idx` is `3`, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[...] = imread(..., 'PixelRegion', {ROWS, COLS})` returns the subimage specified by the boundaries in `ROWS` and `COLS`. For tiled TIFF images, `imread` reads only the tiles that encompass the region specified by `ROWS` and `COLS`, improving memory efficiency and performance. `ROWS` and `COLS` must be either two or three element vectors. If two elements are provided, they denote the 1-based indices `[START STOP]`. If three elements are provided, the indices `[START INCREMENT STOP]` allow image downsampling.

For TIFF files, `imread` can read color data represented in the RGB, CIELAB, or ICCLAB color spaces. To determine which color space is used, look at the value of the `PhotometricInterpretation` field returned by `imfinfo`. Note, however, that if a file contains CIELAB color data, `imread` converts it to ICCLAB before bringing it into the MATLAB workspace. 8- or 16-bit TIFF CIELAB-encoded values use a

mixture of signed and unsigned data types that cannot be represented as a single MATLAB array.

XWD — X Window Dump

The following table lists the supported bitdepths, compression, and output classes for XWD files.

Supported Bitdepths	ZPixmaps	XYBitmaps	XYPixmaps	Output Class
1-bit	x	–	x	`logical`
8-bit	x	–	–	`uint8`

Class Support

For most image file formats, `imread` uses 8 or fewer bits per color plane to store image pixels. The following table lists the class of the returned array for the data types used by the file formats.

Data Type Used in File	Class of Array Returned by imread
1-bit per pixel	`logical`
2- to 8-bits per color plane	`uint8`
9- to 16-bit per pixel	`uint16` (`BMP`, `JPEG`, `PNG`, and `TIFF`) For the 16-bit BMP packed format (5-6-5), MATLAB returns `uint8`

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

Examples

This example reads the sixth image in a TIFF file.

```
[X,map] = imread('your_image.tif',6);
```

This example reads the fourth image in an HDF4 file.

```
info = imfinfo('your_hdf_file.hdf');
[X,map] = imread('your_hdf_file.hdf',info(4).Reference);
```

This example reads a 24-bit PNG image and sets any of its fully transparent (alpha channel) pixels to red.

```
bg = [255 0 0];
A = imread('your_image.png','BackgroundColor',bg);
```

This example returns the alpha channel (if any) of a PNG image.

```
[A,map,alpha] = imread('your_image.png');
```

This example reads an ICO image, applies a transparency mask, and then displays the image.

```
[a,b,c] = imread('your_icon.ico');
% Augment colormap for background color (white).
b2 = [b; 1 1 1];
% Create new image for display.
d = ones(size(a)) * (length(b2) - 1);
% Use the AND mask to mix the background and
% foreground data on the new image
d(c == 0) = a(c == 0);
% Display new image
image(uint8(d)), colormap(b2)
```

See Also

`double`, `fread`, `image`, `imfinfo`, `imformats`, `imwrite`, `uint8`, `uint16`

“Bit-Mapped Images” on page 1-92 for related functions

Purpose

Write image to graphics file

Syntax

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Param1,Val1,Param2,Val2...)
```

Description

`imwrite(A,filename,fmt)` writes the image `A` to the file specified by `filename` in the format specified by *`fmt`*.

`A` can be an M-by-N (grayscale image) or M-by-N-by-3 (truecolor image) array. `A` cannot be an empty array. If the format specified is TIFF, `imwrite` can also accept an M-by-N-by-4 array containing color data that uses the CMYK color space. For information about the class of the input array and the output image, see "Class Support" on page 2-1687.

`filename` is a string that specifies the name of the output file.

`fmt` can be any of the text strings listed in the table in "Supported Formats" on page 2-1676. This list of supported formats is determined by the MATLAB image file format registry. See `imformats` for more information about this registry.

`imwrite(X,map,filename,fmt)` writes the indexed image in `X` and its associated colormap `map` to `filename` in the format specified by *`fmt`*. If `X` is of class `uint8` or `uint16`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, the `imwrite` function offsets the values in the array before writing, using `uint8(X 1)`. The `map` parameter must be a valid MATLAB colormap. Note that most image file formats do not support colormaps with more than 256 entries.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the values for `fmt`, listed in "Supported Formats" on page 2-1676.

`imwrite(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the output file for HDF, JPEG, PBM, PGM, PNG, PPM, and TIFF files. For example, if you are writing a JPEG file, you can specify the quality of the output image. For the

lists of parameters available for each format, see “Format-Specific Parameters” on page 2-1678.

Supported Formats

This table summarizes the types of images that `imwrite` can write. The MATLAB file format registry determines which file formats are supported. See `imformats` for more information about this registry. Note that, for certain formats, `imwrite` may take additional parameters, described in “Format-Specific Parameters” on page 2-1678.

Format	Full Name	Variants
`'bmp'`	Windows Bitmap (BMP)	1-bit, 8-bit, and 24-bit uncompressed images
`'gif'`	Graphics Interchange Format (GIF)	8-bit images
`'hdf'`	Hierarchical Data Format (HDF4)	8-bit raster image data sets, with or without associated colormap, 24-bit raster image data sets; uncompressed or with RLE or JPEG compression
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)	8-bit, 12-bit, and 16-bit Baseline JPEG images **Note** Indexed images are converted to RGB before writing out JPEG files, because the JPEG format does not support indexed images.
`pbm`	Portable Bitmap (PBM)	Any 1-bit PBM image, ASCII (plain) or raw (binary) encoding

Format	Full Name	Variants
'pcx'	Windows Paintbrush (PCX)	8-bit images
'pgm'	Portable Graymap (PGM)	Any standard PGM image; ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per gray value
'png'	Portable Network Graphics (PNG)	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit truecolor images; 24-bit and 48-bit truecolor images with alpha channels
'pnm'	Portable Anymap (PNM)	Any of the PPM/PGM/PBM formats, chosen automatically
'ppm'	Portable Pixmap (PPM)	Any standard PPM image. ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per color component
'ras'	Sun Raster (RAS)	Any RAS image, including 1-bit bitmap, 8-bit indexed, 24-bit truecolor and 32-bit truecolor with alpha

Format	Full Name	Variants
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF)	Baseline TIFF images, including 1-bit, 8-bit, 16-bit, and 24-bit uncompressed images; 1-bit, 8-bit, 16-bit, and 24-bit images with packbits compression; 1-bit images with CCITT 1D, Group 3, and Group 4 compression; CIELAB, ICCLAB, and CMYK images
`'xwd'`	X Windows Dump (XWD)	8-bit ZPixmaps

Format-Specific Parameters

The following tables list parameters that can be used with specific file formats.

GIF-Specific Parameters

This table describes the available parameters for GIF files.

Parameter	Values
`'BackgroundColor'`	A scalar integer. This value specifies which index in the colormap should be treated as the transparent color for the image and is used for certain disposal methods in animated GIFs. If `X` is `uint8` or `logical`, then indexing starts at `0`. If `X` is `double`, then indexing starts at `1`.
`'Comment'`	A string or cell array of strings containing a comment to be added to the image. For a cell array of strings, a carriage return is added after each row.
`'DelayTime'`	A scalar value between 0 and 655 inclusive, that specifies the delay in seconds before displaying the next image.
`'DisposalMethod'`	One of the following strings, which sets the disposal method of an animated GIF: `'leaveInPlace'`, `'restoreBG'`, `'restorePrevious'`, or `'doNotSpecify'`.

Parameter	Values
`'LoopCount'`	A finite integer between 0 and 65535 or the value `Inf` (the default) which specifies the number of times to repeat the animation. By default, the animation loops continuously. For a value of 0, the animation will be played once. For a value of 1, the animation will be played twice, etc.
`'TransparentColor'`	A scalar integer. This value specifies which index in the colormap should be treated as the transparent color for the image. If X is `uint8` or `logical`, then indexing starts at `0`. If X is double, then indexing starts at `1`.
`'WriteMode'`	One of these strings: `'overwrite'` (the default) or `'append'`. In append mode, a single frame is added to the existing file.

HDF4-Specific Parameters

This table describes the available parameters for HDF4 files.

Parameter	Values
`'Compression'`	One of these strings: `'none'` (the default) `'jpeg'`(valid only for grayscale and RGB images) `'rle'` (valid only for grayscale and indexed images)

Parameter	Values
'Quality'	A number between 0 and 100; this parameter applies only if 'Compression' is 'jpeg'. Higher numbers mean higher *quality* (less image degradation due to compression), but the resulting file size is larger. The default value is 75.
'WriteMode'	One of these strings: 'overwrite' (the default) 'append'

JPEG-Specific Parameters

This table describes the available parameters for JPEG files.

Parameter	Values	Default
'Bitdepth'	A scalar value indicating desired bitdepth; for grayscale images this can be 8, 12, or 16; for color images this can be 8 or 12.	8 (grayscale) and 8 bit per plane for color images
'Comment'	A column vector cell array of strings or a character matrix. Each row of input is written out as a comment in the JPEG file.	Empty
'Mode'	Specifies the type of compression used; value can be either of these strings: 'lossy' or 'lossless'	'lossy'
'Quality'	A number between 0 and 100; higher numbers mean higher quality (less image degradation due to compression), but the resulting file size is larger.	75

PBM-, PGM-, and PPM-Specific Parameters

This table describes the available parameters for PBM, PGM, and PPM files.

Parameter	Values	Default
`'Encoding'`	One of these strings: `'ASCII'` for plain encoding `'rawbits'` for binary encoding	`'rawbits'`
`'MaxValue'`	A scalar indicating the maximum gray or color value. Available only for PGM and PPM files. For PBM files, this value is always `1`.	Default is `65535` if image array is `'uint16'`; `255` otherwise.

PNG-Specific Parameters

The following table lists the available parameters for PNG files, in alphabetical order. In addition to these PNG parameters, you can use any parameter name that satisfies the PNG specification for keywords; that is, uses only printable characters, contains 80 or fewer characters, and no contains no leading or trailing spaces. The value corresponding to these user-specified parameters must be a string that contains no control characters other than linefeed.

Parameter	Values
`'Alpha'`	A matrix specifying the transparency of each pixel individually. The row and column dimensions must be the same as the data array; they can be `uint8`, `uint16`, or `double`, in which case the values should be in the range [0,1].
`'Author'`	A string
`'Background'`	The value specifies background color to be used when compositing transparent pixels. For indexed images: an integer in the range [1,`P`], where `P` is the colormap length. For grayscale images: a scalar in the range [0,1]. For truecolor images: a three-element vector in the range [0,1].
`'bitdepth'`	A scalar value indicating desired bit depth. For grayscale images this can be 1, 2, 4, 8, or 16. For grayscale images with an alpha channel this can be 8 or 16. For indexed images this can be 1, 2, 4, or 8. For truecolor images with or without an alpha channel this can be 8 or 16. By default, `imwrite` uses 8 bits per pixel, if image is double or `uint8`; 16 bits per pixel if image is `uint16`; 1 bit per pixel if image is `logical`.

Parameter	Values
'Chromaticities'	An eight-element vector [wx wy rx ry gx gy bx by] that specifies the reference white point and the primary chromaticities
'Comment'	A string
'Copyright'	A string
'CreationTime'	A string
'Description'	A string
'Disclaimer'	A string
'Gamma'	A nonnegative scalar indicating the file gamma
'ImageModTime'	A MATLAB serial date number (see the datenum function) or a string convertible to a date vector via the datevec function. Values should be in Coordinated Universal Time (UTC).
'InterlaceType'	Either 'none' (the default) or 'adam7'
'ResolutionUnit'	Either 'unknown' or 'meter'
'SignificantBits'	A scalar or vector indicating how many bits in the data array should be regarded as significant; values must be in the range [1,BitDepth]. For indexed images: a three-element vector. For grayscale images: a scalar. For grayscale images with an alpha channel: a two-element vector. For truecolor images: a three-element vector. For truecolor images with an alpha channel: a four-element vector.
'Software'	A string
'Source'	A string

Parameter	Values
`'Transparency'`	This value is used to indicate transparency information only when no alpha channel is used. Set to the value that indicates which pixels should be considered transparent. (If the image uses a colormap, this value represents an index number to the colormap.) For indexed images: a `Q`-element vector in the range [0,1], where `Q` is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, `Q = 1`. For grayscale images: a scalar in the range [0,1]. The value indicates the grayscale color to be considered transparent. For truecolor images: a three-element vector in the range [0,1]. The value indicates the truecolor color to be considered transparent. **Note** You cannot specify `'Transparency'` and `'Alpha'` at the same time.
`'Warning'`	A string
`'XResolution'`	A scalar indicating the number of pixels/unit in the horizontal direction
`'YResolution'`	A scalar indicating the number of pixels/unit in the vertical direction

RAS-Specific Parameters

This table describes the available parameters for RAS files.

Parameter	Values	Default
`'Alpha'`	A matrix specifying the transparency of each pixel individually; the row and column dimensions must be the same as the data array; can be `uint8`, `uint16`, or `double`. Can only be used with truecolor images.	Empty matrix (`[]`)
`'Type'`	One of these strings: `'standard'` (uncompressed, b-g-r color order with truecolor images) `'rgb'` (like `'standard'`, but uses r-g-b color order for truecolor images) `'rle'` (run-length encoding of 1-bit and 8-bit images)	`'standard'`

TIFF-Specific Parameters

This table describes the available parameters for TIFF files.

Parameter	Values	Default
`'ColorSpace'`	Specifies one of the following color spaces used to represent the color data. `'rgb'` `'cielab'` `'icclab'` See for more information about this parameter.	`'rgb'`
`'Compression'`	One of these strings: `'none'`, `'packbits'`, `'ccitt'`, `'fax3'`, or `'fax4'` The `'ccitt'`, `'fax3'`, and `'fax4'` compression schemes are valid for binary images only.	`'ccitt'` for binary images; `'packbits'` for nonbinary images
`'Description'`	Any string; fills in the `ImageDescription` field returned by `imfinfo`	Empty

Parameter	Values	Default
'Resolution'	A two-element vector containing the XResolution and YResolution, or a scalar indicating both resolutions	72
'WriteMode'	One of these strings: 'overwrite' 'append'	'overwrite'

L*a*b* Color Data

For TIFF files only, imwrite can write a color image that uses the $L^*a^*b^*$ color space. The 1976 CIE $L^*a^*b^*$ specification defines numeric values that represent luminance (L^*) and chrominance (a^* and b^*) information.

To store $L^*a^*b^*$ color data in a TIFF file, the values must be encoded to fit into either 8-bit or 16-bit storage. imwrite can store $L^*a^*b^*$ color data in a TIFF file using these encodings:

- 8-bit and 16-bit encodings defined by the TIFF specification, called the CIELAB encodings
- 8-bit and 16-bit encodings defined by the International Color Consortium, called ICCLAB encodings

The output class and encoding used by imwrite to store color data depends on the class of the input array and the value you specify for the TIFF-specific ColorSpace parameter. The following table explains these options. (The 8-bit and 16-bit CIELAB encodings cannot be input arrays because they use a mixture of signed and unsigned values and cannot be represented as a single MATLAB array.)

Input Class and Encoding	ColorSpace Parameter Value	Output Class and Encoding
8-bit ICCLAB [1]	'icclab'	8-bit ICCLAB
	'cielab'	8-bit CIELAB
16-bit ICCLAB [2]	'icclab'	16-bit ICCLAB

Input Class and Encoding	ColorSpace Parameter Value	Output Class and Encoding
	'cielab'	16-bit CIELAB
Double-precision 1976 CIE $L^*a^*b^*$ values [3]	'icclab'	8-bit ICCLAB
	'cielab'	8-bit CIELAB

[1] 8-bit ICCLAB represents values as integers in the range [0 255]. L^* values are multiplied by `255/100`; `128` is added to both the a^* and b^* values.

[2] 16-bit ICCLAB multiplies L^* values by `65280/100` and represents the values as integers in the range [0, 65280]. `32768` is added to both the a^* and b^* values, which are represented as integers in the range [0,65535].

[3] L^* is in the dynamic range [0, 100]. a^* and b^* can take any value. Setting a* and b^* to 0 (zero) produces a neutral color (gray).

Class Support

The input array `A` can be of class `logical`, `uint8`, `uint16`, or `double`. Indexed images (`X`) can be of class `uint8`, `uint16`, or `double`; the associated colormap, `map`, must be of class `double`. Input values must be full (non-sparse).

The class of the image written to the file depends on the format specified. For most formats, if the input array is of class `uint8`, `imwrite` outputs the data as 8-bit values. If the input array is of class `uint16` and the format supports 16-bit data (JPEG, PNG, and TIFF), `imwrite` outputs the data as 16-bit values. If the format does not support 16-bit values, `imwrite` issues an error. Several formats, such as JPEG and PNG, support a parameter that lets you specify the bit depth of the output data.

If the input array is of class `double`, and the image is a grayscale or RGB color image, `imwrite` assumes the dynamic range is [0,1] and

automatically scales the data by 255 before writing it to the file as 8-bit values.

If the input array is of class `double`, and the image is an indexed image, `imwrite` converts the indices to zero-based indices by subtracting 1 from each element, and then writes the data as `uint8`.

If the input array is of class `logical`, `imwrite` assumes the data is a binary image and writes it to the file with a bit depth of 1, if the format allows it. BMP, PNG, or TIFF formats accept binary images as input arrays.

Example

This example appends an indexed image `X` and its colormap `map` to an existing uncompressed multipage HDF4 file.

```
imwrite(X,map,'your_hdf_file.hdf','Compression','none',...
'WriteMode','append')
```

See Also

`fwrite`, `getframe`, `imfinfo`, `imformats`, `imread`

"Bit-Mapped Images" on page 1-92 for related functions

Purpose

Convert indexed image to RGB image

Syntax

```
RGB = ind2rgb(X,map)
```

Description

`RGB = ind2rgb(X,map)` converts the matrix `X` and corresponding colormap `map` to RGB (truecolor) format.

Class Support

`X` can be of class `uint8`, `uint16`, or `double`. RGB is an m-by-n-by-3 array of class `double`.

See Also

`image`

"Bit-Mapped Images" on page 1-92 for related functions

ind2sub

Purpose

Subscripts from linear index

Syntax

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

Description

The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

`[I,J] = ind2sub(siz,IND)` returns the matrices `I` and `J` containing the equivalent row and column subscripts corresponding to each linear index in the matrix `IND` for a matrix of size `siz`. `siz` is a 2-element vector, where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

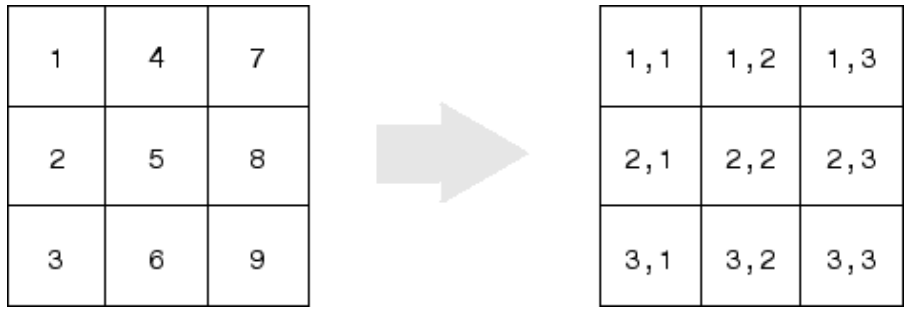
Note For matrices, `[I,J] = ind2sub(size(A),find(A>5))` returns the same values as `[I,J] = find(A>5)`.

`[I1,I2,I3,...,In] = ind2sub(siz,IND)` returns `n` subscript arrays `I1,I2,...,In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

Examples

Example 1 — Two-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 3-by-3 matrix is

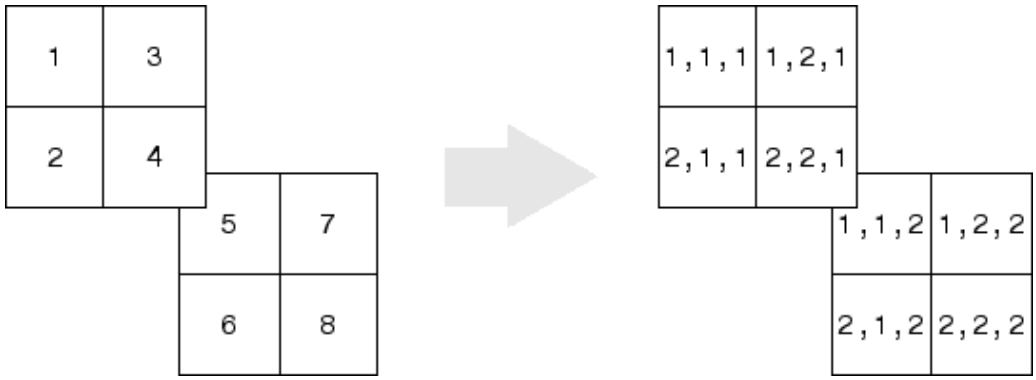


This code determines the row and column subscripts in a 3-by-3 matrix, of elements with linear indices 3, 4, 5, 6.

```
IND = [3 4 5 6]
s = [3,3];
[I,J] = ind2sub(s,IND)

I =
     3     1     2     3

J =
     1     2     2     2
```

Example 2 — Three-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is



This code determines the subscript equivalents in a 2-by-2-by-2 array, of elements whose linear indices 3, 4, 5, 6 are specified in the `IND` matrix.

```
IND = [3 4;5 6];
s = [2,2,2];
[I,J,K] = ind2sub(s,IND)

I =
     1     2
     1     2

J =
     2     2
     1     1

K =
     1     1
     2     2
```

Example 3 — Effects of Returning Fewer Outputs

When calling `ind2sub` for an N-dimensional matrix, you would typically supply N output arguments in the call: one for each dimension of the matrix. This example shows what happens when you return three, two, and one output when calling `ind2sub` on a 3-dimensional matrix.

The matrix is 2-by-2-by-2 and the linear indices are 1 through 8:

```
dims = [2 2 2];
indices = [1 2 3 4 5 6 7 8];
```

The 3-output call to `ind2sub` returns the expected subscripts for the 2-by-2-by-2 matrix:

```
[rowsub colsub pagsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     1     1     2     2
pagsub =
     1     1     1     1     2     2     2     2
```

If you specify only two outputs (row and column), `ind2sub` still returns a subscript for each specified index, but drops the third dimension from the matrix, returning subscripts for a 2-dimensional, 2-by-4 matrix instead:

```
[rowsub colsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     3     3     4     4
```

If you specify one output (row), `ind2sub` drops both the second and third dimensions from the matrix, and returns subscripts for a 1-dimensional, 1-by-8 matrix instead:

```
[rowsub] = ind2sub(dims, indices)
rowsub =
     1     2     3     4     5     6     7     8
```

See Also

`find`, `size`, `sub2ind`

Inf

Purpose

Infinity

Syntax

```
Inf
Inf('double')
Inf('single')
Inf(n)
Inf(m,n)
Inf(m,n,p,...)
Inf(...,classname)
```

Description

`Inf` returns the IEEE arithmetic representation for positive infinity. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

`Inf('double')` is the same as `Inf` with no inputs.

`Inf('single')` is the single precision representation of `Inf`.

`Inf(n)` is an `n`-by-`n` matrix of `Inf`s.

`Inf(m,n)` or `inf([m,n])` is an `m`-by-`n` matrix of `Inf`s.

`Inf(m,n,p,...)` or `Inf([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `Inf`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`Inf(...,classname)` is an array of `Inf`s of class specified by `classname`. `classname` must be either `'single'` or `'double'`.

Examples

`1/0`, `1.e1000`, `2^2000`, and `exp(1000)` all produce `Inf`.

`log(0)` produces `-Inf`.

`Inf-Inf` and `Inf/Inf` both produce `NaN` (Not-a-Number).

See Also

`isinf`, `NaN`

inferiorto

Purpose

Establish inferior class relationship

Syntax

```
inferiorto('class1', 'class2', ...)
```

Description

The `inferiorto` function establishes a hierarchy that determines the order in which MATLAB calls object methods.

`inferiorto('class1', 'class2', ...)` invoked within a class constructor method (say `myclass.m`) indicates that `myclass`'s method should not be invoked if a function is called with an object of class `myclass` and one or more objects of class `class1`, `class2`, and so on.

Remarks

Suppose `A` is of class `'class_a'`, `B` is of class `'class_b'` and `C` is of class `'class_c'`. Also suppose the constructor `class_c.m` contains the statement `inferiorto('class_a')`. Then `e = fun(a, c)` or `e = fun(c, a)` invokes `class_a/fun`.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So `fun(b, c)` calls `class_b/fun`, while `fun(c, b)` calls `class_c/fun`.

See Also

`superiorto`

Purpose

Information about contacting The MathWorks

Syntax

```
info
```

Description

`info` displays in the Command Window, information about contacting The MathWorks.

See Also

`help`, `version`

inline

Purpose

Construct inline object

Syntax

```
inline(expr)
inline(expr,arg1,arg2,...)
inline(expr,n)
```

Description

`inline(expr)` constructs an inline function object from the MATLAB expression contained in the string `expr`. The input argument to the inline function is automatically determined by searching `expr` for an isolated lower case alphabetic character, other than `i` or `j`, that is not part of a word formed from several alphabetic characters. If no such character exists, `x` is used. If the character is not unique, the one closest to `x` is used. If two characters are found, the one later in the alphabet is chosen.

`inline(expr,arg1,arg2,...)` constructs an inline function whose input arguments are specified by the strings `arg1`, `arg2`,.... Multicharacter symbol names may be used.

`inline(expr,n)` where `n` is a scalar, constructs an inline function whose input arguments are `x`, `P1`, `P2`, ...

Remarks

Three commands related to `inline` allow you to examine an inline function object and determine how it was created.

`char(fun)` converts the inline function into a character array. This is identical to `formula(fun)`.

`argnames(fun)` returns the names of the input arguments of the inline object `fun` as a cell array of strings.

`formula(fun)` returns the formula for the inline object `fun`.

A fourth command `vectorize(fun)` inserts a `.` before any `^`, `*` or `/` in the formula for `fun`. The result is a vectorized version of the inline function.

Examples

Example 1

This example creates a simple inline function to square a number.

```
g = inline('t^2')
g =

     Inline function:
     g(t) = t^2
```

You can convert the result to a string using the `char` function.

```
char(g)

ans =

t^2
```

Example 2

This example creates an inline function to represent the formula $f = 3\sin(2x^2)$. The resulting inline function can be evaluated with the `argnames` and `formula` functions.

```
f = inline('3*sin(2*x.^2)')

f =
     Inline function:
     f(x) = 3*sin(2*x.^2)

argnames(f)

ans =
    'x'

formula(f)
ans =

3*sin(2*x.^2)
```

Example 3

This call to `inline` defines the function `f` to be dependent on two variables, `alpha` and `x`:

```
f = inline('sin(alpha*x)')

f =
     Inline function:
     f(alpha,x) = sin(alpha*x)
```

If `inline` does not return the desired function variables or if the function variables are in the wrong order, you can specify the desired variables explicitly with the `inline` argument list.

```
g = inline('sin(alpha*x)','x','alpha')

g =

     Inline function:
     g(x,alpha) = sin(alpha*x)
```

Purpose

Names of M-files, MEX-files, Java classes in memory

Syntax

```
M = inmem
[M, X] = inmem
[M, X, J] = inmem
[...] = inmem('-completenames')
```

Description

`M = inmem` returns a cell array of strings containing the names of the M-files that are currently loaded.

`[M, X] = inmem` returns an additional cell array `X` containing the names of the MEX-files that are currently loaded.

`[M, X, J] = inmem` also returns a cell array `J` containing the names of the Java classes that are currently loaded.

`[...] = inmem('-completenames')` returns not only the names of the currently loaded M- and MEX-files, but the path and filename extension for each as well. No additional information is returned for loaded Java classes.

Examples

Example 1

This example lists the M-files that are required to run `erf`.

```
clear all;        % Clear the workspace
erf(0.5);

M = inmem
M =
   'erf'
```

Example 2

Generate a plot, and then find the M- and MEX-files that had been loaded to perform this operation:

```
clear all
surf(peaks)
```

```
[m x] = inmem('-completenames');

m(1:5)
ans =
    'F:\matlab\toolbox\matlab\ops\ismember.m'
    'F:\matlab\toolbox\matlab\datatypes\@opaque\double.m'
    'F:\matlab\toolbox\matlab\datatypes\isfield.m'
    'F:\matlab\toolbox\matlab\graphics\gcf.m'
    'F:\matlab\toolbox\matlab\elmat\meshgrid.m'

x(1:end)
ans =
    'F:\matlab\toolbox\matlab\graph2d\private\lineseriesmex.dll'
```

See Also

clear

Purpose

Points inside polygonal region

Syntax

```
IN = inpolygon(X,Y,xv,yv)
[IN ON] = inpolygon(X,Y,xv,yv)
```

Description

`IN = inpolygon(X,Y,xv,yv)` returns a matrix `IN` the same size as `X` and `Y`. Each element of `IN` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is inside the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

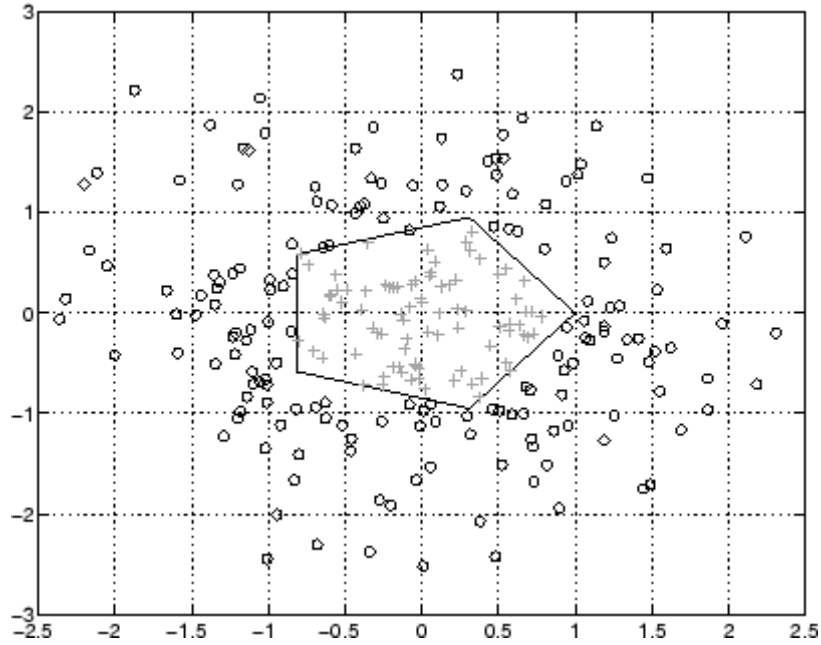
`IN(p,q) = 1`	If `(X(p,q),Y(p,q))` is inside the polygonal region or on the polygon boundary
`IN(p,q) = 0`	If `(X(p,q),Y(p,q))` is outside the polygonal region

`[IN ON] = inpolygon(X,Y,xv,yv)` returns a second matrix `ON` the same size as `X` and `Y`. Each element of `ON` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is on the boundary of the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

`ON(p,q) = 1`	If `(X(p,q),Y(p,q))` is on the polygon boundary
`ON(p,q) = 0`	If `(X(p,q),Y(p,q))` is inside or outside the polygon boundary

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
x = randn(250,1); y = randn(250,1);
in = inpolygon(x,y,xv,yv);
plot(xv,yv,x(in),y(in),'r+',x(~in),y(~in),'bo')
```


3
2
1
0
−1
−2
−3
−2.5
−2
−1.5
−1
−0.5
0
0.5
1
1.5
2
2.5

Purpose

Request user input

Syntax

```
user_entry = input('prompt')
user_entry = input('prompt', 's')
```

Description

The response to the `input` prompt can be any MATLAB expression, which is evaluated using the variables in the current workspace.

`user_entry = input('prompt')` displays *`prompt`* as a prompt on the screen, waits for input from the keyboard, and returns the value entered in `user_entry`.

`user_entry = input('prompt', 's')` returns the entered string as a text variable rather than as a variable name or numerical value.

Remarks

If you press the **Return** key without entering anything, `input` returns an empty matrix.

The text string for the prompt can contain one or more `'\n'` characters. The `'\n'` means to skip to the next line. This allows the prompt string to span several lines. To display just a backslash, use `'\\'`.

If you enter an invalid expression at the prompt, MATLAB displays the relevant error message and then prompts you again to enter input.

Examples

Press **Return** to select a default value by detecting an empty matrix:

```
reply = input('Do you want more? Y/N [Y]: ', 's');
if isempty(reply)
    reply = 'Y';
end
```

See Also

`keyboard`, `menu`, `ginput`, `uicontrol`

inputdlg

Purpose

Create and open input dialog box

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,dlg_title)
answer = inputdlg(prompt,dlg_title,num_lines)
answer = inputdlg(prompt,dlg_title,num_lines,defAns)
answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)
```

Description

`answer = inputdlg(prompt)` creates a modal dialog box and returns user input for multiple prompts in the cell array. `prompt` is a cell array containing prompt strings.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

Note `inputdlg` uses the `uiwait` function to suspend execution until the user responds.

`answer = inputdlg(prompt,dlg_title)` `dlg_title` specifies a title for the dialog box.

`answer = inputdlg(prompt,dlg_title,num_lines)` `num_lines` specifies the number of lines for each user-entered value. `num_lines` can be a scalar, column vector, or matrix.

- If `num_lines` is a scalar, it applies to all prompts.
- If `num_lines` is a column vector, each element specifies the number of lines of input for a prompt.
- If `num_lines` is a matrix, it should be size m-by-2, where m is the number of prompts on the dialog box. Each row refers to a prompt.

The first column specifies the number of lines of input for a prompt. The second column specifies the width of the field in characters.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns)` `defAns` specifies the default value to display for each prompt. `defAns` must contain the same number of elements as `prompt` and all elements must be strings.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)` If `options` is the string `'on'`, the dialog is made resizable in the horizontal direction. If options is a structure, the fields shown in the following table are recognized:

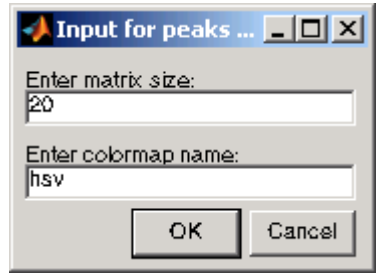
Field	Description
`Resize`	Can be `'on'` or `'off'` (default). If `'on'`, the window is resizable horizontally.
`WindowStyle`	Can be either `'normal'` or `'modal'` (default).
`Interpreter`	Can be either `'none'` (default) or `'tex'`. If the value is `'tex'`, the prompt strings are rendered using LaTeX.

Example

Example 1

Create a dialog box to input an integer and colormap name. Allow one line for each value.

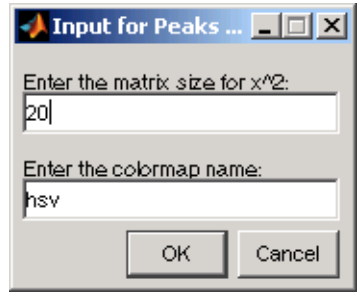
```
prompt = {'Enter matrix size:','Enter colormap name:'};
dlg_title = 'Input for peaks function';
num_lines = 1;
def = {'20','hsv'};
answer = inputdlg(prompt,dlg_title,num_lines,def);
```



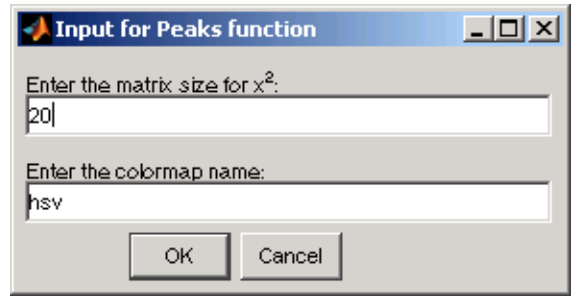
Example 2

Create a dialog box using the default options. Then use the options to make it resizable and not modal, and to interpret the text using LaTeX.

```
prompt={'Enter the matrix size for x^2:',...
        'Enter the colormap name:'};
name='Input for Peaks function';
numlines=1;
defaultanswer={'20','hsv'};
answer=inputdlg(prompt,name,numlines,defaultanswer);
```




```
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';

answer=inputdlg(prompt,name,numlines,defaultanswer,options);
```



See Also

`dialog`, `errordlg`, `helpdlg`, `listdlg`, `msgbox`, `questdlg`, `warndlg`

`figure`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-104 for related functions

inputname

Purpose

Variable name of function input

Syntax

```
inputname(argnum)
```

Description

This command can be used only inside the body of a function.

`inputname(argnum)` returns the workspace variable name corresponding to the argument number *`argnum`*. If the input argument has no name (for example, if it is an expression instead of a variable), the `inputname` command returns the empty string (`''`).

Examples

Suppose the function `myfun.m` is defined as

```
function c = myfun(a,b)
disp(sprintf('First calling variable is "%s".', inputname(1))
```

Then

```
x = 5;  y = 3;  myfun(x,y)
```

produces

```
First calling variable is "x".
```

But

```
myfun(pi+1, pi-1)
```

produces

```
First calling variable is "".
```

See Also

`nargin`, `nargout`, `nargchk`

Purpose

Construct input parser object

Syntax

```
p = inputParser
```

Description

`p = inputParser` constructs an empty `inputParser` object. Use this utility object to parse and validate input arguments to the functions that you develop. The input parser object follows handle semantics; that is, methods called on it affect the original object, not a copy of it.

MATLAB configures `inputParser` objects to recognize an input schema. Use any of the following methods to create the schema for parsing a particular function.

For more information on the `inputParser` class, see "Parsing Inputs with inputParser" in the MATLAB Programming documentation.

Methods

Method	Description
`addOptional`	Add an optional argument to the schema
`addParamValue`	Add a parameter-value pair argument to the schema
`addRequired`	Add a required argument to the schema
`createCopy`	Create a copy of the `inputParser` object
`parse`	Parse and validate the named inputs

Properties

Property	Description
`CaseSensitivity`	Enable or disable case-sensitive matching of argument names
`FunctionName`	Function name to be included in error messages
`KeepUnmatched`	Enable or disable errors on unmatched arguments

Property	Description
`Parameters`	Names of arguments defined in `inputParser` schema
`Results`	Names and values of arguments passed in function call that are in the schema for this function
`StructExpand`	Enable or disable passing arguments in a structure
`Unmatched`	Names and values of arguments passed in function call that are not in the schema for this function
`UsingDefaults`	Names of arguments not passed in function call that are given default values

Property Descriptions

Properties of the `inputParser` class are described below.

CaseSensitivity

Purpose — Enable or disable case sensitive matching of argument names

`p.CaseSensitivity = TF` enables or disables case-sensitivity when matching entries in the argument list with argument names in the schema. Set `CaseSensitivity` to logical 1 (`true`) to enable case-sensitive matching, or to logical 0 (`false`) to disable it. By default, case-sensitive matching is disabled.

FunctionName

Purpose — Function name to be included in error messages

`p.FunctionName = name` stores a function `name` that is to be included in error messages that might be thrown in the process of validating input arguments to the function. The `name` input is a string containing the name of the function for which you are parsing inputs with `inputParser`.

KeepUnmatched

Purpose — Enable or disable errors on unmatched arguments

`p.KeepUnmatched = TF` controls whether MATLAB throws an error when the function being called is passed an argument that has not been defined in the `inputParser` schema for this file. When this property is set to logical 1 (`true`), MATLAB does not throw an error, but instead stores the names and values of unmatched arguments in the `Unmatched` property of object p. When `KeepUnmatched` is set to logical 0 (false), MATLAB does throw an error whenever this condition is encountered and the `Unmatched` property is not affected.

Parameters

Purpose — Names of arguments defined in `inputParser` schema

`c = p.Parameters` is a cell array of strings containing the names of those arguments currently defined in the schema for the object. Each row of the `Parameters` cell array is a string containing the full name of a known argument.

Results

Purpose — Names and values of arguments passed in function call that are in the schema for this function

`arglist = p.Results` is a structure containing the results of the most recent parse of the input argument list. Each argument passed to the function is represented by a field in the `Results` structure, and the value of that argument is represented by the value of that field.

StructExpand

Purpose — Enable or disable passing arguments in a structure

`p.StructExpand = TF`, when set to logical 1 (`true`), tells MATLAB to accept a structure as an input in place of individual parameter-value arguments. If `StructExpand` is set to logical 0 (`false`), a structure is treated as a regular, single input.

Unmatched

Purpose — Names and values of arguments passed in function call that are not in the schema for this function

`c = p.Unmatched` is a structure array containing the names and values of all arguments passed in a call to the function that are not included in the schema for the function. `Unmatched` only contains this list of the `KeepUnmatched` property is set to true. If `KeepUnmatched` is set to `false`, MATLAB throws an error when unmatched arguments are passed in the function call. The `Unmatched` structure has the same format as the `Results` property of the `inputParser` class.

UsingDefaults

Purpose — Names of arguments not passed in function call that are given default values

`defaults = p.UsingDefaults` is a cell array of strings containing the names of those arguments that were not passed in the call to this function and consequently are set to their default values.

Examples

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class. Construct an instance of `inputParser` and assign it to variable `p`:

```
function publish_ip(script, varargin)
p = inputParser;   % Create an instance of the inputParser class.
```

Add arguments to the schema. See the reference pages for the `addRequired`, `addOptional`, and `addParamValue` methods for help with this:

```
p.addRequired('script', @ischar);
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Call the `parse` method of the object to read and validate each argument in the schema:

```
p.parse(script, varargin{:});
```

Execution of the `parse` method validates each argument and also builds a structure from the input arguments. The name of the structure is `Results`, which is accessible as a property of the object. To get the value of any input argument, type

```
p.Results.argname
```

Continuing with the `publish_ip` exercise, add the following lines to your M-file:

```
% Parse and validate all input arguments.
p.parse(script, varargin{:});

% Display the value for maxHeight.
disp(sprintf('\nThe maximum height is %d.\n', p.Results.maxHeight))

% Display all arguments.
disp 'List of all arguments:'
disp(p.Results)
```

When you call the program, MATLAB assigns those values you pass in the argument list to the appropriate fields of the `Results` structure. Save the M-file and execute it at the MATLAB command prompt with this command:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', 'C:/matlab/test', ...
    'maxWidth', 500, 'maxHeight', 300);

The maximum height is 300.

List of all arguments:
              format: 'ppt'
           maxHeight: 300
```

```
                maxWidth: 500
               outputDir: 'C:/matlab/test'
                  script: 'ipscript.m'
```

See Also

addRequired(inputParser), addOptional(inputParser), addParamValue(inputParser), parse(inputParser), createCopy(inputParser), varargin, nargchk, nargin

Purpose

Open Property Inspector

Syntax

```
inspect
inspect(h)
inspect([h1,h2,...])
```

Description

`inspect` creates a separate Property Inspector window to enable the display and modification of the properties of any object you select in the figure window or Layout Editor. If no object is selected, the Property Inspector is blank.

`inspect(h)` creates a Property Inspector window for the object whose handle is `h`.

`inspect([h1,h2,...])` displays properties that objects `h1` and `h2` have in common, or a blank window if there are no such properties; any number of objects can be inspected and edited in this way (for example, handles returned by the `bar` command).

The Property Inspector has the following behaviors:

- Only one Property Inspector window is active at any given time; when you inspect a new object, its properties replace those of the object last inspected.
- When the Property Inspector is open and plot edit mode is on, clicking any object in the figure window displays the properties of that object (or set of objects) in the Property Inspector.
- When you select and inspect two or more objects of different types, the Property Inspector only shows the properties that all objects have in common.
- To change the value of any property, click on the property name shown at the left side of the window, and then enter the new value in the field at the right.

The Property Inspector provides two different views:

- List view — properties are ordered alphabetically (default); this is the only view available for annotation objects.
- Group view — properties are grouped under classified headings (Handle Graphics objects only)

To view alphabetically, click the "AZ" Icon  in the Property Inspector toolbar. To see properties in groups, click

the "++" icon  . When properties are grouped, the "-" and "+" icons are enabled; click  to expand all categories and click  to collapse all categories. You can also expand and collapse individual categories by clicking on the "+" next to the category name. Some properties expand and collapse

---

Notes To see a complete description of any property, right-click on its name or value and select **What's This**; a help window opens that displays the reference page entry for it.

The Property Inspector displays most, but not all, properties of Handle Graphics objects. For example, the `parent` and `children` of HG objects are not shown. `inspect h` displays a Property Inspector window that enables modification of the string `'h'`, not the object whose handle is `h`. If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

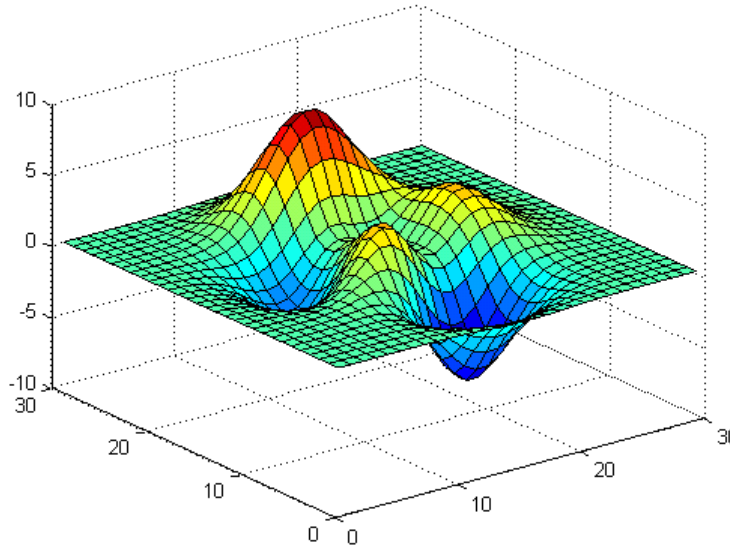
Examples

Example 1

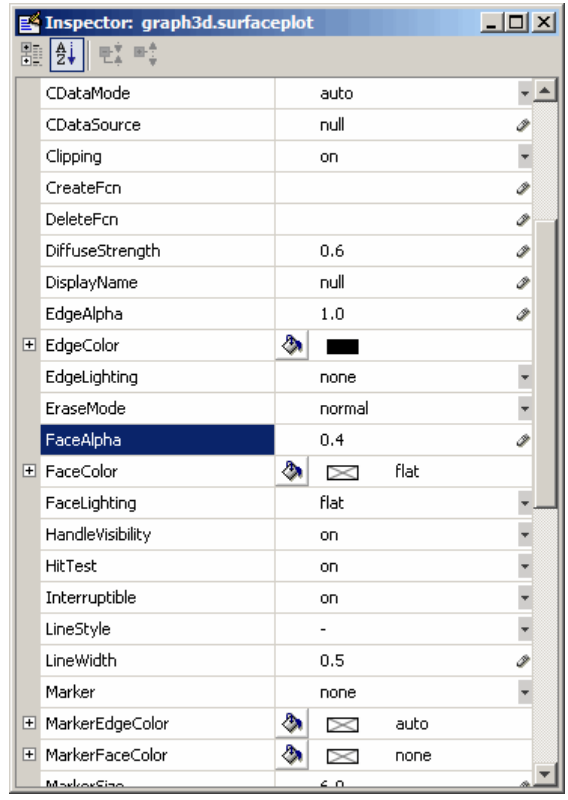
Create a surface mesh plot and view its properties with the Property Inspector:

```
Z = peaks(30);
h = surf(Z)
```

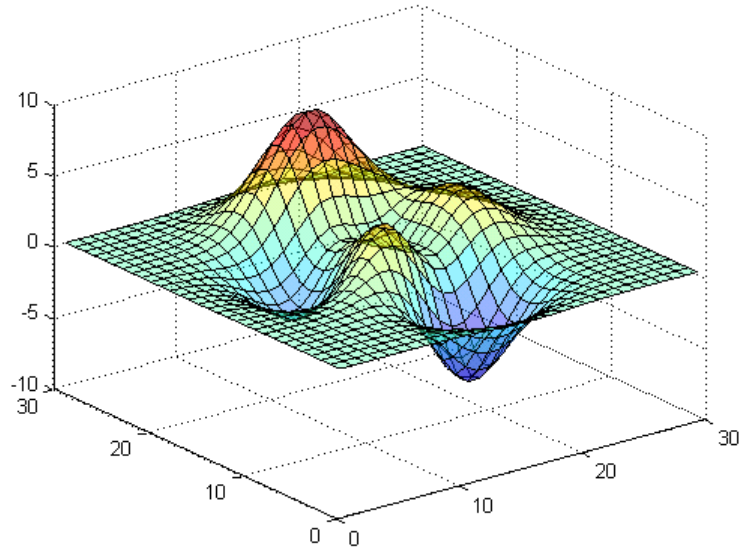
```
inspect(h)
```



Use the Property Inspector to change the `FaceAlpha` property from `1.0` to `0.4` (equivalent to the command `set(h,'FaceAlpha',0.4)`). `FaceAlpha` controls the tranparency of patch faces.



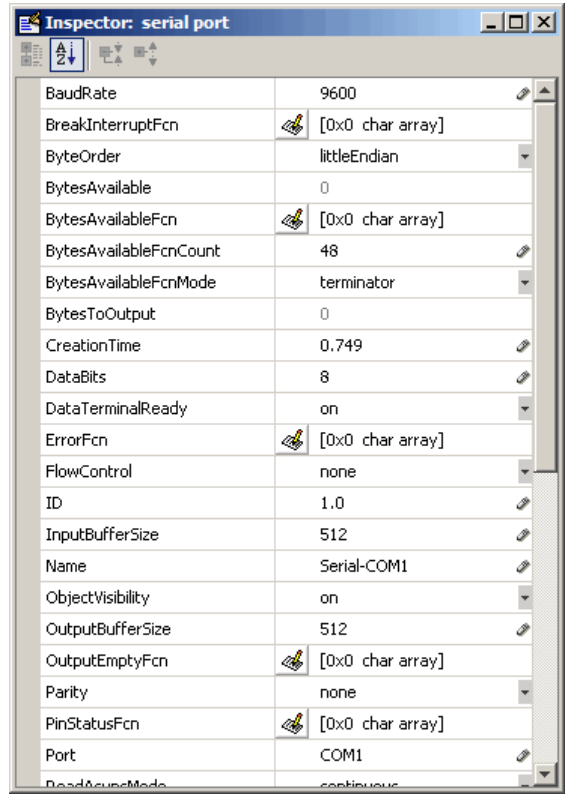
When you press **Enter** or click a different field, the `FaceAlpha` property of the surface object is updated:



Example 2

Create a serial port object for `COM1` and use the Property Inspector to peruse its properties:

```
s = serial('COM1');
inspect(s)
```



Because COM objects do not define property groupings, only the alphabetical list view of their properties is available.

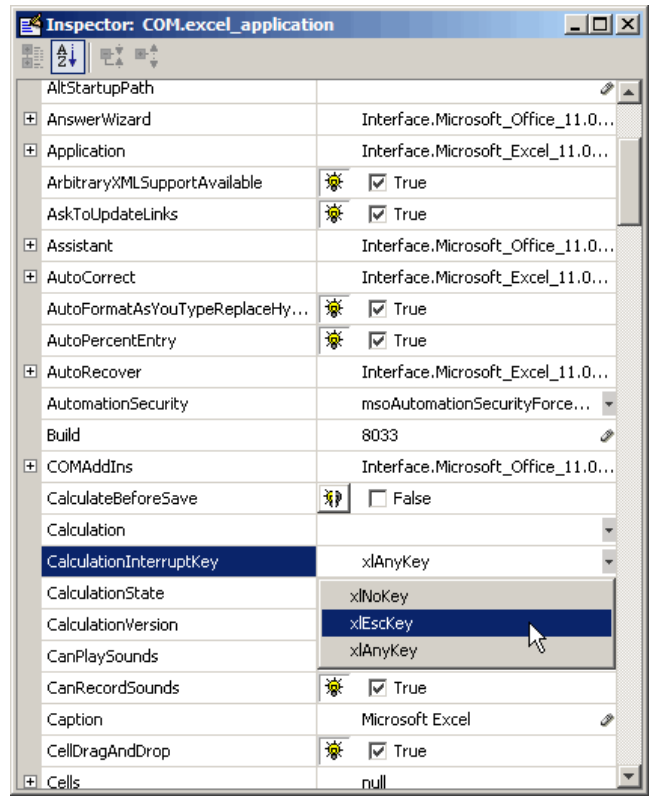
Example 3

Create a COM Excel server and open a Property Inspector window with `inspect`:

```
h = actxserver('excel.application');
inspect(h)
```

Scroll down until you see the `CalculationInterruptKey` property, which by default is `xlAnyKey`. Click on the down-arrow in the right

margin of the property inspector and select `xlEscKey` from the drop-down menu, as shown below:



Check this field in the MATLAB command window using `get` to confirm that it has changed:

```
get(h,'CalculationInterruptKey')

ans =
xlEscKey
```

See Also

`get`, `set`, `isprop`, `guide`, `addproperty`, `deleteproperty`

instrcallback

Purpose

Event information when event occurs

Syntax

```
instrcallback(obj,event)
```

Arguments

`obj`	An serial port object.
`event`	The event that caused the callback to execute.

Description

`instrcallback(obj,event)` displays a message that contains the event type, the time the event occurred, and the name of the serial port object that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.

Remarks

You should use `instrcallback` as a template from which you create callback functions that suit your specific application needs.

Example

The following example creates the serial port objects `s`, and configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

The resulting display from `instrcallback` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
```

```
delete(s)
clear s
```

instrfind

Purpose

Read serial port objects from memory to MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Arguments

`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *`PropertyName`*.
`S`	A structure of property names and property values.
`obj`	A serial port object, or an array of serial port objects.
`out`	An array of serial port objects.

Description

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('PropertyName',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'PropertyName',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

Remarks

Refer to "Displaying Property Names and Property Values" for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a

finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

Example

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'BaudRate',4800)
fopen([s1 s2])
```

You can use `instrfind` to return serial port objects based on property values.

```
out1 = instrfind('Port','COM1');
out2 = instrfind({'Port','BaudRate'},{'COM2',4800});
```

You can also use `instrfind` to return cleared serial port objects to the MATLAB workspace.

```
clear s1 s2
newobjs = instrfind

   Instrument Object Array
   Index:   Type:          Status:     Name:
   1        serial         open        Serial-COM1
   2        serial         open        Serial-COM2
```

To close both `s1` and `s2`

```
fclose(newobjs)
```

See Also

Functions

`clear`, `get`

instrfindall

Purpose

Find visible and hidden serial port objects

Syntax

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

Arguments

`'P1'`	Name of a serial port object property.
`V1`	Value allowed for corresponding *P1*.
`s`	A structure of property names and property values.
`objs`	An array of serial port objects.
`out`	An array of returned serial port objects.

Description

`out = instrfindall` finds all serial port objects, regardless of the value of the objects' `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'P1',V1,...)` restricts the search for objects with matching property name/value pairs to the serial port objects listed in `objs`.

Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

Remarks

`instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that returned by the `get` function. For example, if `get` returns the `Name` property value as `'MyObject'`, `instrfindall` will not find an object with a `Name` property value of `'myobject'`. However, this is not the case for properties that have a finite set of string values. For example, `instrfindall` will find an object with a `Parity` property value of `'Even'` or `'even'`.

Examples

Suppose you create the following serial port objects:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'ObjectVisibility','off')
```

Because object `s2` has its `ObjectVisibility` set to `'off'`, it is not visible to commands like `instrfind`:

```
instrfind

   Serial Port Object : Serial-COM1
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall

 Instrument Object Array
 Index:   Type:          Status:     Name:
 1        serial         closed      Serial-COM1
 2        serial         closed      Serial-COM2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress};
vals = {2,0};
```

```
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

See Also

Functions

`get`, `instrfind`

Properties

`ObjectVisibility`

Purpose

Convert integer to string

Syntax

```
str = int2str(N)
```

Description

`str = int2str(N)` converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.

Examples

`int2str(2+3)` is the string `'5'`.

One way to label a plot is

```
title(['case number ' int2str(n)])
```

For matrix or vector inputs, `int2str` returns a string matrix:

```
int2str(eye(3))

ans =

1  0  0
0  1  0
0  0  1
```

See Also

`fprintf`, `num2str`, `sprintf`

Purpose

Convert to signed integer

Syntax

```
I = int8(X)
I = int16(X)
I = int32(X)
I = int64(X)
```

Description

`I = int*(X)` converts the elements of array `X` into signed integers. `X` can be any numeric object (such as a `double`). The results of an `int*` operation are shown in the next table.

Operation	Output Range	Output Type	Bytes per Element	Output Class
`int8`	-128 to 127	Signed 8-bit integer	1	`int8`
`int16`	-32,768 to 32,767	Signed 16-bit integer	2	`int16`
`int32`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	4	`int32`
`int64`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	`int64`

`double` and `single` values are rounded to the nearest `int*` value on conversion. A value of `X` that is above or below the range for an integer class is mapped to one of the endpoints of the range. For example,

```
int16(40000)
ans =
   32767
```

If X is already a signed integer of the same class, then int* has no effect.

You can define or overload your own methods for int* (as you can for any object) by placing the appropriately named method in an @int* directory within a directory on your path. Type help datatypes for the names of the methods you can overload.

Remarks

Most operations that manipulate arrays without changing their elements are defined for integer values. Examples are reshape, size, the logical and relational operators, subscripted assignment, and subscripted reference.

Some arithmetic operations are defined for integer arrays on interaction with other integer arrays of the same class (e.g., where both operands are int16). Examples of these operations are +, -, .*, ./, .\ and .^. If at least one operand is scalar, then *, /, \, and ^ are also defined. Integer arrays may also interact with scalar double variables, including constants, and the result of the operation is an integer array of the same class. Integer arrays saturate on overflow in arithmetic.

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the zeros, ones, or eye function. For example, to create a 100-by-100 int64 array initialized to zero, type

```
I = zeros(100, 100, 'int64');
```

An easy way to find the range for any MATLAB integer type is to use the intmin and intmax functions as shown here for int32:

```
intmin('int32')              intmax('int32')
ans =                        ans =
   -2147483648                  2147483647
```

See Also

double, single, uint8, uint16, uint32, uint64, intmax, intmin

interfaces

Purpose

List custom interfaces to COM server

Syntax

```
C = h.interfaces
C = interfaces(h)
```

Description

`C = h.interfaces` returns cell array of strings `C` listing all custom interfaces implemented by the component in a specific COM server. The server is designated by input argument, `h`, which is the handle returned by the `actxcontrol` or `actxserver` function when creating that server.

`C = interfaces(h)` is an alternate syntax for the same operation.

Note `interfaces` only lists the custom interfaces; it does not return any interfaces. Use the `invoke` function to return a handle to a specific custom interface.

Examples

Once you have created a COM server, you can query the server component to see if any custom interfaces are implemented. Use the `interfaces` function to return a list of all available custom interfaces:

```
h = actxserver('mytestenv.calculator')
h =
   COM.mytestenv.calculator

customlist = h.interfaces
customlist =
   ICalc1
   ICalc2
   ICalc3
```

To get a handle to the custom interface you want, use the `invoke` function, specifying the handle returned by `actxcontrol` or `actxserver` and also the name of the custom interface:

```
c1 = h.invoke('ICalc1')
c1 =
```

```
Interface.Calc_1.0_Type_Library.ICalc_Interface
```

You can now use this handle with most of the COM client functions to access the properties and methods of the object through the selected custom interface. For example, to list the properties available through the `ICalc1` interface, use

```
c1.get
     background: 'Blue'
         height: 10
          width: 0
```

To list the methods, use

```
c1.invoke
   Add = double Add(handle, double, double)
   Divide = double Divide(handle, double, double)
   Multiply = double Multiply(handle, double, double)
   Subtract = double Subtract(handle, double, double)
```

Add and multiply numbers using the `Add` and `Multiply` methods of the custom object `c1`:

```
sum = c1.Add(4, 7)
sum =
    11

prod = c1.Multiply(4, 7)
prod =
    28
```

See Also

`actxcontrol`, `actxserver`, `invoke`, `get`

Purpose

1-D data interpolation (table lookup)

Syntax

```
yi = interp1(x,Y,xi)
yi = interp1(Y,xi)
yi = interp1(x,Y,xi,method)
yi = interp1(x,Y,xi,method,'extrap')
yi = interp1(x,Y,xi,method,extrapval)
pp = interp1(x,Y,method,'pp')
```

Description

`yi = interp1(x,Y,xi)` interpolates to find `yi`, the values of the underlying function `Y` at the points in the vector or array `xi`. `x` must be a vector. Y can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If `Y` is a scalar or vector, it must have the same length as `x`. A scalar value for `Y` is expanded to have the same length as `x`. `xi` can be a scalar, a vector, or a multidimensional array, and `yi` has the same size as `xi`.
- If `Y` is an array that is not a vector, the size of `Y` must have the form `[n,d1,d2,...,dk]`, where `n` is the length of `x`. The interpolation is performed for each `d1`-by-`d2`-by-...-`dk` value in `Y`. The sizes of `xi` and `yi` are related as follows:
  - If `xi` is a scalar or vector, `size(yi)` equals `[length(xi), d1, d2, ..., dk]`.
  - If `xi` is an array of size `[m1,m2,...,mj]`, `yi` has size `[m1,m2,...,mj,d1,d2,...,dk]`.

`yi = interp1(Y,xi)` assumes that `x = 1:N`, where `N` is the length of `Y` for vector `Y`, or `size(Y,1)` for matrix `Y`.

`yi = interp1(x,Y,xi,method)` interpolates using alternative methods:

`'nearest'`	Nearest neighbor interpolation
`'linear'`	Linear interpolation (default)

'spline'	Cubic spline interpolation
'pchip'	Piecewise cubic Hermite interpolation
'cubic'	(Same as 'pchip')
'v5cubic'	Cubic interpolation used in MATLAB 5. This method does not extrapolate. Also, if x is not equally spaced, 'spline' is used/

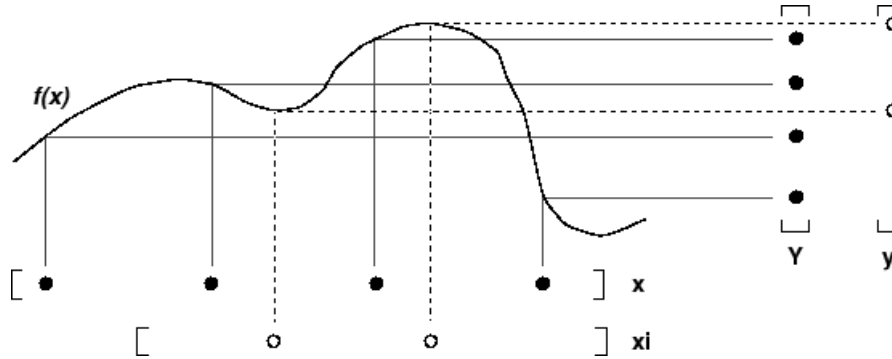
For the 'nearest', 'linear', and 'v5cubic' methods, interp1(x,Y,xi,method) returns NaN for any element of xi that is outside the interval spanned by x. For all other methods, interp1 performs extrapolation for out of range values.

yi = interp1(x,Y,xi,method,'extrap') uses the specified method to perform extrapolation for out of range values.

yi = interp1(x,Y,xi,method,extrapval) returns the scalar extrapval for out of range values. NaN and 0 are often used for extrapval.

pp = interp1(x,Y,method,'pp') uses the specified method to generate the piecewise polynomial form (ppform) of Y. You can use any of the methods in the preceding table, except for 'v5cubic'. pp can then be evaluated via ppval. ppval(pp,xi) is the same as interp1(x,Y,xi,method,'extrap').

The interp1 command interpolates between data points. It finds values at intermediate points, of a one-dimensional function $f(x)$ that underlies the data. This function is shown below, along with the relationship between vectors x, Y, xi, and yi.



Interpolation is the same operation as *table lookup*. Described in table lookup terms, the *table* is `[x,Y]` and `interp1` *looks up* the elements of `xi` in `x`, and, based upon their locations, returns values `yi` interpolated within the elements of `Y`.

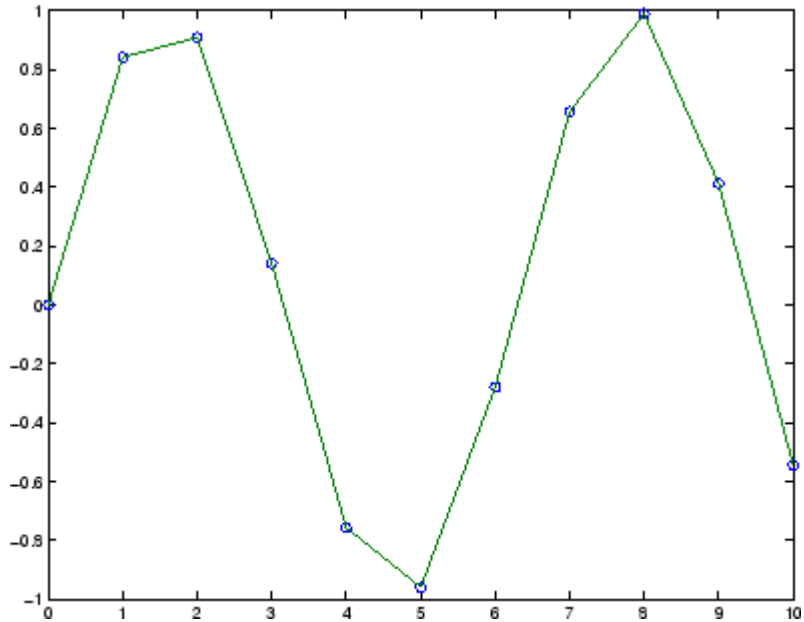
Note `interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking. For `interp1q` to work properly, `x` must be a monotonically increasing column vector and `Y` must be a column vector or matrix with `length(X)` rows. Type `help interp1q` at the command line for more information.

Examples

Example 1

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = 0:10;
y = sin(x);
xi = 0:.25:10;
yi = interp1(x,y,xi);
plot(x,y,'o',xi,yi)
```



Example 2

The following multidimensional example creates 2-by-2 matrices of interpolated function values, one matrix for each of the three functions x^2, x^3, and x^4.

```
x = [1:10]'; y = [ x.^2, x.^3, x.^4 ];
xi = [1.5, 1.75; 7.5, 7.75];
yi = interp1(x,y,xi);
```

The result `yi` has size 2-by-2-by-3.

```
size(yi)

ans =

     2     2     3
```

Example 3

Here are two vectors representing the census years from 1900 to 1990 and the corresponding United States population in millions of people.

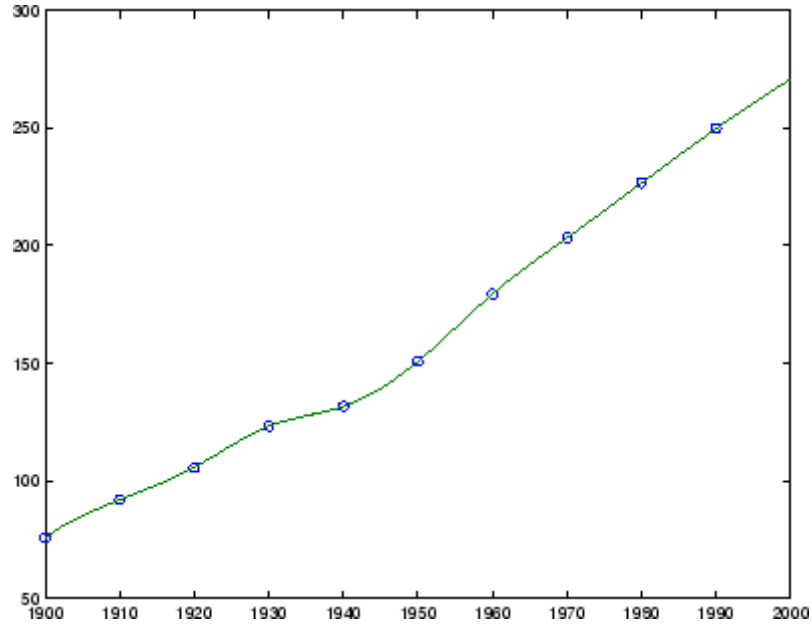
```
t = 1900:10:1990;
p = [75.995  91.972  105.711  123.203  131.669...
     150.697  179.323  203.212  226.505  249.633];
```

The expression `interp1(t,p,1975)` interpolates within the census data to estimate the population in 1975. The result is

```
ans =
    214.8585
```

Now interpolate within the data at every year from 1900 to 2000, and plot the result.

```
x = 1900:1:2000;
y = interp1(t,p,x,'spline');
plot(t,p,'o',x,y)
```



Sometimes it is more convenient to think of interpolation in table lookup terms, where the data are stored in a single table. If a portion of the census data is stored in a single 5-by-2 table,

```
tab =
    1950    150.697
    1960    179.323
    1970    203.212
    1980    226.505
    1990    249.633
```

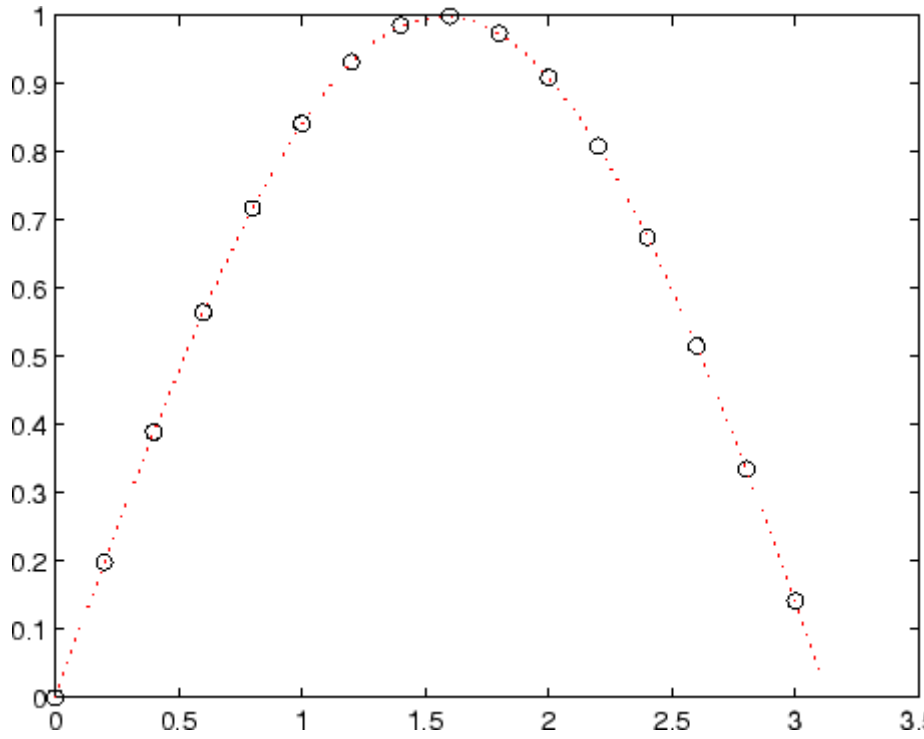
then the population in 1975, obtained by table lookup within the matrix `tab`, is

```
p = interp1(tab(:,1),tab(:,2),1975)
p =
    214.8585
```

Example 4

The following example uses the `'cubic'` method to generate the piecewise polynomial form (ppform) of `Y`, and then evaluates the result using `ppval`.

```
x = 0:.2:pi; y = sin(x);
pp = interp1(x,y,'cubic','pp');
xi = 0:.1:pi;
yi = ppval(pp,xi);
plot(x,y,'ko'), hold on, plot(xi,yi,'r:'), hold off
```



Algorithm

The `interp1` command is a MATLAB M-file. The `'nearest'` and `'linear'` methods have straightforward implementations.

For the 'spline' method, interp1 calls a function spline that uses the functions ppval, mkpp, and unmkpp. These routines form a small suite of functions for working with piecewise polynomials. spline uses them to perform the cubic spline interpolation. For access to more advanced features, see the spline reference page, the M-file help for these functions, and the Spline Toolbox.

For the 'pchip' and 'cubic' methods, interp1 calls a function pchip that performs piecewise cubic interpolation within the vectors x and y. This method preserves monotonicity and the shape of the data. See the pchip reference page for more information.

Interpolating Complex Data

For Real x and Complex Y. For interp1(x,Y,...) where x is real and Y is complex, you can use any interp1 method except for 'pchip'. The shape-preserving aspect of the 'pchip' algorithm involves the signs of the slopes between the data points. Because there is no notion of sign with complex data, it is impossible to talk about whether a function is increasing or decreasing. Consequently, the 'pchip' algorithm does not generalize to complex data.

The 'spline' method is often a good choice because piecewise cubic splines are derived purely from smoothness conditions. The second derivative of the interpolant must be continuous across the interpolating points. This does not involve any notion of sign or shape and so generalizes to complex data.

For Complex x. For interp1(x,Y,...) where x is complex and Y is either real or complex, use the two-dimensional interpolation routine interp2(REAL(x), IMAG(x),Y,...) instead.

See Also

interp1q, interpft, interp2, interp3, interpn, pchip, spline

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

interp1q

Purpose

Quick 1-D linear interpolation

Syntax

```
yi = interp1q(x,Y,xi)
```

Description

`yi = interp1q(x,Y,xi)` returns the value of the 1-D function `Y` at the points of column vector `xi` using linear interpolation. The vector `x` specifies the coordinates of the underlying interval. The length of output `yi` is equal to the length of `xi`.

`interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking.

For `interp1q` to work properly,

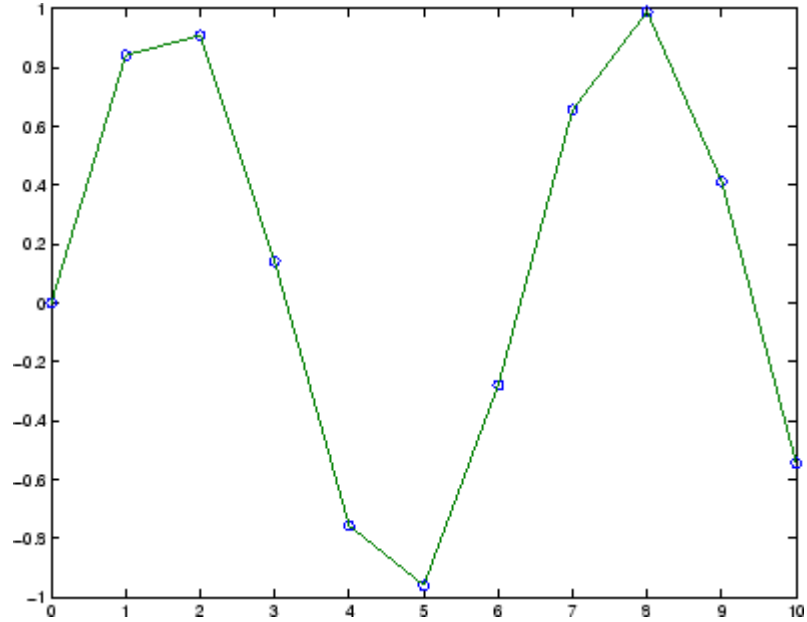
- `x` must be a monotonically increasing column vector.
- `Y` must be a column vector or matrix with `length(x)` rows.
- `xi` must be a column vector

`interp1q` returns NaN for any values of `xi` that lie outside the coordinates in `x`. If `Y` is a matrix, then the interpolation is performed for each column of `Y`, in which case `yi` is `length(xi)-by-size(Y,2)`.

Example

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = 0:10';
y = sin(x);
xi = (0:.25:10)';
yi = interp1q(x,y,xi);
plot(x,y,'o',xi,yi)
```



See Also

`interp1`, `interp2`, `interp3`, `interpn`

Purpose

2-D data interpolation (table lookup)

Syntax

```
ZI = interp2(X,Y,Z,XI,YI)
ZI = interp2(Z,XI,YI)
ZI = interp2(Z,ntimes)
ZI = interp2(X,Y,Z,XI,YI,method)
ZI = interp2(...,method, extrapval)
```

Description

`ZI = interp2(X,Y,Z,XI,YI)` returns matrix `ZI` containing elements corresponding to the elements of `XI` and `YI` and determined by interpolation within the two-dimensional function specified by matrices `X`, `Y`, and `Z`. `X` and `Y` must be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. Matrices `X` and `Y` specify the points at which the data `Z` is given. Out of range values are returned as `NaN`s.

`XI` and `YI` can be matrices, in which case `interp2` returns the values of `Z` corresponding to the points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `interp2` interprets these vectors as if you issued the command `meshgrid(xi,yi)`.

`ZI = interp2(Z,XI,YI)` assumes that `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`.

`ZI = interp2(Z,ntimes)` expands `Z` by interleaving interpolates between every element, working recursively for `ntimes`. `interp2(Z)` is the same as `interp2(Z,1)`.

`ZI = interp2(X,Y,Z,XI,YI,method)` specifies an alternative interpolation method:

`'nearest'`	Nearest neighbor interpolation
`'linear'`	Linear interpolation (default)

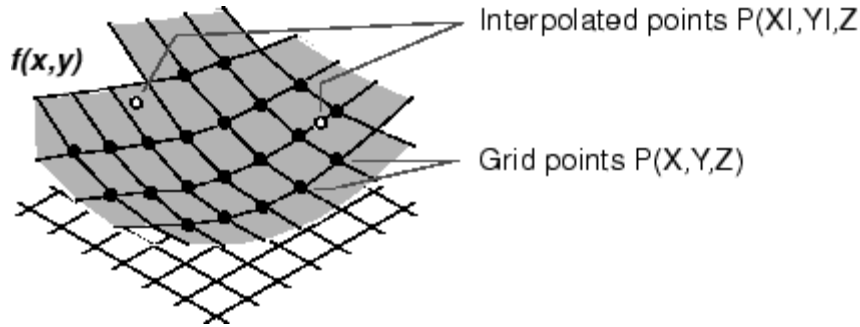
'spline'	Cubic spline interpolation
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as 'spline'.

All interpolation methods require that X and Y be monotonic, and have the same format ("plaid") as if they were produced by meshgrid. If you provide two monotonic vectors, interp2 changes them to a plaid internally. Variable spacing is handled by mapping the given values in X, Y, XI, and YI to an equally spaced domain before interpolating. For faster interpolation when X and Y are equally spaced and monotonic, use the methods '*linear', '*cubic', '*spline', or '*nearest'.

ZI = interp2(...,method, extrapval) specifies a method and a scalar value for ZI outside of the domain created by X and Y. Thus, ZI equals extrapval for any value of YI or XI that is not spanned by Y or X respectively. A method must be specified to use extrapval. The default method is 'linear'.

Remarks

The interp2 command interpolates between data points. It finds values of a two-dimensional function $f(x, y)$ underlying the data at intermediate points.



Interpolation is the same operation as table lookup. Described in table lookup terms, the table is tab = [NaN,Y;X,Z] and interp2 looks up

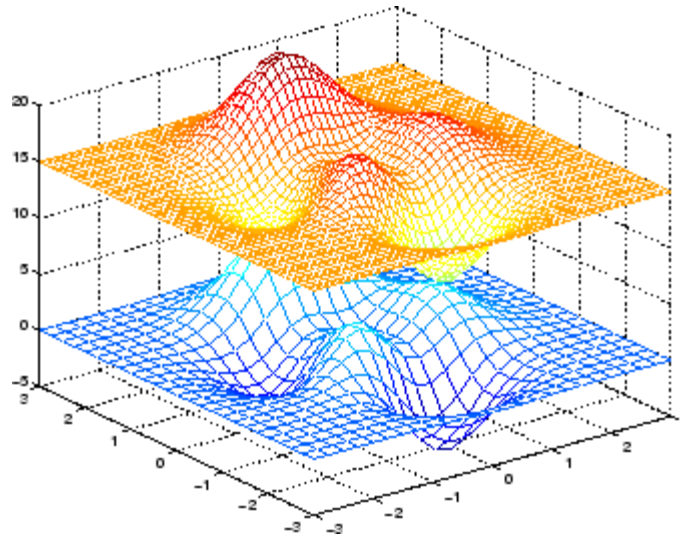
the elements of XI in X, YI in Y, and, based upon their location, returns values ZI interpolated within the elements of Z.

Examples

Example 1

Interpolate the peaks function over a finer grid.

```
[X,Y] = meshgrid(-3:.25:3);
Z = peaks(X,Y);
[XI,YI] = meshgrid(-3:.125:3);
ZI = interp2(X,Y,Z,XI,YI);
mesh(X,Y,Z), hold, mesh(XI,YI,ZI+15)
hold off
axis([-3 3 -3 3 -5 20])
```



Example 2

Given this set of employee data,

```
years = 1950:10:1990;
service = 10:10:30;
```

```
wage = [150.697 199.592 187.625
179.323 195.072 250.287
203.212 179.092 322.767
226.505 153.706 426.730
249.633 120.281 598.243];
```

it is possible to interpolate to find the wage earned in 1975 by an employee with 15 years' service:

```
w = interp2(service,years,wage,15,1975)
w =
  190.6287
```

See Also

griddata, interp1, interp1q, interp3, interpn, meshgrid

Purpose

3-D data interpolation (table lookup)

Syntax

```
VI = interp3(X,Y,Z,V,XI,YI,ZI)
VI = interp3(V,XI,YI,ZI)
VI = interp3(V,ntimes)
VI = interp3(...,method)
VI = interp3(...,method,extrapval)
```

Description

`VI = interp3(X,Y,Z,V,XI,YI,ZI)` interpolates to find `VI`, the values of the underlying three-dimensional function `V` at the points in arrays `XI`, `YI` and `ZI`. `XI`,`YI`, `ZI` must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `meshgrid` to create the `Y1`, `Y2`, `Y3` arrays. Arrays `X`, `Y`, and `Z` specify the points at which the data `V` is given. Out of range values are returned as `NaN`.

`VI = interp3(V,XI,YI,ZI)` assumes `X=1:N`, `Y=1:M`, `Z=1:P` where `[M,N,P]=size(V)`.

`VI = interp3(V,ntimes)` expands `V` by interleaving interpolates between every element, working recursively for `ntimes` iterations. The command `interp3(V)` is the same as `interp3(V,1)`.

`VI = interp3(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)
'spline'	Cubic spline interpolation
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as `'spline'`.

`VI = interp3(...,method,extrapval)` specifies a method and a value for `VI` outside of the domain created by `X`, `Y` and `Z`. Thus, `VI` equals `extrapval` for any value of `XI`, `YI` or `ZI` that is not spanned by `X`, `Y`, and `Z`, respectively. You must specify a method to use `extrapval`. The default method is `'linear'`.

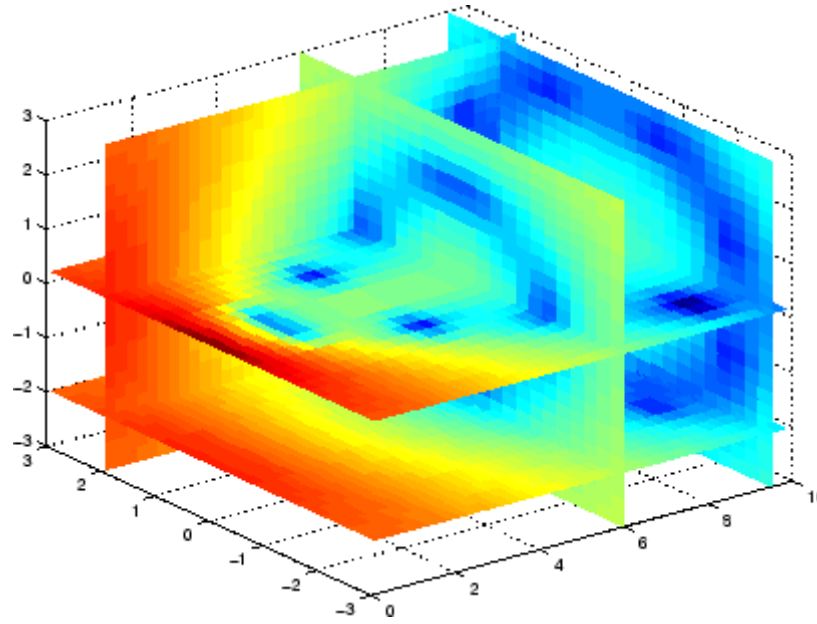
Discussion

All the interpolation methods require that `X`,`Y` and `Z` be monotonic and have the same format ("plaid") as if they were created using `meshgrid`. `X`, `Y`, and `Z` can be non-uniformly spaced. For faster interpolation when `X`, `Y`, and `Z` are equally spaced and monotonic, use the methods '`*linear`', '`*cubic`', or '`*nearest`'.

Examples

To generate a coarse approximation of `flow` and interpolate over a finer mesh:

```
[x,y,z,v] = flow(10);
[xi,yi,zi] = meshgrid(.1:.25:10, -3:.25:3, -3:.25:3);
vi = interp3(x,y,z,v,xi,yi,zi); % vi is 25-by-40-by-25
slice(xi,yi,zi,vi,[6 9.5],2,[-2 .2]), shading flat
```



See Also

`interp1`, `interp1q`, `interp2`, `interpn`, `meshgrid`

interpft

Purpose

1-D interpolation using FFT method

Syntax

```
y = interpft(x,n)
y = interpft(x,n,dim)
```

Description

`y = interpft(x,n)` returns the vector `y` that contains the value of the periodic function `x` resampled to `n` equally spaced points.

If `length(x) = m`, and `x` has sample interval `dx`, then the new sample interval for `y` is `dy = dx*m/n`. Note that `n` cannot be smaller than `m`.

If `X` is a matrix, `interpft` operates on the columns of `X`, returning a matrix `Y` with the same number of columns as `X`, but with `n` rows.

`y = interpft(x,n,dim)` operates along the specified dimension.

Algorithm

The `interpft` command uses the FFT method. The original vector `x` is transformed to the Fourier domain using `fft` and then transformed back with more points.

Examples

Interpolate a triangle-like signal using an interpolation factor of 5. First, set up signal to be interpolated:

```
y = [0 .5 1 1.5 2 1.5 1 .5 0 -.5 -1 -1.5 -2 -1.5 -1 -.5 0];
N = length(y);
```

Perform the interpolation:

```
L = 5;
M = N*L;
x = 0:L:L*N-1;
xi = 0:M-1;
yi = interpft(y,M);
plot(x,y,'o',xi,yi,'*')
legend('Original data','Interpolated data')
```

See Also

`interp1`

Purpose

N-D data interpolation (table lookup)

Syntax

```
VI = interpn(X1,X2,X3,...,V,Y1,Y2,Y3,...)
VI = interpn(V,Y1,Y2,Y3,...)
VI = interpn(V,ntimes)
VI = interpn(...,method)
VI = interpn(...,method,extrapval)
```

Description

`VI = interpn(X1,X2,X3,...,V,Y1,Y2,Y3,...)` interpolates to find `VI`, the values of the underlying multidimensional function `V` at the points in the arrays `Y1`, `Y2`, `Y3`, etc. For an n-dimensional array `V`, `interpn` is called with `2*N+1` arguments. Arrays `X1`, `X2`, `X3`, etc. specify the points at which the data `V` is given. Out of range values are returned as `NaN`s. `Y1`, `Y2`, `Y3`, etc. must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `ndgrid` to create the `Y1`, `Y2`, `Y3`, etc. arrays. `interpn` works for all n-dimensional arrays with 2 or more dimensions.

`VI = interpn(V,Y1,Y2,Y3,...)` interpolates as above, assuming `X1 = 1:size(V,1)`, `X2 = 1:size(V,2)`, `X3 = 1:size(V,3)`, etc.

`VI = interpn(V,ntimes)` expands `V` by interleaving interpolates between each element, working recursively for `ntimes` iterations. `interpn(V)` is the same as `interpn(V,1)`.

`VI = interpn(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)
'spline'	Cubic spline interpolation
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. Otherwise, this method is the same as `'spline'`.

`VI = interpn(...,method,extrapval)` specifies a method and a value for `VI` outside of the domain created by `X1`, `X2`, `...`. Thus, `VI` equals

`extrapval` for any value of `Y1`, `Y2`,... that is not spanned by `X1`, `X2`,... respectively. You must specify a method to use `extrapval`. The default method is `'linear'`.

`interpn` requires that `X1`, `X2`, `X3`, ... be monotonic and plaid (as if they were created using `ndgrid`). `X1`, `X2`, `X3`, and so on can be non-uniformly spaced.

Discussion

All the interpolation methods require that `X1`,`X2`, `X3` ... be monotonic and have the same format ("plaid") as if they were created using `ndgrid`. `X1`,`X2`,`X3`,... and `Y1`, `Y2`, `Y3`, etc. can be non-uniformly spaced. For faster interpolation when `X1`, `X2`, `X3`, etc. are equally spaced and monotonic, use the methods '`*linear`', '`*cubic`', or '`*nearest`'.

Examples

Start by defining an anonymous function to compute $f = te^{-x^2-y^2-z^2}$:

```
f = @(x,y,z,t) t.*exp(-x.^2 - y.^2 - z.^2);
```

Build the lookup table by evaluating the function `f` on a grid constructed by `ndgrid`:

```
[x,y,z,t] = ndgrid(-1:0.2:1,-1:0.2:1,-1:0.2:1,0:2:10);
v = f(x,y,z,t);
```

Now construct a finer grid:

```
[xi,yi,zi,ti] = ndgrid(-1:0.05:1,-1:0.08:1,-1:0.05:1, ...
                       0:0.5:10);
```

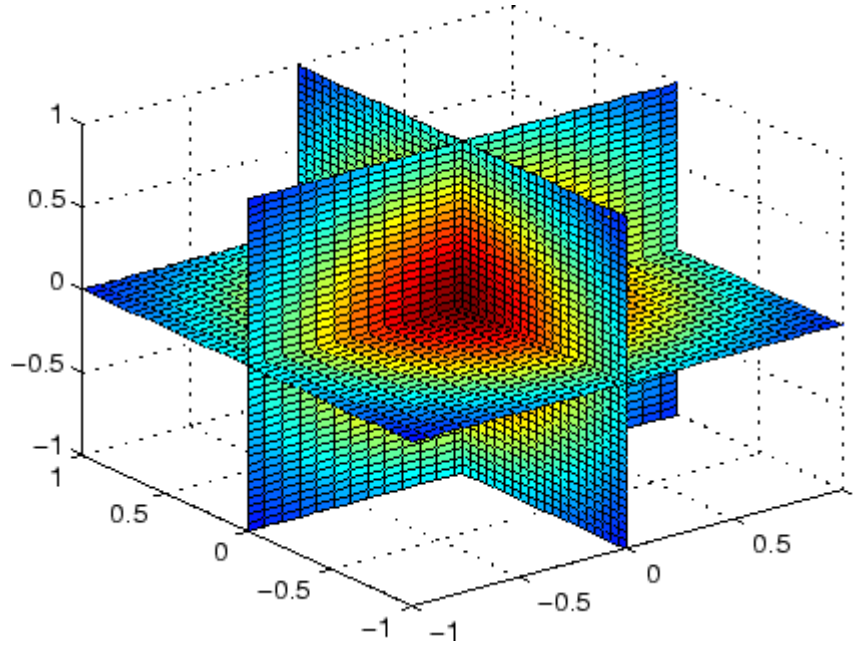
Compute the spline interpolation at `xi`, `yi`, `zi`, and `ti`:

```
vi = interpn(x,y,z,t,v,xi,yi,zi,ti,'spline');
```

Plot the interpolated function, and then create a movie from the plot:

```
nframes = size(ti, 4);
for j = 1:nframes
   slice(yi(:,:,:,j), xi(:,:,:,j), zi(:,:,:,j), ...
```

```
            vi(:,:,:,j),0,0,0);
    caxis([0 10]);
    M(j) = getframe;
end
movie(M);
```



See Also

`interp1`, `interp2`, `interp3`, `ndgrid`

interpstreamspeed

Purpose

Interpolate stream-line vertices from flow speed

Syntax

```
interpstreamspeed(X,Y,Z,U,V,W,vertices)
interpstreamspeed(U,V,W,vertices)
interpstreamspeed(X,Y,Z,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(X,Y,U,V,vertices)
interpstreamspeed(U,V,vertices)
interpstreamspeed(X,Y,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(...,sf)
vertsout = interpstreamspeed(...)
```

Description

`interpstreamspeed(X,Y,Z,U,V,W,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`).

`interpstreamspeed(U,V,W,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(U)`.

`interpstreamspeed(X,Y,Z,speed,vertices)` uses the 3-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p]=size(speed)`.

`interpstreamspeed(X,Y,U,V,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`interpstreamspeed(U,V,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M N]=size(U)`.

`interpstreamspeed(X,Y,speed,vertices)` uses the 2-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M,N]= size(speed)`.

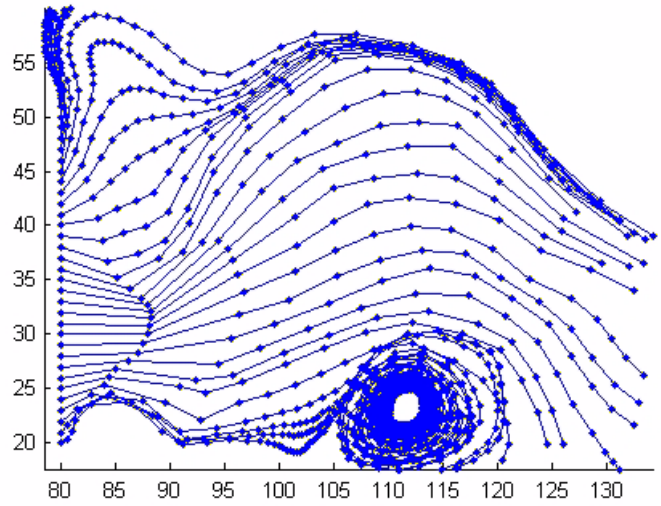
`interpstreamspeed(...,sf)` uses `sf` to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if `sf` is 3, then `interpstreamspeed` creates only one-third of the vertices.

`vertsout = interpstreamspeed(...)` returns a cell array of vertex arrays.

Examples

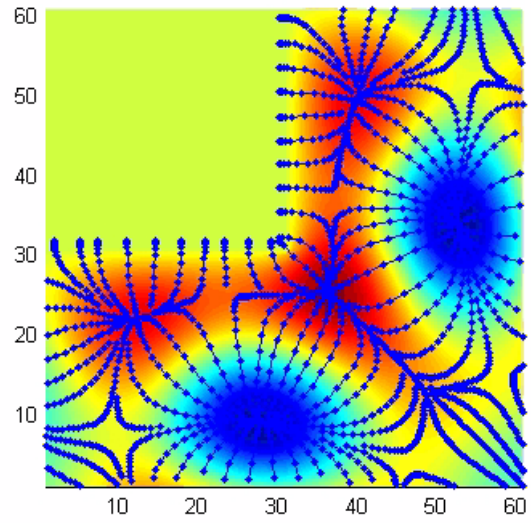
This example draws streamlines using the vertices returned by `interpstreamspeed`. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Streamlines having widely spaced vertices indicate faster flow; those with closely spaced vertices indicate slower flow.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.2);
sl = streamline(iverts);
set(sl,'Marker','.')
axis tight; view(2); daspect([1 1 1])
```



This example plots streamlines whose vertex spacing indicates the value of the gradient along the streamline.

```
z = membrane(6,30);
[u v] = gradient(z);
[verts averts] = streamslice(u,v);
iverts = interpstreamspeed(u,v,verts,15);
sl = streamline(iverts);
set(sl,'Marker','.')
hold on; pcolor(z); shading interp
axis tight; view(2); daspect([1 1 1])
```



See Also

`stream2`, `stream3`, `streamline`, `streamslice`, `streamparticles`

"Volume Visualization" on page 1-102 for related functions

intersect

Purpose

Find set intersection of two vectors

Syntax

```
c = intersect(A, B)
c = intersect(A, B, 'rows')
[c, ia, ib] = intersect(a, b)
```

Description

`c = intersect(A, B)` returns the values common to both `A` and `B`. In set theoretic terms, this is A[[INTERSECT]] B. Inputs `A` and `B` can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

`c = intersect(A, B, 'rows')` when `A` and `B` are matrices with the same number of columns returns the rows common to both `A` and `B`.

`[c, ia, ib] = intersect(a, b)` also returns column index vectors `ia` and `ib` such that `c = a(ia)` and `c = b(ib)` (or `c = a(ia,:)` and `c = b(ib,:)`).

Remarks

Because NaN is considered to be not equal to itself, it is never included in the result `c`.

Examples

```
A = [1 2 3 6]; B = [1 2 3 4 6 10 20];
[c, ia, ib] = intersect(A, B);
disp([c; ia; ib])
     1     2     3     6
     1     2     3     4
     1     2     3     5
```

See Also

`ismember`, `issorted`, `setdiff`, `setxor`, `union`, `unique`

Purpose

Largest value of specified integer type

Syntax

```
v = intmax
v = intmax('classname')
```

Description

`v = intmax` is the largest positive value that can be represented in MATLAB with a 32-bit integer. Any value larger than the value returned by `intmax` saturates to the `intmax` value when cast to a 32-bit integer.

`v = intmax('classname')` is the largest positive value in the integer class `classname`. Valid values for the string `classname` are

`'int8'`	`'int16'`	`'int32'`	`'int64'`
`'uint8'`	`'uint16'`	`'uint32'`	`'uint64'`

`intmax('int32')` is the same as `intmax` with no arguments.

Examples

Find the maximum value for a 64-bit signed integer:

```
v = intmax('int64')
v =
  9223372036854775807
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmax`:

```
isequal(x, intmax)
ans =
     1
```

See Also

`intmin`, `realmax`, `realmin`, `int8`, `uint8`, `isa`, `class`

intmin

Purpose

Smallest value of specified integer type

Syntax

```
v = intmin
v = intmin('classname')
```

Description

`v = intmin` is the smallest value that can be represented in MATLAB with a 32-bit integer. Any value smaller than the value returned by `intmin` saturates to the `intmin` value when cast to a 32-bit integer.

`v = intmin('classname')` is the smallest positive value in the integer class `classname`. Valid values for the string `classname` are

`'int8'`	`'int16'`	`'int32'`	`'int64'`
`'uint8'`	`'uint16'`	`'uint32'`	`'uint64'`

`intmin('int32')` is the same as `intmin` with no arguments.

Examples

Find the minimum value for a 64-bit signed integer:

```
v = intmin('int64')
v =
 -9223372036854775808
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmin`:

```
isequal(x, intmin)
ans =
     1
```

See Also

`intmax`, `realmin`, `realmax`, `int8`, `uint8`, `isa`, `class`

Purpose

Control state of integer warnings

Syntax

```
intwarning('action')
s = intwarning('action')
intwarning(s)
sOld = intwarning(sNew)
```

Description

MATLAB has four types of integer warnings. The `intwarning` function enables, disables, or returns information on these warnings:

- `MATLAB:intConvertNaN` — Warning on an attempt to convert NaN (Not a Number) to an integer. The result of the operation is zero.
- `MATLAB:intConvertNonIntVal` — Warning on an attempt to convert a non-integer value to an integer. The result is that the input value is rounded to the nearest integer for that class.
- `MATLAB:intConvertOverflow` — Warning on overflow when attempting to convert from a numeric class to an integer class. The result is the maximum value for the target class.
- `MATLAB:intMathOverflow` — Warning on overflow when attempting an integer arithmetic operation. The result is the maximum value for the class of the input value. MATLAB also issues this warning when NaN is computed (e.g., `int8(0)/0`).

`intwarning('action')` sets or displays the state of integer warnings in MATLAB according to the string, *`action`*. There are three possible actions, as shown here. The default state is `'off'`.

Action	Description
`off`	Disable the display of integer warnings
`on`	Enable the display of integer warnings
`query`	Display the state of all integer warnings

`s = intwarning('action')` sets the state of integer warnings in MATLAB according to the string *`action`*, and then returns the previous

state in a 4-by-1 structure array, `s`. The return structure array has two fields: `identifier` and `state`.

`intwarning(s)` sets the state of integer warnings in MATLAB according to the `identifier` and `state` fields in structure array `s`.

`sOld = intwarning(sNew)` sets the state of integer warnings in MATLAB according to `sNew`, and then returns the previous state in `sOld`.

Remarks

Caution Enabling the `MATLAB:intMathOverflow` warning slows down integer arithmetic. It is recommended that you enable this particular warning only when you need to diagnose unusual behavior in your code, and disable it during normal program operation. The other integer warnings listed here do not affect program performance.

Examples

General Usage

Examples of the four types of integer warnings are shown here:

- **MATLAB:intConvertNaN**

Attempt to convert NaN (Not a Number) to an unsigned integer:

```
uint8(NaN);
Warning: NaN converted to uint8(0).
```

- **MATLAB:intConvertNonIntVal**

Attempt to convert a floating point number to an unsigned integer:

```
uint8(2.7);
Warning: Conversion rounded non-integer floating point
   value to nearest uint8 value.
```

- **MATLAB:intConvertOverflow**

Attempt to convert a large unsigned integer to a signed integer, where the operation overflows:

```
int8(uint8(200));
Warning: Out of range value converted to intmin('int8')
   or intmax('int8').
```

- **MATLAB:intMathOverflow**

Attempt an integer arithmetic operation that overflows:

```
intmax('uint8') + 5;
Warning: Out of range value or NaN computed in
integer arithmetic.
```

Example 1

Check the initial state of integer warnings:

```
intwarning('query')
The state of warning 'MATLAB:intConvertNaN' is 'off'.
The state of warning 'MATLAB:intConvertNonIntVal' is 'off'.
The state of warning 'MATLAB:intConvertOverflow' is 'off'.
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Convert a floating point value to an 8-bit unsigned integer. MATLAB does the conversion, but that requires rounding the resulting value. Because all integer warnings have been disabled, no warning is displayed:

```
uint8(2.7)
ans =
    3
```

Store this state in structure array `iwState`:

```
iwState = intwarning('query');
```

Change the state of the `ConvertNonIntVal` warning to `'on'` by first setting the state to `'on'` in the `iwState` structure array, and then

loading `iwState` back into the internal integer warning settings for your MATLAB session:

```
maxintwarn = 4;

for k = 1:maxintwarn
   if strcmp(iwState(k).identifier, ...
             'MATLAB:intConvertNonIntVal')
      iwState(k).state = 'on';
      intwarning(iwState);
   end
end
```

Verify that the state of `ConvertNonIntVal` has changed:

```
intwarning('query')
The state of warning 'MATLAB:intConvertNaN' is 'off'.
The state of warning 'MATLAB:intConvertNonIntVal' is 'on'.
The state of warning 'MATLAB:intConvertOverflow' is 'off'.
The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

Now repeat the conversion from floating point to integer. This time MATLAB displays the warning:

```
uint8(2.7)
Warning: Conversion rounded non-integer floating point
   value to nearest uint8 value.
ans =
    3
```

See Also

`warning`, `lastwarn`

Purpose

Matrix inverse

Syntax

```
Y = inv(X)
```

Description

`Y = inv(X)` returns the inverse of the square matrix `X`. A warning message is printed if `X` is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations $Ax = b$. One way to solve this is with `x = inv(A)*b`. A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator `x = A\b`. This produces the solution using Gaussian elimination, without forming the inverse. See `\` and `/` for further information.

Examples

Here is an example demonstrating the difference between solving a linear system by inverting the matrix with `inv(A)*b` and solving it directly with `A\b`. A random matrix `A` of order 500 is constructed so that its condition number, `cond(A)`, is `1.e10`, and its norm, `norm(A)`, is `1`. The exact solution `x` is a random vector of length 500 and the right-hand side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

On a 300 MHz, laptop computer the statements

```
n = 500;
Q = orth(randn(n,n));
d = logspace(0,-10,n);
A = Q*diag(d)*Q';
x = randn(n,1);
b = A*x;
tic, y = inv(A)*b; toc
err = norm(y-x)
res = norm(A*y-b)
```

produce

```
elapsed_time =
```

```
    1.4320
err =
    7.3260e-006
res =
    4.7511e-007
```

while the statements

```
tic, z = A\b, toc
err = norm(z-x)
res = norm(A*z-b)
```

produce

```
elapsed_time =
    0.6410
err =
    7.1209e-006
res =
    4.4509e-015
```

It takes almost two and one half times as long to compute the solution with `y = inv(A)*b` as with `z = A\b`. Both produce computed solutions with about the same error, `1.e-6`, reflecting the condition number of the matrix. But the size of the residuals, obtained by plugging the computed solution back into the original equations, differs by several orders of magnitude. The direct solution produces residuals on the order of the machine accuracy, even though the system is badly conditioned.

The behavior of this example is typical. Using `A\b` instead of `inv(A)*b` is two to three times as fast and produces residuals on the order of machine accuracy, relative to the magnitude of the data.

Algorithm

Inputs of Type Double

For inputs of type `double`, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	DLANGE, DGETRF, DGECON, DGETRI
Complex	ZLANGE, ZGETRF, ZGECON, ZGETRI

Inputs of Type Single

For inputs of type `single`, `inv` uses the following LAPACK routines to compute the matrix inverse:

Matrix	Routine
Real	SLANGE, SGETRF, SGECON, SGETRI
Complex	CLANGE, CGETRF, CGECON, CGETRI

See Also

`det`, `lu`, `rref`

The arithmetic operators `\`, `/`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

invhilb

Purpose

Inverse of Hilbert matrix

Syntax

```
H = invhilb(n)
```

Description

`H = invhilb(n)` generates the exact inverse of the exact Hilbert matrix for `n` less than about 15. For larger `n`, `invhilb(n)` generates an approximation to the inverse Hilbert matrix.

Limitations

The exact inverse of the exact Hilbert matrix is a matrix whose elements are large integers. These integers may be represented as floating-point numbers without roundoff error as long as the order of the matrix, `n`, is less than 15.

Comparing `invhilb(n)` with `inv(hilb(n))` involves the effects of two or three sets of roundoff errors:

- The errors caused by representing `hilb(n)`
- The errors in the matrix inversion process
- The errors, if any, in representing `invhilb(n)`

It turns out that the first of these, which involves representing fractions like 1/3 and 1/5 in floating-point, is the most significant.

Examples

`invhilb(4)` is

```
    16   -120    240   -140
  -120   1200  -2700   1680
   240  -2700   6480  -4200
  -140   1680  -4200   2800
```

See Also

`hilb`

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

Purpose

Invoke method on object or interface, or display methods

Syntax

```
S = h.invoke
S = h.invoke('methodname')
S = h.invoke('methodname', arg1, arg2, ...)
S = h.invoke('custominterfacename')
S = invoke(h, ...)
```

Description

`S = h.invoke` returns structure array `S` containing a list of all methods supported by the object or interface, `h`, along with the prototypes for these methods.

If `S` is empty, either there are no properties or methods in the object, or MATLAB cannot read the object's type library. Refer to the COM vendor's documentation. For Automation objects, if the vendor provides documentation for specific properties or methods, use the `S = invoke(h, ...)` syntax to call them.

`S = h.invoke('methodname')` invokes the method specified in the string `methodname`, and returns an output value, if any, in `S`. The data type of the return value is dependent upon the specific method being invoked and is determined by the specific control or server.

`S = h.invoke('methodname', arg1, arg2, ...)` invokes the method specified in the string `methodname` with input arguments `arg1`, `arg2`, etc.

`S = h.invoke('custominterfacename')` returns an `Interface` object that serves as a handle to a custom interface implemented by the COM component. The `h` argument is a handle to the COM object. The `custominterfacename` argument is a quoted string returned by the `interfaces` function.

`S = invoke(h, ...)` is an alternate syntax for the same operation.

Remarks

If the method returns a COM interface, then `invoke` returns a new MATLAB COM object that represents the interface returned. See "Handling COM Data in MATLAB" in the External Interfaces

documentation for a description of how MATLAB converts COM data types.

Examples

Example 1 — Invoking a Method

Create an `mwsamp` control and invoke its `Redraw` method:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);

h.Radius = 100;
h.invoke('Redraw');
```

Here is a simpler way to use `invoke`. Just call the method directly, passing the handle, and any arguments:

```
h.Redraw;
```

Call `invoke` with only the handle argument to display a list of all `mwsamp` methods:

```
h.invoke
ans =
   AboutBox = void AboutBox(handle)
   Beep = void Beep(handle)
   FireClickEvent = void FireClickEvent(handle)
             .
             .
            etc.
```

Example 2 — Getting a Custom Interface

Once you have created a COM server, you can query the server component to see if any custom interfaces are implemented. Use the `interfaces` function to return a list of all available custom interfaces:

```
h = actxserver('mytestenv.calculator')
h =
   COM.mytestenv.calculator
```

```
customlist = h.interfaces
customlist =
   ICalc1
   ICalc2
   ICalc3
```

To get a handle to the custom interface you want, use the `invoke` function, specifying the handle returned by `actxcontrol` or `actxserver` and also the name of the custom interface:

```
c1 = h.invoke('ICalc1')
c1 =
   Interface.Calc_1.0_Type_Library.ICalc_Interface
```

You can now use this handle with most of the COM client functions to access the properties and methods of the object through the selected custom interface.

See Also

`methods`, `ismethod`, `interfaces`

ipermute

Purpose

Inverse permute dimensions of N-D array

Syntax

```
A = ipermute(B,order)
```

Description

`A = ipermute(B,order)` is the inverse of `permute`. `ipermute` rearranges the dimensions of `B` so that `permute(A,order)` will produce `B`. `B` has the same values as `A` but the order of the subscripts needed to access any particular element are rearranged as specified by `order`. All the elements of `order` must be unique.

Remarks

`permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

Examples

Consider the 2-by-2-by-3 array `a`:

```
a = cat(3,eye(2),2*eye(2),3*eye(2))

a(:,:,1) =              a(:,:,2) =
     1     0                   2     0
     0     1                   0     2

a(:,:,3) =
     3     0
     0     3
```

Permuting and inverse permuting `a` in the same fashion restores the array to its original form:

```
B = permute(a,[3 2 1]);
C = ipermute(B,[3 2 1]);
isequal(a,C)
ans=

   1
```

See Also

`permute`

Purpose

Interquartile range of `timeseries` data

Syntax

```
ts_iqr = iqr(ts)
iqr(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_iqr = iqr(ts)` returns the interquartile range of `ts.Data`. When `ts.Data` is a vector, ts_iqr is the difference between the 75th and the 25th percentiles of the `ts.Data` values. When `ts.Data` is a matrix, ts_iqr is a row vector containing the interquartile range of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `iqr` always operates along the first nonsingleton dimension of `ts.Data`.

`iqr(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

Create a time series with a missing value, represented by `NaN`.

```
ts = timeseries([3.0 NaN 5 6.1 8], 1:5);
```

Calculate the interquartile range of `ts.Data` after removing the missing value from the calculation.

```
iqr(ts,'MissingData','remove')
```

```
ans =

    3.0500
```

See Also

timeseries

Purpose

Detect state

Description

These functions detect the state of MATLAB entities:

`isa`	Detect object of given MATLAB class or Java class
`isappdata`	Determine if object has specific application-defined data
`iscell`	Determine if input is cell array
`iscellstr`	Determine if input is cell array of strings
`ischar`	Determine if input is character array
`iscom`	Determine if input is Component Object Model (COM) object
`isdir`	Determine if input is directory
`isempty`	Determine if input is empty array
`isequal`	Determine if arrays are numerically equal
`isequalwithequalnans`	Determine if arrays are numerically equal, treating `NaN`s as equal
`isevent`	Determine if input is object event
`isfield`	Determine if input is MATLAB structure array field
`isfinite`	Detect finite elements of array
`isfloat`	Determine if input is floating-point array
`isglobal`	Determine if input is global variable
`ishandle`	Detect valid graphics object handles
`ishold`	Determine if graphics hold state is on
`isinf`	Detect infinite elements of array
`isinteger`	Determine if input is integer array
`isinterface`	Determine if input is Component Object Model (COM) interface

isjava	Determine if input is Java object
iskeyword	Determine if input is MATLAB keyword
isletter	Detect elements that are alphabetic letters
islogical	Determine if input is logical array
ismember	Detect members of specific set
ismethod	Determine if input is object method
isnan	Detect elements of array that are not a number (NaN)
isnumeric	Determine if input is numeric array
isobject	Determine if input is MATLAB OOPs object
ispc	Determine if PC (Windows) version of MATLAB
isprime	Detect prime elements of array
isprop	Determine if input is object property
isreal	Determine if all array elements are real numbers
isscalar	Determine if input is scalar
issorted	Determine if set elements are in sorted order
isspace	Detect space characters in array
issparse	Determine if input is sparse array
isstrprop	Determine if string is of specified category
isstruct	Determine if input is MATLAB structure array
isstudent	Determine if Student Version of MATLAB
isunix	Determine if UNIX version of MATLAB
isvarname	Determine if input is valid variable name
isvector	Determine if input is vector

See Also

isa

Purpose

Determine whether input is object of given class

Syntax

```
K = isa(obj, 'class_name')
```

Description

`K = isa(obj, 'class_name')` returns logical `1` (`true`) if `obj` is of class (or a subclass of) *`class_name`*, and logical `0` (`false`) otherwise.

The argument `obj` is a MATLAB object or a Java object. The argument *`class_name`* is the name of a MATLAB (predefined or user-defined) or a Java class. Predefined MATLAB classes include

`logical`	Logical array of `true` and `false` values
`char`	Characters array
`numeric`	Integer or floating-point array
`integer`	Signed or unsigned integer array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array
`float`	Single- or double-precision floating-point array
`single`	Single-precision floating-point array
`double`	Double-precision floating-point array
`cell`	Cell array
`struct`	Structure array

`function_handle`	Function handle
'*`class_name`*'	Custom MATLAB object class or Java class

To check for a sparse array, use `issparse`. To check for a complex array, use `~isreal`.

Examples

```
isa(rand(3,4),'double')
ans =
    1
```

The following example creates an instance of the user-defined MATLAB class named `polynom`. The `isa` function identifies the object as being of the `polynom` class.

```
polynom_obj = polynom([1 0 -2 -5]);
isa(polynom_obj, 'polynom')
ans =
     1
```

See Also

`class`, `is*`

Purpose

True if application-defined data exists

Syntax

```
isappdata(h,name)
```

Description

`isappdata(h,name)` returns `1` if application-defined data with the specified `name` exists on the object specified by handle `h`, and returns `0` otherwise.

See Also

`getappdata`, `rmappdata`, `setappdata`

iscell

Purpose

Determine whether input is cell array

Syntax

```
tf = iscell(A)
```

Description

`tf = iscell(A)` returns logical `1` (`true`) if `A` is a cell array and logical `0` (`false`) otherwise.

Examples

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
A{2,1} = 3+7i;
A{2,2} = -pi:pi/10:pi;

iscell(A)

ans =

     1
```

See Also

`cell`, `iscellstr`, `isstruct`, `isnumeric`, `islogical`, `isobject`, `isa`, `is*`

Purpose

Determine whether input is cell array of strings

Syntax

```
tf = iscellstr(A)
```

Description

`tf = iscellstr(A)` returns logical `1` (`true`) if `A` is a cell array of strings and logical `0` (`false`) otherwise. A cell array of strings is a cell array where every element is a character array.

Examples

```
A{1,1} = 'Thomas Lee';
A{1,2} = 'Marketing';
A{2,1} = 'Allison Jones';
A{2,2} = 'Development';

iscellstr(A)

ans =

     1
```

See Also

`cellstr`, `iscell`, `isstrprop`, `strings`, `char`, `isstruct`, `isa`, `is*`

ischar

Purpose

Determine whether item is character array

Syntax

```
tf = ischar(A)
```

Description

`tf = ischar(A)` returns logical `1` (`true`) if `A` is a character array and logical `0` (`false`) otherwise.

Examples

Given the following cell array,

```
C{1,1} = magic(3);           % double array
C{1,2} = 'John Doe';         % char array
C{1,3} = 2 + 4i              % complex double

C =

    [3x3 double]    'John Doe'    [2.0000+ 4.0000i]
```

`ischar` shows that only `C{1,2}` is a character array.

```
for k = 1:3
x(k) = ischar(C{1,k});
end

x

x =

     0     1     0
```

See Also

`char`, `strings`, `isletter`, `isspace`, `isstrprop`, `iscellstr`, `isnumeric`, `isa`, `is*`

Purpose

Is input COM object

Syntax

```
tf = h.iscom
tf = iscom(h)
```

Description

`tf = h.iscom` returns logical `1` (`true`) if the input handle, `h`, is a COM or ActiveX object. Otherwise, `iscom` returns logical `0` (`false`) .

`tf = iscom(h)` is an alternate syntax for the same operation.

Examples

Create a COM server running Microsoft Excel. The `actxserver` function returns a handle `h` to the server object. Testing this handle with `iscom` returns `true`:

```
h = actxserver('excel.application');

h.iscom
ans =
   1
```

Create an interface to `workbooks`, returning handle `w`. Testing this handle with `iscom` returns `false`:

```
w = h.get('workbooks');

w.iscom
ans =
   0
```

See Also

`isinterface`

isdir

Purpose

Determine whether input is a directory

Syntax

```
tf = isdir('A')
```

Description

`tf = isdir('A')` returns logical `1` (`true`) if `A` is a directory and logical `0` (`false`) otherwise.

Examples

Type

```
tf=isdir('mymfiles/results')
```

and MATLAB returns

```
tf =
 1
```

indicating that `mymfiles/results` is a directory.

See Also

`dir`, `is*`

Purpose

Determine whether array is empty

Syntax

```
TF = isempty(A)
```

Description

`TF = isempty(A)` returns logical 1 (`true`) if `A` is an empty array and logical 0 (`false`) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

Examples

```
B = rand(2,2,2);
B(:,:,:) = [];

isempty(B)

ans = 1
```

See Also

`is*`

isempty (timeseries)

Purpose

Determine whether `timeseries` object is empty

Syntax

```
isempty(ts)
```

Description

`isempty(ts)` returns a logical value for `timeseries` object `ts`, as follows:

- `1` — When `ts` contains no data samples or `ts.Data` is empty.
- `0` — When `ts` contains data samples

See Also

`length (timeseries)`, `size (timeseries)`, `timeseries`, `tsprops`

Purpose

Determine whether `tscollection` object is empty

Syntax

```
isempty(tsc)
```

Description

`isempty(tsc)` returns a logical value for `tscollection` object `tsc`, as follows:

- `1` — When `tsc` contains neither `timeseries` members nor a time vector
- `0` — When `tsc` contains either `timeseries` members or a time vector

See Also

`length (tscollection)`, `size (tscollection)`, `timeseries`, `tscollection`

isequal

Purpose

Test arrays for equality

Syntax

```
tf = isequal(A, B, ...)
```

Description

`tf = isequal(A, B, ...)` returns logical `1` (`true`) if the input arrays have the same contents, and logical `0` (`false`) otherwise. Nonempty arrays must be of the same data type and size.

Remarks

When comparing structures, the order in which the fields of the structures were created is not important. As long as the structures contain the same fields, with corresponding fields set to equal values, `isequal` considers the structures to be equal. See Example 2, below.

When comparing numeric values, `isequal` does not consider the data type used to store the values in determining whether they are equal. See Example 3, below.

`NaN`s (Not a Number), by definition, are not equal. Therefore, arrays that contain `NaN` elements are not equal, and `isequal` returns zero when comparing such arrays. See Example 4, below. Use the `isequalwithequalnans` function when you want to test for equality with `NaN`s treated as equal.

`isequal` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequal` returns logical `1`.

Examples

Example 1

Given

```
A =                B =                C =
     1     0            1     0            1     0
     0     1            0     1            0     0
```

`isequal(A,B,C)` returns `0`, and `isequal(A,B)` returns `1`.

Example 2

When comparing structures with `isequal`, the order in which the fields of the structures were created is not important:

```
A.f1 = 25;    A.f2 = 50
A =
    f1: 25
    f2: 50

B.f2 = 50;    B.f1 = 25
B =
    f2: 50
    f1: 25

isequal(A, B)
ans =
     1
```

Example 3

When comparing numeric values, the data types used to store the values are not important:

```
A = [25 50];   B = [int8(25) int8(50)];

isequal(A, B)
ans =
     1
```

Example 4

Arrays that contain `NaN` (Not a Number) elements cannot be equal, since `NaN`s, by definition, are not equal:

```
A = [32 8 -29 NaN 0 5.7];
B = A;

isequal(A, B)
ans =
```

0

See Also isequalwithequalnans, strcmp, isa, is*, relational operators

Purpose

Compare `MException` objects for equality

Syntax

```
TF = isequal(eObj1, eObj2)
```

Description

`TF = isequal(eObj1, eObj2)` tests `MException` objects `eObj1` and `eObj2` for equality, returning logical 1 (`true`) if the two objects are identical, otherwise returning logical 0 (`false`).

See Also

`try`, `catch`, `error`, `assert`, `MException`, `eq(MException)`, `ne(MException)`, `getReport(MException)`, `disp(MException)`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `last(MException)`,

isequalwithequalnans

Purpose

Test arrays for equality, treating NaNs as equal

Syntax

```
tf = isequalwithequalnans(A, B, ...)
```

Description

`tf = isequalwithequalnans(A, B, ...)` returns logical `1` (`true`) if the input arrays are the same type and size and hold the same contents, and logical `0` (`false`) otherwise. `NaN` (Not a Number) values are considered to be equal to each other. Numeric data types and structure field order do not have to match.

Remarks

`isequalwithequalnans` is the same as `isequal`, except `isequalwithequalnans` considers `NaN` (Not a Number) values to be equal, and `isequal` does not.

`isequalwithequalnans` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequalwithequalnans` returns logical `1`.

Examples

Arrays containing `NaN`s are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider `NaN`s to be equal, while `isequalwithequalnans` does.

```
A = [32 8 -29 NaN 0 5.7];
B = A;
isequal(A, B)
ans =
     0

isequalwithequalnans(A, B)
ans =
     1
```

The position of `NaN` elements in the array does matter. If they are not in the same position in the arrays being compared, then `isequalwithequalnans` returns zero.

```
A = [2 4 6 NaN 8];   B = [2 4 NaN 6 8];
```

```
isequalwithequalnans(A, B)
ans =
     0
```

See Also

isequal, strcmp, isa, is*, relational operators

isevent

Purpose

Is input event

Syntax

```
tf = h.isevent('name')
tf = isevent(h, 'name')
```

Description

`tf = h.isevent('name')` returns logical `1` (`true`) if the specified `name` is an event that can be recognized and responded to by object `h`. Otherwise, `isevent` returns logical `0` (`false`) .

`tf = isevent(h, 'name')` is an alternate syntax for the same operation.

Remarks

The string specified in the `name` argument is not case sensitive.

For COM control objects, `isevent` returns the same value regardless of whether the specified event is registered with the control or not. In order for the control to respond to the event, you must first register the event using either `actxcontrol` or `registerevent`.

Examples

Test an Event Example

Create an `mwsamp` control and test to see if `DblClick` is an event recognized by the control.

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.isevent('DblClick')
```

MATLAB displays `ans = 1` (`true`), showing that `DblClick` is an event.

Test a Method Example

Try the same test on `Redraw`, which is one of the control's methods.

```
h.isevent('Redraw')
```

MATLAB displays `ans = 0` (`false`), showing that `Redraw` is not an event.

Test an Excel Workbook Example

Create an Excel Workbook object.

```
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
```

Test the `Activate` event:

```
wb.isevent('Activate')
```

MATLAB displays `ans = 1` (`true`), showing that `Activate` is an event.

Test `Save` :

```
wb.isevent('Save')
```

MATLAB displays `ans = 0` (`false`), showing that `Save` is not an event; it is a method.

See Also

`events`, `eventlisteners`, `registerevent`, `unregisterevent`, `unregisterallevents`

isfield

Purpose

Determine whether input is structure array field

Syntax

```
tf = isfield(S, 'fieldname')
tf = isfield(S, C)
```

Description

`tf = isfield(S, 'fieldname')` examines structure `S` to see if it includes the field specified by the quoted string `'fieldname'`. Output `tf` is set to logical `1` (`true`) if `S` contains the field, or logical `0` (`false`) if not. If `S` is not a structure array, `isfield` returns `false`.

`tf = isfield(S, C)` examines structure `S` for multiple fieldnames as specified in cell array of strings `C`, and returns an array of logical values to indicate which of these fields are part of the structure. Elements of output array `tf` are set to a logical 1 (`true`) if the corresponding element of `C` holds a fieldname that belongs to structure `S`. Otherwise, logical 0 (`false`) is returned in that element. In other words, if structure `S` contains the field specified in `C{m,n}`, `isfield` returns a logical 1 (`true`) in `tf(m,n)`.

Note `isfield` returns `false` if the `field` or `fieldnames` input is empty.

Examples

Example 1 — Single Fieldname Syntax

Given the following MATLAB structure,

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

`isfield` identifies `billing` as a field of that structure.

```
isfield(patient,'billing')
ans =
     1
```

Example 2 — Multiple Fieldname Syntax

Check structure `S` for any of four possible fieldnames. Only the first is found, so the first element of the return value is set to `true`:

```
S = struct('one', 1, 'two', 2);

fields = isfield(S, {'two', 'pi', 'One', 3.14})
fields =
     1     0     0     0
```

See Also

`fieldnames`, `setfield`, `getfield`, `orderfields`, `rmfield`, `struct`, `isstruct`, `iscell`, `isa`, `is*`, dynamic field names

isfinite

Purpose

Array elements that are finite

Syntax

```
TF = isfinite(A)
```

Description

`TF = isfinite(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of the array `A` are finite and logical 0 (`false`) where they are infinite or `NaN`. For a complex number `z`, `isfinite(z)` returns `1` if both the real and imaginary parts of `z` are finite, and `0` if either the real or the imaginary part is infinite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2];

isfinite(1./a)
Warning: Divide by zero.

ans =
     1     1     0     1     1

isfinite(0./a)
Warning: Divide by zero.

ans =
     1     1     0     1     1
```

See Also

`isinf`, `isnan`, `is*`

Purpose

Determine whether input is floating-point array

Syntax

```
isfloat(A)
```

Description

`isfloat(A)` returns a logical 1 (`true`) if A is a floating-point array and a logical 0 (`false`) otherwise. The only floating-point data types in MATLAB are `single` and `double`.

See Also

`isa`, `isinteger`, `double`, `single`, `isnumeric`

isglobal

Purpose

Determine whether input is global variable

Note Support for the `isglobal` function will be removed in a future release of MATLAB. See Remarks below.

Syntax

```
tf = isglobal(A)
```

Description

`tf = isglobal(A)` returns logical `1` (`true`) if `A` has been declared to be a global variable in the context from which `isglobal` is called, and logical `0` (`false`) otherwise.

Remarks

`isglobal` is most commonly used in conjunction with conditional global declaration. An alternate approach is to use a pair of variables, one local and one declared global.

Instead of using

```
if condition
   global x
end

x = some_value

if isglobal(x)
   do_something
end
```

You can use

```
global gx
if condition
   gx = some_value
else
   x = some_value
end
```

```
if condition
   do_something
end
```

If no other workaround is possible, you can replace the command

```
isglobal(variable)
```

with

```
~isempty(whos('global','variable'))
```

See Also

`global`, `isvarname`, `isa`, `is*`

ishandle

Purpose

Is object handle valid

Syntax

```
ishandle(h)
```

Description

`ishandle(h)` returns an array containing `1`'s where the elements of `h` are valid graphics handles and `0`'s where they are not.

See Also

`findobj`, `gca`, `gcf`, `gco`, `set`

"Accessing Object Handles" for more information.

"Finding and Identifying Graphics Objects" on page 1-93 for related functions

Purpose

Current hold state

Syntax

```
ishold
```

Description

`ishold` returns `1` if `hold` is `on`, and `0` if it is `off`. When `hold` is `on`, the current plot and most axis properties are held so that subsequent graphing commands add to the existing graph.

A state of `hold on` implies that both figure and axes `NextPlot` properties are set to `add`.

See Also

`hold`, `newplot`

"Controlling Graphics Output" for related information

"Axes Operations" on page 1-96 for related functions

isinf

Purpose

Array elements that are infinite

Syntax

```
TF = isinf(A)
```

Description

`TF = isinf(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `+Inf` or `-Inf` and logical 0 (`false`) where they are not. For a complex number `z`, `isinf(z)` returns `1` if either the real or imaginary part of `z` is infinite, and `0` if both the real and imaginary parts are finite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2]

isinf(1./a)
Warning: Divide by zero.

ans =
     0     0     1     0     0

isinf(0./a)
Warning: Divide by zero.

ans =
     0     0     0     0     0
```

See Also

`isfinite`, `isnan`, `is*`

Purpose

Determine whether input is integer array

Syntax

Description

`isinteger(A)` returns a logical 1 (`true`) if the array `A` has integer data type and a logical 0 (`false`) otherwise. The integer data types in MATLAB are

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `int64`
- `uint64`

See Also

`isa`, `isnumeric`, `isfloat`

isinterface

Purpose

Is input COM interface

Syntax

```
tf = h.isinterface
tf = isinterface(h)
```

Description

`tf = h.isinterface` returns logical `1` (`true`) if the input handle, `h`, is a COM interface. Otherwise, `isinterface` returns logical `0` (`false`) .

`tf = isinterface(h)` is an alternate syntax for the same operation.

Examples

Create a COM server running Microsoft Excel. The `actxserver` function returns a handle `h` to the server object. Testing this handle with `isinterface` returns `false`:

```
h = actxserver('excel.application');

h.isinterface
ans =
   0
```

Create an interface to `workbooks`, returning handle `w`. Testing this handle with `isinterface` returns `true`:

```
w = h.get('workbooks');

w.isinterface
ans =
   1
```

See Also

`iscom`, `interfaces`, `get`

Purpose

Determine whether input is Java object

Syntax

```
tf = isjava(A)
```

Description

`tf = isjava(A)` returns logical `1` (`true`) if `A` is a Java object, and logical `0` (`false`) otherwise.

Examples

Create an instance of the Java Frame class and `isjava` indicates that it is a Java object.

```
frame = java.awt.Frame('Frame A');

isjava(frame)

ans =

     1
```

Note that, `isobject`, which tests for MATLAB objects, returns logical `0` (`false`).

```
isobject(frame)

ans =

     0
```

See Also

`isobject`, `javaArray`, `javaMethod`, `javaObject`, `isa`, `is*`

iskeyword

Purpose

Determine whether input is MATLAB keyword

Syntax

```
tf = iskeyword('str')
iskeyword str
iskeyword
```

Description

`tf = iskeyword('str')` returns logical `1` (`true`) if the string `str` is a keyword in the MATLAB language and logical `0` (`false`) otherwise.

`iskeyword str` uses the MATLAB command format.

`iskeyword` returns a list of all MATLAB keywords.

Examples

To test if the word `while` is a MATLAB keyword,

```
iskeyword while
ans =
    1
```

To obtain a list of all MATLAB keywords,

```
iskeyword
    'break'
    'case'
    'catch'
    'classdef'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'parfor'
    'persistent'
    'return'
```

```
        'switch'
        'try'
        'while'
```

See Also

isvarname, genvarname, is*

isletter

Purpose

Array elements that are alphabetic letters

Syntax

```
tf = isletter('str')
```

Description

`tf = isletter('str')` returns an array the same size as `str` containing logical `1` (`true`) where the elements of `str` are letters of the alphabet and logical `0` (`false`) where they are not.

Examples

Find the letters in character array `s`.

```
s = 'A1,B2,C3';

isletter(s)
ans =
     1     0     0     1     0     0     1     0
```

See Also

`ischar`, `isspace`, `isstrprop`, `iscellstr`, `isnumeric`, `char`, `strings`, `isa`, `is*`

Purpose

Determine whether input is logical array

Syntax

```
tf = islogical(A)
```

Description

`tf = islogical(A)` returns logical `1` (`true`) if `A` is a logical array and logical `0` (`false`) otherwise.

Examples

Given the following cell array,

```
C{1,1} = pi;                % double
C{1,2} = 1;                 % double
C{1,3} = ispc;              % logical
C{1,4} = magic(3)           % double array

C =
    [3.1416]    [1]    [1]    [3x3 double]
```

`islogical` shows that only `C{1,3}` is a logical array.

```
for k = 1:4
x(k) = islogical(C{1,k});
end

x
x =
     0     0     1     0
```

See Also

`logical`, `isnumeric`, `ischar`, `isreal`, , logical operators (`elementwise` and `short-circuit`), `isa`, `is*`

ismac

Purpose

Determine whether running Macintosh OS X versions of MATLAB

Syntax

```
tf = ismac
```

Description

`tf = ismac` returns logical `1` (`true`) for the Macintosh OS X versions of MATLAB and logical `0` (`false`) otherwise.

See Also

`isunix`, `ispc`, `isstudent`, `is*`

Purpose

Array elements that are members of set

Syntax

```
tf = ismember(A, S)
tf = ismember(A, S, 'rows')
[tf, loc] = ismember(A, S, ...)
```

Description

`tf = ismember(A, S)` returns a vector the same length as `A`, containing logical `1` (`true`) where the elements of `A` are in the set `S`, and logical `0` (`false`) elsewhere. In set theory terms, `k` is 1 where `A` ϵ `S`. Inputs `A` and `S` can be numeric or character arrays or cell arrays of strings.

`tf = ismember(A, S, 'rows')`, when `A` and `S` are matrices with the same number of columns, returns a vector containing `1` where the rows of `A` are also rows of `S` and `0` otherwise. You cannot use this syntax if `A` or `S` is a cell array of strings.

`[tf, loc] = ismember(A, S, ...)` returns an array `loc` containing the highest index in `S` for each element in `A` that is a member of `S`. For those elements of `A` that do not occur in `S`, `ismember` returns `0`.

Remarks

Because NaN is considered to be not equal to anything, it is never a member of any set.

Examples

```
set = [0 2 4 6 8 10 12 14 16 18 20];
a = reshape(1:5, [5 1])

a =
     1
     2
     3
     4
     5

ismember(a, set)
ans =
 0
     1
```

```
     0
     1
     0
set = [5 2 4 2 8 10 12 2 16 18 20 3];
[tf, index] = ismember(a, set);

index
index =
     0
     8
    12
     3
     1
```

See Also

issorted, intersect, setdiff, setxor, union, unique, is*

Purpose

Determine whether input is object method

Syntax

```
ismethod(h, 'name')
```

Description

`ismethod(h, 'name')` returns a logical `1` (`true`) if the specified `name` is a method that you can call on object `h`. Otherwise, `ismethod` returns logical `0` (`false`).

Examples

Create an Excel application and test to see if `SaveWorkspace` is a method of the object. `ismethod` returns `true`:

```
h = actxserver ('Excel.Application');

ismethod(h, 'SaveWorkspace')
ans =
     1
```

Try the same test on `UsableWidth`, which is a property. `ismethod` returns `false`:

```
ismethod(h, 'UsableWidth')
ans =
     0
```

See Also

`methods`, `methodsview`, `isprop`, `isevent`, `isobject`, `class`, `invoke`

isnan

Purpose

Array elements that are NaN

Syntax

```
TF = isnan(A)
```

Description

`TF = isnan(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `NaN`s and logical 0 (`false`) where they are not. For a complex number `z`, `isnan(z)` returns `1` if either the real or imaginary part of `z` is `NaN`, and `0` if both the real and imaginary parts are finite or `Inf`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2]

isnan(1./a)
Warning: Divide by zero.

ans =
     0     0     0     0     0

isnan(0./a)
Warning: Divide by zero.

ans =
     0     0     1     0     0
```

See Also

`isfinite`, `isinf`, `is*`

Purpose

Determine whether input is numeric array

Syntax

```
tf = isnumeric(A)
```

Description

`tf = isnumeric(A)` returns logical `1` (`true`) if `A` is a numeric array and logical `0` (`false`) otherwise. For example, sparse arrays and double-precision arrays are numeric, while strings, cell arrays, and structure arrays and logicals are not.

Examples

Given the following cell array,

```
C{1,1} = pi;                % double
C{1,2} = 'John Doe';        % char array
C{1,3} = 2 + 4i;            % complex double
C{1,4} = ispc;              % logical
C{1,5} = magic(3)           % double array

C =
   [3.1416] 'John Doe' [2.0000+ 4.0000i] [1][3x3 double]
```

`isnumeric` shows that all but `C{1,2}` and `C{1,4}` are numeric arrays.

```
for k = 1:5
x(k) = isnumeric(C{1,k});
end

x
x =
     1     0     1     0     1
```

See Also

`isstrprop`, `isnan`, `isreal`, `isprime`, `isfinite`, `isinf`, `isa`, `is*`

isobject

Purpose

Determine whether input is MATLAB OOPs object

Syntax

```
tf = isobject(A)
```

Description

`tf = isobject(A)` returns logical `1` (`true`) if `A` is a MATLAB object and logical `0` (`false`) otherwise.

Examples

Create an instance of the `polynom` class as defined in the section "Example — A Polynomial Class" in the MATLAB Programming documentation.

```
p = polynom([1 0 -2 -5])
p =
   x^3 - 2*x - 5
```

`isobject` indicates that `p` is a MATLAB object.

```
isobject(p)
ans =
     1
```

Note that `isjava`, which tests for Java objects in MATLAB, returns `false`.

```
isjava(p)
ans =
     0
```

See Also

`isjava`, `isstruct`, `iscell`, `ischar`, `isnumeric`, `islogical`, `ismethod`, `isprop`, `isevent`, `methods`, `class`, `isa`, is*

Purpose

Compute isosurface end-cap geometry

Syntax

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

Description

`fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end-cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end-caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'enclose')` specifies whether the end-caps enclose data values above or below the value specified in `isovalue`. The string *`enclose`* can be either `above` (default) or `below`.

`fvc = isocaps(...,'whichplane')` specifies on which planes to draw the end-caps. Possible values for *`whichplane`* are `all` (default), `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, or `zmax`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end-caps in three arrays instead of the struct `fvc`.

`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

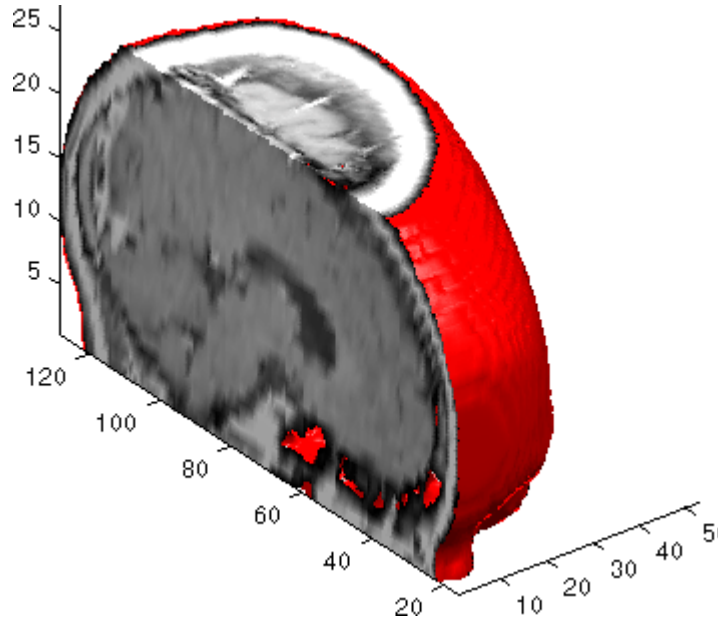
Examples

This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of `isocaps` to draw the end-caps on this cutaway volume.

The red `isosurface` shows the outline of the volume (skull) and the end-caps show what is inside of the volume.

The `patch` created from the end-cap data (`p2`) uses interpolated face coloring, which means the gray `colormap` and the light sources determine how it is colored. The isosurface patch (`p1`) used a flat red face color, which is affected by the lights, but does not use the colormap.

```
load mri
D = squeeze(D);
D(:,1:60,:) = [];
p1 = patch(isosurface(D, 5),'FaceColor','red',...
 'EdgeColor','none');
p2 = patch(isocaps(D, 5),'FaceColor','interp',...
 'EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight left; camlight; lighting gouraud
isonormals(D,p1)
```



See Also

`isosurface`, `isonormals`, `smooth3`, `subvolume`, `reducevolume`, `reducepatch`

"Isocaps Add Context to Visualizations" for more illustrations of isocaps

"Volume Visualization" on page 1-102 for related functions

isocolors

Purpose

Calculate isosurface and patch colors

Syntax

```
nc = isocolors(X,Y,Z,C,vertices)
nc = isocolors(X,Y,Z,R,G,B,vertices)
nc = isocolors(C,vertices)
nc = isocolors(R,G,B,vertices)
nc = isocolors(...,PatchHandle)
isocolors(...,PatchHandle)
```

Description

`nc = isocolors(X,Y,Z,C,vertices)` computes the colors of isosurface (patch object) vertices (`vertices`) using color values `C`. Arrays `X`, `Y`, `Z` define the coordinates for the color data in `C` and must be monotonic vectors or 3-D plaid arrays (as if produced by `meshgrid`). The colors are returned in `nc`. `C` must be 3-D (index colors).

`nc = isocolors(X,Y,Z,R,G,B,vertices)` uses `R`, `G`, `B` as the red, green, and blue color arrays (true color).

`nc = isocolors(C,vertices)`, and `nc = isocolors(R,G,B,vertices)` assume `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

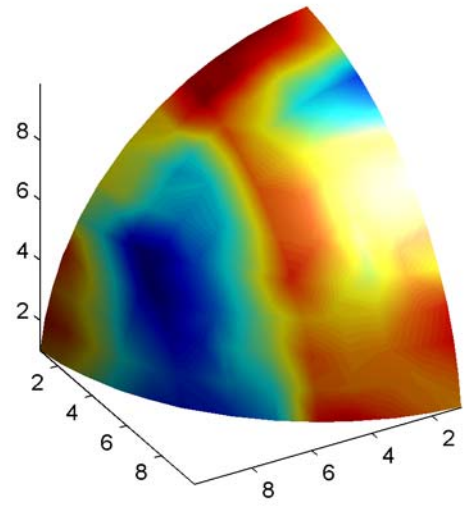
Examples

Indexed Color Data

This example displays an isosurface and colors it with random data using indexed color. (See “Interpolating in Indexed Color Versus Truecolor” for information on how patch objects interpret color data.)

```
[x y z] = meshgrid(1:20,1:20,1:20);
```

```
data = sqrt(x.^2 + y.^2 + z.^2);
cdata = smooth3(rand(size(data)),'box',7);
p = patch(isosurface(x,y,z,data,10));
isonormals(x,y,z,data,p);
isocolors(x,y,z,cdata,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);axis tight
camlight; lighting phong;
```

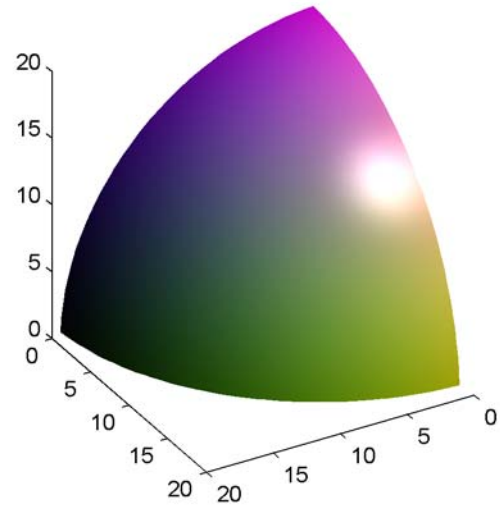


True Color Data

This example displays an isosurface and colors it with true color (RGB) data.

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
```

```
isocolors(x,y,z,r/20,g/20,b/20,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```

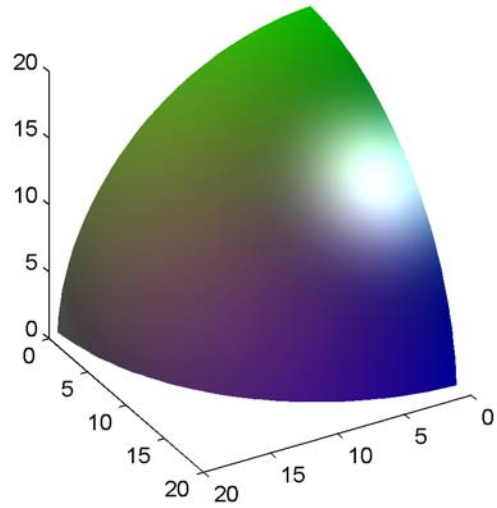


Modified True Color Data

This example uses `isocolors` to calculate the true color data using the isosurface's (patch object's) vertices, but then returns the color data in a variable (`c`) in order to modify the values. It then explicitly sets the isosurface's `FaceVertexCData` to the new data (`1-c`).

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(data,20));
isonormals(data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
c = isocolors(r/20,g/20,b/20,p);
set(p,'FaceVertexCData',1-c)
```

```
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```



See Also

`isosurface`, `isocaps`, `smooth3`, `subvolume`, `reducevolume`, `reducepatch`, `isonormals`

"Volume Visualization" on page 1-102 for related functions

isonormals

Purpose

Compute normals of isosurface vertices

Syntax

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p) and n = isonormals(X,Y,Z,V,p)
n = isonormals(...,'negate')
isonormals(V,p) and isonormals(X,Y,Z,V,p)
```

Description

`n = isonormals(X,Y,Z,V,vertices)` computes the normals of the isosurface vertices from the vertex list, `vertices`, using the gradient of the data `V`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The computed normals are returned in `n`.

`n = isonormals(V,vertices)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`n = isonormals(V,p) and n = isonormals(X,Y,Z,V,p)` compute normals from the vertices of the patch identified by the handle `p`.

`n = isonormals(...,'negate')` negates (reverses the direction of) the normals.

`isonormals(V,p) and isonormals(X,Y,Z,V,p)` set the `VertexNormals` property of the patch identified by the handle `p` to the computed normals rather than returning the values.

Examples

This example compares the effect of different surface normals on the visual appearance of lit isosurfaces. In one case, the triangles used to draw the isosurface define the normals. In the other, the `isonormals` function uses the volume data to calculate the vertex normals based on the gradient of the data points. The latter approach generally produces a smoother-appearing isosurface.

Define a 3-D array of volume data (`cat`, `interp3`):

```
data = cat(3, [0 .2 0; 0 .3 0; 0 0 0], ...
              [.1 .2 0; 0 1 0; .2 .7 0],...
              [0 .4 .2; .2 .4 0;.1 .1 0]);
```

```
data = interp3(data,3,'cubic');
```

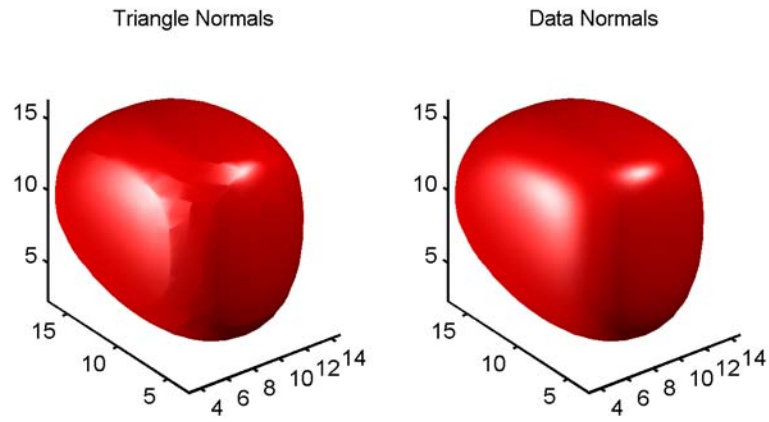
Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals (`patch`, `isosurface`, `view`, `daspect`, `axis`, `camlight`, `lighting`, `title`):

```
subplot(1,2,1)
p1 = patch(isosurface(data,.5),...
'FaceColor','red','EdgeColor','none');
view(3); daspect([1,1,1]); axis tight
camlight; camlight(-80,-10); lighting phong;
title('Triangle Normals')
```

Draw the same lit isosurface using normals calculated from the volume data:

```
subplot(1,2,2)
p2 = patch(isosurface(data,.5),...
    'FaceColor','red','EdgeColor','none');
isonormals(data,p2)
view(3); daspect([1 1 1]); axis tight
camlight;  camlight(-80,-10); lighting phong;
title('Data Normals')
```

These isosurfaces illustrate the difference between triangle and data normals:



See Also

`interp3`, `isosurface`, `isocaps`, `smooth3`, `subvolume`, `reducevolume`, `reducepatch`

"Volume Visualization" on page 1-102 for related functions

Purpose

Extract isosurface data from volume data

Syntax

```
fv = isosurface(X,Y,Z,V,isovalue)
fv = isosurface(V,isovalue)
fvc = isosurface(...,colors)
fv = isosurface(...,'noshare')
fv = isosurface(...,'verbose')
[f,v] = isosurface(...)
[f,v,c] = isosurface(...)
isosurface(...)
```

Description

`fv = isosurface(X,Y,Z,V,isovalue)` computes isosurface data from the volume data `V` at the isosurface value specified in `isovalue`. That is, the isosurface connects points that have the specified value much the way contour lines connect points of equal elevation.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The structure `fv` contains the faces and vertices of the isosurface, which you can pass directly to the `patch` command.

`fv = isosurface(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isosurface(...,colors)` interpolates the array `colors` onto the scalar field and returns the interpolated values in the `facevertexcdata` field of the `fvc` structure. The size of the `colors` array must be the same as `V`. The `colors` argument enables you to control the color mapping of the isosurface with data different from that used to calculate the isosurface (e.g., temperature data superimposed on a wind current isosurface).

`fv = isosurface(...,'noshare')` does not create shared vertices. This is faster, but produces a larger set of vertices.

`fv = isosurface(...,'verbose')` prints progress messages to the command window as the computation progresses.

`[f,v] = isosurface(...)` or `[f,v,c] = isosurface(...)` returns the faces and vertices (and `faceVertexcCData`) in separate arrays instead of a struct.

`isosurface(...)` with no output arguments, creates a patch in the current axes with the computed faces and vertices. If no current axes exists, a new axes is created with a 3-D view and appropriate lighting.

Special Case Behavior — isosurface Called with No Output Arguments

If there is no current axes and you call `isosurface` with without assigning output arguments, MATLAB creates a new axes, sets it to a 3-D view, and adds lighting to the isosurface graph.

Remarks

You can pass the `fv` structure created by `isosurface` directly to the `patch` command, but you cannot pass the individual faces and vertices arrays (`f`, `v`) to `patch` without specifying property names. For example,

```
patch(isosurface(X,Y,Z,V,isovalue))
```

or

```
[f,v] = isosurface(X,Y,Z,V,isovalue);
patch('Faces',f,'Vertices',v)
```

Examples

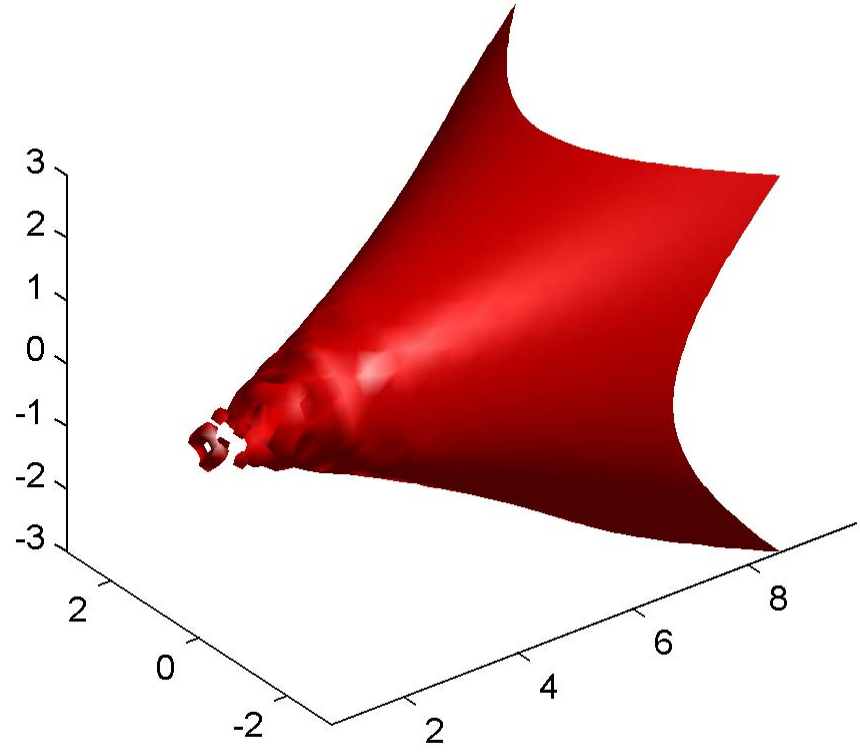
Example 1

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). The isosurface is drawn at the data value of -3. The statements that follow the `patch` command prepare the isosurface for lighting by

- Recalculating the isosurface normals based on the volume data (`isonormals`)
- Setting the face and edge color (`set`, `FaceColor`, `EdgeColor`)
- Specifying the view (`daspect`, `view`)

- Adding lights (`camlight`, `lighting`)

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
set(p,'FaceColor','red','EdgeColor','none');
daspect([1 1 1])
view(3); axis tight
camlight
lighting gouraud
```

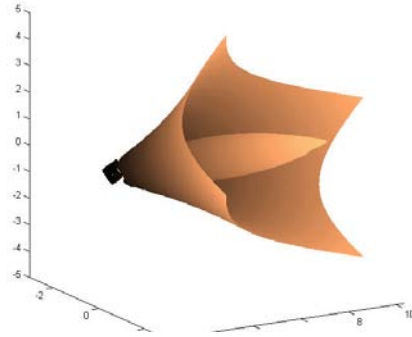


Example 2

Visualize the same flow data as above, but color-code the surface to indicate magnitude along the X-axis. Use a sixth argument to `isosurface`, which provides a means to overlay another data set by coloring the resulting isosurface. The `colors` variable is a vector containing a scalar value for each vertex in the isosurface, to be portrayed with the current color map. In this case, it is one of the

variables that define the surface, but it could be entirely independent. You can apply a different color scheme by changing the current figure color map.

```
[x,y,z,v] = flow;
[faces,verts,colors] = isosurface(x,y,z,v,-3,x);
patch('Vertices', verts, 'Faces', faces, ...
    'FaceVertexCData', colors, ...
    'FaceColor','interp', ...
    'edgecolor', 'interp');
view(30,-15);
axis vis3d;
colormap copper
```



See Also

`isonormals`, `shrinkfaces`, `smooth3`, `subvolume`

"Connecting Equal Values with Isosurfaces" for more examples

"Volume Visualization" on page 1-102 for related functions

ispc

Purpose

Determine whether PC (Windows) version of MATLAB

Syntax

```
tf = ispc
```

Description

`tf = ispc` returns logical `1` (`true`) for the PC version of MATLAB and logical `0` (`false`) otherwise.

See Also

`isunix`, `ismac`, `isstudent`, `is*`

Purpose

Test for existence of preference

Syntax

```
ispref('group','pref')
ispref('group')
ispref('group',{'pref1','pref2',...'prefn'})
```

Description

`ispref('group','pref')` returns `1` if the preference specified by `group` and `pref` exists, and `0` otherwise.

`ispref('group')` returns `1` if the GROUP exists, and `0` otherwise.

`ispref('group',{'pref1','pref2',...'prefn'})` returns a logical array the same length as the cell array of preference names, containing `1` where each preference exists, and `0` elsewhere.

Examples

```
addpref('mytoolbox','version','1.0')
ispref('mytoolbox','version')

ans =
     1.0
```

See Also

`addpref`, `getpref`, `rmpref`, `setpref`, `uigetpref`, `uisetpref`

isprime

Purpose

Array elements that are prime numbers

Syntax

```
TF = isprime(A)
```

Description

`TF = isprime(A)` returns an array the same size as `A` containing logical 1 (`true`) for the elements of `A` which are prime, and logical 0 (`false`) otherwise. `A` must contain only positive integers.

Examples

```
c = [2 3 0 6 10]

c =
     2     3     0     6    10

isprime(c)

ans =
     1     1     0     0     0
```

See Also

`is*`

Purpose

Determine whether input is object property

Syntax

```
isprop(h, 'name')
```

Description

`isprop(h, 'name')` returns logical `1` (`true`) if the specified `name` is a property you can use with object `h`. Otherwise, `isprop` returns logical `0` (`false`).

Examples

Create an Excel application and test to see if `UsableWidth` is a property of the object. `isprop` returns `true`:

```
h = actxserver ('Excel.Application');

isprop(h, 'UsableWidth')
ans =
     1
```

Try the same test on `SaveWorkspace`, which is a method, and `isprop` returns `false`:

```
isprop(h, 'SaveWorkspace')
ans =
     0
```

See Also

`get(COM)`, `inspect`, `addproperty`, `deleteproperty`, `ismethod`, `isevent`, `isobject`, `methods`, `class`

isreal

Purpose

Determine whether input is real array

Syntax

```
TF = isreal(A)
```

Description

`TF = isreal(A)` returns logical `0` (`false`) if any element of array `A` has an imaginary component, even if the value of that component is `0`. For `logical`, `char`, numeric, and function handle data types, `isreal` returns logical `1` (`true`) otherwise.

Note For `cell`, `struct`, and object data types, `isreal` also returns logical `0` (`false`).

`~isreal(x)` returns `true` for arrays that have at least one element with an imaginary component. The value of that component can be `0`.

Remarks

If `A` is real, `complex(A)` returns a complex number whose imaginary component is `0`, and `isreal(complex(A))` returns `false`. In contrast, the addition `A + 0i` returns the real value `A`, and `isreal(A + 0i)` returns `true`.

If `B` is real and `A = complex(B)`, then `A` is a complex matrix and `isreal(A)` returns `false`, while `A(m:n)` returns a real matrix and `isreal(A(m:n))` returns `true`.

Because MATLAB supports complex arithmetic, certain of its functions can introduce significant imaginary components during the course of calculations that appear to be limited to real numbers. Thus, you should use `isreal` with discretion.

Examples

Example 1

These examples use `isreal` to detect the presence or absence of imaginary numbers in an array. Let

```
x = magic(3);
y = complex(x);
```

`isreal(x)` returns `true` because no element of `x` has an imaginary component.

```
isreal(x)
ans =
     1
```

`isreal(y)` returns `false`, because every element of `x` has an imaginary component, even though the value of the imaginary components is `0`.

```
isreal(y)
ans =
     0
```

This expression detects strictly real arrays, i.e., elements with `0`-valued imaginary components are treated as real.

```
~any(imag(y(:)))
ans =
     1
```

Example 2

Given the following cell array,

```
C{1,1} = pi;                % double
C{1,2} = 'John Doe';        % char array
C{1,3} = 2 + 4i;            % complex double
C{1,4} = ispc;              % logical
C{1,5} = magic(3)           % double array
C{1,6} = complex(5,0)       % complex double

C =
  [3.1416]  'John Doe'  [2.0000+ 4.0000i]  [1]  [3x3 double]  [5]
```

`isreal` shows that all but `C{1,3}` and `C{1,6}` are real arrays.

```
for k = 1:6
x(k) = isreal(C{1,k});
end
```

```
x
x =
     1     1     0     1     1     0
```

See Also

complex, isnumeric, isnan, isprime, isfinite, isinf, isa, is*

Purpose

Determine whether input is scalar

Syntax

```
TF = isscalar(A)
```

Description

`TF = isscalar(A)` returns logical `1` (`true`) if `A` is a 1-by-1 matrix, and logical `0` (`false`) otherwise.

The `A` argument can be a structure or cell array. It also be a MATLAB object, as described in “Classes and Objects”, as long as that object overloads the `size` function.

Examples

Test matrix `A` and one element of the matrix:

```
A = rand(5);

isscalar(A)
ans =
     0

isscalar(A(3,2))
ans =
     1
```

See Also

`isvector`, `isempty`, `isnumeric`, `islogical`, `ischar`, `isa`, `is*`

issorted

Purpose

Determine whether set elements are in sorted order

Syntax

```
TF = issorted(A)
TF = issorted(A, 'rows')
```

Description

`TF = issorted(A)` returns logical `1` (`true`) if the elements of `A` are in sorted order, and logical `0` (`false`) otherwise. Input `A` can be a vector or an N-by-1 or 1-by-N cell array of strings. `A` is considered to be sorted if `A` and the output of `sort(A)` are equal.

`TF = issorted(A, 'rows')` returns logical `1` (`true`) if the rows of two-dimensional matrix `A` are in sorted order, and logical `0` (`false`) otherwise. Matrix `A` is considered to be sorted if `A` and the output of `sortrows(A)` are equal.

Note Only the `issorted(A)` syntax supports `A` as a cell array of strings.

Remarks

For character arrays, `issorted` uses ASCII, rather than alphabetical, order.

You cannot use `issorted` on arrays of greater than two dimensions.

Examples

Example 1 — Using issorted on a vector

```
A = [5 12 33 39 78 90 95 107 128 131];

issorted(A)
ans =
     1
```

Example 2 — Using issorted on a matrix

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
```

```
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

issorted(A, 'rows')
ans =
     0

B = sortrows(A)
B =
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
    17    24     1     8    15
    23     5     7    14    16

issorted(B)
ans =
     1
```

Example 3 — Using issorted on a cell array

```
x = {'one'; 'two'; 'three'; 'four'; 'five'};
issorted(x)
ans =
     0

y = sort(x)
y =
    'five'
    'four'
    'one'
    'three'
    'two'

issorted(y)
```

See Also

sort, sortrows, ismember, unique, intersect, union, setdiff, setxor, is*

Purpose

Array elements that are space characters

Syntax

```
tf = isspace('str')
```

Description

`tf = isspace('str')` returns an array the same size as `'str'` containing logical `1` (`true`) where the elements of `str` are ASCII white spaces and logical `0` (`false`) where they are not. White spaces in ASCII are space, newline, carriage return, tab, vertical tab, or formfeed characters.

Examples

```
isspace('  Find spa ces ')
  Columns 1 through 13
     1   1   0   0   0   0   1   0   0   0   1   0   0
  Columns 14 through 15
     0   1
```

See Also

`isletter`, `isstrprop`, `ischar`, `strings`, `isa`, `is*`

issparse

Purpose

Determine whether input is sparse

Syntax

```
TF = issparse(S)
```

Description

`TF = issparse(S)` returns logical 1 (`true`) if the storage class of `S` is sparse and logical 0 (`false`) otherwise.

See Also

`is*`, `sparse`, `full`

Purpose

Determine whether input is character array

Note Use the `ischar` function in place of `isstr`. The `isstr` function will be removed in a future version of MATLAB.

See Also

`ischar`, `isa`, `is*`

isstrprop

Purpose

Determine whether string is of specified category

Syntax

```
tf = isstrprop('str', 'category')
```

Description

`tf = isstrprop('str', 'category')` returns a logical array the same size as `str` containing logical `1` (`true`) where the elements of `str` belong to the specified *`category`*, and logical `0` (`false`) where they do not.

The `str` input can be a character array, cell array, or any MATLAB numeric type. If `str` is a cell array, then the return value is a cell array of the same shape as `str`.

The *`category`* input can be any of the strings shown in the left column below:

Category	Description
`alpha`	`True` for those elements of `str` that are alphabetic
`alphanum`	`True` for those elements of `str` that are alphanumeric
`cntrl`	`True` for those elements of `str` that are control characters (for example, `char(0:20)`)
`digit`	`True` for those elements of `str` that are numeric digits

`graphic`	True for those elements of str that are graphic characters. These are all values that represent any characters except for the following:
	
	`unassigned, space, line separator,`
	`paragraph separator, control characters,`
	`Unicode format control characters,`
	`private user-defined characters,`
	`Unicode surrogate characters,`
	`Unicode other characters`

`lower`	`True` for those elements of `str` that are lowercase letters
`print`	`True` for those elements of `str` that are graphic characters, plus `char(32)`

Category	Description
`punct`	`True` for those elements of `str` that are punctuation characters
`wspace`	`True` for those elements of `str` that are white-space characters. This range includes the ANSI C definition of white space, `{' ','\t','\n','\r','\v','\f'}`.
`upper`	`True` for those elements of `str` that are uppercase letters
`xdigit`	`True` for those elements of `str` that are valid hexadecimal digits

Remarks

Numbers of type `double` are converted to `int32` according to MATLAB rules of double-to-integer conversion. Numbers of type `int64` and `uint64` bigger than `int32(inf)` saturate to `int32(inf)`.

MATLAB classifies the elements of the `str` input according to the Unicode definition of the specified category. If the numeric value of an element in the input array falls within the range that defines a Unicode character category, then this element is classified as being of that category. The set of Unicode character codes includes the set of ASCII character codes, but also covers a large number of languages beyond the scope of the ASCII set. The classification of characters is dependent on the global location of the platform on which MATLAB is installed.

Examples

Test for alphabetic characters in a string:

```
A = isstrprop('abc123def', 'alpha')
A =
   1 1 1 0 0 0 1 1 1
```

Test for numeric digits in a string:

```
A = isstrprop('abc123def', 'digit')
A =
   0 0 0 1 1 1 0 0 0
```

Test for hexadecimal digits in a string:

```
A = isstrprop('abcd1234efgh', 'xdigit')
A =
   1 1 1 1 1 1 1 1 1 1 0 0
```

Test for numeric digits in a character array:

```
A = isstrprop(char([97 98 99 49 50 51 101 102 103]), ...
              'digit')
A =
   0 0 0 1 1 1 0 0 0
```

Test for alphabetic characters in a two-dimensional cell array:

```
A = isstrprop({'abc123def';'456ghi789'}, 'alpha')
A =
   [1x9 logical]
   [1x9 logical]

A{:,:}
ans =
   1 1 1 0 0 0 1 1 1
   0 0 0 1 1 1 0 0 0
```

Test for white-space characters in a string:

```
A = isstrprop(sprintf('a bc\n'), 'wspace')
A =
   0 1 0 0 1
```

See Also

`strings`, `ischar`, `isletter`, `isspace`, `iscellstr`, `isnumeric`, `isa`, `is*`

Purpose

Determine whether input is structure array

Syntax

```
tf = isstruct(A)
```

Description

`tf = isstruct(A)` returns logical `1` (`true`) if `A` is a MATLAB structure and logical `0` (`false`) otherwise.

Examples

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];

isstruct(patient)

ans =

     1
```

See Also

`struct`, `isfield`, `iscell`, `ischar`, `isobject`, `isnumeric`, `islogical`, `isa`, `is*`, dynamic field names

isstudent

Purpose

Determine whether Student Version of MATLAB

Syntax

```
tf = isstudent
```

Description

`tf = isstudent` returns logical `1` (`true`) for the Student Version of MATLAB and logical `0` (`false`) for commercial versions.

See Also

`ver`, `version`, `license`, `ispc`, `isunix`, `is*`

Purpose

Determine whether UNIX version of MATLAB

Syntax

```
tf = isunix
```

Description

`tf = isunix` returns logical `1` (`true`) for the UNIX version of MATLAB and logical `0` (`false`) otherwise.

See Also

`ispc`, `ismac`, `isstudent`, `is*`

isvalid (serial)

Purpose

Determine whether serial port objects are valid

Syntax

```
out = isvalid(obj)
```

Arguments

`obj`	A serial port object or array of serial port objects.
`out`	A logical array.

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a `0` where the elements of `obj` are invalid serial port objects and a `1` where the elements of `obj` are valid serial port objects.

Remarks

`obj` becomes invalid after it is removed from memory with the `delete` function. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command.

Example

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM1');
```

`s2` becomes invalid after it is deleted.

```
delete(s2)
```

`isvalid` verifies that `s1` is valid and `s2` is invalid.

```
sarray = [s1 s2];
isvalid(sarray)
ans =
     1     0
```

See Also

Functions

`clear`, `delete`

Purpose

Determine whether timer object is valid

Syntax

```
out = isvalid(obj)
```

Description

`out = isvalid(obj)` returns a logical array, `out`, that contains a `0` where the elements of `obj` are invalid timer objects and a `1` where the elements of `obj` are valid timer objects.

An invalid timer object is an object that has been deleted and cannot be reused. Use the `clear` command to remove an invalid timer object from the workspace.

Examples

Create a valid timer object.

```
t = timer;
out = isvalid(t)
out =

    1
```

Delete the timer object, making it invalid.

```
delete(t)
out1 = isvalid(t)
out1 =

    0
```

See Also

`timer`, `delete(timer)`

isvarname

Purpose

Determine whether input is valid variable name

Syntax

```
tf = isvarname 'str'
isvarname str
```

Description

`tf = isvarname 'str'` returns logical `1` (`true`) if the string `str` is a valid MATLAB variable name and logical `0` (`false`) otherwise. A valid variable name is a character string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

MATLAB keywords are not valid variable names. Type the command `iskeyword` with no input arguments to see a list of MATLAB keywords.

`isvarname str` uses the MATLAB command format.

Examples

This variable name is valid:

```
isvarname foo
ans =
   1
```

This one is not because it starts with a number:

```
isvarname 8th_column
ans =
   0
```

If you are building strings from various pieces, place the construction in parentheses.

```
d = date;

isvarname(['Monday_', d(1:2)])
ans =
   1
```

See Also

`genvarname`, `isglobal`, `iskeyword`, `namelengthmax`, `is*`

Purpose

Determine whether input is vector

Syntax

```
TF = isvector(A)
```

Description

`TF = isvector(A)` returns logical `1` (`true`) if `A` is a 1-by-`N` or `N`-by-1 vector where `N >= 0`, and logical `0` (`false`) otherwise.

The `A` argument can also be a MATLAB object, as described in "Classes and Objects", as long as that object overloads the `size` function.

Examples

Test matrix `A` and its row and column vectors:

```
A = rand(5);

isvector(A)
ans =
     0

isvector(A(3, :))
ans =
     1

isvector(A(:, 2))
ans =
     1
```

See Also

`isscalar`, `isempty`, `isnumeric`, `islogical`, `ischar`, `isa`, `is*`

Purpose

Imaginary unit

Syntax

```
j
x+yj
x+j*y
```

Description

Use the character `j` in place of the character `i`, if desired, as the imaginary unit.

As the basic imaginary unit `sqrt(-1)`, `j` is used to enter complex numbers. Since `j` is a function, it can be overridden and used as a variable. This permits you to use `j` as an index in `for` loops, etc.

It is possible to use the character `j` without a multiplication sign as a suffix in forming a numerical constant.

Examples

```
Z = 2+3j
Z = x+j*y
Z = r*exp(j*theta)
```

See Also

`conj`, `i`, `imag`, `real`

Purpose

Add entries to dynamic Java class path

Syntax

```
javaaddpath('dpath')
javaaddpath('dpath', '-end')
```

Description

`javaaddpath('dpath')` adds one or more directories or JAR files to the beginning of the current dynamic Java class path. `dpath` is a string or cell array of strings containing the directory or JAR file. (See the Remarks section for a description of static and dynamic Java paths.)

`javaaddpath('dpath', '-end')` adds one or more directories or files to the end of the current dynamic Java path.

Remarks

The Java path consists of two segments: a static path (read only at startup) and a dynamic path. MATLAB always searches the static path (defined in `classpath.txt`) before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

Use the `clear java` command to reload the classes defined on the dynamic Java path. This is necessary if you add new Java classes or if you modify existing Java classes on the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file `classpath.txt`. The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `classpath.txt` and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB.

Examples

Create function to set initial dynamic Java class path:

```
function setdynpath
javaclasspath({
   'C:\Work\Java\ClassFiles', ...
   'C:\Work\JavaTest\curvefit.jar', ...
   'C:\Work\JavaTest\timer.jar', ...
   'C:\Work\JavaTest\patch.jar'});
% end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;

javaclasspath

       STATIC JAVA PATH

   D:\SysO\Java\util.jar
```

```
    D:\SysO\Java\widgets.jar
    D:\SysO\Java\beans.jar
              .
              .

        DYNAMIC JAVA PATH

    C:\Work\Java\ClassFiles
    C:\Work\JavaTest\curvefit.jar
    C:\Work\JavaTest\timer.jar
    C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. One entry specifies a directory and the other a Java Archive (JAR) file. When you add a directory to the path, MATLAB includes all files in that directory as part of the path:

```
javaaddpath({
   'C:\Work\Java\Curvefit\Test', ...
   'C:\Work\Java\mywidgets.jar'});
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
   'C:\Work\Java\ClassFiles'
   'C:\Work\JavaTest\curvefit.jar'
   'C:\Work\JavaTest\timer.jar'
   'C:\Work\JavaTest\patch.jar'
   'C:\Work\Java\Curvefit\Test'
   'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If you modify one or more classes that are defined on the dynamic path, you need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using,

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

Other Examples

Add a JAR file from an internet URL to your dynamic Java path:

```
javaaddpath http://www.example.com/my.jar
```

Add the current directory with the following statement:

```
javaaddpath(pwd)
```

See Also

`javaclasspath`, `javarmpath`, `clear`

See “Bringing Java Classes and Methods into MATLAB” for more information.

Purpose

Construct Java array

Syntax

```
javaArray('package_name.class_name',x1,...,xn)
```

Description

`javaArray('package_name.class_name',x1,...,xn)` constructs an empty Java array capable of storing objects of Java class, '*`class_name`*'. The dimensions of the array are `x1` by `...` by `xn`. You must include the package name when specifying the class.

The array that you create with `javaArray` is equivalent to the array that you would create with the Java code

```
A = new class_name[x1]...[xn];
```

Examples

The following example constructs and populates a 4-by-5 array of `java.lang.Double` objects.

```
dblArray = javaArray ('java.lang.Double', 4, 5);
for m = 1:4
   for n = 1:5
   dblArray(m,n) = java.lang.Double((m*10) + n);
   end
end

dblArray

dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

See Also

`javaObject`, `javaMethod`, `class`, `methodsview`, `isjava`

Purpose

Generate error message based on Java feature support

Syntax

```
javachk(feature)
javachk(feature, component)
```

Description

`javachk(feature)` returns a generic error message if the specified Java `feature` is not available in the current MATLAB session. If it is available, `javachk` returns an empty matrix. Possible `feature` arguments are shown in the following table.

Feature	Description
`'awt'`	Abstract Window Toolkit components[1] are available.
`'desktop'`	The MATLAB interactive desktop is running.
`'jvm'`	The Java Virtual Machine is running.
`'swing'`	Swing components[2] are available.

1. Java's GUI components in the Abstract Window Toolkit

2. Java's lightweight GUI components in the Java Foundation Classes

`javachk(feature, component)` works the same as the above syntax, except that the specified `component` is also named in the error message. (See the example below.)

Examples

The following M-file displays an error with the message `"CreateFrame is not supported on this platform."` when run in a MATLAB session in which the AWT's GUI components are not available. The second argument to `javachk` specifies the name of the M-file, which is then included in the error message generated by MATLAB.

```
javamsg = javachk('awt', mfilename);
if isempty(javamsg)
   myFrame = java.awt.Frame;
   myFrame.setVisible(1);
else
   error(javamsg);
end
```

See Also

usejava

javaclasspath

Purpose

Set and get dynamic Java class path

Syntax

```
javaclasspath
javaclasspath(dpath)
dpath = javaclasspath
spath = javaclasspath('-static')
jpath = javaclasspath('-all')
javaclasspath(statusmsg)
```

Description

`javaclasspath` displays the static and dynamic segments of the Java path. (See the Remarks section, below, for a description of static and dynamic Java paths.)

`javaclasspath(dpath)` sets the dynamic Java path to one or more directory or file specifications given in `dpath`, where `dpath` can be a string or cell array of strings.

`dpath = javaclasspath` returns the dynamic segment of the Java path in cell array, `dpath`. If no dynamic paths are defined, `javaclasspath` returns an empty cell array.

`spath = javaclasspath('-static')` returns the static segment of the Java path in cell array, `spath`. No path information is displayed unless you specify an output variable. If no static paths are defined, `javaclasspath` returns an empty cell array.

`jpath = javaclasspath('-all')` returns the entire Java path in cell array, `jpath`. The returned cell array contains first the static segment of the path, and then the dynamic segment. No path information is displayed unless you specify an output variable. If no dynamic paths are defined, `javaclasspath` returns an empty cell array.

`javaclasspath(statusmsg)` enables or disables the display of status messages from the `javaclasspath`, `javaaddpath`, and `javarmpath` functions. Values for the *`statusmsg`* argument are

statusmsg	Description
'-v1'	Display status messages while loading the Java path from the file system
'-v0'	Do not display status messages. This is the default.

Remarks

The Java path consists of two segments: a static path and a dynamic path. MATLAB always searches the static path before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file `classpath.txt`. The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `classpath.txt` and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB.

Examples

Create a function to set your initial dynamic Java class path:

```
function setdynpath
javaclasspath({
   'C:\Work\Java\ClassFiles', ...
   'C:\Work\JavaTest\curvefit.jar', ...
   'C:\Work\JavaTest\timer.jar', ...
   'C:\Work\JavaTest\patch.jar'});
%             end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;

javaclasspath

       STATIC JAVA PATH

   D:\SysO\Java\util.jar
   D:\SysO\Java\widgets.jar
   D:\SysO\Java\beans.jar
             .
             .

       DYNAMIC JAVA PATH

   C:\Work\Java\ClassFiles
   C:\Work\JavaTest\curvefit.jar
   C:\Work\JavaTest\timer.jar
   C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. One entry specifies a directory and the other a Java Archive (JAR) file. When you add a directory to the path, MATLAB includes all files in that directory as part of the path:

```
javaaddpath({
   'C:\Work\Java\Curvefit\Test', ...
   'C:\Work\Java\mywidgets.jar'});
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
```

```
'C:\Work\Java\ClassFiles'
'C:\Work\JavaTest\curvefit.jar'
'C:\Work\JavaTest\timer.jar'
'C:\Work\JavaTest\patch.jar'
'C:\Work\Java\Curvefit\Test'
'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If, at some time, you modify one or more classes that are defined on the dynamic path, you will need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using,

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

See Also

`javaaddpath`, `javarmpath`, `clear`

javaMethod

Purpose

Invoke Java method

Syntax

```
javaMethod('method_name','class_name',x1,...,xn)
javaMethod('method_name',J,x1,...,xn)
```

Description

`javaMethod('method_name','class_name',x1,...,xn)` invokes the `static` method `method_name` in the class `class_name`, with the argument list that matches `x1,...,xn`.

`javaMethod('method_name',J,x1,...,xn)` invokes the nonstatic method `method_name` on the object J, with the argument list that matches `x1,...,xn`.

Remarks

Using the `javaMethod` function enables you to

- Use methods having names longer than 31 characters
- Specify the method you want to invoke at run-time, for example, as input from an application user

The `javaMethod` function enables you to use methods having names longer than 31 characters. This is the only way you can invoke such a method in MATLAB. For example:

```
javaMethod('DataDefinitionAndDataManipulationTransactions', T);
```

With `javaMethod`, you can also specify the method to be invoked at run-time. In this situation, your code calls `javaMethod` with a string variable in place of the method name argument. When you use `javaMethod` to invoke a static method, you can also use a string variable in place of the class name argument.

Note Typically, you do not need to use `javaMethod`. The default MATLAB syntax for invoking a Java method is somewhat simpler and is preferable for most applications. Use `javaMethod` primarily for the two cases described above.

Examples

To invoke the `static` Java method `isNaN` on class, `java.lang.Double`, use

```
javaMethod('isNaN','java.lang.Double',2.2)
```

The following example invokes the nonstatic method `setTitle`, where `frameObj` is a `java.awt.Frame` object.

```
frameObj = java.awt.Frame;
javaMethod('setTitle', frameObj, 'New Title');
```

See Also

`javaArray`, `javaObject`, `import`, `methods`, `isjava`

javaObject

Purpose

Construct Java object

Syntax

```
javaObject('class_name',x1,...,xn)
```

Description

`javaObject('class_name',x1,...,xn)` invokes the Java constructor for class '`class_name`' with the argument list that matches `x1,...,xn`, to return a new object.

If there is no constructor that matches the class name and argument list passed to `javaObject`, an error occurs.

Remarks

Using the `javaObject` function enables you to

- Use classes having names with more than 31 consecutive characters
- Specify the class for an object at run-time, for example, as input from an application user

The default MATLAB constructor syntax requires that no segment of the input class name be longer than 31 characters. (A *name segment*, is any portion of the class name before, between, or after a period. For example, there are three segments in class, `java.lang.String`.) Any class name segment that exceeds 31 characters is truncated by MATLAB. In the rare case where you need to use a class name of this length, you must use `javaObject` to instantiate the class.

The `javaObject` function also allows you to specify the Java class for the object being constructed at run-time. In this situation, you call `javaObject` with a string variable in place of the class name argument.

```
class  = 'java.lang.String';
text = 'hello';
strObj = javaObject(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, you would use

```
strObj = java.lang.String('hello');
```

Note Typically, you will not need to use `javaObject`. The default MATLAB syntax for instantiating a Java class is somewhat simpler and is preferable for most applications. Use `javaObject` primarily for the two cases described above.

Examples

The following example constructs and returns a Java object of class `java.lang.String`:

```
strObj = javaObject('java.lang.String','hello')
```

See Also

`javaArray`, `javaMethod`, `import`, `methods`, `fieldnames`, `isjava`

javarmpath

Purpose

Remove entries from dynamic Java class path

Syntax

```
javarmpath('dpath')
javarmpath dpath1 dpath2 ... dpathN
javarmpath(v1, v2, ..., vN)
```

Description

`javarmpath('dpath')` removes a directory or file from the current dynamic Java path. `dpath` is a string containing the directory or file specification. (See the Remarks section, below, for a description of static and dynamic Java paths.)

`javarmpath dpath1 dpath2 ... dpathN` removes those directories and files specified by `dpath1`, `dpath2`, `...`, `dpathN` from the dynamic Java path. Each input argument is a string containing a directory or file specification.

`javarmpath(v1, v2, ..., vN)` removes those directories and files specified by `v1`, `v2`, `...`, `vN` from the dynamic Java path. Each input argument is a variable to which a directory or file specification is assigned.

Remarks

The Java path consists of two segments: a static path and a dynamic path. MATLAB always searches the static path before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the file `classpath.txt`. The static Java path offers better Java class loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `classpath.txt` and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB.

Examples

Create a function to set your initial dynamic Java class path:

```
function setdynpath
javaclasspath({
   'C:\Work\Java\ClassFiles', ...
   'C:\Work\JavaTest\curvefit.jar', ...
   'C:\Work\JavaTest\timer.jar', ...
   'C:\Work\JavaTest\patch.jar'});
% end of file
```

Call this function to set up your dynamic class path. Then, use the `javaclasspath` function with no arguments to display all current static and dynamic paths:

```
setdynpath;

javaclasspath

       STATIC JAVA PATH
```

```
D:\SysO\Java\util.jar
D:\SysO\Java\widgets.jar
D:\SysO\Java\beans.jar
          .
          .

    DYNAMIC JAVA PATH

C:\Work\Java\ClassFiles
C:\Work\JavaTest\curvefit.jar
C:\Work\JavaTest\timer.jar
C:\Work\JavaTest\patch.jar
```

At some later time, add the following two entries to the dynamic path. One entry specifies a directory and the other a Java Archive (JAR) file. When you add a directory to the path, MATLAB includes all files in that directory as part of the path:

```
javaaddpath({
   'C:\Work\Java\Curvefit\Test', ...
   'C:\Work\Java\mywidgets.jar'});
```

Use `javaclasspath` with just an output argument to return the dynamic path alone:

```
p = javaclasspath
p =
   'C:\Work\Java\ClassFiles'
   'C:\Work\JavaTest\curvefit.jar'
   'C:\Work\JavaTest\timer.jar'
   'C:\Work\JavaTest\patch.jar'
   'C:\Work\Java\Curvefit\Test'
   'C:\Work\Java\mywidgets.jar'
```

Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

If, at some time, you modify one or more classes that are defined on the dynamic path, you will need to clear the former definition for those classes from MATLAB memory. You can clear all dynamic Java class definitions from memory using,

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

Use `javarmpath` to remove a file or directory from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

See Also

`javaclasspath`, `javaaddpath`, `clear`

keyboard

Purpose

Input from keyboard

Syntax

```
keyboard
```

Description

`keyboard` , when placed in an M-file, stops execution of the file and gives control to the keyboard. The special status is indicated by a `K` appearing before the prompt. You can examine or change variables; all MATLAB commands are valid. This keyboard mode is useful for debugging your M-files..

To terminate the keyboard mode, type the command

```
return
```

then press the **Return** key.

See Also

`dbstop`, `input`, `quit`, `pause`, `return`

Purpose

Kronecker tensor product

Syntax

```
K = kron(X,Y)
```

Description

`K = kron(X,Y)` returns the Kronecker tensor product of `X` and `Y`. The result is a large array formed by taking all possible products between the elements of `X` and those of `Y`. If `X` is `m`-by-`n` and `Y` is `p`-by-`q`, then `kron(X,Y)` is `m*p`-by-`n*q`.

Examples

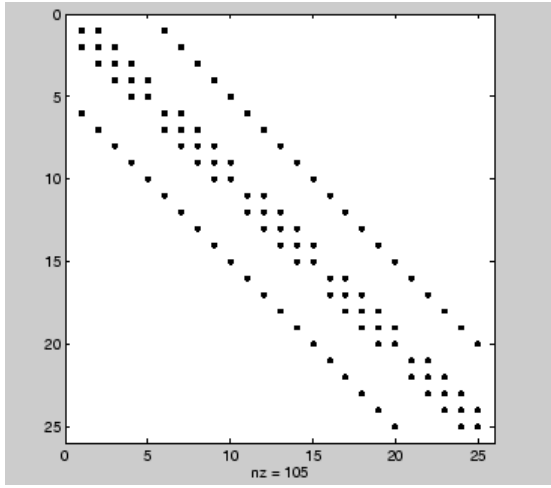
If `X` is 2-by-3, then `kron(X,Y)` is

```
[ X(1,1)*Y X(1,2)*Y X(1,3)*Y
  X(2,1)*Y X(2,2)*Y X(2,3)*Y ]
```

The matrix representation of the discrete Laplacian operator on a two-dimensional, `n`-by-`n` grid is a `n^2`-by-`n^2` sparse matrix. There are at most five nonzero elements in each row or column. The matrix can be generated as the Kronecker product of one-dimensional difference operators with these statements:

```
I = speye(n,n);
E = sparse(2:n,1:n-1,1,n,n);
D = E+E'-2*I;
A = kron(D,I)+kron(I,D);
```

Plotting this with the `spy` function for `n = 5` yields:



See Also

`hankel`, `toeplitz`

Purpose

Last uncaught exception

Syntax

```
ME = MException.last
MException.last('reset')
```

Description

`ME = MException.last` displays the contents of the `MException` object representing your most recent uncaught error. This is a static method of the `MException` class; it is not a method of an `MException` class object. Use this method from the MATLAB command line only, and not within an M-file.

`MException.last('reset')` sets the `identifier` and `message` properties of the most recent exception to the empty string, the `stack` property to a 0-by-1 structure, and `cause` property to an empty cell array.

`last` is not set in a `try-catch` statement.

Examples

This example displays the last error that was caught during this MATLAB session:

```
A = 25;
A(2)
??? Index exceeds matrix dimensions.

MException.last
ans =

 MException object with properties:

    identifier: 'MATLAB:badsubscript'
       message: 'Index exceeds matrix dimensions.'
         stack: [0x1 struct]
         cause: {}
```

See Also

`try`, `catch`, `error`, `assert`, `MException`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`,

addCause(MException), getReport(MException), disp(MException), isequal(MException), eq(MException), ne(MException)

Purpose

Last error message

Note `lasterr` has been replaced by `lasterror`, but will be maintained for backward compatibility.

Syntax

```
msgstr = lasterr
[msgstr, msgid] = lasterr
lasterr('new_msgstr')
lasterr('new_msgstr', 'new_msgid')
[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')
```

Description

`msgstr = lasterr` returns the last error message generated by MATLAB.

`[msgstr, msgid] = lasterr` returns the last error in `msgstr` and its message identifier in `msgid`. If the error was not defined with an identifier, `lasterr` returns an empty string for `msgid`. See and in the MATLAB Programming documentation for more information on the `msgid` argument and how to use it.

`lasterr('new_msgstr')` sets the last error message to a new string, `new_msgstr`, so that subsequent invocations of `lasterr` return the new error message string. You can also set the last error to an empty string with `lasterr('')`.

`lasterr('new_msgstr', 'new_msgid')` sets the last error message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lasterr` return the new error message and message identifier.

`[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')` returns the last error message and its identifier, also changing these values so that subsequent invocations of `lasterr` return the message and identifier strings specified by `new_msgstr` and `new_msgid` respectively.

Examples

Example 1

Here is a function that examines the `lasterr` string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply:

```
function matrix_multiply(A, B)
try
   A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
   else
      if(strfind(errmsg, 'not defined for variables of class'))
         disp('** Both arguments must be double matrices')
      end
   end
end
```

If you call this function with matrices that are incompatible for matrix multiplication (e.g., the column dimension of `A` is not equal to the row dimension of `B`), MATLAB catches the error and uses `lasterr` to determine its source:

```
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];

matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

Example 2

Specify a message identifier and error message string with `error`:

```
error('MyToolbox:angleTooLarge', ...
      'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterr` to determine the message identifier and error message string for the failing operation:

```
[errmsg, msgid] = lasterr
errmsg =
   The angle specified must be less than 90 degrees.
msgid =
   MyToolbox:angleTooLarge
```

See Also

`error`, `lasterror`, `rethrow`, `warning`, `lastwarn`

lasterror

Purpose

Last error message and related information

Syntax

```
s = lasterror
s = lasterror(err)
s = lasterror('reset')
```

Description

`s = lasterror` returns a structure `s` containing information about the most recent error issued by MATLAB. The return structure contains the following fields:

Fieldname	Description
`message`	Character array containing the text of the error message.
`identifier`	Character array containing the message identifier of the error message. If the last error issued by MATLAB had no message identifier, then the `identifier` field is an empty character array.
`stack`	Structure providing information on the location of the error. The structure has fields `file`, `name`, and `line`, and is the same as the structure returned by the `dbstack` function. If `lasterror` returns no stack information, `stack` is a 0-by-1 structure having the same three fields.

Note The `lasterror` return structure might contain additional fields in future versions of MATLAB.

The fields of the structure returned in `stack` are

Fieldname	Description
file	Name of the file in which the function generating the error appears. This field is the empty string if there is no file.
name	Name of the function in which the error occurred. If this is the primary function of the M-file, and the function name differs from the M-file name, name is set to the M-file name.
line	M-file line number where the error occurred.

See in the MATLAB Programming documentation for more information on the syntax and usage of message identifiers.

`s = lasterror(err)` sets the last error information to the error message and identifier specified in the structure `err`. Subsequent invocations of `lasterror` return this new error information. The optional return structure `s` contains information on the previous error.

`s = lasterror('reset')` sets the last error information to the default state. In this state, the `message` and `identifier` fields of the return structure are empty strings, and the `stack` field is a 0-by-1 structure.

Examples

Example 1

Save the following MATLAB code in an M-file called `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.
check_inputs(x)
y = sum(x)/length(x);      % The actual computation

function check_inputs(x)
[m,n] = size(x);
if (~((m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('AVG:NotAVector', 'Input must be a vector.')
```

```
end
```

Now run the function. Because this function requires vector input, passing a scalar value to it forces an error. The error occurs in subroutine `check_inputs`:

```
average(200)
??? Error using ==> average>check_inputs
Input must be a vector.

Error in ==> average at 5
check_inputs(x)
```

Get the three fields from `lasterror`:

```
err = lasterror
err =
       message: [1x61 char]
    identifier: 'AVG:NotAVector'
         stack: [2x1 struct]
```

Display the text of the error message:

```
msg = err.message
msg =
    Error using ==> average>check_inputs
    Input must be a vector.
```

Display the fields containing the `stack` information. `err.stack` is a 2-by-1 structure because it provides information on the failing subroutine `check_inputs` and also the outer, primary function `average`:

```
st1 = err.stack(1,1)
st1 =
    file: 'd:\matlab_test\average.m'
    name: 'check_inputs'
    line: 11
```

```
st2 = err.stack(2,1)
st2 =
    file: 'd:\matlab_test\average.m'
    name: 'average'
    line: 5
```

Note As a rule, the name of your primary function should be the same as the name of the M-file containing that function. If these names differ, MATLAB uses the M-file name in the `name` field of the `stack` structure.

Example 2

`lasterror` is often used in conjunction with the `rethrow` function in `try-catch` statements. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(lasterror)
end
```

See Also

`try`, `catch`, `error`, `assert`, `MException`, `rethrow`, `lastwarn`, `dbstack`

lastwarn

Purpose

Last warning message

Syntax

```
msgstr = lastwarn
[msgstr, msgid] = lastwarn
lastwarn('new_msgstr')
lastwarn('new_msgstr', 'new_msgid')
[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')
```

Description

`msgstr = lastwarn` returns the last warning message generated by MATLAB.

`[msgstr, msgid] = lastwarn` returns the last warning in `msgstr` and its message identifier in `msgid`. If the warning was not defined with an identifier, `lastwarn` returns an empty string for `msgid`. See and "Warning Control" in the MATLAB Programming documentation for more information on the `msgid` argument and how to use it.

`lastwarn('new_msgstr')` sets the last warning message to a new string, `new_msgstr`, so that subsequent invocations of `lastwarn` return the new warning message string. You can also set the last warning to an empty string with `lastwarn('')`.

`lastwarn('new_msgstr', 'new_msgid')` sets the last warning message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lastwarn` return the new warning message and message identifier.

`[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')` returns the last warning message and its identifier, also changing these values so that subsequent invocations of `lastwarn` return the message and identifier strings specified by `new_msgstr` and `new_msgid`, respectively.

Remarks

`lastwarn` does not return warnings that are reported during the parsing of MATLAB commands. (Warning messages that include the failing file name and line number are parse-time warnings.)

Examples

Specify a message identifier and warning message string with `warning`:

```
warning('MATLAB:divideByZero', 'Divide by zero');
```

Use `lastwarn` to determine the message identifier and error message string for the operation:

```
[warnmsg, msgid] = lastwarn
warnmsg =
   Divide by zero
msgid =
   MATLAB:divideByZero
```

See Also

`warning`, `error`, `lasterr`, `lasterror`

lcm

Purpose

Least common multiple

Syntax

```
L = lcm(A,B)
```

Description

`L = lcm(A,B)` returns the least common multiple of corresponding elements of arrays `A` and `B`. Inputs `A` and `B` must contain positive integer elements and must be the same size (or either can be scalar).

Examples

```
lcm(8,40)

ans =

    40
lcm(pascal(3),magic(3))

ans =
     8     1     6
     3    10    21
     4     9     6
```

See Also

`gcd`

Purpose

Block ldl' factorization for Hermitian indefinite matrices

Syntax

```
L = ldl(A)
[L,D] = ldl(A)
[L,D,P] = ldl(A)
[L,D,p] = ldl(A,'vector')
[U,D,P] = ldl(A,'upper')
[U,D,p] = ldl(A,'upper','vector')
[U,D,P,S] = ldl(A)
```

Description

`L = ldl(A)` returns only the "psychologically lower triangular matrix" `L` as in the two-output form. The permutation information is lost, as is the block diagonal factor `D`. By default, `ldl` references only the diagonal and lower triangle of `A`, and assumes that the upper triangle is the complex conjugate transpose of the lower triangle. Therefore `[L,D,P] = ldl(TRIL(A))` and `[L,D,P] = ldl(A)`both return the exact same factors. Note, this syntax is not valid for sparse `A`.

`[L,D] = ldl(A)` stores a block diagonal matrix `D` and a "psychologically lower triangular matrix" (i.e a product of unit lower triangular and permutation matrices) in `L` such that `A = L*D*L'`. The block diagonal matrix `D` has 1-by-1 and 2-by-2 blocks on its diagonal. Note, this syntax is not valid for sparse `A`.

`[L,D,P] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, and permutation matrix `P` such that `P'*A*P = L*D*L'`. This is equivalent to `[L,D,P] = ldl(A,'matrix')`.

`[L,D,p] = ldl(A,'vector')` returns the permutation information as a vector, `p`, instead of a matrix. The `p` output is a row vector such that `A(p,p) = L*D*L'`.

`[U,D,P] = ldl(A,'upper')` references only the diagonal and upper triangle of `A` and assumes that the lower triangle is the complex conjugate transpose of the upper triangle. This syntax returns a unit upper triangular matrix `U` such that `P'*A*P = U'*D*U` (assuming that `A` is Hermitian, and not just upper triangular). Similarly, `[L,D,P] = ldl(A,'lower')` gives the default behavior.

`[U,D,p] = ldl(A,'upper','vector')` returns the permutation information as a vector, `p`, as does `[L,D,p] = ldl(A,'lower','vector')`. `A` must be a full matrix.

`[U,D,P,S] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, permutation matrix `P`, and scaling matrix `S` such that `P'*S*A*S*P = L*D*L'`. This syntax is only available for real sparse matrices, and only the lower triangle of `A` is referenced. `ldl` uses MA57 for sparse real symmetric `A`.

Examples

These examples illustrate the use of the various forms of the `ldl` function, including the one-, two-, and three-output form, and the use of the `vector` and `upper` options. The topics covered are:

Before running any of these examples, you will need to generate the following positive definite and indefinite Hermitian matrices:

```
A = full(delsq(numgrid('L', 10)));
rand('state', 0);
B = rand(10);
M = [eye(10) B; B' zeros(10)];
```

The structure of `M` here is very common in optimization and fluid-flow problems, and `M` is in fact indefinite. Note that the positive definite matrix `A` must be full, as `ldl` does not accept sparse arguments.

Example 1 — One-Output Form of ldl

The one-output form of `ldl` returns the psychologically unit lower-triangular matrix as above. Note that this is a different matrix from that which you would derive with the `lu` function, as `lu` just returns what comes from LAPACK. Although `ldl` is also implemented using LAPACK routines (`ssytrf`, `dsytrf`, `chetrf`, `zhetrf`), you must decipher the output in ways that are lost when only one output is returned:

```
Lm = ldl(M); Dm = Lm\(M/Lm');
fprintf(1, ...
'The error norm ||M - Lm*Dm*Lm''|| is %g\n', norm(M - Lm*Dm*Lm'));
```

You can apply the L output from this command to the input matrix to recover `D` (approximately).

Example 2 — Two-Output Form of ldl

The two-output form of `ldl` returns `L` and `D` such that `A-(L*D*L')` is small, `L` is "psychologically unit lower triangular" (i.e., a permuted unit lower triangular matrix), and `D` is a block 2-by-2 diagonal. Note also that, because `A` is positive definite, the diagonal of `D` is all positive:

```
[LA,DA] = ldl(A);
fprintf(1, ...
'The factorization error ||A - LA*DA*LA''|| is %g\n', ...
norm(A - LA*DA*LA'));
neginds = find(diag(DA) < 0)
```

Given a `b`, solve `Ax=b` using `LA`, `DA`:

```
bA = sum(A,2);
x = LA'\(DA\(LA\bA));
fprintf(...
'The absolute error norm ||x - ones(size(bA))|| is %g\n', ...
norm(x - ones(size(bA))));
```

Example 3 — Three Output Form of ldl

The three output form returns the permutation matrix as well, so that `L` is in fact unit lower triangular:

```
[Lm, Dm, Pm] = ldl(M);
fprintf(1, ...
'The error norm ||Pm''*M*Pm - Lm*Dm*Lm''|| is %g\n', ...
norm(Pm'*M*Pm - Lm*Dm*Lm'));
fprintf(1, ...
'The difference between Lm and tril(Lm) is %g\n', ...
norm(Lm - tril(Lm)));
```

Given `b`, solve `Mx=b` using `Lm`, `Dm`, and `Pm`:

```
bM = sum(M,2);
x = Pm*(Lm'\(Dm\(Lm\(Pm'*bM))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

Example 4 — The Structure of D

`D` is a block diagonal matrix with 1-by-1 blocks and 2-by-2 blocks. That makes it a special case of a tridiagonal matrix. When the input matrix is positive definite, `D` is almost always diagonal (depending on how definite the matrix is). When the matrix is indefinite however, `D` may be diagonal or it may express the block structure. For example, with `A` as above, `DA` is diagonal. But if you shift `A` just a bit, you end up with an indefinite matrix, and then you can compute a `D` that has the block structure.

```
figure; spy(DA); title('Structure of D from ldl(A)');
[Las, Das] = ldl(A - 4*eye(size(A)));
figure; spy(Das);
title('Structure of D from ldl(A - 4*eye(size(A)))');
```

Example 5 — Using the 'vector' Option

Like the `lu` function, `ldl` accepts an argument that determines whether the function returns a permutation vector or permutation matrix. `ldl` returns the latter by default. When you select `'vector'`, the function executes faster and uses less memory. For this reason, specifying the `'vector'` option is recommended. Another thing to note is that indexing is typically faster than multiplying for this kind of operation:

```
[Lm, Dm, pm] = ldl(M, 'vector');
fprintf(1, 'The error norm ||M(pm,pm) - Lm*Dm*Lm''|| is %g\n', ...
  norm(M(pm,pm) - Lm*Dm*Lm'));

% Solve a system with this kind of factorization.
clear x;
x(pm,:) = Lm'\(Dm\(Lm\(bM(pm,:))));
fprintf('The absolute error norm ||x - ones(size(b))|| is %g\n', ...
  norm(x - ones(size(bM))));
```

Example 6 — Using the 'upper' Option

Like the `chol` function, `ldl` accepts an argument that determines which triangle of the input matrix is referenced, and also whether `ldl` returns a lower (`L`) or upper (`L'`) triangular factor. For dense matrices, there are no real savings with using the upper triangular version instead of the lower triangular version:

```
Ml = tril(M);
[Lml, Dml, Pml] = ldl(Ml, 'lower'); % 'lower' is default behavior.
fprintf(1, ...
'The difference between Lml and Lm is %g\n', norm(Lml - Lm));
[Umu, Dmu, pmu] = ldl(triu(M), 'upper', 'vector');
fprintf(1, ...
'The difference between Umu and Lm'' is %g\n', norm(Umu - Lm'));

% Solve a system using this factorization.
clear x;
x(pm,:) = Umu\(Dmu\(Umu'\(bM(pmu,:))));
fprintf(...
```

```
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

When specifying both the `'upper'` and `'vector'` options, `'upper'` must precede `'vector'` in the argument list.

Example 7 — linsolve and the Hermitian indefinite solver

When using the `linsolve` function, you may experience better performance by exploiting the knowledge that a system has a symmetric matrix. The matrices used in the examples above are a bit small to see this so, for this example, generate a larger matrix. The matrix here is symmetric positive definite, and below we will see that with each bit of knowledge about the matrix, there is a corresponding speedup. That is, the symmetric solver is faster than the general solver while the symmetric positive definite solver is faster than the symmetric solver:

```
Abig = full(delsq(numgrid('L', 30)));
bbig = sum(Abig, 2);
LSopts.POSDEF = false;
LSopts.SYM = false;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.SYM = true;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.POSDEF = true;
tic; linsolve(Abig, bbig, LSopts); toc;
```

Algorithm

`ldl` uses the LAPACK routines listed in the following table.

	Real	Complex
Double	DSYTRF	ZHETRF
Single	SSYTRN	CHETRF

See Also

`chol`, `lu`, `qr`

Purpose

Left or right array division

Syntax

```
ldivide(A,B)
A.\B
rdivide(A,B)
A./B
```

Description

`ldivide(A,B)` and the equivalent `A.\B` divides each entry of `B` by the corresponding entry of `A`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

`rdivide(A,B)` and the equivalent `A./B` divides each entry of `A` by the corresponding entry of `B`. `A` and `B` must be arrays of the same size. A scalar value for either `A` or `B` is expanded to an array of the same size as the other.

Example

```
A = [1 2 3;4 5 6];
B = ones(2, 3);
A.\B

ans =

    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
```

See Also

Arithmetic Operators, `mldivide`, `mrdivide`

Purpose

Test for less than or equal to

Syntax

```
A <= B
le(A, B)
```

Description

`A <= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is less than or equal to `B`, or set to logical 0 (`false`) where `A` is greater than `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then MATLAB returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`le(A, B)` is called for the syntax `A < =B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are less than or equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A <= B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     1     0
     1     0     0     1     0     1
```

```
     0     1     0     0     1     1
     1     0     0     0     1     0
```

See Also

`lt`, `eq`, `ge`, `gt`, `ne`, Relational Operators

legend

Purpose

Graph legend for lines and patches

GUI Alternatives

Add a legend to a selected axes on a graph with the **Insert Legend** tool  on the figure toolbar, or use **Insert —> Legend** from the figure menu. Use the Property Editor to modify the position, font, and other properties of a legend. For details, see Using Plot Edit Mode in the MATLAB Graphics documentation.

Syntax

```
legend('string1','string2',...)
legend(h,'string1','string2',...)
legend(M)
legend(h,M)
legend(M,'parameter_name','parameter_value',...)
legend(h,M,'parameter_name','parameter_value',...)
legend(axes_handle,...)
legend('off'), legend(axes_handle,'off')
legend('toggle'), legend(axes_handle,'toggle')
legend('hide'), legend(axes_handle,'hide')
legend('show'), legend(axes_handle,'show')
legend('boxoff'), legend(axes_handle,'boxoff')
legend('boxon'), legend(axes_handle,'boxon')
legend_handle = legend(...)
legend
legend(legend_handle)
legend(...,'Location',location)
legend(...,'Orientation','orientation')
[legend_h,object_h,plot_h,text_strings] = legend(...)
legend(li_object,string1,string2,string3)
legend(li_objects,M)
legend('v6',M,...)
legend('v6',AX)
```

Description

`legend` places a legend on various types of graphs (line plots, bar graphs, pie charts, etc.). For each line plotted, the legend shows a sample of the line type, marker symbol, and color beside the text label

you specify. When plotting filled areas (patch or surface objects), the legend contains a sample of the face color next to the text label.

The font size and font name for the legend strings match the axes `FontSize` and `FontName` properties.

`legend('string1','string2',...)` displays a legend in the current axes using the specified strings to label each set of data.

`legend(h,'string1','string2',...)` displays a legend on the plot containing the objects identified by the handles in the vector `h` and uses the specified strings to label the corresponding graphics object (line, barseries, etc.).

`legend(M)` adds a legend containing the rows of the matrix or cell array of strings `M` as labels. For matrices, this is the same as `legend(M(1,:),M(2,:),...)`.

`legend(h,M)` associates each row of the matrix or cell array of strings `M` with the corresponding graphics object (patch or line) in the vector of handles `h`.

`legend(M,'parameter_name','parameter_value',...)` and `legend(h,M,'parameter_name','parameter_value',...)` allow parameter/value pairs to be set when creating a legend (you can also assign them with `set` or with the Property Editor or Property Inspector). `M` must be a cell array of names. Legends inherit the properties of axes, although not all of them are relevant to legend objects.

`legend(axes_handle,...)` displays the legend for the axes specified by `axes_handle`.

`legend('off'), legend(axes_handle,'off')` removes the legend in the current axes or the axes specified by `axes_handle`.

`legend('toggle'), legend(axes_handle,'toggle')` toggles the legend on or off. If no legend exists for the current axes, one is created using default strings.

The *default string* for an object is the value of the object's `DisplayName` property, if you have defined a value for `DisplayName` (which you can do using the Property Editor or calling `set`). Otherwise, `legend` constructs

a string of the form `data1`, `data2`, etc. Setting display names is useful when you are experimenting with legends and might forget how objects in a lineseries, for example, are ordered.

When you specify legend strings in a `legend` command, their respective `DisplayNames` are set to these strings. If you delete a legend and then create a new legend without specifying labels for it, the values of `DisplayName` are (re)used as label names. Naturally, the associated plot objects must have a `DisplayName` property for this to happen: all *`_series`* and *`_group`* plot objects have a `DisplayName` property; Handle Graphics primitives, such as `line` and `patch`, do not.

`legend('hide'), legend(axes_handle,'hide')` makes the legend in the current axes or the axes specified by `axes_handle` invisible.

`legend('show'), legend(axes_handle,'show')` makes the legend in the current axes or the axes specified by `axes_handle` visible. A legend is created if one did not exist previously. Legends created automatically are limited to depict only the first 20 lines in the plot; if you need more legend entries, you can manually create a legend for them all with `legend('string1','string2',...)` syntax.

`legend('boxoff'), legend(axes_handle,'boxoff')` removes the box from the legend in the current axes or the axes specified by `axes_handle`, and makes its background transparent.

`legend('boxon'), legend(axes_handle,'boxon')` adds a box with an opaque background to the legend in the current axes or the axes specified by `axes_handle`.

You can also type the above six commands using the syntax

```
legend keyword
```

If the keyword is not recognized, it is used as legend text, creating a legend or replacing the current legend.

`legend_handle = legend(...)` returns the handle to the legend on the current axes, or `[]` if no legend exists.

`legend` with no arguments refreshes all the legends in the current figure.

`legend(legend_handle)` refreshes the specified legend.

`legend(...,'Location',location)` uses *`location`* to determine where to place the legend. *`location`* can be either a 1-by-4 position vector ([left bottom width height]) or one of the following strings.

Specifier	Location in Axes
`North`	Inside plot box near top
`South`	Inside bottom
`East`	Inside right
`West`	Inside left
`NorthEast`	Inside top right (default)
`NorthWest`	Inside top left
`SouthEast`	Inside bottom right
`SouthWest`	Inside bottom left
`NorthOutside`	Outside plot box near top
`SouthOutside`	Outside bottom
`EastOutside`	Outside right
`WestOutside`	Outside left
`NorthEastOutside`	Outside top right
`NorthWestOutside`	Outside top left
`SouthEastOutside`	Outside bottom right
`SouthWestOutside`	Outside bottom left
`Best`	Least conflict with data in plot
`BestOutside`	Least unused space outside plot

If the legend text does not fit in the 1-by-4 position vector, the position vector is resized around the midpoint to fit the legend text given its font and size, making the legend taller or wider. The *`location`* string can be all lowercase and can be abbreviated by sentinel letter (e.g., N, NE, NEO, etc.). Using one of the `...Outside` values for *`location`* ensures that the legend does not overlap the plot, whereas overlaps can occur when you specify any of the other cardinal values. The *`location`* property applies to colorbars and legends, but not to axes.

Obsolete Location Values

The first column of the following table shows the now-obsolete specifiers for legend locations that were in use prior to Version 7, along with a description of the locations and their current equivalent syntaxes:

Obsolete Specifier	Location in Axes	Current Specifier
`-1`	Outside axes on right side	`NorthEastOutside`
`0`	Inside axes	`Best`
`1`	Upper right corner of axes	`NorthEast`
`2`	Upper left corner of axes	`NorthWest`
`3`	Lower left corner of axes	`SouthWest`
`4`	Lower right corner of axes	`SouthEast`

`legend(...,'Orientation','`*`orientation`*`')` creates a legend with the legend items arranged in the specified orientation. *`orientation`* can be `vertical` (the default) or `horizontal`.

`[legend_h,object_h,plot_h,text_strings] = legend(...)` returns

- `legend_h` — Handle of the legend axes
- `object_h` — Handles of the line, patch, and text graphics objects used in the legend
- `plot_h` — Handles of the lines and other objects used in the plot

- `text_strings` — Cell array of the text strings used in the legend

These handles enable you to modify the properties of the respective objects.

`legend(li_object,string1,string2,string3)` creates a legend for legendinfo objects `li_objects` with strings `string1`, etc.

`legend(li_objects,M)` creates a legend of legendinfo objects `li_objects`, where `M` is a string matrix or cell array of strings corresponding to the legendinfo objects.

Backward Compatibility

`legend('v6',M,...)`, for a cell array of strings `M`, creates a legend compatible with MATLAB 6.5 from the strings in `M` and any additional inputs.

`legend('v6',AX)`, for an axes handle `AX`, updates any Version 6 legends and returns the legend handle.

The following calls to `legend` are passed to the Version 6 legend mechanism to maintain backward compatibility:

```
legend('DeleteLegend')
legend('EditLegend',h)
legend('ShowLegendPlot',h)
legend('ResizeLegend')
legend('RestoreSize',hLegend)
legend('RecordSize',hPlot)
```

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

Remarks

`legend` associates strings with the objects in the axes in the same order that they are listed in the axes `Children` property. By default, the legend annotates the current axes.

MATLAB displays only one legend per axes. `legend` positions the legend based on a variety of factors, such as what objects the legend obscures.

The properties that legends do not share with axes are

- `Location`
- `Orientation`
- `EdgeColor`
- `TextColor`
- `Interpreter`
- `String`

Legends for graphs that contain groups of objects such as lineseries, barseries, contourgroups, etc. created by high-level plotting commands such as `plot`, `bar`, `contour`, etc., by default display a single legend entry for the entire group, regardless of how many member objects it contains. However, you can customize such legends to show individual entries for all or selected member objects and assign a unique `DisplayName` to any of them. You control how groups appear in the legend by setting values for their `Annotation` and `DisplayName` properties with M-code. For information and examples about customizing legends in this manner, see “Controlling Legends” in the MATLAB Graphics documentation.

You can specify `EdgeColor` and `TextColor` as RGB triplets or as `ColorSpecs`. You cannot set these colors to `'none'`. To hide the box surrounding a legend, set the `Box` property to `'off'`. To allow the background to show through the legend box, set the legend’s `Color` property to `'none'`, for example,

```
set(legend_handle, 'Box', 'off')
set(legend_handle, 'Color', 'none')
```

This is similar to the effect of the command `legend boxoff`, except that `boxoff` also hides the legend’s border.

You can use a legend's handle to set text properties for all the strings in a legend at once with a cell array of strings, rather than looping through each of them. See the last line of the example below, which demonstrates setting a legend's `Interpreter` property. In that example, you could reset the `String` property of the legend as follows:

```
set(h,'String',{'cos(x)','sin(x)'})
```

See the documentation for Text Properties for additional details.

`legend` installs a figure `ResizeFcn` if there is not already a user-defined `ResizeFcn` assigned to the figure. This `ResizeFcn` attempts to keep the legend the same size.

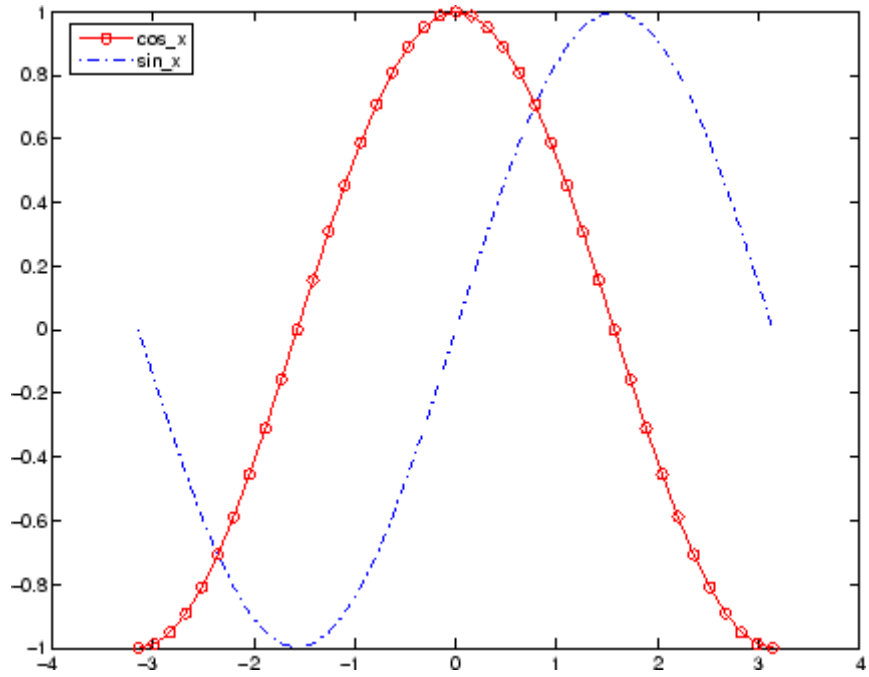
Moving the Legend

Move the legend by pressing the left mouse button while the cursor is over the legend and dragging the legend to a new location. Double-clicking a label allows you to edit the label.

Example

Add a legend to a graph showing a sine and cosine function:

```
x = -pi:pi/20:pi;
plot(x,cos(x),'-ro',x,sin(x),'-.b')
h = legend('cos_x','sin_x',2);
set(h,'Interpreter','none')
```



In this example, the `plot` command specifies a solid, red line (`'-r'`) for the cosine function and a dash-dot, blue line (`'-.b'`) for the sine function.

See Also

`LineSpec`, `plot`

"Adding a Legend to a Graph" for more information on using legends

"Annotating Plots" on page 1-87 for related functions

Purpose

Associated Legendre functions

Syntax

```
P = legendre(n,X)
S = legendre(n,X,'sch')
N = legendre(n,X,'norm')
```

Definitions

Associated Legendre Functions

The Legendre functions are defined by

$$P_n^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_n(x)$$

where

$$P_n(x)$$

is the Legendre polynomial of degree n.

$$P_n(x) = \frac{1}{2^n n!}\left[\frac{d^n}{dx^n}(x^2-1)^n\right]$$

Schmidt Seminormalized Associated Legendre Functions

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions $P_n^m(x)$ by

$P_n(x)$ for $m = 0$

$$S_n^m(x) = (-1)^m\sqrt{\frac{2(n-m)!}{(n+m)!}}\,P_n^m(x) \text{ for } m > 0.$$

Fully Normalized Associated Legendre Functions

The fully normalized associated Legendre functions are normalized such that

$$\int_{-1}^{1} \left(N_n^m(x)\right)^2 dx = 1$$

and are related to the unnormalized associated Legendre functions $P_n^m(x)$ by

$$N_n^m(x) = (-1)^m \sqrt{\frac{\left(n+\frac{1}{2}\right)(n-m)!}{(n+m)!}} P_n^m(x)$$

Description

`P = legendre(n,X)` computes the associated Legendre functions $P_n^m(x)$ of degree `n` and order `m = 0,1,...,n`, evaluated for each element of `X`. Argument `n` must be a scalar integer, and `X` must contain real values in the domain $-1 \le x \le 1$.

If `X` is a vector, then `P` is an `(n+1)`-by-`q` matrix, where `q = length(X)`. Each element `P(m+1,i)` corresponds to the associated Legendre function of degree `n` and order `m` evaluated at `X(i)`.

In general, the returned array `P` has one more dimension than `X`, and each element `P(m+1,i,j,k,...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(i,j,k,...)`. Note that the first row of `P` is the Legendre polynomial evaluated at `X`, i.e., the case where `m = 0`.

`S = legendre(n,X,'sch')` computes the Schmidt seminormalized associated Legendre functions $S_n^m(x)$.

`N = legendre(n,X,'norm')` computes the fully normalized associated Legendre functions $N_n^m(x)$.

Examples

Example 1

The statement `legendre(2,0:0.1:0.2)` returns the matrix

	x = 0	x = 0.1	x = 0.2
m = 0	-0.5000	-0.4850	-0.4400
m = 1	0	-0.2985	-0.5879
m = 2	3.0000	2.9700	2.8800

Example 2

Given,

```
X = rand(2,4,5);
n = 2;
P = legendre(n,X)
```

then

```
size(P)
ans =
     3     2     4     5
```

and

```
P(:,1,2,3)
ans =
   -0.2475
   -1.1225
    2.4950
```

is the same as

```
legendre(n,X(1,2,3))
ans =
   -0.2475
   -1.1225
    2.4950
```

Algorithm

`legendre` uses a three-term backward recursion relationship in `m`. This recursion is on a version of the Schmidt seminormalized associated

Legendre functions $Q_n^m(x)$, which are complex spherical harmonics. These functions are related to the standard Abramowitz and Stegun [1] functions $P_n^m(x)$ by

$$P_n^m(x) = \sqrt{\frac{(n+m)!}{(n-m)!}}\, Q_n^m(x)$$

They are related to the Schmidt form given previously by

$$S_n^m(x) = Q_n^0(x) \text{ for } m = 0$$
$$S_n^m(x) = (-1)^m \sqrt{2}\, Q_n^m(x) \text{ for } m > 0$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Ch.8.

[2] Jacobs, J. A., *Geomagnetism*, Academic Press, 1987, Ch.4.

Purpose

Length of vector

Syntax

```
n = length(X)
```

Description

The statement `length(X)` is equivalent to `max(size(X))` for nonempty arrays and `0` for empty arrays.

`n = length(X)` returns the size of the longest dimension of `X`. If `X` is a vector, this is the same as its length.

Examples

```
x = ones(1,8);
n = length(x)

n =
     8
x = rand(2,10,3);
n = length(x)

n =
    10
```

See Also

`ndims`, `size`

length (serial)

Purpose

Length of serial port object array

Syntax

```
length(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`length(obj)` returns the length of `obj`. It is equivalent to the command `max(size(obj))`.

See Also

Functions

`size`

Purpose

Length of time vector

Syntax

```
length(ts)
```

Description

`length(ts)` returns an integer that represents the length of the time vector for the `timeseries` object `ts`. It returns `0` if `ts` is empty.

See Also

`isempty (timeseries)`, `size (timeseries)`

length (tscollection)

Purpose

Length of time vector

Syntax

```
length(tsc)
```

Description

`length(tsc)` returns an integer that represents the length of the time vector for the `tscollection` object `tsc`.

See Also

`isempty (tscollection)`, `size (tscollection)`, `tscollection`

Purpose

Information on functions in external library

Syntax

```
m = libfunctions('libname')
m = libfunctions('libname', '-full')
libfunctions libname -full
```

Description

`m = libfunctions('libname')` returns the names of all functions defined in the external shared library, `libname`, that has been loaded into MATLAB with the `loadlibrary` function. The return value, `m`, is a cell array of strings.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

`m = libfunctions('libname', '-full')` returns a full description of the functions in the library, including function signatures. This includes duplicate function names with different signatures. The return value, `m`, is a cell array of strings.

`libfunctions libname -full` is the command format for this function.

Examples

List the functions in the MATLAB `libmx` library:

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile)

libfunctions libmx

Methods for class lib.libmx:
mxAddField               mxGetFieldNumber  mxIsLogicalScalarTrue
mxArrayToString          mxGetImagData     mxIsNaN
mxCalcSingleSubscript    mxGetInf          mxIsNumeric
mxCalloc                 mxGetIr           mxIsObject
mxClearScalarDoubleFlag  mxGetJc           mxIsOpaque
mxCreateCellArray        mxGetLogicals     mxIsScalarDoubleFlagSet
       .                        .                 .
       .                        .                 .
```

To list the functions along with their signatures, use the **-full** switch with `libfunctions`:

```
libfunctions libmx -full

Methods for class lib.libmx:
[mxClassID, MATLAB array] mxGetClassID(MATLAB array)
[lib.pointer, MATLAB array] mxGetData(MATLAB array)
[MATLAB array, voidPtr] mxSetData(MATLAB array, voidPtr)
[uint8, MATLAB array] mxIsNumeric(MATLAB array)
[uint8, MATLAB array] mxIsCell(MATLAB array)
[lib.pointer, MATLAB array] mxGetPr(MATLAB array)
[MATLAB array, doublePtr] mxSetPr(MATLAB array, doublePtr)
          .
          .

unloadlibrary libmx
```

See Also

`loadlibrary`, `libfunctionsview`, `libpointer`, `libstruct`, `calllib`, `libisloaded`, `unloadlibrary`

Purpose

Create window displaying information on functions in external library

Syntax

```
libfunctionsview libname
libfunctionsview libname
```

Description

`libfunctionsview libname` displays the names of the functions in the external shared library, `libname`, that has been loaded into MATLAB with the `loadlibrary` function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

MATLAB creates a new window in response to the `libfunctionsview` command. This window displays all of the functions defined in the specified library. For each of these functions, the following information is supplied:

- Data type returned by the function
- Name of the function
- Arguments passed to the function

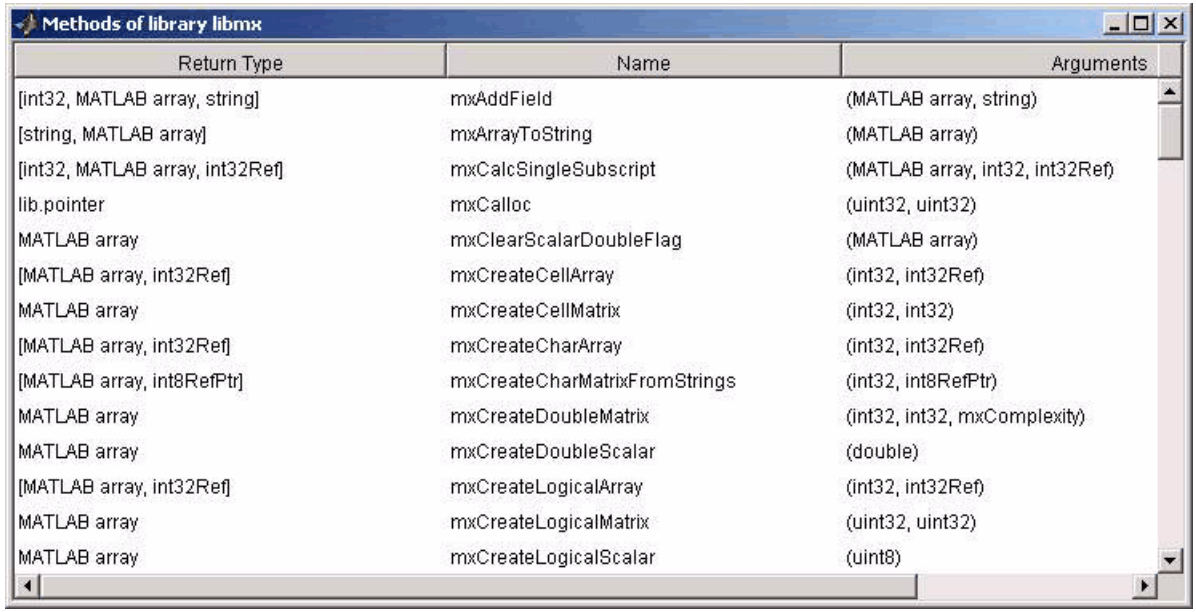
An additional column entitled “Inherited From” is displayed at the far right of the window. The information in this column is not useful for external libraries.

`libfunctionsview libname` is the command format for this function.

Examples

The following command opens the window shown below for the `libmx` library:

```
libfunctionsview libmx
```



See Also

`loadlibrary`, `libfunctions`, `libpointer`, `libstruct`, `calllib`, `libisloaded`, `unloadlibrary`

Purpose

Determine whether external library is loaded

Syntax

```
libisloaded('libname')
libisloaded libname
```

Description

`libisloaded('libname')` returns logical `1` (`true`) if the shared library `libname` is loaded and logical `0` (`false`) otherwise.

`libisloaded libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

Examples

Example 1

Load the `shrlibsample` library and check to see if the load was successful before calling one of its functions:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

if libisloaded('shrlibsample')
   x = calllib('shrlibsample', 'addDoubleRef', 1.78, 5.42, 13.3)
end
```

Since the library is successfully loaded, the call to `addDoubleRef` works as expected and returns

```
x =
   20.5000

unloadlibrary shrlibsample
```

Example 2

Load the same library, this time giving it an alias. If you use `libisloaded` with the library name, `shrlibsample`, it now returns `false`. Since you loaded the library using an alias, all further references to the library must also use that alias:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h alias lib

libisloaded shrlibsample
ans =
     0

libisloaded lib
ans =
     1

unloadlibrary lib
```

See Also

loadlibrary, libfunctions, libfunctionsview, libpointer, libstruct, calllib, unloadlibrary

Purpose

Create pointer object for use with external libraries

Syntax

```
p = libpointer
p = libpointer('type')
p = libpointer('type',value)
```

Description

`p = libpointer` returns an empty (void) pointer.

`p = libpointer('type')` returns an empty pointer that contains a reference to the specified data `type`. This `type` can be any MATLAB numeric type, or a structure or enumerated type defined in an external library that has been loaded into MATLAB with the `loadlibrary` function. For valid `types`, see the table under "Primitive Types" in the MATLAB External Interfaces documentation.

Note Using this syntax, `p` is a `NULL` pointer. You, therefore, must ensure that any library function to which you pass `p` must be able to accept a `NULL` pointer as an argument.

`p = libpointer('type',value)` returns a pointer to the specified data `type` and initialized to the `value` supplied.

Remarks

MATLAB automatically converts data passed to and from external library functions to the data type expected by the external function. The `libpointer` function enables you to convert your argument data manually. This is an advanced feature available to experienced C programmers. For more information about using pointer objects, see "Creating References" in the MATLAB External Interfaces documentation. Additional examples for using `libpointer` can be found in "Reference Pointers" in the same documentation.

Examples

This example passes an `int16` pointer to a function that multiplies each value in a matrix by its index. The function `multiplyShort` is defined in the MATLAB sample shared library, `shrlibsample`.

Here is the C function:

```
void multiplyShort(short *x, int size)
{
    int i;
    for (i = 0; i < size; i++)
    *x++ *= i;
}
```

Load the `shrlibsample` library. Create the matrix, `v`, and also a pointer to it, `pv`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

v = [4 6 8; 7 5 3];

pv = libpointer('int16Ptr', v);
get(pv, 'Value')
ans =
     4     6     8
     7     5     3
```

Now call the C function in the library, passing the pointer to `v`. If you were to pass a *copy* of `v`, the results would be lost once the function terminates. Passing a pointer to `v` enables you to get back the results:

```
calllib('shrlibsample', 'multiplyShort', pv, 6);
get(pv, 'Value')
ans =
     0    12    32
     7    15    15

unloadlibrary shrlibsample
```

See Also

`loadlibrary`, `libfunctions`, `libfunctionsview`, `libstruct`, `calllib`, `libisloaded`, `unloadlibrary`

Purpose

Construct structure as defined in external library

Syntax

```
s = libstruct('structtype')
s = libstruct('structtype',mlstruct)
```

Description

`s = libstruct('structtype')` returns a `libstruct` object `s` that is a MATLAB object designed to resemble a C structure of type specified by `structtype`. The structure type, `structtype`, is defined in an external library that must be loaded into MATLAB using the `loadlibrary` function.

Note Using this syntax, `s` is a `NULL` pointer. You, therefore, must ensure that any library function to which you pass `s` must be able to accept a `NULL` pointer as an argument.

`s = libstruct('structtype',mlstruct)` returns a `libstruct` object `s` with its fields initialized from MATLAB structure, `mlstruct`.

The `libstruct` function essentially creates a C-like structure that you can pass to functions in an external library. You can handle this structure in MATLAB as you would a true MATLAB structure.

What Data Types Are Available

To determine which MATLAB data types to use when passing arguments to library functions, see the output of `libfunctionsview` or `libfunctions -full`. These functions list all of the functions found in a particular library along with a specification of the data types required for each argument.

Examples

This example performs a simple addition of the fields of a structure. The function `addStructFields` is defined in the MATLAB sample shared library, `shrlibsample`.

Here is the C function:

```
double addStructFields(struct c_struct st)
```

```
{
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

Start by loading the `shrlibsample` library and creating MATLAB structure, `sm`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
```

Construct a `libstruct` object `sc` that uses the `c_struct` template:

```
sc = libstruct('c_struct', sm);

get(sc)
    p1: 476
    p2: -299
    p3: 1000
```

Now call the function, passing the `libstruct` object, `sc`:

```
calllib('shrlibsample', 'addStructFields', sc)
ans =
    1177
```

You must clear the `libstruct` object before unloading the library:

```
clear sc
unloadlibrary shrlibsample
```

Note In most cases, you can pass a MATLAB structure and MATLAB automatically converts the argument to a C structure. See “Structures” in the MATLAB External Interfaces documentation for more information.

See Also

`loadlibrary`, `libfunctions`, `libfunctionsview`, `libpointer`, `calllib`, `libisloaded`, `unloadlibrary`

license

Purpose

Return license number or perform licensing task

Syntax

```
license
license('inuse')
S = license('inuse')
S = license('inuse', feature)
license('test',feature)
license('test',feature,toggle)
result = license('checkout',feature)
```

Description

`license` returns the license number for this MATLAB. The return value is always a string but is not guaranteed to be a number. The following table lists text strings that `license` can return.

String	Description
`'demo'`	MATLAB is a demonstration version
`'student'`	MATLAB is the student version
`'unknown'`	License number cannot be determined

`license('inuse')` returns a list of licenses checked out in the current MATLAB session. In the list, products are listed alphabetically by their license feature names, i.e., the text string used to identify products in the `INCREMENT` lines in a License File (`license.dat`). Note that the feature names returned in the list contain only lower-case characters.

`S = license('inuse')` returns an array of structures, where each structure represents a checked-out license. The structures contains two fields: feature and user. The feature field contains the license feature name. The user field contains the username of the person who has the license checked out.

`S = license('inuse', feature)` checks if the product specified by the text string `feature` is checked out in the current MATLAB session. If the product is checked out, the `license` function returns the product name and the username of the person who has it checked out in the

structure S. If the product is not currently checked out, the fields in the structure are empty.

The feature string must be a license feature name, spelled exactly as it appears in the INCREMENT lines in a License File. For example, the string 'Identification_Toolbox' is the feature name for the System Identification Toolbox. The feature string is not case-sensitive and must not exceed 27 characters.

license('test',feature) tests if a license exists for the product specified by the text string feature. The license command returns 1 if the license exists and 0 if the license does not exist. The feature string identifies a product, as described in the previous syntax.

Note Testing for a license only confirms that the license exists. It does not confirm that the license can be checked out. For example, license will return 1 if a license exists, even if the license has expired or if a system administrator has excluded you from using the product in an options file.

---

license('test',feature,toggle) enables or disables testing of the product specified by the text string feature, depending on the value of toggle. The parameter toggle can have either of two values:

'enable' The syntax license('test',feature) returns 1 if the product license exists and 0 if the product license does not exist.

'disable' The syntax license('test',feature) always returns 0 (product license does not exist) for the specified product.

Note Disabling a test for a particular product can impact other tests for the existence of the license, not just tests performed using the license command.

`result = license('checkout',feature)` checks out a license for the product identified by the text string `feature`. The `license` command returns `1` if it could check out a license for the product and `0` if it could not check out a license for the product.

Examples

Get the license number for this MATLAB.

```
license
```

Get a list of licenses currently being used. Note that the products appear in alphabetical order by their license feature name in the list returned.

```
license('inuse')

image_toolbox
map_toolbox
matlab
```

Get a list of licenses in use with information about who is using the license.

```
S = license('inuse');

S(1)

ans =

    feature: 'image_toolbox'
       user: 'juser'
```

Determine if the license for MATLAB is currently in use.

```
S = license('inuse','MATLAB')

S =

    feature: 'matlab'
       user: 'jsmith'
```

Determine if a license exists for the Mapping Toolbox.

```
license('test','map_toolbox')

ans =

    1
```

Check out a license for the Control System Toolbox.

```
license('checkout','control_toolbox')

ans =

    1
```

Determine if the license for the Control System Toolbox is checked out.

```
license('inuse')

control_toolbox
image_toolbox
map_toolbox
matlab
```

See Also

`isstudent`

light

Purpose

Create light object

Syntax

```
light('PropertyName',propertyvalue,...)
handle = light(...)
```

Description

`light` creates a light object in the current axes. Lights affect only patch and surface objects.

`light('PropertyName',propertyvalue,...)` creates a light object using the specified values for the named properties. MATLAB parents the light to the current axes unless you specify another axes with the `Parent` property.

`handle = light(...)` returns the handle of the light object created.

Remarks

You cannot see a light object *per se*, but you can see the effects of the light source on patch and surface objects. You can also specify an axes-wide ambient light color that illuminates these objects. However, ambient light is visible only when at least one light object is present and visible in the axes.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see `set` and `get` for examples of how to specify these data types).

See also the patch and surface `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, `SpecularColorReflectance`, and `VertexNormals` properties. Also see the `lighting` and `material` commands.

Examples

Light the `peaks` surface plot with a light source located at infinity and oriented along the direction defined by the vector `[1 0 0]`, that is, along the *x*-axis.

```
h = surf(peaks);
set(h,'FaceLighting','phong','FaceColor','interp',...
      'AmbientStrength',0.5)
light('Position',[1 0 0],'Style','infinite');
```

Object Hierarchy

Setting Default Properties

You can set default light properties on the axes, figure, and root levels:

```
set(0,'DefaultLightProperty',PropertyValue...)
set(gcf,'DefaultLightProperty',PropertyValue...)
set(gca,'DefaultLightProperty',PropertyValue...)
```

where *Property* is the name of the light property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access light properties.

See Also

`lighting`, `material`, `patch`, `surface`

"Lighting as a Visualization Tool" for more information about lighting

"Lighting" on page 1-101 for related functions

Light Properties for property descriptions

Light Properties

Purpose

Light properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Light Property Descriptions

This section lists property names along with the type of values each accepts.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted and, therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
function handle

This property is not used on lights.

`Children`
handles

The empty matrix; light objects have no children.

`Clipping`
`on | off`

`Clipping` has no effect on light objects.

`Color`
`ColorSpec`

Color of light. This property defines the color of the light emanating from the light object. Define it as a three-element RGB vector or one of the MATLAB predefined names. See the `ColorSpec` reference page for more information.

`CreateFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during object creation. A callback function that executes when MATLAB creates a light object. You must define this property as a default value for lights or in a call to the `light` function to create a new light object. For example, the following statement:

```
set(0,'DefaultLightCreateFcn',@light_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the current figure colormap to `gray` and uses a reddish light color whenever you create a light object.

```
function light_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 set(src,'Color',[.9 .2 .2])
 set(gcbf,'Colormap',gray)
end
```

MATLAB executes this function after setting all light properties. Setting this property on an existing light object has no effect. The function must define at least two input arguments (handle of light object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended

Delete light callback function. A callback function that executes when you delete the light object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in

its parent's list of children. HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close.

When a handle's visibility is restricted using callback or off, the object's handle does not appear in its parent's Children property, figures do not appear in the root's CurrentFigure property, objects do not appear in the root's CallbackObject property or in the figure's CurrentObject property, and axes do not appear in their parent's CurrentAxes property.

You can set the root ShowHiddenHandles property to on to make all handles visible regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

This property is not used by light objects.

`Interruptible`
`{on} | off`

Callback routine interruption mode. Light object callback routines defined for the `DeleteFcn` property are not affected by the `Interruptible` property.

`Parent`
handle of parent axes

Parent of light object. This property contains the handle of the light object's parent. The parent of a light object is the `axes` object that contains it.

Note that light objects cannot be parented to hggroup or hgtransform objects.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Position`
`[x,y,z]` in axes data units

Location of light object. This property specifies a vector defining the location of the light object. The vector is defined from the origin to the specified *x*-, *y*-, and *z*-coordinates. The placement of the light depends on the setting of the `Style` property:

- If the `Style` property is set to `local`, `Position` specifies the actual location of the light (which is then a point source that radiates from the location in all directions).
- If the `Style` property is set to `infinite`, `Position` specifies the direction from which the light shines in parallel rays.

`Selected`
`on | off`

This property is not used by light objects.

`SelectionHighlight`
`{on} | off`

This property is not used by light objects.

`Style`
`{infinite} | local`

Parallel or divergent light source. This property determines whether MATLAB places the light object at infinity, in which case the light rays are parallel, or at the location specified by the `Position` property, in which case the light rays diverge in all directions. See the `Position` property.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For light objects, `Type` is always `'light'`.

`UIContextMenu`
handle of a uicontextmenu object

This property is not used by light objects.

`UserData`
matrix

User-specified data. This property can be any data you want to associate with the light object. The light does not use this property, but you can access it using `set` and `get`.

`Visible`
`{on} | off`

Light visibility. While light objects themselves are not visible, you can see the light on patch and surface objects. When you set `Visible` to `off`, the light emanating from the source is not visible. There must be at least one light object in the axes whose `Visible` property is `on` for any lighting features to be enabled (including the axes `AmbientLightColor` and patch and surface `AmbientStrength`).

lightangle

Purpose

Create or position `light` object in spherical coordinates

Syntax

```
lightangle(az,el)
light_handle = lightangle(az,el)
lightangle(light_handle,az,el)
[az,el] = lightangle(light_handle)
```

Description

`lightangle(az,el)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `el` is the vertical elevation (both in degrees). The interpretation of azimuth and elevation is the same as that of the `view` command.

`light_handle = lightangle(az,el)` creates a light and returns the handle of the light in `light_handle`.

`lightangle(light_handle,az,el)` sets the position of the light specified by `light_handle`.

`[az,el] = lightangle(light_handle)` returns the azimuth and elevation of the light specified by `light_handle`.

Remarks

By default, when a light is created, its style is `infinite`. If the light handle passed in to `lightangle` refers to a local light, the distance between the light and the camera target is preserved as the position is changed.

Examples

```
surf(peaks)
axis vis3d
h = light;
for az = -50:10:50
 lightangle(h,az,30)
 drawnow
end
```

See Also

`light`, `camlight`, `view`

"Lighting as a Visualization Tool" for more information about lighting

"Lighting" on page 1-101 for related functions

Purpose

Specify lighting algorithm

Syntax

```
lighting flat
lighting gouraud
lighting phong
lighting none
```

Description

`lighting` selects the algorithm used to calculate the effects of `light` objects on all `surface` and `patch` objects in the current axes.

`lighting flat` selects flat lighting.

`lighting gouraud` selects gouraud lighting.

`lighting phong` selects phong lighting.

`lighting none` turns off lighting.

Remarks

The `surf`, `mesh`, `pcolor`, `fill`, `fill3`, `surface`, and `patch` functions create graphics objects that are affected by light sources. The `lighting` command sets the `FaceLighting` and `EdgeLighting` properties of surfaces and patches appropriately for the graphics object.

See Also

`light`, `material`, `patch`, `surface`

"Lighting as a Visualization Tool" for more information about lighting

"Lighting" on page 1-101 for related functions

lin2mu

Purpose

Convert linear audio signal to mu-law

Syntax

```
mu = lin2mu(y)
```

Description

`mu = lin2mu(y)` converts linear audio signal amplitudes in the range `-1 ≤ Y ≤ 1` to mu-law encoded "flints" in the range `0 ≤ u ≤ 255`.

See Also

`auwrite`, `mu2lin`

Purpose

Create line object

Syntax

```
line(X,Y)
line(X,Y,Z)
line(X,Y,Z,'PropertyName',propertyvalue,...)
line('XData',x,'YData',y,'ZData',z,...)
h = line(...)
```

Description

`line` creates a line object in the current axes. You can specify the color, width, line style, and marker type, as well as other characteristics.

The `line` function has two forms:

- Automatic color and line style cycling. When you specify matrix coordinate data using the informal syntax (i.e., the first three arguments are interpreted as the coordinates),

  ```
  line(X,Y,Z)
  ```

  MATLAB cycles through the axes `ColorOrder` and `LineStyleOrder` property values the way the `plot` function does. However, unlike `plot`, `line` does not call the `newplot` function.

- Purely low-level behavior. When you call `line` with only property name/property value pairs,

  ```
  line('XData',x,'YData',y,'ZData',z)
  ```

  MATLAB draws a line object in the current axes using the default line color (see the `colordef` function for information on color defaults). Note that you cannot specify matrix coordinate data with the low-level form of the `line` function.

`line(X,Y)` adds the line defined in vectors `X` and `Y` to the current axes. If `X` and `Y` are matrices of the same size, `line` draws one line per column.

`line(X,Y,Z)` creates lines in three-dimensional coordinates.

`line(X,Y,Z,'PropertyName',propertyvalue,...)` creates a line using the values for the property name/property value pairs specified and default values for all other properties.

See the `LineStyle` and `Marker` properties for a list of supported values.

`line('XData',x,'YData',y,'ZData',z,...)` creates a line in the current axes using the property values defined as arguments. This is the low-level form of the `line` function, which does not accept matrix coordinate data as the other informal forms described above.

`h = line(...)` returns a column vector of handles corresponding to each line object the function creates.

Remarks

In its informal form, the `line` function interprets the first three arguments (two for 2-D) as the `X`, `Y`, and `Z` coordinate data, allowing you to omit the property names. You must specify all other properties as name/value pairs. For example,

```
line(X,Y,Z,'Color','r','LineWidth',4)
```

The low-level form of the `line` function can have arguments that are only property name/property value pairs. For example,

```
line('XData',x,'YData',y,'ZData',z,'Color','r','LineWidth',4)
```

Line properties control various aspects of the line object and are described in the "Line Properties" section. You can also set and query property values after creating the line using `set` and `get`.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

Unlike high-level functions such as `plot`, `line` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds line objects to the current axes. However, axes properties that are under automatic control, such as the axis limits, can change to accommodate the line within the current axes.

Connecting the dots

The coordinate data is interpreted as vectors of corresponding x, y, and z values:

```
X = [x(1) x(2) x(3)...x(n)]
Y = [y(1) x(2) y(3)...y(n)]
Z = [z(1) z(2) x(3)...z(n)]
```

where a point is determined by the corresponding vector elements:

```
p1(x(i),y(i),z(i))
```

For example, to draw a line from the point located at `x = .3` and `y = .4` and `z = 1` to the point located at `x = .7` and `y = .9` and `z = 1`, use the following data:

```
axis([0 1 0 1])
line([.3 .7],[.4 .9],[1 1],'Marker','.','LineStyle','-')
```

Examples

This example uses the `line` function to add a shadow to plotted data. First, plot some data and save the line's handle:

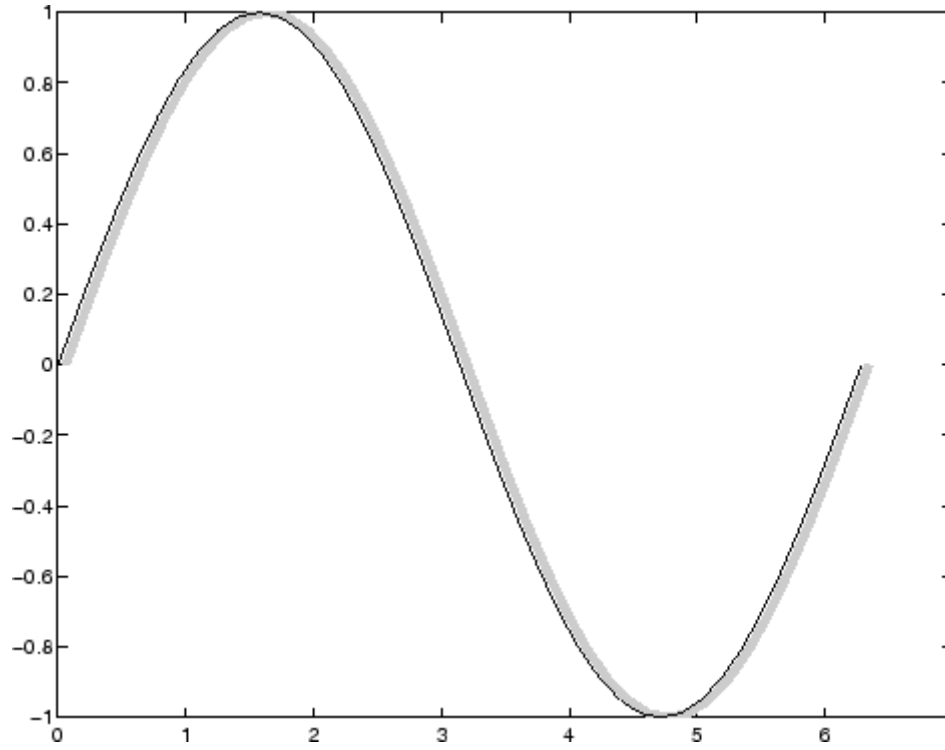
```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t),'k');
```

Next, add a shadow by offsetting the *x*-coordinates. Make the shadow line light gray and wider than the default `LineWidth`:

```
hline2 = line(t+.06,sin(t),'LineWidth',4,'Color',[.8 .8 .8]);
```

Finally, pop the first line to the front:

```
set(gca,'Children',[hline1 hline2])
```



Drawing Lines Interactively

You can use the `ginput` function to select points from a figure. For example:

```
axis([0 1 0 1])
for n = 1:5
 [x(n),y(n)] = ginput(1);
end
line(x,y)
```

The `for` loop enables you to select five points and build the `x` and `y` arrays. Because `line` requires arrays of corresponding x and y coordinates, you can just pass these arrays to the `line` function.

Drawing with mouse motion

You can use the axes `CurrentPoint` property and the figure `WindowButtonDownFcn` and `WindowButtonMotionFcn` properties to select a point with a mouse click and draw a line to another point by dragging the mouse, like a simple drawing program. The following example illustrates a few useful techniques for doing this type of interactive drawing.

Click to view in editor — This example enables you to click and drag the cursor to draw lines.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

Input Argument Dimensions — Informal Form

This statement reuses the one-column matrix specified for `ZData` to produce two lines, each having four points.

```
line(rand(4,2),rand(4,2),rand(4,1))
```

If all the data has the same number of columns and one row each, MATLAB transposes the matrices to produce data for plotting. For example,

```
line(rand(1,4),rand(1,4),rand(1,4))
```

is changed to

```
line(rand(4,1),rand(4,1),rand(4,1))
```

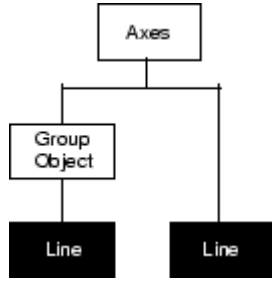
This also applies to the case when just one or two matrices have one row. For example, the statement

```
line(rand(2,4),rand(2,4),rand(1,4))
```

is equivalent to

```
line(rand(4,2),rand(4,2),rand(4,1))
```

Object Hierarchy



Setting Default Properties

You can set default line properties on the axes, figure, and root levels:

```
set(0,'DefaultLine<PropertyName>',<Property>Value,...)
set(gcf,'DefaultLine<PropertyName>',<Property>Value,...)
set(gca,'DefaultLine<PropertyName>',<Property>Value,...)
```

Where *`PropertyName`* is the name of the line property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access line properties.

See Also

`annotationaxes`, `newplot`, `plot`, `plot3`

"Object Creation Functions" on page 1-94 for related functions

Line Properties for property descriptions

Purpose

Line properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See Core Graphics Objects for general information about this type of object.

Line Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of line objects in legends. The `Annotation` property enables you to specify whether this line object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the line object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this line object in a legend (default)
`off`	Do not include this line object in a legend
`children`	Same as `on` because line objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted

and, therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the line object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of line associated with the button down event and an event structure, which is empty for this property)

The following example shows how to access the callback object's handle as well as the handle of the figure that contains the object from the callback function.

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a line object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`Children`
vector of handles

The empty matrix; line objects have no children.

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips lines to the axes plot box by default. If you set `Clipping` to `off`, lines are displayed outside the axes plot box. This can occur if you create a line, set `hold` to `on`, freeze axis scaling (set `axis` to `manual`), and then create a longer line.

`Color`
`ColorSpec`

Line color. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during object creation. A callback function that executes when MATLAB creates a line object. You must define this property as a default value for lines or in a call to the `line` function to create a new line object. For example, the statement

```
set(0,'DefaultLineCreateFcn',@line_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the axes `LineStyleOrder` whenever you create a line object. The callback function must be on your MATLAB path when you execute the above statement.

```
function line_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 axh = get(src,'Parent');
 set(axh,'LineStyleOrder','-.|--')
end
```

MATLAB executes this function after setting all line properties. Setting this property on an existing line object has no effect. The

function must define at least two input arguments (handle of line object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DeleteFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Delete line callback function. A callback function that executes when you delete the line object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of line object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DisplayName`
string (default is empty string)

String used by legend for this line object. The `legend` function uses the string defined by the `DisplayName` property to label this line object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this line object’s corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

The following code shows how to use the `DisplayName` property from the command line or in an M-file.

```
t = 0:.1:2*pi;
a(:,1)=sin(t); a(:,2)=cos(t);
```

```
h = plot(a);
set(h,{'DisplayName'},{'Sine','Cosine'}')
legend show
```

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase line objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the line when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it, because MATLAB stores no information about its former location.
- `xor` — Draw and erase the line by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the line. However, the line's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the line by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased line, but lines are always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing nonnormal mode objects.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines if the line can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line. If `HitTest` is `off`, clicking the line selects the object below it (which may be the axes containing it).

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from

command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`Interruptible`

```
{on} | off
```

Callback routine interruption mode. The `Interruptible` property controls whether a line callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible`

property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style. This property specifies the line style. Available line styles are shown in the table.

Symbol	Line Style
`' '`	Solid line (default)
`'--'`	Dashed line
`':'`	Dotted line
`' .'`	Dash-dot line
`'none'`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of the line object. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`

character (see table)

Marker symbol. The `Marker` property specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the table.

Marker Specifier	Description
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the line's `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the

four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or the figure color, if the axes `Color` property is set to `none` (which is the factory default for axes).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of line object. This property contains the handle of the line object's parent. The parent of a line object is the axes that contains it. You can reparent line objects to other axes, hggroup, or hgtransform objects.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | off`

Is object selected? When this property is `on`. MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing

handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`
string (read only)

Class of graphics object. For line objects, `Type` is always the string `'line'`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the line. Assign this property the handle of a uicontextmenu object created in the same figure as the line. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the line.

`UserData`
matrix

User-specified data. Any data you want to associate with the line object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Line visibility. By default, all lines are visible. When set to `off`, the line is not visible, but still exists, and you can `get` and `set` its properties.

`XData`
vector of coordinates

X-coordinates. A vector of *x*-coordinates defining the line. `YData` and `ZData` must be the same length and have the same number of rows. (See "Examples" on page 2-1951.)

`YData`
vector of coordinates

Y-coordinates. A vector of *y*-coordinates defining the line. `XData` and `ZData` must be the same length and have the same number of rows.

`ZData`
vector of coordinates

Z-coordinates. A vector of *z*-coordinates defining the line. `XData` and `YData` must have the same number of rows.

Lineseries Properties

Purpose

Define lineseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

See “Plot Objects” for more information on lineseries objects.

Note that you cannot define default properties for lineseries objects.

Lineseries Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of lineseries objects in legends. The `Annotation` property enables you to specify whether this lineseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the lineseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the lineseries object in a legend as one entry, but not its children objects
`off`	Do not include the lineseries or its children in a legend (default)
`children`	Include only the children of the lineseries as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See "Controlling Legends" for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See "Function Handle Callbacks" for information on how to use function handles to define the callbacks.

`Children`
vector of handles

The empty matrix; line objects have no children.

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this lineseries object. The `legend` function uses the string defined by the `DisplayName` property to label this lineseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this lineseries object’s corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object

based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.

- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of

the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.

- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

```
{on} | callback | off
```

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`
`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the

`Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`

string (read only)

Class of graphics object. For lineseries objects, `Type` is always the string `line`.

`UIContextMenu`

handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the

context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
vector or matrix

The x-axis values for a graph. The *x*-axis values for graphs are specified by the `X` input argument. If `XData` is a vector, `length(XData)` must equal `length(YData)` and must be monotonic. If `XData` is a matrix, `size(XData)` must equal `size(YData)` and each column must be monotonic.

You can use `XData` to define meaningful coordinates for an underlying surface whose topography is being mapped. See "Setting the Axis Limits on Contour Plots" on page 2-640 for more information.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `x` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
vector or matrix of coordinates

Y-coordinates. A vector of *y*-coordinates defining the values along the *y*-axis for the graph. `XData` and `ZData` must be the same length and have the same number of rows.

`YDataSource`
string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`
vector of coordinates

Z-coordinates. A vector defining the *z*-coordinates for the graph. `XData` and `YData` must be the same length and have the same number of rows.

`ZDataSource`
string (MATLAB variable)

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `ZData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Line specification string syntax

GUI Alternative

To modify the style, width, and color of lines on a graph, use the Property Editor, one of the plotting tools  . For details, see The Property Editor in the MATLAB Graphics documentation.

Description

This page describes how to specify the properties of lines used for plotting. MATLAB gives you control over these graphic characteristics:

- Line style
- Line width
- Color
- Marker type
- Marker size
- Marker face and edge coloring (for filled markers)

You indicate the line styles, marker types, and colors you want to display to MATLAB using *string specifiers*, detailed in the following tables:

Line Style Specifiers

Specifier	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line

Marker Specifiers

Specifier	Marker Type
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`'square' or s`	Square
`'diamond' or d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`'pentagram' or p`	Five-pointed star (pentagram)
`'hexagram' or h`	Six-pointed star (hexagram)

Color Specifiers

Specifier	Color
`r`	Red
`g`	Green
`b`	Blue
`c`	Cyan
`m`	Magenta
`y`	Yellow

Specifier	Color
k	Black
w	White

All high-level plotting functions (except for the ez... family of function-plotting functions) accept a LineSpec argument that defines three components used to specify lines:

- Line style
- Marker symbol
- Color

For example,

```
plot(x,y,'-.or')
```

plots y versus x using a dash-dot line (-.), places circular markers (o) at the data points, and colors both line and marker red (r). Specify the components (in any order) as a quoted string after the data arguments. Note that linespecs are single strings, not property-value pairs.

Plotting Data Points with No Line

If you specify a marker, but not a line style, MATLAB plots only the markers. For example,

```
plot(x,y,'d')
```

Related Properties

When using the plot and plot3 functions, you can also specify other characteristics of lines using graphics properties:

- LineWidth — Specifies the width (in points) of the line
- MarkerEdgeColor — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles)

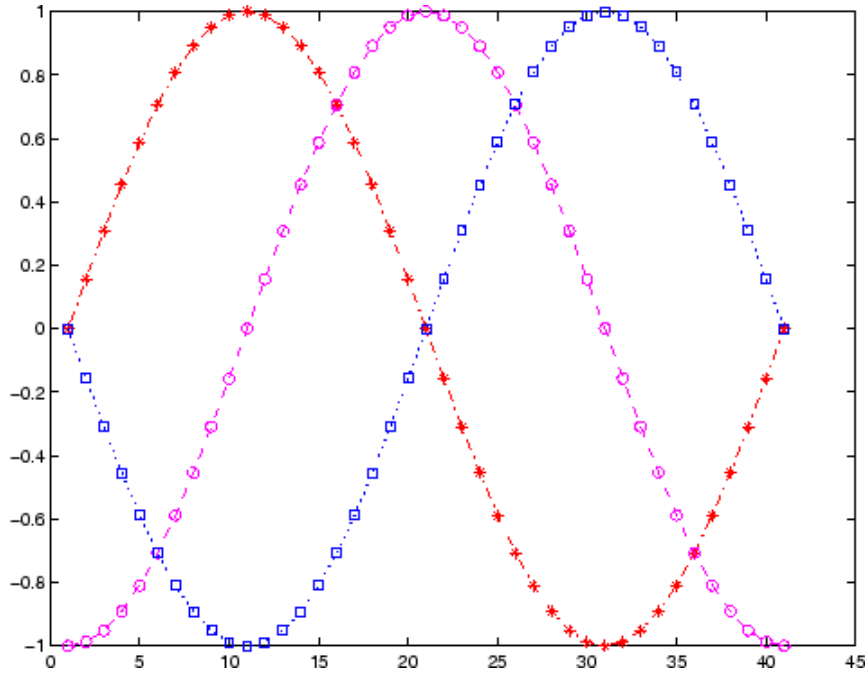
- `MarkerFaceColor` — Specifies the color of the face of filled markers
- `MarkerSize` — Specifies the size of the marker in points

In addition, you can specify the `LineStyle`, `Color`, and `Marker` properties instead of using the symbol string. This is useful if you want to specify a color that is not in the list by using RGB values. See Line Properties for details on these properties and `ColorSpec` for more information on color.

Examples

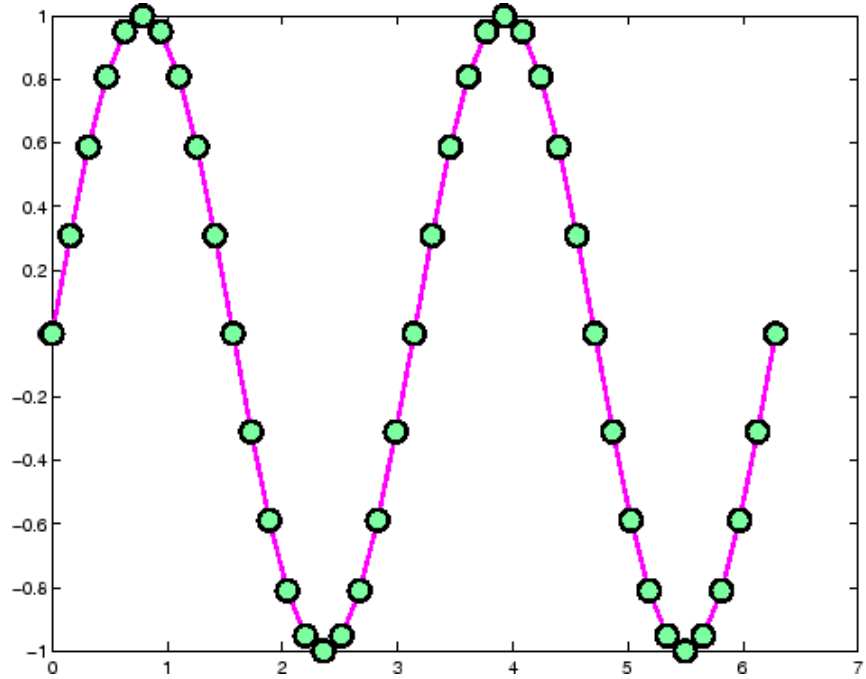
Plot the sine function over three different ranges using different line styles, colors, and markers.

```
t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*')
hold on
plot(t,sin(t-pi/2),'--mo')
plot(t,sin(t-pi),':bs')
hold off
```



Create a plot illustrating how to set line properties.

```
plot(t,sin(2*t),'-mo',...
                'LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor',[.49 1 .63],...
                'MarkerSize',12)
```



See Also

`line`, `plot`, `patch`, `set`, `surface`, `axes` , `Line Properties`, `ColorSpec`

"Line Styles Used for Plotting — LineStyleOrder" for information about defining an order for applying linestyles

"Types of Plots Available in MATLAB" for functions that use linespecs

"Basic Plots and Graphs" on page 1-86 for related functions

Purpose

Synchronize limits of specified 2-D axes

Syntax

```
linkaxes(axes_handles)
linkaxes(axes_handles,'option')
```

Description

Use `linkaxes` to synchronize the individual axis limits across several figures or subplots within a figure. Calling `linkaxes` will make all input axes have identical limits. Linking axes is most useful when you want to zoom or pan in one subplot and display the same range of data in another subplot.

`linkaxes(axes_handles)` links the *x*- and *y*-axis limits of the axes specified in the vector `axes_handles`. You can link any number of existing plots or subplots.

`linkaxes(axes_handles,'option')` links the axes' `axes_handles` according to the specified option. The *`option`* argument can be one of the following strings:

`x`	Link *x*-axis only
`y`	Link *y*-axis only
`xy`	Link *x*-axis and *y*-axis
`off`	Remove linking

See the `linkprop` function for more advanced capabilities that allow linking object properties on any graphics object.

Remarks

The first axes provided to `linkaxes` determines the x-limits and y-limits for all axes linked. This can cause plots to partly or entirely disappear if their limits or scaling are very different. To override this behavior, after calling `linkaxes` specify the limits of the axes that you wish to control with the `set` command, as shown in Example 3, below.

Examples

You can use interactive zooming or panning (selected from the figure toolbar) to see the effect of axes linking. For example, pan in one graph and notice how the *x*-axis also changes in the other. The axes

will respond in the same way to `zoom` and `pan` directives typed in the Command Window.

Example 1

This example creates two subplots and links the *x*-axis limits of the two axes:

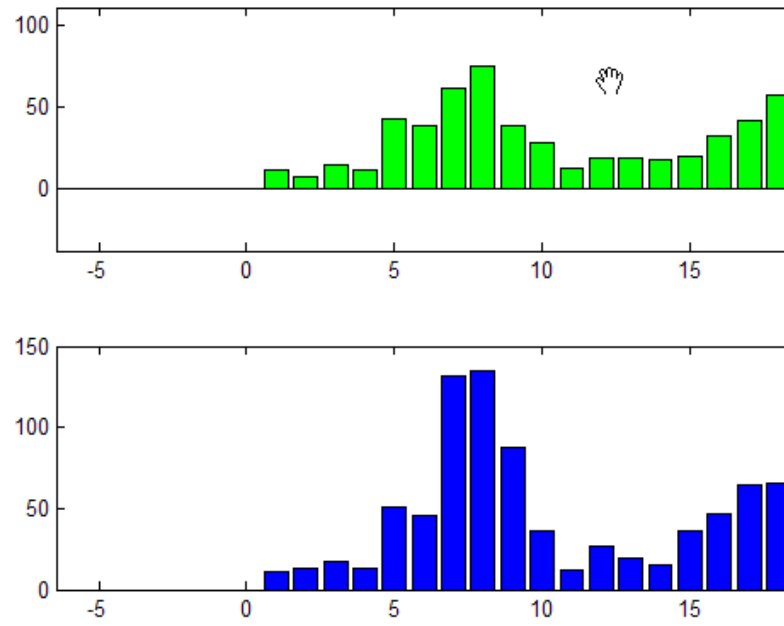
```
ax(1) = subplot(2,2,1);
plot(rand(1,10)*10,'Parent',ax(1));
ax(2) = subplot(2,2,2);
plot(rand(1,10)*100,'Parent',ax(2));
linkaxes(ax,'x');
```

Example 2

This example creates two figures and links the *x*-axis limits of the two axes. The illustration shows the effect of manually panning the top subplot:

```
load count.dat
figure; ax(1) = subplot(2,1,1);
h(1) = bar(ax(1),count(:,1),'g');
ax(2) = subplot(2,1,2);
h(2) = bar(ax(2),count(:,2),'b');
linkaxes(ax,'x');
```

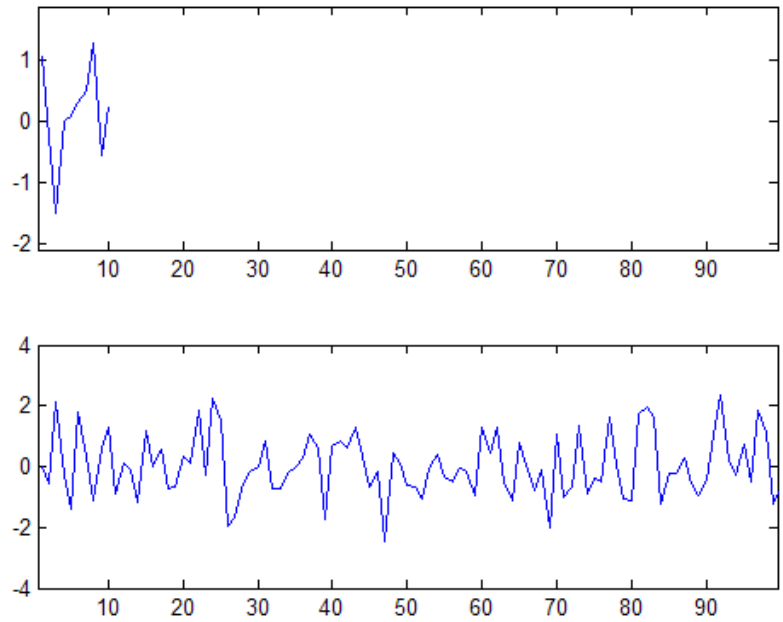
Choose the Pan tool (**Tools** ⇒ **Pan**) and drag the top axes. Both axes will pan in step in *x*, but only the top one pans in *y*.



Example 3

Create two subplots containing data having different ranges. The first axes handle passed to `linkaxes` determines the data range for all other linked axes. In this example, calling `set` for the lower axes overrides the *x*-limits established by the call to `linkaxes`:

```
a1 = subplot(2,1,1);
plot(randn(10,1));          % Plot 10 numbers on top
a2 = subplot(2,1,2);
plot(a2,randn(100,1))       % Plot 100 numbers below
linkaxes([a1 a2], 'x');     % Link the axes; subplot 2 now out of range
set(a2,'xlimmode','auto'); % Now both axes run from 1-100 in x
                            % You could also set(a2,'xlim',[1 100])
```



See Also

`linkprop`, `zoom`, `pan`

Purpose

Keep same value for corresponding properties

Syntax

```
hlink = linkprop(obj_handles,'PropertyName')
hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})
```

Description

Use `linkprop` to maintain the same values for the corresponding properties of different objects.

`hlink = linkprop(obj_handles,'PropertyName')` maintains the same value for the property *`PropertyName`* on all objects whose handles appear in `obj_handles`. `linkprop` returns the link object in `hlink`. See "Link Object" on page 2-1997 for more information.

`hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})` maintains the same respective values for all properties passed as a cell array on all objects whose handles appear in `obj_handles`.

Note that the linked properties of all linked objects are updated immediately when `linkprop` is called. The first object in the list (`obj_handles`) determines the property values for the rest of the objects.

Link Object

The mechanism to link the properties of different graphics objects is stored in the link object, which is returned by `linkprop`. Therefore, the link object must exist within the context where you want property linking to occur (such as in the base workspace if users are to interact with the objects from the command line or figure tools).

The following list describes ways to maintain a reference to the link object.

- Return the link object as an output argument from a function and keep it in the base workspace while interacting with the linked objects.
- Make the `hlink` variable global.

- Store the `hlink` variable in an object's `UserData` property or in application data. See the "Examples" on page 2-1998 section for an example that uses application data.

Modifying Link Object

If you want to change either the graphics objects or the properties that are linked, you need to use the link object methods designed for that purpose. These methods are functions that operate only on link objects. To use them, you must first create a link object using `linkprop`.

Method	Purpose
`addtarget`	Add specified graphics object to the link object's targets.
`removetarget`	Remove specified graphics object from the link object's targets.
`addprop`	Add specified property to the linked properties.
`removeprop`	Remove specified property from the linked properties.

Method Syntax

```
addtarget(hlink,obj_handles)
removetarget(hlink,obj_handles)
addprop(hlink,'PropertyName')
removeprop(hlink,'PropertyName')
```

Arguments

- `hlink` — Link object returned by `linkprop`
- `obj_handles` — One or more graphic object handles
- *`PropertyName`* — Name of a property common to all target objects

Examples

This example creates four isosurface graphs of fluid flow data, each displaying a different isovalue. The `CameraPosition` and `CameraUpVector` properties of each subplot axes are linked so that the user can rotate all subplots in unison.

After running the example, select **Rotate 3D** from the figure **Tools** menu and observe how all subplots rotate together.

Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

The property linking code is in step 3.

1. Define the data using the `flow` M-file and specify property values for the isosurface (which is a patch object).

```
function linkprop_example
[x y z v] = flow;
isoval = [-3 -1 0 1];
props.FaceColor = [0 0 .5];
props.EdgeColor = 'none';
props.AmbientStrength = 1;
props.FaceLighting = 'gouraud';
```

2. Create four subplot axes and add an isosurface graph to each one. Add a title and set viewing and lighting parameters using a local function (`set_view`). (`subplot`, `patch`, `isosurface`, `title`, `num2str`)

```
for k = 1:4
    h(k) = subplot(2,2,k);
    patch(isosurface(x,y,z,v,isoval(k)),props)
    title(h(k),['Isovalue = ',num2str(k)])
    set_view(h(k))
end
```

3. Link the `CameraPosition` and `CameraTarget` properties of all subplot axes. Since this example function will have completed execution when the user is rotating the subplots, the link object is stored in the first subplot axes application data. See `setappdata` for more information on using application data.

```
hlink = linkprop(h,{'CameraPosition','CameraUpVector'});
key = 'graphics_linkprop';
% Store link object on first subplot axes
setappdata(h(1),key,hlink);
```

4 The following local function contains viewing and lighting commands issued on each axes. It is called with the creation of each subplot (`view`, `axis`, `camlight`).

```
function set_view(ax)
% Set the view and add lighting
view(ax,3); axis(ax,'tight','equal')
camlight left; camlight right
% Make axes invisible and title visible
axis(ax,'off')
set(get(ax,'title'),'Visible','on')
```

Linking an Additional Property

Suppose you want to add the axes `PlotBoxAspectRatio` to the linked properties in the previous example. You can do this by modifying the link object that is stored in the first subplot axes' application data.

1 First click the first subplot axes to make it the current axes (since its handle was saved only within the creating function). Then get the link object's handle from application data (`getappdata`).

```
hlink = getappdata(gca,'graphics_linkprop');
```

2 Use the `addprop` method to add a new property to the link object.

```
addprop(hlink,'PlotBoxAspectRatio')
```

Since `hlink` is a reference to the link object (i.e., not a copy), `addprop` can change the object that is stored in application data.

See Also

`getappdata`, `linkaxes`, `setappdata`

Purpose

Solve linear system of equations

Syntax

```
X = linsolve(A,B)
X = linsolve(A,B,opts)
```

Description

`X = linsolve(A,B)` solves the linear system `A*X = B` using LU factorization with partial pivoting when A is square and QR factorization with column pivoting otherwise. The number of rows of `A` must equal the number of rows of `B`. If `A` is m-by-n and `B` is m-by-k, then `X` is n-by-k. `linsolve` returns a warning if `A` is square and ill conditioned or if it is not square and rank deficient.

`[X, R] = linsolve(A,B)` suppresses these warnings and returns `R`, which is the reciprocal of the condition number of `A` if `A` is square, or the rank of `A` if `A` is not square.

`X = linsolve(A,B,opts)` solves the linear system `A*X = B` or `A'*X = B`, using the solver that is most appropriate given the properties of the matrix `A`, which you specify in `opts`. For example, if `A` is upper triangular, you can set `opts.UT = true` to make `linsolve` use a solver designed for upper triangular matrices. If `A` has the properties in `opts`, `linsolve` is faster than `mldivide`, because `linsolve` does not perform any tests to verify that `A` has the specified properties.

Notes If `A` does not have the properties that you specify in `opts`, `linsolve` returns incorrect results and does not return an error message. If you are not sure whether `A` has the specified properties, use `mldivide` instead.
For small problems, there is no speed benefit in using `linsolve` on triangular matrices as opposed to using the `mldivide` function.

The `TRANSA` field of the `opts` structure specifies the form of the linear system you want to solve:

- If you set `opts.TRANSA = false`, `linsolve(A,B,opts)` solves `A*X = B`.

- If you set `opts.TRANSA = true`, `linsolve(A,B,opts)` solves `A'*X = B`.

The following table lists all the field of `opts` and their corresponding matrix properties. The values of the fields of `opts` must be `logical` and the default value for all fields is `false`.

Field Name	Matrix Property
`LT`	Lower triangular
`UT`	Upper triangular
`UHESS`	Upper Hessenberg
`SYM`	Real symmetric or complex Hermitian
`POSDEF`	Positive definite
`RECT`	General rectangular
`TRANSA`	Conjugate transpose — specifies whether the function solves `A*X = B` or `A'*X = B`

The following table lists all combinations of field values in `opts` that are valid for `linsolve`. A true/false entry indicates that `linsolve` accepts either true or false.

LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
`true`	`false`	`false`	`false`	`false`	`true/false`	`true/false`
`false`	`true`	`false`	`false`	`false`	`true/false`	`true/false`
`false`	`false`	`true`	`false`	`false`	`false`	`true/false`
`false`	`false`	`false`	`true`	`true/false`	`false`	`true/false`
`false`	`false`	`false`	`false`	`false`	`true/false`	`true/false`

Example

The following code solves the system `A'x = b` for an upper triangular matrix `A` using both `mldivide` and `linsolve`.

```
A = triu(rand(5,3)); x = [1 1 1 0 0]'; b = A'*x;
y1 = (A')\b
opts.UT = true; opts.TRANSA = true;
y2 = linsolve(A,b,opts)

y1 =

    1.0000
    1.0000
    1.0000
         0
         0


y2 =

    1.0000
    1.0000
    1.0000
         0
         0
```

Note If you are working with matrices having different properties, it is useful to create an options structure for each type of matrix, such as `opts_sym`. This way you do not need to change the fields whenever you solve a system with a different type of matrix `A`.

See Also

`mldivide`

linspace

Purpose

Generate linearly spaced vectors

Syntax

```
y = linspace(a,b)
y = linspace(a,b,n)
```

Description

The `linspace` function generates linearly spaced vectors. It is similar to the colon operator ":", but gives direct control over the number of points.

`y = linspace(a,b)` generates a row vector `y` of 100 points linearly spaced between and including `a` and `b`.

`y = linspace(a,b,n)` generates a row vector `y` of `n` points linearly spaced between and including `a` and `b`.

See Also

`logspace`

The colon operator `:`

Purpose

Create and open list-selection dialog box

Syntax

```
[Selection,ok] = listdlg('ListString',S)
```

Description

`[Selection,ok] = listdlg('ListString',S)` creates a modal dialog box that enables you to select one or more items from a list. `Selection` is a vector of indices of the selected strings (in single selection mode, its length is 1). `Selection` is `[]` when `ok` is 0. `ok` is 1 if you click the **OK** button, or 0 if you click the **Cancel** button or close the dialog box. Double-clicking on an item or pressing **Return** when multiple items are selected has the same effect as clicking the **OK** button. The dialog box has a **Select all** button (when in multiple selection mode) that enables you to select all list items.

Inputs are in parameter/value pairs:

Parameter	Description
`'ListString'`	Cell array of strings that specify the list box items.
`'SelectionMode'`	String indicating whether one or many items can be selected: `'single'` or `'multiple'` (the default).
`'ListSize'`	List box size in pixels, specified as a two-element vector `[width height]`. Default is `[160 300]`.
`'InitialValue'`	Vector of indices of the list box items that are initially selected. Default is 1, the first item.
`'Name'`	String for the dialog box's title. Default is ''.
`'PromptString'`	String matrix or cell array of strings that appears as text above the list box. Default is {}.
`'OKString'`	String for the OK button. Default is `'OK'`.
`'CancelString'`	String for the Cancel button. Default is 'Cancel'.
'uh'	Uicontrol button height, in pixels. Default is 18.

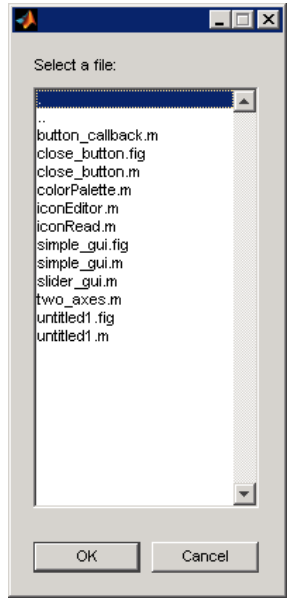
Parameter	Description
'fus'	Frame/uicontrol spacing, in pixels. Default is 8.
'ffs'	Frame/figure spacing, in pixels. Default is 8.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in the MATLAB Figure Properties.

Example

This example displays a dialog box that enables the user to select a file from the current directory. The function returns a vector. Its first element is the index to the selected file; its second element is 0 if no selection is made, or 1 if a selection is made.

```
d = dir;
str = {d.name};
[s,v] = listdlg('PromptString','Select a file:',...
                'SelectionMode','single',...
                'ListString',str)
```



See Also

`dialog`, `errordlg`, `helpdlg`, `inputdlg`, `msgbox`, `questdlg`, `warndlg`

`dir`, `figure`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-104 for related functions

listfonts

Purpose

List available system fonts

Syntax

```
c = listfonts
c = listfonts(h)
```

Description

`c = listfonts` returns sorted list of available system fonts.

`c = listfonts(h)` returns sorted list of available system fonts and includes the `FontName` property of the object with handle `h`.

Examples

Example 1

This example returns a list of available system fonts similar in format to the one shown.

```
list = listfonts

list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
    'ZapfChancery'
    'ZapfDingbats'
    'ZWAdobeF'
```

Example 2

This example returns a list of available system fonts with the value of the`FontName` property, for the object with handle `h`, sorted into the list.

```
 h = uicontrol('Style','text','String','My Font','FontName','MyFont');
 list = listfonts(h)

list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
```

```
        'MyFont'
        ...
        'ZapfChancery'
        'ZapfDingbats'
        'ZWAdobeF'
```

See Also

uisetfont

load

Purpose

Load workspace variables from disk

Syntax

```
load
load filename
load filename X Y Z ...
load filename -regexp expr1 expr2 ...
load -ascii filename
load -mat filename
S = load('arg1', 'arg2', 'arg3', ...)
```

Description

`load` loads all the variables from the MAT-file `matlab.mat`, if it exists, or returns an error if the file doesn't exist.

`load filename` loads all the variables from the file specified by `filename`. `filename` is an unquoted string specifying a file name, and can also include a file extension and a full or `partial path` name. If `filename` has no extension, `load` looks for a file named `filename.mat` and treats it as a binary MAT-file. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

`load filename X Y Z ...` loads just the specified variables `X`, `Y`, `Z`, etc. from the MAT-file. The wildcard `'*'` loads variables that match a pattern (MAT-file only).

`load filename -regexp expr1 expr2 ...` loads those variables that match any of the "Regular Expressions" given by `expr1`, `expr1`, etc.

`load -ascii filename` forces `load` to treat the file as an ASCII file, regardless of file extension. If the file is not numeric text, `load` returns an error. Use `load -ascii` only on files that have been created with the `save -ascii` command.

`load -mat filename` forces `load` to treat the file as a MAT-file, regardless of file extension. If the file is not a MAT-file, `load` returns an error.

`S = load('arg1', 'arg2', 'arg3', ...)` calls `load` using MATLAB *function syntax*, (as opposed to the MATLAB *command syntax* that has been shown thus far). You can use function syntax with any form

of the `load` command shown above, replacing `arg1`, `arg2`, etc. with the arguments shown. For example,

```
S = load('myfile.mat', '-regexp', '^Mon', '^Tue')
```

To specify a command line option, such as `-mat`, with the functional form, specify the option as a string argument, and include the hyphen. For example,

```
load('myfile.dat', '-mat')
```

Function syntax enables you to assign values returned by `load` to an output variable. You can also use function syntax when loading from a file having a name that contains space characters, or a filename that is stored in a variable.

If the file you are loading from is a MAT-file, then output `S` is a structure containing fields that match the variables retrieved. If the file contains ASCII data, then `S` is a double-precision array.

Remarks

For information on any of the following topics related to saving to MAT-files, see in the MATLAB Programming documentation:

- Previewing MAT-file contents
- Loading binary data
- Loading ASCII data

You can also use the Current Directory browser to view the contents of a MAT-file without loading it — see “Viewing and Making Changes to Directories”.

MATLAB saves numeric data in MAT-files in the native byte format. The header of the MAT-file contains a 2-byte Endian Indicator that MATLAB uses to determine the byte format when loading the MAT-file. When MATLAB reads a MAT-file, it determines whether byte-swapping needs to be performed by the state of this indicator.

Examples

Example 1 — Loading From a Binary MAT-file

To see what is in the MAT-file prior to loading it, use `whos -file`:

```
whos -file mydata.mat
  Name           Size         Bytes  Class

  javArray      10x1                 java.lang.Double[][]
  spArray        5x5              84  double array (sparse)
  strArray       2x5             678  cell array
  x              3x2x2            96  double array
  y              4x5            1230  cell array
```

Clear the workspace and load it from MAT-file `mydata.mat`:

```
clear
load mydata

whos
  Name           Size         Bytes  Class

  javArray      10x1                 java.lang.Double[][]
  spArray        5x5              84  double array (sparse)
  strArray       2x5             678  cell array
  x              3x2x2            96  double array
  y              4x5            1230  cell array
```

Example 2 — Loading a List of Variables

You can use a comma-separated list to pass the names of those variables you want to load from a file. This example generates a comma-separated list from a cell array

In this example, the file name is stored in a variable, `saved_file`. You must call `load` using the function syntax of the command if you intend to reference the file name through a variable:

```
saved_file = 'myfile.mat';
saved_file = 'ptarray.mat';
whos('-file', saved_file)
```

```
  Name          Size                Bytes  Class

  AName         1x24                   48  char array
  AVal          1x1                     8  double array
  BName         1x24                   48  char array
  BVal          1x1                     8  double array
  CVal          5x5                    84  double array (sparse)
  DArr          2x5                   678  cell array

filevariables = {'AName', 'BVal', 'DArr'};
load(saved_file, filevariables{:});
```

The second part of this example generates a comma-separated list from the name field of a structure array, and loads the first ten variables from the specified file:

```
saved_file = 'myfile.mat';
vars = whos('-file', saved_file);
load(saved_file, vars(1:10).name);
```

Example 3 — Loading From an ASCII File

Create several 4-column matrices and save them to an ASCII file:

```
a = magic(4);  b = ones(2, 4) * -5.7;  c = [8 6 4 2];
save -ascii mydata.dat
```

Clear the workspace and load it from the file `mydata.dat`. If the filename has an extension other than `.mat`, MATLAB assumes that it is ASCII:

```
clear
load mydata.dat
```

MATLAB loads all data from the ASCII file, merges it into a single matrix, and assigns the matrix to a variable named after the filename:

```
mydata
```

```
mydata =
   16.0000    2.0000    3.0000   13.0000
    5.0000   11.0000   10.0000    8.0000
    9.0000    7.0000    6.0000   12.0000
    4.0000   14.0000   15.0000    1.0000
   -5.7000   -5.7000   -5.7000   -5.7000
   -5.7000   -5.7000   -5.7000   -5.7000
    8.0000    6.0000    4.0000    2.0000
```

Example 4 — Using Regular Expressions

Using regular expressions, load from MAT-file `mydata.mat` those variables with names that begin with `Mon`, `Tue`, or `Wed`:

```
load('mydata', '-regexp', '^Mon|^Tue|^Wed');
```

Here is another way of doing the same thing. In this case, there are three separate expression arguments:

```
load('mydata', '-regexp', '^Mon', '^Tue', '^Wed');
```

See Also

`clear`, `fprintf`, `fscanf`, `partialpath`, `save`, `spconvert`, `who`

Purpose

Initialize control object from file

Syntax

```
h.load('filename')
load(h, 'filename')
```

Description

`h.load('filename')` initializes the COM object associated with the interface represented by the MATLAB COM object `h` from file specified in the string `filename`. The file must have been created previously by serializing an instance of the same control.

`load(h, 'filename')` is an alternate syntax for the same operation.

Note The COM `load` function is only supported for controls at this time.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.save('mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```
h.load('mwsample');
h.get
ans =
     Label: 'Label'
    Radius: 20
```

See Also

`save`, `actxcontrol`, `actxserver`, `release`, `delete`

load (serial)

Purpose

Load serial port objects and variables into MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2...
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2...`	Serial port objects or arrays of serial port objects.
`out`	A structure containing the specified serial port objects.

Description

`load filename` returns all variables from the MAT-file specified by `filename` into the MATLAB workspace.

`load filename obj1 obj2...` returns the serial port objects specified by `obj1 obj2 ...` from the MAT-file `filename` into the MATLAB workspace.

`out = load('filename','obj1','obj2',...)` returns the specified serial port objects from the MAT-file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded serial port objects.

Remarks

Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

Example

Suppose you create the serial port objects `s1` and `s2`, configure a few properties for `s1`, and connect both objects to their instruments:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s1,'Parity','mark','DataBits',7);
fopen(s1);
fopen(s2);
```

Save s1 and s2 to the file MyObject.mat, and then load the objects back into the workspace:

```
save MyObject s1 s2;
load MyObject s1;
load MyObject s2;

get(s1, {'Parity', 'DataBits'})
ans =
    'mark'    [7]
get(s2, {'Parity', 'DataBits'})
ans =
    'none'    [8]
```

See Also

Functions

save

Properties

Status

loadlibrary

Purpose

Load external library into MATLAB

Syntax

```
loadlibrary('shrlib', 'hfile')
loadlibrary('shrlib', @protofile)
loadlibrary('shrlib', ..., 'options')
loadlibrary shrlib hfile options
```

Description

`loadlibrary('shrlib', 'hfile')` loads the functions defined in header file `hfile` and found in shared library `shrlib` into MATLAB.

The `hfile` and `shrlib` file names are case sensitive. The name you use in `loadlibrary` must use the same case as the file on your system.

On Windows systems, `shrlib` refers to the name of a dynamic link library (`.dll`) file. On Linux systems, it refers to the name of a shared object (`.so`) file. On Intel-based Macintosh, it refers to a dynamic shared library (`.dylib`). See "File Extensions for Libraries" on page 2-2018 for more information.

`loadlibrary('shrlib', @protofile)` uses the prototype M-file `protofile` in place of a header file in loading the library `shrlib`. The string `@protofile` specifies a function handle to the prototype M-file. (See the description of "Prototype M-Files" on page 2-2020 below).

Note The MATLAB Generic Shared Library interface does not support library functions that have function pointer inputs.

File Extensions for Libraries

If you do not include a file extension with the `shrlib` argument, `loadlibrary` attempts to find the library with either the appropriate platform MEX-file extension or the appropriate platform library extension (usually `.dll`, `.so`, or `.dylib`). See `mex` for a list of extensions.

If you do not include a file extension with the second argument, and this argument is not a function handle, `loadlibrary` uses `.h` for the extension.

`loadlibrary('shrlib', ..., 'options')` loads the library `shrlib` with one or more of the following *`options`*.

Option	Description

`addheader` `hfileN`	Loads the functions defined in the additional header file, `hfileN`. Note that each file specified by `addheader` must be referenced by a corresponding `#include` statement in the base header file.
	
	Specify the string `hfileN` as a filename without a file extension. MATLAB does not verify the existence of the header files and ignores any that are not needed.
	
	You can specify as many additional header files as you need using the syntax
	
	`loadlibrary shrlib hfile ...`
	`  addheader hfile1 ...`
	`  addheader hfile2 ...  % and so on`

`alias` `name`	Associates the specified alias `name` with the library. All subsequent calls to MATLAB functions that reference this library must use this alias until the library is unloaded.
`includepath` `path`	Specifies an additional `path` in which to look for included header files.
`mfilename` `mfile`	Generates a prototype M-file `mfile` in the current directory. You can use this file in place of a header file when loading the library. (See the description of "Prototype M-Files" on page 2-2020 below).

Only the **`alias`** option is available when loading using a prototype M-file.

If you have more than one library file of the same name, load the first using the library filename, and load the additional libraries using the **`alias`** option.

`loadlibrary shrlib hfile` *`options`* is the command format for this function.

Remarks

How to Use the addheader Option

The `addheader` option enables you to add functions for MATLAB to load from those listed in header files included in the base header file (with a `#include` statement). For example, if your library header file contains the statement:

```
#include header2.h
```

then to load the functions in `header2.h`, you need to use `addheader` in the call to `loadlibrary`:

```
loadlibrary libname libname.h addheader header2.h
```

You can use the `addheader` option with a header file that lists function prototypes for only the functions that are needed by your library, and thereby avoid loading functions that you do not define in your library. To do this, you might need to create a header file that contains a subset of the functions listed in large header file.

addheader Syntax

When using `addheader` to specify which functions to load, ensure that there are `#include` statements in the base header file for each additional header file in the `loadlibrary` call. For example, to use the following statement:

```
loadlibrary mylib mylib.h addheader header2.h
```

the file `mylib.h` must contain this statement:

```
#include header2.h
```

Prototype M-Files

When you use the **`mfilename`** option with `loadlibrary`, MATLAB generates an M-file called a prototype file. This file can then be used on subsequent calls to `loadlibrary` in place of a header file.

Like a header file, the prototype file supplies MATLAB with function prototype information for the library. You can make changes to the prototypes by editing this file and reloading the library.

Here are some reasons for using a prototype file, along with the changes you would need to make to the file:

- You want to make temporary changes to signatures of the library functions.

  Edit the prototype file, changing the `fcns.LHS` or `fcns.RHS` field for that function. This changes the types of arguments on the left hand side or right hand side, respectively.

- You want to rename some of the library functions.

  Edit the prototype file, defining the `fcns.alias` field for that function.

- You expect to use only a small percentage of the functions in the library you are loading.

  Edit the prototype file, commenting out the unused functions. This reduces the amount of memory required for the library.

- You need to specify a number of include files when loading a particular library.

  Specify the full list of include files (plus the **`mfilename`** option) in the first call to `loadlibrary`. This puts all the information from the include files into the prototype file. After that, specify just the prototype file.

Examples

Example 1

Use `loadlibrary` to load the MATLAB sample shared library, `shrlibsample`:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h
```

Example 2

Load sample library `shrlibsample`, giving it an alias name of `lib`. Once you have set an alias, you need to use this name in all further interactions with the library for this session:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h alias lib

libfunctionsview lib

str = 'This was a Mixed Case string';
calllib('lib', 'stringToUpper', str)
ans =
   THIS WAS A MIXED CASE STRING
unloadlibrary lib
```

Example 3

Load the library, specifying an additional path in which to search for included header files:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary('shrlibsample','shrlibsample.h','includepath', ...
           fullfile(matlabroot , 'extern', 'include'));
```

Example 4

Load the `libmx` library and generate a prototype M-file containing the prototypes defined in header file `matrix.h`:

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile, 'mfilename', 'mxproto')

dir mxproto.m
   mxproto.m
```

Edit the generated file `mxproto.m` and locate the function `'mxGetNumberOfDimensions'`. Give it an alias of `'mxGetDims'` by adding this text to the line before `fcnNum` is incremented:

```
fcns.alias{fcnNum}='mxGetDims';
```

Here is the new function prototype. The change is shown in bold:

```
fcns.name{fcnNum}='mxGetNumberOfDimensions';
fcns.calltype{fcnNum}='cdecl';
fcns.LHS{fcnNum}='int32';
fcns.RHS{fcnNum}={'MATLAB array'};
fcns.alias{fcnNum}='mxGetDims'; % Alias defined
fcnNum=fcnNum+1; % Increment fcnNum
```

Unload the library and then reload it using the prototype M-file.

```
unloadlibrary libmx

loadlibrary('libmx', @mxproto)
```

Now call `mxGetNumberOfDimensions` using the alias function name:

```
y = rand(4, 7, 2);

calllib('libmx', 'mxGetDims', y)
ans =
     3

unloadlibrary libmx
```

See Also

`libisloaded`, `unloadlibrary`, `libfunctions`, `libfunctionsview`, `libpointer`, `libstruct`, `calllib`

loadobj

Purpose

User-defined extension of `load` function for user objects

Syntax

```
b = loadobj(a)
```

Description

`b = loadobj(a)` extends the `load` function for user objects. When an object is loaded from a MAT-file, the `load` function calls the `loadobj` method for the object's class if it is defined. The `loadobj` method must have the calling syntax shown. The input argument `a` is the object as loaded from the MAT-file or a structure created by `load` if the object cannot be resolved, and the output argument `b` is the object that the `load` function loads into the workspace.

The following steps describe how an object is loaded from a MAT-file into the workspace:

1. The `load` function detects the object `a` in the MAT-file.
2. The `load` function looks in the current workspace for an object of the same class as the object `a`. If there isn't an object of the same class in the workspace, `load` calls the default constructor, registering an object of that class in the workspace. The default constructor is the constructor function called with no input arguments.
3. The `load` function checks to see if the structure of the object `a` matches the structure of the object registered in the workspace. If the objects match, `a` is loaded. If the objects don't match, `load` converts `a` to a structure variable and issues a warning if no `loadobj` method exists.
4. The `load` function calls the `loadobj` method for the object's class if it is defined. `load` passes the object `a` to the `loadobj` method as an input argument. Note that the format of the object `a` is dependent on the results of step 3 (object or structure). The output argument of `loadobj`, `b`, is loaded into the workspace in place of the object `a` and MATLAB issues no warning because the class' `loadobj` method is assumed to have converted the structure to a proper object conforming to the current class definition.

See “The loadobj Method” for an example of a `loadobj` method.

Remarks

`loadobj` can be overloaded only for user objects. `load` does not call `loadobj` for built-in data types (such as `double`).

`loadobj` is invoked separately for each object in the MAT-file. The `load` function recursively descends cell arrays and structures, applying the `loadobj` method to each object encountered.

A child object inherits the `loadobj` method of its parent class. First the child object's `loadobj` method is called, then the parents `loadobj` is called. Note that this behavior is different from that of the `saveobj` method, which is not inherited from its parent.

See Also

`load`, `save`, `saveobj`

log

Purpose

Natural logarithm

Syntax

```
Y = log(X)
```

Description

The `log` function operates element-wise on arrays. Its domain includes complex and negative numbers, which may lead to unexpected results if used unintentionally.

`Y = log(X)` returns the natural logarithm of the elements of `X`. For complex or negative z, where $z = x + y*i$, the complex logarithm is returned.

```
log(z) = log(abs(z)) + i*atan2(y,x)
```

Examples

The statement `abs(log(-1))` is a clever way to generate π.

```
ans =

    3.1416
```

See Also

`exp`, `log10`, `log2`, `logm`, `reallog`

Purpose

Common (base 10) logarithm

Syntax

```
Y = log10(X)
```

Description

The `log10` function operates element-by-element on arrays. Its domain includes complex numbers, which may lead to unexpected results if used unintentionally.

`Y = log10(X)` returns the base 10 logarithm of the elements of `X`.

Examples

```
log10(realmax) is 308.2547
```

and

```
log10(eps) is -15.6536
```

See Also

`exp`, `log`, `log2`, `logm`

log1p

Purpose

Compute `log(1+x)` accurately for small values of `x`

Syntax

```
y = log1p(x)
```

Description

`y = log1p(x)` computes `log(1+x)`, compensating for the roundoff in `1+x`. `log1p(x)` is more accurate than `log(1+x)` for small values of `x`. For small `x`, `log1p(x)` is approximately `x`, whereas `log(1+x)` can be zero.

See Also

`log`, `expm1`

Purpose

Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

Syntax

```
Y = log2(X)
[F,E] = log2(X)
```

Description

`Y = log2(X)` computes the base 2 logarithm of the elements of `X`.

`[F,E] = log2(X)` returns arrays `F` and `E`. Argument `F` is an array of real values, usually in the range `0.5 <= abs(F) < 1`. For real `X`, `F` satisfies the equation: `X = F.*2.^E`. Argument `E` is an array of integers that, for real `X`, satisfy the equation: `X = F.*2.^E`.

Remarks

This function corresponds to the ANSI C function `frexp()` and the IEEE floating-point standard function `logb()`. Any zeros in `X` produce `F = 0` and `E = 0`.

Examples

For IEEE arithmetic, the statement `[F,E] = log2(X)` yields the values:

X	F	E
1	1/2	1
pi	pi/4	2
-3	-3/4	2
eps	1/2	-51
realmax	1-eps/2	1024
realmin	1/2	-1021

See Also

`log`, `pow2`

logical

Purpose

Convert numeric values to logical

Syntax

```
K = logical(A)
```

Description

`K = logical(A)` returns an array that can be used for logical indexing or logical tests.

`A(B)`, where `B` is a logical array that is the same size as `A`, returns the values of `A` at the indices where the real part of `B` is nonzero.

`A(B)`, where `B` is a logical array that is smaller than `A`, returns the values of column vector `A(:)` at the indices where the real part of column vector `B(:)` is nonzero.

Remarks

Most arithmetic operations remove the logicalness from an array. For example, adding zero to a logical array removes its logical characteristic. `A = +A` is the easiest way to convert a logical array, `A`, to a numeric double array.

Logical arrays are also created by the relational operators (==,<,>,~, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

Examples

Given `A = [1 2 3; 4 5 6; 7 8 9]`, the statement `B = logical(eye(3))` returns a logical array

```
B =
   1   0   0
   0   1   0
   0   0   1
```

which can be used in logical indexing that returns `A`'s diagonal elements:

```
A(B)

ans =
     1
     5
     9
```

However, attempting to index into `A` using the *numeric* array `eye(3)` results in:

```
A(eye(3))
??? Subscript indices must either be real positive integers or
logicals.
```

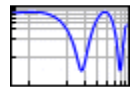
See Also

`islogical`, logical operators (elementwise and short-circuit),

loglog

Purpose

Log-log scale plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
hlines = loglog('v6',...)
```

Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`loglog(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

`loglog(...,'PropertyName',PropertyValue,...)` sets property values for all lineseries graphics objects created by `loglog`. See the `line` reference page for more information.

`h = loglog(...)` returns a column vector of handles to lineseries graphics objects, one handle per line.

Backward-Compatible Version

`hlines = loglog('v6',...)` returns the handles to line objects instead of lineseries objects.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

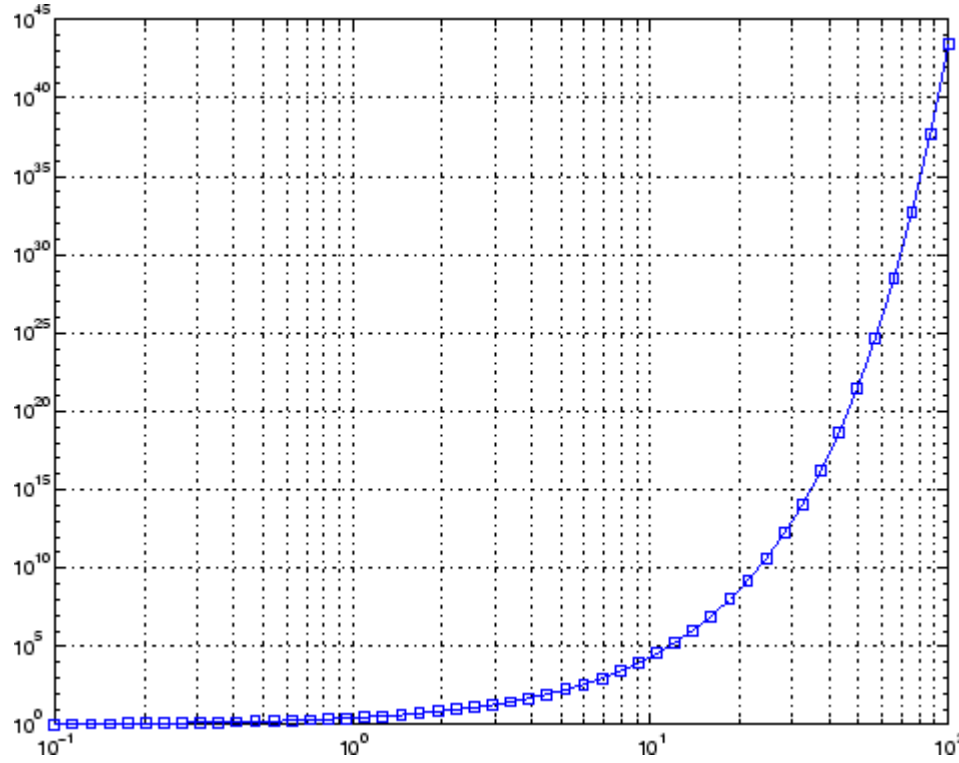
If you do not specify a color when plotting more than one line, `loglog` automatically cycles through the colors and line styles in the order specified by the current axes.

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode will remain as it is and the new data will plot as linear.

Examples

Create a simple `loglog` plot with square markers.

```
x = logspace(-1,2);
loglog(x,exp(x),'-s')
grid on
```



See Also

`LineSpec`, `plot`, `semilogx`, `semilogy`

"Basic Plots and Graphs" on page 1-86 for related functions

Purpose

Matrix logarithm

Syntax

```
L = logm(A)
[L, exitflag] = logm(A)
```

Description

`L = logm(A)` is the principal matrix logarithm of A, the inverse of `expm(A)`. `L` is the unique logarithm for which every eigenvalue has imaginary part lying strictly between -π and π. If `A` is singular or has any eigenvalues on the negative real axis, the principal logarithm is undefined. In this case, `logm` computes a non-principal logarithm and returns a warning message.

`[L, exitflag] = logm(A)` returns a scalar `exitflag` that describes the exit condition of `logm`:

- If `exitflag = 0`, the algorithm was successfully completed.
- If `exitflag = 1`, one or more Taylor series evaluations did not converge. However, the computed value of `L` might still be accurate.

The input `A` can have class `double` or `single`.

Remarks

If `A` is real symmetric or complex Hermitian, then so is `logm(A)`.

Some matrices, like `A = [0 1; 0 0]`, do not have any logarithms, real or complex, so `logm` cannot be expected to produce one.

Limitations

For most matrices:

```
logm(expm(A)) = A = expm(logm(A))
```

These identities may fail for some `A`. For example, if the computed eigenvalues of `A` include an exact zero, then `logm(A)` generates infinity. Or, if the elements of `A` are too large, `expm(A)` may overflow.

Examples

Suppose `A` is the 3-by-3 matrix

```
     1     1     0
```

```
          0         0         2
          0         0        -1
```

and `Y = expm(A)` is

```
  Y =
      2.7183    1.7183    1.0862
           0    1.0000    1.2642
           0         0    0.3679
```

Then `A = logm(Y)` produces the original matrix `A`.

```
  Y =
      1.0000    1.0000    0.0000
           0         0    2.0000
           0         0   -1.0000
```

But `log(A)` involves taking the logarithm of zero, and so produces

```
  ans =
      1.0000    0.5413    0.0826
        -Inf         0    0.2345
        -Inf      -Inf   -1.0000
```

Algorithm

The algorithm `logm` uses is described in [1].

See Also

`expm`, `funm`, `sqrtm`

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Cheng, S. H., N. J. Higham, C. S. Kenney, and A. J. Laub, "Approximating the logarithm of a matrix to specified accuracy," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1112-1125, 2001.

[3] Higham, N. J., "Evaluating Pade approximants of the matrix logarithm," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1126-1135, 2001.

[4] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[5] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review* 20, 1978, pp. 801-836.

logspace

Purpose

Generate logarithmically spaced vectors

Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
y = logspace(a,pi)
```

Description

The `logspace` function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of `linspace` and the ":" or colon operator.

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades `10^a` and `10^b`.

`y = logspace(a,b,n)` generates `n` points between decades `10^a` and `10^b`.

`y = logspace(a,pi)` generates the points between `10^a` and `pi`, which is useful for digital signal processing where frequencies over this interval go around the unit circle.

Remarks

All the arguments to `logspace` must be scalars.

See Also

`linspace`

The colon operator `:`

Purpose

Search for keyword in all help entries

Syntax

```
lookfor topic
lookfor topic -all
```

Description

`lookfor topic` searches for the string `topic` in the first comment line (the H1 line) of the help text in all M-files found on the MATLAB search path. For all files in which a match occurs, `lookfor` displays the H1 line.

`lookfor topic -all` searches the entire first comment block of an M-file looking for `topic`.

Examples

For example

```
lookfor inverse
```

finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with

```
which inverse
```

or

```
what inverse
```

These functions run more quickly, but probably fail to find anything because MATLAB does not have a function `inverse`.

In summary, `what` lists the functions in a given directory, `which` finds the directory containing a given function or file, and `lookfor` finds all functions in all directories that might have something to do with a given keyword.

Even more extensive than the `lookfor` function is the find feature in the Current Directory browser. It looks for all occurrences of a specified word in all the M-files in the current directory. For instructions, see the topic “Finding Files and Content Within Files” in the MATLAB Desktop Tools and Development Environment documentation.

See Also

dir, doc, filebrowser, findstr, help, helpdesk, helpwin, regexp, what, which, who

Purpose

Convert string to lowercase

Syntax

```
t = lower('str')
B = lower(A)
```

Description

`t = lower('str')` returns the string formed by converting any uppercase characters in `str` to the corresponding lowercase characters and leaving all other characters unchanged.

`B = lower(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `lower` to each string within `A`.

Examples

`lower('MathWorks')` is `mathworks`.

Remarks

Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`upper`

ls

Purpose

Directory contents on UNIX system

Syntax

```
ls
```

Description

`ls` displays the results of the `ls` command on UNIX. On UNIX, `ls` returns a character row vector of filenames separated by tab and space characters. On Windows, `ls` returns an `m`-by-`n` character array of filenames, where `m` is the number of filenames and `n` is the number of characters in the longest filename found. Filenames shorter than `n` characters are padded with space characters.

On UNIX, you can pass any flags to `ls` that your operating system supports.

See Also

`dir`

Purpose

Least-squares solution in presence of known covariance

Syntax

```
x = lscov(A,b)
x = lscov(A,b,w)
x = lscov(A,b,V)
x = lscov(A,b,V,alg)
[x,stdx] = lscov(...)
[x,stdx,mse] = lscov(...)
[x,stdx,mse,S] = lscov(...)
```

Description

`x = lscov(A,b)` returns the ordinary least squares solution to the linear system of equations `A*x = b`, i.e., `x` is the n-by-1 vector that minimizes the sum of squared errors `(b - A*x)'*(b - A*x)`, where `A` is m-by-n, and `b` is m-by-1. `b` can also be an m-by-k matrix, and `lscov` returns one solution for each column of `b`. When `rank(A) < n`, `lscov` sets the maximum possible number of elements of `x` to zero to obtain a "basic solution".

`x = lscov(A,b,w)`, where `w` is a vector length m of real positive weights, returns the weighted least squares solution to the linear system `A*x = b`, that is, `x` minimizes `(b - A*x)'*diag(w)*(b - A*x)`. `w` typically contains either counts or inverse variances.

`x = lscov(A,b,V)`, where `V` is an m-by-m real symmetric positive definite matrix, returns the generalized least squares solution to the linear system `A*x = b` with covariance matrix proportional to `V`, that is, `x` minimizes `(b - A*x)'*inv(V)*(b - A*x)`.

More generally, `V` can be positive semidefinite, and `lscov` returns `x` that minimizes `e'*e`, subject to `A*x + T*e = b`, where the minimization is over `x` and `e`, and `T*T' = V`. When `V` is semidefinite, this problem has a solution only if `b` is consistent with `A` and `V` (that is, `b` is in the column space of `[A T]`), otherwise `lscov` returns an error.

By default, `lscov` computes the Cholesky decomposition of `V` and, in effect, inverts that factor to transform the problem into ordinary least squares. However, if `lscov` determines that `V` is semidefinite, it uses an orthogonal decomposition algorithm that avoids inverting `V`.

`x = lscov(A,b,V,alg)` specifies the algorithm used to compute `x` when `V` is a matrix. `alg` can have the following values:

- `'chol'` uses the Cholesky decomposition of `V`.
- `'orth'` uses orthogonal decompositions, and is more appropriate when `V` is ill-conditioned or singular, but is computationally more expensive.

`[x,stdx] = lscov(...)` returns the estimated standard errors of `x`. When `A` is rank deficient, `stdx` contains zeros in the elements corresponding to the necessarily zero elements of `x`.

`[x,stdx,mse] = lscov(...)` returns the mean squared error.

`[x,stdx,mse,S] = lscov(...)` returns the estimated covariance matrix of `x`. When `A` is rank deficient, `S` contains zeros in the rows and columns corresponding to the necessarily zero elements of `x`. `lscov` cannot return `S` if it is called with multiple right-hand sides, that is, if `size(B,2) > 1`.

The standard formulas for these quantities, when `A` and `V` are full rank, are

- `x = inv(A'*inv(V)*A)*A'*inv(V)*B`
- `mse = B'*(inv(V) - inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)`
- `S = inv(A'*inv(V)*A)*mse`
- `stdx = sqrt(diag(S))`

However, `lscov` uses methods that are faster and more stable, and are applicable to rank deficient cases.

`lscov` assumes that the covariance matrix of `B` is known only up to a scale factor. `mse` is an estimate of that unknown scale factor, and `lscov` scales the outputs `S` and `stdx` appropriately. However, if `V` is known to be exactly the covariance matrix of `B`, then that scaling is unnecessary.

To get the appropriate estimates in this case, you should rescale `S` and `stdx` by `1/mse` and `sqrt(1/mse)`, respectively.

Algorithm

The vector `x` minimizes the quantity `(A*x-b)'*inv(V)*(A*x-b)`. The classical linear algebra solution to this problem is

```
x = inv(A'*inv(V)*A)*A'*inv(V)*b
```

but the `lscov` function instead computes the QR decomposition of `A` and then modifies `Q` by `V`.

Examples

Example 1 — Computing Ordinary Least Squares

The MATLAB backslash operator (`\`) enables you to perform linear regression by computing ordinary least-squares (OLS) estimates of the regression coefficients. You can also use `lscov` to compute the same OLS estimates. By using `lscov`, you can also compute estimates of the standard errors for those coefficients, and an estimate of the standard deviation of the regression error term:

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
X = [ones(size(x1)) x1 x2];
y = [.17 .26 .28 .23 .27 .34]';

a = X\y
a =
    0.1203
    0.3284
   -0.1312

[b,se_b,mse] = lscov(X,y)
b =
    0.1203
    0.3284
   -0.1312
se_b =
    0.0643
```

```
    0.2267
    0.1488
mse =
    0.0015
```

Example 2 — Computing Weighted Least Squares

Use `lscov` to compute a weighted least-squares (WLS) fit by providing a vector of relative observation weights. For example, you might want to downweight the influence of an unreliable observation on the fit:

```
w = [1 1 1 1 1 .1]';

[bw,sew_b,msew] = lscov(X,y,w)
bw =
    0.1046
    0.4614
   -0.2621
sew_b =
    0.0309
    0.1152
    0.0814
msew =
  3.4741e-004
```

Example 3 — Computing General Least Squares

Use `lscov` to compute a general least-squares (GLS) fit by providing an observation covariance matrix. For example, your data may not be independent:

```
V = .2*ones(length(x1)) + .8*diag(ones(size(x1)));

[bg,sew_b,mseg] = lscov(X,y,V)
bg =
    0.1203
    0.3284
   -0.1312
sew_b =
```

```
    0.0672
    0.2267
    0.1488
mseg =
    0.0019
```

Example 4 — Estimating the Coefficient Covariance Matrix

Compute an estimate of the coefficient covariance matrix for either OLS, WLS, or GLS fits. The coefficient standard errors are equal to the square roots of the values on the diagonal of this covariance matrix:

```
[b,se_b,mse,S] = lscov(X,y);

S
S =
    0.0041   -0.0130    0.0075
   -0.0130    0.0514   -0.0328
    0.0075   -0.0328    0.0221

[se_b sqrt(diag(S))]
ans =
    0.0643    0.0643
    0.2267    0.2267
    0.1488    0.1488
```

See Also

`lsqnonneg`, `qr`

The arithmetic operator `\`

Reference

[1] Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

lsqnonneg

Purpose

Solve nonnegative least-squares constraints problem

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,x0)
x = lsqnonneg(C,d,x0,options)
[x,resnorm] = lsqnonneg(...)
[x,resnorm,residual] = lsqnonneg(...)
[x,resnorm,residual,exitflag] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)
```

Description

`x = lsqnonneg(C,d)` returns the vector `x` that minimizes `norm(C*x-d)` subject to `x >= 0`. `C` and `d` must be real.

`x = lsqnonneg(C,d,x0)` uses `x0` as the starting point if all `x0 >= 0`; otherwise, the default is used. The default start point is the origin (the default is used when `x0==[ ]` or when only two input arguments are provided).

`x = lsqnonneg(C,d,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `lsqnonneg` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`TolX`	Termination tolerance on `x`.
`OutputFcn`	User-defined function that is called at each iteration. See “Output Function” in the Optimization Toolbox for more information.
`PlotFcns`	User-defined plot function that is called at each iteration. See “Plot Functions” in the Optimization Toolbox for more information.

`[x,resnorm] = lsqnonneg(...)` returns the value of the squared 2-norm of the residual: `norm(C*x-d)^2`.

`[x,resnorm,residual] = lsqnonneg(...)` returns the residual, `d-C*x`.

`[x,resnorm,residual,exitflag] = lsqnonneg(...)` returns a value `exitflag` that describes the exit condition of `lsqnonneg`:

>0	Indicates that the function converged to a solution `x`.
0	Indicates that the iteration count was exceeded. Increasing the tolerance (`TolX` parameter in `options`) may lead to a solution.

`[x,resnorm,residual,exitflag,output] = lsqnonneg(...)` returns a structure `output` that contains information about the operation:

`output.algorithm`	The algorithm used
`output.iterations`	The number of iterations taken

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)` returns the dual vector (Lagrange multipliers) `lambda`, where `lambda(i)<=0` when `x(i)` is (approximately) `0`, and `lambda(i)` is (approximately) `0` when `x(i)>0`.

Examples

Compare the unconstrained least squares solution to the `lsqnonneg` solution for a 4-by-2 problem:

```
C = [
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170];
d = [
    0.8587
    0.1781
    0.0747
```

```
    0.8405];
[C\d lsqnonneg(C,d)] =
    -2.5627         0
     3.1108    0.6929
[norm(C*(C\d)-d) norm(C*lsqnonneg(C,d)-d)] =
     0.6674 0.9118
```

The solution from `lsqnonneg` does not fit as well (has a larger residual), as the least squares solution. However, the nonnegative least squares solution has no negative components.

Algorithm

`lsqnonneg` uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector `lambda`. It then selects the basis vector corresponding to the maximum value in `lambda` in order to swap out of the basis in exchange for another possible candidate. This continues until `lambda <= 0`.

See Also

The arithmetic operator `\`, `optimset`

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974, Chapter 23, p. 161.

Purpose

LSQR method

Syntax

```
x = lsqr(A,b)
lsqr(A,b,tol)
lsqr(A,b,tol,maxit)
lsqr(A,b,tol,maxit,M)
lsqr(A,b,tol,maxit,M1,M2)
lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,
   x0)
```

Description

`x = lsqr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x` if `A` is consistent, otherwise it attempts to solve the least squares solution `x` that minimizes `norm(b-A*x)`. The `m`-by-`n` coefficient matrix `A` need not be square but it should be large and sparse. The column vector `b` must have length `m`. `A` can be a function handle `afun` such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parameterizing Functions Called by Function Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `lsqr` converges, a message to that effect is displayed. If `lsqr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`lsqr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `lsqr` uses the default, `1e-6`.

`lsqr(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `lsqr` uses the default, `min([m,n,20])`.

`lsqr(A,b,tol,maxit,M)` and `lsqr(A,b,tol,maxit,M1,M2)` use n-by-n preconditioner `M` or `M = M1*M2` and effectively solve the system `A*inv(M)*y = b` for `y`, where `y = M*x`. If `M` is `[]` then `lsqr` applies no preconditioner. `M` can be a function `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`lsqr(A,b,tol,maxit,M1,M2,x0)` specifies the n-by-1 initial guess. If `x0` is `[]`, then `lsqr` uses the default, an all zero vector.

`[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a convergence flag.

Flag	Convergence
0	`lsqr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`lsqr` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`lsqr` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `lsqr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if you specify the `flag` output.

`[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns an estimate of the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norm estimates at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of estimates of the scaled normal equations residual at each iteration: `norm((A*inv(M))'*(B-A*X))/norm(A*inv(M),'fro')`. Note that the estimate of `norm(A*inv(M),'fro')` changes, and hopefully improves, at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = lsqr(A,b,tol,maxit,M1,M2);
```

displays the following message:

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_lsqr` that

- Calls `lsqr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_lsqr` are available to `afun`.

The following shows the code for `run_lsqr`:

```
function x1 = run_lsqr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = lsqr(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
       elseif strcmp(transp_flag,'notransp') % y = A*x
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - x(2:n);
       end
    end
end
```

When you enter

```
x1=run_lsqr;
```

MATLAB displays the message

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

See Also

`bicg`, `bicgstab`, `cgs`, `gmres`, `minres`, `norm`, `pcg`, `qmr`, `symmlq`, `function_handle (@)`

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations And Sparse Least Squares," *ACM Trans. Math. Soft.*, Vol.8, 1982, pp. 43-71.

Purpose

Test for less than

Syntax

```
A < B
lt(A, B)
```

Description

`A < B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is less than `B`, or set to logical 0 (`false`) where `A` is greater than or equal to `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then MATLAB returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`lt(A, B)` is called for the syntax `A < B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are less than the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A < B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     0     0
     1     0     0     1     0     1
```

```
     0     1     0     0     1     1
     1     0     0     0     1     0
```

See Also

`gt`, `le`, `ge`, `ne`, `eq`, “Relational Operators” in the MATLAB Programming documentation

Purpose

LU matrix factorization

Syntax

```
Y = lu(A)
[L,U] = lu(A)
[L,U,P] = lu(A)
[L,U,P,Q] = lu(A)
[L,U,P,Q,R] = lu(A)
[...] = lu(A,'vector')
[...] = lu(A,thresh)
[...] = lu(A,thresh,'vector')
```

Description

The `lu` function expresses a matrix `A` as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the *LU*, or sometimes the *LR*, factorization. `A` can be rectangular. For a full matrix `A`, `lu` uses the Linear Algebra Package (LAPACK) routines described in “Algorithm” on page 2-2064.

`Y = lu(A)` returns matrix `Y` that, for sparse `A`, contains the strictly lower triangular `L`, i.e., without its unit diagonal, and the upper triangular `U` as submatrices. That is, if `[L,U,P] = lu(A)`, then `Y = U+L-eye(size(A))`. For nonsparse `A`, `Y` is the output from the LAPACK `dgetrf` or `zgetrf` routine. The permutation matrix `P` is not returned.

`[L,U] = lu(A)` returns an upper triangular matrix in `U` and a permuted lower triangular matrix in `L` such that `A = L*U`. Return value `L` is a product of lower triangular and permutation matrices.

`[L,U,P] = lu(A)` returns an upper triangular matrix in `U`, a lower triangular matrix `L` with a unit diagonal, and a permutation matrix `P`, such that `L*U = P*A`. The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q] = lu(A)` for sparse nonempty `A`, returns a unit lower triangular matrix `L`, an upper triangular matrix `U`, a row permutation matrix `P`, and a column reordering matrix `Q`, so that `P*A*Q = L*U`. This syntax uses UMFPACK and is significantly more time and memory efficient than the other syntaxes, even when used with `colamd`. If `A`

is empty or not sparse, `lu` displays an error message. The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q,R] = lu(A)` returns unit lower triangular matrix `L`, upper triangular matrix `U`, permutation matrices `P` and `Q`, and a diagonal scaling matrix `R` so that `P*(R\A)*Q = L*U` for sparse non-empty `A`. This uses UMFPACK as well. Typically, but not always, the row-scaling leads to a sparser and more stable factorization. Note that this factorization is the same as that used by sparse `mldivide` when UMFPACK is used. The statement `lu(A,'matrix')` returns identical output values.

`[...] = lu(A,'vector')` returns the permutation information in two row vectors `p` and `q`. You can specify from 1 to 5 outputs. Output `p` is defined as `A(p,:)=L*U`, output `q` is defined as `A(p,q)=L*U`, and output `R` is defined as `R(:,p)\A(:,q)=L*U`.

`[...] = lu(A,thresh)` controls pivoting in UMFPACK. This syntax applies to sparse matrices only. The `thresh` input is a one- or two-element vector of type `single` or `double` that defaults to [0.1, 0.001]. If `A` is a square matrix with a mostly symmetric structure and mostly nonzero diagonal, UMFPACK uses a symmetric pivoting strategy. For this strategy, the diagonal where

```
A(i,j) >= thresh(2) * max(abs(A(j:m,j)))
```

is selected. If the diagonal entry fails this test, a pivot entry below the diagonal is selected, using `thresh(1)`. In this case, `L` has entries with absolute value `1/min(thresh)` or less.

If A is not as described above, UMFPACK uses an asymmetric strategy. In this case, the sparsest row `i` where

```
A(i,j) >= thresh(1) * max(abs(A(j:m,j)))
```

is selected. A value of 1.0 results in conventional partial pivoting. Entries in `L` have an absolute value of `1/thresh(1)` or less. The second element of the `thresh` input vector is not used when UMFPACK uses an asymmetric strategy.

Smaller values of `thresh(1)` and `thresh(2)` tend to lead to sparser LU factors, but the solution can become inaccurate. Larger values can lead to a more accurate solution (but not always), and usually an increase in the total work and memory usage. The statement `lu(A,thresh,'matrix')` returns identical output values.

`[...] = lu(A,thresh,'vector')` controls the pivoting strategy and also returns the permutation information in row vectors, as described above. The `thresh` input must precede `'vector'` in the input argument list.

Note In rare instances, incorrect factorization results in `P*A*Q` ≠ `L*U`. Increase `thresh`, to a maximum of `1.0` (regular partial pivoting), and try again.

Remarks

Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with `inv` and the determinant with `det`. It is also the basis for the linear equation solution or matrix division obtained with `\` and `/`.

Arguments

`A`	Rectangular matrix to be factored.
`thresh`	Pivot threshold for sparse matrices. Valid values are in the interval `[0,1]`. If you specify the fourth output `Q`, the default is `0.1`. Otherwise, the default is `1.0`.
`L`	Factor of `A`. Depending on the form of the function, `L` is either a unit lower triangular matrix, or else the product of a unit lower triangular matrix with `P'`.
`U`	Upper triangular matrix that is a factor of `A`.
`P`	Row permutation matrix satisfying the equation `L*U = P*A`, or `L*U = P*A*Q`. Used for numerical stability.

`Q`	Column permutation matrix satisfying the equation `P*A*Q = L*U`. Used to reduce fill-in in the sparse case.
`R`	Row-scaling matrix

Examples

Example 1

Start with

```
A = [ 1    2    3
      4    5    6
      7    8    0 ];
```

To see the LU factorization, call `lu` with two output arguments.

```
[L1,U] = lu(A)

L1 =
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
    1.0000         0         0

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000
```

Notice that `L1` is a permutation of a lower triangular matrix: if you switch rows 2 and 3, and then switch rows 1 and 2, the resulting matrix is lower triangular and has `1`s on the diagonal. Notice also that `U` is upper triangular. To check that the factorization does its job, compute the product

```
L1*U
```

which returns the original `A`. The inverse of the example matrix, `X = inv(A)`, is actually computed from the inverses of the triangular factors

```
X = inv(U)*inv(L1)
```

Using three arguments on the left side to get the permutation matrix as well,

```
[L2,U,P] = lu(A)
```

returns a truly lower triangular `L2`, the same value of `U`, and the permutation matrix `P`.

```
L2 =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000

P =
    0    0    1
    1    0    0
    0    1    0
```

Note that `L2 = P*L1`.

```
P*L1

ans =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
```

To verify that `L2*U` is a permuted version of `A`, compute `L2*U` and subtract it from `P*A`:

```
P*A - L2*U

ans =
     0     0     0
     0     0     0
     0     0     0
```

In this case, `inv(U)*inv(L)` results in the permutation of `inv(A)` given by `inv(P)*inv(A)`.

The determinant of the example matrix is

```
d = det(A)

d =    27
```

It is computed from the determinants of the triangular factors

```
d = det(L)*det(U)
```

The solution to $Ax = b$ is obtained with matrix division

```
x = A\b
```

The solution is actually computed by solving two triangular systems

```
y = L\b
x = U\y
```

Example 2

The 1-norm of their difference is within roundoff error, indicating that `L*U = P*B*Q`.

Generate a 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster-Fuller geodesic dome.

```
B = bucky;
```

Use the sparse matrix syntax with four outputs to get the row and column permutation matrices.

```
[L,U,P,Q] = lu(B);
```

Apply the permutation matrices to `B`, and subtract the product of the lower and upper triangular matrices.

```
Z = P*B*Q - L*U;
norm(Z,1)

ans =
  7.9936e-015
```

Example 3

This example illustrates the benefits of using the `'vector'` option. Note how much memory is saved by using the `lu(F,'vector')` syntax.

```
rand('state',0);
F = rand(1000,1000);
g = sum(F,2);
[L,U,P] = lu(F);
[L,U,p] = lu(F,'vector');
whos P p

Name          Size            Bytes  Class      Attributes
P          1000x1000       8000000  double
p             1x1000          8000  double
```

The following two statements are equivalent. The first typically requires less time:

```
x = U\(L\(g(p,:)));
y = U\(L\(P*g));
```

Algorithm

For full matrices `X`, `lu` uses the LAPACK routines listed in the following table.

	Real	Complex
`X double`	DGETRF	ZGETRF
`X single`	SGETRF	CGETRF

For sparse `X`, with four outputs, `lu` uses UMFPACK routines. With three or fewer outputs, `lu` uses its own sparse matrix routines.

See Also

`cond`, `det`, `inv`, `luinc`, `qr`, `rref`

The arithmetic operators `\` and `/`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

[2] Davis, T. A., *UMFPACK Version 4.6 User Guide* (`http://www.cise.ufl.edu/research/sparse/umfpack`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2002.

luinc

Purpose

Sparse incomplete LU factorization

Syntax

```
luinc(A,'0')
luinc(A,droptol)
luinc(A,options)
[L,U] = luinc(A,0)
[L,U] = luinc(A,options)
[L,U,P] = luinc(...)
```

Description

`luinc` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`luinc(A,'0')` computes the incomplete LU factorization of level `0` of a square sparse matrix. The triangular factors have the same sparsity pattern as the permutation of the original sparse matrix `A`, and their product agrees with the permuted `A` over its sparsity pattern. `luinc(A,'0')` returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost, but `nnz(luinc(A,'0')) = nnz(A)`, with the possible exception of some zeros due to cancellation.

`luinc(A,droptol)` computes the incomplete LU factorization of any sparse matrix using the drop tolerance specified by the non-negative scalar `droptol`. The result is an approximation of the complete LU factors returned by `lu(A)`. For increasingly smaller values of the drop tolerance, this approximation improves until the drop tolerance is `0`, at which time the complete LU factorization is produced, as in `lu(A)`.

As each column `j` of the triangular incomplete factors is being computed, the entries smaller in magnitude than the local drop tolerance (the product of the drop tolerance and the norm of the corresponding column of `A`)

```
droptol*norm(A(:,j))
```

are dropped from the appropriate factor.

The only exceptions to this dropping rule are the diagonal entries of the upper triangular factor, which are preserved to avoid a singular factor.

`luinc(A,options)` computes the factorization with up to four options. These options are specified by fields of the input structure `options`. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`droptol`	Drop tolerance of the incomplete factorization.
`milu`	If `milu` is `1`, `luinc` produces the modified incomplete LU factorization that subtracts the dropped elements in any column from the diagonal element of the upper triangular factor. The default value is `0`.
`udiag`	If `udiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is `0`.
`thresh`	Pivot threshold between `0` (forces diagonal pivoting) and `1`, the default, which always chooses the maximum magnitude entry in the column to be the pivot. `thresh` is described in greater detail in the `lu` reference page.

`luinc(A,options)` is the same as `luinc(A,droptol)` if options has `droptol` as its only field.

`[L,U] = luinc(A,0)` returns the product of permutation matrices and a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The exact sparsity patterns of `L`, `U`, and `A` are not comparable but the number of nonzeros is maintained with the possible exception of some zeros in `L` and `U` due to cancellation:

```
nnz(L)+nnz(U) = nnz(A)+n, where A is n-by-n.
```

The product `L*U` agrees with `A` over its sparsity pattern. `(L*U).*spones(A)-A` has entries of the order of `eps`.

`[L,U] = luinc(A,options)` returns a permutation of a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The product `L*U` is an approximation to `A`. `luinc(A,options)` returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost.

`[L,U,P] = luinc(...)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

`[L,U,P] = luinc(A,'0')` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U` and a permutation matrix in `P`. `L` has the same sparsity pattern as the lower triangle of permuted `A`

```
spones(L) = spones(tril(P*A))
```

with the possible exceptions of `1`s on the diagonal of `L` where `P*A` may be zero, and zeros in `L` due to cancellation where `P*A` may be nonzero. `U` has the same sparsity pattern as the upper triangle of `P*A`

```
spones(U) = spones(triu(P*A))
```

with the possible exceptions of zeros in `U` due to cancellation where `P*A` may be nonzero. The product `L*U` agrees within rounding error with the permuted matrix `P*A` over its sparsity pattern. `(L*U).*spones(P*A)-P*A` has entries of the order of `eps`.

`[L,U,P] = luinc(A,options)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`. The nonzero entries of `U` satisfy

```
abs(U(i,j)) >= droptol*norm((A:,j)),
```

with the possible exception of the diagonal entries, which were retained despite not satisfying the criterion. The entries of `L` were tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`

```
abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).
```

The product `L*U` is an approximation to the permuted `P*A`.

Remarks

These incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. The lower triangular factors all have 1s along the main diagonal but a single 0 on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the `udiag` option to replace a zero diagonal only gets rid of the symptoms of the problem but does not solve it. The preconditioner may not be singular, but it probably is not useful and a warning message is printed.

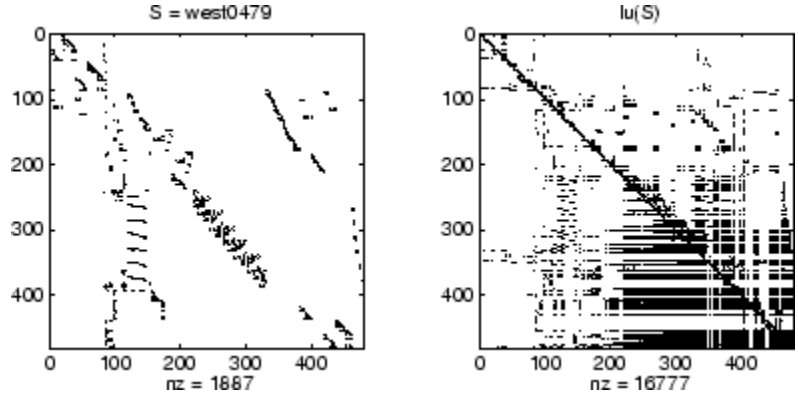
Limitations

`luinc(X,'0')` works on square matrices only.

Examples

Start with a sparse matrix and compute its LU factorization.

```
load west0479;
S = west0479;
[L,U] = lu(S);
```

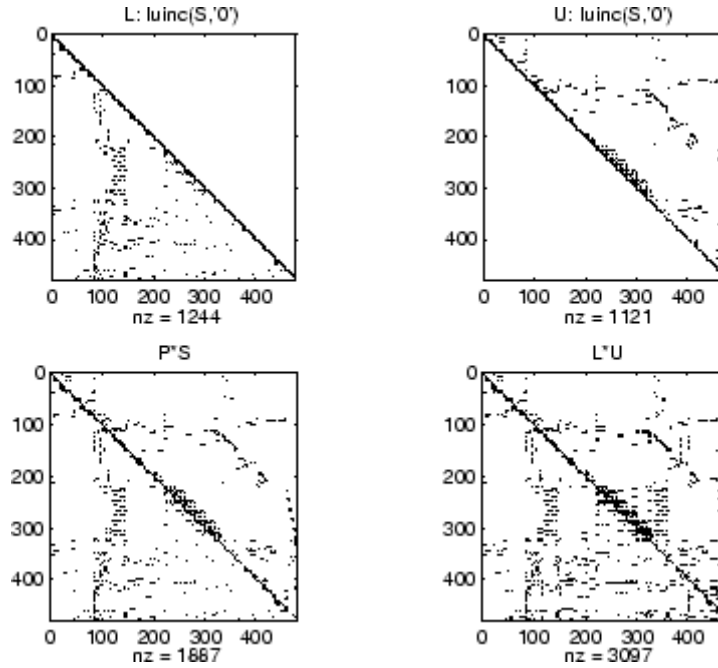


Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S,'0');
D = (L*U).*spones(P*S)-P*S;
```

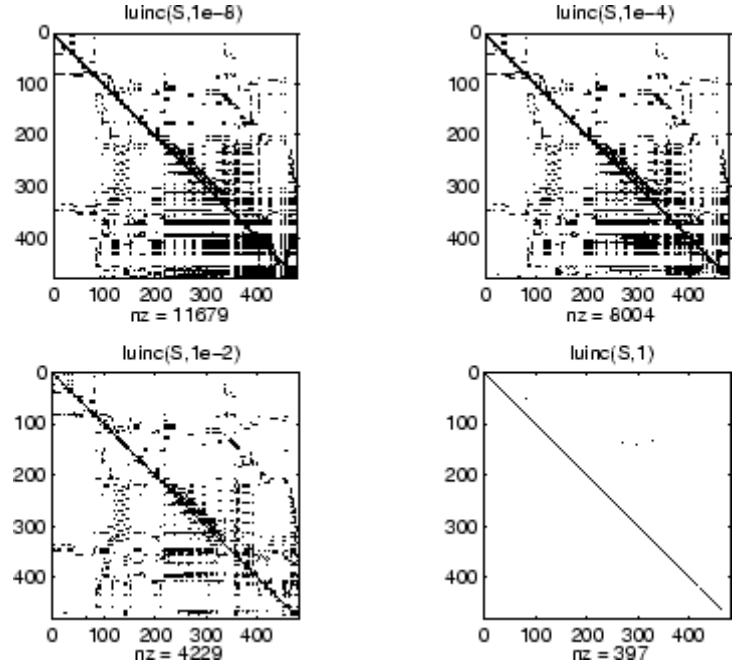
`spones(U)` and `spones(triu(P*S))` are identical.

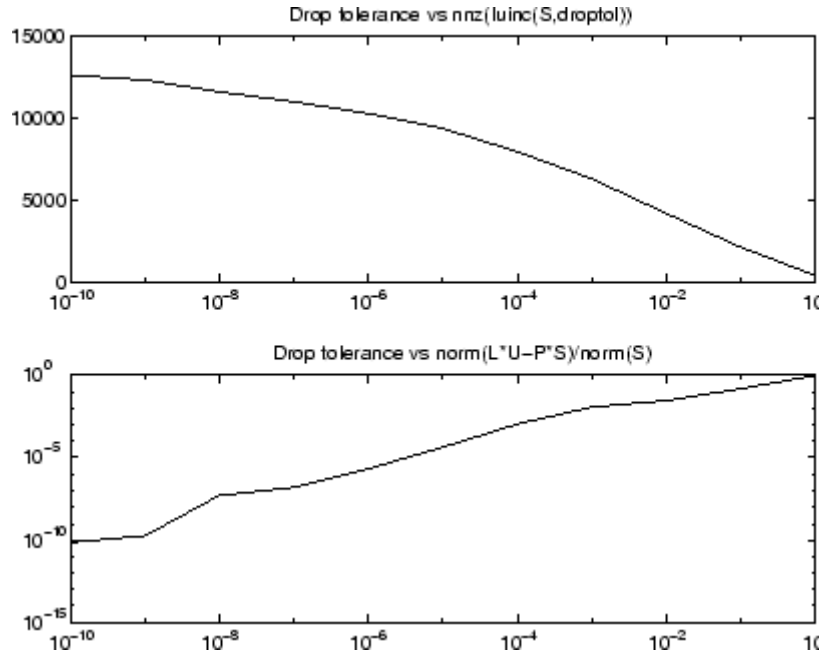
`spones(L)` and `spones(tril(P*S))` disagree at 73 places on the diagonal, where `L` is `1` and `P*S` is `0`, and also at position (206,113), where `L` is `0` due to cancellation, and `P*S` is `-1`. `D` has entries of the order of eps.




```
[IL0,IU0,IP0] = luinc(S,0);
[IL1,IU1,IP1] = luinc(S,1e-10);
                    .
                    .
                    .
```

A drop tolerance of `0` produces the complete LU factorization. Increasing the drop tolerance increases the sparsity of the factors (decreases the number of nonzeros) but also increases the error in the factors, as seen in the plot of drop tolerance versus `norm(L*U-P*S,1)/norm(S,1)` in the second figure below.


luinc(S,1e−8)
nz = 11679
luinc(S,1e−4)
nz = 8004
luinc(S,1e−2)
nz = 4229
luinc(S,1)
nz = 397



Algorithm

`luinc(A,'0')` is based on the "KJI" variant of the LU factorization with partial pivoting. Updates are made only to positions which are nonzero in `A`.

`luinc(A,droptol)` and `luinc(A,options)` are based on the column-oriented `lu` for sparse matrices.

See Also

`bicg`, `cholinc`,`ilu`, `lu`

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

Purpose

Magic square

Syntax

```
M = magic(n)
```

Description

`M = magic(n)` returns an `n`-by-`n` matrix constructed from the integers `1` through `n^2` with equal row and column sums. The order `n` must be a scalar greater than or equal to `3`.

Remarks

A magic square, scaled by its magic sum, is doubly stochastic.

Examples

The magic square of order 3 is

```
M = magic(3)

M =

     8     1     6
     3     5     7
     4     9     2
```

This is called a magic square because the sum of the elements in each column is the same.

```
sum(M) =

    15    15    15
```

And the sum of the elements in each row, obtained by transposing twice, is the same.

```
sum(M')' =

    15
    15
    15
```

This is also a special magic square because the diagonal elements have the same sum.

```
sum(diag(M)) =

     15
```

The value of the characteristic sum for a magic square of order `n` is

```
sum(1:n^2)/n
```

which, when `n = 3`, is 15.

Algorithm

There are three different algorithms:

- `n` odd
- `n` even but not divisible by four
- `n` divisible by four

To make this apparent, type

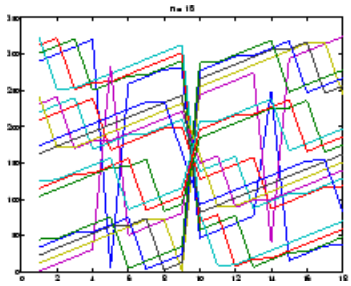
```
for n = 3:20
    A = magic(n);
    r(n) = rank(A);
end
```

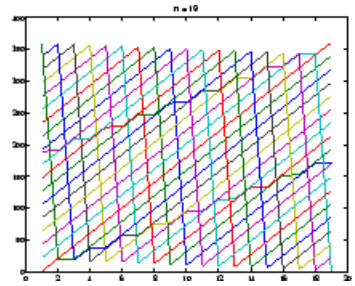
For `n` odd, the rank of the magic square is `n`. For `n` divisible by 4, the rank is 3. For `n` even but not divisible by 4, the rank is `n/2 + 2`.

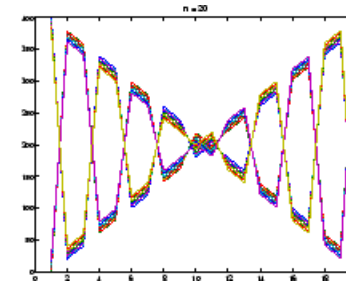
```
[(3:20)',r(3:20)']
ans =
     3     3
     4     3
     5     5
     6     5
     7     7
     8     3
     9     9
    10     7
    11    11
```

```
    12     3
    13    13
    14     9
    15    15
    16     3
    17    17
    18    11
    19    19
    20     3
```

Plotting A for n = 18, 19, 20 shows the characteristic plot for each category.







Limitations

If you supply `n` less than 3, `magic` returns either a nonmagic square, or else the degenerate magic squares `1` and `[]`.

See Also

`ones`, `rand`

makehgtform

Purpose

Create 4-by-4 transform matrix

Syntax

```
M = makehgtform
M = makehgtform('translate',[tx ty tz])
M = makehgtform('scale',s)
M = makehgtform('scale',[sx,sy,sz])
M = makehgtform('xrotate',t)
M = makehgtform('yrotate',t)
M = makehgtform('zrotate',t)
M = makehgtform('axisrotate',[ax,ay,az],t)
```

Description

Use `makehgtform` to create transform matrices for translation, scaling, and rotation of graphics objects. Apply the transform to graphics objects by assigning the transform to the `Matrix` property of a parent hgtransform object. See Examples for more information.

`M = makehgtform` returns an identity transform.

`M = makehgtform('translate',[tx ty tz])` or M = makehgtform('translate',tx,ty,tz) returns a transform that translates along the *x*-axis by `tx`, along the *y*-axis by `ty`, and along the *z*-axis by `tz`.

`M = makehgtform('scale',s)` returns a transform that scales uniformly along the *x*-, *y*-, and *z*-axes.

`M = makehgtform('scale',[sx,sy,sz])` returns a transform that scales along the *x*-axis by `sx`, along the *y*-axis by `sy`, and along the *z*-axis by `sz`.

`M = makehgtform('xrotate',t)` returns a transform that rotates around the *x*-axis by `t` radians.

`M = makehgtform('yrotate',t)` returns a transform that rotates around the *y*-axis by `t` radians.

`M = makehgtform('zrotate',t)` returns a transform that rotates around the *z*-axis by `t` radians.

`M = makehgtform('axisrotate',[ax,ay,az],t)` Rotate around axis `[ax ay az]` by `t` radians.

Note that you can specify multiple operations in one call to `makehgtform` and MATLAB returns a transform matrix that is the result of concatenating all specified operations. For example,

```
m = makehgtform('xrotate',pi/2,'yrotate',pi/2);
```

is the same as

```
m = makehgtform('xrotate',pi/2)*makehgtform('yrotate',pi/2);
```

See Also

`hgtransform`

Purpose

Divide matrix into cell array of matrices

Syntax

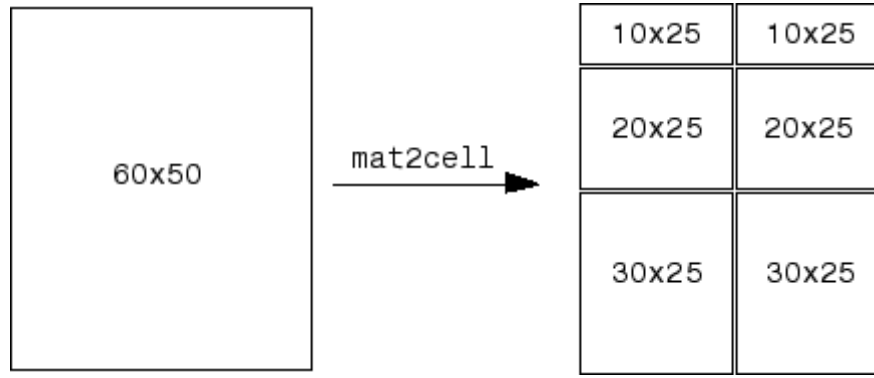
```
c = mat2cell(x, m, n)
c = mat2cell(x, d1, d2, ..., dn)
c = mat2cell(x, r)
```

Description

`c = mat2cell(x, m, n)` divides the two-dimensional matrix `x` into adjacent submatrices, each contained in a cell of the returned cell array `c`. Vectors `m` and `n` specify the number of rows and columns, respectively, to be assigned to the submatrices in `c`.

The example shown below divides a 60-by-50 matrix into six smaller matrices. MATLAB returns the new matrices in a 3-by-2 cell array:

```
mat2cell(x, [10 20 30], [25 25])
```



The sum of the element values in `m` must equal the total number of rows in `x`. And the sum of the element values in `n` must equal the number of columns in `x`.

The elements of `m` and `n` determine the size of each cell in `c` by satisfying the following formula for `i = 1:length(m)` and `j = 1:length(n)`:

```
size(c{i,j}) == [m(i) n(j)]
```

`c = mat2cell(x, d1, d2, ..., dn)` divides the multidimensional array `x` and returns a multidimensional cell array of adjacent submatrices of `x`. Each of the vector arguments `d1` through `dn` should sum to the respective dimension sizes of `x` such that, for `p = 1:n`,

```
size(x,p) == sum(dp)
```

The elements of `d1` through `dn` determine the size of each cell in `c` by satisfying the following formula for `ip = 1:length(dp)`:

```
size(c{i1,i2,...,in}) == [d1(i1) d2(i2) ... dn(in)]
```

If `x` is an empty array, `mat2cell` returns an empty cell array. This requires that all `dn` inputs that correspond to the zero dimensions of `x` be equal to `[]`.

For example,

```
a = rand(3,0,4);
c = mat2cell(a, [1 2], [], [2 1 1]);
```

`c = mat2cell(x, r)` divides an array `x` by returning a single-column cell array containing full rows of `x`. The sum of the element values in vector `r` must equal the number of rows of `x`.

The elements of `r` determine the size of each cell in `c`, subject to the following formula for `i = 1:length(r)`:

```
size(c{i},1) == r(i)
```

Remarks

`mat2cell` supports all array types.

Examples

Divide matrix `X` into 2-by-3 and 2-by-2 matrices contained in a cell array:

```
X = [1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15; 16 17 18 19 20]
X =
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
```

```
    16    17    18    19    20

C = mat2cell(X, [2 2], [3 2])
C =
    [2x3 double]    [2x2 double]
    [2x3 double]    [2x2 double]

C{1,1}                          C{1,2}
ans =                           ans =
     1     2     3                    4     5
     6     7     8                    9    10

C{2,1}                          C{2,2}
ans =                           ans =
    11    12    13                   14    15
    16    17    18                   19    20
```

See Also

cell2mat, num2cell

Purpose

Convert matrix to string

Syntax

```
str = mat2str(A)
str = mat2str(A,n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

Description

`str = mat2str(A)` converts matrix `A` into a string. This string is suitable for input to the `eval` function such that `eval(str)` produces the original matrix to within 15 digits of precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

`str = mat2str(A, 'class')` creates a string with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` digits of precision and includes the class information.

Limitations

The `mat2str` function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if `A` is a multidimensional array.

Examples

Example 1

Consider the matrix

```
x = [3.85 2.91; 7.74 8.99]
x =
    3.8500    2.9100
    7.7400    8.9900
```

The statement

```
A = mat2str(x)
```

produces

```
A =
```

```
[3.85 2.91;7.74 8.99]
```

where A is a string of 21 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(x))` reproduces `x`.

Example 2

Create a 1-by-6 matrix of signed 16-bit integers, and then use `mat2str` to convert the matrix to a 1-by-33 character array, `A`. Note that output string `A` includes the class name, `int16`:

```
x1 = int16([-300 407 213 418 32 -125]);

A = mat2str(x1, 'class')
A =
   int16([-300 407 213 418 32 -125])

class(A)
ans =
   char
```

Evaluating the string `A` gives you an output `x2` that is the same as the original `int16` matrix:

```
x2 = eval(A);

if isnumeric(x2) && isa(x2, 'int16') && all(x2 == x1)
   disp 'Conversion back to int16 worked'
end

Conversion back to int16 worked
```

See Also

`num2str`, `int2str`, `str2num`, `sprintf`, `fprintf`

Purpose

Control reflectance properties of surfaces and patches

Syntax

```
material shiny
material dull
material metal
material([ka kd ks])
material([ka kd ks n])
material([ka kd ks n sc])
material default
```

Description

`material` sets the lighting characteristics of `surface` and `patch` objects.

`material shiny` sets the reflectance properties so that the object has a high specular reflectance relative to the diffuse and ambient light, and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light and has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects.

`material default` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects to their defaults.

Remarks

The `material` command sets the `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, and `SpecularColorReflectance` properties of all `surface` and `patch` objects in the axes. There must be visible `light` objects in the `axes` for lighting to be enabled. Look at the `materal.m` M-file to see the actual values set (enter the command `type material`).

See Also

`light`, `lighting`, `patch`, `surface`

Lighting as a Visualization Tool for more information on lighting

“Lighting” on page 1-101 for related functions

Purpose

Run specified function via hyperlink

Syntax

```
disp('<a href="matlab: stmnt_1; stmnt_n;">hyperlink_text</a>')
```

Description

`matlab:` executes `stmnt_1` through `stmnt_n` when you click (or press **Ctrl+Enter**) in `hyperlink_text`. This must be used with another function, such as `disp`, where `disp` creates and displays underlined and colored `hyperlink_text` in the Command Window. Use `disp`, `error`, `fprintf`, `help`, or `warning` functions to display the hyperlink. The `hyperlink_text` is interpreted as HTML—you might need to use HTML character entity references or ASCII values for some special characters. Include the full hypertext string, from `'<a href=` to `</a>'` within a single line, that is, do not continue a long string on a new line. No spaces are allowed after the opening < and before the closing >. A single space is required between `a` and `href`.

Remarks

The `matlab:` function behaves differently with `diary`, `notebook`, `type`, and similar functions than might be expected. For example, if you enter the following statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

the diary file, when viewed in a text editor, shows

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
<a href="matlab:magic(4)">Generate magic square</a>
```

If you view the output of `diary` in the Command Window, the Command Window interprets the `<a href ...>` statement and does display it as a hyperlink.

Examples

Single Function

The statement

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

displays

Generate magic square

in the Command Window. When you click the link `Generate magic square`, MATLAB runs `magic(4)`.

Multiple Functions

You can include multiple functions in the statement, such as

```
disp('<a href="matlab:  x=0:1:8;y=sin(x);plot(x,y)">Plot x,y</a>')
```

which displays

Plot x,y

in the Command Window. When you click the link, MATLAB runs

```
x = 0:1:8;
y = sin(x);
plot(x,y)
```

Clicking the Hyperlink Again

After running the statements in the hyperlink `Plot x,y` defined in the previous example, “Multiple Functions” on page 2-2086, you can subsequently redefine `x` in the base workspace, for example, as

```
x = -2*pi:pi/16:2*pi;
```

If you then click the hyperlink, `Plot x,y`, it changes the current value of `x` back to

```
0:1:8
```

because the `matlab:` statement defines `x` in the base workspace. In the `matlab:` statement that displayed the hyperlink, `Plot x,y`, `x` was defined as 0:1:8.

Presenting Options

Use multiple `matlab:` statements in an M-file to present options, such as

```
disp('<a href = "matlab:state = 0">Disable feature</a>')
disp('<a href = "matlab:state = 1">Enable feature</a>')
```

The Command Window displays

Disable feature
Enable feature

and depending on which link is clicked, sets `state` to `0` or `1`.

Special Characters

MATLAB correctly interprets most strings that includes special characters, such as a greater than sign. For example, the following statement includes a `>`

```
disp('<a href="matlab:str = ''Value > 0''">Positive</a>
```

and generates the following hyperlink.

Positive

Some symbols might not be interpreted correctly and you might need to use the HTML character entity reference for the symbol. For example, an alternative way to run the same statement is to use the `>` character entity reference instead of the > symbol:

```
disp('<a href="matlab:str = ''Value &gt; 0''">Positive</a>')
```

Instead of the HTML character entity reference, you can use the ASCII value for the symbol. For example, the greater than sign, >, is ASCII 62. The above example becomes

```
disp('<a href="matlab:str=[''Value '' char(62) '' 0'']">Positive</a>')
```

Here are some values for common special characters.

Character	HTML Character Entity Reference	ASCII Value
>	>	62
<	<	60
&	&	38
"	"	34

For a list of all HTML character entity references, see `http://www.w3.org/`.

Links from M-File Help

For functions you create, you can include `matlab:` links within the M-file help, but you do not need to include a `disp` or similar statement because the `help` function already includes it for displaying hyperlinks. Use the links to display additional help in a browser when the user clicks them. The M-file `soundspeed` contains the following statements:

```
function c=soundspeed(s,t,p)

% Speed of sound in water, using
% <a href="matlab: web('http://www.zu.edu')">Wilson's
formula</a>
% Where c is the speed of sound in water in m/s
```

etc.

Run `help soundspeed` and MATLAB displays the following in the Command Window.

```
>> help soundspeed
  Speed of sound in water, using
  Wilson's formula
  Where c is the speed of sound in water in m/s
```

When you click the link `Wilson's formula`, MATLAB displays the HTML page http://www.zu.edu in the Web browser. Note that this URL is only an example and is invalid.

See Also

`disp`, `error`, `fprintf`, `input`, `run`, `warning`

matlabrc

Purpose

MATLAB startup M-file for single-user systems or system administrators

Description

At startup time, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on the MATLAB search path.

As an individual user, you can create a startup file in your own MATLAB directory. Use the startup file to define physical constants, engineering conversion factors, graphics defaults, or anything else you want predefined in your workspace.

Algorithm

Only `matlabrc` is actually invoked by MATLAB at startup. However, `matlabrc.m` contains the statements

```
if exist('startup') == 2
    startup
end
```

that invoke `startup.m`. Extend this process to create additional startup M-files, if required.

Remarks

You can also start MATLAB using options you define at the Command Window prompt or in your Windows shortcut for MATLAB.

Examples

Turning Off the Figure Window Toolbar

If you do not want the toolbar to appear in the figure window, remove the comment marks from the following line in the `matlabrc.m` file, or create a similar line in your own `startup.m` file.

```
%   set(0,'defaultfiguretoolbar','none')
```

See Also

`matlabroot`, `quit`, `restoredefaultpath`, `startup`

Startup Options in the MATLAB Desktop Tools and Development Environment documentation

matlabroot

Purpose

Root directory of MATLAB installation

Syntax

```
matlabroot
rd = matlabroot
```

Description

`matlabroot` returns the name of the directory in which the MATLAB software is installed. In compiled M-code, it returns the path to the executable. Use `matlabroot` to create a path to MATLAB and toolbox directories that does not depend on a specific platform, MATLAB version, or installation directory.

`rd = matlabroot` returns the name of the directory in which the MATLAB software is installed and assigns it to `rd`.

Remarks

matlabroot

Run

```
matlabroot
```

MATLAB returns, for example,

```
\\H:\Programs\matlab
```

matlabroot as Directory Name

The term *`matlabroot`* is sometimes used to represent the directory where MATLAB files are installed and should not be confused with the `matlabroot` function. For example, "save to *`matlabroot`*`/toolbox/local`" means save to the `toolbox/local` directory in the MATLAB root directory.

$matlabroot

Sometimes the term `$matlabroot` is used to represent the value returned by the `matlabroot` function.

But in some files, such as `info.xml` and `classpath.txt`, `$matlabroot`, the preceding `$` is literal. MATLAB actually interprets `$matlabroot`

as the full path to the MATLAB root directory. For example, including the line

```
$matlabroot/toolbox/local/myfile.jar
```

in `classpath.txt`, adds `myfile.jar`, which is located in the `toolbox/local` directory, to `classpath.txt`.

Examples

`fullfile(matlabroot,'toolbox','matlab','general')`

produces a full path to the `toolbox/matlab/general` directory that is correct for the platform it is executed on.

`cd(matlabroot)` changes the current working directory to the MATLAB root directory.

`addpath([matlabroot '/toolbox/local/myfiles'])` adds the directory `myfiles` to the MATLAB search path.

See Also

`ctfroot` (in MATLAB Compiler), `fullfile`, `partialpath`, `path`, `toolboxdir`

matlab (UNIX)

Purpose

Start MATLAB (UNIX systems)

Syntax

```
matlab helpOption
matlab archOption
matlab dispOption
matlab modeOption
matlab mgrOption
matlab -c licensefile
matlab -r command
matlab -logfile filename
matlab -mwvisual visualid
matlab -nosplash
matlab -timing
matlab -debug
matlab -Ddebugger options
```

Note You can enter more than one of these options in the same MATLAB command. If you use `-Ddebugger` to start MATLAB in debug mode, the first option in the command must be `-Ddebugger`.

Description

`matlab` is a Bourne shell script that starts the MATLAB executable. (In this document, `matlab` refers to this script; MATLAB refers to the application program). Before actually initiating the execution of MATLAB, this script configures the run-time environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Processing any command line options
- Reading the MATLAB startup file, `.matlab7rc.sh`
- Setting MATLAB environment variables

There are two ways in which you can control the way the `matlab` script works:

- By specifying command line options
- By assigning values in the MATLAB startup file, `.matlab7rc.sh`

Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab helpOption` displays information that matches the specified `helpOption` argument without starting MATLAB. `helpOption` can be any one of the keywords shown in the table below. Enter only one `helpOption` keyword in a `matlab` command.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.
`-n`	Display all the final values of the environment variables and arguments passed to the MATLAB executable as well as other diagnostic information.
`-e`	Display *all* environment variables and their values just prior to exiting. This argument must have been parsed before exiting for anything to be displayed. The last possible exiting point is just before the MATLAB image would have been executed and a status of `0` is returned. If the exit status is not `0` on return, then the variables and values may not be correct.

`matlab archOption` starts MATLAB and assumes that you are running on the system architecture specified by `arch`, or using the MATLAB version specified by `variant`, or both. The values for the `archOption` argument are shown in the table below. Enter only one of these options in a `matlab` command.

Values for archOption

Option	Description
`-arch`	Run MATLAB assuming this architecture rather than the actual architecture of the machine you are using. Replace the term `arch` with a string representing a recognized system architecture.
`v``=variant`	Execute the version of MATLAB found in the directory `bin/$ARCH/variant` instead of `bin/$ARCH`. Replace the term `variant` with a string representing a MATLAB version.
`v``=arch/variant`	Execute the version of MATLAB found in the directory `bin/arch/variant` instead of `bin/$ARCH`. Replace the terms `arch` and `variant` with strings representing a specific architecture and MATLAB version.

`matlab dispOption` starts MATLAB using one of the display options shown in the table below. Enter only one of these options in a `matlab` command.

Values for dispOption

Option	Description
`-display` `xDisp`	Send X commands to X Window Server display `xDisp`. This supersedes the value of the `DISPLAY` environment variable.
`-nodisplay`	Start the Java virtual machine, but do not start the MATLAB desktop. Do not display any X commands, and ignore the `DISPLAY` environment variable,

`matlab modeOption` starts MATLAB without its usual desktop component. Enter only one of the options shown below.

Values for modeOption

Option	Description
`-desktop`	Allow the MATLAB desktop to be started by a process without a controlling terminal. This is usually a required command line argument when attempting to start MATLAB from a window manager menu or desktop icon.
`-nodesktop`	Start MATLAB without its desktop. The Java virtual machine (JVM) is started. Use the current window to enter commands. Start any desktop tools using command equivalents, such as `helpbrowser` to open the Help browser. MATLAB does not save statements to the Command History.
`-nojvm`	Start MATLAB without the Java virtual machine (JVM). Use the current window to enter commands. The MATLAB desktop does not open and any tools that require Java, such as the desktop tools, cannot be used. Also, figures do not display the menu bar or toolbar.

`matlab mgrOption` starts MATLAB in the memory management mode specified by `mgrOption`. Enter only one of the options shown below.

Values for mgrOption

Option	Description
-memmgr manager	Set environment variable MATLAB_MEM_MGR to manager. The manager argument can have one of the following values: • **cache** — The default. • **compact** — This is useful for large models or MATLAB code that uses many structure or object variables. It is not helpful for large arrays. (This option applies only to 32-bit architectures.) • **debug** — Does memory integrity checking and is useful for debugging memory problems caused by user-created MEX files.
-check_malloc	The same as using '**-memmgr debug**'.

matlab **-c** licensefile starts MATLAB using the specified license file. The licensefile argument can have the form port@host or it can be a colon-separated list of license filenames. This option causes the LM_LICENSE_FILE and MLM_LICENSE_FILE environment variables to be ignored.

matlab **-r** command starts MATLAB and executes the specified MATLAB command.

matlab **-logfile** filename starts MATLAB and makes a copy of any output to the command window in file log. This includes all crash reports.

matlab **-mwvisual** visualid starts MATLAB and uses visualid as the default X visual for figure windows. visualid is a hexadecimal number that can be found using xdpyinfo.

`matlab -nosplash` starts MATLAB but does not display the splash screen during startup.

`matlab -timing` starts MATLAB and prints a summary of startup time to the command window. This information is also recorded in a timing log, the name of which is printed to the shell window in which MATLAB is started. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab -debug` starts MATLAB and displays debugging information that can be useful, especially for X based problems. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab -Ddebugger options` starts MATLAB in debug mode, using the named `debugger` (e.g., `dbx`, `gdb`, `xdb`, `cvd`). A full path can be specified for debugger.

The `options` argument can include *only* those options that follow the debugger name in the syntax of the actual debug command. For most debuggers, there is a very limited number of such options. Options that would normally be passed to the MATLAB executable should be used as parameters of a command inside the debugger (like `run`). They should not be used when running the MATLAB script.

If any other `matlab` command options are placed before the `-Ddebugger` argument, they will be handled as if they were part of the options after the `-Ddebugger` argument and will be treated as illegal options by most debuggers. The `MATLAB_DEBUG` environment variable is set to the filename part of the debugger argument.

To customize your debugging session, use a startup file. See your debugger documentation for details.

Note For certain debuggers like `gdb`, the `SHELL` environment variable is *always* set to `/bin/sh`.

Specifying Options in the MATLAB Startup File

The `.matlab7rc.sh` shell script contains definitions for a number of variables that the `matlab` script uses. These variables are defined within the `matlab` script, but can be redefined in `.matlab7rc.sh`. When invoked, `matlab` looks for the first occurrence of `.matlab7rc.sh` in the current directory, in the home directory (`$HOME`), and in the *`matlabroot`*`/bin` directory, where the template version of `.matlab7rc.sh` is located.

You can edit the template file to redefine information used by the `matlab` script. If you do not want your changes applied systemwide, copy the edited version of the script to your current or home directory. Ensure that you edit the section that applies to your machine architecture.

The following table lists the variables defined in the `.matlab7rc.sh` file. See the comments in the `.matlab7rc.sh` file for more information about these variables.

Variable	Definition and Standard Assignment Behavior
`ARCH`	The machine architecture. The value `ARCH` passed with the `-arch` or `-arch/ext` argument to the script is tried first, then the value of the environment variable `MATLAB_ARCH` is tried next, and finally it is computed. The first one that gives a valid architecture is used.
`AUTOMOUNT_MAP`	Path prefix map for automounting. The value set in `.matlab7rc.sh` (initially by the installer) is used unless the value differs from that determined by the script, in which case the value in the environment is used.

Variable	Definition and Standard Assignment Behavior
DISPLAY	The hostname of the X Window display MATLAB uses for output. The value of Xdisplay passed with the -display argument to the script is used; otherwise, the value in the environment is used. DISPLAY is ignored by MATLAB if the -nodisplay argument is passed.
LD_LIBRARY_PATH	Final Load library path. The name LD_LIBRARY_PATH is platform dependent. The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in .matlab7rc.sh as LDPATH_PREFIX. The second sublist is computed in the script and includes directories inside the MATLAB root directory and relevant Java directories. The third sublist contains any nonempty value of LD_LIBRARY_PATH from the environment possibly augmented in .matlab7rc.sh. The final sublist is defined in .matlab7rc.sh as LDPATH_SUFFIX.

Variable	Definition and Standard Assignment Behavior
LM_LICENSE_FILE	The FLEX lm license variable. The license file value passed with the -c argument to the script is used; otherwise it is the value set in .matlab7rc.sh. In general, the final value is a colon-separated list of license files and/or port@host entries. The shipping .matlab7rc.sh file starts out the value by prepending LM_LICENSE_FILE in the environment to a default license.file. Later in the MATLAB script if the -c option is not used, the *matlabroot*/etc directory is searched for the files that start with license.dat.DEMO. These files are assumed to contain demo licenses and are added automatically to the end of the current list.
MATLAB	The MATLAB root directory. The default computed by the script is used unless MATLABdefault is reset in .matlab7rc.sh. Currently MATLABdefault is not reset in the shipping .matlab7rc.sh.
MATLAB_DEBUG	Normally set to the name of the debugger. The -Ddebugger argument passed to the script sets this variable. Otherwise, a nonempty value in the environment is used.

Variable	Definition and Standard Assignment Behavior
MATLAB_JAVA	The path to the root of the Java Runtime Environment. The default set in the script is used unless MATLAB_JAVA is already set. Any nonempty value from .matlab7rc.sh is used first, then any nonempty value from the environment. Currently there is no value set in the shipping .matlab67rc.sh, so that environment alone is used.
MATLAB_MEM_MGR	Turns on MATLAB memory integrity checking. The -check_malloc argument passed to the script sets this variable to 'debug'. Otherwise, a nonempty value set in .matlab7rc.sh is used, or a nonempty value in the environment is used. If a nonempty value is not found, the variable is not exported to the environment.
MATLABPATH	The MATLAB search path. The final value is a colon-separated list with the MATLABPATH from the environment prepended to a list of computed defaults.

Variable	Definition and Standard Assignment Behavior
SHELL	The shell to use when the "!" or unix command is issued in MATLAB. This is taken from the environment unless SHELL is reset in .matlab7rc.sh. Note that an additional environment variable called MATLAB_SHELL takes precedence over SHELL. MATLAB checks internally for MATLAB_SHELL first and, if empty or not defined, then checks SHELL. If SHELL is also empty or not defined, MATLAB uses /bin/sh. The value of MATLAB_SHELL should be an absolute path, i.e. /bin/sh, not simply sh. Currently, the shipping .matlab7rc.sh file does not reset SHELL and also does not reference or set MATLAB_SHELL.
TOOLBOX	Path of the toolbox directory. A nonempty value in the environment is used first. Otherwise, *matlabroot*/toolbox, computed by the script, is used unless TOOLBOX is reset in .matlab7rc.sh. Currently TOOLBOX is not reset in the shipping .matlab7rc.sh.

Variable	Definition and Standard Assignment Behavior
XAPPLRESDIR	The X application resource directory. A nonempty value in the environment is used first unless XAPPLRESDIR is reset in .matlab7rc.sh. Otherwise, *matlabroot*/X11/app-defaults, computed by the script, is used.
XKEYSYMDB	The X keysym database file. A nonempty value in the environment is used first unless XKEYSYMDB is reset in .matlab7rc.sh. Otherwise, *matlabroot*/X11/app-defaults/XKeysymDB, computed by the script, is used. The matlab script determines the path of the MATLAB root directory as one level up the directory tree from the location of the script. Information in the AUTOMOUNT_MAP variable is used to fix the path so that it is correct to force a mount. This can involve deleting part of the pathname from the front of the MATLAB root path. The MATLAB variable is then used to locate all files within the MATLAB directory tree.

The matlab script determines the path of the MATLAB root directory by looking up the directory tree from the *matlabroot*/bin directory (where the matlab script is located). The MATLAB variable is then used to locate all files within the MATLAB directory tree.

You can change the definition of MATLAB if, for example, you want to run a different version of MATLAB or if, for some reason, the path determined by the matlab script is not correct. (This can happen when certain types of automounting schemes are used by your system.)

AUTOMOUNT_MAP is used to modify the MATLAB root directory path. The pathname that is assigned to AUTOMOUNT_MAP is deleted from the

front of the MATLAB root path. (It is unlikely that you will need to use this option.)

See Also

`mex`

"Startup Options" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Start MATLAB (Windows systems)

Syntax

```
matlab helpOption
matlab mgrOption
matlab -automation
matlab -c licensefile
matlab -logfile filename
matlab -nosplash
matlab -noFigureWindows
matlab -r "statement"
matlab -regserver
matlab -sd "startdir"
matlab -timing
matlab -unregserver
```

Note You can enter more than one of these options in the same MATLAB command.

Description

`matlab` is a script that runs the main MATLAB executable. (In this document, the term `matlab` refers to the script, and MATLAB refers to the main executable). Before actually initiating the execution of MATLAB, it configures the run-time environment by

- Determining the MATLAB root directory
- Determining the host machine architecture
- Selectively processing command line options with the rest passed to MATLAB.
- Setting certain MATLAB environment variables

There are two ways in which you can control the way `matlab` works:

- By specifying command line options
- By setting environment variables before calling the program

Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab` *`helpOption`* displays information that matches the specified *`helpOption`* argument without starting MATLAB. *`helpOption`* can be any one of the keywords shown in the table below. Enter only one *`helpOption`* keyword in a `matlab` command.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.
`-?`	The same as **`-help`**.

`matlab` *`mgrOption`* starts MATLAB in the memory management mode specified by *`mgrOption`*. Enter only one of the options shown below.

Values for mgrOption

Option	Description
-memmgr *manager*	Set environment variable MATLAB_MEM_MGR to *manager*. The manager argument can have one of the following values: • **cache** — The default. • **fast** — For large models or MATLAB code that uses many structure or object variables. It is not helpful for large arrays. • **debug** — Does memory integrity checking and is useful for debugging memory problems caused by user-created MEX files.
-check_malloc	The same as using '**-memmgr debug**'.

matlab **-automation** starts MATLAB as an automation server. The server window is minimized, and the MATLAB splash screen is not displayed on startup.

matlab **-c** licensefile starts MATLAB using the specified license file. The licensefile argument can have the form port@host. This option causes the LM_LICENSE_FILE and MLM_LICENSE_FILE environment variables to be ignored.

matlab **-logfile** filename starts MATLAB and makes a copy of any output to the Command Window in filename. This includes all crash reports.

matlab **-nosplash** starts MATLAB but does not display the splash screen during startup.

matlab **-noFigureWindows** starts MATLAB but disables the display of any figure windows in MATLAB.

matlab **-r** "statement" starts MATLAB and executes the specified MATLAB statement. Any required file must be on the MATLAB path or in the startup directory.

`matlab` **`-regserver`** registers MATLAB as a Component Object Model (COM) server.

`matlab` **`-sd`** `"startdir"` specifies the startup directory for MATLAB (the current directory in MATLAB after startup). When you do not specify the **`-sd`** option, the startup directory is the directory from which you ran `matlab`. For more information, see "Startup Directory (Folder) on Windows Platforms".

`matlab` **`-timing`** starts MATLAB and prints a summary of startup time to the command window. This information is also recorded in a timing log, the name of which is printed to the MATLAB Command Window. This option should be used only when working with a Technical Support Representative from The MathWorks.

`matlab` **`-unregserver`** removes all MATLAB COM server entries from the registry.

Setting Environment Variables

You can set any of the following environment variables before starting MATLAB.

Variable Name	Description
`LM_LICENSE_FILE`	This is the FLEX lm license variable. The license file value passed with the **`-c`** argument to the script is used; otherwise it is the value set in the environment. The final value is a colon-separated list of license files and/or `port@host` entries.
`MATLAB_MEM_MGR`	This determines the type of memory manager used by MATLAB. If not set in the environment, it is controlled by passing its value via the '**`-memmgr`**' option. If no value is predefined, then MATLAB uses '**`cache`**'.

See Also

`mex`

"Startup Options" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Largest elements in array

Syntax

```
C = max(A)
C = max(A,B)
C = max(A,[],dim)
[C,I] = max(...)
```

Description

`C = max(A)` returns the largest elements along different dimensions of an array.

If `A` is a vector, `max(A)` returns the largest element in `A`.

If `A` is a matrix, `max(A)` treats the columns of `A` as vectors, returning a row vector containing the maximum element from each column.

If `A` is a multidimensional array, `max(A)` treats the values along the first non-singleton dimension as vectors, returning the maximum value of each vector.

`C = max(A,B)` returns an array the same size as `A` and `B` with the largest elements taken from `A` or `B`. The dimensions of `A` and `B` must match, or they may be scalar.

`C = max(A,[],dim)` returns the largest elements along the dimension of `A` specified by scalar `dim`. For example, `max(A,[],1)` produces the maximum values along the first dimension (the rows) of `A`.

`[C,I] = max(...)` finds the indices of the maximum values of `A`, and returns them in output vector `I`. If there are several identical maximum values, the index of the first one found is returned.

Remarks

For complex input `A`, `max` returns the complex number with the largest complex modulus (magnitude), computed with `max(abs(A))`. Then computes the largest phase angle with `max(angle(x))`, if necessary.

The `max` function ignores `NaN`s.

See Also

`isnan`, `mean`, `median`, `min`, `sort`

Purpose

Maximum value of `timeseries` data

Syntax

```
ts_max = max(ts)
ts_max = max(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_max = max(ts)` returns the maximum value in the time-series data. When `ts.Data` is a vector, ts_max is the maximum value of `ts.Data` values. When `ts.Data` is a matrix, ts_max is a row vector containing the maximum value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `max` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_max = max(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example illustrates how to find the maximum values in multivariate time-series data.

1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Find the maximum in each data column for this `timeseries` object.

```
max(count_ts)

ans =

   114   145   257
```

The maximum is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `min (timeseries)`, `median (timeseries)`, `mean (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

Purpose

Open server window on Windows desktop

Syntax

MATLAB Client

```
h.MaximizeCommandWindow
MaximizeCommandWindow(h)
invoke(h, 'MaximizeCommandWindow')
```

Method Signature

```
HRESULT MaximizeCommandWindow(void)
```

Visual Basic Client

```
MaximizeCommandWindow
```

Description

`MaximizeCommandWindow` displays the window for the server attached to handle `h`, and makes it the currently active window on the desktop. If the server window was not in a minimized state to begin with, then `MaximizeCommandWindow` does nothing.

Note `MaximizeCommandWindow` does not maximize the server window to its maximum possible size on the desktop. It restores the window to the size it had at the time it was minimized.

Remarks

Server function names, like `MaximizeCommandWindow`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Examples

Create a COM server and minimize its window. Then maximize the window and make it the currently active window.

MATLAB Client

```
h = actxserver('matlab.application');
```

```
h.MinimizeCommandWindow;
% Now return the server window to its former state on
% the desktop and make it the currently active window.
h.MaximizeCommandWindow;
```

Visual Basic .NET Client

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

MinimizeCommandWindow

Purpose

Controls maximum number of computational threads

Syntax

```
N = maxNumCompThreads
LASTN = maxNumCompThreads(N)
LASTN = maxNumCompThreads('automatic')
```

Description

`N = maxNumCompThreads` returns the current maximum number of computational threads `N`.

`LASTN = maxNumCompThreads(N)` sets the maximum number of computational threads to `N`, and returns the previous maximum number of computational threads, `LASTN`.

`LASTN = maxNumCompThreads('automatic')` sets the maximum number of computational threads using what MATLAB determines to be the most desirable. It additionally returns the previous maximum number of computational threads, `LASTN`.

Currently, the maximum number of computational threads is equal to the number of computational cores on your machine.

Note Unlike enabling multithreading using the Preferences panel, setting the maximum number of computational threads using `maxNumCompThreads` will not propagate to your next MATLAB session.

Purpose

Average or mean value of array

Syntax

```
M = mean(A)
M = mean(A,dim)
```

Description

`M = mean(A)` returns the mean values of the elements along different dimensions of an array.

If `A` is a vector, `mean(A)` returns the mean value of `A`.

If `A` is a matrix, `mean(A)` treats the columns of `A` as vectors, returning a row vector of mean values.

If `A` is a multidimensional array, `mean(A)` treats the values along the first non-singleton dimension as vectors, returning an array of mean values.

`M = mean(A,dim)` returns the mean values for elements along the dimension of `A` specified by scalar `dim`. For matrices, `mean(A,2)` is a column vector containing the mean value of each row.

Examples

```
A = [1 2 3; 3 3 6; 4 6 8; 4 7 7];
mean(A)
ans =
    3.0000    4.5000    6.0000

mean(A,2)
ans =
    2.0000
    4.0000
    6.0000
    6.0000
```

See Also

`corrcoef`, `cov`, `max`, `median`, `min`, `mode`, `std`, `var`

Purpose

Mean value of `timeseries` data

Syntax

```
ts_mn = mean(ts)
ts_mn = mean(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_mn = mean(ts)` returns the mean value of `ts.Data`. When `ts.Data` is a vector, ts_mn is the mean value of `ts.Data` values. When `ts.Data` is a matrix, ts_mn is a row vector containing the mean value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `mean` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_mn = mean(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example illustrates how to find the mean values in multivariate time-series data.

1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Find the mean of each data column for this `timeseries` object.

```
mean(count_ts)

ans =

   32.0000   46.5417   65.5833
```

The mean is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `max (timeseries)`, `min (timeseries)`, `median (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

Purpose

Median value of array

Syntax

```
M = median(A)
M = median(A,dim)
```

Description

`M = median(A)` returns the median values of the elements along different dimensions of an array.

If `A` is a vector, `median(A)` returns the median value of `A`.

If `A` is a matrix, `median(A)` treats the columns of `A` as vectors, returning a row vector of median values.

If `A` is a multidimensional array, `median(A)` treats the values along the first nonsingleton dimension as vectors, returning an array of median values.

`M = median(A,dim)` returns the median values for elements along the dimension of `A` specified by scalar `dim`.

Examples

```
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
median(A)

ans =

     4     5     7     7

median(A,2)

ans =

     3
     5
     7
     7
```

See Also

`corrcoef`, `cov`, `max`, `mean`, `min`, `mode`, `std`, `var`

median (timeseries)

Purpose

Median value of `timeseries` data

Syntax

```
ts_med = median(ts)
ts_med = median(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_med = median(ts)` returns the median value of `ts.Data`. When `ts.Data` is a vector, ts_med is the median value of `ts.Data` values. When `ts.Data` is a matrix, ts_med is a row vector containing the median value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `median` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_med = median(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example illustrates how to find the median values in multivariate time-series data.

1. Load a 24-by-3 data array.

   ```
   load count.dat
   ```

2. Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Find the median of each data column for this `timeseries` object.

```
median(count_ts)

ans =

   23.5000   36.0000   39.0000
```

The median is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `max (timeseries)`, `min (timeseries)`, `mean (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

memmapfile

Purpose

Construct memmapfile object

Syntax

```
m = memmapfile(filename)
m = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

Description

`m = memmapfile(filename)` constructs an object of the `memmapfile` class that maps file `filename` to memory using the default property values. The `filename` input is a quoted string that specifies the path and name of the file to be mapped into memory. `filename` must include a filename extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), is case sensitive on UNIX platforms, but is not case sensitive on Windows.

`m = memmapfile(filename, prop1, value1, prop2, value2, ...)` constructs an object of the `memmapfile` class that maps file `filename` into memory and sets the properties of that object that are named in the argument list (`prop1`, `prop2`, etc.) to the given values (`value1`, `value2`, etc.). All property name arguments must be quoted strings (e.g., `'Writable'`). Any properties that are not specified are given their default values.

Optional properties are shown in the table below and are described in the sections that follow.

Property	Description	Data Type	Default
`Format`	Format of the contents of the mapped region, including data type, array shape, and variable or field name by which to access the data	`char` array or N-by-3 `cell` array	`uint8`
`Offset`	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file.	`double`	`0`
`Repeat`	Number of times to apply the specified format to the mapped region of the file	`double`	`Inf`
`Writable`	Type of access allowed to the mapped region	`logical`	false

There are three different ways you can specify a value for the `Format` property. See the following sections in the MATLAB Programming documentation for more information on this:

-
-
-

Any of the following data types can be used when you specify a `Format` value. The default type is `uint8`.

Format String	Data Type Description
`'int8'`	Signed 8-bit integers
`'int16'`	Signed 16-bit integers
`'int32'`	Signed 32-bit integers
`'int64'`	Signed 64-bit integers
`'uint8'`	Unsigned 8-bit integers
`'uint16'`	Unsigned 16-bit integers
`'uint32'`	Unsigned 32-bit integers
`'uint64'`	Unsigned 64-bit integers
`'single'`	32-bit floating-point
`'double'`	64-bit floating-point

Remarks

You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Once `memmapfile` locates the file, MATLAB stores the absolute pathname for the file internally, and then uses this stored path to locate the file from that point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. To set a new offset value for memory map object `m`, type

```
m.Offset = 2048;
```

Property names are not case sensitive. For example, MATLAB considers `m.Offset` to be the same as `m.offset`.

Examples

Example 1

To construct a map for the file `records.dat` that resides in your current working directory, type the following:

```
m = memmapfile('records.dat');
```

MATLAB constructs an instance of the `memmapfile` class, assigns it to the variable `m`, and maps the entire `records.dat` file to memory, setting all properties of the object to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

Example 2

To construct a map using nondefault values for the `Offset`, `Format`, and `Writable` properties, type the following, enclosing all property names in single quotation marks:

```
m = memmapfile('records.dat',         ...
               'Offset', 1024,        ...
               'Format', 'uint32',    ...
               'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m

m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
      Format: 'uint32'
      Repeat: Inf
        Data: 4778x1 uint32 array
```

Example 3

Construct a `memmapfile` object for the entire file `records.dat` and set the `Format` property for that object to `uint64`. Any read or write operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64');
```

Example 4

Construct a `memmapfile` object for a region of `records.dat` such that the contents of the region are handled by MATLAB as a 4-by-10-by-18 array of unsigned 32-bit integers, and can be referenced in the structure of the returned object using the field name `x`:

```
m = memmapfile('records.dat',    ...
               'Offset', 1024,   ...
               'Format', {'uint32' [4 10 18] 'x'});

A = m.Data.x;

whos A
  Name      Size                     Bytes  Class

  A        4x10x18                   2880   uint32 array

Grand total is 720 elements using 2880 bytes
```

Example 5

Map a 24 kilobyte file containing data of three different data types: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`.

Each of these fields belongs to the 800-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat',                    ...
               'Offset', 2048,                   ...
```

```
                    'Format', {                        ...
                       'int16'  [2 2] 'model';         ...
                       'uint32' [1 1] 'serialno';      ...
                       'single' [1 3] 'expenses'});
```

Example 6

Map a file region identical to that of the previous example, except repeat the pattern of `int16`, `uint32`, and `single` data types only three times within the mapped region of the file. Allow write access to the file by setting the `Writable` property to `true`:

```
m = memmapfile('records.dat',                    ...
                    'Offset', 2048,              ...
                    'Format', {                  ...
                       'int16'  [2 2] 'model';   ...
                       'uint32' [1 1] 'serialno';...
                       'single' [1 3] 'expenses'},...
                    'Repeat', 3,                 ...
                    'Writable', true);
```

See Also

`disp(memmapfile)`, `get(memmapfile)`

memory

Purpose

Help for memory limitations

Description

If the `out of memory` error message is encountered, there is no more room in memory for new variables. You must free some space before you can proceed. One way to free space is to use the `clear` function to remove some of the variables residing in memory. Another is to issue the `pack` command to compress data in memory. This opens larger contiguous blocks of memory for you to use.

Here are some additional system-specific tips:

Windows: Increase virtual memory by using System in the Control Panel.

UNIX: Ask your system manager to increase your swap space.

See Also

`clear`, `pack`

The Technical Support Guide to Memory Management at `http://www.mathworks.com/support/tech-notes/1100/1106.html`

Purpose

Construct `MException` object

Syntax

```
ME = MException(identifier, msgstring)
ME = MException(identifier, msgformat, s1, s2, ...)
```

Description

`ME = MException(identifier, msgstring)` constructs an object `ME` of class `MException` and assigns an `identifier` and error message `msgstring` to that object. This object then provides properties and methods that you can use in generating or responding to errors in your program code.

The `identifer` input is a message identifier string that you can specify to uniquely identify the `MException`. The `msgstring` input is a character string that informs the user about the cause of the error. MATLAB displays this error message if the program aborts due to the error.

`ME = MException(identifier, msgformat, s1, s2, ...)` constructs an `MException` object where the error message is constructed by the format string `msgformat` and additional string or scalar numeric values `s1`, `s2`, etc. The `msgformat` argument differs from `msgstring` (used in the previous syntax) in that it may contain escape sequences, such as `\t` or `\n`, and C language conversion specifiers, such as `%s` and `%d` that are supported by the `sprintf` function. Additional arguments `s1`, `s2`, etc. provide the values that correspond to these conversion specifiers. See the `sprintf` function reference page for more information on valid conversion specifiers.

There are two ways to generate an error in your MATLAB code. Although the latter method is more work, it can provide you with a more extensible system for reporting and handling errors:

- Call the MATLAB `error` function.
- Construct an `MException` object, store identifying information in the object, and use the `throw` or `throwAsCaller` methods of that object to generate the error.

Properties

The `MException` object has four properties: `identifier`, `message`, `stack`, and `cause`.

Property	Description
`identifier`	Identifies the `MException` string.
`message`	Formatted error message that is displayed.
`stack`	Structure containing stack trace information such as M-file function name and line number where the `MException` was thrown.
`cause`	Cell array of `MException` that caused this exception to be created.

Methods

Method	Description
`AddCause`	Appends an `MException` to the `cause` field of another `MException.`
`eq`	Compares two `MException` objects for equality.
`getReport`	Returns a formatted message string based on the current exception that uses the same format as errors thrown by internal MATLAB code.
`isequal`	Compares two `MException` objects for equality.
`last`	Returns an `MException` object for the most recently thrown exception.
`ne`	Compares two `MException` objects for inequality.
`rethrow`	Reissues an exception that has been caught, causing the program to stop.

Method	Description
`throw`	Issues an exception from the currently running M-file.
`throwAsCaller`	Issues an exception from the currently running M-file, also omitting the current stack frame from the `stack` field of the `MException`.

Remarks

When MATLAB encounters an error in its internal code or in your own program code, it *throws an exception*. In this exception process, MATLAB

- Interrupts the program at the point of the error.
- Constructs an object of the `MException` class.
- Records information about the error in that object.
- Displays this information at the user's terminal.
- Aborts the program.

If your program code implements a `try-catch` mechanism to intercept the error before MATLAB aborts the program, you can obtain access to the `MException` object that MATLAB associates with this error instance via the `catch` statement and then handle the condition based on the records you can retrieve from the object.

Examples

Example 1

If your message string requires formatting specifications, like those available with the `sprintf` function, you can use this syntax for the `MException` constructor:

```
ME = MException(identifier, formatstring, arg1, arg2, ...)
```

For example,

```
S = 'Accounts';  f1 = 'ClientName';
ME = MException('AcctError:Incomplete', ...
      'Field ''%s.%s'' is not defined.', S, f1);

ME.message
ans =
    Field 'Accounts.ClientName' is not defined.
```

Example 2

This example reads the contents of an image file. The attempt to open and then read the file is done in a `try` block. If either the open or read fails, the program catches the resulting exception and saves the `MException` object in the variable `ME1`.

The `catch` block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension. This is done using a `try-catch` statement that is nested within the original `try-catch`.

```
function d_in = read_image(filename)
file_format = regexp(filename, '(?<=\.)\w+$', 'match');

try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch ME1
   % Get last segment of the error message identifier.
   idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

   % Did the read fail because the file could not be found?
   if strcmp(idSegLast, 'InvalidFid') && ...
      ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch file_format
```

```
        case 'jpg'    % Change jpg to jpeg
           filename = regexprep(filename, '(?<=\.)\w+$', 'jpeg');
        case 'jpeg'   % Change jpeg to jpg
           filename = regexprep(filename, '(?<=\.)\w+$', 'jpg');
        otherwise
           disp(sprintf('File %s not found', filename));
           rethrow(ME1);
        end

        % Try again, with modifed filenames.
        try
           fid = fopen(filename, 'r');
           d_in = fread(fid);
        catch ME2
           disp(sprintf('Unable to access file %s', filename));
           ME2 = addCause(ME2, ME1);
           rethrow(ME2)
        end
     end
  end
```

Example 3

This example attempts to open a file in a directory that is not on the MATLAB path. It uses a nested `try-catch` block to give the user the opportunity to extend the path. If the still cannot be found, the program issues an exception with the first error appended to the second:

```
function data = read_it(filename);
try
   fid = fopen(filename, 'r');
   data = fread(fid);
catch ME1
   if strcmp(ME1.identifier, 'MATLAB:FileIO:InvalidFid')
      msg = sprintf('\n%s%s%s', 'Cannot open file ', ...
         filename, '. Try another location?  ');
      reply = input(msg, 's')
      if reply(1) == 'y'
```

```
            newdir = input('Enter directory name:  ', 's');
        else
            throw(ME1);
        end
        addpath(newdir);
        try
           fid = fopen(filename, 'r');
           data = fread(fid);
        c ME2
           ME3 = addCause(ME2, ME1)
           throw(ME3);
        end
        rmpath(newdir);
    end
end
fclose(fid);
```

If you run this function in a `try-catch` block at the command line, you can look at the `MException` object by assigning it to a variable (`e`) with the `catch` command.

```
try
   d = read_it('anytextfile.txt');
catch e
end

e
e =
 MException object with properties:

    identifier: 'MATLAB:FileIO:InvalidFid'
       message: 'Invalid file identifier.  Use fopen to
generate a valid file identifier.'
         stack: [1x1 struct]
         cause: {[1x1 MException]}

  Cannot open file anytextfile.txt. Try another location?y
```

```
Enter directory name:  xxxxxxx
Warning: Name is nonexistent or not a directory: xxxxxxx.
> In path at 110
  In addpath at 89
```

See Also

throw(MException), rethrow(MException), throwAsCaller(MException), addCause(MException), getReport(MException), disp(MException), isequal(MException), eq(MException), ne(MException), last(MException), error, try, catch

menu

Purpose

Generate menu of choices for user input

Syntax

```
k = menu('mtitle','opt1','opt2',...,'optn')
```

Description

`k = menu('mtitle','opt1','opt2',...,'optn')` displays the menu whose title is in the string variable '*mtitle*' and whose choices are string variables '*opt1*', '*opt2*', and so on. `menu` returns thenumber of the selected menu item.

If the user's terminal provides a graphics capability, `menu` displays the menu items as push buttons in a figure window (Example 1), otherwise they will be given as a numbered list in the command window (Example 2).

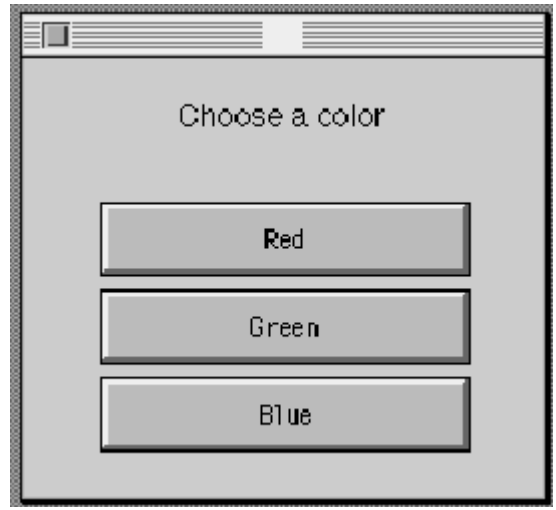
Remarks

To call `menu` from another ui object, set that object's `Interruptible` property to `'yes'`. For more information, see the MATLAB Graphics documentation.

Examples

Example 1

`k = menu('Choose a color','Red','Green','Blue')` displays



After input is accepted, use `k` to control the color of a graph.

```
color = ['r','g','b']
plot(t,s,color(k))
```

Example 2

```
K = menu('Choose a color','Red','Blue','Green')
```

displays on the Command Window

```
----- Choose a color -----
1) Red
2) Blue
3) Green
Select a menu number:
```

The number entered by the user in response to the prompt is returned as `K` (i.e. `K = 2` implies that the user selected `Blue`).

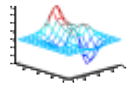
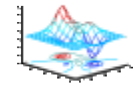
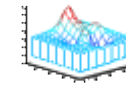
See Also

`guide`, `input`, `uicontrol`, `uimenu`

mesh, meshc, meshz

Purpose

Mesh plots

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
mesh(...,'PropertyName',PropertyValue,...)
mesh(axes_handles,...)
meshc(...)
meshz(...)
h = mesh(...)
hsurface = mesh('v6',...) hsurface = meshc('v6',...),
```

Description

`mesh`, `meshc`, and `meshz` create wireframe parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `C`.

`mesh(X,Y,Z)` draws a wireframe mesh with color determined by `Z` so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, $(X(j), Y(i), Z(i,j))$ are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, $(X(i,j), Y(i,j), Z(i,j))$ are the intersections of the wireframe grid lines.

`mesh(Z)` draws a wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`mesh(...,C)` draws a wireframe mesh with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`mesh(...,'PropertyName',PropertyValue,...)` sets the value of the specified surface property. Multiple property values can be set with a single statement.

`mesh(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`meshc(...)` draws a contour plot beneath the mesh.

`meshz(...)` draws a curtain plot (i.e., a reference plane) around the mesh.

`h = mesh(...)`, h = meshc(...), and h = meshz(...) return a handle to a surfaceplot graphics object.

Backward-Compatible Version

`hsurface = mesh('v6',...)` `hsurface = meshc('v6',...)`, and `hsurface` = meshc('v6',...) returns the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

`mesh`, `meshc`, and `meshz` do not accept complex inputs.

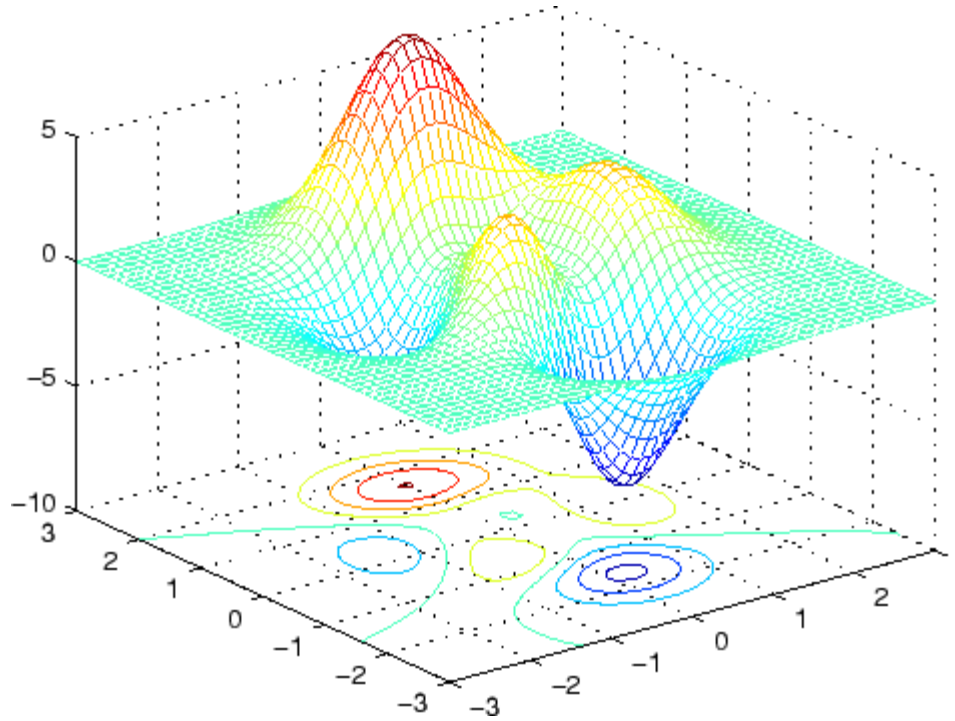
A mesh is drawn as a surface graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or `none` when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the

simulation of hidden-surface elimination in the mesh, and the `shading` command controls the shading model.

Examples

Produce a combination mesh and contour plot of the `peaks` surface:

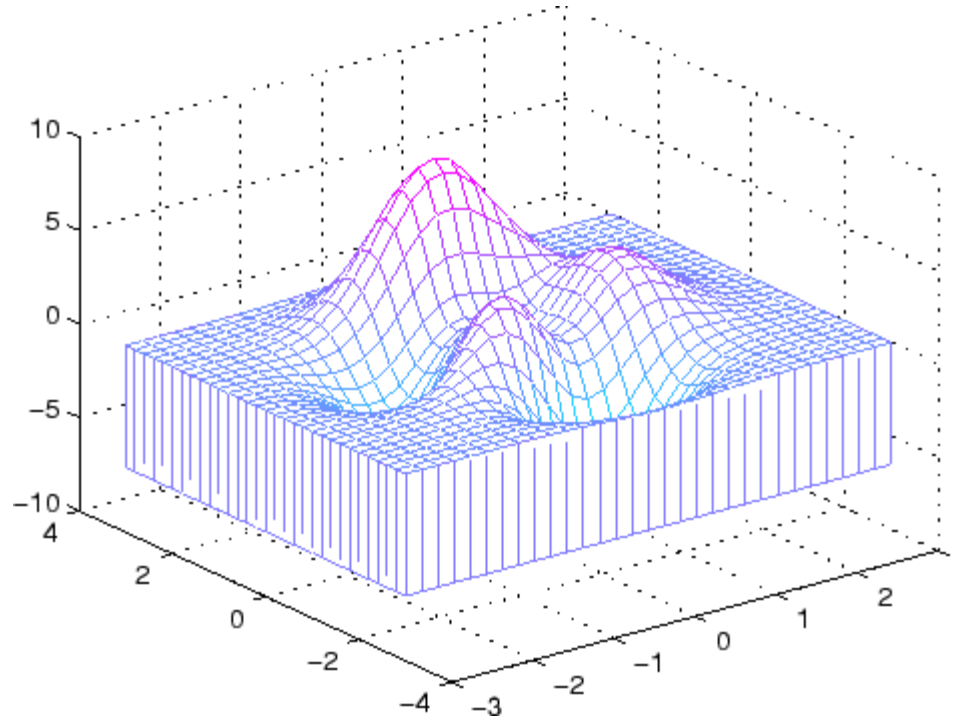
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(X,Y,Z);
axis([-3 3 -3 3 -10 5])
```



Generate the curtain plot for the `peaks` function:

```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
```

```
meshz(X,Y,Z)
```



Algorithm

The range of `X`, `Y`, and `Z`, or the current settings of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties, determine the axis limits. `axis` sets these properties.

The range of `C`, or the current settings of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function), determine the color scaling. The scaled color values are used as indices into the current colormap.

The mesh rendering functions produce color values by mapping the *z* data values (or an explicit color array) onto the current colormap. The MATLAB default behavior is to compute the color limits automatically using the minimum and maximum data values (also set using `caxis`

auto). The minimum data value maps to the first color value in the colormap and the maximum data value maps to the last color value in the colormap. MATLAB performs a linear transformation on the intermediate values to map them to the current colormap.

meshc calls mesh, turns hold on, and then calls contour and positions the contour on the *x-y* plane. For additional control over the appearance of the contours, you can issue these commands directly. You can combine other types of graphs in this manner, for example surf and pcolor plots.

meshc assumes that X and Y are monotonically increasing. If X or Y is irregularly spaced, contour3 calculates contours using a regularly spaced contour grid, then transforms the data to X or Y.

See Also

contour, hidden, meshgrid, surface, surf, surfc, surfl, waterfall

"Creating Surfaces and Meshes" on page 1-97 for related functions

Surfaceplot Properties for a list of surfaceplot properties

The functions axis, caxis, colormap, hold, shading, and view all set graphics object properties that affect mesh, meshc, and meshz.

For a discussion of parametric surfaces plots, refer to surf.

Purpose

Generate X and Y arrays for 3-D plots

Syntax

```
[X,Y] = meshgrid(x,y)
[X,Y] = meshgrid(x)
[X,Y,Z] = meshgrid(x,y,z)
```

Description

`[X,Y] = meshgrid(x,y)` transforms the domain specified by vectors `x` and `y` into arrays `X` and `Y`, which can be used to evaluate functions of two variables and three-dimensional mesh/surface plots. The rows of the output array `X` are copies of the vector `x`; columns of the output array `Y` are copies of the vector `y`.

`[X,Y] = meshgrid(x)` is the same as `[X,Y] = meshgrid(x,x)`.

`[X,Y,Z] = meshgrid(x,y,z)` produces three-dimensional arrays used to evaluate functions of three variables and three-dimensional volumetric plots.

Remarks

The `meshgrid` function is similar to `ndgrid` except that the order of the first two input and output arguments is switched. That is, the statement

```
[X,Y,Z] = meshgrid(x,y,z)
```

produces the same result as

```
[Y,X,Z] = ndgrid(y,x,z)
```

Because of this, `meshgrid` is better suited to problems in two- or three-dimensional Cartesian space, while `ndgrid` is better suited to multidimensional problems that aren't spatially based.

`meshgrid` is limited to two- or three-dimensional Cartesian space.

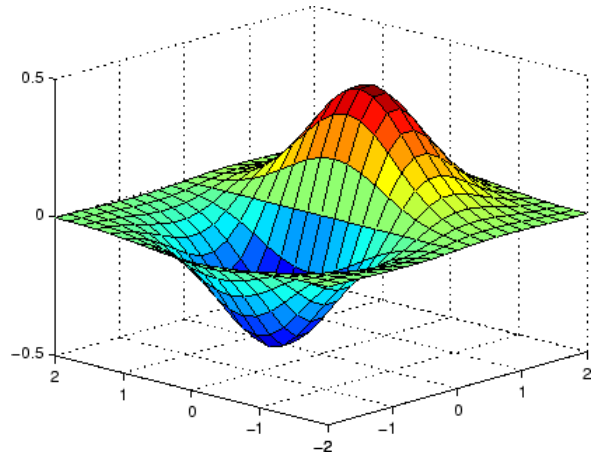
Examples

```
[X,Y] = meshgrid(1:3,10:14)

X =

     1     2     3
     1     2     3
```

```
     1     2     3
     1     2     3
     1     2     3

Y =

    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14
```

The following example shows how to use meshgrid to create a surface plot of a function.

```
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
```



See Also

`griddata`, `mesh`, `ndgrid`, `slice`, `surf`

Purpose

Information on class methods

Syntax

```
m = methods('classname')
m = methods('object')
m = methods(..., '-full')
```

Description

`m = methods('classname')` returns, in a cell array of strings, the names of all methods for the MATLAB, COM, or Java class `classname`.

`m = methods('object')` returns the names of all methods for the MATLAB, COM, or Java class of which `object` is an instance.

`m = methods(..., '-full')` returns the full description of the methods defined for the class, including inheritance information and, for COM and Java methods, attributes and signatures. For any overloaded method, the returned array includes a description of each of its signatures.

For MATLAB classes, inheritance information is returned only if that class has been instantiated.

For some classes it may not be possible for MATLAB to know inherited methods until after the class has been instantiated. In these cases, `methods -full` displays only the methods defined by the class itself until after the class has been instantiated. After an instance has been created `methods -full` also shows inherited methods.

Examples

List the methods of MATLAB class `stock`:

```
m = methods('stock')
m =
    'display'
    'get'
    'set'
    'stock'
    'subsasgn'
    'subsref'
```

Create a MathWorks sample COM control and list its methods:

```
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
methods(h)

Methods for class com.mwsamp.mwsampctrl.1:

AboutBox          GetR8Array        SetR8             move
Beep              GetR8Vector       SetR8Array        propedit
FireClickEvent    GetVariantArray   SetR8Vector       release
GetBSTR           GetVariantVector  addproperty       save
GetBSTRArray      Redraw            delete            send
GetI4             SetBSTR           deleteproperty    set
GetI4Array        SetBSTRArray      events
GetI4Vector       SetI4             get
GetIDispatch      SetI4Array        invoke
GetR8             SetI4Vector       load
```

Display a full description of all methods on Java object `java.awt.Dimension`:

```
methods java.awt.Dimension -full

Dimension(java.awt.Dimension)
Dimension(int,int)
Dimension()
void wait() throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
java.lang.Class getClass()  % Inherited from java.lang.Object
            .
            .
```

See Also

methodsview, invoke, ismethod, help, what, which

Purpose

Information on class methods in separate window

Syntax

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

Description

`methodsview packagename.classname` displays information describing the Java class `classname` that is available from the package of Java classes `packagename`.

`methodsview classname` displays information describing the MATLAB, COM, or imported Java class `classname`.

`methodsview(object)` displays information describing the `object` instantiated from a COM or Java class.

MATLAB creates a new window in response to the `methodsview` command. This window displays all the methods defined in the specified class. For each of these methods, the following additional information is supplied:

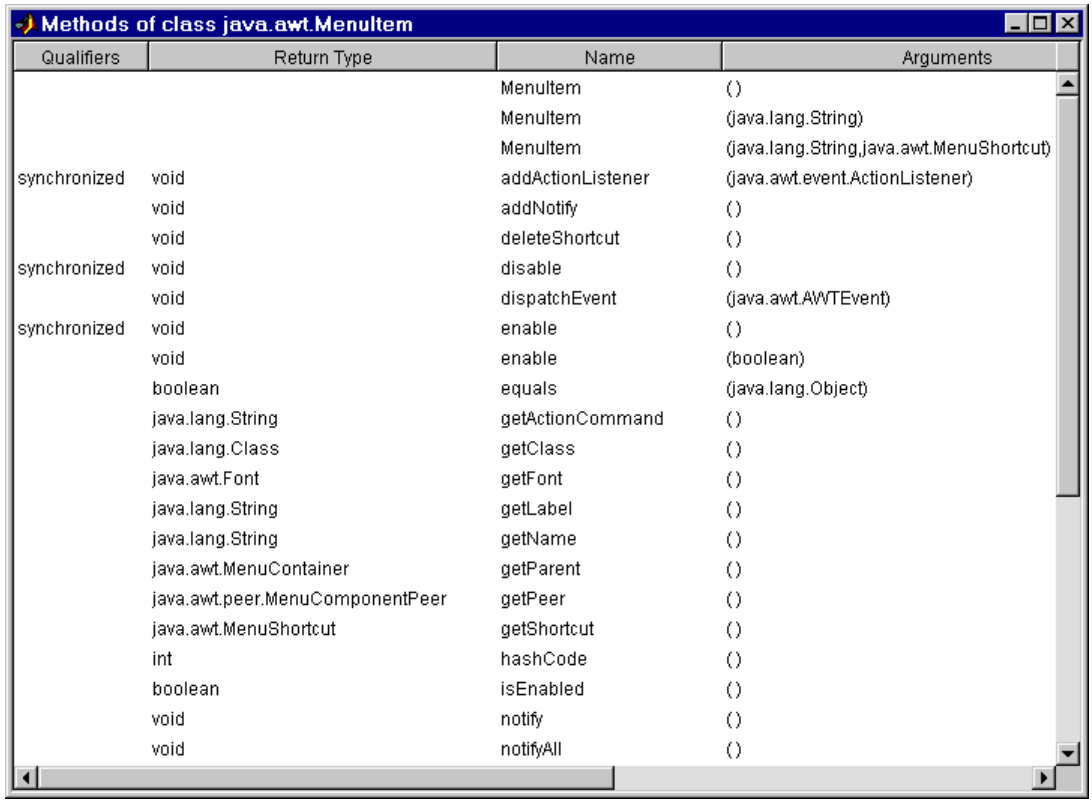
- Name of the method
- Method type qualifiers (for example, `abstract` or `synchronized`)
- Data type returned by the method
- Arguments passed to the method
- Possible exceptions thrown
- Parent of the specified class

Examples

The following command lists information on all methods in the `java.awt.MenuItem` class.

```
methodsview java.awt.MenuItem
```

MATLAB displays this information in a new window, as shown below



Qualifiers	Return Type	Name	Arguments
		MenuItem	()
		MenuItem	(java.lang.String)
		MenuItem	(java.lang.String,java.awt.MenuShortcut)
synchronized	void	addActionListener	(java.awt.event.ActionListener)
	void	addNotify	()
	void	deleteShortcut	()
synchronized	void	disable	()
	void	dispatchEvent	(java.awt.AWTEvent)
synchronized	void	enable	()
	void	enable	(boolean)
	boolean	equals	(java.lang.Object)
	java.lang.String	getActionCommand	()
	java.lang.Class	getClass	()
	java.awt.Font	getFont	()
	java.lang.String	getLabel	()
	java.lang.String	getName	()
	java.awt.MenuContainer	getParent	()
	java.awt.peer.MenuComponentPeer	getPeer	()
	java.awt.MenuShortcut	getShortcut	()
	int	hashCode	()
	boolean	isEnabled	()
	void	notify	()
	void	notifyAll	()

See Also

`methods`, `import`, `class`, `javaArray`

Purpose

Compile MEX-function from C, C++, or Fortran source code

Syntax

```
mex -help
mex -setup
mex filenames
mex options filenames
```

Description

`mex -help` displays the M-file help for `mex`.

`mex -setup` lets you select or change the default compiler.

`mex filenames` compiles and links one or more C, C++, or Fortran source files specified in `filenames` into a shared library called a MEX-file executable from MATLAB.

`mex options filenames` compiles and links one or more source files specified in `filenames` using one or more of the specified command-line `options`.

The MEX-file has a platform-dependent extension. Use the `mexext` function to return the extension for the current machine or for all supported platforms.

`filenames` can be any combination of source files, object files, and library files. Include both the file name and the file extension in `filenames`. A non-source-code `filenames` parameter is passed to the linker without being compiled.

All valid command-line `options` are shown in the MEX Script Switches on page 2-2152 table. These options are available on all platforms except where noted.

`mex` also can build executable files for stand-alone MATLAB engine and MAT-file applications. For more information, see "Engine/MAT Stand-Alone Application Details" on page 2-2157.

You can run `mex` from the MATLAB Command Prompt, Windows Command Prompt, or the UNIX shell. `mex` is a script named `mex.bat` on Windows and `mex` on UNIX. It is located in the *`matlabroot`*`/bin` directory.

The first file listed in `filenames` becomes the name of the resulting MEX-file. You can list other source, object, or library files as additional `filenames` parameters to satisfy external references.

`mex` uses an options file to specify variables and values that are passed as arguments to the compiler, linker, and other tools (e.g., the resource linker on Windows). Command-line options to `mex` may supplement or override contents of the options file. For more information, see "Options File Details" on page 2-2156. The default name for the options file is `mexopts.bat` (Windows) or `mexopts.sh` (UNIX).

The `setup` option causes `mex` to search for installed compilers and allows you to choose an options file as the default for future invocations of `mex`.

For a list of compilers supported with this release, refer to Technical Note 1601 at `http://www.mathworks.com/support/tech-notes/1600/1601.html`.

MEX Script Switches

Switch	Function
`@<rsp_file>`	(Windows only) Include the contents of the text file `<rsp_file>` as command-line arguments to `mex`.
`-<arch>`	Build an output file for architecture `<arch>`. To determine the value for `<arch>`, type `computer('arch')` at the MATLAB Command Prompt on the target machine. Valid values for `<arch>` depend on the architecture of the build platform.

MEX Script Switches (Continued)

Switch	Function
`-ada <sfcn.ads>`	Use this option to compile a Simulink® S-function written in Ada, where `<sfcn.ads>` is the Package Specification for the S-function. When this option is specified, only the `-v` (verbose) and `-g` (debug) options are relevant. All other options are ignored. For examples and information on supported compilers and other requirements, see `README` in the `simulink/ada/examples` directory.
`-argcheck`	(C functions only) Add argument checking. This adds code so arguments passed incorrectly to MATLAB API functions cause assertion failures.
`-c`	Compile only. Creates an object file, but not a MEX-file.
`-compatibleArrayDims`	Build a MEX-file using the MATLAB Version 7.2 array-handling API, which limits arrays to 2^31-1 elements. This option is the default. (See also the `-largeArrayDims` option.)
`-cxx`	(UNIX only) Use the C++ linker to link the MEX-file if the first source file is in C and there are one or more C++ source or object files. This option overrides the assumption that the first source file in the list determines which linker to use.
`-D<name>`	Define a symbol name to the C preprocessor. Equivalent to a `#define <name>` directive in the source.

MEX Script Switches (Continued)

Switch	Function
`-D<name>=<value>`	Define a symbol name and value to the C preprocessor. Equivalent to a `#define <name> <value>` directive in the source.
`-f <optionsfile>`	Specify location and name of options file to use. Overrides the `mex` default-options-file search mechanism.
`-fortran`	(UNIX only) Specify that the gateway routine is in Fortran. This option overrides the assumption that the first source file in the list determines which linker to use.
`-g`	Create a MEX-file containing additional symbolic information for use in debugging. This option disables the `mex` default behavior of optimizing built object code (see the `-O` option).
`-h[elp]`	Print help for `mex`.
`-I<pathname>`	Add `<pathname>` to the list of directories to search for `#include` files.
`-inline`	Inline matrix accessor functions (`mx*`). The generated MEX-function may not be compatible with future versions of MATLAB.
`-l<name>`	Link with object library. On Windows, `<name>` expands to `<name>.lib` or `lib<name>.lib` and on UNIX to `lib<name>.so` or `lib<name>.dylib`.

MEX Script Switches (Continued)

Switch	Function
`-L<directory>`	Add `<directory>` to the list of directories to search for libraries specified with the `-l` option. On UNIX systems, you must also set the run-time library path, as explained in "Setting Run-Time Library Path".
`-largeArrayDims`	Build a MEX-file using the MATLAB large-array-handling API. This API can handle arrays with more than 2^31–1 elements when compiled on 64-bit platforms. (See also the `-compatibleArrayDims` option.)
`-n`	No execute mode. Print any commands that `mex` would otherwise have executed, but do not actually execute any of them.
`-O`	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the `-g` option appears without the `-O` option, optimization is disabled.
`-outdir <dirname>`	Place all output files in directory `<dirname>`.
`-output <resultname>`	Create MEX-file named `<resultname>`. The appropriate MEX-file extension is automatically appended. Overrides the default MEX-file naming mechanism.
`-setup`	Interactively specify the compiler options file to use as the default for future invocations of `mex` by placing it in the user profile directory (returned by the `prefdir` command). When this option is specified, no other command-line input is accepted.

MEX Script Switches (Continued)

Switch	Function
`-U<name>`	Remove any initial definition of the C preprocessor symbol `<name>`. (Inverse of the `-D` option.)
`-v`	Verbose mode. Print the values for important internal variables after the options file is processed and all command-line arguments are considered. Prints each compile step and final link step fully evaluated.
`<name>=<value>`	Supplement or override an options file variable for variable `<name>`. This option is processed after the options file is processed and all command line arguments are considered.

Remarks

Options File Details

MATLAB provides template options files for the compilers that are supported by `mex`. These templates are located in the *`matlabroot`*`\bin\win32\mexopts` or the *`matlabroot`*`\bin\win64\mexopts` directories on Windows, or the *`matlabroot`*`/bin` directory on UNIX. These template options files are used by the `-setup` option to define the selected default options file.

Override Option Details

Any variable specified in the options file can be overridden at the command line by using the `<name>=<value>` command-line argument. When using this command-line option, you may need to use the shell's quoting syntax to protect characters such as spaces, which have a meaning in the shell syntax. On Windows, use double quotes (e.g., `COMPFLAGS="opt1 opt2"`) and on UNIX, use single quotes (e.g., `CFLAGS='opt1 opt2'`).

It is common to use this option to supplement variables already defined. To do this refer to the variable by prepending a `$` (e.g., `COMPFLAGS="$COMPFLAGS opt2"` on Windows or `CFLAGS='$CFLAGS opt2'` on UNIX).

Engine/MAT Stand-Alone Application Details

`mex` can build executable files for stand-alone MATLAB engine and MAT-file applications. For these applications, `mex` does not use the default options file; you must use the `-f` option to specify an options file.

The options files used to generate stand-alone MATLAB engine and MAT-file executables are named `*engmatopts.bat` on Windows, or `engopts.sh` and `matopts.sh` on UNIX, and are located in the same directory as the template options files referred to above in Options File Details.

Examples

The following command compiles `yprime.c`:

```
mex yprime.c
```

When debugging, it is often useful to use verbose mode, as well as include symbolic debugging information:

```
mex -v -g yprime.c
```

See Also

`computer`, `dbmex`, `inmem`, `loadlibrary`, `mexext`, `pcode`, `prefdir`, `system`

Purpose

MEX-filename extension

Syntax

```
ext = mexext
extlist = mexext('all')
```

Description

`ext = mexext` returns the filename extension for the current platform.

`extlist = mexext('all')` returns a struct with fields `arch` and `ext` describing MEX-file name extensions for the all platforms.

Remarks

See Using MEX-Files for a table of file extensions.

Examples

Find the MEX-file extension for the system you are currently working on:

```
ext = mexext

ext =
   mexw32
```

Find the MEX-file extension for a PowerPC Macintosh system:

```
extlist = mexext('all');

for k=1:length(extlist)
   if strcmp(extlist(k).arch, 'mac')
   disp(sprintf('Arch: %s      Ext: %s', ...
            extlist(k).arch, extlist(k).ext))
end, end

Arch: mac       Ext: mexmac
```

See Also

`mex`

Purpose

Name of currently running M-file

Syntax

```
mfilename
p = mfilename('fullpath')
c = mfilename('class')
```

Description

`mfilename` returns a string containing the name of the most recently invoked M-file. When called from within an M-file, it returns the name of that M-file, allowing an M-file to determine its name, even if the filename has been changed.

`p = mfilename('fullpath')` returns the full path and name of the M-file in which the call occurs, not including the filename extension.

`c = mfilename('class')` in a method, returns the class of the method, not including the leading `@` sign. If called from a nonmethod, it yields the empty string.

Remarks

If `mfilename` is called with any argument other than the above two, it behaves as if it were called with no argument.

When called from the command line, `mfilename` returns an empty string.

To get the names of the callers of an M-file, use `dbstack` with an output argument.

See Also

`dbstack`, `function`, `nargin`, `nargout`, `inputname`

Purpose

Download file from FTP server

Syntax

```
mget(f,'filename')
mget(f,'dirname')
mget(...,'target')
```

Description

`mget(f,'filename')` retrieves `filename` from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`.

`mget(f,'dirname')` retrieves the directory `dirname` and its contents from the FTP server `f` into the MATLAB current directory, where `f` was created using `ftp`. You can use a `wildcard` (`*`) in `dirname`.

`mget(...,'target')` retrieves the specified items from the FTP server `f`, where `f` was created using `ftp`, into the local directory specified by `target`, where `target` is an absolute pathname.

Examples

Connect to an FTP server, change to the `documents/rfc` directory, and retrieve the file `rfc0959.txt` into the current MATLAB directory.

```
ftpobj = ftp('nic.merit.edu');
cd(ftpobj, 'documents/rfc');

mget(ftpobj, 'rfc0959.txt')
ans =
    'C:\work\rfc0959.txt'
```

See Also

`cd (ftp)`, `ftp`, `mput`

Purpose

Smallest elements in array

Syntax

```
C = min(A)
C = min(A,B)
C = min(A,[],dim)
[C,I] = min(...)
```

Description

`C = min(A)` returns the smallest elements along different dimensions of an array.

If `A` is a vector, `min(A)` returns the smallest element in `A`.

If `A` is a matrix, `min(A)` treats the columns of `A` as vectors, returning a row vector containing the minimum element from each column.

If `A` is a multidimensional array, `min` operates along the first nonsingleton dimension.

`C = min(A,B)` returns an array the same size as `A` and `B` with the smallest elements taken from `A` or `B`. The dimensions of `A` and `B` must match, or they may be scalar.

`C = min(A,[],dim)` returns the smallest elements along the dimension of `A` specified by scalar `dim`. For example, `min(A,[],1)` produces the minimum values along the first dimension (the rows) of `A`.

`[C,I] = min(...)` finds the indices of the minimum values of `A`, and returns them in output vector `I`. If there are several identical minimum values, the index of the first one found is returned.

Remarks

For complex input `A`, `min` returns the complex number with the largest complex modulus (magnitude), computed with `min(abs(A))`. Then computes the largest phase angle with `min(angle(x))`, if necessary.

The `min` function ignores `NaN`s.

See Also

`max`, `mean`, `median`, `sort`

min (timeseries)

Purpose

Minimum value of `timeseries` data

Syntax

```
ts_min = min(ts)
ts_min = min(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_min = min(ts)` returns the minimum value in the time-series data. When `ts.Data` is a vector, ts_min is the minimum value of `ts.Data` values. When `ts.Data` is a matrix, ts_min is a row vector containing the minimum value of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `min` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_min = min(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example illustrates how to find the minimum values in multivariate time-series data.

1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Find the minimum in each data column for this `timeseries` object.

```
min(count_ts)

ans =

     7     9     7
```

The minimum is found independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `max (timeseries)`, `median (timeseries)`, `mean (timeseries)`, `std (timeseries)`, `timeseries`, `var (timeseries)`

MinimizeCommandWindow

Purpose

Minimize size of server window

Syntax

MATLAB Client

```
h.MinimizeCommandWindow
MinimizeCommandWindow(h)
invoke(h, 'MinimizeCommandWindow')
```

Method Signature

```
HRESULT MinimizeCommandWindow(void)
```

Visual Basic Client

```
MinimizeCommandWindow
```

Description

`MinimizeCommandWindow` minimizes the window for the server attached to handle `h`, and makes it inactive. If the server window was already in a minimized state to begin with, then `MinimizeCommandWindow` does nothing.

Remarks

Server function names, like `MinimizeCommandWindow`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Examples

Create a COM server and minimize its window. Then maximize the window and make it the currently active window.

MATLAB Client

```
h = actxserver('matlab.application');
h.MinimizeCommandWindow;
% Now return the server window to its former state on
% the desktop and make it the currently active window.
h.MaximizeCommandWindow;
```

Visual Basic .NET Client

Create a COM server and minimize its window.

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

MaximizeCommandWindow

Purpose

Minimum residual method

Syntax

```
x = minres(A,b)
minres(A,b,tol)
minres(A,b,tol,maxit)
minres(A,b,tol,maxit,M)
minres(A,b,tol,maxit,M1,M2)
minres(A,b,tol,maxit,M1,M2,x0)
[x,flag] = minres(A,b,...)
[x,flag,relres] = minres(A,b,...)
[x,flag,relres,iter] = minres(A,b,...)
[x,flag,relres,iter,resvec] = minres(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)
```

Description

`x = minres(A,b)` attempts to find a minimum norm residual solution `x` to the system of linear equations `A*x=b`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `minres` converges, a message to that effect is displayed. If `minres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`minres(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `minres` uses the default, `1e-6`.

`minres(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `minres` uses the default, `min(n,20)`.

`minres(A,b,tol,maxit,M)` and `minres(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = inv(sqrt(M))*y`. If `M` is `[]` then `minres` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x)` returns `M\x`.

`minres(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `minres` uses the default, an all-zero vector.

`[x,flag] = minres(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`minres` converged to the desired tolerance `tol` within `maxit` iterations.
1	`minres` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`minres` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `minres` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = minres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = minres(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = minres(A,b,...)` also returns a vector of estimates of the `minres` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)` also returns a vector of estimates of the Conjugate Gradients residual norms at each iteration.

Examples

Example 1

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M1 = spdiags(4*on,0,n,n);

x = minres(A,b,tol,maxit,M1);
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_minres` that

- Calls `minres` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_minres` are available to `afun`.

The following shows the code for `run_minres`:

```
function x1 = run_minres
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M = spdiags(4*on,0,n,n);
x1 = minres(@afun,b,tol,maxit,M);
```

```
        function y = afun(x)
           y = 4 * x;
           y(2:n) = y(2:n) - 2 * x(1:n-1);
           y(1:n-1) = y(1:n-1) - 2 * x(2:n);
        end
end
```

When you enter

```
x1=run_minres;
```

MATLAB displays the message

```
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Example 3

Use a symmetric indefinite matrix that fails with pcg.

```
A = diag([20:-1:1, -1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
```

displays the following message:

```
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, minres can handle the indefinite matrix A.

```
x = minres(A,b,1e-6,40);
minres converged at iteration 39 to a solution with relative
residual 1.3e-007
```

See Also

bicg, bicgstab, cgs, cholinc, gmres, lsqr, pcg, qmr, symmlq

`function_handle` (`@`), `mldivide` (`\`)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

Purpose

Determine whether M-file or MEX-file cannot be cleared from memory

Syntax

```
mislocked
mislocked(fun)
```

Description

`mislocked` by itself returns logical `1` (`true`) if the currently running M-file or MEX-file is locked, and logical `0` (`false`) otherwise.

`mislocked(fun)` returns logical `1` (`true`) if the function named *`fun`* is locked in memory, and logical `0` (`false`) otherwise. Locked M-files and MEX-files cannot be removed with the `clear` function.

See Also

`mlock`, `munlock`

mkdir

Purpose

Make new directory

Graphical Interface

As an alternative to the `mkdir` function, you can click the **New folder** button  in the "Current Directory Browser" to add a directory.

Syntax

```
mkdir('dirname')
mkdir('parentdir','dirname')
status = mkdir(...,'dirname')
[status,message,messageid] = mkdir(...,'dirname')
```

Description

`mkdir('dirname')` creates the directory `dirname` in the current directory, if `dirname` represents a relative path. Otherwise, `dirname` represents an absolute path and `mkdir` attempts to create the absolute directory `dirname` in the root of the current volume. An absolute path starts with any one of the following: a Windows drive letter, a UNC path `'\\'` string, or a UNIX `'/'` character.

`mkdir('parentdir','dirname')` creates the directory `dirname` in the existing directory `parentdir`, where `parentdir` is an absolute or relative pathname. If `parentdir` does not exist, MATLAB attempts to create it. See the Remarks section below.

`status = mkdir(...,'dirname')` creates the specified directory and returns a `status` of logical 1 if the operation was successful, or logical 0 if unsuccessful.

`[status,message,messageid] = mkdir(...,'dirname')` creates the specified directory, and returns `status`, message string, and MATLAB error message ID. The value given to `status` is logical `1` for success and logical `0` for error.

See the help for `error` and `lasterror` for more information.)

Remarks

If the `dirname` or `parentdir` argument specifies not only a directory name, but also a directory path (e.g., `'mydir\xdir1\xdir2\targetdir'`), and this path includes one or more nonexistent directories (e.g., `xdir1` and/or `xdir2` in the path above), MATLAB attempts to create each

nonexistent parent directory, in turn, in the process of creating the specified target directory.

Examples

Create a Subdirectory in Current Directory

To create a subdirectory in the current directory called `newdir`, type

```
mkdir('newdir')
```

Create a Subdirectory in Specified Parent Directory

To create a subdirectory called `newdir` in the directory `testdata`, which is at the same level as the current directory, type

```
mkdir('../testdata','newdir')
```

Return Status When Creating Directory

In this example, the first attempt to create `newdir` succeeds, returning a status of `1`, and no error or warning message or message identifier:

```
[s, mess, messid] = mkdir('../testdata', 'newdir')
s =
   1
mess =
   ''
messid =
   ''
```

If you attempt to create the same directory again, `mkdir` again returns a success status, and also a warning and message identifier informing you that the directory already existed:

```
[s,mess,messid] = mkdir('../testdata','newdir')
s =
   1
mess =
   Directory "newdir" already exists.
messid =
   MATLAB:MKDIR:DirectoryExists
```

See Also

copyfile, cd, dir, fileattrib, filebrowser, fileparts, ls, mfilename, movefile, rmdir

Purpose

Create new directory on FTP server

Syntax

```
mkdir(f,'dirname')
```

Description

`mkdir(f,'dirname')` creates the directory `dirname` in the current directory of the FTP server `f`, where `f` was created using `ftp`, and where `dirname` is a pathname relative to the current directory on `f`.

Examples

Connect to server `testsite`, view the contents, and create the directory `newdir` in the directory `testdir`.

```
test=ftp('ftp.testsite.com')
dir(test)
.               ..       otherfile.m          testdir
mkdir(test,'testdir/newdir');
dir(test,'testdir)
.               ..       newdir
```

See Also

`dir (ftp)`, `ftp`, `rmdir (ftp)`

Purpose

Make piecewise polynomial

Syntax

```
pp = mkpp(breaks,coefs)
pp = mkpp(breaks,coefs,d)
```

Description

`pp = mkpp(breaks,coefs)` builds a piecewise polynomial `pp` from its breaks and coefficients. `breaks` is a vector of length `L+1` with strictly increasing elements which represent the start and end of each of `L` intervals. `coefs` is an `L`-by-`k` matrix with each row `coefs(i,:)` containing the coefficients of the terms, from highest to lowest exponent, of the order `k` polynomial on the interval `[breaks(i),breaks(i+1)]`.

`pp = mkpp(breaks,coefs,d)` indicates that the piecewise polynomial `pp` is `d`-vector valued, i.e., the value of each of its coefficients is a vector of length `d`. `breaks` is an increasing vector of length `L+1`. `coefs` is a `d`-by-`L`-by-`k` array with `coefs(r,i,:)` containing the `k` coefficients of the `i`th polynomial piece of the `r`th component of the piecewise polynomial.

Use `ppval` to evaluate the piecewise polynomial at specific points. Use `unmkpp` to extract details of the piecewise polynomial.

Note. The *order* of a polynomial tells you the number of coefficients used in its description. A *k*th order polynomial has the form

$$c_1 x^{k-1} + c_2 x^{k-2} + \ldots + c_{k-1} x + c_k$$

It has *k* coefficients, some of which can be 0, and maximum exponent *k*-1. So the order of a polynomial is usually one greater than its degree. For example, a cubic polynomial is of order 4.

Examples

The first plot shows the quadratic polynomial

$$1 - \left(\frac{x}{2} - 1\right)^2 = \frac{-x^2}{4} + x$$

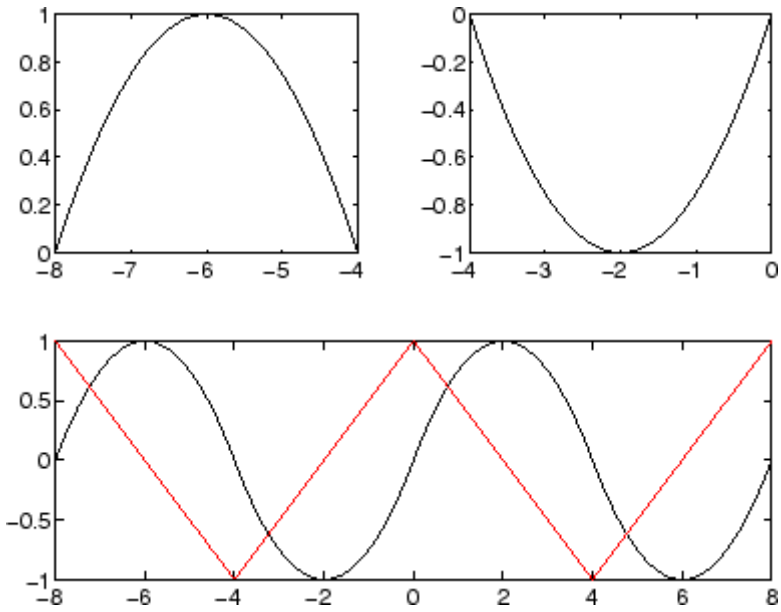
shifted to the interval [-8,-4]. The second plot shows its negative

$$\left(\frac{x}{2}-1\right)^2-1=\frac{x^2}{4}-x$$

but shifted to the interval [-4,0].

The last plot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. It also shows its first derivative, which was constructed after breaking the piecewise polynomial apart using `unmkpp`.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
[breaks,coefs,l,k,d] = unmkpp(pp);
dpp = mkpp(breaks,repmat(k-1:-1:1,d*l,1).*coefs(:,1:k-1),d);
hold on, plot(xx,ppval(dpp,xx),'r-'), hold off
```



See Also

`ppval`, `spline`, `unmkpp`

Purpose

Left or right matrix division

Syntax

```
mldivide(A,B)      A\B
mrdivide(B,A)      B/A
```

Description

`mldivide(A,B)` and the equivalent `A\B` perform matrix left division (back slash). `A` and `B` must be matrices that have the same number of rows, unless `A` is a scalar, in which case `A\B` performs element-wise division — that is, `A\B = A.\B`.

If `A` is a square matrix, `A\B` is roughly the same as `inv(A)*B`, except it is computed in a different way. If `A` is an `n`-by-`n` matrix and `B` is a column vector with `n` elements, or a matrix with several such columns, then `X = A\B` is the solution to the equation $AX = B$ computed by Gaussian elimination with partial pivoting (see "Algorithm" on page 2-2183 for details). A warning message is displayed if `A` is badly scaled or nearly singular.

If `A` is an `m`-by-`n` matrix with `m ~= n` and `B` is a column vector with `m` components, or a matrix with several such columns, then `X = A\B` is the solution in the least squares sense to the under- or overdetermined system of equations $AX = B$. In other words, `X` minimizes `norm(A*X - B)`, the length of the vector $AX - B$. The rank `k` of `A` is determined from the QR decomposition with column pivoting (see "Algorithm" on page 2-2183 for details). The computed solution `X` has at most `k` nonzero elements per column. If `k < n`, this is usually not the same solution as `x = pinv(A)*B`, which returns a least squares solution.

`mrdivide(B,A)` and the equivalent `B/A` perform matrix right division (forward slash). `B` and `A` must have the same number of columns.

If `A` is a square matrix, `B/A` is roughly the same as `B*inv(A)`. If `A` is an `n`-by-`n` matrix and `B` is a row vector with `n` elements, or a matrix with several such rows, then `X = B/A` is the solution to the equation $XA = B$ computed by Gaussian elimination with partial pivoting. A warning message is displayed if `A` is badly scaled or nearly singular.

If `B` is an `m`-by-`n` matrix with `m ~= n` and `A` is a column vector with `m` components, or a matrix with several such columns, then `X = B/A` is

the solution in the least squares sense to the under- or overdetermined system of equations $XA = B$.

Note Matrix right division and matrix left division are related by the equation `B/A = (A'\B')'`.

Least Squares Solutions

If the equation $Ax = b$ does not have a solution (and A is not a square matrix), `x = A\b` returns a *least squares solution* — in other words, a solution that minimizes the length of the vector $Ax - b$, which is equal to `norm(A*x - b)`. See "Example 3" on page 2-2182 for an example of this.

Examples

Example 1

Suppose that `A` and `b` are the following.

```
A = magic(3)

A =

     8     1     6
     3     5     7
     4     9     2

b = [1;2;3]

b =

     1
     2
     3
```

To solve the matrix equation $Ax = b$, enter

```
x=A\b
```

```
x =

    0.0500
    0.3000
    0.0500
```

You can verify that `x` is the solution to the equation as follows.

```
A*x

ans =

    1.0000
    2.0000
    3.0000
```

Example 2 — A Singular

If `A` is singular, `A\b` returns the following warning.

```
Warning: Matrix is singular to working precision.
```

In this case, $Ax = b$ might not have a solution. For example,

```
A = magic(5);
A(:,1) = zeros(1,5); % Set column 1 of A to zeros
b = [1;2;5;7;7];
x = A\b
Warning: Matrix is singular to working precision.

ans =

   NaN
   NaN
   NaN
   NaN
   NaN
```

If you get this warning, you can still attempt to solve $Ax = b$ using the pseudoinverse function `pinv`.

```
x = pinv(A)*b

x =

         0
    0.0209
    0.2717
    0.0808
   -0.0321
```

The result `x` is least squares solution to $Ax = b$. To determine whether `x` is a exact solution — that is, a solution for which $Ax - b = 0$ — simply compute

```
A*x-b

ans =

   -0.0603
    0.6246
   -0.4320
    0.0141
    0.0415
```

The answer is not the zero vector, so `x` is not an exact solution.

"Pseudoinverses", in the online MATLAB Mathematics documentation, provides more examples of solving linear systems using `pinv`.

Example 3

Suppose that

```
A = [1 0 0;1 0 0];
b = [1; 2];
```

Note that $Ax = b$ cannot have a solution, because `A*x` has equal entries for any `x`. Entering

```
x = A\b
```

returns the least squares solution

```
x =

    1.5000
         0
         0
```

along with a warning that `A` is rank deficient. Note that `x` is not an exact solution:

```
A*x-b

ans =

    0.5000
   -0.5000
```

Data Type Support

When computing `X = A\B` or `X = A/B`, the matrices `A` and `B` can have data type `double` or `single`. The following rules determine the data type of the result:

- If both `A` and `B` have type `double`, `X` has type `double`.
- If either `A` or `B` has type single, `X` has type `single`.

Algorithm

The specific algorithm used for solving the simultaneous linear equations denoted by `X = A\B` and `X = B/A` depends upon the structure of the coefficient matrix `A`. To determine the structure of `A` and select the appropriate algorithm, MATLAB follows this precedence:

1 **If A is sparse and diagonal**, `X` is computed by dividing by the diagonal elements of `A`.

2 **If A is sparse, square, and banded**, then banded solvers are used. Band density is (# nonzeros in the band)/(# nonzeros in a full band). Band density = `1.0` if there are no zeros on any of the three diagonals.

- If `A` is real and tridiagonal, i.e., band density = `1.0`, and `B` is real with only one column, `X` is computed quickly using Gaussian elimination without pivoting.
- If the tridiagonal solver detects a need for pivoting, or if `A` or `B` is not real, or if `B` has more than one column, but `A` is banded with band density greater than the `spparms` parameter `'bandden'` (default = `0.5`), then `X` is computed using the Linear Algebra Package (LAPACK) routines in the following table.

	Real	Complex
`A` and `B` `double`	DGBTRF, DGBTRS	ZGBTRF, ZGBTRS
`A` or `B` `single`	SGBTRF, SGBTRS	CGBTRF, CGBTRS

3 **If A is an upper or lower triangular matrix**, then `X` is computed quickly with a backsubstitution algorithm for upper triangular matrices, or a forward substitution algorithm for lower triangular matrices. The check for triangularity is done for full matrices by testing for zero elements and for sparse matrices by accessing the sparse data structure.

If A is a full matrix, computations are performed using the Basic Linear Algebra Subprograms (BLAS) routines in the following table.

	Real	Complex
`A and B double`	DTRSV, DTRSM	ZTRSV, ZTRSM
`A` or `B` `single`	STRSV, STRSM	CTRSV, CTRSM

4 **If A is a permutation of a triangular matrix**, then `X` is computed with a permuted backsubstitution algorithm.

5 **If A is symmetric, or Hermitian, and has real positive diagonal elements**, then a Cholesky factorization is attempted (see `chol`). If `A` is found to be positive definite, the Cholesky factorization attempt is successful and requires less than half the time of a general factorization. Nonpositive definite matrices are usually detected almost immediately, so this check also requires little time.

If successful, the Cholesky factorization for full `A` is

```
A = R'*R
```

where `R` is upper triangular. The solution `X` is computed by solving two triangular systems,

```
X = R\(R'\B)
```

Computations are performed using the LAPACK routines in the following table.

	Real	**Complex**
`A` and `B` `double`	DLANSY, DPOTRF, DPOTRS, DPOCON	ZLANHE, ZPOTRF, ZPOTRS, ZPOCON
`A` or `B` `single`	SLANSY, SPOTRF, SPOTRS, SDPOCON	CLANHE, CPOTRF, CPOTRS, CPOCON

6 If `A` is sparse, then MATLAB uses CHOLMOD to compute `X`. The computations result in

```
P'*A*P = R'*R
```

where `P` is a permutation matrix generated by `amd`, and `R` is an upper triangular matrix. In this case,

```
X = P*(R\(R'\(P'*B)))
```

7 if A is not sparse but is symmetric, and the Cholesky factorization failed, then MATLAB solves the system using a symmetric, indefinite factorization. That is, MATLAB computes the factorization `P'*A*P=L*D*L'`, and computes the solution `X` by `X=P*(L'\(D\(L\(P*B))))`. Computations are performed using the LAPACK routines in the following table:

	Real	Complex
A and B double	DLANSY, DSYTRF, DSYTRS, DSYCON	ZLANHE, ZHETRF, ZHETRS, ZHECON
A or B single	SLANSY, SSYTRF, SSYTRS, SSYCON	CLANHE, CHETRF, CHETRS, CHECON

8 **If A is Hessenberg**, but not sparse, it is reduced to an upper triangular matrix and that system is solved via substitution.

9 **If A is square** and does not satisfy criteria 1 through 6, then a general triangular factorization is computed by Gaussian elimination with partial pivoting (see `lu`). This results in

```
A = L*U
```

where `L` is a permutation of a lower triangular matrix and `U` is an upper triangular matrix. Then `X` is computed by solving two permuted triangular systems.

```
X = U\(L\B)
```

If `A` is not sparse, computations are performed using the LAPACK routines in the following table.

	Real	Complex
A and B double	DLANGE, DGESV, DGECON	ZLANGE, ZGESV, ZGECON
A or B single	SLANGE, SGESV, SGECON	CLANGE, CGESV, CGECON

If A is sparse, then UMFPACK is used to compute X. The computations result in

```
P*(R\A)*Q = L*U
```

where

- P is a row permutation matrix
- R is a diagonal matrix that scales the rows of A
- Q is a column reordering matrix.

Then `X = Q*(U\L\(P*(R\B)))`.

Note The factorization P*(R\A)*Q = L*U differs from the factorization used by the function lu, which does not scale the rows of A.

---

10 **If A is not square**, then Householder reflections are used to compute an orthogonal-triangular factorization.

```
A*P = Q*R
```

where P is a permutation, Q is orthogonal and R is upper triangular (see qr). The least squares solution X is computed with

```
X = P*(R\(Q'*B))
```

If A is sparse, MATLAB computes a least squares solution using the sparse qr factorization of A.

If A is full, MATLAB uses the LAPACK routines listed in the following table to compute these matrix factorizations.

	Real	Complex
A and B double	DGEQP3, DORMQR, DTRTRS	ZGEQP3, ZORMQR, ZTRTRS
A or B single	SGEQP3, SORMQR, STRTRS	CGEQP3, CORMQR, CTRTRS

Note To see information about choice of algorithm and storage allocation for sparse matrices, set the spparms parameter 'spumoni' = 1.

Note mldivide and mrdivide are not implemented for sparse matrices A that are complex but not square.

See Also

Arithmetic Operators, linsolve, ldivide, rdivide

Purpose

Check M-files for possible problems

GUI Alternatives

From the Current Directory browser, select **View > Directory Reports > M-Lint Code Check Report** on the menu bar. See also the automatic "M-Lint Code Analyzer" in the Editor/Debugger.

Syntax

```
mlint('filename')
inform=mlint('filename','-struct')
msg=mlint('filename','-string')
[inform,filepaths]=mlint('filename')
inform=mlint('filename','-id')
inform=mlint('filename','-fullpath')
inform=mlint('filename','-notok')
mlint('filename','-cyc')
%#ok
```

Description

`mlint('filename')` displays M-Lint information about filename, where the information reports potential problems and opportunities for code improvement, referred to as suspicious constructs. The line number in the message is a hyperlink that opens the file in the Editor/Debugger, scrolled to that line. If filename is a cell array, information is displayed for each file. For `mlint(F1,F2,F3,...)`, where each input is a character array, MATLAB displays information about each input filename. You cannot combine cell arrays and character arrays of filenames. Note that the exact text of the `mlint` messages is subject to some change between versions.

`inform=mlint('filename','-struct')` returns the M-Lint information in a structure array whose length is the number of suspicious constructs found. The structure has the following fields:

Field	Description
`line`	Vector of line numbers to which the message refers

Field	Description
`column`	Two-column array of columns to which the message applies, for each line
`message`	Message describing the suspicious construct that M-Lint caught

If multiple filenames are input, or if a cell array is input, `inform` will contain a cell array of structures.

`msg=mlint('filename','-string')` returns the M-Lint information as a string to the variable `msg`. If multiple filenames are input, or if a cell array is input, `msg` will contain a string where each file's information is separated by 10 equal sign characters (=), a space, the filename, a space, and 10 equal sign characters.

If the **`-struct`** or **`-string`** argument is omitted and an output argument is specified, the default behavior is **`-struct`**. If the argument is omitted and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepaths]=mlint('filename')` additionally returns `filepaths`, the absolute paths to the filenames, in the same order as they were input.

`inform=mlint('filename','-id')` requests the message ID from M-Lint, where ID is a string of the form `ABC...`. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=mlint('filename','-fullpath')` assumes that the input filenames are absolute paths, so that M-Lint does not try to locate them.

`inform=mlint('filename','-notok')` runs `mlint` for all lines in `filename`, even those lines that end with the `mlint` suppression syntax, `%#ok`.

`mlint('filename','-cyc')` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there

is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`%#ok` at the end of a line in an M-file causes `mlint` to ignore those lines in the file. MATLAB comments can follow the `%#ok` pragma. `mlint` ignores specified messages `1` through *`n`* when `%#ok<id1,id2,...id`*`n`*`>` appears at the end of the line.

Examples

`lengthofline.m` is an example M-file with code that can be improved. It is found in *`matlabroot`*`/matlab/help/techdoc/matlab_env/examples`.

mlint for a File with No Options

To run `mlint` on the example file, `lengthofline`, run

```
mlint(fullfile(matlabroot,'help','techdoc','matlab_env','examples','lengthofline'))
```

MATLAB displays M-Lint messages for `lengthofline` in the Command Window:

```
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop.Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
```

```
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses
to group, if needed.
```

For details about these messages and how to improve the code, see "Making Changes Based on M-Lint Messages" in the MATLAB Desktop Tools and Development Environment documentation.

mlint with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=mlint('lengthofline','-id')
```

MATLAB returns

```
inform =

14x1 struct array with fields:
    message
    line
    column
    id
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans =

    message: 'The value assigned here to variable 'nothandle' might never be used.'
       line: 22
     column: [1 9]
```

```
           id: 'NASGU'
```

Here, `NASGU` is the ID for the message `'The value assigned here to variable 'nothandle' might never be used.'`.

Ignoring Messages on a Line with mlint

This examples shows how to instruct `mlint` to ignore lines, where these are lines in the example M-file, `lengthofline`:

```
22 nothandle = ~ishandle(hline);
```

The M-Lint message is

```
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
```

To suppress the message, add `%#ok` to the end of line 22 in the M-file:

```
22 nothandle = ~ishandle(hline); %#ok
```

When you run `mlint` for `lengthofline`, no messages are shown for line 22 because it contains the `%#ok` message suppression syntax.

Ignoring Specific Messages with mlint

When you add `%#ok` to a line, it suppresses all `mlint` messages for that line. If there are multiple messages in a line and you want to suppress some but not all of them, or if you want to suppress a specific message but not all messages that might arise in the future due to changes you make, use the `%#ok` syntax in conjunction with message IDs.

Run `mlint` with the `-id` option:

```
mlint('lengthofline','-id')
```

Results displayed to the Command Window show two messages for line 34:

```
L 34 (C 13-16): AGROW: 'data' might be growing inside a loop.
                Consider preallocating for speed.
L 34 (C 24-31): GFLD: Use dynamic fieldnames with structures instead of GETFIELD.
```

```
Type 'doc struct' for more information.
```

To suppress only the first message about `'data'` growing inside a loop, use its message ID, `GFLD`, with the `%#ok` syntax as shown here:

```
data{nd} = getfield(flds,fdata{nd}); %#ok<GFLD>
```

When you run `mlint` for `lengthofline`, only one message now displays for line 34.

To display multiple specific messages for a line, separate message IDs with commas in the `%#ok` syntax:

```
data{nd} = getfield(flds,fdata{nd}); %#ok<GFLD,AGROW>
```

Now when you run `mlint` for `lengthofline`, no messages display for line 34.

Displaying McCabe Complexity with mlint

To display the McCabe complexity of an M-File, run`mlint` with the `-cyc` option, as shown in the following example:

```
mlint('lengthofline.m', '-cyc')
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the M-File messages, as shown here:

```
L 1 (C 14-21): The McCabe complexity of 'lengthofline' is 12.
L 33 (C 18): Use || instead of | as the OR operator in (scalar) conditional statements.
L 34 (C 7): 'f' might be growing inside a loop. Consider preallocating for speed.
L 37 (C 23): Use && instead of & as the AND operator in (scalar) conditional statements.
L 38 (C 10): 'f' might be growing inside a loop. Consider preallocating for speed.
L 39 (C 27): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 42): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 51): Use && instead of & as the AND operator in (scalar) conditional statements.
L 39 (C 66): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 10): 'f' might be growing inside a loop. Consider preallocating for speed.
L 42 (C 10): 'f' might be growing inside a loop. Consider preallocating for speed.
```

See Also

`mlintrpt`, `profile`

mlintrpt

Purpose

Run `mlint` for file or directory, reporting results in browser

GUI Alternatives

From the Current Directory browser, select **View > Directory Reports > M-Lint Code Check Report** on the menu toolbar. See also the automatic "M-Lint Code Analyzer" in the Editor/Debugger.

Syntax

```
mlintrpt
mlintrpt(filename,'file')
mlintrpt(dirname,'dir')
mlintrpt(filename,'file', 'fullpath_to_configname.txt')
mlintrpt(dirname,'dir', 'fullpath_to_configname.txt')
```

Description

`mlintrpt` scans all M-files in the current directory for M-Lint messages and reports the results in a MATLAB Web browser.

`mlintrpt(filename,'file')` scans the M-file `filename` for messages and reports results. You can omit `'file'` in this form of the syntax because it is the default.

`mlintrpt(dirname,'dir')` scans the specified directory. Here, `dirname` can be in the current directory or can be a full pathname.

`mlintrpt(filename,'file', 'fullpath_to_configname.txt')` applies the M-Lint preference settings to enable or suppress messages as specified in the file `configname.txt`; you must specify the full pathname to `configname.txt`. For information about creating a `fullpath_to_configname.txt` file, select **File > Preferences > M-Lint**, and click **Help**.

`mlintrpt(dirname,'dir', 'fullpath_to_configname.txt')` applies the M-Lint preference settings specified in the file `fullpath_to_configname.txt`; you must specify the full pathname to `configname.txt`.

Examples

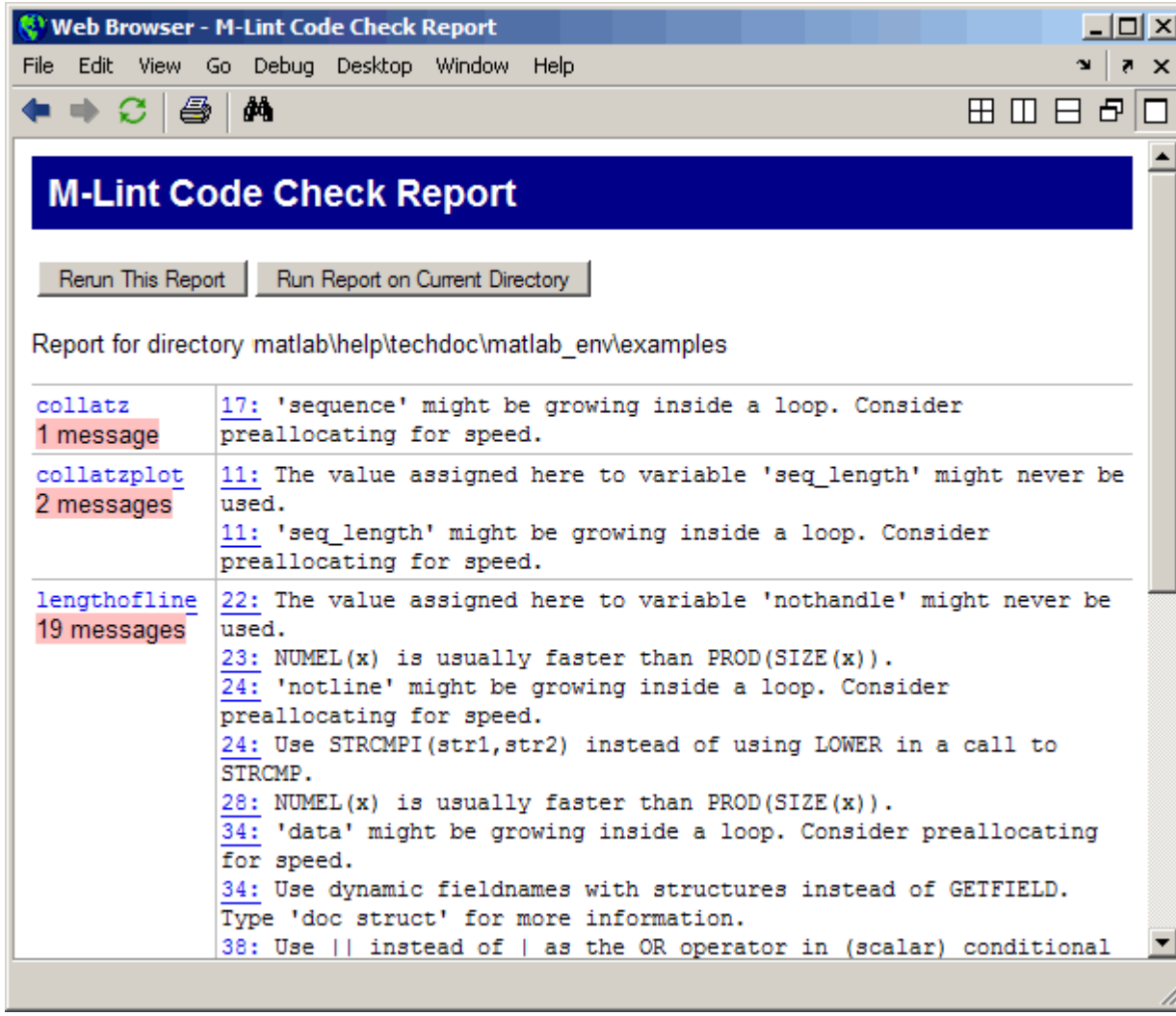
`lengthofline.m` is an example M-file with code that can be improved. It is found in *`matlabroot`*`/matlab/help/techdoc/matlab_env/examples`.

Run Report for All Files in a Directory

Run

```
mlintrpt(fullfile(matlabroot,'help','techdoc','matlab_env','examples'),'dir')
```

and MATLAB displays a report of potential problems and improvements for all M-files in the `examples` directory.



For details about these messages and how to improve the code, see "Making Changes Based on M-Lint Messages" in the MATLAB Desktop Tools and Development Environment documentation.

Run Report Using M-Lint Preference Settings

In **File > Preferences > M-Lint**, save preference settings to a file, for example, `MLintNoSemis.txt`. To apply those settings when you run `mlintrpt`, use the `file` option and supply the full path to the settings filename as shown in this example:

```
mlintrpt('lengthofline.m', 'file', ...
'C:\WINNT\Profiles\me\Application Data\MathWorks\MATLAB\R2007a\MLintNoSemis.txt')
```

Alternatively, use `fullfile` if the settings file is stored in the preferences directory:

```
mlintrpt('lengthofline.m', 'file', fullfile(prefdir,'MLintNoSemis.txt'))
```

Assuming that in that example `MLintNoSemis.txt` file, the setting for `Terminate statement with semicolon to suppress output` has been disabled, the results of `mlintrpt` for `lengthofline` do not show that message for line 49.

When `mlintrpt` cannot locate the settings file, the first message in the report is

```
0: Unable to open or read the configuration file
```

See Also

`mlint`

mlock

Purpose

Prevent clearing M-file or MEX-file from memory

Syntax

```
mlock
```

Description

`mlock` locks the currently running M-file or MEX-file in memory so that subsequent `clear` functions do not remove it.

Use the `munlock` function to return the file to its normal, clearable state.

Locking an M-file or MEX-file in memory also prevents any `persistent` variables defined in the file from getting reinitialized.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked('testfun')
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock('testfun')

mislocked('testfun')
ans =
     0
```

See Also

`mislocked`, `munlock`, `persistent`

Purpose

Information about multimedia file

Syntax

```
info = mmfileinfo(filename)
```

Description

`info = mmfileinfo(filename)` returns a structure, `info`, with fields containing information about the contents of the multimedia file identified by `filename`. The `filename` input is a string enclosed in single quotes.

Note `mmfileinfo` can be used only on Windows systems.

If `filename` is a URL, `mmfileinfo` might take a long time to return because it must first download the file. For large files, downloading can take several minutes. To avoid blocking the MATLAB command line while this processing takes place, download the file before calling `mmfileinfo`.

The `info` structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Filename`	String indicating the name of the file
`Duration`	Length of the file in seconds
`Audio`	Structure containing information about the audio data in the file. See "Audio Data" on page 2-2202 for more information about this data structure.
`Video`	Structure containing information about the video data in the file. See "Video Data" on page 2-2202 for more information about this data structure.

Audio Data

The `Audio` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain audio data, the fields in the structure are empty.

Field	Description
`Format`	Text string, indicating the audio format
`NumberOfChannels`	Number of audio channels

Video Data

The `Video` structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Format`	Text string, indicating the video format
`Height`	Height of the video frame
`Width`	Width of the video frame

Examples

This example gets information about the contents of a file containing audio data.

```
info = mmfileinfo('my_audio_data.mp3')

info =

    Filename: 'my_audio_data.mp3'
    Duration: 1.6030e+002
       Audio: [1x1 struct]
       Video: [1x1 struct]
```

To look at the information returned about the audio data in the file, examine the fields in the `Audio` structure.

```
audio_data = info.Audio

audio_data =

               Format: 'MPEGLAYER3'
    NumberOfChannels: 2
```

Because the file contains only audio data, the fields in the `Video` structure are empty.

```
info.Video

ans =

    Format: ''
    Height: []
     Width: []
```

Purpose

Create multimedia reader object for reading video files

Syntax

```
obj = mmreader(filename)
obj = mmreader(filename, 'P1', V1, 'P2', V2,...)
```

Description

`obj = mmreader(filename)` constructs a multimedia reader object, `obj`, that can read video data from a multimedia file. `filename` is a string specifying the name of a multimedia file. There are no restrictions on file extensions. By default, MATLAB looks for the file `filename` on the MATLAB path. The file formats that `mmreader` supports are AVI, MPG, MPEG, WMV, ASF, and ASX.

If the object cannot be constructed for any reason (for example, if the file cannot be opened or does not exist, or if the file format is not recognized or supported), MATLAB throws an error.

`obj = mmreader(filename, 'P1', V1, 'P2', V2,...)` constructs a multimedia reader object, assigning values `V1`, `V2`, etc. to the specified properties *`P1`*, *`P2`*, etc., respectively. If an invalid property name or property value is specified, MATLAB throws an error and the object is not created. Note that the property value pairs can be in any format supported by the `set` function, i.e., parameter-value string pairs, structures, or parameter-value cell array pairs. The `mmreader` object supports the following properties.

Property	Description	Read-Only	Default Value
`Duration`	Total length of file in seconds	Yes	
`Name`	Name of the file from which the reader object was created	Yes	
`Path`	String containing the full path to the file associated with the reader	Yes	

Property	Description	Read-Only	Default Value
`Tag`	Generic string for the user to set	No	`''`
`Type`	Classname of the object	Yes	`mmreader`
`UserData`	Generic field for any user-defined data	No	`[]`
`BitsPerPixel`	Bits per pixel of the video data	Yes	
`FrameRate`	Frame rate of the video in frames per second	Yes	
`Height`	Height of the video frame in pixels	Yes	
`NumberOfFrames`	Total number of frames in the video stream	Yes	
`VideoFormat`	String indicating the video format as it is represented in MATLAB, e.g., `RGB24`	Yes	
`Width`	Width of the video frame in pixels	Yes	

Remarks

Working with Variable Frame Rate Video

If the video file provided to `mmreader` is a variable frame rate file (as with many Windows Media Video files), MATLAB shows a warning, as in this hypothetical case:

```
>> obj = mmreader('VarFrameRate.wmv')
Warning: Unable to determine the number of frames in this file.

Summary of Multimedia Reader Object for 'VarFrameRate.wmv'.

  Video Parameters:  23.98 frames per second, RGB24 1280x720.
                     Unable to determine video frames available.
```

Because the file `VarFrameRate.wmv` was encoded as a variable frame rate video, the number of frames is not known when you construct the `mmreader` object.

Attempting to Read Beyond the End of the File

You can still read from a variable frame rate file by specifying the number of frames, but `mmreader` and `read` will behave slightly differently depending on the context of the read request.

If you ask for a frame range beyond the end of the file, the system generates an error. For example, suppose you attempt to read frame 3000 in a file that has only 2825 frames:

```
>> images = read(obj, 3000);
??? The frame range requested is beyond the end of the file.
```

If the requested frame range straddles the end of the file, the system returns a warning as shown in the next example, where frames 2800–3000 are requested in a file that has only 2825 frames:

```
>> images = read(obj, [2800 3000]);
Warning: The end of file was reached before the
requested frames were read completely.
Frames 2800 through 2825 were returned.
```

Examples

Construct a multimedia reader object associated with file `xylophone.mpg` with the user `tag` property set to `'myreader1'`.

```
readerobj = mmreader('xylophone.mpg', 'tag', 'myreader1');
```

Read in all the video frames.

```
vidFrames = read(readerobj);
```

Find out how many frames there are.

```
numFrames = get(readerobj, 'numberOfFrames');
```

Create a MATLAB movie struct from the video frames.

```
for k = 1 : numFrames
    mov(k).cdata = vidFrames(:,:,:,k);
    mov(k).colormap = [];
end
```

Play back the movie once at the video's frame rate.

```
movie(mov, 1, readerobj.FrameRate);
```

See Also

`get`, `mmfileinfo`, `read`, `set`

mod

Purpose

Modulus after division

Syntax

```
M = mod(X,Y)
```

Description

`M = mod(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = floor(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

The following are true by convention:

- `mod(X,0)` is `X`
- `mod(X,X)` is `0`
- `mod(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `Y`.

Remarks

`rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

`mod(X,Y)` and `rem(X,Y)` are equal if `X` and `Y` have the same sign, but differ by `Y` if `X` and `Y` have different signs.

The `mod` function is useful for congruence relationships:
x and y are congruent (mod m) if and only if `mod(x,m) == mod(y,m)`.

Examples

```
mod(13,5)
ans =
     3

mod([1:5],3)
ans =
     1     2     0     1     2

mod(magic(3),3)
ans =
     2     1     0
     0     2     1
     1     0     2
```

See Also

`rem`

mode

Purpose

Most frequent values in array

Syntax

```
M = mode(X)
M = mode(X, dim)
[M,F] = mode(X, ...)
[M,F,C] = mode(X, ...)
```

Description

`M = mode(X)` for vector `X` computes the sample mode `M`, (i.e., the most frequently occurring value in `X`). If `X` is a matrix, then `M` is a row vector containing the mode of each column of that matrix. If `X` is an N-dimensional array, then `M` is the mode of the elements along the first nonsingleton dimension of that array.

When there are multiple values occurring equally frequently, `mode` returns the smallest of those values. For complex inputs, this is taken to be the first value in a sorted list of values.

`M = mode(X, dim)` computes the mode along the dimension `dim` of `X`.

`[M,F] = mode(X, ...)` also returns array `F`, each element of which represents the number of occurrences of the corresponding element of `M`. The `M` and `F` output arrays are of equal size.

`[M,F,C] = mode(X, ...)` also returns cell array `C`, each element of which is a sorted vector of all values that have the same frequency as the corresponding element of `M`. All three output arrays `M`, `F`, and `C` are of equal size.

Remarks

The `mode` function is most useful with discrete or coarsely rounded data. The mode for a continuous probability distribution is defined as the peak of its density function. Applying the `mode` function to a sample from that distribution is unlikely to provide a good estimate of the peak; it would be better to compute a histogram or density estimate and calculate the peak of that estimate. Also, the `mode` function is not suitable for finding peaks in distributions having multiple modes.

Examples

Example 1

Find the mode of the 3-by-4 matrix shown here:

```
X = [3 3 1 4; 0 0 1 1; 0 1 2 4]
X =
     3     3     1     4
     0     0     1     1
     0     1     2     4

mode(X)
ans =
     0     0     1     4
```

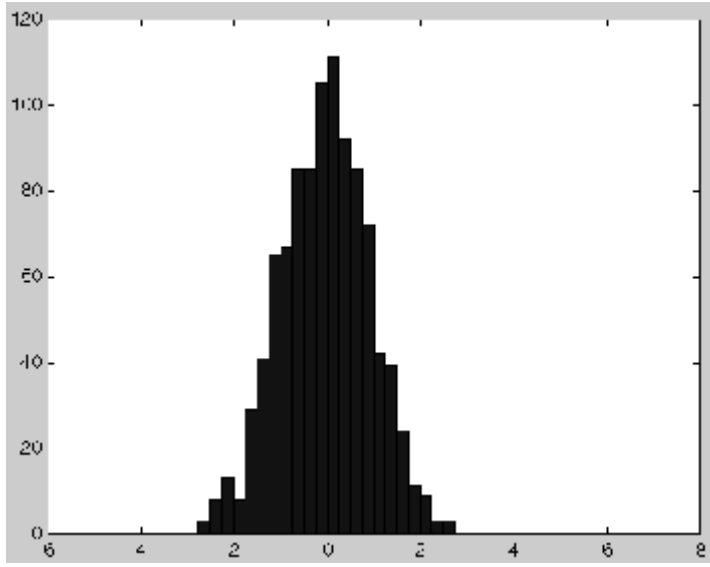
Find the mode along the second (row) dimension:

```
mode(X, 2)
ans =
     3
     0
     0
```

Example 2

Find the mode of a continuous variable grouped into bins:

```
randn('state', 0);       % Reset the random number generator

y = randn(1000,1);
edges = -6:.25:6;
[n,bin] = histc(y,edges);

m = mode(bin)
m =
    22

edges([m, m+1])
ans =
   -0.7500   -0.5000

hist(y,edges+.125)
```



See Also

`mean`, `median`, `hist`, `histc`

Purpose

Control paged output for Command Window

Syntax

```
more on
more off
more(n)
A = more(state)
```

Description

`more on` enables paging of the output in the MATLAB Command Window. MATLAB displays output one page at a time. Use the keys defined in the table below to control paging.

`more off` disables paging of the output in the MATLAB Command Window.

`more(n)` defines the length of a page to be `n` lines.

`A = more(state)` returns in `A` the number of lines that are currently defined to be a page. The `state` input can be one of the quoted strings `'on'` or `'off'`, or the number of lines to set as the new page length.

By default, the length of a page is equal to the number of lines available for display in the MATLAB command window. Manually changing the size of the command window adjusts the page length accordingly.

If you set the page length to a specific value, MATLAB uses that value for the page size, regardless of the size of the command window. To have MATLAB return to matching page size to window size, type `more off` followed by `more on`.

To see the status of `more`, type `get(0,'More')`. MATLAB returns either `on` or `off` indicating the `more` status. You can also set status for `more` by using `set(0,'More', 'status')`, where `'status'` is either `'on'` or `'off'`.

When you have enabled `more` and are examining output, you can do the following.

Press the...	To...
Return key	Advance to the next line of output.
Space bar	Advance to the next page of output.
Q (for quit) key	Terminate display of the text. Do not use **Ctrl+C** to terminate `more` or you might generate error messages in the Command Window.

`more` is in the **`off`** state, by default.

See Also

`diary`

Purpose

Move or resize control in parent window

Syntax

```
V = h.move(position)
V = move(h, position)
```

Description

`V = h.move(position)` moves the control to the position specified by the `position` argument. When you use move with only the handle argument, `h`, it returns a four-element vector indicating the current position of the control.

`V = move(h, position)` is an alternate syntax for the same operation.

The position argument is a four-element vector specifying the position and size of the control in the parent figure window. The elements of the vector are

```
[x, y, width, height]
```

where `x` and `y` are offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control, and `width` and `height` are the size of the control itself.

Examples

This example moves the control:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200], f);
pos = h.move([50 50 200 200])
pos =
    50    50   200   200
```

The next example resizes the control to always be centered in the figure as you resize the figure window. Start by creating the script `resizectrl.m` that contains

```
% Get the new position and size of the figure window
  fpos = get(gcbo, 'position');

% Resize the control accordingly
```

```
    h.move([0 0 fpos(3) fpos(4)]);
```

Now execute the following in MATLAB or in an M-file:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
set(f, 'ResizeFcn', 'resizectrl');
```

As you resize the figure window, notice that the circle moves so that it is always positioned in the center of the window.

See Also

`set`, `get`

Purpose

Move file or directory

Graphical Interface

As an alternative to the `movefile` function, you can use the Current Directory browser to move files and directories.

Syntax

```
movefile('source')
movefile('source','destination')
movefile('source','destination','f')
[status,message,messageid]=movefile('source','destination',
   'f')
```

Description

`movefile('source')` moves the file or directory named `source` to the current directory, where `source` is the absolute or relative pathname for the directory or file. Use the wildcard `*` at the end of `source` to move all matching files. Note that the archive attribute of `source` is not preserved.

`movefile('source','destination')` moves the file or directory named `source` to the location `destination`, where `source` and `destination` are the absolute or relative pathnames for the directory or files. To rename a file or directory when moving it, make `destination` a different name than `source`. Use the wildcard `*` at the end of `source` to move all matching files.

`movefile('source','destination','f')` moves the file or directory named `source` to the location `destination`, regardless of the read-only attribute of `destination`.

`[status,message,messageid]=movefile('source','destination','f')` moves the file or directory named `source` to the location `destination`, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterror`). Here, `status` is logical `1` for success or logical `0` for error. Only one output argument is required and the `f` input argument is optional.

The `*` wildcard in a path string is supported.

Examples

Move Source to Current Directory

To move the file myfiles/myfunction.m to the current directory, type

```
movefile('myfiles/myfunction.m')
```

If the current directory is `projects/testcases` and you want to move `projects/myfiles` and its contents to the current directory, use `../` in the source pathname to navigate up one level to get to the directory.

```
movefile('../myfiles')
```

Move All Matching Files by Using a Wildcard

To move all files in the directory `myfiles` whose names begin with my to the current directory, type

```
movefile('myfiles/my*')
```

Move Source to Destination

To move the file `myfunction.m` from the current directory to the directory `projects`, where `projects` and the current directory are at the same level, type

```
movefile('myfunction.m','../projects')
```

Move Directory Down One Level

This example moves the a directory down a level. For example to move the directory `projects/testcases` and all its contents down a level in `projects` to `projects/myfiles`, type

```
movefile('projects/testcases','projects/myfiles/')
```

The directory `testcases` and its contents now appear in the directory `myfiles`.

Rename When Moving File to Read-Only Directory

Move the file `myfile.m` from the current directory to `d:/work/restricted`, assigning it the name `test1.m`, where `restricted` is a read-only directory.

```
movefile('myfile.m','d:/work/restricted/test1.m','f')
```

The read-only file `myfile.m` is no longer in the current directory. The file `test1.m` is in `d:/work/restricted` and is read only.

Return Status When Moving Files

In this example, all files in the directory `myfiles` whose names start with `new` are to be moved to the current directory. However, if `new*` is accidentally written as `nex*`. As a result, the move is unsuccessful, as seen in the status and messages returned:

```
[s,mess,messid]=movefile('myfiles/nex*')

s =
     0

mess =

A duplicate filename exists, or the file cannot be found.

messid =

MATLAB:MOVEFILE:OSError
```

See Also

`cd`, `copyfile`, `delete`, `dir`, `fileattrib`, `filebrowser`, `ls`, `mkdir`, `rmdir`

Purpose

Move GUI figure to specified location on screen

Syntax

```
movegui(h,'position')
movegui('position')
movegui(h)
movegui
```

Description

`movegui(h,'position')` moves the figure identified by handle `h` to the specified screen location, preserving the figure's size. The *`position`* argument can be any of the following strings:

- `north` – top center edge of screen
- `south` – bottom center edge of screen
- `east` – right center edge of screen
- `west` – left center edge of screen
- `northeast` – top right corner of screen
- `northwest` – top left corner of screen
- `southeast` – bottom right corner of screen
- `southwest` – bottom left corner
- `center` – center of screen
- `onscreen` – nearest location with respect to current location that is on screen

The *`position`* argument can also be a two-element vector `[h,v]`, where depending on sign, `h` specifies the figure's offset from the left or right edge of the screen, and `v` specifies the figure's offset from the top or bottom of the screen, in pixels. The following table summarizes the possible values.

`h` (for `h >= 0`)	offset of left side from left edge of screen
`h` (for `h < 0`)	offset of right side from right edge of screen
`v` (for `v >= 0`)	offset of bottom edge from bottom of screen
`v` (for `v < 0`)	offset of top edge from top of screen

`movegui('position')` move the callback figure (`gcbf`) or the current figure (`gcf`) to the specified position.

`movegui(h)` moves the figure identified by the handle `h` to the `onscreen` position.

`movegui` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the `onscreen` position. This is useful as a string-based `CreateFcn` callback for a saved figure. It ensures the figure appears on screen when reloaded, regardless of its saved position.

Examples

This example demonstrates the usefulness of `movegui` to ensure that saved GUIs appear on screen when reloaded, regardless of the target computer's screen sizes and resolution. It creates a figure off the screen, assigns `movegui` as its `CreateFcn` callback, then saves and reloads the figure.

```
f = figure('Position',[10000,10000,400,300]);
set(f,'CreateFcn','movegui')
hgsave(f,'onscreenfig')
close(f)
f2 = hgload('onscreenfig');
```

See Also

`guide`

"Creating GUIs" in the MATLAB documentation

movie

Purpose

Play recorded movie frames

Syntax

```
movie
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

Description

`movie` plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once, using the current axes as the default target. If you want to play the movie in the figure instead of the axes, specify the figure handle (or `gcf`) as the first argument: `movie(figure_handle,...)`. `M` must be an array of movie frames (usually from `getframe`).

`movie(M,n)` plays the movie `n` times. If `n` is negative, each cycle is shown forward then backward. If `n` is a vector, the first element is the number of times to play the movie, and the remaining elements make up a list of frames to play in the movie.

For example, if `M` has four frames then `n = [10 4 4 2 1]` plays the movie ten times, and the movie consists of frame `4` followed by frame `4` again, followed by frame `2` and finally frame `1`.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie centered in the figure or axes identified by the handle `h`.

`movie(h,M,n,fps,loc)` specifies `loc`, a four-element location vector, `[x y 0 0]`, where the lower left corner of the movie frame is anchored (only the first two elements in the vector are used). The location is relative to the lower left corner of the figure or axes specified by `handle h` and in units of pixels, regardless of the object's `Units` property.

Remarks

The `movie` function uses a default figure size of 560-by-420 and does not resize figures to fit movies with larger or smaller frames. To accommodate other frame sizes, you can resize the figure to fit the movie, as shown in the second example below.

`movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale or 24–bit truecolor image frames.

You can abort a movie by typing **Ctrl-C**.

Examples

Example 1: Animate the `peaks` function as you scale the values of `Z`:

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
 surf(sin(2*pi*j/20)*Z,Z)
 F(j) = getframe;
end
% Play the movie ten times
movie(F,10)
```

Example 2: Specify figure when calling movie to fit the movie to the figure:

```
r = subplot(2,1,1)
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');

s = subplot(2,1,2)
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
 axes(r)
```

```
    surf(sin(2*pi*j/20)*Z,Z)
 axes(s)
 surf(sin(2*pi*(j+5)/20)*Z,Z)
    F(j) = getframe(gcf);
 pause(.0333)
end
% Play the movie; note that it does not fit the figure properly:
h2 = figure;
movie(F,10)
% Use the figure handle to make the frames fit:
movie(h2,F,10)
```

Example 3: With larger frames, first adjust the figure's size to fit the movie:

```
figure('position',[100 100 850 600])
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
% Record the movie
for j = 1:20
 surf(sin(2*pi*j/20)*Z,Z)
 F(j) = getframe;
end
[h, w, p] = size(F(1).cdata);  % use 1st frame to get dimensions
hf = figure;
% resize figure based on frame's w x h, and place at (150, 150)
set(hf, 'position', [150 150 w h]);
axis off
% tell movie command to place frames at bottom left
movie(hf,F,4,30,[0 0 0 0]);
```

See Also

`aviread`, `getframe`, `frame2im`, `im2frame`

"Animation" on page 1-91 for related functions

See Example – Visualizing an FFT as a Movie for another example

Purpose

Create Audio/Video Interleaved (AVI) movie from MATLAB movie

Syntax

```
movie2avi(mov,filename)
movie2avi(mov,filename,param,value,param,value...)
```

Description

`movie2avi(mov,filename)` creates the AVI movie `filename` from the MATLAB movie `mov`. The `filename` input is a string enclosed in single quotes.

`movie2avi(mov,filename,param,value,param,value...)` creates the AVI movie `filename` from the MATLAB movie `mov` using the specified parameter settings.

Parameter	Value	Default
`'colormap'`	An `m`-by-3 matrix defining the colormap to be used for indexed AVI movies, where `m` must be no greater than 256 (236 if using `Indeo` compression).	There is no default colormap.
`'compression'`	A text string specifying the compression codec to use.	
	On Windows: `'Indeo3'` `'Indeo5'` `'Cinepak'` `'MSVC'` `'RLE'` `'None'` On UNIX: `'None'`	`'Indeo5'` on Windows. `'None'` on UNIX.

Parameter	Value	Default
	To use a custom compression codec, specify the four-character code that identifies the codec (typically included in the codec documentation). The `addframe` function reports an error if it can not find the specified custom compressor.	
`'fps'`	A scalar value specifying the speed of the AVI movie in frames per second (fps).	15 fps
`'keyframe'`	For compressors that support temporal compression, this is the number of key frames per second.	2 key frames per second.
`'quality'`	A number between 0 and 100 the specifies the desired quality of the output. Higher numbers result in higher video quality and larger file sizes. Lower numbers result in lower video quality and smaller file sizes. This parameter has no effect on uncompressed movies.	75
`'videoname'`	A descriptive name for the video stream. This parameter must be no greater than 64 characters long.	The default is the `filename`.

See Also

`avifile`, `aviread`, `aviinfo`, `movie`

Purpose

Upload file or directory to FTP server

Syntax

```
mput(f,'filename')
mput(ftp,'directoryname')
mput(f,'wildcard')
```

Description

`mput(f,'filename')` uploads `filename` from the MATLAB current directory to the current directory of the FTP server `f`, where `filename` is a file, and where `f` was created using `ftp`. You can use a wildcard (`*`) in `filename`. MATLAB returns a cell array listing the full path to the uploaded files on the server.

`mput(ftp,'directoryname')` uploads the directory `directoryname` and its contents. MATLAB returns a cell array listing the full path to the uploaded files on the server.

`mput(f,'wildcard')` uploads a set of files or directories specified by a wildcard. MATLAB returns a cell array listing the full path to the uploaded files on the server.

See Also

`ftp`, `mget`, `mkdir (ftp)`, `rename`

Purpose

Create and open message box

Syntax

```
h = msgbox(Message)
h = msgbox(Message,Title)
h = msgbox(Message,Title,Icon)
h = msgbox(Message,Title,'custom',IconData,IconCMap)
h = msgbox(...,CreateMode)
```

Description

`h = msgbox(Message)` creates a message dialog box that automatically wraps `Message` to fit an appropriately sized figure. `Message` is a string vector, string matrix, or cell array. `msgbox` returns the handle of the message box in `h`.

`h = msgbox(Message,Title)` specifies the title of the message box.

`h = msgbox(Message,Title,Icon)` specifies which icon to display in the message box. `Icon` is `'none'`, `'error'`, `'help'`, `'warn'`, or `'custom'`. The default is `'none'`.

Error Icon

Help Icon

Warning Icon

`h = msgbox(Message,Title,'custom',IconData,IconCMap)` defines a customized icon. `IconData` contains image data defining the icon. `IconCMap` is the colormap used for the image.

`h = msgbox(...,CreateMode)` specifies whether the message box is modal or nonmodal. Optionally, it can also specify an interpreter for `Message` and `Title`.

If *`CreateMode`* is a string, it must be one of the values shown in the following table.

CreateMode Value	Description
`'modal'`	Replaces the message box having the specified `Title`, that was last created or clicked on, with a modal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.
`'non-modal'` (default)	Creates a new nonmodal message box with the specified parameters. Existing message boxes with the same title are not deleted.
`'replace'`	Replaces the message box having the specified `Title`, that was last created or clicked on, with a nonmodal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the`uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLABFigure Properties.

If *`CreateMode`* is a structure, it can have fields `WindowStyle` and `Interpreter`. The `WindowStyle` field must be one of the values in the table above. `Interpreter` is one of the strings`'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

See Also

`dialog`, `errordlg`, `helpdlg`, `inputdlg`, `listdlg`, `questdlg`, `warndlg`

`figure`, `textwrap`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-104 for related functions

Purpose

Matrix multiplication

Syntax

```
C = A*B
```

Description

`C = A*B` is the linear algebraic product of the matrices `A` and `B`. If `A` is an m-by-p and `B` is a p-by-n matrix, the `i,j` entry of `C` is defined by

$$C(i,j) = \sum_{k=1}^{p} A(i,k)B(k,j)$$

The product `C` is an m-by-n matrix. For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. You can multiply a scalar by a matrix of any size.

The preceding definition says that `C(i,j)` is the inner product of the `i`th row of `A` with the `j`th column of `B`. You can write this definition using the MATLAB colon operator as

```
C(i,j) = A(i,:)*B(:,j)
```

where `A(i,:)` is the `i`th row of `A` and `B(:,j)` is the `j`th row of `B`.

Note If `A` is an m-by-0 empty matrix and `B` is a 0-by-n empty matrix, where m and n are positive integers, `A*B` is an m-by-n matrix of all zeros.

Examples

Example 1

If `A` is a row vector and `B` is a column vector with the same number of elements as `A`, `A*B` is simply the inner product of `A` and `B`. For example,

```
A = [5 3 2 6]

A =

     5     3     2     6
```

```
B = [-4 9 0 1]'

B =

    -4
     9
     0
     1
A*B

ans =

    13
```

Example 2

```
A = [1 3 5; 2 4 7]

A =

     1     3     5
     2     4     7
B = [-5 8 11; 3 9 21;4 0 8]

B =

    -5     8    11
     3     9    21
     4     0     8
```

The product of `A` and `B` is

```
C = A*B

C =

    24    35   114
    30    52   162
```

Note that the second row of A is

```
A(2,:)

ans =

     2     4     7
```

while the third column of B is

```
B(:,3)

ans =

    11
    21
     8
```

The inner product of `A(2,:)` and `B(:,3)` is

```
A(2,:)*B(:,3)

ans =

   162
```

which is the same as `C(2,3)`.

Algorithm

`mtimes` uses the following Basic Linear Algebra Subroutines (BLAS):

- `DDOT`
- `DGEMV`
- `DGEMM`
- `DSYRK`
- `DSYRZK`

For inputs of type `single`, mtimes using corresponding routines that begin with “`S`” instead of “`D`”.

See Also

Arithmetic Operators

mu2lin

Purpose

Convert mu-law audio signal to linear

Syntax

```
y = mu2lin(mu)
```

Description

`y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as “flints” in the range $0 \le mu \le 255$, to linear signal amplitude in the range `-s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. "Flints" are MATLAB integers — floating-point numbers whose values are integers.

See Also

`auread`, `lin2mu`

Purpose

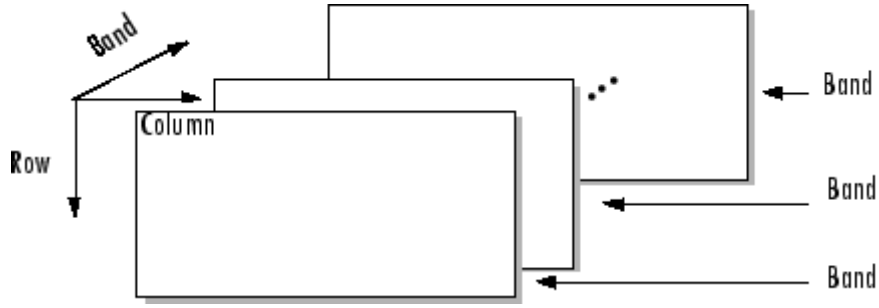
Read band-interleaved data from binary file

Syntax

```
X = multibandread(filename, size, precision, offset,
   interleave,  byteorder)
X = multibandread(...,subset1,subset2,subset3)
```

Description

`X = multibandread(filename, size, precision, offset, interleave, byteorder)` reads band-sequential (BSQ), band-interleaved-by-line (BIL), or band-interleaved-by-pixel (BIP) data from the binary file `filename`. The `filename` input is a string enclosed in single quotes. This function defines *band* as the third dimension in a 3-D array, as shown in this figure.



You can use the parameters to `multibandread` to specify many aspects of the read operation, such as which bands to read. See “Parameters” on page 2-2235 for more information.

`X` is a 2-D array if only one band is read; otherwise it is 3-D. `X` is returned as an array of data type `double` by default. Use the `precision` parameter to map the data to a different data type.

`X = multibandread(...,subset1,subset2,subset3)` reads a subset of the data in the file. You can use up to three subsetting parameters to specify the data subset along row, column, and band dimensions. See “Subsetting Parameters” on page 2-2237 for more information.

Parameters

This table describes the arguments accepted by `multibandread`.

Argument	Description
`filename`	String containing the name of the file to be read.
`size`	Three-element vector of integers consisting of `[height, width, N]`, where • `height` is the total number of rows • `width` is the total number of elements in each row • `N` is the total number of bands. This will be the dimensions of the data if it is read in its entirety.
`precision`	String specifying the format of the data to be read, such as `'uint8'`, `'double'`, `'integer*4'`, or any of the other precisions supported by the `fread` function. Note: You can also use the `precision` parameter to specify the format of the output data. For example, to read `uint8` data and output a `uint8` array, specify a precision of `'uint8=>uint8'` (or `'*uint8'`). To read `uint8` data and output it in MATLAB in single precision, specify `'uint8=>single'`. See `fread` for more information.
`offset`	Scalar specifying the zero-based location of the first data element in the file. This value represents the number of bytes from the beginning of the file to where the data begins.

Argument	Description
`interleave`	String specifying the format in which the data is stored • `'bsq'` — Band-Sequential • `'bil'` — Band-Interleaved-by-Line • `'bip'` — Band-Interleaved-by-Pixel For more information about these interleave methods, see the `multibandwrite` reference page.
`byteorder`	String specifying the byte ordering (machine format) in which the data is stored, such as • `'ieee-le'` — Little-endian • `'ieee-be'` — Big-endian See `fopen` for a complete list of supported formats.

Subsetting Parameters

You can specify up to three subsetting parameters. Each subsetting parameter is a three-element cell array, {*dim*, *method*, index}, where

Parameter	Description
dim	Text string specifying the dimension to subset along. It can have any of these values: • `'Column'` • `'Row'` • `'Band'`

Parameter	Description
method	Text string specifying the subsetting method. It can have either of these values: • 'Direct' • 'Range' If you leave out this element of the subset cell array, multibandread uses 'Direct' as the default.
index	If method is 'Direct', index is a vector specifying the indices to read along the Band dimension. If method is 'Range', index is a three-element vector of [start, increment, stop] specifying the range and step size to read along the dimension specified in dim. If index is a two-element vector, multibandread assumes that the value of increment is 1.

Examples

Example 1

Setup initial parameters for a data set.

```
rows=3; cols=3; bands=5;
filename = tempname;
```

Define the data set.

```
fid = fopen(filename, 'w', 'ieee-le');
fwrite(fid, 1:rows*cols*bands, 'double');
fclose(fid);
```

Read every other band of the data using the Band-Sequential format.

```
im1 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bsq', 'ieee-le', ...
```

```
                        {'Band', 'Range', [1 2 bands]} )
```

Read the first two rows and columns of data using Band-Interleaved-by-Pixel format.

```
im2 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bip', 'ieee-le', ...
                    {'Row', 'Range', [1 2]}, ...
                    {'Column', 'Range', [1 2]} )
```

Read the data using Band-Interleaved-by-Line format.

```
im3 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bil', 'ieee-le')
```

Delete the file created in this example.

```
delete(filename);
```

Example 2

Read `int16` BIL data from the FITS file `tst0012.fits`, starting at byte 74880.

```
im4 = multibandread('tst0012.fits', [31 73 5], ...
                    'int16', 74880, 'bil', 'ieee-be', ...
                    {'Band', 'Range', [1 3]} );
im5 = double(im4)/max(max(max(im4)));
imagesc(im5);
```

See Also

`fread`, `fwrite`, `multibandwrite`

multibandwrite

Purpose

Write band-interleaved data to file

Syntax

```
multibandwrite(data,filename,interleave)
multibandwrite(data,filename,interleave,start,totalsize)
multibandwrite(...,param,value...)
```

Description

`multibandwrite(data,filename,interleave)` writes `data`, a two- or three-dimensional numeric or logical array, to the binary file specified by `filename`. The `filename` input is a string enclosed in single quotes. The length of the third dimension of `data` determines the number of bands written to the file. The bands are written to the file in the form specified by `interleave`. See "Interleave Methods" on page 2-2242 for more information about this argument.

If filename already exists, `multibandwrite` overwrites it unless you specify the optional `offset` parameter. See the last alternate syntax for `multibandwrite` for information about other optional parameters.

`multibandwrite(data,filename,interleave,start,totalsize)` writes `data` to the binary file `filename` in chunks. In this syntax, `data` is a subset of the complete data set.

`start` is a 1-by-3 array `[firstrow firstcolumn firstband]` that specifies the location to start writing data. `firstrow` and `firstcolumn` specify the location of the upper left image pixel. `firstband` gives the index of the first band to write. For example, `data(I,J,K)` contains the data for the pixel at `[firstrow+I-1, firstcolumn+J-1]` in the `(firstband+K-1)`-th band.

`totalsize` is a 1-by-3 array, `[totalrows,totalcolumns,totalbands]`, which specifies the full, three-dimensional size of the data to be written to the file.

Note In this syntax, you must call `multibandwrite` multiple times to write all the data to the file. The first time it is called, `multibandwrite` writes the complete file, using the fill value for all values outside the data subset. In each subsequent call, `multibandwrite` overwrites these fill values with the data subset in `data`. The parameters `filename`, `interleave`, `offset`, and `totalsize` must remain constant throughout the writing of the file.

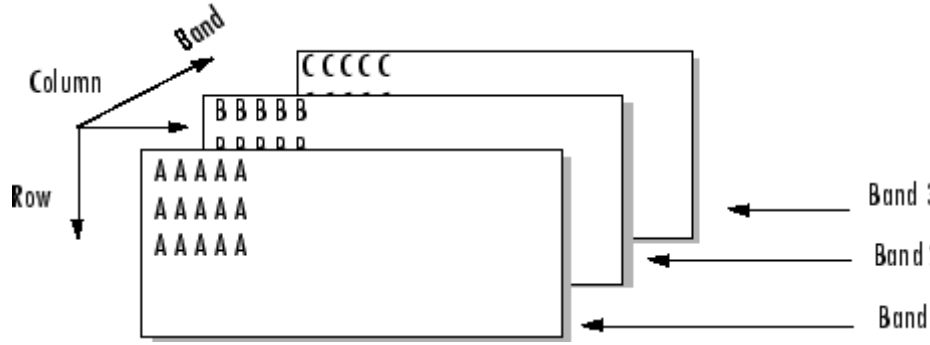
`multibandwrite(...,param,value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

Parameter	Description
`'precision'`	String specifying the form and size of each element written to the file. See the help for `fwrite` for a list of valid values. The default precision is the class of the data.
`'offset'`	The number of bytes to skip before the first data element. If the file does not already exist, `multibandwrite` writes ASCII null values to fill the space. To specify a different fill value, use the parameter `'fillvalue'`. This option is useful when you are writing a header to the file before or after writing the data. When writing the header to the file after the data is written, open the file with `fopen` using `'r+'` permission.

Parameter	Description
`'machfmt'`	String to control the format in which the data is written to the file. Typical values are `'ieee-le'` for little endian and `'ieee-be'` for big endian. See the help for `fopen` for a complete list of available formats. The default machine format is the local machine format.
`'fillvalue'`	A number specifying the value to use in place of missing data. `'fillvalue'` can be a single number, specifying the fill value for all missing data, or a 1-by-Number-of-bands vector of numbers specifying the fill value for each band. This value is used to fill space when data is written in chunks.

Interleave Methods

`interleave` is a string that specifies how `multibandwrite` interleaves the bands as it writes data to the file. If `data` is two-dimensional, `multibandwrite` ignores the `interleave` argument. The following table lists the supported methods and uses this example multiband file to illustrate each method.



Supported methods of interleaving bands include those listed below.

Method	String	Description	Example
Band-Interleaved-by-Line	`'bil'`	Write an entire row from each band	AAAAABBBBBCCCCC AAAAABBBBBCCCCC AAAAABBBBBCCCCC
Band-Interleaved-by-Pixel	`'bip'`	Write a pixel from each band	ABCABCABCABCABC...
Band-Sequential	`'bsq'`	Write each band in its entirety	AAAAA AAAAA AAAAA BBBBB BBBBB BBBBB CCCCC CCCCC CCCCC

Examples

Note To run these examples successfully, you must be in a writable directory.

Example 1

Write all data (interleaved by line) to the file in one call.

```
data = reshape(uint16(1:600), [10 20 3]);
multibandwrite(data,'data.bil','bil');
```

Example 2

Write the bands (interleaved by pixel) to the file in separate calls.

```
totalRows    = size(data, 1);
totalColumns = size(data, 2);
totalBands   = size(data, 3);
for i = 1:totalBands
   bandData = data(:, :, i);
   multibandwrite(bandData, 'data.bip', 'bip', [1 1 i],...
                  [totalColumns, totalRows, totalBands]);
end
```

Example 3

Write a single-band tiled image with one call for each tile. This is only useful if a subset of each band is available at each call to `multibandwrite`.

```
numBands = 1;
dataDims = [1024 1024 numBands];
data = reshape(uint32(1:(1024 * 1024 * numBands)), dataDims);

for band = 1:numBands
   for row = 1:2
      for col = 1:2

         subsetRows = ((row - 1) * 512 + 1):(row * 512);
         subsetCols = ((col - 1) * 512 + 1):(col * 512);

         upperLeft = [subsetRows(1), subsetCols(1), band];
         multibandwrite(data(subsetRows, subsetCols, band), ...
                        'banddata.bsq', 'bsq', upperLeft, dataDims);

      end
   end
```

```
    end
```

See Also

multibandread, fwrite, fread

munlock

Purpose

Allow clearing M-file or MEX-file from memory

Syntax

```
munlock
munlock fun
munlock('fun')
```

Description

`munlock` unlocks the currently running M-file or MEX-file in memory so that subsequent `clear` functions can remove it.

`munlock fun` unlocks the M-file or MEX-file named `fun` from memory. By default, these files are unlocked so that changes to the file are picked up. Calls to munlock are needed only to unlock M-files or MEX-files that have been locked with `mlock`.

`munlock('fun')` is the function form of `munlock`.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
  .
  .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked testfun
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock testfun

mislocked testfun
ans =
```

```
0
```

See Also

mlock, mislocked, persistent

namelengthmax

Purpose

Maximum identifier length

Syntax

```
len = namelengthmax
```

Description

`len = namelengthmax` returns the maximum length allowed for MATLAB identifiers. MATLAB identifiers are

- Variable names
- Function and subfunction names
- Structure fieldnames
- Object names
- M-file names
- MEX-file names
- MDL-file names

Rather than hard-coding a specific maximum name length into your programs, use the `namelengthmax` function. This saves you the trouble of having to update these limits should the identifier length change in some future MATLAB release.

Examples

Call `namelengthmax` to get the maximum identifier length:

```
maxid = namelengthmax
maxid =
    63
```

See Also

`isvarname`, `genvarname`

Purpose

Not-a-Number

Syntax

```
NaN
```

Description

`NaN` returns the IEEE arithmetic representation for Not-a-Number (`NaN`). These result from operations which have undefined numerical results.

`NaN('double')` is the same as `NaN` with no inputs.

`NaN('single')` is the single precision representation of `NaN`.

`NaN(n)` is an `n`-by-`n` matrix of `NaN`s.

`NaN(m,n)` or `NaN([m,n])` is an `m`-by-`n` matrix of `NaN`s.

`NaN(m,n,p,...)` or `NaN([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `NaN`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`NaN(...,classname)` is an array of `NaN`s of class specified by `classname`. `classname` must be either `'single'` or `'double'`.

Examples

These operations produce `NaN`:

- Any arithmetic operation on a `NaN`, such as `sqrt(NaN)`
- Addition or subtraction, such as magnitude subtraction of infinities as `(+Inf)+(-Inf)`
- Multiplication, such as `0*Inf`
- Division, such as `0/0` and `Inf/Inf`
- Remainder, such as `rem(x,y)` where `y` is zero or `x` is infinity

Remarks

Because two NaNs are not equal to each other, logical operations involving NaNs always return false, except ~= (not equal). Consequently,

```
NaN ~= NaN
ans =
     1
NaN == NaN
ans =
     0
```

and the NaNs in a vector are treated as different unique elements.

```
unique([1 1 NaN NaN])
ans =
     1 NaN NaN
```

Use the isnan function to detect NaNs in an array.

```
isnan([1 1 NaN NaN])
ans =
     0     0     1     1
```

See Also

Inf, isnan

Purpose

Validate number of input arguments

Syntax

```
msgstring = nargchk(minargs, maxargs, numargs)
msgstring = nargchk(minargs, maxargs, numargs, 'string')
msgstruct = nargchk(minargs, maxargs, numargs, 'struct')
```

Description

Use `nargchk` inside an M-file function to check that the desired number of input arguments is specified in the call to that function.

`msgstring = nargchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of inputs specified in the call `numargs` is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty matrix.

It is common to use the `nargin` function to determine the number of input arguments specified in the call.

`msgstring = nargchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargchk` returns a string by default.

`msgstruct = nargchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty structure.

When too few inputs are supplied, the message string and identifier are

```
   message: 'Not enough input arguments.'
identifier: 'MATLAB:nargchk:notEnoughInputs'
```

When too many inputs are supplied, the message string and identifier are

```
   message: 'Too many input arguments.'
identifier: 'MATLAB:nargchk:tooManyInputs'
```

Remarks

nargchk is often used together with the error function. The error function accepts either type of return value from nargchk: a message string or message structure. For example, this command provides the error function with a message string and identifier regarding which error was caught:

```
error(nargchk(2, 4, nargin, 'struct'))
```

If nargchk detects no error, it returns an empty string or structure. When nargchk is used with the error function, as shown here, this empty string or structure is passed as an input to error. When error receives an empty string or structure, it simply returns and no error is generated.

Examples

Given the function foo,

```
function f = foo(x, y, z)
error(nargchk(2, 3, nargin))
```

Then typing foo(1) produces

```
Not enough input arguments.
```

See Also

nargoutchk, nargin, nargout, varargin, varargout, error

Purpose

Number of function arguments

Syntax

```
nargin
nargin(fun)
nargout
nargout(fun)
```

Description

In the body of a function M-file, `nargin` and `nargout` indicate how many input or output arguments, respectively, a user has supplied. Outside the body of a function M-file, `nargin` and `nargout` indicate the number of input or output arguments, respectively, for a given function. The number of arguments is negative if the function has a variable number of arguments.

`nargin` returns the number of input arguments specified for a function.

`nargin(fun)` returns the number of declared inputs for the function `fun`. If the function has a variable number of input arguments, `nargin` returns a negative value. `fun` may be the name of a function, or the name of "Function Handles" that map to specific functions.

`nargout` returns the number of output arguments specified for a function.

`nargout(fun)` returns the number of declared outputs for the function `fun`. `fun` may be the name of a function, or the name of "Function Handles" that map to specific functions.

Examples

This example shows portions of the code for a function called `myplot`, which accepts an optional number of input and output arguments:

```
function [x0, y0] = myplot(x, y, npts, angle, subdiv)
% MYPLOT  Plot a function.
% MYPLOT(x, y, npts, angle, subdiv)
%     The first two input arguments are
%     required; the other three have default values.
 ...
if nargin < 5, subdiv = 20; end
if nargin < 4, angle = 10; end
```

```
if nargin < 3, npts = 25; end
 ...
if nargout == 0
     plot(x, y)
else
     x0 = x;
     y0 = y;
end
```

See Also

inputname, varargin, varargout, nargchk, nargoutchk

Purpose

Validate number of output arguments

Syntax

```
msgstring = nargoutchk(minargs, maxargs, numargs)
msgstring = nargoutchk(minargs, maxargs, numargs, 'string')
msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')
```

Description

Use `nargoutchk` inside an M-file function to check that the desired number of output arguments is specified in the call to that function.

`msgstring = nargoutchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of outputs specified in the call, `numargs`, is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty matrix.

It is common to use the `nargout` function to determine the number of output arguments specified in the call.

`msgstring = nargoutchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargoutchk` returns a string by default.

`msgstruct = nargoutchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargoutchk` returns an empty structure.

When too few outputs are supplied, the message string and identifier are

```
   message: 'Not enough output arguments.'
identifier: 'MATLAB:nargoutchk:notEnoughOutputs'
```

When too many outputs are supplied, the message string and identifier are

```
   message: 'Too many output arguments.'
identifier: 'MATLAB:nargoutchk:tooManyOutputs'
```

Remarks

nargoutchk is often used together with the error function. The error function accepts either type of return value from nargoutchk: a message string or message structure. For example, this command provides the error function with a message string and identifier regarding which error was caught:

```
error(nargoutchk(2, 4, nargout, 'struct'))
```

If nargoutchk detects no error, it returns an empty string or structure. When nargoutchk is used with the error function, as shown here, this empty string or structure is passed as an input to error. When error receives an empty string or structure, it simply returns and no error is generated.

Examples

You can use nargoutchk to determine if an M-file has been called with the correct number of output arguments. This example uses nargout to return the number of output arguments specified when the function was called. The function is designed to be called with one, two, or three output arguments. If called with no arguments or more than three arguments, nargoutchk returns an error message:

```
function [s, varargout] = mysize(x)
msg = nargoutchk(1, 3, nargout);
if isempty(msg)
    nout = max(nargout, 1) - 1;
    s = size(x);
    for k = 1:nout, varargout(k) = {s(k)}; end
else
    disp(msg)
end
```

See Also

nargchk, nargout, nargin, varargout, varargin, error

Purpose

Convert numeric bytes to Unicode characters

Syntax

```
unicodestr = native2unicode(bytes)
unicodestr = native2unicode(bytes, encoding)
```

Description

`unicodestr = native2unicode(bytes)` takes a vector containing numeric values in the range [0,255] and converts these values as a stream of 8-bit bytes to Unicode characters. The stream of bytes is assumed to be in MATLAB's default character encoding scheme. Return value `unicodestr` is a `char` vector that has the same general array shape as `bytes`.

`unicodestr = native2unicode(bytes, encoding)` does the conversion with the assumption that the byte stream is in the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. For common names and aliases, see the Web site http://www.iana.org/assignments/character-sets. If `encoding` is unspecified or is the empty string (`''`), MATLAB's default encoding scheme is used.

Note If `bytes` is a `char` vector, it is returned unchanged.

Examples

This example begins with a vector of bytes in an unknown character encoding scheme. The user-written function `detect_encoding` determines the encoding scheme. If successful, it returns the encoding scheme name or alias as a string. If unsuccessful, it throws an error. The example calls `native2unicode` to convert the bytes to Unicode characters.

```
try
 enc = detect_encoding(bytes);
 str = native2unicode(bytes, enc);
 disp(str);
```

```
catch
 rethrow(lasterror);
end
```

Note that the computer must be configured to display text in a language represented by the detected encoding scheme for the output of `disp(str)` to be correct.

See Also

`unicode2native`

Purpose

Binomial coefficient or all combinations

Syntax

```
C = nchoosek(n,k)
C = nchoosek(v,k)
```

Description

`C = nchoosek(n,k)` where `n` and `k` are nonnegative integers, returns $n!/((n–k)!\ k!)$. This is the number of combinations of n things taken k at a time.

`C = nchoosek(v,k)`, where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible combinations of the n elements of `v` taken k at a time. Matrix `C` contains $n!/((n–k)!\ k!)$ rows and k columns.

Inputs `n`, `k`, and `v` support classes of `float double` and `float single`.

Examples

The command `nchoosek(2:2:10,4)` returns the even numbers from two to ten, taken four at a time:

```
     2     4     6     8
     2     4     6    10
     2     4     8    10
     2     6     8    10
     4     6     8    10
```

Limitations

When `C = nchoosek(n,k)` has a large coefficient, a warning will be produced indicating possible inexact results. In such cases, the result is only accurate to 15 digits for double-precision inputs, or 8 digits for single-precision inputs.

`C = nchoosek(v,k)` is only practical for situations where `n` is less than about 15.

See Also

`perms`

Purpose

Generate arrays for N-D functions and interpolation

Syntax

```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
[X1,X2,...] = ndgrid(x)
```

Description

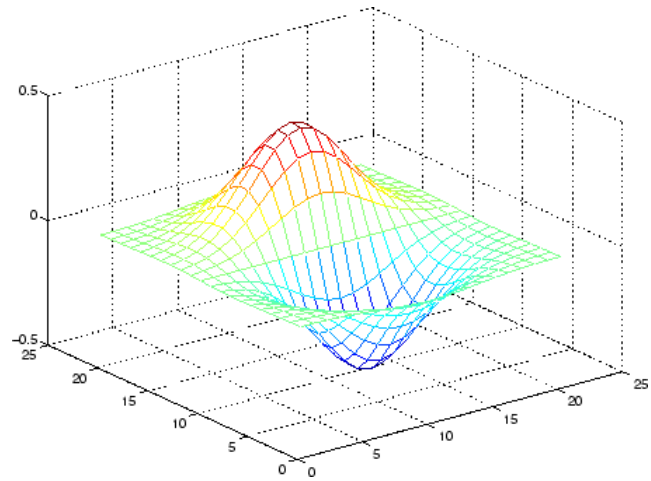
`[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)` transforms the domain specified by vectors `x1,x2,x3...` into arrays `X1,X2,X3...` that can be used for the evaluation of functions of multiple variables and multidimensional interpolation. The `i`th dimension of the output array `Xi` are copies of elements of the vector `xi`.

`[X1,X2,...] = ndgrid(x)` is the same as `[X1,X2,...] = ndgrid(x,x,...)`.

Examples

Evaluate the function $x_1 e^{-x_1^2 - x_2^2}$ over the range $-2 < x_1 < 2, -2 < x_2 < 2$.

```
[X1,X2] = ndgrid(-2:.2:2, -2:.2:2);
Z = X1 .* exp(-X1.^2 - X2.^2);
mesh(Z)
```



Remarks

The `ndgrid` function is like `meshgrid` except that the order of the first two input arguments are switched. That is, the statement

```
[X1,X2,X3] = ndgrid(x1,x2,x3)
```

produces the same result as

```
[X2,X1,X3] = meshgrid(x2,x1,x3)
```

Because of this, `ndgrid` is better suited to multidimensional problems that aren't spatially based, while `meshgrid` is better suited to problems in two- or three-dimensional Cartesian space.

See Also

`meshgrid`, `interpn`

ndims

Purpose

Number of array dimensions

Syntax

```
n = ndims(A)
```

Description

`n = ndims(A)` returns the number of dimensions in the array `A`. The number of dimensions in an array is always greater than or equal to `2`. Trailing singleton dimensions are ignored. A singleton dimension is any dimension for which `size(A,dim) = 1`.

Algorithm

`ndims(x)` is `length(size(x))`.

See Also

`size`

Purpose

Test for inequality

Syntax

```
A ~= B
ne(A, B)
```

Description

`A ~= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` and `B` are unequal, or logical 0 (`false`) where they are equal. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then MATLAB returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`ne(A, B)` is called for the syntax `A ~= B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are not equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(magic(3), 2, 2);

A ~= B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     1     1
     0     1     1     1     1     1
     1     0     1     1     1     1
```

```
     0     1     1     1     1     1
```

See Also

eq, le, ge, lt, gt, relational operators

Purpose

Compare `MException` objects for inequality

Syntax

```
eObj1 ~= eObj2
```

Description

`eObj1 ~= eObj2` tests `MException` objects `eObj1` and `eObj2` for inequality, returning logical 1 (`true`) if the two objects are not identical, otherwise returning logical 0 (`false`).

See Also

`try`, `catch`, , `error`, `assert`, `MException`, `isequal(MException)`, `eq(MException)`, `getReport(MException)`, `disp(MException)`, `throw(MException)`, `rethrow(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `last(MException)`

newplot

Purpose

Determine where to draw graphics objects

Syntax

```
newplot
h = newplot
h = newplot(hsave)
```

Description

`newplot` prepares a figure and axes for subsequent graphics commands.

`h = newplot` prepares a figure and axes for subsequent graphics commands and returns a handle to the current axes.

`h = newplot(hsave)` prepares and returns an axes, but does not delete any objects whose handles appear in `hsave`. If `hsave` is specified, the figure and axes containing `hsave` are prepared for plotting instead of the current axes of the current figure. If `hsave` is empty, `newplot` behaves as if it were called without any inputs.

Remarks

Use `newplot` at the beginning of high-level graphics M-files to determine which figure and axes to target for graphics output. Calling `newplot` can change the current figure and current axes. Basically, there are three options when you are drawing graphics in existing figures and axes:

- Add the new graphics without changing any properties or deleting any objects.
- Delete all existing objects whose handles are not hidden before drawing the new objects.
- Delete all existing objects regardless of whether or not their handles are hidden, and reset most properties to their defaults before drawing the new objects (refer to the following table for specific information).

The figure and axes `NextPlot` properties determine how `newplot` behaves. The following two tables describe this behavior with various property values.

First, `newplot` reads the current figure's `NextPlot` property and acts accordingly.

NextPlot	What Happens
`new`	Create a new figure and use it as the current figure.
`add`	Draw to the current figure without clearing any graphics objects already present.
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on` and reset figure `NextPlot` property to `add`. This clears the current figure and is equivalent to issuing the `clf` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset figure properties to their defaults, except `NextPlot` is reset to `add` regardless of user-defined defaults. • `Position`, `Units`, `PaperPosition`, and `PaperUnits` are not reset. This clears and resets the current figure and is equivalent to issuing the `clf reset` command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

NextPlot	Description
`add`	Draw into the current axes, retaining all graphics objects already present.

NextPlot	Description
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on`, but do not reset axes properties. This clears the current axes like the `cla` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset axes properties to their defaults, except `Position` and `Units`. This clears and resets the current axes like the `cla reset` command.

See Also

`axes`, `cla`, `clf`, `figure`, `hold`, `ishold`, `reset`

The `NextPlot` property for figure and axes graphics objects

“Figure Windows” on page 1-95 for related functions

Controlling Graphics Output for more examples.

Purpose

Next higher power of 2

Syntax

```
p = nextpow2(A)
```

Description

`p = nextpow2(A)` returns the smallest power of two that is greater than or equal to the absolute value of `A`. (That is, `p` that satisfies `2^p >= abs(A)`).

This function is useful for optimizing FFT operations, which are most efficient when sequence length is an exact power of two.

If `A` is non-scalar, `nextpow2` returns the smallest power of two greater than or equal to `length(A)`.

Examples

For any integer `n` in the range from `513` to `1024`, `nextpow2(n)` is `10`.

For a 1-by-30 vector `A`, `length(A)` is `30` and `nextpow2(A)` is `5`.

See Also

`fft`, `log2`, `pow2`

nnz

Purpose

Number of nonzero matrix elements

Syntax

```
n = nnz(X)
```

Description

`n = nnz(X)` returns the number of nonzero elements in matrix `X`.

The density of a sparse matrix is `nnz(X)/prod(size(X))`.

Examples

The matrix

```
w = sparse(wilkinson(21));
```

is a tridiagonal matrix with 20 nonzeros on each of three diagonals, so `nnz(w) = 60`.

See Also

`find`, `isa`, `nonzeros`, `nzmax`, `size`, `whos`

Purpose

Change `EraseMode` of all objects to `normal`

Syntax

```
noanimate(state,fig_handle)
noanimate(state)
```

Description

`noanimate(state,fig_handle)` sets the `EraseMode` of all image, line, patch, surface, and text graphics objects in the specified figure to `normal`. `state` can be the following strings:

- `'save'` — Set the values of the `EraseMode` properties to `normal` for all the appropriate objects in the designated figure.
- `'restore'` — Restore the `EraseMode` properties to the previous values (i.e., the values before calling `noanimate` with the `'save'` argument).

`noanimate(state)` operates on the current figure.

`noanimate` is useful if you want to print the figure to a TIFF or JPEG format.

See Also

`print`

"Animation" on page 1-91 for related functions

nonzeros

Purpose

Nonzero matrix elements

Syntax

```
s = nonzeros(A)
```

Description

`s = nonzeros(A)` returns a full column vector of the nonzero elements in `A`, ordered by columns.

This gives the `s`, but not the `i` and `j`, from `[i,j,s] = find(A)`. Generally,

```
length(s) = nnz(A) <= nzmax(A) <= prod(size(A))
```

See Also

`find`, `isa`, `nnz`, `nzmax`, `size`, `whos`

Purpose

Vector and matrix norms

Syntax

```
n = norm(A)
n = norm(A,p)
```

Description

The *norm* of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. The `norm` function calculates several different types of matrix norms:

`n = norm(A)` returns the largest singular value of `A`, `max(svd(A))`.

`n = norm(A,p)` returns a different kind of norm, depending on the value of `p`.

If `p` is...	Then `norm` returns...
`1`	The 1-norm, or largest column sum of `A`, `max(sum(abs(A))`.
`2`	The largest singular value (same as `norm(A)`).
`inf`	The infinity norm, or largest row sum of `A`, `max(sum(abs(A')))`.
`'fro'`	The Frobenius-norm of matrix `A`, `sqrt(sum(diag(A'*A)))`.

When `A` is a vector:

`norm(A,p)`	Returns `sum(abs(A).^p)^(1/p)`, for any 1 <= p <= ∞.
`norm(A)`	Returns `norm(A,2)`.
`norm(A,inf)`	Returns `max(abs(A))`.
`norm(A,-inf)`	Returns `min(abs(A))`.

Remarks

Note that `norm(x)` is the Euclidean length of a vector `x`. On the other hand, MATLAB uses “length” to denote the number of elements `n` in a vector. This example uses `norm(x)/sqrt(n)` to obtain the root-mean-square (RMS) value of an `n`-element vector `x`.

```
x = [0 1 2 3]
x =
     0     1     2     3

sqrt(0+1+4+9)   % Euclidean length
ans =
    3.7417

norm(x)
ans =
    3.7417

n = length(x)   % Number of elements
n =
     4

rms = 3.7417/2  % rms = norm(x)/sqrt(n)
rms =
    1.8708
```

See Also

cond, condest, hypot, normest, rcond

Purpose

2-norm estimate

Syntax

```
nrm = normest(S)
nrm = normest(S,tol)
[nrm,count] = normest(...)
```

Description

This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.

`nrm = normest(S)` returns an estimate of the 2-norm of the matrix `S`.

`nrm = normest(S,tol)` uses relative error `tol` instead of the default tolerance `1.e-6`. The value of `tol` determines when the estimate is considered acceptable.

`[nrm,count] = normest(...)` returns an estimate of the 2-norm and also gives the number of power iterations used.

Examples

The matrix `W = gallery('wilkinson',101)` is a tridiagonal matrix. Its order, 101, is small enough that `norm(full(W))`, which involves `svd(full(W))`, is feasible. The computation takes 4.13 seconds (on one computer) and produces the exact norm, 50.7462. On the other hand, `normest(sparse(W))` requires only 1.56 seconds and produces the estimated norm, 50.7458.

Algorithm

The power iteration involves repeated multiplication by the matrix `S` and its transpose, `S'`. The iteration is carried out until two successive estimates agree to within the specified relative tolerance.

See Also

`cond`, `condest`, `norm`, `rcond`, `svd`

Purpose

Find logical NOT of array or scalar input

Syntax

```
~A
not(A)
```

Description

`~A` performs a logical NOT of input array `A`, and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if the input array contains a zero value element at that same array location. Otherwise, that element is set to 0.

The input of the expression can be an array or can be a scalar value. If the input is an array, then the output is an array of the same dimensions. If the input is scalar, then the output is scalar.

`not(A)` is called for the syntax `~A` when `A` is an object.

Example

If matrix `A` is

```
     0    29     0    36     0
    23    34    35     0    39
     0    24    31    27     0
     0    29     0     0    34
```

then

```
~A
ans =
     1     0     1     0     1
     0     0     0     1     0
     1     0     0     0     1
     1     0     1     1     0
```

See Also

`bitcmp`, `and`, `or`, `xor`, `any`, `all`, “Logical Operators”, “Logical Types”, “Bit-Wise Functions”

Purpose

Open M-book in Microsoft Word (Windows)

Syntax

```
notebook
notebook('filename')
notebook('-setup')
```

Description

`notebook` starts Microsoft Word and creates a new M-book titled `Document 1`.

`notebook('filename')` starts Microsoft Word and opens the M-book `filename`, where filename is either in the MATLAB current directory or is a full pathname. If `filename` does not exist, MATLAB creates a new M-book titled `filename`. If the `filename` extension is not specified, MATLAB assumes `.doc`.

`notebook('-setup')` runs an interactive setup function for Notebook. It copies the Notebook template, `m-book.dot`, to the Microsoft Word template directory, whose location MATLAB automatically determines from the Windows system registry. Upon completion, MATLAB displays a message indicating whether or not the setup was successful.

See Also

MATLAB Desktop Tools and Development Environment documentation

- Notebook for Publishing to Word
- "Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells"

Purpose

Current date and time

Syntax

```
t = now
```

Description

`t = now` returns the current date and time as a serial date number. To return the time only, use `rem(now,1)`. To return the date only, use `floor(now)`.

Examples

```
t1 = now, t2 = rem(now,1)

t1 =

   7.2908e+05


t2 =

    0.4013
```

See Also

`clock`, `date`, `datenum`

Purpose

Real nth root of real numbers

Syntax

```
y = nthroot(X, n)
```

Description

`y = nthroot(X, n)` returns the real nth root of the elements of `X`. Both `X` and `n` must be real and `n` must be a scalar. If `X` has negative entries, `n` must be an odd integer.

Example

```
nthroot(-2, 3)
```

returns the real cube root of `-2`.

```
ans =

   -1.2599
```

By comparison,

```
(-2)^(1/3)
```

returns a complex cube root of `-2`.

```
ans =

   0.6300 + 1.0911i
```

See Also

`power`

Purpose

Null space

Syntax

```
Z = null(A)
Z = null(A,'r')
```

Description

`Z = null(A)` is an orthonormal basis for the null space of `A` obtained from the singular value decomposition. That is, `A*Z` has negligible elements, `size(Z,2)` is the nullity of `A`, and `Z'*Z = I`.

`Z = null(A,'r')` is a "rational" basis for the null space obtained from the reduced row echelon form. `A*Z` is zero, `size(Z,2)` is an estimate for the nullity of `A`, and, if `A` is a small matrix with integer elements, the elements of the reduced row echelon form (as computed using `rref`) are ratios of small integers.

The orthonormal basis is preferable numerically, while the rational basis may be preferable pedagogically.

Example

Example 1

Compute the orthonormal basis for the null space of a matrix A.

```
A = [1     2     3
     1     2     3
     1     2     3];

Z = null(A);
A*Z

ans =
  1.0e-015 *
     0.2220    0.2220
     0.2220    0.2220
     0.2220    0.2220

Z'*Z

ans =
```

```
    1.0000   -0.0000
   -0.0000    1.0000
```

Example 2

Compute the 1-norm of the matrix A*Z and determine that it is within a small tolerance.

```
norm(A*Z,1) < 1e-12
ans =
    1
```

Example 3

Compute the rational basis for the null space of the same matrix `A`.

```
ZR = null(A,'r')

ZR =
    -2    -3
     1     0
     0     1

A*ZR

ans =

     0     0
     0     0
     0     0
```

See Also

`orth`, `rank`, `rref`, `svd`

num2cell

Purpose

Convert numeric array to cell array

Syntax

```
c = num2cell(A)
c = num2cell(A, dims)
```

Description

`c = num2cell(A)` converts the matrix `A` into a cell array by placing each element of `A` into a separate cell. Cell array `c` will be the same size as matrix `A`.

`c = num2cell(A, dims)` converts the matrix `A` into a cell array by placing the dimensions specified by `dims` into separate cells. `C` will be the same size as `A` except that the dimensions matching `dims` will be `1`.

Examples

The statement

```
num2cell(A,2)
```

places the rows of `A` into separate cells. Similarly

```
num2cell(A,[1 3])
```

places the column-depth pages of `A` into separate cells.

See Also

`cat`, `mat2cell`, `cell2mat`

Purpose

Convert singles and doubles to IEEE hexadecimal strings

Syntax

```
num2hex(X)
```

Description

If `X` is a single or double precision array with n elements, `num2hex(X)` is an n-by-8 or n-by-16 char array of the hexadecimal floating-point representation. The same representation is printed with format `hex`.

Examples

```
num2hex([1 0 0.1 -pi Inf NaN])
```

returns

```
ans =

3ff0000000000000
0000000000000000
3fb999999999999a
c00921fb54442d18
7ff0000000000000
fff8000000000000
num2hex(single([1 0 0.1 -pi Inf NaN]))
```

returns

```
ans =

3f800000
00000000
3dcccccd
c0490fdb
7f800000
ffc00000
```

See Also

`hex2num`, `dec2hex`, `format`

Purpose

Convert number to string

Syntax

```
str = num2str(A)
str = num2str(A, precision)
str = num2str(A, format)
```

Description

The num2str function converts numbers to their string representations. This function is useful for labeling and titling plots with numeric values.

`str = num2str(A)` converts array `A` into a string representation `str` with roughly four digits of precision and an exponent if required.

`str = num2str(A, precision)` converts the array `A` into a string representation `str` with maximum precision specified by *`precision`*. Argument *`precision`* specifies the number of digits the output string is to contain. The default is four.

`str = num2str(A, format)` converts array `A` using the supplied *`format`*. (See `fprintf` for format string details.) By default, `num2str` displays floating point values using `'%11.4g'` format (four significant digits in exponential or fixed-point notation, whichever is shorter).

If the input array is integer-valued, `num2str` returns the exact string representation of that integer. The term integer-valued includes large floating-point numbers that lose precision due to limitations of the hardware.

`num2str` removes any leading spaces from the output string. Thus, `num2str(42.67, '%10.2f')` returns a 1-by-5 character array `'42.67'`.

Examples

`num2str(pi)` is `3.142`.

`num2str(eps)` is `2.22e-16`.

`num2str` with a format of `%10.5e\n` returns a matrix of strings in exponential format, having 5 decimal places, with each element separated by a `newline` character:

```
x = rand(3) * 9999;              % Create a 2-by-3 matrix.
x(3,:) = [];
```

```
A = num2str(x, '%10.5e\n')       % Convert to string array.
A =
6.87255e+003
1.55597e+003
8.55890e+003

3.46077e+003
1.91097e+003
4.90201e+003
```

See Also

mat2str, int2str, str2num, sprintf, fprintf

numel

Purpose

Number of elements in array or subscripted array expression

Syntax

```
n = numel(A)
n = numel(A, index1, index2, ... indexn)
```

Description

`n = numel(A)` returns the number of elements, `n`, in array `A`.

`n = numel(A, index1, index2, ... indexn)` returns the number of subscripted elements, `n`, in `A(index1, index2, ..., indexn)`. To handle the variable number of arguments, `numel` is typically written with the header `function n = numel(A, varargin)`, where `varargin` is a cell array with elements `index1`, `index2`, `... indexn`.

MATLAB implicitly calls the `numel` built-in function whenever an expression generates a comma-separated list. This includes brace indexing (i.e., `A{index1,index2,...,indexN}`), and dot indexing (i.e., `A.fieldname`).

Remarks

It is important to note the significance of `numel` with regards to the overloaded `subsref` and `subsasgn` functions. In the case of the overloaded `subsref` function for brace and dot indexing (as described in the last paragraph), `numel` is used to compute the number of expected outputs (`nargout`) returned from `subsref`. For the overloaded `subsasgn` function, `numel` is used to compute the number of expected inputs (`nargin`) to be assigned using `subsasgn`. The `nargin` value for the overloaded `subsasgn` function is the value returned by `numel` plus 2 (one for the variable being assigned to, and one for the structure array of subscripts).

As a class designer, you must ensure that the value of `n` returned by the built-in `numel` function is consistent with the class design for that object. If `n` is different from either the `nargout` for the overloaded `subsref` function or the `nargin` for the overloaded `subsasgn` function, then you need to overload `numel` to return a value of `n` that is consistent with the class' `subsref` and `subsasgn` functions. Otherwise, MATLAB produces errors when calling these functions.

Examples

Create a 4-by-4-by-2 matrix. `numel` counts 32 elements in the matrix.

```
a = magic(4);
a(:,:,2) = a'

a(:,:,1) =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

a(:,:,2) =
    16     5     9     4
     2    11     7    14
     3    10     6    15
    13     8    12     1

numel(a)
ans =
    32
```

See Also

`nargin`, `nargout`, `prod`, `size`, `subsasgn`, `subsref`

Purpose

Amount of storage allocated for nonzero matrix elements

Syntax

```
n = nzmax(S)
```

Description

`n = nzmax(S)` returns the amount of storage allocated for nonzero elements.

If `S` is a sparse matrix...	`nzmax(S)` is the number of storage locations allocated for the nonzero elements in `S`.
If `S` is a full matrix...	`nzmax(S) = prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i,j,s,m,n,nzmax)` or its simpler form, `spalloc(m,n,nzmax)`, can set `nzmax` in anticipation of later fill-in.

See Also

`find`, `isa`, `nnz`, `nonzeros`, `size`, `whos`

Purpose

Solve fully implicit differential equations, variable order method

Syntax

```
[T,Y] = ode15i(odefun,tspan,y0,yp0)
[T,Y] = ode15i(odefun,tspan,y0,yp0,options)
[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)
sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)
```

Arguments

The following table lists the input arguments for `ode15i`.

`odefun`	A function handle that evaluates the left side of the differential equations, which are of the form $f(t, y, y') = 0$. See "Function Handles" in the MATLAB Programming documentation for more information.
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0, yp0`	Vectors of initial conditions for y and y' respectively.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for `ode15i`.

`T`	Column vector of time points
`Y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

Description

`[T,Y] = ode15i(odefun,tspan,y0,yp0)` with `tspan = [t0 tf]` integrates the system of differential equations $f(t, y, y') = 0$ from time `t0` to `tf` with initial conditions `y0` and `yp0`. `odefun` is a function handle. Function `ode15i` solves ODEs and DAEs of index 1. The initial conditions must be consistent, meaning that $f(t0, y0, yp0) = 0$. You can use the function `decic` to compute consistent initial conditions

close to guessed values. Function `odefun(t,y,yp)`, for a scalar `t` and column vectors `y` and `yp`, must return a column vector corresponding to $f(t, y, y')$. Each row in the solution array `Y` corresponds to a time returned in the column vector `T`. To obtain solutions at specific times `t0,t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

`[T,Y] = ode15i(odefun,tspan,y0,yp0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used options include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components `1e-6` by default). See `odeset` for details.

`[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)` with the `'Events'` property in `options` set to a function `events`, solves as above while also finding where functions of (t, y, y'), called event functions, are zero. The function `events` is of the form `[value,isterminal,direction] = events(t,y,yp)` and includes the necessary event functions. Code the function `events` so that the `i`th element of each output vector corresponds to the `i`th event. For the `i`th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Output `TE` is a column vector of times at which events occur. Rows of `YE` are the corresponding solutions, and indices in vector `IE` specify

which event occurred. See “Changing ODE Integration Properties” in the MATLAB Mathematics documentation for more information.

`sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)` returns a structure that can be used with `deval` to evaluate the solution at any point between `t0` and `tfinal`. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver. If you specify the `Events` option and a terminal event is detected, `sol.x(end)` contains the end of the step at which the event occurred.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Options

`ode15i` accepts the following parameters in `options`. For more information, see `odeset` and Changing ODE Integration Properties in the MATLAB documentation.

Error control	`RelTol`, `AbsTol`, `NormControl`
Solver output	`OutputFcn`, `OutputSel`, `Refine`, `Stats`
Event location	`Events`

Step size	`MaxStep`, `InitialStep`
Jacobian matrix	`Jacobian`, `JPattern`, `Vectorized`

Solver Output

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

Jacobian Matrices

The Jacobian matrices $\partial f / \partial y$ and $\partial f / \partial y'$ are critical to reliability and efficiency. You can provide these matrices as one of the following:

- Function of the form `[dfdy,dfdyp] = FJAC(t,y,yp)` that computes the Jacobian matrices. If `FJAC` returns an empty matrix `[]` for either `dfdy` or `dfdyp`, then `ode15i` approximates that matrix by finite differences.
- Cell array of two constant matrices `{dfdy,dfdyp}`, either of which could be empty.

Use `odeset` to set the `Jacobian` option to the function or cell array. If you do not set the `Jacobian` option, `ode15i` approximates both Jacobian matrices by finite differences.

For `ode15i`, `Vectorized` is a two-element cell array. Set the first element to `'on'` if `odefun(t,[y1,y2,...],yp)` returns

`[odefun(t,y1,yp),odefun(t,y2,yp),...]`. Set the second element to `'on'` if `odefun(t,y,[yp1,yp2,...])` returns `[odefun(t,y,yp1),odefun(t,y,yp2),...]`. The default value of `Vectorized` is `{'off','off'}`.

For `ode15i`, `JPattern` is also a two-element sparse matrix cell array. If $\partial f/\partial y$ or $\partial f/\partial y'$ is a sparse matrix, set `JPattern` to the sparsity patterns, `{SPDY,SPDYP}`. A sparsity pattern of $\partial f/\partial y$ is a sparse matrix `SPDY` with `SPDY(i,j) = 1` if component `i` of `f(t,y,yp)` depends on component `j` of `y`, and `0` otherwise. Use `SPDY = []` to indicate that $\partial f/\partial y$ is a full matrix. Similarly for $\partial f/\partial y'$ and `SPDYP`. The default value of `JPattern` is `{[],[]}`.

Examples

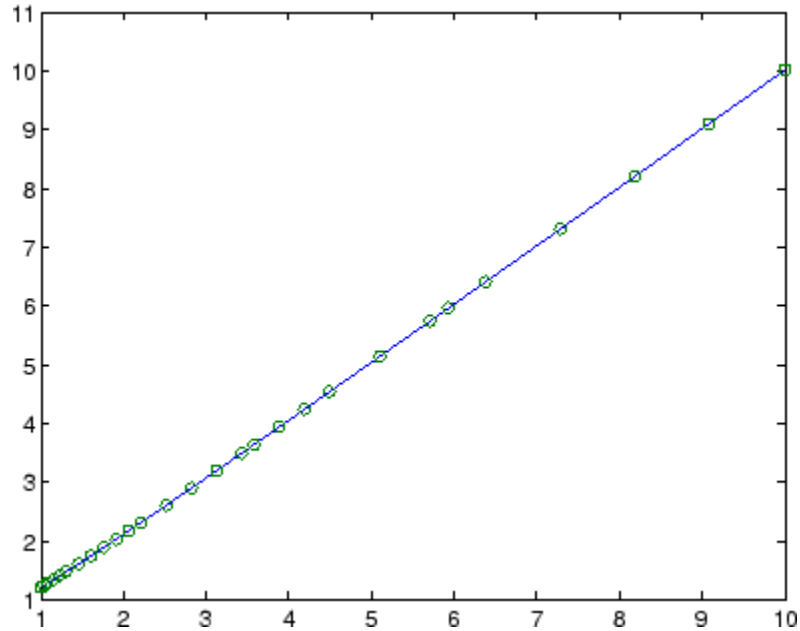
Example 1

This example uses a helper function `decic` to hold fixed the initial value for $y(t_0)$ and compute a consistent initial value for $y'(t_0)$ for the Weissinger implicit ODE. The Weissinger function evaluates the residual of the implicit ODE.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

The example uses `ode15i` to solve the ODE, and then plots the numerical solution against the analytical solution.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,t,ytrue,'o');
```



Other Examples

These demos provide examples of implicit ODEs: `ihb1dae`, `iburgersode`.

See Also

`decic`, `deval`, `odeget`, `odeset`, `function_handle` (`@`)

Other ODE initial value problem solvers: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`

ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb

Purpose

Solve initial value problems for ordinary differential equations

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

where *solver* is one of `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. See "Function Handles" in the MATLAB Programming documentation for more information. All solvers solve systems of equations in the form $y' = f(t, y)$ or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.

`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument.
	
	`[t,y] =`
	`solver(odefun,tspan,y0,options)`
	
	You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points
`Y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t, y)$ from time `t0` to `tf`

with initial conditions `y0`. `odefun` is a function handle. See Function Handles in the MATLAB Programming documentation for more information. Function `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t, y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0, t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of

`OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property 'on' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t, y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t, y)y' = f(t, y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to 'none'.
- If the mass matrix depends weakly on y, set `MStateDependence` to 'weak' (the default); otherwise, set it to 'strong'. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t, y)$.

- Supply the sparsity pattern of $\partial f / \partial y$ using the `JPattern` property or a sparse $\partial f / \partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t, y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t, y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t, y)y' = f(t, y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0, y_0)yp_0 = f(t_0, y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 2-2308 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Changing ODE Integration Properties” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the NonNegative parameter with ode15s, ode23t, and ode23tb only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$
\begin{aligned}
y'_1 &= y_2\, y_3 & y_1(0) &= 0 \\
y'_2 &= -y_1\, y_3 & y_2(0) &= 1 \\
y'_3 &= -0.51\, y_1\, y_2 & y_3(0) &= 1
\end{aligned}
$$

To simulate this system, create a function `rigid` containing the equations

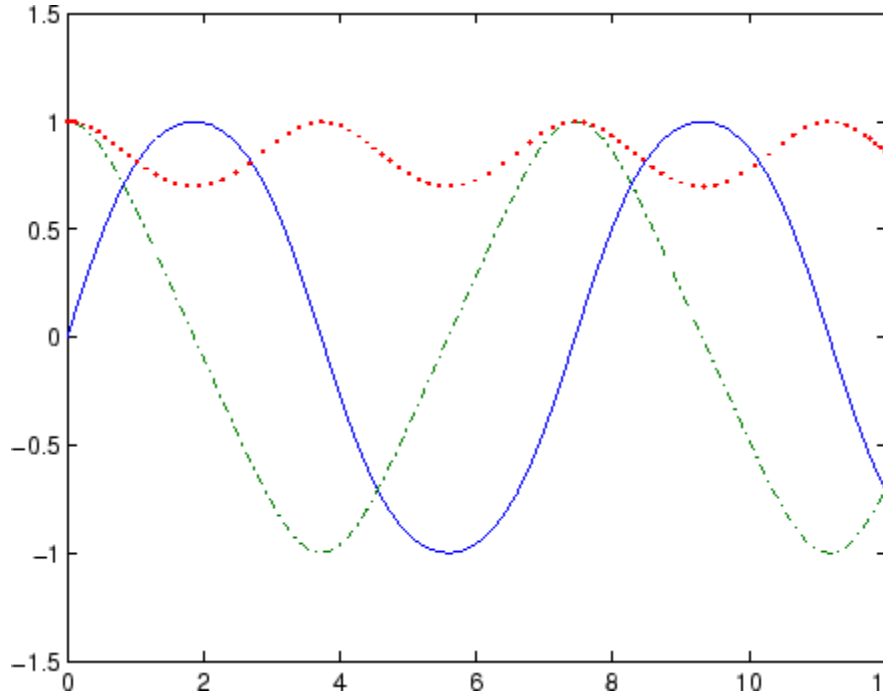
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y'_1 = y_2 \qquad y_1(0) = 0$$
$$y'_2 = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 1$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
dy = zeros(2,1);    % a column vector
```

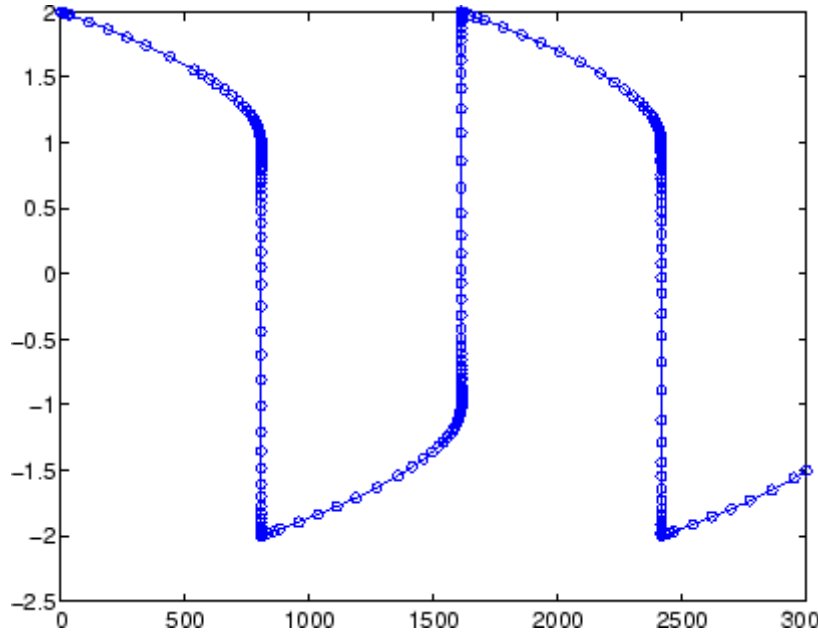
```
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write an M-file function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

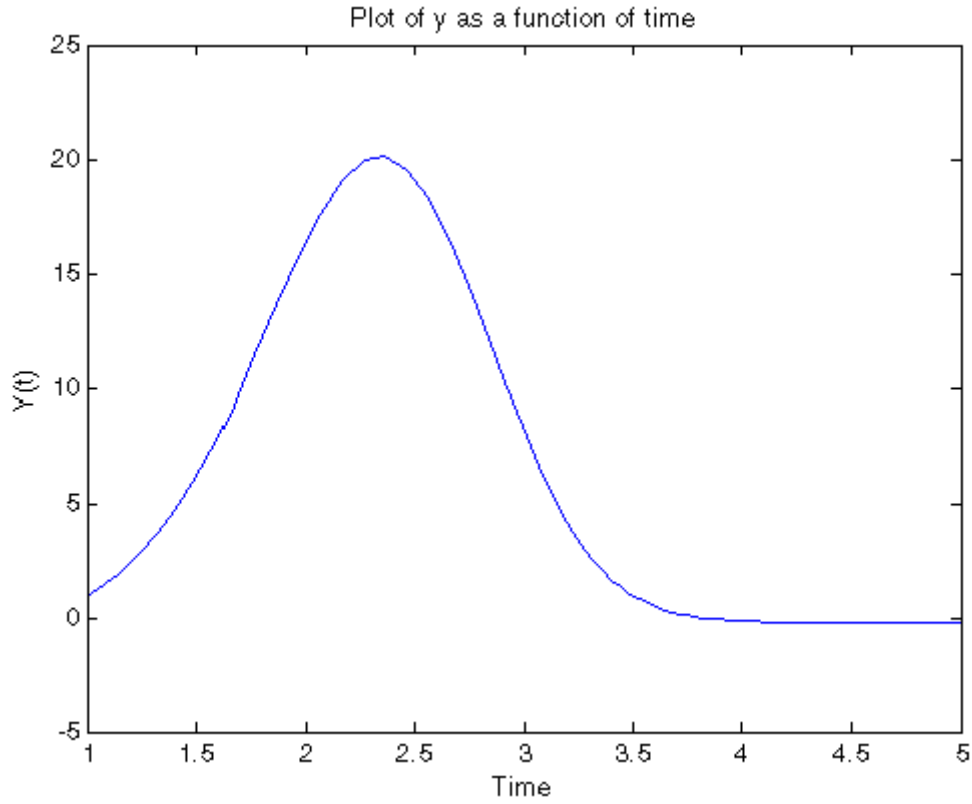
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),TSPAN,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

See Also

`deval`, `ode15i`, `odeget`, `odeset`, `function_handle (@)`

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp 436-451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp 1-9.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp 19-26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp 1-22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp 538-552.

Purpose

Define differential equation problem for ordinary differential equation solvers

---

Note This reference page describes the `odefile` and the syntax of the ODE solvers used in MATLAB, Version 5. MATLAB, Version 6, supports the `odefile` for backward compatibility, however the new solver syntax does not use an ODE file. New functionality is available only with the new syntax. For information about the new syntax, see `odeset` or any of the ODE solvers.

Description

`odefile` is not a command or function. It is a help entry that describes how to create an M-file defining the system of equations to be solved. This definition is the first step in using any of the MATLAB ODE solvers. In MATLAB documentation, this M-file is referred to as an `odefile`, although you can give your M-file any name you like.

You can use the `odefile` M-file to define a system of differential equations in one of these forms

$$y' = f(t, y)$$

or

$$M(t, y)y' = f(t, y)v$$

where:

- t is a scalar independent variable, typically representing time.
- y is a vector of dependent variables.
- f is a function of t and y returning a column vector the same length as y.
- $M(t, y)$ is a time-and-state-dependent mass matrix.

The ODE file must accept the arguments `t` and `y`, although it does not have to use them. By default, the ODE file must return a column vector the same length as `y`.

All of the solvers of the ODE suite can solve $M(t, y)y' = f(t, y)$, except `ode23s`, which can only solve problems with constant mass matrices. The `ode15s` and `ode23t` solvers can solve some differential-algebraic equations (DAEs) of the form $M(t)y' = f(t, y)$.

Beyond defining a system of differential equations, you can specify an entire initial value problem (IVP) within the ODE M-file, eliminating the need to supply time and initial value vectors at the command line (see "Examples" on page 2-2314).

To Use the ODE File Template

- Enter the command `help odefile` to display the help entry.
- Cut and paste the ODE file text into a separate file.
- Edit the file to eliminate any cases not applicable to your IVP.
- Insert the appropriate information where indicated. The definition of the ODE system is required information.

```
switch flag
 case ''  % Return dy/dt = f(t,y).
   varargout{1} = f(t,y,p1,p2);
 case 'init'  % Return default [tspan,y0,options].
   [varargout{1:3}] = init(p1,p2);
 case 'jacobian'  % Return Jacobian matrix df/dy.
   varargout{1} = jacobian(t,y,p1,p2);
 case 'jpattern'  % Return sparsity pattern matrix S.
   varargout{1} = jpattern(t,y,p1,p2);
 case 'mass'  % Return mass matrix.
   varargout{1} = mass(t,y,p1,p2);
 case 'events'  % Return [value,isterminal,direction].
   [varargout{1:3}] = events(t,y,p1,p2);
 otherwise
   error(['Unknown flag ''' flag '''.']);
```

```
 end
% --------------------------------------------------------
function dydt = f(t,y,p1,p2)
 dydt = Insert a function of t and/or y, p1, and p2 here.>
% --------------------------------------------------------
function [tspan,y0,options] = init(p1,p2)
 tspan = <Insert tspan here.>;
 y0 = <Insert y0 here.>;
 options = <Insert options = odeset(...) or [] here.>;
% --------------------------------------------------------
function dfdy = jacobian(t,y,p1,p2)
 dfdy = <Insert Jacobian matrix here.>;
% --------------------------------------------------------
function S = jpattern(t,y,p1,p2)
 S = <Insert Jacobian matrix sparsity pattern here.>;
% --------------------------------------------------------
function M = mass(t,y,p1,p2)
 M = <Insert mass matrix here.>;
% --------------------------------------------------------
function [value,isterminal,direction] = events(t,y,p1,p2)
 value = <Insert event function vector here.>
 isterminal = <Insert logical ISTERMINAL vector here.>;
 direction = <Insert DIRECTION vector here.>;
```

Notes

1 The ODE file must accept `t` and `y` vectors from the ODE solvers and must return a column vector the same length as `y`. The optional input argument `flag` determines the type of output (mass matrix, Jacobian, etc.) returned by the ODE file.

2 The solvers repeatedly call the ODE file to evaluate the system of differential equations at various times. *This is required information* – you must define the ODE system to be solved.

3 The `switch` statement determines the type of output required, so that the ODE file can pass the appropriate information to the solver. (See notes 4 - 9.)

4 In the default *initial conditions (*`'init'`*)* case, the ODE file returns basic information (time span, initial conditions, options) to the solver. If you omit this case, you must supply all the basic information on the command line.

5 In the `'jacobian'` case, the ODE file returns a Jacobian matrix to the solver. You need only provide this case when you want to improve the performance of the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`.

6 In the `'jpattern'` case, the ODE file returns the Jacobian sparsity pattern matrix to the solver. You need to provide this case only when you want to generate sparse Jacobian matrices numerically for a stiff solver.

7 In the `'mass'` case, the ODE file returns a mass matrix to the solver. You need to provide this case only when you want to solve a system in the form $M(t, y)y' = f(t, y)$.

8 In the `'events'` case, the ODE file returns to the solver the values that it needs to perform event location. When the `Events` property is set to `on`, the ODE solvers examine any elements of the `event` vector for transitions to, from, or through zero. If the corresponding element of the logical `isterminal` vector is set to `1`, integration will halt when a zero-crossing is detected. The elements of the `direction` vector are `-1`, `1`, or `0`, specifying that the corresponding event must be decreasing, increasing, or that any crossing is to be detected.

9 An unrecognized `flag` generates an error.

Examples

The van der Pol equation, $y''_1 - \mu(1 - y_1^2)y' + y_1 = 0$, is equivalent to a system of coupled first-order differential equations.

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

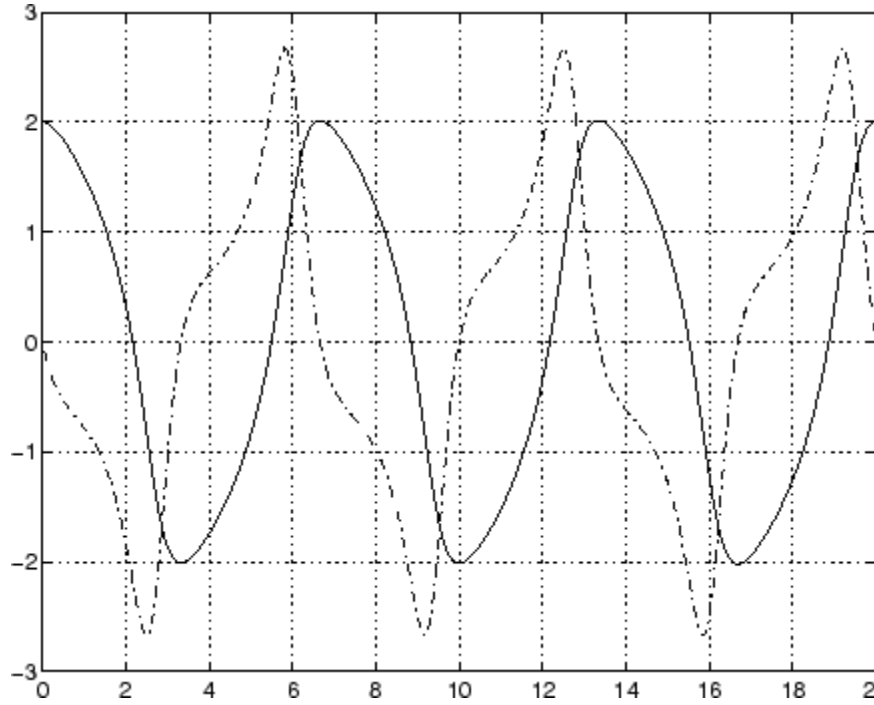
The M-file

```
function out1 = vdp1(t,y)
out1 = [y(2); (1-y(1)^2)*y(2) - y(1)];
```

defines this system of equations (with $\mu = 1$).

To solve the van der Pol system on the time interval `[0 20]` with initial values (at time `0`) of `y(1) = 2` and `y(2) = 0`, use

```
[t,y] = ode45('vdp1',[0 20],[2; 0]);
plot(t,y(:,1),'-',t,y(:,2),'-.')
```



To specify the entire initial value problem (IVP) within the M-file, rewrite `vdp1` as follows.

```
function [out1,out2,out3] = vdp1(t,y,flag)
if nargin < 3 | isempty(flag)
  out1 = [y(1).*(1-y(2).^2)-y(2); y(1)];
else
  switch(flag)
    case 'init'               % Return tspan, y0, and options.
      out1 = [0 20];
      out2 = [2; 0];
      out3 = [];
  otherwise
    error(['Unknown request ''' flag '''.']);
  end
end
```

You can now solve the IVP without entering any arguments from the command line.

```
[t,Y] = ode23('vdp1')
```

In this example the `ode23` function looks to the `vdp1` M-file to supply the missing arguments. Note that, once you've called `odeset` to define `options`, the calling syntax

```
[t,Y] = ode23('vdp1',[],[],options)
```

also works, and that any options supplied via the command line override corresponding options specified in the M-file (see `odeset`).

See Also

The MATLAB Version 5 help entries for the ODE solvers and their associated functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `odeget`, `odeset`

Type at the MATLAB command line: `more on, type function, more off`. The Version 5 help follows the Version 6 help.

Purpose

Ordinary differential equation options parameters

Syntax

```
o = odeget(options,'name')
o = odeget(options,'name',default)
```

Description

`o = odeget(options,'name')` extracts the value of the property specified by string `'name'` from integrator options structure `options`, returning an empty matrix if the property value is not specified in `options`. It is only necessary to type the leading characters that uniquely identify the property name. Case is ignored for property names. The empty matrix `[]` is a valid `options` argument.

`o = odeget(options,'name',default)` returns `o = default` if the named property is not specified in `options`.

Example

Having constructed an ODE options structure,

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-3 2e-3 3e-3]);
```

you can view these property settings with `odeget`.

```
odeget(options,'RelTol')
ans =

       1.0000e-04

odeget(options,'AbsTol')
ans =

    0.0010    0.0020    0.0030
```

See Also

`odeset`

odeset

Purpose

Create or alter options structure for ordinary differential equation solvers

Syntax

```
options = odeset('name1',value1,'name2',value2,...)
options = odeset(oldopts,'name1',value1,...)
options = odeset(oldopts,newopts)
odeset
```

Description

The `odeset` function lets you adjust the integration parameters of the following ODE solvers.

For solving fully implicit differential equations:

```
ode15i
```

For solving initial value problems:

```
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
```

See below for information about the integration parameters.

`options = odeset('name1',value1,'name2',value2,...)` creates an options structure that you can pass as an argument to any of the ODE solvers. In the resulting structure, `options`, the named properties have the specified values. For example, `'name1'` has the value `value1`. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify a property name. Case is ignored for property names.

`options = odeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This sets `options` equal to the existing structure `oldopts`, overwrites any values in `oldopts` that are respecified using name/value pairs, and adds any new pairs to the structure. The modified structure is returned as an output argument.

`options = odeset(oldopts,newopts)` alters an existing options structure `oldopts` by combining it with a new options structure `newopts`. Any new options not equal to the empty matrix overwrite corresponding options in `oldopts`.

`odeset` with no input arguments displays all property names as well as their possible and default values.

ODE Properties

The following sections describe the properties that you can set using `odeset`. The available properties depend on the ODE solver you are using. There are several categories of properties:

Note This reference page describes the ODE properties for MATLAB, Version 7. The Version 5 properties are supported only for backward compatibility. For information on the Version 5 properties, type at the MATLAB command line: `more on, type odeset, more off`.

Error Control Properties

At each step, the solver estimates the local error `e` in the `i`th component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

```
|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the ODE solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For the

absolute error tolerance, the scaling of the solution components is important: if `|y|` is somewhat smaller than `AbsTol`, the solver is not constrained to obtain any correct digits in `y`. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.

The following table describes the error control properties. Further information on each property is given following the table.

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	Relative error tolerance that applies to all components of the solution vector `y`.
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the individual components of the solution vector.
`NormControl`	`on` | {`off`}	Control error relative to norm of solution.

Description of Error Control Properties

`RelTol` — This tolerance is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components, except those smaller than thresholds `AbsTol(i)`.

The default, `1e-3`, corresponds to 0.1% accuracy.

`AbsTol` — `AbsTol(i)` is a threshold below which the value of the `i`th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero.

If AbsTol is a vector, the length of AbsTol must be the same as the length of the solution vector y. If AbsTol is a scalar, the value applies to all components of y.

NormControl — Set this property on to request that the solvers control the error in each integration step with norm(e) <= max(RelTol*norm(y),AbsTol). By default the solvers use a more stringent componentwise error control.

Solver Output Properties

The following table lists the solver output properties that control the output that the solvers generate. Further information on each property is given following the table.

Property	Value	Description
NonNegative	Vector of integers	Specifies which components of the solution vector must be nonnegative. The default value is [].
OutputFcn	Function handle	A function for the solver to call after every successful integration step.
OutputSel	Vector of indices	Specifies which components of the solution vector are to be passed to the output function.
Refine	Positive integer	Increases the number of output points by a factor of Refine.
Stats	on | {off}	Determines whether the solver should display statistics about its computations. By default, Stats is off.

Description of Solver Output Properties

NonNegative — The NonNegative property is not available in ode23s, ode15i. In ode15s, ode23t, and ode23tb, NonNegative is not available for problems where there is a mass matrix.

`OutputFcn` — To specify an output function, set `'OutputFcn'` to a function handle. For example,

```
options = odeset('OutputFcn',@myfun)
```

sets `'OutputFcn'` to `@myfun`, a handle to the function `myfun`. See "Function Handles" in the MATLAB Programming documentation for more information.

The output function must be of the form

```
status = myfun(t,y,flag)
```

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to `myfun`, if necessary.

The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:

Flag	Description
`init`	The solver calls `myfun(tspan,y0,'init')` before beginning the integration to allow the output function to initialize. `tspan` and `y0` are the input arguments to the ODE solver.

Flag	Description
`{[]}`	The solver calls `status = myfun(t,y,[])` after each integration step on which output is requested. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points in `t`. If `t` is a vector, the `i`th column of `y` corresponds to the `i`th element of `t`. When `length(tspan) > 2` the output is produced at every point in `tspan`. When `length(tspan) = 2` the output is produced according to the `Refine` option. `myfun` must return a `status` output value of `0` or `1`. If `status = 1`, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.
`done`	The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores.

You can use these general purpose output functions or you can edit them to create your own. Type `help function` at the command line for more information.

- `odeplot` — Time series plotting (default when you call the solver with no output arguments and you have not specified an output function)
- `odephas2` — Two-dimensional phase plane plotting
- `odephas3` — Three-dimensional phase plane plotting
- `odeprint` — Print solution as it is computed

Note If you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history.

`OutputSel` — Use `OutputSel` to specify which components of the solution vector you want passed to the output function. For example, if

you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using

```
options = ...
odeset('OutputFcn',@odeplot,'OutputSel',[1 3]);
```

By default, the solver passes all components of the solution to the output function.

`Refine` — If `Refine` is `1`, the solver returns solutions only at the end of each time step. If `Refine` is `n` >`1`, the solver subdivides each time step into `n` smaller intervals and returns solutions at each time point. `Refine` does not apply when `length(tspan)>2`.

Note In all the solvers, the default value of `Refine` is `1`. Within `ode45`, however, the default is `4` to compensate for the solver's large step sizes. To override this and see only the time steps chosen by `ode45`, set `Refine` to `1`.

The extra values produced for `Refine` are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

`Stats` — By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays

- Number of successful steps
- Number of failed attempts
- Number of times the ODE function was called to evaluate $f(t,y)$

Solvers based on implicit methods, including `ode23s`, `ode23t`, `ode23t`, `ode15s`, and `ode15i`, also display

- Number of times that the partial derivatives matrix $\partial f / \partial x$ was formed
- Number of LU decompositions
- Number of solutions of linear systems

Step-Size Properties

The step-size properties specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step-size properties. Further information on each property is given following the table.

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size.
`MaxStep`	Positive scalar {`0.1*abs(t0-tf)`}	Upper bound on solver step size.

Description of Step-Size Properties

`InitialStep` — `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the initial step size is based on the slope of the solution at the initial time `tspan(1)`, and if the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.

`MaxStep` — If the differential equation has periodic coefficients or solutions, it might be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep` for any of the following purposes:

- To produce more output points. This can significantly slow down solution time. Instead, use `Refine` to compute additional outputs by continuous extension at very low cost.
- When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. See "Error Control Properties" on page 2-2319 for a description of the error tolerance properties.
- To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth. While solving a problem, the ODE solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property. Further information on each property is given following the table.

ODE Events Property

String	Value	Description
`Events`	Function handle	Handle to a function that includes one or more event functions.

Description of Event Location Properties

`Events` — The function is of the form

```
[value,isterminal,direction] = events(t,y)
```

`value`, `isterminal`, and `direction` are vectors for which the `i`th element corresponds to the `i`th event function:

- `value(i)` is the value of the `i`th event function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function, otherwise, `0`.
- `direction(i) = 0` if all zeros are to be located (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.

If you specify an events function and events are detected, the solver returns three additional outputs:

- `A` column vector of times at which events occur
- Solution values corresponding to these times
- Indices into the vector returned by the events function. The values indicate which event the solver detected.

If you call the solver as

```
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `TE`, `YE`, and `IE` respectively. If you call the solver as

```
sol = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `sol.xe`, `sol.ye`, and `sol.ie`, respectively.

For examples that use an event function, see “Example: Simple Event Location” and “Example: Advanced Event Location” in the MATLAB Mathematics documentation.

Jacobian Matrix Properties

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix $\partial f / \partial y$, a matrix of partial derivatives of the function that defines the differential equations.

$$\frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots \\ \vdots & \vdots & \end{bmatrix}$$

The Jacobian matrix properties pertain only to those solvers for stiff problems (`ode15s`, `ode23s`, `ode23t`, `ode23tb`, and `ode15i`) for which the Jacobian matrix $\partial f / \partial y$ can be critical to reliability and efficiency. If you do not provide a function to calculate the Jacobian, these solvers approximate the Jacobian numerically using finite differences. In this case, you might want to use the `Vectorized` or `JPattern` properties.

The following table describes the Jacobian matrix properties for all implicit solvers except `ode15i`. Further information on each property is given following the table. See Jacobian Properties for ode15i on page 2-2331 for `ode15i`-specific information.

Jacobian Properties for All Implicit Solvers Except ode15i

Property	Value	Description
`Jacobian`	Function | handle constant matrix	Matrix or function that evaluates the Jacobian.

Jacobian Properties for All Implicit Solvers Except ode15i (Continued)

Property	Value	Description
`JPattern`	Sparse matrix of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Allows the solver to reduce the number of function evaluations required.

Description of Jacobian Properties

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function `FJac`, where `FJac(t,y)` computes $\partial f / \partial y$, or to the constant value of $\partial f / \partial y$.

The Jacobian for the stiff van der Pol problem example, described in the MATLAB Mathematics documentation, can be coded as

```
function J = vdp1000jac(t,y)
J = [  0                    1
     (-2000*y(1)*y(2)-1)  (1000*(1-y(1)^2)) ];
```

`JPattern` — `JPattern` is a sparsity pattern with `1`s where there might be nonzero entries in the Jacobian.

Note If you specify `Jacobian`, the solver ignores any setting for `JPattern`.

Set this property to a sparse matrix *S* with *S(i,j) = 1* if component *i* of $f(t,y)$ depends on component *j* of *y*, and 0 otherwise. The solver uses this sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large and sparse, this can greatly

accelerate execution. For an example using the `JPattern` property, see Example: Large, Stiff, Sparse Problem in the MATLAB Mathematics documentation.

`Vectorized` — The `Vectorized` property allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Set `on` to inform the solver that you have coded the ODE function `F` so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`. This allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Note If you specify `Jacobian`, the solver ignores a setting of `'on'` for `'Vectorized'`.

With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. For example, you can vectorize the stiff van der Pol problem example, described in the MATLAB Mathematics documentation, by introducing colon notation into the subscripts and by using the array power (`.^`) and array multiplication (`.*`) operators.

```
function dydt = vdp1000(t,y)
dydt = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)];
```

Note Vectorization of the ODE function used by the ODE solvers differs from the vectorization used by the boundary value problem (BVP) solver, `bvp4c`. For the ODE solvers, the ODE function is vectorized only with respect to the second argument, while `bvp4c` requires vectorization with respect to the first and second arguments.

The following table describes the Jacobian matrix properties for `ode15i`.

Jacobian Properties for ode15i

Property	Value	Description
`Jacobian`	Function handle|Cell array of constant values	Function that evaluates the Jacobian or a cell array of constant values.
`JPattern`	Sparse matrices of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Vectorized ODE function

Description of Jacobian Properties for ode15i

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function

```
[dFdy, dFdp] = Fjac(t,y,yp)
```

or to a cell array of constant values $\{\partial F/\partial y, (\partial F/\partial y)'\}$.

`JPattern` — `JPattern` is a sparsity pattern with 1's where there might be nonzero entries in the Jacobian.

Set this property to `{dFdyPattern, dFdypPattern}`, the sparsity patterns of $\partial F/\partial y$ and $\partial F/\partial y'$, respectively.

`Vectorized` —

Set this property to `{yVect, ypVect}`. Setting `yVect` to `'on'` indicates that

```
F(t, [y1 y2 ...], yp)
```

returns

```
[F(t,y1,yp), F(t,y2,yp) ...]
```

Setting `ypVect` to `'on'` indicates that

```
F(t,y,[yp1 yp2 ...])
```

returns

```
[F(t,y,yp1) F(t,y,yp2) ...]
```

Mass Matrix and DAE Properties

This section describes mass matrix and differential-algebraic equation (DAE) properties, which apply to all the solvers except `ode15i`. These properties are not applicable to `ode15i` and their settings do not affect its behavior.

The solvers of the ODE suite can solve ODEs of the form

$$M(t,y)y' = f(t,y) \qquad \textbf{(2-1)}$$

with a mass matrix $M(t,y)$ that can be sparse.

When $M(t,y)$ is nonsingular, the equation above is equivalent to $y' = M^{-1}f(t,y)$ and the ODE has a solution for any initial values y_0 at t_0. The more general form (Equation 2-1) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse $M(t,y)$, solving Equation 2-1 directly reduces the storage and run-time needed to solve the problem.

When $M(t,y)$ is singular, then $M(t,y)$ times $M(t,y)y' = f(t,y)$ is a DAE. A DAE has a solution only when y_0 is consistent; that is, there exists an initial slope yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. If y_0 and yp_0 are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The `ode15s` and `ode23t` solvers can solve DAEs of index 1. For examples of DAE problems, see Example: Differential-Algebraic Problem, in the MATLAB Mathematics documentation, and the examples `amp1dae` and `hb1dae`.

The following table describes the mass matrix and DAE properties. Further information on each property is given following the table.

Mass Matrix and DAE Properties (Solvers Other Than ode15i)

Property	Value	Description
`Mass`	Matrix | function handle	Mass matrix or a function that evaluates the mass matrix $M(t,y)$.
`MStateDependence`	`none` | `{weak}` | `strong`	Dependence of the mass matrix on y.
`MvPattern`	Sparse matrix	$\partial(M(t,y)v)/\partial y$ sparsity pattern.
`MassSingular`	`yes` | `no` | `{maybe}`	Indicates whether the mass matrix is singular.
`InitialSlope`	Vector {zero vector}	Vector representing the consistent initial slope yp_0.

Description of Mass Matrix and DAE Properties

`Mass` — For problems of the form $M(t)y' = f(t,y)$, set `'Mass'` to a mass matrix M. For problems of the form $M(t)y' = f(t,y)$, set `'Mass'` to a function handle `@Mfun`, where `Mfun(t,y)` evaluates the mass matrix $M(t,y)$. The `ode23s` solver can only solve problems with a constant mass matrix M. When solving DAEs, using `ode15s` or `ode23t`, it is advantageous to formulate the problem so that M is a diagonal matrix (a semiexplicit DAE).

For example problems, see "Example: Finite Element Discretization" in the MATLAB Mathematics documentation, or the examples `fem2ode` or `batonode`.

MStateDependence — Set this property to none for problems $M(t)y' = f(t,y)$. Both weak and strong indicate $M(t,y)$, but weak results in implicit solvers using approximations when solving algebraic equations.

MvPattern — Set this property to a sparse matrix S with *S(i,j) = 1* if, for any k, the *(i,k)* component of $M(t,y)$ depends on component j of y, and 0 otherwise. For use with the ode15s, ode23t, and ode23tb solvers when MStateDependence is strong. See burgersode as an example.

MassSingular — Set this property to no if the mass matrix is not singular and you are using either the ode15s or ode23t solver. The default value of maybe causes the solver to test whether the problem is a DAE, by testing whether $M(t_0, y_0)$ is singular.

InitialSlope — Vector representing the consistent initial slope yp_0, where yp_0 satisfies $M(t_0, y_0) \cdot y_0' = f(t_0, y_0)$. The default is the zero vector.

This property is for use with the ode15s and ode23t solvers when solving DAEs.

ode15s and ode15i-Specific Properties

ode15s is a variable-order solver for stiff problems. It is based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear's methods. The ode15s properties let you choose among these formulas, as well as specifying the maximum order for the formula used.

ode15i solves fully implicit differential equations of the form

$$f(t, y, y') = 0$$

using the variable order BDF method.

The following table describes the ode15s and ode15i-specific properties. Further information on each property is given following the table. Use odeset to set these properties.

ode15s and ode15i-Specific Properties

Property	Value	Description
`MaxOrder`	`1 | 2 | 3 | 4 | {5}`	Maximum order formula used to compute the solution.
`BDF` (`ode15s` only)	`on | {off}`	Specifies whether you want to use the BDFs instead of the default NDFs.

Description of ode15s and ode15i-Specific Properties

`MaxOrder` — Maximum order formula used to compute the solution.

`BDF` (`ode15s` only) — Set `BDF` `on` to have `ode15s` use the BDFs.

For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently if `MaxOrder` is reduced (for example to `2`) so that only the most stable formulas are used.

See Also

`deval`, `odeget`, `ode45`, `ode23`, `ode23t`, `ode23tb`, `ode113`, `ode15s`, `ode23s`, `function_handle` (`@`)

odextend

Purpose

Extend solution of initial value problem for ordinary differential equation

Syntax

```
solext = odextend(sol, odefun, tfinal)
solext = odextend(sol, [], tfinal)
solext = odextend(sol, odefun, tfinal, yinit)
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
solext = odextend(sol, odefun, tfinal, yinit, options)
```

Description

`solext = odextend(sol, odefun, tfinal)` extends the solution stored in `sol` to an interval with upper bound `tfinal` for the independent variable. `odefun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. `sol` is an ODE solution structure created using an ODE solver. The lower bound for the independent variable in `solext` is the same as in `sol`. If you created `sol` with an ODE solver other than `ode15i`, the function `odefun` computes the right-hand side of the ODE equation, which is of the form $y' = f(t, y)$. If you created `sol` using `ode15i`, the function `odefun` computes the left-hand side of the ODE equation, which is of the form $f(t, y, y') = 0$.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `odefun`, if necessary.

`odextend` extends the solution by integrating `odefun` from the upper bound for the independent variable in `sol` to `tfinal`, using the same ODE solver that created `sol`. By default, `odextend` uses

- The initial conditions `y = sol.y(:, end)` for the subsequent integration
- The same integration properties and additional input arguments the ODE solver originally used to compute `sol`. This information is stored as part of the solution structure `sol` and is subsequently passed to `solext`. Unless you want to change these values, you do not need to pass them to `odextend`.

`solext = odextend(sol, [], tfinal)` uses the same ODE function that the ODE solver uses to compute `sol` to extend the solution. It is not necessary to pass in `odefun` explicitly unless it differs from the original ODE function.

`solext = odextend(sol, odefun, tfinal, yinit)` uses the column vector `yinit` as new initial conditions for the subsequent integration, instead of the vector `sol.y(end)`.

Note To extend solutions obtained with `ode15i`, use the following syntax, in which the column vector `ypinit` is the initial derivative of the solution:

```
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
```

`solext = odextend(sol, odefun, tfinal, yinit, options)` uses the integration properties specified in `options` instead of the options the ODE solver originally used to compute `sol`. The new options are then stored within the structure `solext`. See `odeset` for details on setting `options` properties. Set `yinit = []` as a placeholder to specify the default initial conditions.

Example

The following command

```
sol=ode45(@vdp1,[0 10],[2 0]);
```

uses `ode45` to solve the system `y' = vdp1(t,y)`, where `vdp1` is an example of an ODE function provided with MATLAB, on the interval `[0 10]`. Then, the commands

```
sol=odextend(sol,@vdp1,20);
plot(sol.x,sol.y(1,:));
```

extend the solution to the interval `[0 20]` and plot the first component of the solution on `[0 20]`.

See Also

deval, ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb, ode15i, odeset, odeget, deval, function_handle (@)

Purpose

Create array of all ones

Syntax

```
Y = ones(n)
Y = ones(m,n)
Y = ones([m n])
Y = ones(m,n,p,...)
Y = ones([m n p ...])
Y = ones(size(A))
ones(m, n,...,classname)
ones([m,n,...],classname)
```

Description

`Y = ones(n)` returns an `n`-by-`n` matrix of `1`s. An error message appears if `n` is not a scalar.

`Y = ones(m,n)` or `Y = ones([m n])` returns an `m`-by-`n` matrix of ones.

`Y = ones(m,n,p,...)` or `Y = ones([m n p ...])` returns an `m`-by-`n`-by-`p`-by-`...` array of `1`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`Y = ones(size(A))` returns an array of `1`s that is the same size as `A`.

`ones(m, n,...,classname)` or `ones([m,n,...],classname)` is an `m`-by-`n`-by-... array of ones of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Example

```
x = ones(2,3,'int8');
```

See Also

`eye`, `zeros`, `complex`

Purpose

Open files based on extension

Syntax

```
open('name')
```

Description

`open('name')` opens the object specified by the string `name`. The specific action taken upon opening depends on the type of object specified by `name`.

name	Action
DOC file (`*.doc`)	Open document in Microsoft Word.
EXE file (`*.exe`)	Run Microsoft Windows executable file.
Figure file (`*.fig`)	Open figure in a MATLAB figure window.
HTML file (`*.html, *.htm`)	Open HTML document in a separate window.
M-file (`name.m`)	Open M-file `name` in M-file Editor.
MAT-file (`name.mat`)	Open MAT-file and store variables in a structure in the workspace.
Model (`name.mdl`)	Open model `name` in Simulink.
P-file (`name.p`)	Open the corresponding M-file, `name.m`, if it exists, in the M-file Editor.
PDF file (`*.pdf`)	Open PDF document in Adobe Acrobat.
PPT file (`*.ppt`)	Open document in Microsoft PowerPoint.
Project file (`*.prj`)	Open the project file in the MATLAB Compiler Deployment Tool. If the MATLAB Compiler or Deployment Tool is not installed, open the project file in a text editor.
URL file (`*.url`)	Open an Internet location in your default Web browser
Variable	Open array `name` in the Array Editor (the array must be numeric).

name	Action
Other extensions (`name.xxx`)	Open `name.xxx` by calling the helper function `openxxx`, where `openxxx` is a user-defined function.
No extension (`name`)	Open `name` in the default editor. If `name` does not exist, then `open` checks to see if `name.mdl` or `name.m` is on the path or in the current directory and, if so, opens the file returned by `which('name')`.

If more than one file with the specified filename `name` exists on the MATLAB path, then `open` opens the file returned by `which('name')`.

If no such file `name` exists, then `open` displays an error message.

You can create your own `openxxx` functions to set up handlers for new file types. This does not apply to the file types shown in the table above. `open('filename.xxx')` calls the `openxxx` function it finds on the path. For example, create a function `openlog` if you want a handler for opening files with file extension `.log`.

Examples

Example 1 — Opening a File on the Path

To open the M-file `copyfile.m`, type

```
open copyfile.m
```

MATLAB opens the `copyfile.m` file that resides in `toolbox\matlab\general`. If you have a `copyfile.m` file in a directory that is before `toolbox\matlab\general` on the MATLAB path, then `open` opens that file instead.

Example 2 — Opening a File Not on the Path

To open a file that is not on the MATLAB path, enter the complete file specification. If no such file is found, then MATLAB displays an error message.

```
open('D:\temp\data.mat')
```

Example 3 — Specifying a File Without a File Extension

When you specify a file without including its file extension, MATLAB determines which file to open for you. It does this by calling

```
which('filename')
```

In this example, `open matrixdemos` could open either an M-file or a Simulink model of the same name, since both exist on the path.

```
dir matrixdemos.*

  matrixdemos.m    matrixdemos.mdl
```

Because the call `which('matrixdemos')` returns the name of the Simulink model, `open` opens the `matrixdemos` model rather than the M-file of that name.

```
open matrixdemos          % Opens model matrixdemos.mdl
```

Example 4 — Opening a MAT-File

This example opens a MAT-file containing MATLAB data and then keeps just one of the variables from that file. The others are overwritten when `ans` is reused by MATLAB.

```
% Open a MAT-file containing miscellaneous data.
open D:\temp\data.mat

ans =

          x: [3x2x2 double]
          y: {4x5 cell}
          k: 8
    spArray: [5x5 double]
   dblArray: [4x1 java.lang.Double[][]]
   strArray: {2x5 cell}


% Keep the dblArray value by assigning it to a variable.
```

```
dbl = ans.dblArray

dbl =

java.lang.Double[][]:
    [ 5.7200]    [ 6.7200]    [ 7.7200]
    [10.4400]    [11.4400]    [12.4400]
    [15.1600]    [16.1600]    [17.1600]
    [19.8800]    [20.8800]    [21.8800]
```

Example 5 — Using a User-Defined Handler Function

If you create an M-file function called `opencht` to handle files with extension `.cht`, and then issue the command

```
open myfigure.cht
```

`open` calls your handler function with the following syntax:

```
opencht('myfigure.cht')
```

See Also

`edit`, `load`, `save`, `saveas`, `uiopen`, `which`, `file_formats`, `path`

openfig

Purpose

Open new copy or raise existing copy of saved figure

Syntax

```
openfig('filename.fig','new')
openfig('filename.fig','new','visible')
openfig('filename.fig','new','visible')
openfig('filename.fig','reuse')
openfig('filename.fig')
openfig(...,'PropertyName',PropertyValue,...)
figure_handle = openfig(...)
```

Description

`openfig` is designed for use with GUI figures. Use this function to:

- Open the FIG-file creating the GUI and ensure it is displayed on screen. This provides compatibility with different screen sizes and resolutions.
- Control whether MATLAB displays one or multiple instances of the GUI at any given time.
- Return the handle of the figure created, which is typically hidden for GUI figures.

`openfig('filename.fig','new')` opens the figure contained in the FIG-file, `filename.fig`, and ensures it is visible and positioned completely on screen. You do not have to specify the full path to the FIG-file as long as it is on your MATLAB path. The `.fig` extension is optional.

openfig('filename.fig','new','invisible') or openfig('filename.fig','reuse','invisible') opens the figure as in the preceding example, while forcing the figure to be invisible.

`openfig('filename.fig','new','visible')` or `openfig('filename.fig','new','visible')` opens the figure, while forcing the figure to be visible.

`openfig('filename.fig','reuse')` opens the figure contained in the FIG-file only if a copy is not currently open; otherwise `openfig` brings

the existing copy forward, making sure it is still visible and completely on screen.

`openfig('filename.fig')` is the same as `openfig('filename.fig','new')`.

`openfig(...,'PropertyName',PropertyValue,...)` opens the FIG-file setting the specified figure properties before displaying the figure.

`figure_handle = openfig(...)` returns the handle to the figure.

Remarks

If the FIG-file contains an invisible figure, `openfig` returns its handle and leaves it invisible. The caller should make the figure visible when appropriate.

See Also

`guide`, `guihandles`, `movegui`, `open`, `hgload`, `save`

See Deploying User Interfaces in the MATLAB documentation for related functions

opengl

Purpose

Control OpenGL rendering

Syntax

```
opengl info
s = opengl('data')
opengl software
opengl hardware
opengl verbose
opengl quiet
opengl DriverBugWorkaround
opengl('DriverBugWorkaround',WorkaroundState)
```

Description

The OpenGL autoselection mode applies when the `RendererMode` of the figure is `auto`. Possible values for *`selection_mode`* are

- `autoselect` – allows OpenGL to be automatically selected if OpenGL is available and if there is graphics hardware on the host machine.
- `neverselect` – disables autoselection of OpenGL.
- `advise` – prints a message to the command window if OpenGL rendering is advised, but `RenderMode` is set to `manual`.

`opengl`, by itself, returns the current autoselection state.

Note that the autoselection state only specifies whether OpenGL should or should not be considered for rendering; it does not explicitly set the rendering to OpenGL. You can do this by setting the `Renderer` property of the figure to `OpenGL`. For example,

```
set(figure_handle,'Renderer','OpenGL')
```

`opengl info` prints information with the version and vendor of the OpenGL on your system. Also indicates wether your system is currently using hardware of software OpenGL and the state of various driver bug workarounds. Note that calling `opengl info` loads the OpenGL Library.

For example, the following output is generated on a Windows XP computer that uses ATI Technologies graphics hardware:

```
>> opengl info
Version         = 1.3.4010 WinXP Release
Vendor          = ATI Technologies Inc.
Renderer        = RADEON 9600SE x86/SSE2
MaxTextureSize  = 2048
Visual          = 05 (RGB 16 bits(05 06 05 00) zdepth 16, Hardware
Accelerated, Opengl, Double Buffered, Window)
Software        = false
# of Extensions = 85
Driver Bug Workarounds:
OpenGLBitmapZbufferBug    = 0
OpenGLWobbleTesselatorBug = 0
OpenGLLineSmoothingBug    = 0
OpenGLDockingBug          = 0
OpenGLClippedImageBug     = 0
```

Note that different computer systems may not list all OpenGL bugs.

`s = opengl('data')` returns a structure containing the same data that is displayed when you call `opengl info`, with the exception of the driver bug workaround state.

`opengl software` forces MATLAB to use software OpenGL rendering instead of hardware OpenGL. Note that Macintosh systems do not support software OpenGL.

`opengl hardware` reverses the `opengl software` command and enables MATLAB to use hardware OpenGL rendering if it is available. If your computer does not have OpenGL hardware acceleration, MATLAB automatically switches to software OpenGL rendering (except on Macintosh systems, which do not support software OpenGL).

Note that on UNIX systems, the `software` or `hardware` options with the `opengl` command works only if MATLAB has not yet used the OpenGL renderer or you have not issued the `opengl info` command (which attempts to load the OpenGL Library).

`opengl verbose` displays verbose messages about OpenGL initialization (if OpenGL is not already loaded) and other runtime messages.

`opengl quiet` disables verbose message setting.

`opengl` *`DriverBugWorkaround`* queries the state of the specified driver bug workaround. Use the command `opengl info` to see a list of all driver bug workarounds. See “Driver Bug Workarounds” on page 2-2348 for more information.

`opengl('`*`DriverBugWorkaround`*`',WorkaroundState)` sets the state of the specified driver bug workaround. You can set WorkaroundState to one of three values:

- `0` – Disable the specified *`DriverBugWorkaround`* (if enabled) and do not allow MATLAB to autoselect this workaround.
- `1` – Enable the specified *`DriverBugWorkaround`*.
- `-1` – Set the specified *`DriverBugWorkaround`* to autoselection mode, which allows MATLAB to enable this workaround if the requisite conditions exist.

Driver Bug Workarounds

MATLAB enables various OpenGL driver bug workarounds when it detects certain known problems with installed hardware. However, because there are many versions of graphics drivers, you might encounter situations when MATLAB does not enable a workaround that would solve a problem you are having with OpenGL rendering.

This section describes the symptoms that each workaround is designed to correct so you can decide if you want to try using one to fix an OpenGL rendering problem.

Use the `opengl info` command to see what driver bug workarounds are available on your computer.

Note These workarounds have not been tested under all driver combinations and therefore might produce undesirable results under certain conditions.

OpenGLBitmapZbufferBug

Symptom: text with background color (including data tips) and text displayed on image, patch, or surface objects is not visible when using OpenGL renderer.

Possible side effect: text is always on top of other objects.

Command to enable:

```
opengl('OpenGLBitmapZbufferBug',1)
```

OpenGLWobbleTesselatorBug

Symptom: Rendering complex patch object causes segmentation violation and returns a tesselator error message in the stack trace.

Command to enable:

```
opengl('OpenGLWobbleTesselatorBug',1)
```

OpenGLLineSmoothingBug

Symptom: Lines with a `LineWidth` greater than 3 look bad.

Command to enable:

```
opengl('OpenGLLineSmoothingBug',1)
```

OpenGLDockingBug

Symptom: MATLAB crashes when you dock a figure that has its `Renderer` property set to `opengl`.

Command to enable:

```
opengl('OpenGLDockingBug',1)
```

OpenGLClippedImageBug

Symptom: Images (as well as `colorbar` displays) do not display when the `Renderer` property set to `opengl`.

Command to enable:

```
opengl('OpenGLClippedImageBug',1)
```

OpenGLEraseModeBug

Symptom: Graphics objects with `EraseMode` property set to non-normal erase modes (`xor`, `none`, or `background`) do not draw when the figure `Renderer` property is set to `opengl`.

Command to enable:

```
opengl('OpenGLEraseModeBug',1)
```

See Also

Figure `Renderer` property for information on autoselection.

Purpose

Open workspace variable in Array Editor or other tool for graphical editing

GUI Alternatives

As an alternative to the `openvar` function, double-click a variable in the Workspace browser.

Syntax

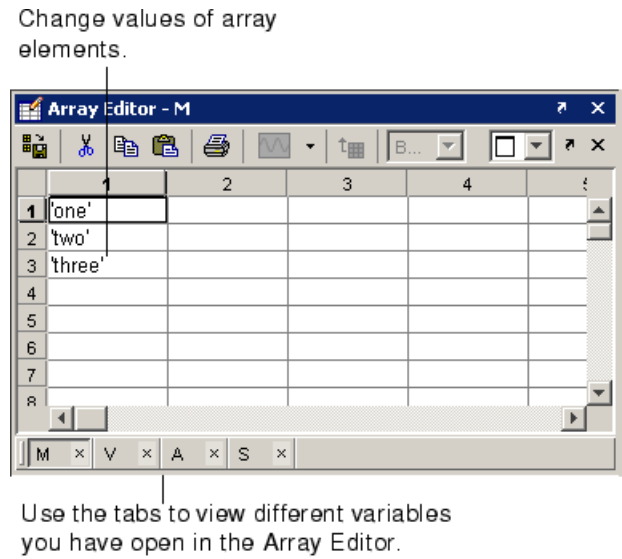
```
openvar('name')
```

Description

`openvar('name')` opens the workspace variable `name` in the Array Editor for graphical editing, where `name` is a numeric array, string, or cell array of strings.

MATLAB does not impose any limitation on the size of an array that can be opened in the Array Editor. Array size is limited only by the operating system or the amount of physical memory installed on your system.

For some toolboxes, `openvar` instead opens a tool appropriate for viewing or editing that type of object.



See Also

load, save, workspace

Purpose

Optimization options values

Syntax

```
val = optimget(options,'param')
val = optimget(options,'param',default)
```

Description

`val = optimget(options,'param')` returns the value of the specified parameter in the optimization options structure `options`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = optimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`. Note that this form of the function is used primarily by other optimization functions.

Examples

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options`.

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'`.

```
optnew = optimget(my_options,'Display','final');
```

See Also

`optimset`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

optimset

Purpose

Create or edit optimization options structure

Syntax

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,'param1',value1,...)
options = optimset(oldopts,newopts)
```

Description

The function `optimset` creates an `options` structure that you can pass as an input argument to the following four MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`
- `lsqnonneg`

You can use the `options` structure to change the default parameters for these functions.

Note If you have purchased the Optimization Toolbox, you can also use `optimset` to create an expanded `options` structure containing additional options specifically designed for the functions provided in that toolbox. See the reference page for the enhanced `optimset` function in the Optimization Toolbox for more information about these additional options.

`options = optimset('param1',value1,'param2',value2,...)` creates an optimization options structure called `options`, in which the specified parameters (`param`) have specified values. Any unspecified parameters are set to `[]` (parameters with value `[]` indicate to use the default value for that parameter when `options` is passed to the

optimization function). It is sufficient to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`optimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all fields are set to `[]`.

`options = optimset(optimfun)` creates an options structure `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = optimset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the available options for the MATLAB optimization functions.

Option	Value	Description
`Display`	`'off'` | `'iter'` | `{'final'}` | `'notify'`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` displays output only if the function does not converge.

Option	Value	Description
FunValCheck	{'off'} | 'on'	Check whether objective function values are valid. 'on' displays an error when the objective function returns a value that is complex or NaN. 'off' displays no error.
MaxFunEvals	positive integer	Maximum number of function evaluations allowed.
MaxIter	positive integer	Maximum number of iterations allowed.
OutputFcn	function | {[]}	User-defined function that an optimization function calls at each iteration. See "Output Function" in the Optimization Toolbox for more information.
PlotFcns	function | {[]}	User-defined plot function that an optimization function calls at each iteration. See "Plot Functions" in the Optimization Toolbox for more information.
TolFun	positive scalar	Termination tolerance on the function value.
TolX	positive scalar	Termination tolerance on x.

Examples

This statement creates an optimization options structure called `options` in which the `Display` parameter is set to `'iter'` and the `TolFun` parameter is set to `1e-8`.

```
options = optimset('Display','iter','TolFun',1e-8)
```

This statement makes a copy of the options structure called `options`, changing the value of the `TolX` parameter and storing new values in `optnew`.

```
optnew = optimset(options,'TolX',1e-4);
```

This statement returns an optimization options structure that contains all the parameter names and default values relevant to the function `fminbnd`.

```
optimset('fminbnd')
```

See Also

`optimset` (Optimization Toolbox version), `optimget`, `fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

Purpose

Find logical OR of array or scalar inputs

Syntax

```
A | B | ...
or(A, B)
```

Description

`A | B | ...` performs a logical OR of all input arrays `A`, `B`, etc., and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if any input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input `A` is a 3-by-5 matrix and input `B` is the number 1, then `B` is treated as if it were a 3-by-5 matrix of ones.

`or(A, B)` is called for the syntax `A | B` when either `A` or `B` is an object.

Note The symbols | and || perform different operations in MATLAB. The element-wise OR operator described here is |. The short-circuit OR operator is ||.

Example

If matrix `A` is

```
    0.4235    0.5798         0    0.7942         0
    0.5155         0         0         0    0.8744
         0         0         0    0.4451    0.0150
    0.4329    0.6405    0.6808         0         0
```

and matrix `B` is

```
     0     1     0     1     0
     1     1     0     0     1
     0     0     0     1     0
     0     1     0     0     1
```

then

```
A | B
ans =
     1     1     0     1     0
     1     1     0     0     1
     0     0     0     1     1
     1     1     1     0     1
```

See Also

`bitor`, `and`, `xor`, `not`, `any`, `all`, logical operators, logical types, bitwise functions

ordeig

Purpose

Eigenvalues of quasitriangular matrices

Syntax

```
E = ordeig(T)
E = ordeig(AA,BB)
```

Description

`E = ordeig(T)` takes a quasitriangular Schur matrix `T`, typically produced by `schur`, and returns the vector `E` of eigenvalues in their order of appearance down the diagonal of T.

`E = ordeig(AA,BB)` takes a quasitriangular matrix pair `AA` and `BB`, typically produced by `qz`, and returns the generalized eigenvalues in their order of appearance down the diagonal of `AA-λ*BB`.

`ordeig` is an order-preserving version of `eig` for use with `ordschur` and `ordqz`. It is also faster than `eig` for quasitriangular matrices.

Examples

Example 1

```
T=diag([1 -1 3 -5 2]);
```

`ordeig(T)` returns the eigenvalues of `T` in the same order they appear on the diagonal.

```
ordeig(T)

ans =

     1
    -1
     3
    -5
     2
```

`eig(T)`, on the other hand, returns the eigenvalues in order of increasing magnitude.

```
eig(T)

ans =
```

```
    -5
    -1
     1
     2
     3
```

Example 2

```
A = rand(10);
[U, T] = schur(A);
abs(ordeig(T))

ans =

    5.3786
    0.7564
    0.7564
    0.7802
    0.7080
    0.7080
    0.5855
    0.5855
    0.1445
    0.0812
% Move eigenvalues with magnitude < 0.5 to the
% upper-left corner of T.
[U,T] = ordschur(U,T,abs(E)<0.5);
abs(ordeig(T))

ans =

    0.1445
    0.0812
    5.3786
    0.7564
    0.7564
    0.7802
```

```
    0.7080
    0.7080
    0.5855
    0.5855
```

See Also

schur, qz, ordschur, ordqz, eig

Purpose

Order fields of structure array

Syntax

```
s = orderfields(s1)
s = orderfields(s1, s2)
s = orderfields(s1, c)
s = orderfields(s1, perm)
[s, perm] = orderfields(...)
```

Description

`s = orderfields(s1)` orders the fields in `s1` so that the new structure array `s` has field names in ASCII dictionary order.

`s = orderfields(s1, s2)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in `s2`. Structures `sl` and `s2` must have the same fields.

`s = orderfields(s1, c)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in the cell array of field name strings `c`. Structure `s1` and cell array `c` must contain the same field names.

`s = orderfields(s1, perm)` orders the fields in `s1` so that the new structure array `s` has fieldnames in the order specified by the indices in permutation vector `perm`.

If `s1` has `N` fieldnames, the elements of `perm` must be an arrangement of the numbers from `1` to `N`. This is particularly useful if you have more than one structure array that you would like to reorder in the same way.

`[s, perm] = orderfields(...)` returns a permutation vector representing the change in order performed on the fields of the structure array that results in `s`.

Remarks

`orderfields` only orders top-level fields. It is not recursive.

Examples

Create a structure `s`. Then create a new structure from `s`, but with the fields ordered alphabetically:

```
s = struct('b', 2, 'c', 3, 'a', 1)
s =
```

```
    b: 2
    c: 3
    a: 1

snew = orderfields(s)
snew =
    a: 1
    b: 2
    c: 3
```

Arrange the fields of `s` in the order specified by the second (cell array) argument of `orderfields`. Return the new structure in `snew` and the permutation vector used to create it in `perm`:

```
[snew, perm] = orderfields(s, {'b', 'a', 'c'})
snew =
    b: 2
    a: 1
    c: 3
perm =
     1
     3
     2
```

Now create a new structure, `s2`, having the same fieldnames as `s`. Reorder the fields using the permutation vector returned in the previous operation:

```
s2 = struct('b', 3, 'c', 7, 'a', 4)
s2 =
    b: 3
    c: 7
    a: 4

snew = orderfields(s2, perm)
snew =
    b: 3
    a: 4
```

```
    c: 7
```

See Also

`struct`, `fieldnames`, `setfield`, `getfield`, `isfield`, `rmfield`, "Using Dynamic Field Names"

Purpose

Reorder eigenvalues in QZ factorization

Syntax

```
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)
[...] = ordqz(AA,BB,Q,Z,keyword)
[...] = ordqz(AA,BB,Q,Z,clusters)
```

Description

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)` reorders the QZ factorizations `Q*A*Z = AA` and `Q*B*Z = BB` produced by the `qz` function for a matrix pair `(A,B)`. It returns the reordered pair `(AAS,BBS)` and the cumulative orthogonal transformations `QS` and `ZS` such that `QS*A*ZS = AAS` and `QS*B*ZS = BBS`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular pair `(AAS,BBS)`, and the corresponding invariant subspace is spanned by the leading columns of `ZS`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along the diagonal of `AA-λ*BB`.

Note To extract `E` from `AA` and `BB`, use `ordeig(BB)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `AA-λ*BB`.

`[...] = ordqz(AA,BB,Q,Z,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[...] = ordqz(AA,BB,Q,Z,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices commensurate with `E = ordeig(AA,BB)`, such that all eigenvalues with the same `clusters`

value form one cluster, `ordqz` sorts the specified clusters in descending order along the diagonal of `(AAS,BBS)`. The cluster with highest index appears in the upper left corner.

Algorithm

For full matrices `AA` and `BB`, `qz` uses the LAPACK routines listed in the following table.

	AA and BB Real	AA or BB Complex
`A` and `B double`	`DTGSEN`	`ZTGSEN`
`A` or `B single`	`STGSEN`	`CTGSEN`

See Also

`ordeig`, `ordschur`, `qz`

Purpose

Reorder eigenvalues in Schur factorization

Syntax

```
[US,TS] = ordschur(U,T,select)
[US,TS] = ordschur(U,T,keyword)
[US,TS] = ordschur(U,T,clusters)
```

Description

`[US,TS] = ordschur(U,T,select)` reorders the Schur factorization `X = U*T*U'` produced by the `schur` function and returns the reordered Schur matrix `TS` and the cumulative orthogonal transformation `US` such that `X = US*TS*US'`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular Schur matrix `TS`, and the corresponding invariant subspace is spanned by the leading columns of `US`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along `T`'s diagonal.

Note To extract `E` from `T`, use `E = ordeig(T)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `TS`.

`[US,TS] = ordschur(U,T,keyword)` sets the selected cluster to include all eigenvalues in one of the following regions:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[US,TS] = ordschur(U,T,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices, commensurate with `E = ordeig(T)`, and such that all eigenvalues with the same `clusters` value form one cluster, `ordschur` sorts the specified clusters

in descending order along the diagonal of `TS`, the cluster with highest index appearing in the upper left corner.

Algorithm

Input of Type Double

If `U` and `T` have type `double`, `ordschur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

Matrix Type	Routine
Real	`DTRSEN`
Complex	`ZTRSEN`

Input of Type Single

If `U` and `T` have type `single`, `ordschur` uses the LAPACK routines listed in the following table to reorder the Schur form of a matrix:

Matrix Type	Routine
Real	`STRSEN`
Complex	`CTRSEN`

See Also

`ordeig`, `ordqz`, `schur`

orient

Purpose

Hardcopy paper orientation

GUI Alternative

Use **File —> Print Preview** on the figure window menu to directly manipulate print layout, paper size, headers, fonts and other properties when printing figures. For details, see Using Print Preview in the MATLAB Graphics documentation.

Syntax

```
orient
orient landscape
orient portrait
orient tall
orient(fig_handle), orient(simulink_model)
orient(fig_handle,orientation), orient(simulink_model,
   orientation)
```

Description

`orient` returns a string with the current paper orientation: `portrait`, `landscape`, or `tall`.

`orient landscape` sets the paper orientation of the current figure to full-page landscape, orienting the longest page dimension horizontally. The figure is centered on the page and scaled to fit the page with a 0.25 inch border.

`orient portrait` sets the paper orientation of the current figure to portrait, orienting the longest page dimension vertically. The `portrait` option returns the page orientation to the MATLAB default. (Note that the result of using the `portrait` option is affected by changes you make to figure properties. See the "Algorithm" section for more specific information.)

`orient tall` maps the current figure to the entire page in portrait orientation, leaving a 0.25 inch border.

`orient(fig_handle), orient(simulink_model)` returns the current orientation of the specified figure or Simulink model.

`orient(fig_handle,`*`orientation`*`),`
`orient(simulink_model,`*`orientation`*`)` sets the

orientation for the specified figure or Simulink model to the specified orientation (`landscape`, `portrait`, or `tall`).

Algorithm

`orient` sets the `PaperOrientation`, `PaperPosition`, and `PaperUnits` properties of the current figure. Subsequent print operations use these properties. The result of using the `portrait` option can be affected by default property values as follows:

- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the current values of `PaperOrientation` and `PaperPosition` to place the figure on the page.
- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the default figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.
- If the current figure `PaperType` is different from the default figure `PaperType`, then the `orient portrait` command uses the current figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.

See Also

`print`, `printpreview`, `set`

`PaperOrientation`, `PaperPosition`, `PaperSize`, `PaperType`, and `PaperUnits` properties of figure graphics objects

“Printing” on page 1-92 for related functions

orth

Purpose

Range space of matrix

Syntax

```
B = orth(A)
```

Description

`B = orth(A)` returns an orthonormal basis for the range of `A`. The columns of `B` span the same space as the columns of `A`, and the columns of `B` are orthogonal, so that `B'*B = eye(rank(A))`. The number of columns of `B` is the rank of `A`.

See Also

`null`, `svd`, `rank`

Purpose

Default part of switch statement

Syntax

```
switch switch_expr
 case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

Description

`otherwise` is part of the `switch` statement syntax, which allows for conditional execution. The statements following `otherwise` are executed only if none of the preceding case expressions (`case_expr`) matches the switch expression (`sw_expr`).

Examples

The general form of the `switch` statement is

```
switch sw_expr
    case case_expr
        statement
        statement
    case {case_expr1,case_expr2,case_expr3}
        statement
        statement
    otherwise
        statement
        statement
end
```

See `switch` for more details.

See Also

`switch`, `case`, `end`, `if`, `else`, `elseif`, `while`

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

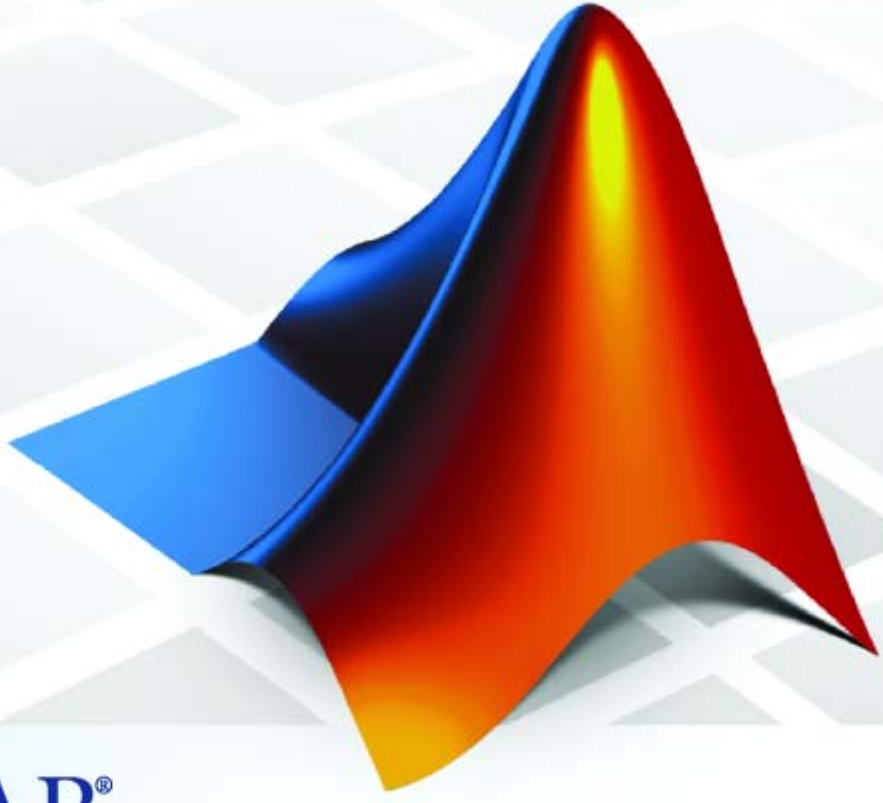
W

X

Y

Z

MATLAB® 7
Function Reference: Volume 3 (P-Z)



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Function Reference

Revision History

December 1996	First printing	For MATLAB 5.0 (Release 8)
June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online only	Revised for MATLAB 5.3 (Release 11)
June 1999	Second printing	For MATLAB 5.3 (Release 11)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for 6.5 (Release 13)
June 2004	Online only	Revised for 7.0 (Release 14)
September 2006	Online only	Revised for 7.3 (Release 2006b)
September 2007	Online only	Revised for 7.5 (Release 2007b)

Contents

1 Functions — By Category

Functions — By Category

Creating Graphical User Interfaces (p. 1-104)	GUIDE, programming graphical user interfaces
External Interfaces (p. 1-109)	Interfaces to DLLs, Java, COM and ActiveX, Web services, and serial port devices, and C and Fortran routines

Desktop Tools and Development Environment

Startup and Shutdown

exit	Terminate MATLAB (same as `quit`)
finish	MATLAB termination M-file
matlab (UNIX)	Start MATLAB (UNIX systems)
matlab (Windows)	Start MATLAB (Windows systems)
matlabrc	MATLAB startup M-file for single-user systems or system administrators
prefdir	Directory containing preferences, history, and layout files
preferences	Open Preferences dialog box for MATLAB and related products

quit	Terminate MATLAB
startup	MATLAB startup M-file for user-defined options

Command Window and History

clc	Clear Command Window
commandhistory	Open Command History window, or select it if already open
commandwindow	Open Command Window, or select it if already open
diary	Save session to file
dos	Execute DOS command and return result
format	Set display format for output
home	Move cursor to upper-left corner of Command Window
matlabcolon (matlab:)	Run specified function via hyperlink
more	Control paged output for Command Window
perl	Call Perl script using appropriate operating system executable
system	Execute operating system command and return result
unix	Execute UNIX command and return result

Help for Using MATLAB

builddocsearchdb	Build searchable documentation database
demo	Access product demos via Help browser
doc	Reference page in Help browser
docopt	Web browser for UNIX platforms
docsearch	Open Help browser **Search** pane and search for specified term
echodemo	Run M-file demo step-by-step in Command Window
help	Help for MATLAB functions in Command Window
helpbrowser	Open Help browser to access all online documentation and demos
helpwin	Provide access to M-file help for all functions
info	Information about contacting The MathWorks
lookfor	Search for keyword in all help entries
playshow	Run M-file demo (deprecated; use `echodemo` instead)
support	Open MathWorks Technical Support Web page
web	Open Web site or file in Web browser or Help browser
whatsnew	Release Notes for MathWorks products

Workspace, Search Path, and File Operations

Workspace

assignin	Assign value to variable in specified workspace
clear	Remove items from workspace, freeing up system memory
evalin	Execute MATLAB expression in specified workspace
exist	Check existence of variable, function, directory, or Java class
openvar	Open workspace variable in Array Editor or other tool for graphical editing
pack	Consolidate workspace memory
uiimport	Open Import Wizard to import data
which	Locate functions and files
workspace	Open Workspace browser to manage workspace

Search Path

addpath	Add directories to MATLAB search path
genpath	Generate path string
partialpath	Partial pathname description

path	View or change MATLAB directory search path
path2rc	Save current MATLAB search path to `pathdef.m` file
pathdef	Directories in MATLAB search path
pathsep	Path separator for current platform
pathtool	Open Set Path dialog box to view and change MATLAB path
restoredefaultpath	Restore default MATLAB search path
rmpath	Remove directories from MATLAB search path
savepath	Save current MATLAB search path to `pathdef.m` file

File Operations

See also "File I/O" on page 1-76 functions.

cd	Change working directory
copyfile	Copy file or directory
delete	Remove files or graphics objects
dir	Directory listing
exist	Check existence of variable, function, directory, or Java class
fileattrib	Set or get attributes of file or directory
filebrowser	Current Directory browser
isdir	Determine whether input is a directory
lookfor	Search for keyword in all help entries

ls	Directory contents on UNIX system
matlabroot	Root directory of MATLAB installation
mkdir	Make new directory
movefile	Move file or directory
pwd	Identify current directory
recycle	Set option to move deleted files to recycle folder
rehash	Refresh function and file system path caches
rmdir	Remove directory
toolboxdir	Root directory for specified toolbox
type	Display contents of file
web	Open Web site or file in Web browser or Help browser
what	List MATLAB files in current directory
which	Locate functions and files

Programming Tools

Edit and Debug M-Files (p. 1-9)	Edit and debug M-files
Improve Performance and Tune M-Files (p. 1-9)	Improve performance and find potential problems in M-files
Source Control (p. 1-10)	Interface MATLAB with source control system
Publishing (p. 1-10)	Publish M-file code and results

Edit and Debug M-Files

clipboard	Copy and paste strings to and from system clipboard
datatipinfo	Produce short description of input variable
dbclear	Clear breakpoints
dbcont	Resume execution
dbdown	Change local workspace context when in debug mode
dbquit	Quit debug mode
dbstack	Function call stack
dbstatus	List all breakpoints
dbstep	Execute one or more lines from current breakpoint
dbstop	Set breakpoints
dbtype	List M-file with line numbers
dbup	Change local workspace context
debug	List M-file debugging functions
edit	Edit or create M-file
keyboard	Input from keyboard

Improve Performance and Tune M-Files

memory	Help for memory limitations
mlint	Check M-files for possible problems
mlintrpt	Run `mlint` for file or directory, reporting results in browser
pack	Consolidate workspace memory
profile	Profile execution time for function

profsave	Save profile report in HTML format
rehash	Refresh function and file system path caches
sparse	Create sparse matrix
zeros	Create array of all zeros

Source Control

checkin	Check files into source control system (UNIX)
checkout	Check files out of source control system (UNIX)
cmopts	Name of source control system
customverctrl	Allow custom source control system (UNIX)
undocheckout	Undo previous checkout from source control system (UNIX)
verctrl	Source control actions (Windows)

Publishing

grabcode	MATLAB code from M-files published to HTML
notebook	Open M-book in Microsoft Word (Windows)
publish	Publish M-file containing cells, saving output to file of specified type

System

Operating System Interface (p. 1-11)	Exchange operating system information and commands with MATLAB
MATLAB Version and License (p. 1-12)	Information about MATLAB version and license

Operating System Interface

clipboard	Copy and paste strings to and from system clipboard
computer	Information about computer on which MATLAB is running
dos	Execute DOS command and return result
getenv	Environment variable
hostid	MATLAB server host identification number
maxNumCompThreads	Controls maximum number of computational threads
perl	Call Perl script using appropriate operating system executable
setenv	Set environment variable
system	Execute operating system command and return result
unix	Execute UNIX command and return result
winqueryreg	Item from Microsoft Windows registry

MATLAB Version and License

ismac	Determine whether running Macintosh OS X versions of MATLAB
ispc	Determine whether PC (Windows) version of MATLAB
isstudent	Determine whether Student Version of MATLAB
isunix	Determine whether UNIX version of MATLAB
javachk	Generate error message based on Java feature support
license	Return license number or perform licensing task
prefdir	Directory containing preferences, history, and layout files
usejava	Determine whether Java feature is supported in MATLAB
ver	Version information for MathWorks products
verLessThan	Compare toolbox version to specified version string
version	Version number for MATLAB

Mathematics

Arrays and Matrices (p. 1-14)	Basic array operators and operations, creation of elementary and specialized arrays and matrices
Linear Algebra (p. 1-19)	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
Elementary Math (p. 1-23)	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
Polynomials (p. 1-28)	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
Interpolation and Computational Geometry (p. 1-28)	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
Cartesian Coordinate System Conversion (p. 1-31)	Conversions between Cartesian and polar or spherical coordinates
Nonlinear Numerical Methods (p. 1-31)	Differential equations, optimization, integration
Specialized Math (p. 1-35)	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
Sparse Matrices (p. 1-36)	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
Math Constants (p. 1-39)	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

Arrays and Matrices

Basic Information (p. 1-14)	Display array contents, get array information, determine array type
Operators (p. 1-15)	Arithmetic operators
Elementary Matrices and Arrays (p. 1-16)	Create elementary arrays of different types, generate arrays for plotting, array indexing, etc.
Array Operations (p. 1-17)	Operate on array content, apply function to each array element, find cumulative product or sum, etc.
Array Manipulation (p. 1-17)	Create, sort, rotate, permute, reshape, and shift array contents
Specialized Matrices (p. 1-18)	Create Hadamard, Companion, Hankel, Vandermonde, Pascal matrices, etc.

Basic Information

disp	Display text or array
display	Display text or array (overloaded method)
isempty	Determine whether array is empty
isequal	Test arrays for equality
isequalwithequalnans	Test arrays for equality, treating NaNs as equal
isfinite	Array elements that are finite
isfloat	Determine whether input is floating-point array
isinf	Array elements that are infinite
isinteger	Determine whether input is integer array

islogical	Determine whether input is logical array
isnan	Array elements that are NaN
isnumeric	Determine whether input is numeric array
isscalar	Determine whether input is scalar
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
length	Length of vector
max	Largest elements in array
min	Smallest elements in array
ndims	Number of array dimensions
numel	Number of elements in array or subscripted array expression
size	Array dimensions

Operators

+	Addition
+	Unary plus
-	Subtraction
-	Unary minus
*	Matrix multiplication
^	Matrix power
\	Backslash or left matrix divide
/	Slash or right matrix divide
’	Transpose
.’	Nonconjugated transpose
.*	Array multiplication (element-wise)

.^	Array power (element-wise)
.\	Left array divide (element-wise)
./	Right array divide (element-wise)

Elementary Matrices and Arrays

blkdiag	Construct block diagonal matrix from input arguments
diag	Diagonal matrices and diagonals of matrix
eye	Identity matrix
freqspace	Frequency spacing for frequency response
ind2sub	Subscripts from linear index
linspace	Generate linearly spaced vectors
logspace	Generate logarithmically spaced vectors
meshgrid	Generate `X` and `Y` arrays for 3-D plots
ndgrid	Generate arrays for N-D functions and interpolation
ones	Create array of all ones
rand	Uniformly distributed pseudorandom numbers
randn	Normally distributed random numbers
sub2ind	Single index from subscripts
zeros	Create array of all zeros

Array Operations

See “Linear Algebra” on page 1-19 and “Elementary Math” on page 1-23 for other array operations.

accumarray	Construct array with accumulation
arrayfun	Apply function to each element of array
bsxfun	Apply element-by-element binary operation to two arrays with singleton expansion enabled
cast	Cast variable to different data type
cross	Vector cross product
cumprod	Cumulative product
cumsum	Cumulative sum
dot	Vector dot product
idivide	Integer division with rounding option
kron	Kronecker tensor product
prod	Product of array elements
sum	Sum of array elements
tril	Lower triangular part of matrix
triu	Upper triangular part of matrix

Array Manipulation

blkdiag	Construct block diagonal matrix from input arguments
cat	Concatenate arrays along specified dimension
circshift	Shift array circularly

diag	Diagonal matrices and diagonals of matrix
end	Terminate block of code, or indicate last array index
flipdim	Flip array along specified dimension
fliplr	Flip matrix left to right
flipud	Flip matrix up to down
horzcat	Concatenate arrays horizontally
inline	Construct inline object
ipermute	Inverse permute dimensions of N-D array
permute	Rearrange dimensions of N-D array
repmat	Replicate and tile array
reshape	Reshape array
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
vectorize	Vectorize expression
vertcat	Concatenate arrays vertically

Specialized Matrices

compan	Companion matrix
gallery	Test matrices
hadamard	Hadamard matrix
hankel	Hankel matrix

hilb	Hilbert matrix
invhilb	Inverse of Hilbert matrix
magic	Magic square
pascal	Pascal matrix
rosser	Classic symmetric eigenvalue test problem
toeplitz	Toeplitz matrix
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix

Linear Algebra

Matrix Analysis (p. 1-19)	Compute norm, rank, determinant, condition number, etc.
Linear Equations (p. 1-20)	Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.
Eigenvalues and Singular Values (p. 1-21)	Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.
Matrix Logarithms and Exponentials (p. 1-22)	Matrix logarithms, exponentials, square root
Factorization (p. 1-22)	Cholesky, LU, and QR factorizations, diagonal forms, singular value decomposition

Matrix Analysis

cond	Condition number with respect to inversion
condeig	Condition number with respect to eigenvalues

det	Matrix determinant
norm	Vector and matrix norms
normest	2-norm estimate
null	Null space
orth	Range space of matrix
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
rref	Reduced row echelon form
subspace	Angle between two subspaces
trace	Sum of diagonal elements

Linear Equations

chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cond	Condition number with respect to inversion
condest	1-norm condition number estimate
funm	Evaluate general matrix function
ilu	Sparse incomplete LU factorization
inv	Matrix inverse
linsolve	Solve linear system of equations
lscov	Least-squares solution in presence of known covariance
lsqnonneg	Solve nonnegative least-squares constraints problem
lu	LU matrix factorization

luinc	Sparse incomplete LU factorization
pinv	Moore-Penrose pseudoinverse of matrix
qr	Orthogonal-triangular decomposition
rcond	Matrix reciprocal condition number estimate

Eigenvalues and Singular Values

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
condeig	Condition number with respect to eigenvalues
eig	Find eigenvalues and eigenvectors
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
gsvd	Generalized singular value decomposition
hess	Hessenberg form of matrix
ordeig	Eigenvalues of quasitriangular matrices
ordqz	Reorder eigenvalues in QZ factorization
ordschur	Reorder eigenvalues in Schur factorization
poly	Polynomial with specified roots
polyeig	Polynomial eigenvalue problem

rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtm	Matrix square root
ss2tf	Convert state-space filter parameters to transfer function form
svd	Singular value decomposition
svds	Find singular values and vectors

Matrix Logarithms and Exponentials

expm	Matrix exponential
logm	Matrix logarithm
sqrtm	Matrix square root

Factorization

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cholupdate	Rank 1 update to Cholesky factorization
gsvd	Generalized singular value decomposition
ilu	Sparse incomplete LU factorization
lu	LU matrix factorization

luinc	Sparse incomplete LU factorization
planerot	Givens plane rotation
qr	Orthogonal-triangular decomposition
qrdelete	Remove column or row from QR factorization
qrinsert	Insert column or row into QR factorization
qrupdate	
qz	QZ factorization for generalized eigenvalues
rsf2csf	Convert real Schur form to complex Schur form
svd	Singular value decomposition

Elementary Math

Trigonometric (p. 1-24)	Trigonometric functions with results in radians or degrees
Exponential (p. 1-25)	Exponential, logarithm, power, and root functions
Complex (p. 1-26)	Numbers with real and imaginary components, phase angles
Rounding and Remainder (p. 1-27)	Rounding, modulus, and remainder
Discrete Math (e.g., Prime Factors) (p. 1-27)	Prime factors, factorials, permutations, rational fractions, least common multiple, greatest common divisor

Trigonometric

acos	Inverse cosine; result in radians
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent
acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine; result in radians
asind	Inverse sine; result in degrees
asinh	Inverse hyperbolic sine
atan	Inverse tangent; result in radians
atan2	Four-quadrant inverse tangent
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent
cos	Cosine of argument in radians
cosd	Cosine ofo argument in degrees
cosh	Hyperbolic cosine
cot	Cotangent of argument in radians
cotd	Cotangent of argument in degrees
coth	Hyperbolic cotangent
csc	Cosecant of argument in radians

cscd	Cosecant of argument in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant of argument in radians
secd	Secant of argument in degrees
sech	Hyperbolic secant
sin	Sine of argument in radians
sind	Sine of argument in degrees
sinh	Hyperbolic sine of argument in radians
tan	Tangent of argument in radians
tand	Tangent of argument in degrees
tanh	Hyperbolic tangent

Exponential

exp	Exponential
expm1	Compute `exp(x)-1` accurately for small values of `x`
log	Natural logarithm
log10	Common (base 10) logarithm
log1p	Compute `log(1+x)` accurately for small values of `x`
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
nextpow2	Next higher power of 2
nthroot	Real `n`th root of real numbers
pow2	Base 2 power and scale floating-point numbers

reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays
sqrt	Square root

Complex

abs	Absolute value and complex magnitude
angle	Phase angle
complex	Construct complex data from real and imaginary components
conj	Complex conjugate
cplxpair	Sort complex numbers into complex conjugate pairs
i	Imaginary unit
imag	Imaginary part of complex number
isreal	Determine whether input is real array
j	Imaginary unit
real	Real part of complex number
sign	Signum function
unwrap	Correct phase angles to produce smoother phase plots

Rounding and Remainder

ceil	Round toward infinity
fix	Round toward zero
floor	Round toward minus infinity
idivide	Integer division with rounding option
mod	Modulus after division
rem	Remainder after division
round	Round to nearest integer

Discrete Math (e.g., Prime Factors)

factor	Prime factors
factorial	Factorial function
gcd	Greatest common divisor
isprime	Array elements that are prime numbers
lcm	Least common multiple
nchoosek	Binomial coefficient or all combinations
perms	All possible permutations
primes	Generate list of prime numbers
rat, rats	Rational fraction approximation

Polynomials

conv	Convolution and polynomial multiplication
deconv	Deconvolution and polynomial division
poly	Polynomial with specified roots
polyder	Polynomial derivative
polyeig	Polynomial eigenvalue problem
polyfit	Polynomial curve fitting
polyint	Integrate polynomial analytically
polyval	Polynomial evaluation
polyvalm	Matrix polynomial evaluation
residue	Convert between partial fraction expansion and polynomial coefficients
roots	Polynomial roots

Interpolation and Computational Geometry

Interpolation (p. 1-29)	Data interpolation, data gridding, polynomial evaluation, nearest point search
Delaunay Triangulation and Tessellation (p. 1-30)	Delaunay triangulation and tessellation, triangular surface and mesh plots
Convex Hull (p. 1-30)	Plot convex hull, plotting functions
Voronoi Diagrams (p. 1-30)	Plot Voronoi diagram, patch graphics object, plotting functions
Domain Generation (p. 1-31)	Generate arrays for 3-D plots, or for N-D functions and interpolation

Interpolation

dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
griddata	Data gridding
griddata3	Data gridding and hypersurface fitting for 3-D data
griddatan	Data gridding and hypersurface fitting (dimension >= 2)
interp1	1-D data interpolation (table lookup)
interp1q	Quick 1-D linear interpolation
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpft	1-D interpolation using FFT method
interpn	N-D data interpolation (table lookup)
meshgrid	Generate `X` and `Y` arrays for 3-D plots
mkpp	Make piecewise polynomial
ndgrid	Generate arrays for N-D functions and interpolation
padecoef	Padé approximation of time delays
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
ppval	Evaluate piecewise polynomial
spline	Cubic spline data interpolation
tsearchn	N-D closest simplex search
unmkpp	Piecewise polynomial details

Delaunay Triangulation and Tessellation

delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search

Convex Hull

convhull	Convex hull
convhulln	N-D convex hull
patch	Create patch graphics object
plot	2-D line plot
trisurf	Triangular surface plot

Voronoi Diagrams

dsearch	Search Delaunay triangulation for nearest point
patch	Create patch graphics object
plot	2-D line plot

voronoi	Voronoi diagram
voronoin	N-D Voronoi diagram

Domain Generation

meshgrid	Generate `X` and `Y` arrays for 3-D plots
ndgrid	Generate arrays for N-D functions and interpolation

Cartesian Coordinate System Conversion

cart2pol	Transform Cartesian coordinates to polar or cylindrical
cart2sph	Transform Cartesian coordinates to spherical
pol2cart	Transform polar or cylindrical coordinates to Cartesian
sph2cart	Transform spherical coordinates to Cartesian

Nonlinear Numerical Methods

Ordinary Differential Equations (IVP) (p. 1-32)	Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations (p. 1-33)	Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Boundary Value Problems (p. 1-33)	Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution

Partial Differential Equations (p. 1-34)	Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution
Optimization (p. 1-34)	Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature) (p. 1-34)	Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

Ordinary Differential Equations (IVP)

decic	Compute consistent initial conditions for `ode15i`
deval	Evaluate solution of differential equation problem
ode15i	Solve fully implicit differential equations, variable order method
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb	Solve initial value problems for ordinary differential equations
odefile	Define differential equation problem for ordinary differential equation solvers
odeget	Ordinary differential equation options parameters
odeset	Create or alter options structure for ordinary differential equation solvers
odextend	Extend solution of initial value problem for ordinary differential equation

Delay Differential Equations

dde23	Solve delay differential equations (DDEs) with constant delays
ddeget	Extract properties from delay differential equations options structure
ddesd	Solve delay differential equations (DDEs) with general delays
ddeset	Create or alter delay differential equations options structure
deval	Evaluate solution of differential equation problem

Boundary Value Problems

bvp4c	Solve boundary value problems for ordinary differential equations
bvp5c	Solve boundary value problems for ordinary differential equations
bvpget	Extract properties from options structure created with `bvpset`
bvpinit	Form initial guess for `bvp4c`
bvpset	Create or alter options structure of boundary value problem
bvpxtend	Form guess structure for extending boundary value solutions
deval	Evaluate solution of differential equation problem

Partial Differential Equations

pdepe	Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D
pdeval	Evaluate numerical solution of PDE using output of pdepe

Optimization

fminbnd	Find minimum of single-variable function on fixed interval
fminsearch	Find minimum of unconstrained multivariable function using derivative-free method
fzero	Find root of continuous function of one variable
lsqnonneg	Solve nonnegative least-squares constraints problem
optimget	Optimization options values
optimset	Create or edit optimization options structure

Numerical Integration (Quadrature)

dblquad	Numerically evaluate double integral
quad	Numerically evaluate integral, adaptive Simpson quadrature
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature
quadl	Numerically evaluate integral, adaptive Lobatto quadrature

quadv	Vectorized quadrature
triplequad	Numerically evaluate triple integral

Specialized Math

airy	Airy functions
besselh	Bessel function of third kind (Hankel function)
besseli	Modified Bessel function of first kind
besselj	Bessel function of first kind
besselk	Modified Bessel function of second kind
bessely	Bessel function of second kind
beta	Beta function
betainc	Incomplete beta function
betaln	Logarithm of beta function
ellipj	Jacobi elliptic functions
ellipke	Complete elliptic integrals of first and second kind
erf, erfc, erfcx, erfinv, erfcinv	Error functions
expint	Exponential integral
gamma, gammainc, gammaln	Gamma functions
legendre	Associated Legendre functions
psi	Psi (polygamma) function

Sparse Matrices

Elementary Sparse Matrices (p. 1-36)	Create random and nonrandom sparse matrices
Full to Sparse Conversion (p. 1-37)	Convert full matrix to sparse, sparse matrix to full
Working with Sparse Matrices (p. 1-37)	Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern.
Reordering Algorithms (p. 1-37)	Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
Linear Algebra (p. 1-38)	Compute norms, eigenvalues, factorizations, least squares, structural rank
Linear Equations (Iterative Methods) (p. 1-38)	Methods for conjugate and biconjugate gradients, residuals, lower quartile
Tree Operations (p. 1-39)	Elimination trees, tree plotting, factorization analysis

Elementary Sparse Matrices

spdiags	Extract and create sparse band and diagonal matrices
speye	Sparse identity matrix
sprand	Sparse uniformly distributed random matrix
sprandn	Sparse normally distributed random matrix
sprandsym	Sparse symmetric random matrix

Full to Sparse Conversion

find	Find indices and values of nonzero elements
full	Convert sparse matrix to full matrix
sparse	Create sparse matrix
spconvert	Import matrix from sparse matrix external format

Working with Sparse Matrices

issparse	Determine whether input is sparse
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
nzmax	Amount of storage allocated for nonzero matrix elements
spalloc	Allocate space for sparse matrix
spfun	Apply function to nonzero sparse matrix elements
spones	Replace nonzero sparse matrix elements with ones
spparms	Set parameters for sparse matrix routines
spy	Visualize sparsity pattern

Reordering Algorithms

amd	Approximate minimum degree permutation
colamd	Column approximate minimum degree permutation

colperm	Sparse column permutation based on nonzero count
dmperm	Dulmage-Mendelsohn decomposition
ldl	Block ldl' factorization for Hermitian indefinite matrices
randperm	Random permutation
symamd	Symmetric approximate minimum degree permutation
symrcm	Sparse reverse Cuthill-McKee ordering

Linear Algebra

cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
condest	1-norm condition number estimate
eigs	Find largest eigenvalues and eigenvectors of sparse matrix
ilu	Sparse incomplete LU factorization
luinc	Sparse incomplete LU factorization
normest	2-norm estimate
spaugment	Form least squares augmented system
sprank	Structural rank
svds	Find singular values and vectors

Linear Equations (Iterative Methods)

bicg	Biconjugate gradients method
bicgstab	Biconjugate gradients stabilized method

cgs	Conjugate gradients squared method
gmres	Generalized minimum residual method (with restarts)
lsqr	LSQR method
minres	Minimum residual method
pcg	Preconditioned conjugate gradients method
qmr	Quasi-minimal residual method
symmlq	Symmetric LQ method

Tree Operations

etree	Elimination tree
etreeplot	Plot elimination tree
gplot	Plot nodes and links representing adjacency matrix
symbfact	Symbolic factorization analysis
treelayout	Lay out tree or forest
treeplot	Plot picture of tree

Math Constants

eps	Floating-point relative accuracy
i	Imaginary unit
Inf	Infinity
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
j	Imaginary unit

NaN	Not-a-Number
pi	Ratio of circle's circumference to its diameter, π
realmax	Largest positive floating-point number
realmin	Smallest positive normalized floating-point number

Data Analysis

Basic Operations

cumprod	Cumulative product
cumsum	Cumulative sum
prod	Product of array elements
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
sum	Sum of array elements

Descriptive Statistics

corrcoef	Correlation coefficients
cov	Covariance matrix
max	Largest elements in array
mean	Average or mean value of array
median	Median value of array

min	Smallest elements in array
mode	Most frequent values in array
std	Standard deviation
var	Variance

Filtering and Convolution

conv	Convolution and polynomial multiplication
conv2	2-D convolution
convn	N-D convolution
deconv	Deconvolution and polynomial division
detrend	Remove linear trends
filter	1-D digital filter
filter2	2-D digital filter

Interpolation and Regression

interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpn	N-D data interpolation (table lookup)
mldivide \, mrdivide /	Left or right matrix division
polyfit	Polynomial curve fitting
polyval	Polynomial evaluation

Fourier Transforms

abs	Absolute value and complex magnitude
angle	Phase angle
cplxpair	Sort complex numbers into complex conjugate pairs
fft	Discrete Fourier transform
fft2	2-D discrete Fourier transform
fftn	N-D discrete Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
fftw	Interface to FFTW library run-time algorithm tuning control
ifft	Inverse discrete Fourier transform
ifft2	2-D inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform
ifftshift	Inverse FFT shift
nextpow2	Next higher power of 2
unwrap	Correct phase angles to produce smoother phase plots

Derivatives and Integrals

cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives

gradient	Numerical gradient
polyder	Polynomial derivative
polyint	Integrate polynomial analytically
trapz	Trapezoidal numerical integration

Time Series Objects

General Purpose (p. 1-44)	Combine `timeseries` objects, query and set `timeseries` object properties, plot `timeseries` objects
Data Manipulation (p. 1-45)	Add or delete data, manipulate `timeseries` objects
Event Data (p. 1-46)	Add or delete events, create new `timeseries` objects based on event data
Descriptive Statistics (p. 1-46)	Descriptive statistics for `timeseries` objects

General Purpose

get (timeseries)	Query `timeseries` object property values
getdatasamplesize	Size of data sample in `timeseries` object
getqualitydesc	Data quality descriptions
isempty (timeseries)	Determine whether `timeseries` object is empty
length (timeseries)	Length of time vector
plot (timeseries)	Plot time series
set (timeseries)	Set properties of `timeseries` object
size (timeseries)	Size of `timeseries` object

timeseries	Create `timeseries` object
tsdata.event	Construct event object for `timeseries` object
tsprops	Help on `timeseries` object properties
tstool	Open Time Series Tools GUI

Data Manipulation

addsample	Add data sample to `timeseries` object
ctranspose (timeseries)	Transpose `timeseries` object
delsample	Remove sample from `timeseries` object
detrend (timeseries)	Subtract mean or best-fit line and all `NaN`s from time series
filter (timeseries)	Shape frequency content of time series
getabstime (timeseries)	Extract date-string time vector into cell array
getinterpmethod	Interpolation method for `timeseries` object
getsampleusingtime (timeseries)	Extract data samples into new `timeseries` object
idealfilter (timeseries)	Apply ideal (noncausal) filter to `timeseries` object
resample (timeseries)	Select or interpolate `timeseries` data using new time vector
setabstime (timeseries)	Set times of `timeseries` object as date strings
setinterpmethod	Set default interpolation method for `timeseries` object

synchronize	Synchronize and resample two `timeseries` objects using common time vector
transpose (timeseries)	Transpose `timeseries` object
vertcat (timeseries)	Vertical concatenation of `timeseries` objects

Event Data

addevent	Add event to `timeseries` object
delevent	Remove `tsdata.event` objects from `timeseries` object
gettsafteratevent	New `timeseries` object with samples occurring at or after event
gettsafterevent	New `timeseries` object with samples occurring after event
gettsatevent	New `timeseries` object with samples occurring at event
gettsbeforeatevent	New `timeseries` object with samples occurring before or at event
gettsbeforeevent	New `timeseries` object with samples occurring before event
gettsbetweenevents	New `timeseries` object with samples occurring between events

Descriptive Statistics

iqr (timeseries)	Interquartile range of `timeseries` data
max (timeseries)	Maximum value of `timeseries` data
mean (timeseries)	Mean value of `timeseries` data
median (timeseries)	Median value of `timeseries` data

min (timeseries)	Minimum value of `timeseries` data
std (timeseries)	Standard deviation of `timeseries` data
sum (timeseries)	Sum of `timeseries` data
var (timeseries)	Variance of `timeseries` data

Time Series Collections

General Purpose (p. 1-47)	Query and set `tscollection` object properties, plot `tscollection` objects
Data Manipulation (p. 1-48)	Add or delete data, manipulate `tscollection` objects

General Purpose

get (tscollection)	Query `tscollection` object property values
isempty (tscollection)	Determine whether `tscollection` object is empty
length (tscollection)	Length of time vector
plot (timeseries)	Plot time series
set (tscollection)	Set properties of `tscollection` object
size (tscollection)	Size of `tscollection` object
tscollection	Create `tscollection` object
tstool	Open Time Series Tools GUI

Data Manipulation

addsampletocollection	Add sample to `tscollection` object
addts	Add `timeseries` object to `tscollection` object
delsamplefromcollection	Remove sample from `tscollection` object
getabstime (tscollection)	Extract date-string time vector into cell array
getsampleusingtime (tscollection)	Extract data samples into new `tscollection` object
gettimeseriesnames	Cell array of names of `timeseries` objects in `tscollection` object
horzcat (tscollection)	Horizontal concatenation for `tscollection` objects
removets	Remove `timeseries` objects from `tscollection` object
resample (tscollection)	Select or interpolate data in `tscollection` using new time vector
setabstime (tscollection)	Set times of `tscollection` object as date strings
settimeseriesnames	Change name of `timeseries` object in `tscollection`
vertcat (tscollection)	Vertical concatenation for `tscollection` objects

Programming and Data Types

Data Types

Cell Arrays (p. 1-53)	Data of varying types and sizes stored in cells of array
Function Handles (p. 1-54)	Invoke a function indirectly via handle
MATLAB Classes and Objects (p. 1-55)	MATLAB object-oriented class system
Java Classes and Objects (p. 1-55)	Access Java classes through MATLAB interface
Data Type Identification (p. 1-57)	Determine data type of a variable

Numeric Types

arrayfun	Apply function to each element of array
cast	Cast variable to different data type
cat	Concatenate arrays along specified dimension
class	Create object or return class of object
find	Find indices and values of nonzero elements
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
intwarning	Control state of integer warnings
ipermute	Inverse permute dimensions of N-D array
isa	Determine whether input is object of given class
isequal	Test arrays for equality

isequalwithequalnans	Test arrays for equality, treating NaNs as equal
isfinite	Array elements that are finite
isinf	Array elements that are infinite
isnan	Array elements that are NaN
isnumeric	Determine whether input is numeric array
isreal	Determine whether input is real array
isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
permute	Rearrange dimensions of N-D array
realmax	Largest positive floating-point number
realmin	Smallest positive normalized floating-point number
reshape	Reshape array
squeeze	Remove singleton dimensions
zeros	Create array of all zeros

Characters and Strings

See “String Functions” on page 1-63 for all string-related functions.

cellstr	Create cell array of strings from character array
char	Convert to character array (string)
eval	Execute string containing MATLAB expression
findstr	Find string within another, longer string

isstr	Determine whether input is character array
regexp, regexpi	Match regular expression
sprintf	Write formatted data to string
sscanf	Read formatted data from string
strcat	Concatenate strings horizontally
strcmp, strcmpi	Compare strings
strings	MATLAB string handling
strjust	Justify character array
strmatch	Find possible matches for string
strread	Read formatted data from string
strrep	Find and replace substring
strtrim	Remove leading and trailing white space from string
strvcat	Concatenate strings vertically

Structures

arrayfun	Apply function to each element of array
cell2struct	Convert cell array to structure array
class	Create object or return class of object
deal	Distribute inputs to outputs
fieldnames	Field names of structure, or public fields of object
getfield	Field of structure array
isa	Determine whether input is object of given class
isequal	Test arrays for equality

isfield	Determine whether input is structure array field
isscalar	Determine whether input is scalar
isstruct	Determine whether input is structure array
isvector	Determine whether input is vector
orderfields	Order fields of structure array
rmfield	Remove fields from structure
setfield	Set value of structure array field
struct	Create structure array
struct2cell	Convert structure to cell array
structfun	Apply function to each field of scalar structure

Cell Arrays

cell	Construct cell array
cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array
celldisp	Cell array contents
cellfun	Apply function to each cell in cell array
cellplot	Graphically display structure of cell array
cellstr	Create cell array of strings from character array
class	Create object or return class of object
deal	Distribute inputs to outputs

isa	Determine whether input is object of given class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
isequal	Test arrays for equality
isscalar	Determine whether input is scalar
isvector	Determine whether input is vector
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
struct2cell	Convert structure to cell array

Function Handles

class	Create object or return class of object
feval	Evaluate function
func2str	Construct function name string from function handle
functions	Information about function handle
function_handle (@)	Handle used in calling functions indirectly
isa	Determine whether input is object of given class
isequal	Test arrays for equality
str2func	Construct function handle from function name string

MATLAB Classes and Objects

class	Create object or return class of object
fieldnames	Field names of structure, or public fields of object
inferiorto	Establish inferior class relationship
isa	Determine whether input is object of given class
isobject	Determine whether input is MATLAB OOPs object
loadobj	User-defined extension of `load` function for user objects
methods	Information on class methods
methodsview	Information on class methods in separate window
saveobj	User-defined extension of `save` function for user objects
subsasgn	Subscripted assignment for objects
subsindex	Subscripted indexing for objects
subsref	Subscripted reference for objects
substruct	Create structure argument for `subsasgn` or `subsref`
superiorto	Establish superior class relationship

Java Classes and Objects

cell	Construct cell array
class	Create object or return class of object
clear	Remove items from workspace, freeing up system memory
depfun	List dependencies of M-file or P-file

exist	Check existence of variable, function, directory, or Java class
fieldnames	Field names of structure, or public fields of object
im2java	Convert image to Java image
import	Add package or class to current Java import list
inmem	Names of M-files, MEX-files, Java classes in memory
isa	Determine whether input is object of given class
isjava	Determine whether input is Java object
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Set and get dynamic Java class path
javaMethod	Invoke Java method
javaObject	Construct Java object
javarmpath	Remove entries from dynamic Java class path
methods	Information on class methods
methodsview	Information on class methods in separate window
usejava	Determine whether Java feature is supported in MATLAB
which	Locate functions and files

Data Type Identification

is*	Detect state
isa	Determine whether input is object of given class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isfield	Determine whether input is structure array field
isfloat	Determine whether input is floating-point array
isinteger	Determine whether input is integer array
isjava	Determine whether input is Java object
islogical	Determine whether input is logical array
isnumeric	Determine whether input is numeric array
isobject	Determine whether input is MATLAB OOPs object
isreal	Determine whether input is real array
isstr	Determine whether input is character array
isstruct	Determine whether input is structure array

validateattributes	Check validity of array
who, whos	List variables in workspace

Data Type Conversion

Numeric (p. 1-58)	Convert data of one numeric type to another numeric type
String to Numeric (p. 1-58)	Convert characters to numeric equivalent
Numeric to String (p. 1-59)	Convert numeric to character equivalent
Other Conversions (p. 1-59)	Convert to structure, cell array, function handle, etc.

Numeric

cast	Cast variable to different data type
double	Convert to double precision
int8, int16, int32, int64	Convert to signed integer
single	Convert to single precision
typecast	Convert data types without changing underlying data
uint8, uint16, uint32, uint64	Convert to unsigned integer

String to Numeric

base2dec	Convert base N number string to decimal number
bin2dec	Convert binary number string to decimal number
cast	Cast variable to different data type

hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
str2double	Convert string to double-precision value
str2num	Convert string to number
unicode2native	Convert Unicode characters to numeric bytes

Numeric to String

cast	Cast variable to different data type
char	Convert to character array (string)
dec2base	Convert decimal to base N number in string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
int2str	Convert integer to string
mat2str	Convert matrix to string
native2unicode	Convert numeric bytes to Unicode characters
num2str	Convert number to string

Other Conversions

cell2mat	Convert cell array of matrices to single matrix
cell2struct	Convert cell array to structure array

datestr	Convert date and time to string format
func2str	Construct function name string from function handle
logical	Convert numeric values to logical
mat2cell	Divide matrix into cell array of matrices
num2cell	Convert numeric array to cell array
num2hex	Convert singles and doubles to IEEE hexadecimal strings
str2func	Construct function handle from function name string
str2mat	Form blank-padded character matrix from strings
struct2cell	Convert structure to cell array

Operators and Special Characters

Arithmetic Operators (p. 1-60)	Plus, minus, power, left and right divide, transpose, etc.
Relational Operators (p. 1-61)	Equal to, greater than, less than or equal to, etc.
Logical Operators (p. 1-61)	Element-wise and short circuit and, or, not
Special Characters (p. 1-62)	Array constructors, line continuation, comments, etc.

Arithmetic Operators

+	Plus
-	Minus

.	Decimal point
=	Assignment
*	Matrix multiplication
/	Matrix right division
\	Matrix left division
^	Matrix power
'	Matrix transpose
.*	Array multiplication (element-wise)
./	Array right division (element-wise)
.\	Array left division (element-wise)
.^	Array power (element-wise)
.'	Array transpose

Relational Operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Logical Operators

See also "Logical Functions" on page 1-66 for functions like `xor`, `all`, `any`, etc.

&&	Logical AND
||	Logical OR
&	Logical AND for arrays

|	Logical OR for arrays
~	Logical NOT

Special Characters

:	Create vectors, subscript arrays, specify for-loop iterations
( )	Pass function arguments, prioritize operators
[ ]	Construct array, concatenate elements, specify multiple outputs from function
{ }	Construct cell array, index into cell array
.	Insert decimal point, define structure field, reference methods of object
.( )	Reference dynamic field of structure
..	Reference parent directory
...	Continue statement to next line
,	Separate rows of array, separate function input/output arguments, separate commands
;	Separate columns of array, suppress output from current command
%	Insert comment line into code
%{ %}	Insert block of comments into code
!	Issue command to operating system
' '	Construct character array
@	Construct function handle, reference class directory

String Functions

Description of Strings in MATLAB

strings	MATLAB string handling

String Creation

blanks	Create string of blank characters
cellstr	Create cell array of strings from character array
char	Convert to character array (string)
sprintf	Write formatted data to string
strcat	Concatenate strings horizontally
strvcat	Concatenate strings vertically

String Identification

class	Create object or return class of object
isa	Determine whether input is object of given class
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isletter	Array elements that are alphabetic letters
isscalar	Determine whether input is scalar
isspace	Array elements that are space characters
isstrprop	Determine whether string is of specified category
isvector	Determine whether input is vector
validatestring	Check validity of text string

String Manipulation

deblank	Strip trailing blanks from end of string
lower	Convert string to lowercase
strjust	Justify character array
strrep	Find and replace substring
strtrim	Remove leading and trailing white space from string
upper	Convert string to uppercase

String Parsing

findstr	Find string within another, longer string
regexp, regexpi	Match regular expression
regexprep	Replace string using regular expression
regexptranslate	Translate string into regular expression
sscanf	Read formatted data from string
strfind	Find one string within another
strread	Read formatted data from string
strtok	Selected parts of string

String Evaluation

eval	Execute string containing MATLAB expression
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace

String Comparison

strcmp, strcmpi	Compare strings
strmatch	Find possible matches for string
strncmp, strncmpi	Compare first n characters of strings

Bit-wise Functions

bitand	Bitwise AND
bitcmp	Bitwise complement
bitget	Bit at specified position
bitmax	Maximum double-precision floating-point integer
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR
swapbytes	Swap byte ordering

Logical Functions

all	Determine whether all array elements are nonzero
and	Find logical AND of array or scalar inputs
any	Determine whether any array elements are nonzero
false	Logical 0 (false)
find	Find indices and values of nonzero elements
isa	Determine whether input is object of given class
iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
logical	Convert numeric values to logical

not	Find logical NOT of array or scalar input
or	Find logical OR of array or scalar inputs
true	Logical 1 (true)
xor	Logical exclusive-OR

See “Operators and Special Characters” on page 1-60 for logical operators.

Relational Functions

eq	Test for equality
ge	Test for greater than or equal to
gt	Test for greater than
le	Test for less than or equal to
lt	Test for less than
ne	Test for inequality

See “Operators and Special Characters” on page 1-60 for relational operators.

Set Functions

intersect	Find set intersection of two vectors
ismember	Array elements that are members of set
issorted	Determine whether set elements are in sorted order
setdiff	Find set difference of two vectors
setxor	Find set exclusive OR of two vectors
union	Find set union of two vectors
unique	Find unique elements of vector

Date and Time Functions

addtodate	Modify date number by field
calendar	Calendar for specified month
clock	Current time as date vector
cputime	Elapsed CPU time
date	Current date string
datenum	Convert date and time to serial date number
datestr	Convert date and time to string format
datevec	Convert date and time to vector of components
eomday	Last day of month
etime	Time elapsed between date vectors
now	Current date and time
weekday	Day of week

Programming in MATLAB

M-File Functions and Scripts (p. 1-69)	Declare functions, handle arguments, identify dependencies, etc.
Evaluation of Expressions and Functions (p. 1-70)	Evaluate expression in string, apply function to array, run script file, etc.
Timer Functions (p. 1-71)	Schedule execution of MATLAB commands
Variables and Functions in Memory (p. 1-72)	List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches

Control Flow (p. 1-73)	if-then-else, for loops, switch-case, try-catch
Error Handling (p. 1-74)	Generate warnings and errors, test for and catch errors, retrieve most recent error message
MEX Programming (p. 1-75)	Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

M-File Functions and Scripts

addOptional (inputParser)	Add optional argument to `inputParser` schema
addParamValue (inputParser)	Add parameter-value argument to `inputParser` schema
addRequired (inputParser)	Add required argument to `inputParser` schema
createCopy (inputParser)	Create copy of `inputParser` object
depdir	List dependent directories of M-file or P-file
depfun	List dependencies of M-file or P-file
echo	Echo M-files during execution
end	Terminate block of code, or indicate last array index
function	Declare M-file function
input	Request user input
inputname	Variable name of function input
inputParser	Construct input parser object
mfilename	Name of currently running M-file
namelengthmax	Maximum identifier length
nargchk	Validate number of input arguments

nargin, nargout	Number of function arguments
nargoutchk	Validate number of output arguments
parse (inputParser)	Parse and validate named inputs
pcode	Create preparsed pseudocode file (P-file)
script	Script M-file description
syntax	Two ways to call MATLAB functions
varargin	Variable length input argument list
varargout	Variable length output argument list

Evaluation of Expressions and Functions

ans	Most recent answer
arrayfun	Apply function to each element of array
assert	Generate error when condition is violated
builtin	Execute built-in function from overloaded method
cellfun	Apply function to each cell in cell array
echo	Echo M-files during execution
eval	Execute string containing MATLAB expression
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace
feval	Evaluate function

iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
pause	Halt execution temporarily
run	Run script that is not on current path
script	Script M-file description
structfun	Apply function to each field of scalar structure
symvar	Determine symbolic variables in expression
tic, toc	Measure performance using stopwatch timer

Timer Functions

delete (timer)	Remove timer object from memory
disp (timer)	Information about timer object
get (timer)	Timer object properties
isvalid (timer)	Determine whether timer object is valid
set (timer)	Configure or display timer object properties
start	Start timer(s) running
startat	Start timer(s) running at specified time
stop	Stop timer(s)
timer	Construct timer object
timerfind	Find timer objects

timerfindall	Find timer objects, including invisible objects
wait	Wait until timer stops running

Variables and Functions in Memory

ans	Most recent answer
assignin	Assign value to variable in specified workspace
datatipinfo	Produce short description of input variable
genvarname	Construct valid variable name from string
global	Declare global variables
inmem	Names of M-files, MEX-files, Java classes in memory
isglobal	Determine whether input is global variable
mislocked	Determine whether M-file or MEX-file cannot be cleared from memory
mlock	Prevent clearing M-file or MEX-file from memory
munlock	Allow clearing M-file or MEX-file from memory
namelengthmax	Maximum identifier length
pack	Consolidate workspace memory
persistent	Define persistent variable
rehash	Refresh function and file system path caches

Control Flow

break	Terminate execution of `for` or `while` loop
case	Execute block of code if condition is `true`
catch	Specify how to respond to error in `try` statement
continue	Pass control to next iteration of `for` or `while` loop
else	Execute statements if condition is false
elseif	Execute statements if additional condition is true
end	Terminate block of code, or indicate last array index
error	Display message and abort function
for	Execute block of code specified number of times
if	Execute statements if condition is true
otherwise	Default part of switch statement
return	Return to invoking function
switch	Switch among several cases, based on expression
try	Attempt to execute block of code, and catch errors
while	Repeatedly execute statements while condition is true

Error Handling

addCause (MException)	Append `MException` objects
assert	Generate error when condition is violated
catch	Specify how to respond to error in `try` statement
disp (MException)	Display `MException` object
eq (MException)	Compare `MException` objects for equality
error	Display message and abort function
ferror	Query MATLAB about errors in file input or output
getReport (MException)	Get error message for exception
intwarning	Control state of integer warnings
isequal (MException)	Compare `MException` objects for equality
last (MException)	Last uncaught exception
lasterr	Last error message
lasterror	Last error message and related information
lastwarn	Last warning message
MException	Construct `MException` object
ne (MException)	Compare `MException` objects for inequality
rethrow	Reissue error
rethrow (MException)	Reissue existing exception
throw (MException)	Terminate function and issue exception

try	Attempt to execute block of code, and catch errors
warning	Warning message

MEX Programming

dbmex	Enable MEX-file debugging
inmem	Names of M-files, MEX-files, Java classes in memory
mex	Compile MEX-function from C, C++, or Fortran source code
mexext	MEX-filename extension

File I/O

File Name Construction (p. 1-76)	Get path, directory, filename information; construct filenames
Opening, Loading, Saving Files (p. 1-77)	Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-77)	Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-77)	Low-level operations that use a file identifier
Text Files (p. 1-78)	Delimited or formatted I/O to text files
XML Documents (p. 1-79)	Documents written in Extensible Markup Language
Spreadsheets (p. 1-79)	Excel and Lotus 1-2-3 files
Scientific Data (p. 1-80)	CDF, FITS, HDF formats
Audio and Audio/Video (p. 1-81)	General audio functions; SparcStation, WAVE, AVI files
Images (p. 1-83)	Graphics files
Internet Exchange (p. 1-84)	URL, FTP, zip, tar, and e-mail

To see a listing of file formats that are readable from MATLAB, go to file formats.

File Name Construction

filemarker	Character to separate file name and internal function name
fileparts	Parts of file name and path
filesep	Directory separator for current platform
fullfile	Build full filename from parts

tempdir	Name of system's temporary directory
tempname	Unique name for temporary file

Opening, Loading, Saving Files

daqread	Read Data Acquisition Toolbox (.daq) file
filehandle	Construct file handle object
importdata	Load data from disk file
load	Load workspace variables from disk
open	Open files based on extension
save	Save workspace variables to disk
uiimport	Open Import Wizard to import data
winopen	Open file in appropriate application (Windows)

Memory Mapping

disp (memmapfile)	Information about memmapfile object
get (memmapfile)	Memmapfile object properties
memmapfile	Construct memmapfile object

Low-Level File I/O

fclose	Close one or more open files
feof	Test for end-of-file
ferror	Query MATLAB about errors in file input or output

fgetl	Read line from file, discarding newline character
fgets	Read line from file, keeping newline character
fopen	Open file, or obtain information about open files
fprintf	Write formatted data to file
fread	Read binary data from file
frewind	Move file position indicator to beginning of open file
fscanf	Read formatted data from file
fseek	Set file position indicator
ftell	File position indicator
fwrite	Write binary data to file

Text Files

csvread	Read comma-separated value file
csvwrite	Write comma-separated value file
dlmread	Read ASCII-delimited file of numeric data into matrix
dlmwrite	Write matrix to ASCII-delimited file
textread	Read data from text file; write to multiple outputs
textscan	Read formatted data from text file or string

XML Documents

xmlread	Parse XML document and return Document Object Model node
xmlwrite	Serialize XML Document Object Model node
xslt	Transform XML document using XSLT engine

Spreadsheets

Microsoft Excel Functions (p. 1-79)	Read and write Microsoft Excel spreadsheet
Lotus 1-2-3 Functions (p. 1-79)	Read and write Lotus WK1 spreadsheet

Microsoft Excel Functions

xlsfinfo	Determine whether file contains Microsoft Excel (`.xls`) spreadsheet
xlsread	Read Microsoft Excel spreadsheet file (`.xls`)
xlswrite	Write Microsoft Excel spreadsheet file (`.xls`)

Lotus 1-2-3 Functions

wk1finfo	Determine whether file contains 1-2-3 WK1 worksheet
wk1read	Read Lotus 1-2-3 WK1 spreadsheet file into matrix
wk1write	Write matrix to Lotus 1-2-3 WK1 spreadsheet file

Scientific Data

Common Data Format (CDF) (p. 1-80)	Work with CDF files
Flexible Image Transport System (p. 1-80)	Work with FITS files
Hierarchical Data Format (HDF) (p. 1-81)	Work with HDF files
Band-Interleaved Data (p. 1-81)	Work with band-interleaved files

Common Data Format (CDF)

cdfepoch	Construct `cdfepoch` object for Common Data Format (CDF) export
cdfinfo	Information about Common Data Format (CDF) file
cdfread	Read data from Common Data Format (CDF) file
cdfwrite	Write data to Common Data Format (CDF) file
todatenum	Convert CDF epoch object to MATLAB datenum

Flexible Image Transport System

fitsinfo	Information about FITS file
fitsread	Read data from FITS file

Hierarchical Data Format (HDF)

hdf	Summary of MATLAB HDF4 capabilities
hdf5	Summary of MATLAB HDF5 capabilities
hdf5info	Information about HDF5 file
hdf5read	Read HDF5 file
hdf5write	Write data to file in HDF5 format
hdfinfo	Information about HDF4 or HDF-EOS file
hdfread	Read data from HDF4 or HDF-EOS file
hdftool	Browse and import data from HDF4 or HDF-EOS files

Band-Interleaved Data

multibandread	Read band-interleaved data from binary file
multibandwrite	Write band-interleaved data to file

Audio and Audio/Video

General (p. 1-82)	Create audio player object, obtain information about multimedia files, convert to/from audio signal
SPARCstation-Specific Sound Functions (p. 1-82)	Access NeXT/SUN (`.au`) sound files

Microsoft WAVE Sound Functions (p. 1-83)	Access Microsoft WAVE (`.wav`) sound files
Audio/Video Interleaved (AVI) Functions (p. 1-83)	Access Audio/Video interleaved (`.avi`) sound files

General

audioplayer	Create audio player object
audiorecorder	Create audio recorder object
beep	Produce beep sound
lin2mu	Convert linear audio signal to mu-law
mmfileinfo	Information about multimedia file
mmreader	Create multimedia reader object for reading video files
mu2lin	Convert mu-law audio signal to linear
read	Read video frame data from multimedia reader object
sound	Convert vector into sound
soundsc	Scale data and play as sound

SPARCstation-Specific Sound Functions

aufinfo	Information about NeXT/SUN (`.au`) sound file
auread	Read NeXT/SUN (`.au`) sound file
auwrite	Write NeXT/SUN (`.au`) sound file

Microsoft WAVE Sound Functions

wavfinfo	Information about Microsoft WAVE (`.wav`) sound file
wavplay	Play recorded sound on PC-based audio output device
wavread	Read Microsoft WAVE (`.wav`) sound file
wavrecord	Record sound using PC-based audio input device
wavwrite	Write Microsoft WAVE (`.wav`) sound file

Audio/Video Interleaved (AVI) Functions

addframe	Add frame to Audio/Video Interleaved (AVI) file
avifile	Create new Audio/Video Interleaved (AVI) file
aviinfo	Information about Audio/Video Interleaved (AVI) file
aviread	Read Audio/Video Interleaved (AVI) file
close (avifile)	Close Audio/Video Interleaved (AVI) file
movie2avi	Create Audio/Video Interleaved (AVI) movie from MATLAB movie

Images

exifread	Read EXIF information from JPEG and TIFF image files
im2java	Convert image to Java image

imfinfo	Information about graphics file
imread	Read image from graphics file
imwrite	Write image to graphics file

Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-84)	Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files
FTP Functions (p. 1-84)	Connect to FTP server, download from server, manage FTP files, close server connection

URL, Zip, Tar, E-Mail

gunzip	Uncompress GNU zip files
gzip	Compress files into GNU zip files
sendmail	Send e-mail message to address list
tar	Compress files into tar file
untar	Extract contents of tar file
unzip	Extract contents of zip file
urlread	Read content at URL
urlwrite	Save contents of URL to file
zip	Compress files into zip file

FTP Functions

ascii	Set FTP transfer type to ASCII
binary	Set FTP transfer type to binary

cd (ftp)	Change current directory on FTP server
close (ftp)	Close connection to FTP server
delete (ftp)	Remove file on FTP server
dir (ftp)	Directory contents on FTP server
ftp	Connect to FTP server, creating FTP object
mget	Download file from FTP server
mkdir (ftp)	Create new directory on FTP server
mput	Upload file or directory to FTP server
rename	Rename file on FTP server
rmdir (ftp)	Remove directory on FTP server

Graphics

Basic Plots and Graphs (p. 1-86)	Linear line plots, log and semilog plots
Plotting Tools (p. 1-87)	GUIs for interacting with plots
Annotating Plots (p. 1-87)	Functions for and properties of titles, axes labels, legends, mathematical symbols
Specialized Plotting (p. 1-88)	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images (p. 1-92)	Display image object, read and write graphics file, convert to movie frames
Printing (p. 1-92)	Printing and exporting figures to standard formats
Handle Graphics (p. 1-93)	Creating graphics objects, setting properties, finding handles

Basic Plots and Graphs

box	Axes border
errorbar	Plot error bars along curve
hold	Retain current graph in figure
LineSpec	Line specification string syntax
loglog	Log-log scale plot
plot	2-D line plot
plot3	3-D line plot
plotyy	2-D line plots with y-axes on both left and right side
polar	Polar coordinate plot

semilogx, semilogy	Semilogarithmic plots
subplot	Create axes in tiled positions

Plotting Tools

figurepalette	Show or hide figure palette
pan	Pan view of graph interactively
plotbrowser	Show or hide figure plot browser
plotedit	Interactively edit and annotate plots
plottools	Show or hide plot tools
propertyeditor	Show or hide property editor
rotate3d	Rotate 3-D view using mouse
showplottool	Show or hide figure plot tool
zoom	Turn zooming on or off or magnify by factor

Annotating Plots

annotation	Create annotation objects
clabel	Contour plot elevation labels
datacursormode	Enable or disable interactive data cursor mode
datetick	Date formatted tick labels
gtext	Mouse placement of text in 2-D view
legend	Graph legend for lines and patches
line	Create line object
rectangle	Create 2-D rectangle object
texlabel	Produce TeX format from character string

title	Add title to current axes
xlabel, ylabel, zlabel	Label *x*-, *y*-, and *z*-axis

Specialized Plotting

Area, Bar, and Pie Plots (p. 1-88)	1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-89)	Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-89)	Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-89)	Stair, step, and stem plots
Function Plots (p. 1-89)	Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-90)	Plots for showing distributions of data
Polygons and Surfaces (p. 1-90)	Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-91)	Plots of point distributions
Animation (p. 1-91)	Functions to create and play movies of plots

Area, Bar, and Pie Plots

area	Filled area 2-D plot
bar, barh	Plot bar graph (vertical and horizontal)
bar3, bar3h	Plot 3-D bar chart
pareto	Pareto chart
pie	Pie chart
pie3	3-D pie chart

Contour Plots

contour	Contour plot of matrix
contour3	3-D contour plot
contourc	Low-level contour plot computation
contourf	Filled 2-D contour plot
ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter

Direction and Velocity Plots

comet	2-D comet plot
comet3	3-D comet plot
compass	Plot arrows emanating from origin
feather	Plot velocity vectors
quiver	Quiver or velocity plot
quiver3	3-D quiver or velocity plot

Discrete Data Plots

stairs	Stairstep graph
stem	Plot discrete sequence data
stem3	Plot 3-D discrete sequence data

Function Plots

ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter
ezmesh	Easy-to-use 3-D mesh plotter

ezmeshc	Easy-to-use combination mesh/contour plotter
ezplot	Easy-to-use function plotter
ezplot3	Easy-to-use 3-D parametric curve plotter
ezpolar	Easy-to-use polar coordinate plotter
ezsurf	Easy-to-use 3-D colored surface plotter
ezsurfc	Easy-to-use combination surface/contour plotter
fplot	Plot function between specified limits

Histograms

hist	Histogram plot
histc	Histogram count
rose	Angle histogram plot

Polygons and Surfaces

convhull	Convex hull
cylinder	Generate cylinder
delaunay	Delaunay triangulation
delaunay3	3-D Delaunay tessellation
delaunayn	N-D Delaunay tessellation
dsearch	Search Delaunay triangulation for nearest point
dsearchn	N-D nearest point search
ellipsoid	Generate ellipsoid

fill	Filled 2-D polygons
fill3	Filled 3-D polygons
inpolygon	Points inside polygonal region
pcolor	Pseudocolor (checkerboard) plot
polyarea	Area of polygon
rectint	Rectangle intersection area
ribbon	Ribbon plot
slice	Volumetric slice plot
sphere	Generate sphere
tsearch	Search for enclosing Delaunay triangle
tsearchn	N-D closest simplex search
voronoi	Voronoi diagram
waterfall	Waterfall plot

Scatter/Bubble Plots

plotmatrix	Scatter plot matrix
scatter	Scatter plot
scatter3	3-D scatter plot

Animation

frame2im	Convert movie frame to indexed image
getframe	Capture movie frame
im2frame	Convert image to movie frame

movie	Play recorded movie frames
noanimate	Change `EraseMode` of all objects to `normal`

Bit-Mapped Images

frame2im	Convert movie frame to indexed image
im2frame	Convert image to movie frame
im2java	Convert image to Java image
image	Display image object
imagesc	Scale data and display image object
imfinfo	Information about graphics file
imformats	Manage image file format registry
imread	Read image from graphics file
imwrite	Write image to graphics file
ind2rgb	Convert indexed image to RGB image

Printing

frameedit	Edit print frames for Simulink and Stateflow block diagrams
hgexport	Export figure
orient	Hardcopy paper orientation
print, printopt	Print figure or save to file and configure printer defaults
printdlg	Print dialog box

printpreview	Preview figure to print
saveas	Save figure or Simulink block diagram using specified format

Handle Graphics

Finding and Identifying Graphics Objects (p. 1-93)	Find and manipulate graphics objects via their handles
Object Creation Functions (p. 1-94)	Constructors for core graphics objects
Plot Objects (p. 1-94)	Property descriptions for plot objects
Figure Windows (p. 1-95)	Control and save figures
Axes Operations (p. 1-96)	Operate on axes objects
Operating on Object Properties (p. 1-96)	Query, set, and link object properties

Finding and Identifying Graphics Objects

allchild	Find all children of specified objects
ancestor	Ancestor of graphics object
copyobj	Copy graphics objects and their descendants
delete	Remove files or graphics objects
findall	Find all graphics objects
findfigs	Find visible offscreen figures
findobj	Locate graphics objects with specific properties
gca	Current axes handle
gcbf	Handle of figure containing object whose callback is executing

gcbo	Handle of object whose callback is executing
gco	Handle of current object
get	Query object properties
ishandle	Is object handle valid
propedit	Open Property Editor
set	Set object properties

Object Creation Functions

axes	Create axes graphics object
figure	Create figure graphics object
hggroup	Create hggroup object
hgtransform	Create hgtransform graphics object
image	Display image object
light	Create light object
line	Create line object
patch	Create patch graphics object
rectangle	Create 2-D rectangle object
root object	Root object properties
surface	Create surface object
text	Create text object in current axes
uicontextmenu	Create context menu

Plot Objects

Annotation Arrow Properties	Define annotation arrow properties
Annotation Doublearrow Properties	Define annotation doublearrow properties

Annotation Ellipse Properties	Define annotation ellipse properties
Annotation Line Properties	Define annotation line properties
Annotation Rectangle Properties	Define annotation rectangle properties
Annotation Textarrow Properties	Define annotation textarrow properties
Annotation Textbox Properties	Define annotation textbox properties
Areaseries Properties	Define areaseries properties
Barseries Properties	Define barseries properties
Contourgroup Properties	Define contourgroup properties
Errorbarseries Properties	Define errorbarseries properties
Image Properties	Define image properties
Lineseries Properties	Define lineseries properties
Quivergroup Properties	Define quivergroup properties
Scattergroup Properties	Define scattergroup properties
Stairseries Properties	Define stairseries properties
Stemseries Properties	Define stemseries properties
Surfaceplot Properties	Define surfaceplot properties

Figure Windows

clf	Clear current figure window
close	Remove specified figure
closereq	Default figure close request function
drawnow	Flushes event queue and updates figure window
gcf	Current figure handle
hgload	Load Handle Graphics object hierarchy from file

hgsave	Save Handle Graphics object hierarchy to file
newplot	Determine where to draw graphics objects
opengl	Control OpenGL rendering
refresh	Redraw current figure
saveas	Save figure or Simulink block diagram using specified format

Axes Operations

axis	Axis scaling and appearance
box	Axes border
cla	Clear current axes
gca	Current axes handle
grid	Grid lines for 2-D and 3-D plots
ishold	Current hold state
makehgtform	Create 4-by-4 transform matrix

Operating on Object Properties

get	Query object properties
linkaxes	Synchronize limits of specified 2-D axes
linkprop	Keep same value for corresponding properties
refreshdata	Refresh data in graph when data source is specified
set	Set object properties

3-D Visualization

Surface and Mesh Plots

Creating Surfaces and Meshes

hidden	Remove hidden lines from mesh plot
mesh, meshc, meshz	Mesh plots
peaks	Example function of two variables
surf, surfc	3-D shaded surface plot
surface	Create surface object
surfl	Surface plot with colormap-based lighting

tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot

Domain Generation

griddata	Data gridding
meshgrid	Generate `X` and `Y` arrays for 3-D plots

Color Operations

brighten	Brighten or darken colormap
caxis	Color axis scaling
colorbar	Colorbar showing color scale
colordef	Set default property values to display different color schemes
colormap	Set and get current colormap
colormapeditor	Start colormap editor
ColorSpec	Color specification
graymon	Set default figure properties for grayscale monitors
hsv2rgb	Convert HSV colormap to RGB colormap
rgb2hsv	Convert RGB colormap to HSV colormap
rgbplot	Plot colormap
shading	Set color shading properties
spinmap	Spin colormap

surfnorm	Compute and display 3-D surface normals
whitebg	Change axes background color

Colormaps

contrast	Grayscale colormap for contrast enhancement

View Control

Controlling the Camera Viewpoint (p. 1-99)	Orbiting, dollying, pointing, rotating camera positions and setting fields of view
Setting the Aspect Ratio and Axis Limits (p. 1-100)	Specifying what portions of axes to view and how to scale them
Object Manipulation (p. 1-100)	Panning, rotating, and zooming views
Selecting Region of Interest (p. 1-101)	Interactively identifying rectangular regions

Controlling the Camera Viewpoint

camdolly	Move camera position and target
cameratoolbar	Control camera toolbar programmatically
camlookat	Position camera to view object or group of objects
camorbit	Rotate camera position around camera target
campan	Rotate camera target around camera position

campos	Set or query camera position
camproj	Set or query projection type
camroll	Rotate camera about view axis
camtarget	Set or query location of camera target
camup	Set or query camera up vector
camva	Set or query camera view angle
camzoom	Zoom in and out on scene
makehgtform	Create 4-by-4 transform matrix
view	Viewpoint specification
viewmtx	View transformation matrices

Setting the Aspect Ratio and Axis Limits

daspect	Set or query axes data aspect ratio
pbaspect	Set or query plot box aspect ratio
xlim, ylim, zlim	Set or query axis limits

Object Manipulation

pan	Pan view of graph interactively
reset	Reset graphics object properties to their defaults
rotate	Rotate object in specified direction
rotate3d	Rotate 3-D view using mouse
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
zoom	Turn zooming on or off or magnify by factor

Selecting Region of Interest

dragrect	Drag rectangles with mouse
rbbox	Create rubberband box for area selection

Lighting

camlight	Create or move light object in camera coordinates
diffuse	Calculate diffuse reflectance
light	Create light object
lightangle	Create or position `light` object in spherical coordinates
lighting	Specify lighting algorithm
material	Control reflectance properties of surfaces and patches
specular	Calculate specular reflectance

Transparency

alim	Set or query axes alpha limits
alpha	Set transparency properties for objects in current axes
alphamap	Specify figure alphamap (transparency)

Volume Visualization

coneplot	Plot velocity vectors as cones in 3-D vector field
contourslice	Draw contours in volume slice planes
curl	Compute curl and angular velocity of vector field
divergence	Compute divergence of vector field
flow	Simple function of three variables
interpstreamspeed	Interpolate stream-line vertices from flow speed
isocaps	Compute isosurface end-cap geometry
isocolors	Calculate isosurface and patch colors
isonormals	Compute normals of isosurface vertices
isosurface	Extract isosurface data from volume data
reducepatch	Reduce number of patch faces
reducevolume	Reduce number of elements in volume data set
shrinkfaces	Reduce size of patch faces
slice	Volumetric slice plot
smooth3	Smooth 3-D data
stream2	Compute 2-D streamline data
stream3	Compute 3-D streamline data
streamline	Plot streamlines from 2-D or 3-D vector data
streamparticles	Plot stream particles
streamribbon	3-D stream ribbon plot from vector volume data

streamslice	Plot streamlines in slice planes
streamtube	Create 3-D stream tube plot
subvolume	Extract subset of volume data set
surf2patch	Convert surface data to patch data
volumebounds	Coordinate and color limits for volume data

Creating Graphical User Interfaces

Predefined Dialog Boxes

dialog	Create and display dialog box
errordlg	Create and open error dialog box
export2wsdlg	Export variables to workspace
helpdlg	Create and open help dialog box
inputdlg	Create and open input dialog box
listdlg	Create and open list-selection dialog box
msgbox	Create and open message box
printdlg	Print dialog box
printpreview	Preview figure to print
questdlg	Create and open question dialog box
uigetdir	Open standard dialog box for selecting a directory

uigetfile	Open standard dialog box for retrieving files
uigetpref	Open dialog box for retrieving preferences
uiopen	Open file selection dialog box with appropriate file filters
uiputfile	Open standard dialog box for saving files
uisave	Open standard dialog box for saving workspace variables
uisetcolor	Open standard dialog box for setting object's `ColorSpec`
uisetfont	Open standard dialog box for setting object's font characteristics
waitbar	Open waitbar
warndlg	Open warning dialog box

Deploying User Interfaces

guidata	Store or retrieve GUI data
guihandles	Create structure of handles
movegui	Move GUI figure to specified location on screen
openfig	Open new copy or raise existing copy of saved figure

Developing User Interfaces

addpref	Add preference
getappdata	Value of application-defined data
getpref	Preference

ginput	Graphical input from mouse or cursor
guidata	Store or retrieve GUI data
guide	Open GUI Layout Editor
inspect	Open Property Inspector
isappdata	True if application-defined data exists
ispref	Test for existence of preference
rmappdata	Remove application-defined data
rmpref	Remove preference
setappdata	Specify application-defined data
setpref	Set preference
uigetpref	Open dialog box for retrieving preferences
uisetpref	Manage preferences used in `uigetpref`
waitfor	Wait for condition before resuming execution
waitforbuttonpress	Wait for key press or mouse-button click

User Interface Objects

menu	Generate menu of choices for user input
uibuttongroup	Create container object to exclusively manage radio buttons and toggle buttons
uicontextmenu	Create context menu
uicontrol	Create user interface control object

uimenu	Create menus on figure windows
uipanel	Create panel container object
uipushtool	Create push button on toolbar
uitoggletool	Create toggle button on toolbar
uitoolbar	Create toolbar on figure

Finding Objects from Callbacks

findall	Find all graphics objects
findfigs	Find visible offscreen figures
findobj	Locate graphics objects with specific properties
gcbf	Handle of figure containing object whose callback is executing
gcbo	Handle of object whose callback is executing

GUI Utility Functions

align	Align user interface controls (uicontrols) and axes
getpixelposition	Get component position in pixels
listfonts	List available system fonts
selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
setpixelposition	Set component position in pixels
textwrap	Wrapped string matrix for given uicontrol
uistack	Reorder visual stacking order of objects

Controlling Program Execution

uiresume, uiwait	Control program execution

External Interfaces

Dynamic Link Libraries (p. 1-109)	Access functions stored in external shared library (.dll) files
Java (p. 1-110)	Work with objects constructed from Java API and third-party class packages
Component Object Model and ActiveX (p. 1-111)	Integrate COM components into your application
Web Services (p. 1-113)	Communicate between applications over a network using SOAP and WSDL
Serial Port Devices (p. 1-113)	Read and write to devices connected to your computer's serial port

See also MATLAB C and Fortran API Reference for functions you can use in external routines that interact with MATLAB programs and the data in MATLAB workspaces.

Dynamic Link Libraries

calllib	Call function in external library
libfunctions	Information on functions in external library
libfunctionsview	Create window displaying information on functions in external library
libisloaded	Determine whether external library is loaded
libpointer	Create pointer object for use with external libraries
libstruct	Construct structure as defined in external library

loadlibrary	Load external library into MATLAB
unloadlibrary	Unload external library from memory

Java

class	Create object or return class of object
fieldnames	Field names of structure, or public fields of object
import	Add package or class to current Java import list
inspect	Open Property Inspector
isa	Determine whether input is object of given class
isjava	Determine whether input is Java object
ismethod	Determine whether input is object method
isprop	Determine whether input is object property
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Set and get dynamic Java class path
javaMethod	Invoke Java method
javaObject	Construct Java object
javarmpath	Remove entries from dynamic Java class path
methods	Information on class methods

methodsview	Information on class methods in separate window
usejava	Determine whether Java feature is supported in MATLAB

Component Object Model and ActiveX

actxcontrol	Create ActiveX control in figure window
actxcontrollist	List all currently installed ActiveX controls
actxcontrolselect	Open GUI to create ActiveX control
actxGetRunningServer	Get handle to running instance of Automation server
actxserver	Create COM server
addproperty	Add custom property to object
class	Create object or return class of object
delete (COM)	Remove COM control or server
deleteproperty	Remove custom property from object
enableservice	Enable, disable, or report status of Automation server
eventlisteners	List of events attached to listeners
events	List of events control can trigger
Execute	Execute MATLAB command in server
Feval (COM)	Evaluate MATLAB function in server
fieldnames	Field names of structure, or public fields of object
get (COM)	Get property value from interface, or display properties

GetCharArray	Get character array from server
GetFullMatrix	Get matrix from server
GetVariable	Get data from variable in server workspace
GetWorkspaceData	Get data from server workspace
inspect	Open Property Inspector
interfaces	List custom interfaces to COM server
invoke	Invoke method on object or interface, or display methods
isa	Determine whether input is object of given class
iscom	Is input COM object
isevent	Is input event
isinterface	Is input COM interface
ismethod	Determine whether input is object method
isprop	Determine whether input is object property
load (COM)	Initialize control object from file
MaximizeCommandWindow	Open server window on Windows desktop
methods	Information on class methods
methodsview	Information on class methods in separate window
MinimizeCommandWindow	Minimize size of server window
move	Move or resize control in parent window
propedit (COM)	Open built-in property page for control
PutCharArray	Store character array in server

PutFullMatrix	Store matrix in server
PutWorkspaceData	Store data in server workspace
Quit (COM)	Terminate MATLAB server
registerevent	Register event handler with control's event
release	Release interface
save (COM)	Serialize control object to file
set (COM)	Set object or interface property to specified value
unregisterallevents	Unregister all events for control
unregisterevent	Unregister event handler with control's event

Web Services

callSoapService	Send SOAP message off to endpoint
createClassFromWsdl	Create MATLAB object based on WSDL file
createSoapMessage	Create SOAP message to send to server
parseSoapResponse	Convert response string from SOAP server into MATLAB data types

Serial Port Devices

clear (serial)	Remove serial port object from MATLAB workspace
delete (serial)	Remove serial port object from memory
disp (serial)	Serial port object summary information

fclose (serial)	Disconnect serial port object from device
fgetl (serial)	Read line of text from device and discard terminator
fgets (serial)	Read line of text from device and include terminator
fopen (serial)	Connect serial port object to device
fprintf (serial)	Write text to device
fread (serial)	Read binary data from device
fscanf (serial)	Read data from device, and format as text
fwrite (serial)	Write binary data to device
get (serial)	Serial port object properties
instrcallback	Event information when event occurs
instrfind	Read serial port objects from memory to MATLAB workspace
instrfindall	Find visible and hidden serial port objects
isvalid (serial)	Determine whether serial port objects are valid
length (serial)	Length of serial port object array
load (serial)	Load serial port objects and variables into MATLAB workspace
readasync	Read data asynchronously from device
record	Record data and event information to file
save (serial)	Save serial port objects and variables to MAT-file
serial	Create serial port object

serialbreak	Send break to device connected to serial port
set (serial)	Configure or display serial port object properties
size (serial)	Size of serial port object array
stopasync	Stop asynchronous read and write operations

2

Functions — Alphabetical List

binary
bitand
bitcmp
bitget
bitmax
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
builddocsearchdb
builtin
bsxfun
bvp4c
bvp5c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit
campan
campos
camproj
camroll
camtarget
camup

camva
camzoom
cart2pol
cart2sph
case
cast
cat
catch
caxis
cd
cd (ftp)
cdf2rdf
cdfepoch
cdfinfo
cdfread
cdfwrite
ceil
cell
cell2mat
cell2struct
celldisp
cellfun
cellplot
cellstr
cgs
char
checkin
checkout
chol
cholinc
cholupdate
circshift
cla
clabel
class
clc
clear

clear (serial)
clf
clipboard
clock
close
close (avifile)
close (ftp)
closereq
cmopts
colamd
colmmd
colorbar
colordef
colormap
colormapeditor
ColorSpec
colperm
comet
comet3
commandhistory
commandwindow
compan
compass
complex
computer
cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contourgroup Properties
contourslice

contrast
conv
conv2
convhull
convhulln
convn
copyfile
copyobj
corrcoef
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
createClassFromWsdl
createCopy (inputParser)
createSoapMessage
cross
csc
cscd
csch
csvread
csvwrite
ctranspose (timeseries)
cumprod
cumsum
cumtrapz
curl
customverctrl
cylinder
daqread
daspect
datacursormode

datatipinfo
date
datenum
datestr
datetick
datevec
dbclear
dbcont
dbdown
dblquad
dbmex
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeadv
ddeexec
ddeget
ddeinit
ddepoke
ddereq
ddesd
ddeset
ddeterm
ddeunadv
deal
deblank
debug
dec2base
dec2bin
dec2hex
decic
deconv

del2
delaunay
delaunay3
delaunayn
delete
delete (COM)
delete (ftp)
delete (serial)
delete (timer)
deleteproperty
delevent
delsample
delsamplefromcollection
demo
depdir
depfun
det
detrend
detrend (timeseries)
deval
diag
dialog
diary
diff
diffuse
dir
dir (ftp)
disp
disp (memmapfile)
disp (MException)
disp (serial)
disp (timer)
display
divergence
dlmread
dlmwrite
dmperm

eventlisteners
events
Execute
exifread
exist
exit
exp
expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh
ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
factor
factorial
false
fclose
fclose (serial)
feather
feof
ferror
feval
Feval (COM)
fft
fft2
fftn
fftshift
fftw
fgetl

fgetl (serial)
fgets
fgets (serial)
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
File Formats
filemarker
fileparts
filehandle
filesep
fill
fill3
filter
filter (timeseries)
filter2
find
findall
findfigs
findobj
findstr
finish
fitsinfo
fitsread
fix
flipdim
fliplr
flipud
floor
flops
flow
fminbnd
fminsearch
fopen

fopen (serial)
for
format
fplot
fprintf
fprintf (serial)
frame2im
frameedit
fread
fread (serial)
freqspace
frewind
fscanf
fscanf (serial)
fseek
ftell
ftp
full
fullfile
func2str
function
function_handle (@)
functions
funm
fwrite
fwrite (serial)
fzero
gallery
gamma, gammainc, gammaln
gca
gcbf
gcbo
gcd
gcf
gco
ge
genpath

genvarname
get
get (COM)
get (memmapfile)
get (serial)
get (timer)
get (timeseries)
get (tscollection)
getabstime (timeseries)
getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getenv
getfield
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getReport (MException)
getsampleusingtime (timeseries)
getsampleusingtime (tscollection)
gettimeseriesnames
gettsafteratevent
gettsafterevent
gettsatevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents
GetVariable
GetWorkspaceData
ginput
global
gmres
gplot

grabcode
gradient
graymon
grid
griddata
griddata3
griddatan
gsvd
gt
gtext
guidata
guide
guihandles
gunzip
gzip
hadamard
hankel
hdf
hdf5
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
hex2dec
hex2num
hgexport
hggroup
Hggroup Properties
hgload

hgsave
hgtransform
Hgtransform Properties
hidden
hilb
hist
histc
hold
home
horzcat
horzcat (tscollection)
hostid
hsv2rgb
hypot
i
idealfilter (timeseries)
idivide
if
ifft
ifft2
ifftn
ifftshift
ilu
im2frame
im2java
imag
image
Image Properties
imagesc
imfinfo
imformats
import
importdata
imread
imwrite
ind2rgb
ind2sub

Inf
inferiorto
info
inline
inmem
inpolygon
input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8, int16, int32, int64
interfaces
interp1
interp1q
interp2
interp3
interpft
interpn
interpstreamspeed
intersect
intmax
intmin
intwarning
inv
invhilb
invoke
ipermute
iqr (timeseries)
is*
isa
isappdata
iscell

iscellstr
ischar
iscom
isdir
isempty
isempty (timeseries)
isempty (tscollection)
isequal
isequal (MException)
isequalwithequalnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishold
isinf
isinteger
isinterface
isjava
iskeyword
isletter
islogical
ismac
ismember
ismethod
isnan
isnumeric
isobject
isocaps
isocolors
isonormals
isosurface
ispc
ispref
isprime

isprop
isreal
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
isunix
isvalid (serial)
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
javaMethod
javaObject
javarmpath
keyboard
kron
last (MException)
lasterr
lasterror
lastwarn
lcm
ldl
ldivide, rdivide
le
legend
legendre
length
length (serial)

length (timeseries)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec
linkaxes
linkprop
linsolve
linspace
listdlg
listfonts
load
load (COM)
load (serial)
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor

lower
ls
lscov
lsqnonneg
lsqr
lt
lu
luinc
magic
makehgtform
mat2cell
mat2str
material
matlabcolon (matlab:)
matlabrc
matlabroot
matlab (UNIX)
matlab (Windows)
max
max (timeseries)
MaximizeCommandWindow
maxNumCompThreads
mean
mean (timeseries)
median
median (timeseries)
memmapfile
memory
MException
menu
mesh, meshc, meshz
meshgrid
methods
methodsview
mex
mexext
mfilename

mget
min
min (timeseries)
MinimizeCommandWindow
minres
mislocked
mkdir
mkdir (ftp)
mkpp
mldivide \, mrdivide /
mlint
mlintrpt
mlock
mmfileinfo
mmreader
mod
mode
more
move
movefile
movegui
movie
movie2avi
mput
msgbox
mtimes
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin, nargout
nargoutchk
native2unicode
nchoosek

ndgrid
ndims
ne
ne (MException)
newplot
nextpow2
nnz
noanimate
nonzeros
norm
normest
not
notebook
now
nthroot
null
num2cell
num2hex
num2str
numel
nzmax
ode15i
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
odefile
odeget
odeset
odextend
ones
open
openfig
opengl
openvar
optimget
optimset
or
ordeig
orderfields

ordqz
ordschur
orient
orth
otherwise
pack
padecoef
pagesetupdlg
pan
pareto
parse (inputParser)
parseSoapResponse
partialpath
pascal
patch
Patch Properties
path
path2rc
pathdef
pathsep
pathtool
pause
pbaspect
pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie
pie3

pinv
planerot
playshow
plot
plot (timeseries)
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval
polyvalm
pow2
power
ppval
prefdir
preferences
primes
print, printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
propertyeditor
psi

publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quadgk
quadl
quadv
questdlg
quit
Quit (COM)
quiver
quiver3
Quivergroup Properties
qz
rand
randn
randperm
rank
rat, rats
rbbox
rcond
read
readasync
real
reallog
realmax
realmin
realpow
realsqrt
record

rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp, regexpi
regexprep
regexptranslate
registerevent
rehash
release
rem
removets
rename
repmat
resample (timeseries)
resample (tscollection)
reset
reshape
residue
restoredefaultpath
rethrow
rethrow (MException)
return
rgb2hsv
rgbplot
ribbon
rmappdata
rmdir
rmdir (ftp)
rmfield
rmpath
rmpref
root object

Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rref
rsf2csf
run
save
save (COM)
save (serial)
saveas
saveobj
savepath
scatter
scatter3
Scattergroup Properties
schur
script
sec
secd
sech
selectmoveresize
semilogx, semilogy
sendmail
serial
serialbreak
set
set (COM)
set (serial)
set (timer)
set (timeseries)
set (tscollection)
setabstime (timeseries)

setabstime (tscollection)
setappdata
setdiff
setenv
setfield
setinterpmethod
setpixelposition
setpref
setstr
settimeseriesnames
setxor
shading
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size (serial)
size (timeseries)
size (tscollection)
slice
smooth3
sort
sortrows
sound
soundsc
spalloc
sparse
spaugment
spconvert
spdiags
specular
speye

spfun
sph2cart
sphere
spinmap
spline
spones
spparms
sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stairs
Stairseries Properties
start
startat
startup
std
std (timeseries)
stem
stem3
Stemseries Properties
stop
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp, strcmpi
stream2

stream3
streamline
streamparticles
streamribbon
streamslice
streamtube
strfind
strings
strjust
strmatch
strncmp, strncmpi
strread
strrep
strtok
strtrim
struct
struct2cell
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substruct
subvolume
sum
sum (timeseries)
superiorto
support
surf, surfc
surf2patch
surface
Surface Properties
Surfaceplot Properties
surfl

surfnorm
svd
svds
swapbytes
switch
symamd
symbfact
symmlq
symmmd
symrcm
symvar
synchronize
syntax
system
tan
tand
tanh
tar
tempdir
tempname
tetramesh
texlabel
text
Text Properties
textread
textscan
textwrap
throw (MException)
throwAsCaller (MException)
tic, toc
timer
timerfind
timerfindall
timeseries
title
todatenum
toeplitz

toolboxdir
trace
transpose (timeseries)
trapz
treelayout
treeplot
tril
trimesh
triplequad
triplot
trisurf
triu
true
try
tscollection
tsdata.event
tsearch
tsearchn
tsprops
tstool
type
typecast
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties
uigetdir
uigetfile
uigetpref
uiimport
uimenu
Uimenu Properties
uint8, uint16, uint32, uint64
uiopen
uipanel

vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (timeseries)
vertcat (tscollection)
view
viewmtx
volumebounds
voronoi
voronoin
wait
waitbar
waitfor
waitforbuttonpress
warndlg
warning
waterfall
wavfinfo
wavplay
wavread
wavrecord
wavwrite
web
weekday
what
whatsnew
which
while
whitebg
who, whos
wilkinson
winopen
winqueryreg
wk1finfo

wk1read
wk1write
workspace
xlabel, ylabel, zlabel
xlim, ylim, zlim
xlsfinfo
xlsread
xlswrite
xmlread
xmlwrite
xor
xslt
zeros
zip
zoom

pack

Purpose

Consolidate workspace memory

Syntax

```
pack
pack filename
pack('filename')
```

Description

`pack` frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from your base and global workspaces are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, `[]`).

MATLAB temporarily stores your workspace data in a file called `tp######.mat` (where `######` is a numeric value) that is located in your temporary directory. (You can use the command `dir(tempdir)` to see the files in this directory).

`pack filename` frees space in memory, temporarily storing workspace data in a file specified by `filename`. This file resides in your current working directory and, unless specified otherwise, has a `.mat` file extension.

`pack('filename')` is the function form of `pack`.

Remarks

You can only run `pack` from the MATLAB command line.

If you specify a `filename` argument, that file must reside in a directory for which you have write permission.

The `pack` function does not affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory.

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the `Out of memory` message from MATLAB, the `pack` function may find you some free memory without forcing you to delete variables.

The `pack` function frees space by

- Saving all variables in the base and global workspaces to a temporary file.
- Clearing all variables and functions from memory.
- Reloading the base and global workspace variables back from the temporary file and then deleting the file.

If you use `pack` and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, you can allocate larger matrices earlier in the MATLAB session and use these system-specific tips:

- UNIX: Ask your system manager to increase your swap space.
- Windows: Increase virtual memory using the Windows Control Panel.

To maintain persistent variables when you run pack, use `mlock` in the function.

Examples

Change the current directory to one that is writable, run `pack`, and return to the previous directory.

```
cwd = pwd;
cd(tempdir);
pack
cd(cwd)
```

See Also

`clear`, `memory`

padecoef

Purpose

Padé approximation of time delays

Syntax

```
[num,den] = padecoef(T,N)
```

Description

`[num,den] = padecoef(T,N)` returns the `N`th-order Padé approximation of the continuous-time delay T in transfer function form. The row vectors `num` and `den` contain the numerator and denominator coefficients in descending powers of T. Both are `N`th-order polynomials.

Class support for input T:

```
float: double, single
```

Class Support

Input T support floating-point values of type `single` or `double`.

References

[1] Golub, G. H. and C. F. Van Loan *Matrix Computations* Johns Hopkins University Press, Baltimore: 1989, pp. 557-558.

See Also

`pade`

Purpose

Page setup dialog box

Syntax

```
dlg = pagesetupdlg(fig)
```

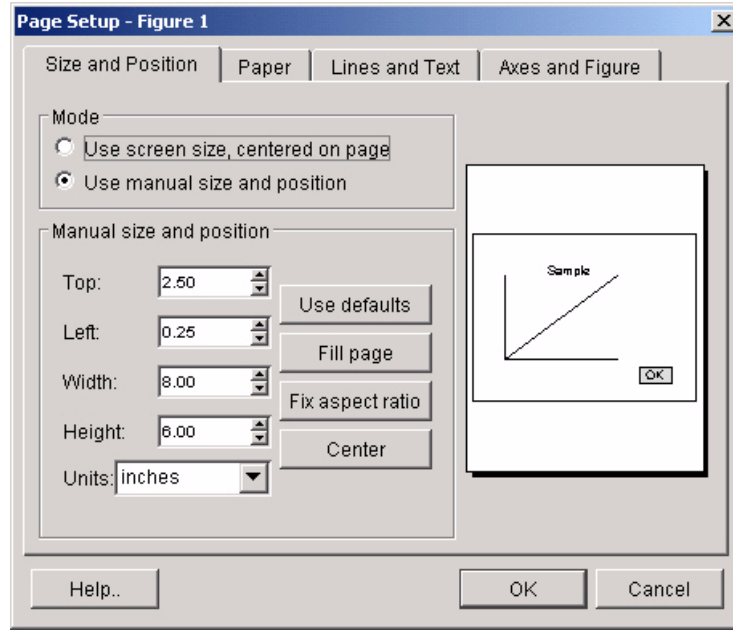
Note This function is obsolete. Use `printpreview` instead.

Description

`dlg = pagesetupdlg(fig)` creates a dialog box from which a set of pagelayout properties for the figure window, `fig`, can be set.

`pagesetupdlg` implements the "Page Setup..." option in the **Figure File Menu**.

`pagesetupdlg` supports setting the layout for a single figure. `fig` must be a single figure handle, not a vector of figures or a simulink diagram.



See Also printdlg, printpreview, printopt

Purpose

Pan view of graph interactively

GUI Alternatives

Use the **Pan** tool  on the figure toolbar to enable and disable pan mode on a plot, or select **Pan** from the figure's **Tools** menu. For details, see "Panning — Shifting Your View of the Graph" in the MATLAB Graphics documentation.

Syntax

```
pan on
pan xon
pan yon
pan off
pan
pan(figure_handle,...)
h = pan(figure_handle)
```

Description

`pan on` turns on mouse-based panning in the current figure.

`pan xon` turns on panning only in the *x* direction in the current figure.

`pan yon` turns on panning only in the *y* direction in the current figure.

`pan off` turns panning off in the current figure.

`pan` toggles the pan state in the current figure `on` or `off`.

`pan(figure_handle,...)` sets the pan state in the specified figure.

`h = pan(figure_handle)` returns the figure's pan *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Pan Mode Objects

Access the following properties of pan mode objects via `get` and modify some of them using `set`:

`Enable` `'on'|'off'`

Specifies whether this figure mode is currently enabled on the figure.

`Motion` `'horizontal'|'vertical'|'both'`

The type of panning enabled for the figure.

`FigureHandle <handle>`

The associated figure handle. This read-only property cannot be set.

`ButtonDownFilter <function_handle>`

The application can inhibit the panning operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function [res] = myfunction(obj,event_obj)
% obj        handle to the object that has been clicked on.
% event_obj  handle to event object (empty in this release).
% res        a logical flag to determine whether the pan
%            operation should take place or the 'ButtonDownFcn'
%            property of the object should take precedence.
```

`ActionPreCallback <function_handle>`

Set this callback to listen to when a pan operation will start. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on.
% event_obj   handle to event object.
```

The event object has the following read-only property:

`Axes`	The handle of the axes that is being panned

`ActionPostCallback <function_handle>`

Set this callback to listen to when a pan operation has finished. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function myfunction(obj,event_obj)
```

```
% obj         handle to the figure that has been clicked on.
% event_obj   handle to event object. The object has the same
%             properties as the event_obj of the
%             'ActionPreCallback' callback.
```

```
flags = isAllowAxesPan(h,axes)
```

Calling the function `isAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a pan operation is permitted on the axes objects.

```
setAllowAxesPan(h,axes,flag)
```

Calling the function `setAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a pan operation on the axes objects.

```
info = getAxesPanMotion(h,axes)
```

Calling the function `getAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, as input will return a character cell array of the same dimension as the axes handle vector, which indicates the type of pan operation for each axes. Possible values for the type of operation are `'horizontal'`, `'vertical'` or `'both'`.

```
setAxesPanMotion(h,axes,style)
```

Calling the function `setAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, sets the style of panning on each axes.

Examples

Example 1 — Entering Pan Mode

Plot a graph and turn on Pan mode:

```
plot(magic(10));
pan on
% pan on the plot
```

Example 2 — Constrained Pan

Constrain pan to *x*-axis using `set`:

```
plot(magic(10));
h = pan;
set(h,'Motion','horizontal','Enable','on');
% pan on the plot in the horizontal direction.
```

Example 3 — Constrained Pan in Subplots

Create four axes as subplots and give each one a different panning behavior:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = pan;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesPan(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesPanMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesPanMotion(h,ax4,'vertical');
% pan on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for pan mode objects to trigger. Copy the following code to a new M-file, execute it, and observe panning behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
```

```
h = pan;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-ButtonDown events for pan mode objects to trigger. Copy the following code to a new M-file, execute it, and observe panning behavior:

```
function demo
% Listen to pan events
plot(1:10);
h = pan;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A pan is about to occur.');
%
function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%.2f %.2f].',newLim));
```

Example 6 — Creating a Context Menu for Pan Mode

Coding a context menu that lets the user to switch to Zoom mode by right-clicking:

```
figure; plot(magic(10));
hCM = uicontextmenu;
hMenu = uimenu('Parent',hCM,'Label','Switch to zoom',...
        'Callback','zoom(gcbf,''on'')');
hPan = pan(gcf);
set(hPan,'UIContextMenu',hCM);
pan('on')
```

You cannot add items to the built-in pan context menu, but you can replace it with your own.

Remarks

You can create a pan mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

When you assign different pan behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

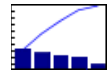
See Also

`zoom`, `linkaxes`, `rotate3d`

"Object Manipulation" on page 1-100 for related functions

Purpose

Pareto chart



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
pareto(Y)
pareto(Y,names)
pareto(Y,X)
H = pareto(...)
```

Description

Pareto charts display the values in the vector `Y` as bars drawn in descending order. Values in `Y` must be nonnegative and not include `NaN`s. Only the first 95% of the cumulative distribution is displayed.

`pareto(Y)` labels each bar with its element index in `Y` and also plots a line displaying the cumulative sum of `Y`.

`pareto(Y,names)` labels each bar with the associated name in the string matrix or cell array `names`.

`pareto(Y,X)` labels each bar with the associated value from `X`.

`pareto(ax,..)` plots a Pareto chart in existing axes `ax` rather than `GCA`.

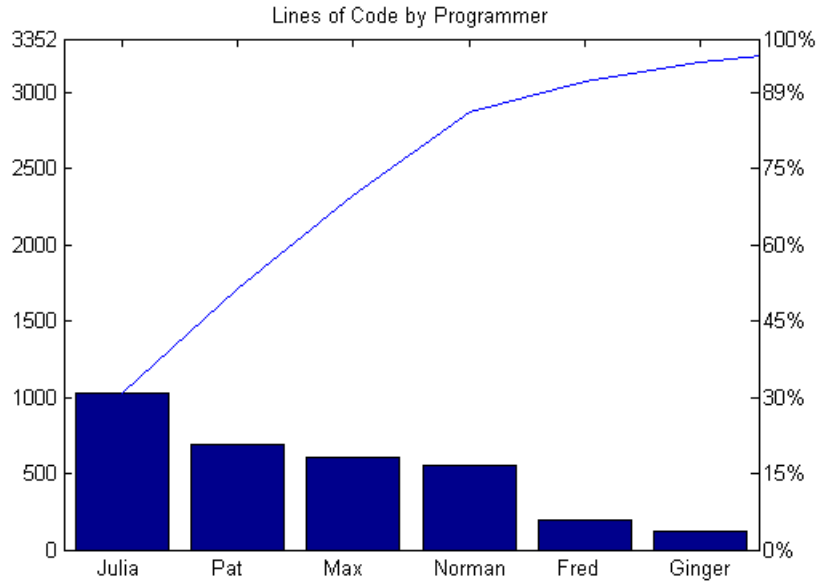
`H = pareto(...)` returns a combination of `patch` and `line` object handles.

Examples

Example 1:

Examine the cumulative productivity of a group of programmers to see how normal its distribution is:

```
codelines = [200 120 555 608 1024 101 57 687];
coders =
{'Fred','Ginger','Norman','Max','Julia','Wally','Heidi','Pat'};
pareto(codelines, coders)
title('Lines of Code by Programmer')
```



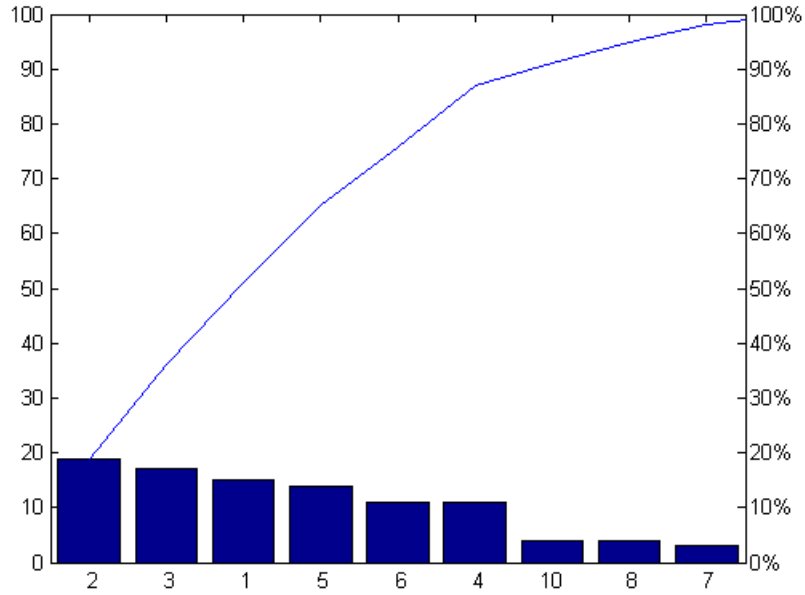
Example 2:

Generate a vector, `X`, representing diagnostic codes with values from 1 to 10 indicating various faults on devices emerging from a production line:

```
X = min(round(abs(randn(100,1)*4))+1,10);
```

Plot a Pareto chart showing the frequency of failure for each diagnostic code from the most to the least common:

```
pareto(hist(X))
```



Remarks

You can use `pareto` to display the output of `hist`, even for vectors that include negative numbers. Because only the first 95 percent of values are displayed, one or more of the smallest bars may not appear. If you extend the `Xlim` of your chart, you can display all the values, but the new bars will not be labeled.

See Also

`hist`, `bar`

parse (inputParser)

Purpose

Parse and validate named inputs

Syntax

```
p.parse(arglist)
parse(p, arglist)
```

Description

`p.parse(arglist)` parses and validates the inputs named in `arglist`.

`parse(p, arglist)`is functionally the same as the syntax above.

Note For more information on the `inputParser` class, see Parsing Inputs with inputParser in the MATLAB Programming documentation.

Examples

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class. Construct an instance of `inputParser` and assign it to variable `p`:

```
function publish_ip(script, varargin)
p = inputParser;   % Create an instance of the inputParser class.
```

Add arguments to the schema. See the reference pages for the `addRequired`, `addOptional`, and `addParamValue` methods for help with this:

```
p.addRequired('script', @ischar);
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Call the `parse` method of the object to read and validate each argument in the schema:

```
p.parse(script, varargin{:});
```

Execution of the `parse` method validates each argument and also builds a structure from the input arguments. The name of the structure is `Results`, which is accessible as a property of the object. To get the value of any input argument, type

```
p.Results.argname
```

Continuing with the `publish_ip` exercise, add the following lines to your M-file:

```
% Parse and validate all input arguments.
p.parse(script, varargin{:});

% Display the value for maxHeight.
disp(sprintf('\nThe maximum height is %d.\n', p.Results.maxHeight))

% Display all arguments.
disp 'List of all arguments:'
disp(p.Results)
```

When you call the program, MATLAB assigns those values you pass in the argument list to the appropriate fields of the `Results` structure. Save the M-file and execute it at the MATLAB command prompt with this command:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', 'C:/matlab/test', ...
    'maxWidth', 500, 'maxHeight', 300);

The maximum height is 300.

List of all arguments:
             format: 'ppt'
          maxHeight: 300
           maxWidth: 500
          outputDir: 'C:/matlab/test'
             script: 'ipscript.m'
```

See Also

`inputParser`, `addRequired(inputParser)`, `addOptional(inputParser)`, `addParamValue(inputParser)`, `createCopy(inputParser)`

Purpose

Convert response string from SOAP server into MATLAB data types

Syntax

```
parseSoapResponse(response)
```

Description

`parseSoapResponse(response)` converts `response`, a string returned by a SOAP server, into a cell array of appropriate MATLAB data types.

Example

```
message = createSoapMessage(...
'urn:xmethods-delayed-quotes','getQuote',{'GOOG'},{'symbol'},...
{'{http://www.w3.org/2001/XMLSchema}string'},'rpc')
response = callSoapService('http://64.124.140.30:9090/soap',...
'urn:xmethods-delayed-quotes#getQuote',message)
price = parseSoapResponse(response)
```

See Also

`callSoapService`, `createClassFromWsdl`, `createSoapMessage`

Purpose

Partial pathname description

Description

A partial pathname is a pathname relative to the MATLAB path, `matlabpath`. It is used to locate private and method files, which are usually hidden, or to restrict the search for files when more than one file with the given name exists.

A partial pathname contains the last component, or last several components, of the full pathname separated by `/`. For example, `matfun/trace`, `private/children`, and `demos/clown.mat` are valid partial pathnames. Specifying the `@` in method directory names is optional.

Partial pathnames make it easy to find a toolbox or MATLAB relative files on your path, independent of the location where MATLAB is installed.

Many commands accept partial pathnames instead of a full pathname. Some of these commands are

```
help, type, load, exist, what, which, edit, dbtype,
dbstop, dbclear, fopen
```

Examples

The following example uses a partial pathname:

```
what graph2d/@figobj

M-files in directory
matlabroot\toolbox\matlab\graph2d\@figobj

deselectall    enddrag        middrag        subsref
doclick        figobj         set
doresize       get            subsasgn


P-files in directory
matlabroot\toolbox\matlab\graph2d\@figobj

deselectall    enddrag        middrag        subsref
```

```
doclick        figobj         set
doresize       get            subsasgn
```

The `@` in the class directory name `@figobj` is not necessary. You get the same response from the following command:

```
what graph2d/figobj
```

See Also

`fileparts`, `matlabroot`, `path`

Purpose

Pascal matrix

Syntax

```
A = pascal(n)
A = pascal(n,1)
A = pascal(n,2)
```

Description

`A = pascal(n)` returns the Pascal matrix of order `n`: a symmetric positive definite matrix with integer entries taken from Pascal's triangle. The inverse of `A` has integer entries.

`A = pascal(n,1)` returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. It is *involutary*, that is, it is its own inverse.

`A = pascal(n,2)` returns a transposed and permuted version of `pascal(n,1)`. `A` is a cube root of the identity matrix.

Examples

`pascal(4)` returns

```
   1     1     1     1
   1     2     3     4
   1     3     6    10
   1     4    10    20
```

`A = pascal(3,2)` produces

```
   A =
         1     1     1
        -2    -1     0
         1     0     0
```

See Also

`chol`

Purpose

Create patch graphics object

Syntax

```
patch(X,Y,C)
patch(X,Y,Z,C)
patch(FV)
patch(...'PropertyName',propertyvalue...)
patch('PropertyName',propertyvalue,...)
handle = patch(...)
```

Description

`patch` is the low-level graphics function for creating patch graphics objects. A patch object is one or more polygons defined by the coordinates of its vertices. You can specify the coloring and lighting of the patch. See "Creating 3-D Models with Patches" for more information on using patch objects.

`patch(X,Y,C)` adds the filled two-dimensional patch to the current axes. The elements of `X` and `Y` specify the vertices of a polygon. If `X` and `Y` are matrices, MATLAB draws one polygon per column. `C` determines the color of the patch. It can be a single `ColorSpec`, one color per face, or one color per vertex (see "Remarks" on page 2-2396). If C is a 1-by-3 vector, it is assumed to be an RGB triplet, specifying a color directly.

`patch(X,Y,Z,C)` creates a patch in three-dimensional coordinates.

`patch(FV)` creates a patch using structure `FV`, which contains the fields `vertices`, `faces`, and optionally `facevertexcdata`. These fields correspond to the `Vertices`, `Faces`, and `FaceVertexCData` patch properties.

`patch(...'PropertyName',propertyvalue...)` follows the `X`, `Y`, `(Z)`, and `C` arguments with property name/property value pairs to specify additional patch properties.

`patch('PropertyName',propertyvalue,...)` specifies all properties using property name/property value pairs. This form enables you to omit the color specification because MATLAB uses the default face color and edge color unless you explicitly assign a value to the `FaceColor` and `EdgeColor` properties. This form also allows you to specify the patch using the `Faces` and `Vertices` properties instead of *x*-, *y*-, and

z-coordinates. See the "Examples" on page 2-2399 section for more information.

`handle = patch(...)` returns the handle of the patch object it creates.

Remarks

Unlike high-level area creation functions, such as `fill` or `area`, `patch` does not check the settings of the figure and axes `NextPlot` properties. It simply adds the patch object to the current axes.

If the coordinate data does not define closed polygons, `patch` closes the polygons. The data can define concave or intersecting polygons. However, if the edges of an individual patch face intersect themselves, the resulting face may or may not be completely filled. In that case, it is better to break up the face into smaller polygons.

Specifying Patch Properties

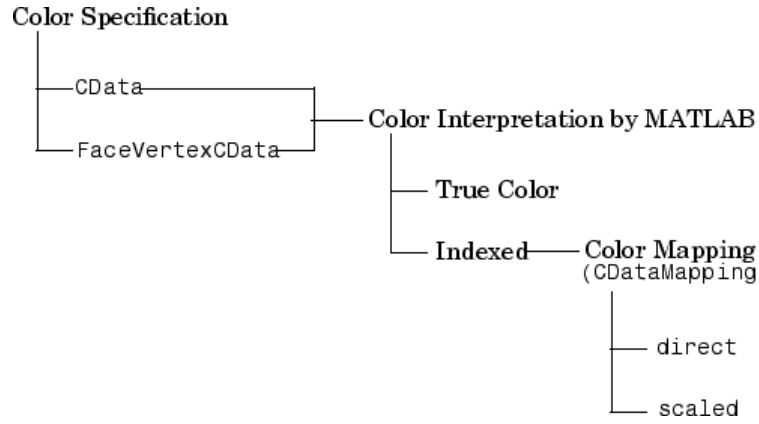
You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

There are two patch properties that specify color:

- `CData` — Use when specifying *x*-, *y*-, and *z*-coordinates (`XData`, `YData`, `ZData`).
- `FaceVertexCData` — Use when specifying vertices and connection matrix (`Vertices` and `Faces`).

The `CData` and `FaceVertexCData` properties accept color data as indexed or true color (RGB) values. See the `CData` and `FaceVertexCData` property descriptions for information on how to specify color.

Indexed color data can represent either direct indices into the colormap or scaled values that map the data linearly to the entire colormap (see the `caxis` function for more information on this scaling). The `CDataMapping` property determines how MATLAB interprets indexed color data.



Color Data Interpretation

You can specify patch colors as

- A single color for all faces
- One color for each face, enabling flat coloring
- One color for each vertex, enabling interpolated coloring

The following tables summarize how MATLAB interprets color data defined by the `CData` and `FaceVertexCData` properties.

Interpretation of the CData Property

[X,Y,Z]Data	CData Required for		Results Obtained
Dimensions	Indexed	True Color	
m-by-n	scalar	1-by-1-by-3	Use the single color specified for all patch faces. Edges can be only a single color.
m-by-n	1-by-n (n >= 4)	1-by-n-by-3	Use one color for each patch face. Edges can be only a single color.
m-by-n	m-by-n	m-by-n-3	Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated.

Interpretation of the FaceVertexCData Property

Vertices	Faces	FaceVertexCData Required for		Results Obtained
Dimensions	Dimensions	Indexed	True Color	
m-by-n	k-by-3	scalar	1-by-3	Use the single color specified for all patch faces. Edges can be only a single color.
m-by-n	k-by-3	k-by-1	k-by-3	Use one color for each patch face. Edges can be only a single color.
m-by-n	k-by-3	m-by-1	m-by-3	Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated.

Examples

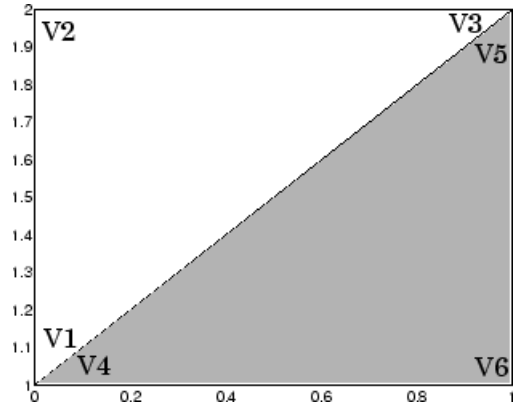
This example creates a patch object using two different methods:

- Specifying *x*-, *y*-, and *z*-coordinates and color data (`XData`, `YData`, `ZData`, and `CData` properties)
- Specifying vertices, the connection matrix, and color data (`Vertices`, `Faces`, `FaceVertexCData`, and `FaceColor` properties)

Specifying X, Y, and Z Coordinates

The first approach specifies the coordinates of each vertex. In this example, the coordinate data defines two triangular faces, each having three vertices. Using true color, the top face is set to white and the bottom face to gray.

```
x = [0 0;0 1;1 1];
y = [1 1;2 2;2 1];
z = [1 1;1 1;1 1];
tcolor(1,1,1:3) = [1 1 1];
tcolor(1,2,1:3) = [.7 .7 .7];
patch(x,y,z,tcolor)
```



Notice that each face shares two vertices with the other face (V_1-V_4 and V_3-V_5).

Specifying Vertices and Faces

The `Vertices` property contains the coordinates of each *unique* vertex defining the patch. The `Faces` property specifies how to connect these vertices to form each face of the patch. For this example, two vertices share the same location so you need to specify only four of the six vertices. Each row contains the *x*-, *y*-, and *z*-coordinates of each vertex.

```
vert = [0 1 1;0 2 1;1 2 1;1 1 1];
```

There are only two faces, defined by connecting the vertices in the order indicated.

```
fac = [1 2 3;1 3 4];
```

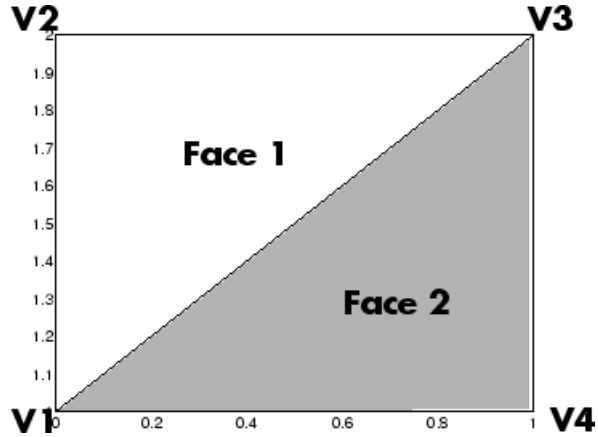
To specify the face colors, define a 2-by-3 matrix containing two RGB color definitions.

```
tcolor = [1 1 1;.7 .7 .7];
```

With two faces and two colors, MATLAB can color each face with flat shading. This means you must set the `FaceColor` property to `flat`, since the faces/vertices technique is available only as a low-level function call (i.e., only by specifying property name/property value pairs).

Create the patch by specifying the `Faces`, `Vertices`, and `FaceVertexCData` properties as well as the `FaceColor` property.

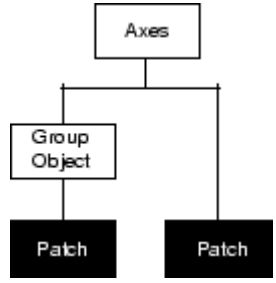
```
patch('Faces',fac,'Vertices',vert,'FaceVertexCData',tcolor,...
      'FaceColor','flat')
```



Specifying only unique vertices and their connection matrix can reduce the size of the data for patches having many faces. See the descriptions of the `Faces`, `Vertices`, and `FaceVertexCData` properties for information on how to define them.

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the `Faces` matrix with `NaNs`. To define a patch with faces that do not close, add one or more `NaNs` to the row in the `Vertices` matrix that defines the vertex you do not want connected.

Object Hierarchy



Setting Default Properties

You can set default patch properties on the axes, figure, and root levels:

```
set(0,'DefaultPatchPropertyName',PropertyValue...)
set(gcf,'DefaultPatchPropertyName',PropertyValue...)
set(gca,'DefaultPatchPropertyName',PropertyValue...)
```

PropertyName is the name of the patch property and PropertyValue is the value you are specifying. Use set and get to access patch properties.

See Also

area, caxis, fill, fill3, isosurface, surface

"Object Creation Functions" on page 1-94 for related functions

Patch Properties for property descriptions

"Creating 3-D Models with Patches" for examples that use patches

Purpose

Patch properties

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Patch Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`AlphaDataMapping`

```
none| {scaled} | direct
```

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. This property can be any of the following:

- `none` — The transparency values of `FaceVertexAlphaData` are between 0 and 1 or are clamped to this range.
- `scaled` — Transform the `FaceVertexAlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values. (`scaled` is the default)
- `direct` — Use the `FaceVertexAlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the

last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If `FaceVertexAlphaData` is an array of `uint8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength`

scalar >= 0 and <= 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the `DiffuseStrength` and `SpecularStrength` properties.

`Annotation`

`hg.Annotation` object Read Only

Control the display of patch objects in legends. The `Annotation` property enables you to specify whether this patch object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the patch object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this patch object in a legend (default)
`off`	Do not include this patch object in a legend
`children`	Same as `on` because patch objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

Selecting which objects to display in legend

Some graphics functions create multiple objects. For example, `contour3` uses patch objects to create a 3D contour graph. You can use the `Annotation` property set select a subset of the objects for display in the legend.

```
[X,Y] = meshgrid(-2:.1:2);
[Cm hC] = contour3(X.*exp(-X.^2-Y.^2));
hA = get(hC,'Annotation');
hLL = get([hA{:}],'LegendInformation');
% Set the IconDisplayStyle property to display
% the first, fifth, and ninth patch in the legend
set([hLL{:}],{'IconDisplayStyle'},...
   {'on','off','off','off','on','off','off','off','on'}')
```

```
% Assign DisplayNames for the three patch
that are displayed in the legend
set(hC([1,5,9]),{'DisplayName'},{'bottom','middle','top'}')
legend show
```

`BackFaceLighting`
`unlit | lit | {reverselit}`

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera:

- `unlit` — Face is not lit.
- `lit` — Face is lit in normal way.
- `reverselit` — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See the Using MATLAB Graphics manual for an example.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the patch object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. You can also use a string that is a valid MATLAB expression or the name of an M-file. The expressions execute in the MATLAB workspace.

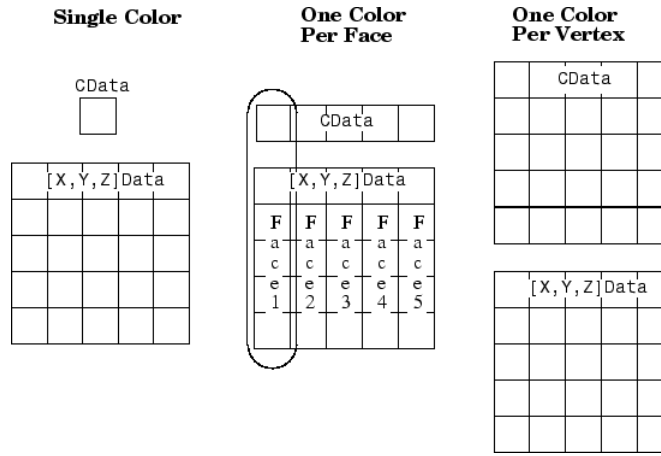
See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`CData`

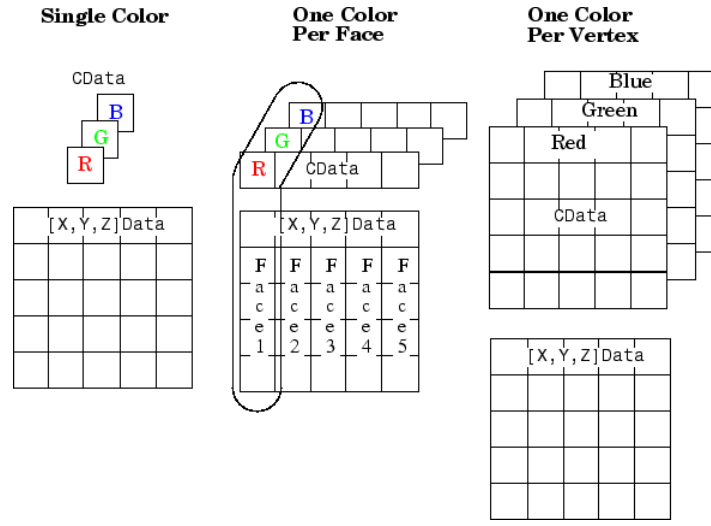
scalar, vector, or matrix

Patch colors. This property specifies the color of the patch. You can specify color for each vertex, each face, or a single color for the entire patch. The way MATLAB interprets `CData` depends on the type of data supplied. The data can be numeric values that are scaled to map linearly into the current colormap, integer values that are used directly as indices into the current colormap, or arrays of RGB values. RGB values are not mapped into the current colormap, but interpreted as the colors defined. On true color systems, MATLAB uses the actual colors defined by the RGB triples.

The following two diagrams illustrate the dimensions of `CData` with respect to the coordinate data arrays, `XData`, `YData`, and `ZData`. The first diagram illustrates the use of indexed color.



The second diagram illustrates the use of true color. True color requires *m*-by-*n*-by-3 arrays to define red, green, and blue components for each color.



Note that if `CData` contains `NaN`s, MATLAB does not color the faces.

See also the `Faces`, `Vertices`, and `FaceVertexCData` properties for an alternative method of patch definition.

`CDataMapping`

```
{scaled} | direct
```

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the patch. (If you use true color specification for `CData` or `FaceVertexCData`, this property has no effect.)

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` command for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. When not scaled, the data are usually integer values

ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`Children`
matrix of handles

Always the empty matrix; patch objects have no children.

`Clipping`
`{on} | off`

Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the patch outside the axes rectangle.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a patch object. You must define this property as a default value for patches or in a call to the `patch` function that creates a new object.

For example, the following statement creates a patch (assuming `x`, `y`, `z`, and `c` are defined), and executes the function referenced by the function handle `@myCreateFcn`.

```
patch(x,y,z,c,'CreateFcn',@myCreateFcn)
```

MATLAB executes the create function after setting all properties for the patch created. Setting this property on an existing patch object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Delete patch callback routine. A callback routine that executes when you delete the patch object (e.g., when you issue a `delete` command or clear the axes (`cla`) or figure (`clf`) containing the patch). MATLAB executes the routine before deleting the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DiffuseStrength`
scalar >= 0 and <= 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the patch. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the patch object. See the `AmbientStrength` and `SpecularStrength` properties.

`DisplayName`
string (default is empty string)

String used by legend for this patch object. The `legend` function uses the string defined by the `DisplayName` property to label this patch object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this patch object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EdgeAlpha`

`{scalar = 1} | flat | interp`

Transparency of the edges of patch faces. This property can be any of the following:

- scalar — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.
- `flat` — The alpha data (`FaceVertexAlphaData`) of each vertex controls the transparency of the edge that follows it.
- `interp` — Linear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determines the transparency of the edge.

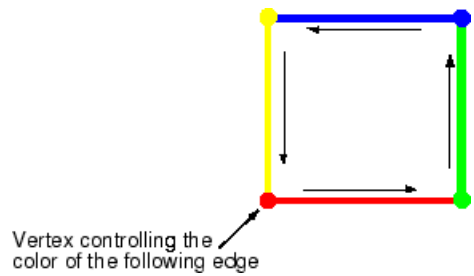
Note that you cannot specify `flat` or `interp` `EdgeAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

`EdgeColor`

`{ColorSpec} | none | flat | interp`

Color of the patch edge. This property determines how MATLAB colors the edges of the individual faces that make up the patch.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default edge color is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The color of each vertex controls the color of the edge that follows it. This means `flat` edge coloring is dependent on the order in which you specify the vertices:



- `interp` — Linear interpolation of the `CData` or `FaceVertexCData` values at the vertices determines the edge color.

`EdgeLighting`

`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on patch edges. Choices are

- `none` — Lights do not affect the edges of this object.
- `flat` — The effect of light objects is uniform across each edge of the patch.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase patch objects. Alternative erase modes are useful in creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the patch when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor`— Draw and erase the patch by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the patch does not damage the color of the objects behind it. However, patch color depends on the color of the screen behind it and is correctly colored only when over the axes background

Color, or the figure background Color if the axes Color is set to none.

- background — Erase the patch by drawing it in the axes background Color, or the figure background Color if the axes Color is set to none. This damages objects that are behind the erased patch, but the patch is always properly colored.

  Printing with Nonnormal Erase Modes

  MATLAB always prints figures as if the EraseMode of all objects is normal. This means graphics objects created with EraseMode set to none, xor, or background can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., perform an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

  You can use the MATLAB getframe command or other screen capture application to create an image of a figure containing nonnormal mode objects.

FaceAlpha

```
{scalar = 1} | flat | interp
```

Transparency of the patch face. This property can be any of the following:

- A scalar — A single non-NaN value between 0 and 1 that controls the transparency of all the faces of the object. 1 (the default) means fully opaque and 0 means completely transparent (invisible).
- flat — The values of the alpha data (FaceVertexAlphaData) determine the transparency for each face. The alpha data at the first vertex determines the transparency of the entire face.
- interp — Bilinear interpolation of the alpha data (FaceVertexAlphaData) at each vertex determines the transparency of each face.

Note that you cannot specify `flat` or `interp` `FaceAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

`FaceColor`

`{ColorSpec} | none | flat | interp`

Color of the patch face. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The `CData` or `FaceVertexCData` property must contain one value per face and determines the color for each face in the patch. The color data at the first vertex determines the color of the entire face.
- `interp` — Bilinear interpolation of the color at each vertex determines the coloring of each face. The `CData` or `FaceVertexCData` property must contain one value per vertex.

`FaceLighting`

`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of `light` objects on patch faces. Choices are

- `none` — Lights do not affect the faces of this object.
- `flat` — The effect of light objects is uniform across the faces of the patch. Select this choice to view faceted objects.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.

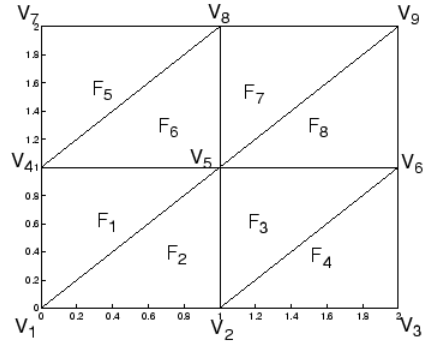
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`Faces`

m-by-n matrix

Vertex connection defining each face. This property is the connection matrix specifying which vertices in the `Vertices` property are connected. The `Faces` matrix defines *m* faces with up to *n* vertices each. Each row designates the connections for a single face, and the number of elements in that row that are not `NaN` defines the number of vertices for that face.

The `Faces` and `Vertices` properties provide an alternative way to specify a patch that can be more efficient than using *x*, *y*, and *z* coordinates in most cases. For example, consider the following patch. It is composed of eight triangular faces defined by nine vertices.



`Faces` property

F_1	V_1	V_4	V_5
F_2	V_1	V_5	V_2
F_3	V_2	V_5	V_6
F_4	V_2	V_6	V_3
F_5	V_4	V_7	V_8
F_6	V_4	V_8	V_5
F_7	V_5	V_8	V_9
F8	V_5	V_9	V_6

`Vertices` property

V_1	X_1	Y_1	Z_1
V_2	X_2	Y_2	Z_2
V_3	X_3	Y_3	Z_3
V_4	X_4	Y_4	Z_4
V_5	X_5	Y_5	Z_5
V_6	X_6	Y_6	Z_6
V_7	X_7	Y_7	Z_7
V_8	X_8	Y_8	Z_8
V_9	X_9	Y_9	Z_9

The corresponding `Faces` and `Vertices` properties are shown to the right of the patch. Note how some faces share vertices with

other faces. For example, the fifth vertex (V5) is used six times, once each by faces one, two, and three and six, seven, and eight. Without sharing vertices, this same patch requires 24 vertex definitions.

`FaceVertexAlphaData`
m-by-1 matrix

Face and vertex transparency data. The `FaceVertexAlphaData` property specifies the transparency of patches that have been defined by the `Faces` and `Vertices` properties. The interpretation of the values specified for `FaceVertexAlphaData` depends on the dimensions of the data.

`FaceVertexAlphaData` can be one of the following:

- A single value, which applies the same transparency to the entire patch. The `FaceAlpha` property must be set to `flat`.
- An m-by-1 matrix (where m is the number of rows in the `Faces` property), which specifies one transparency value per face. The `FaceAlpha` property must be set to `flat`.
- An m-by-1 matrix (where m is the number of rows in the `Vertices` property), which specifies one transparency value per vertex. The `FaceAlpha` property must be set to `interp`.

The `AlphaDataMapping` property determines how MATLAB interprets the `FaceVertexAlphaData` property values.

`FaceVertexCData`
matrix

Face and vertex colors. The `FaceVertexCData` property specifies the color of patches defined by the `Faces` and `Vertices` properties. You must also set the values of the `FaceColor`, `EdgeColor`, `MarkerFaceColor`, or `MarkerEdgeColor` appropriately. The interpretation of the values specified for `FaceVertexCData` depends on the dimensions of the data.

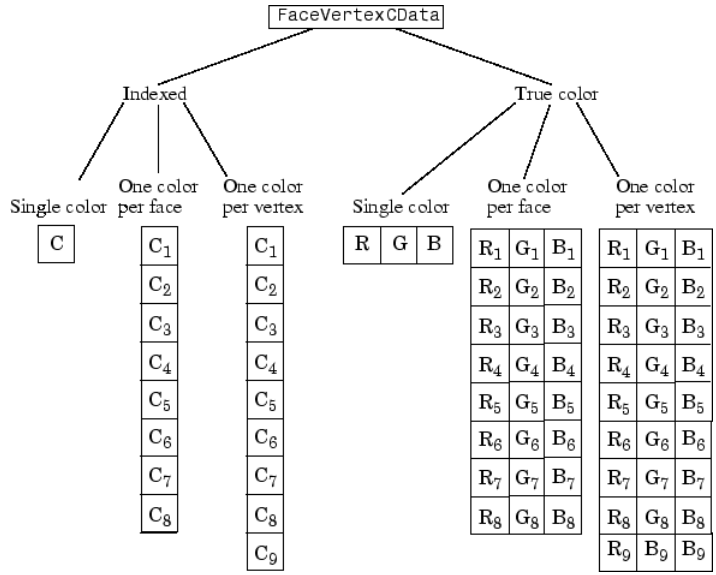
For indexed colors, `FaceVertexCData` can be

- A single value, which applies a single color to the entire patch
- An *n*-by-1 matrix, where *n* is the number of rows in the `Faces` property, which specifies one color per face
- An *n*-by-1 matrix, where *n* is the number of rows in the `Vertices` property, which specifies one color per vertex

For true colors, `FaceVertexCData` can be

- A 1-by-3 matrix, which applies a single color to the entire patch
- An *n*-by-3 matrix, where *n* is the number of rows in the `Faces` property, which specifies one color per face
- An *n*-by-3 matrix, where *n* is the number of rows in the `Vertices` property, which specifies one color per vertex

The following diagram illustrates the various forms of the `FaceVertexCData` property for a patch having eight faces and nine vertices. The `CDataMapping` property determines how MATLAB interprets the `FaceVertexCData` property when you specify indexed colors.



`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close.

When a handle's visibility is restricted using callback or off, the object's handle does not appear in its parent's Children property, figures do not appear in the root's CurrentFigure property, objects do not appear in the root's CallbackObject property or in the figure's CurrentObject property, and axes do not appear in their parent's CurrentAxes property.

You can set the root ShowHiddenHandles property to on to make all handles visible regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties).

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HitTest

```
{on} | off
```

Selectable by mouse click. HitTest determines if the patch can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click on the patch. If HitTest is off, clicking the patch selects the object below it (which may be the axes containing it).

Interruptible

```
{on} | off
```

Callback routine interruption mode. The `Interruptible` property controls whether a patch callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

`LineStyle`

`{-} | -- | : | -. | none`

Edge linestyle. This property specifies the line style of the patch edges. The following table lists the available line styles.

Symbol	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

Edge line width. The width, in points, of the patch edges (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`

character (see table)

Marker symbol. The `Marker` property specifies marks that locate vertices. You can set values for the `Marker` property independently from the `LineStyle` property. The following tables lists the available markers.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto} | flat`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — Defines the color to use.
- `none` — Specifies no color, which makes nonfilled markers invisible.

- `auto` — Sets `MarkerEdgeColor` to the same color as the `EdgeColor` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto | flat`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — Defines the color to use.
- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Sets the fill color to the axes color, or the figure color, if the axes `Color` property is set to `none`.

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is 6 points (1 point = $^1/_{72}$ inch). Note that MATLAB draws the `point` marker at 1/3 of the specified size.

`NormalMode`
`{auto} | manual`

MATLAB generated or user-specified normal vectors. When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to manual and does not generate its own data. See also the `VertexNormals` property.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of patch object. This property contains the handle of the patch object's parent. The parent of a patch object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When this property is `on`, MATLAB displays selection handles or a dashed box (depending on the number of faces) if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by

- Drawing handles at each vertex for a single-faced patch
- Drawing a dashed bounding box for a multifaced patch

When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`SpecularColorReflectance`
scalar in the range 0 to 1

Color of specularly reflected light. When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color of the light source (i.e., the light object `Color` property). The proportions vary linearly for values in between.

`SpecularExponent`
scalar >= 1

Harshness of specular reflection. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

`SpecularStrength`
scalar >= 0 and <= 1

Intensity of specular light. This property sets the intensity of the specular component of the light falling on the patch. Specular light comes from `light` objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the patch object. See the `AmbientStrength` and `DiffuseStrength` properties.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines.

For example, suppose you use patch objects to create borders for a group of `uicontrol` objects and want to change the color of the borders in a uicontrol's callback routine. You can specify a `Tag` with the patch definition

```
patch(X,Y,'k','Tag','PatchBorder')
```

Then use `findobj` in the uicontrol's callback routine to obtain the handle of the patch and set its `FaceColor` property.

```
set(findobj('Tag','PatchBorder'),'FaceColor','w')
```

`Type`
string (read only)

Class of the graphics object. For patch objects, `Type` is always the string `'patch'`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the patch. Assign this property the handle of a uicontextmenu object created in the same figure as the patch. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the patch.

`UserData`
matrix

User-specified data. Any matrix you want to associate with the patch object. MATLAB does not use this data, but you can access it using `set` and `get`.

`VertexNormals`
matrix

Surface normal vectors. This property contains the vertex normals for the patch. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

`Vertices`
matrix

Vertex coordinates. A matrix containing the *x*-, *y*-, *z*-coordinates for each vertex. See the `Faces` property for more information.

`Visible`
`{on} | off`

Patch object visibility. By default, all patches are visible. When set to `off`, the patch is not visible, but still exists, and you can query and set its properties.

`XData`
vector or matrix

X-coordinates. The *x*-coordinates of the patch vertices. If `XData` is a matrix, each column represents the *x*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

`YData`
vector or matrix

Y-coordinates. The *y*-coordinates of the patch vertices. If `YData` is a matrix, each column represents the *y*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

`ZData`
vector or matrix

Z-coordinates. The *z*-coordinates of the patch vertices. If `ZData` is a matrix, each column represents the *z*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

See Also

`patch`

Purpose

View or change MATLAB directory search path

GUI Alternatives

As an alternative to the `path` function, select **File > Set Path** to use the Set Path dialog box.

Syntax

```
path
path('newpath')
path(path,'newpath')
path('newpath',path)
p = path(...)
```

Description

`path` displays the current MATLAB search path. The initial search path list is defined by `toolbox/local/pathdef.m`.

`path('newpath')` changes the search path to `newpath`, where `newpath` is a string array of directories.

`path(path,'newpath')` adds the `newpath` directory to the bottom of the current search path. If `newpath` is already on the path, then `path(path, 'newpath')` moves `newpath` to the end of the path.

`path('newpath',path)` adds the `newpath` directory to the top of the current search path. If `newpath` is already on the path, then `path('newpath', path)` moves `newpath` to the beginning of the path.

`p = path(...)` returns the specified path in string variable `p`.

Note Save any M-files you create and any MathWorks supplied M-files that you edit in a directory that is not in the *matlabroot*/toolbox directory tree. If you keep your files in *matlabroot*/toolbox directories, they can be overwritten when you install a new version of MATLAB. Also note that locations of files in the *matlabroot*/toolbox directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to *matlabroot*/toolbox directories using an external editor or add or remove files from these directories using file system operations, run `rehash toolbox` before you use the files in the current session. If you make changes to existing files in *matlabroot*/toolbox directories using an external editor, run `clear functionname` before you use the files in the current session. For more information, see the `rehash` reference page or the Toolbox Path Caching topic in the MATLAB Desktop Tools and Development Environment documentation.

Examples

Add a new directory to the search path on Windows.

```
path(path,'c:/tools/goodstuff')
```

Add a new directory to the search path on UNIX.

```
path(path,'/home/tools/goodstuff')
```

See Also

`addpath`, `cd`, `dir`, `genpath`, `matlabroot`, `partialpath`, `pathdef`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`, `startup`, `what`

Search Path in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Save current MATLAB search path to `pathdef.m` file

Syntax

```
path2rc
```

Description

`path2rc` runs `savepath`. The `savepath` function is replacing `path2rc`. Use `savepath` instead of `path2rc` and replace instances of `path2rc` with `savepath`.

pathdef

Purpose

Directories in MATLAB search path

GUI Alternatives

As an alternative to the `pathdef` function, select **File > Set Path** to use the Set Path dialog box.

Syntax

```
pathdef
```

Description

`pathdef` returns a string listing of the directories in the MATLAB search path. Use `path` to view each directory in `pathdef.m` on a separate line.

When you start a new session, MATLAB creates the search path defined in the `pathdef.m` file located in the MATLAB startup directory. If that directory does not contain a `pathdef.m` file, MATLAB uses the search path defined in *`matlabroot`*`/toolbox/local/pathdef.m`. It modifies the search path using any path statements contained in the `startup.m` file.

Make changes to the path using the **Set Path** dialog box and `addpath` and `rmpath`. While you can edit `pathdef.m` directly, use caution so you do not accidentally make MATLAB supplied directories unusable. Use `savepath` to save `pathdef.m`, and to use that path in future sessions, specify the MATLAB startup directory as its location.

See Also

`addpath`, `cd`, `dir`, `genpath`, `matlabroot`, `partialpath`, `path`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`, `startup`, `what`

MATLAB Desktop Tools and Development Environment documentation topics

- How MATLAB Finds the Search Path, pathdef.m
- Saving Settings to the Path
- Using the Path in Future Sessions
- Recovering from Problems with the Search Path

Purpose

Path separator for current platform

Syntax

```
c = pathsep
```

Description

`c = pathsep` returns the path separator character for this platform. The path separator is the character that separates directories in the string returned by the `matlabpath` function.

Examples

Extract each individual path from the string returned by `matlabpath`. Use `pathsep` to define the path separator:

```
s = matlabpath;
p = 1;

while true
   t = strtok(s(p:end), pathsep);
   disp(sprintf('%s', t))
   p = p + length(t) + 1;
   if isempty(strfind(s(p:end),';')) break, end;
end

disp(sprintf('%s', s(p:end)))
```

Here is the output:

```
D:\Applications\matlabR14beta2\toolbox\matlab\general
D:\Applications\matlabR14beta2\toolbox\matlab\ops
D:\Applications\matlabR14beta2\toolbox\matlab\lang
D:\Applications\matlabR14beta2\toolbox\matlab\elmat
D:\Applications\matlabR14beta2\toolbox\matlab\elfun
       .
       .
       .
```

See Also

`filesep`, `fullfile`, `fileparts`

Purpose

Open Set Path dialog box to view and change MATLAB path

GUI Alternatives

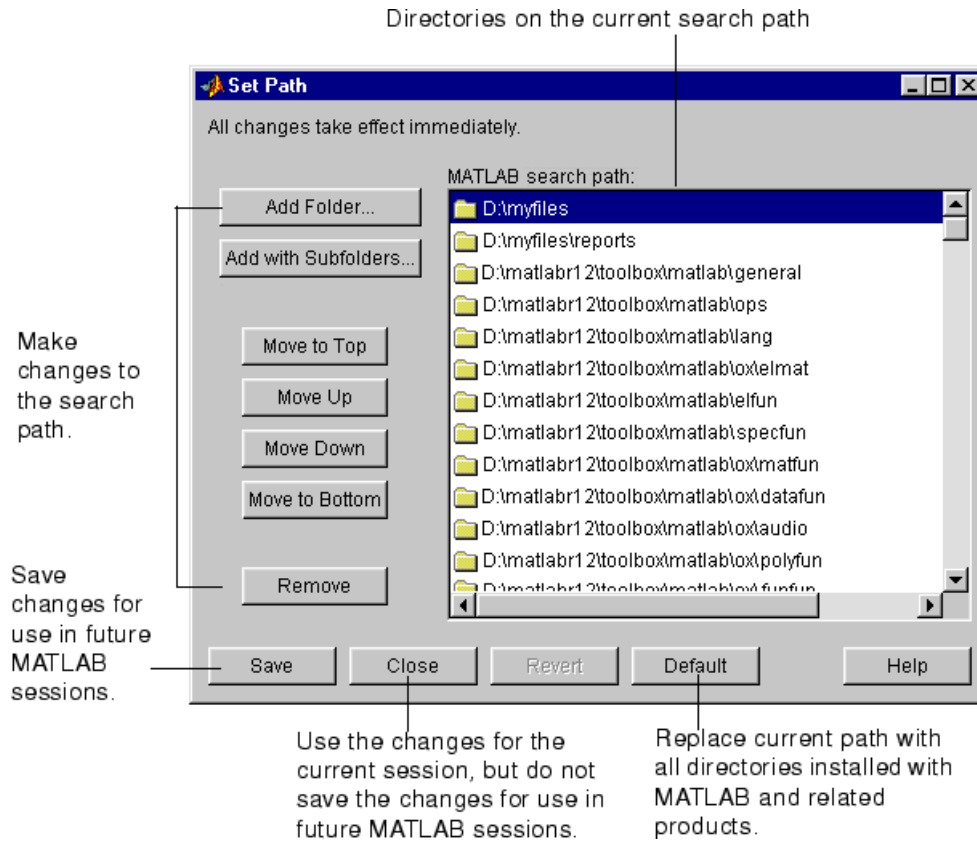
As an alternative to the `pathtool` function, select **File > Set Path** in the MATLAB desktop.

Syntax

```
pathtool
```

Description

`pathtool` opens the **Set Path** dialog box, a graphical user interface you use to view and modify the MATLAB search path.



See Also

`addpath`, `cd`, `dir`, `genpath`, `matlabroot`, `partialpath`, `path`, `pathdef`, `pathsep`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`, `startup`, `what`

Search Path topics, including Setting the Search Path, in the MATLAB Desktop Tools and Development Environment documentation

pause

Purpose

Halt execution temporarily

Syntax

```
pause
pause(n)
pause on
pause off
```

Description

`pause`, by itself, causes M-files to stop and wait for you to press any key before continuing.

`pause(n)` pauses execution for `n` seconds before continuing, where `n` can be any nonnegative real number. The resolution of the clock is platform specific. A fractional pause of 0.01 seconds should be supported on most platforms.

Typing `pause(inf)` puts you into an infinite loop. To return to the MATLAB prompt, type **Ctrl+C**.

`pause on` allows subsequent `pause` commands to pause execution.

`pause off` ensures that any subsequent `pause` or `pause(n)` statements do not pause execution. This allows normally interactive scripts to run unattended.

Remarks

While MATLAB is paused, the following continue to execute:

- Repainting of figure windows, block diagrams, and Java windows
- HG callbacks from figure windows
- Event handling from Java windows

See Also

`drawnow`

Purpose

Set or query plot box aspect ratio

Syntax

```
pbaspect
pbaspect([aspect_ratio])
pbaspect('mode')
pbaspect('auto')
pbaspect('manual')
pbaspect(axes_handle,...)
```

Description

The plot box aspect ratio determines the relative size of the *x*-, *y*-, and *z*-axes.

`pbaspect` with no arguments returns the plot box aspect ratio of the current axes.

`pbaspect([aspect_ratio])` sets the plot box aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the *x*-, *y*-, and *z*-axes size. For example, a value of `[1 1 1]` (the default) means the plot box is a cube (although with stretch-to-fill enabled, it may not appear as a cube). See Remarks.

`pbaspect('mode')` returns the current value of the plot box aspect ratio mode, which can be either `auto` (the default) or `manual`. See Remarks.

`pbaspect('auto')` sets the plot box aspect ratio mode to `auto`.

`pbaspect('manual')` sets the plot box aspect ratio mode to `manual`.

`pbaspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. If you do not specify an axes handle, `pbaspect` operates on the current axes.

Remarks

`pbaspect` sets or queries values of the axes object `PlotBoxAspectRatio` and `PlotBoxAspectRatioMode` properties.

When the plot box aspect ratio mode is `auto`, MATLAB sets the ratio to `[1 1 1]`, but may change it to accommodate manual settings of the data aspect ratio, camera view angle, or axis limits. See the axes `DataAspectRatio` property for a table listing the interactions between various properties.

Setting a value for the plot box aspect ratio or setting the plot box aspect ratio mode to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the plot box aspect ratio to its current value,

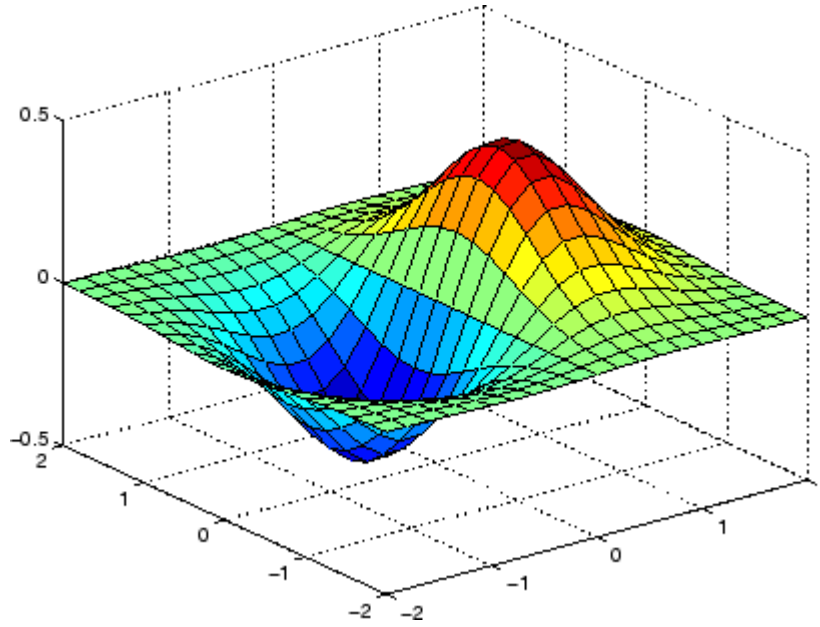
```
pbaspect(pbaspect)
```

can cause a change in the way the graphs look. See the Remarks section of the `axes` reference description, "Axes Aspect Ratio Properties" in the 3-D Visualization manual, and "Setting Aspect Ratio" in the MATLAB Graphics manual for a discussion of stretch-to-fill.

Examples

The following surface plot of the function $z = xe^{(-x^2 - y^2)}$ is useful to illustrate the plot box aspect ratio. First plot the function over the range $-2 \le x \le 2$, $-2 \le y \le 2$,

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```



Querying the plot box aspect ratio shows that the plot box is square.

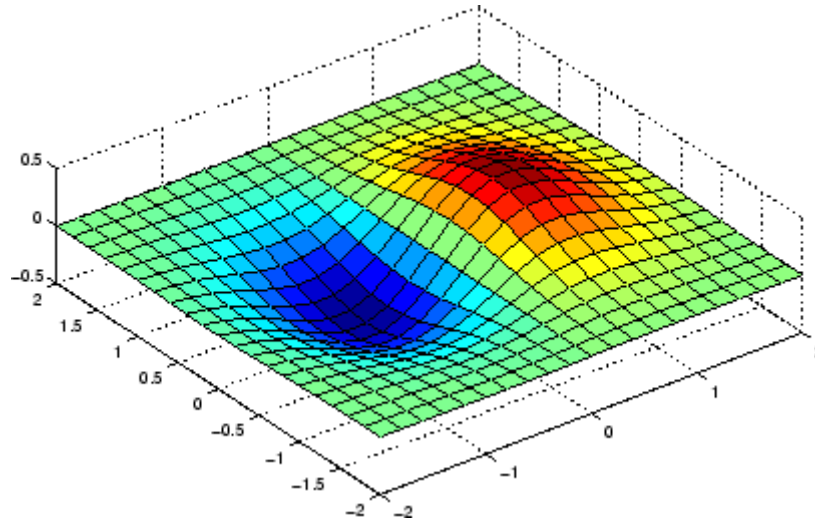
```
pbaspect
ans =
     1   1   1
```

It is also interesting to look at the data aspect ratio selected by MATLAB.

```
daspect
ans =
     4   4   1
```

To illustrate the interaction between the plot box and data aspect ratios, set the data aspect ratio to `[1 1 1]` and again query the plot box aspect ratio.

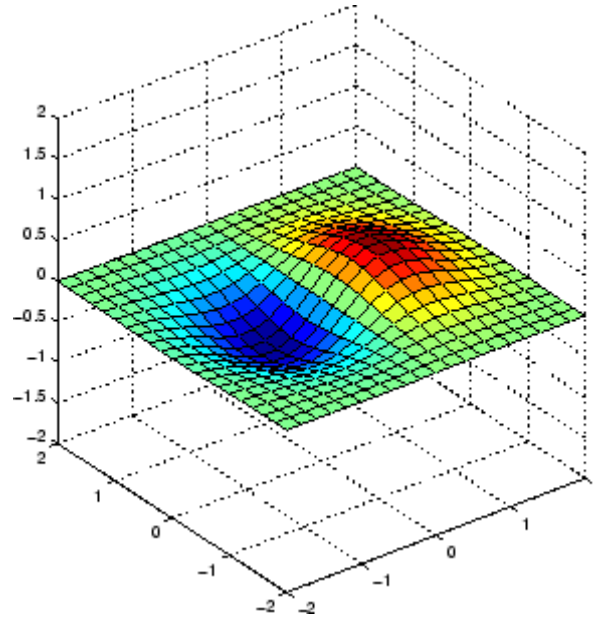
```
daspect([1 1 1])
```




```
pbaspect
ans =
     4   4   1
```

The plot box aspect ratio has changed to accommodate the specified data aspect ratio. Now suppose you want the plot box aspect ratio to be `[1 1 1]` as well.

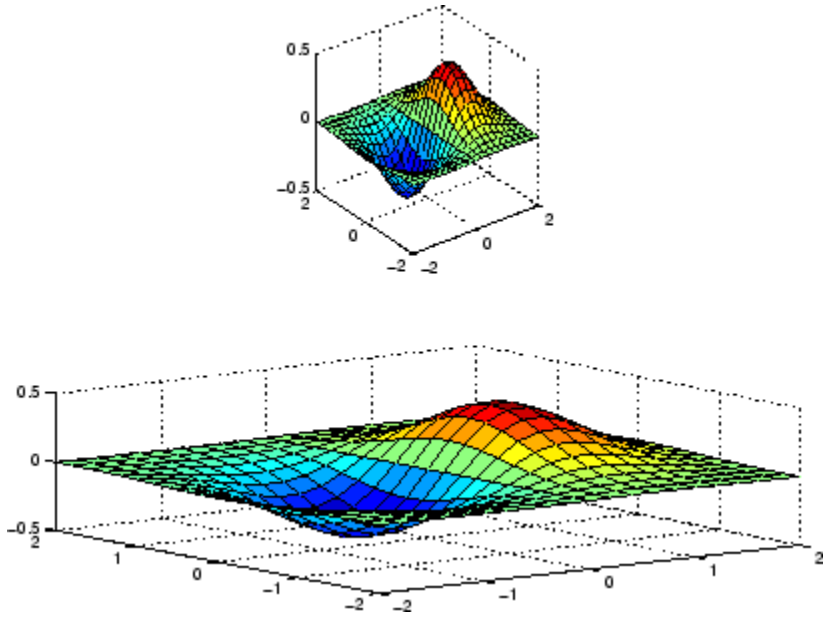
```
pbaspect([1 1 1])
```



Notice how MATLAB changed the axes limits because of the constraints introduced by specifying both the plot box and data aspect ratios.

You can also use `pbaspect` to disable stretch-to-fill. For example, displaying two subplots in one figure can give surface plots a squashed appearance. Disabling stretch-to-fill,

```
upper_plot = subplot(211);
surf(x,y,z)
lower_plot = subplot(212);
surf(x,y,z)
pbaspect(upper_plot,'manual')
```



See Also

`axis`, `daspect`, `xlim`, `ylim`, `zlim`

The axes properties `DataAspectRatio`, `PlotBoxAspectRatio`, `XLim`, `YLim`, `ZLim`

Setting Aspect Ratio in the MATLAB Graphics manual

Axes Aspect Ratio Properties in the 3-D Visualization manual

Purpose

Preconditioned conjugate gradients method

Syntax

```
x = pcg(A,b)
pcg(A,b,tol)
pcg(A,b,tol,maxit)
pcg(A,b,tol,maxit,M)
pcg(A,b,tol,maxit,M1,M2)
pcg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = pcg(A,b,...)
[x,flag,relres] = pcg(A,b,...)
[x,flag,relres,iter] = pcg(A,b,...)
[x,flag,relres,iter,resvec] = pcg(A,b,...)
```

Description

`x = pcg(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric and positive definite, and should also be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See Function Handles in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `pcg` converges, a message to that effect is displayed. If `pcg` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`pcg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `pcg` uses the default, `1e-6`.

`pcg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `pcg` uses the default, `min(n,20)`.

`pcg(A,b,tol,maxit,M)` and `pcg(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and

effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `pcg` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`pcg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `pcg` uses the default, an all-zero vector.

`[x,flag] = pcg(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`pcg` converged to the desired tolerance `tol` within `maxit` iterations.
1	`pcg` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`pcg` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `pcg` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = pcg(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = pcg(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = pcg(A,b,...)` also returns a vector of the residual norms at each iteration including `norm(b-A*x0)`.

Examples

Example 1

```
n1 = 21;
A = gallery('moler',n1);
b1 = A*ones(n1,1);
tol = 1e-6;
```

```
maxit = 15;
M = diag([10:-1:1 1 1:10]);
[x1,flag1,rr1,iter1,rv1] = pcg(A,b1,tol,maxit,M);
```

Alternatively, you can use the following parameterized matrix-vector product function `afun` in place of the matrix `A`:

```
afun = @(x,n)gallery('moler',n)*x;
n2 = 21;
b2 = afun(ones(n2,1),n2);
[x2,flag2,rr2,iter2,rv2] = pcg(@(x)afun(x,n2),b2,tol,maxit,M);
```

Example 2

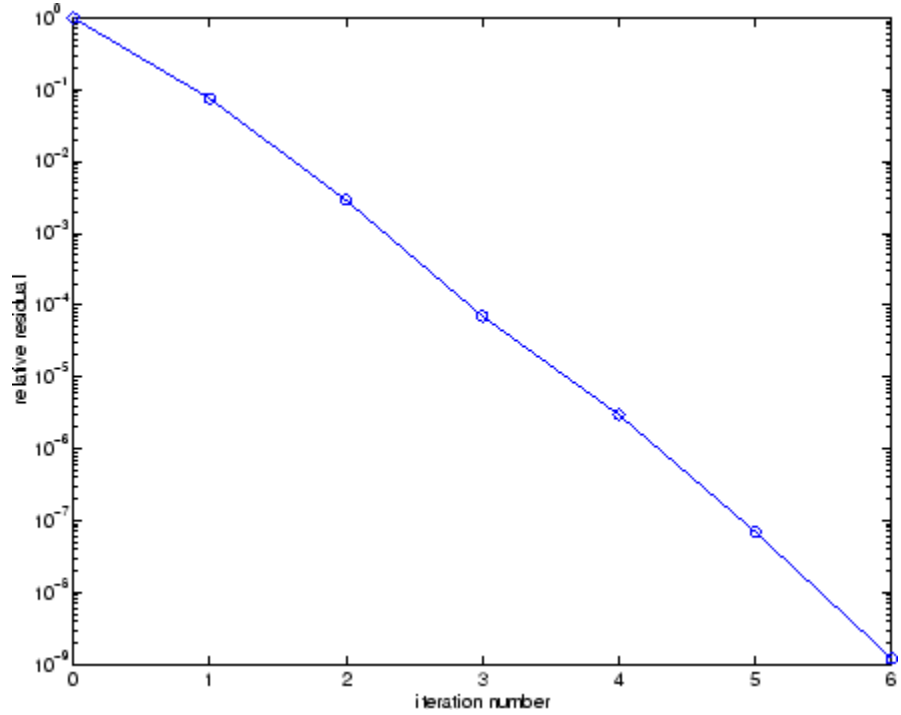
```
A = delsq(numgrid('C',25));
b = ones(length(A),1);
[x,flag] = pcg(A,b)
```

`flag` is `1` because `pcg` does not converge to the default tolerance of `1e-6` within the default 20 iterations.

```
R = cholinc(A,1e-3);
[x2,flag2,relres2,iter2,resvec2] = pcg(A,b,1e-8,10,R',R)
```

`flag2` is `0` because `pcg` converges to the tolerance of `1.2e-9` (the value of `relres2`) at the sixth iteration (the value of `iter2`) when preconditioned by the incomplete Cholesky factorization with a drop tolerance of `1e-3`. `resvec2(1) = norm(b)` and `resvec2(7) = norm(b-A*x2)`. You can follow the progress of `pcg` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```



See Also

`bicg`, `bicgstab`, `cgs`, `cholinc`, `gmres`, `lsqr`, `minres`, `qmr`, `symmlq`

`function_handle` (`@`), `mldivide` (`\`)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Purpose

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)

Syntax

```
yi = pchip(x,y,xi)
pp = pchip(x,y)
```

Description

`yi = pchip(x,y,xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by piecewise cubic interpolation within vectors `x` and `y`. The vector `x` specifies the points at which the data `y` is given. If `y` is a matrix, then the interpolation is performed for each column of `y` and `yi` is `length(xi)`-by-`size(y,2)`.

`pp = pchip(x,y)` returns a piecewise polynomial structure for use by `ppval`. `x` can be a row or column vector. `y` is a row or column vector of the same length as `x`, or a matrix with `length(x)` columns.

`pchip` finds values of an underlying interpolating function $P(x)$ at intermediate points, such that:

- On each subinterval $x_k \le x \le x_{k+1}$, $P(x)$ is the cubic Hermite interpolant to the given values and certain slopes at the two endpoints.
- $P(x)$ interpolates y, i.e., $P(x_j) = y_j$, and the first derivative $P'(x)$ is continuous. $P''(x)$ is probably not continuous; there may be jumps at the x_j.
- The slopes at the x_j are chosen in such a way that $P(x)$ preserves the shape of the data and respects monotonicity. This means that, on intervals where the data are monotonic, so is $P(x)$; at points where the data has a local extremum, so does $P(x)$.

Note If y is a matrix, $P(x)$ satisfies the above for each column of y.

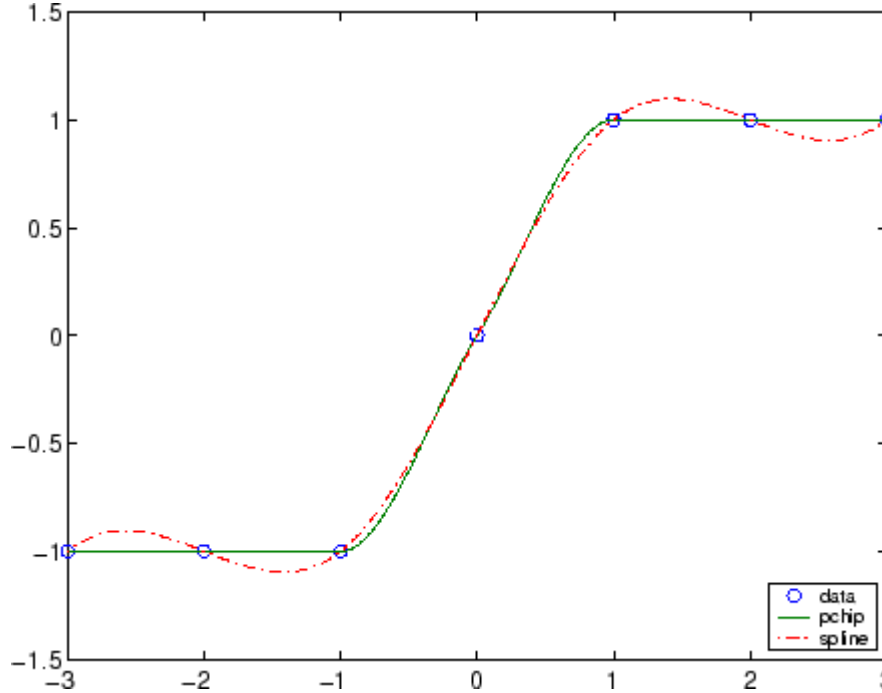
Remarks

`spline` constructs $S(x)$ in almost the same way `pchip` constructs $P(x)$. However, `spline` chooses the slopes at the x_j differently, namely to make even $S''(x)$ continuous. This has the following effects:

- `spline` produces a smoother result, i.e. $S''(x)$ is continuous.
- `spline` produces a more accurate result if the data consists of values of a smooth function.
- `pchip` has no overshoots and less oscillation if the data are not smooth.
- `pchip` is less expensive to set up.
- The two are equally expensive to evaluate.

Examples

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
t = -3:.01:3;
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'o',t,p,'-',t,s,'-.')
legend('data','pchip','spline',4)
```



See Also

`interp1`, `spline`, `ppval`

References

[1] Fritsch, F. N. and R. E. Carlson, "Monotone Piecewise Cubic Interpolation," *SIAM J. Numerical Analysis*, Vol. 17, 1980, pp.238-246.

[2] Kahaner, David, Cleve Moler, Stephen Nash, *Numerical Methods and Software*, Prentice Hall, 1988.

Purpose

Create preparsed pseudocode file (P-file)

Syntax

```
pcode fun
pcode *.m
pcode fun1 fun2 ...
pcode... -inplace
```

Description

`pcode fun` parses the M-file `fun.m` into the P-file `fun.p` and puts it into the current directory. The original M-file can be anywhere on the search path.

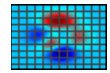
`pcode *.m` creates P-files for all the M-files in the current directory.

`pcode fun1 fun2 ...` creates P-files for the listed functions.

`pcode... -inplace` creates P-files in the same directory as the M-files. An error occurs if the files can't be created.

Purpose

Pseudocolor (checkerboard) plot



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
pcolor(C)
pcolor(X,Y,C)
pcolor(axes_handles,...)
h = pcolor(...)
```

Description

A pseudocolor plot is a rectangular array of cells with colors determined by `C`. MATLAB creates a pseudocolor plot using each set of four adjacent points in `C` to define a surface rectangle (i.e., cell).

The default `shading` is `faceted`, which colors each cell with a single color. The last row and column of `C` are not used in this case. With `shading interp`, each cell is colored by bilinear interpolation of the colors at its four vertices, using all elements of `C`.

The minimum and maximum elements of `C` are assigned the first and last colors in the colormap. Colors for the remaining elements in `C` are determined by a linear mapping from value to colormap element.

`pcolor(C)` draws a pseudocolor plot. The elements of `C` are linearly mapped to an index into the current colormap. The mapping from `C` to the current colormap is defined by `colormap` and `caxis`.

`pcolor(X,Y,C)` draws a pseudocolor plot of the elements of `C` at the locations specified by `X` and `Y`. The plot is a logically rectangular, two-dimensional grid with vertices at the points `[X(i,j), Y(i,j)]`. `X` and `Y` are vectors or matrices that specify the spacing of the grid lines. If

`X` and `Y` are vectors, `X` corresponds to the columns of `C` and `Y` corresponds to the rows. If `X` and `Y` are matrices, they must be the same size as `C`.

`pcolor(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = pcolor(...)` returns a handle to a `surface` graphics object.

Remarks

A pseudocolor plot is a flat surface plot viewed from above. `pcolor(X,Y,C)` is the same as viewing `surf(X,Y,zeros(size(X)),C)` using `view([0 90])`.

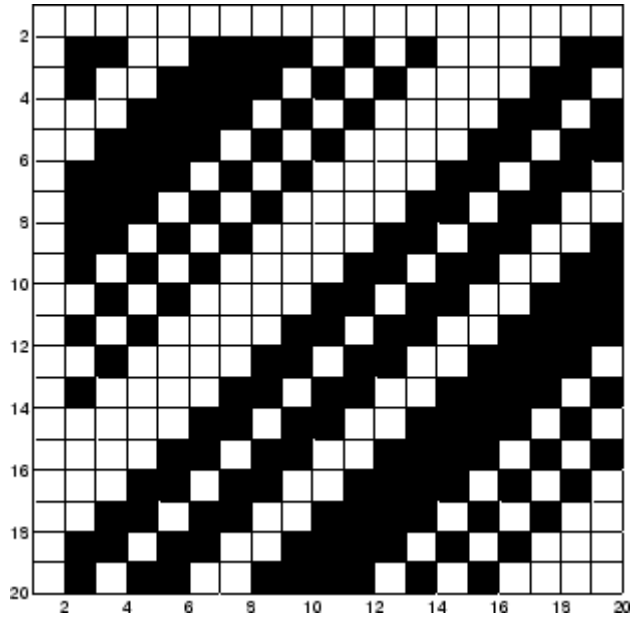
When you use `shading faceted` or `shading flat`, the constant color of each cell is the color associated with the corner having the smallest *x*-*y* coordinates. Therefore, `C(i,j)` determines the color of the cell in the *i*th row and *j*th column. The last row and column of `C` are not used.

When you use `shading interp`, each cell's color results from a bilinear interpolation of the colors at its four vertices, and all elements of `C` are used.

Examples

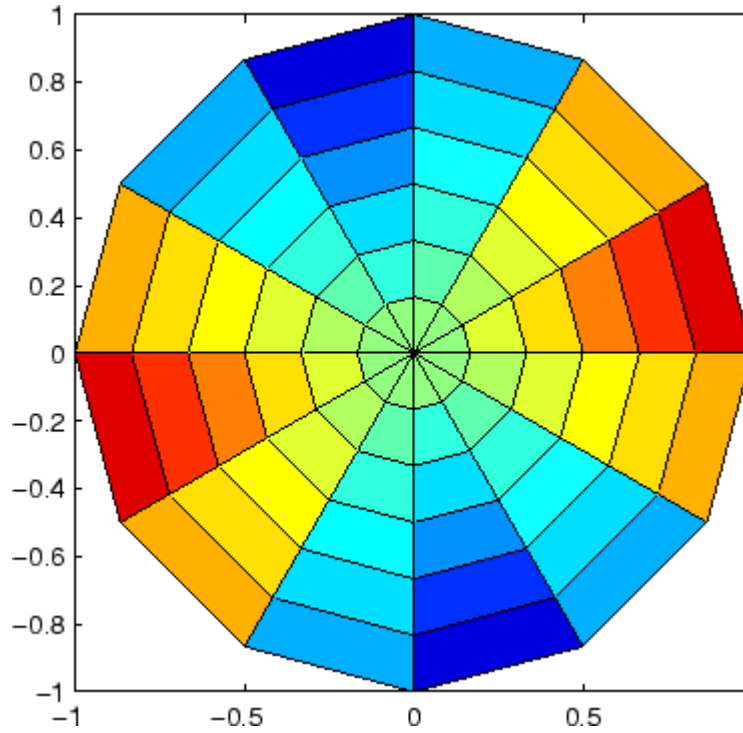
A Hadamard matrix has elements that are `+1` and `-1`. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix.

```
pcolor(hadamard(20))
colormap(gray(2))
axis ij
axis square
```



A simple color wheel illustrates a polar coordinate system.

```
n = 6;
r = (0:n)'/n;
theta = pi*(-n:n)/n;
X = r*cos(theta);
Y = r*sin(theta);
C = r*cos(2*theta);
pcolor(X,Y,C)
axis equal tight
```



Algorithm

The number of vertex colors for `pcolor(C)` is the same as the number of cells for `image(C)`. `pcolor` differs from `image` in that `pcolor(C)` specifies the colors of vertices, which are scaled to fit the colormap; changing the axes `clim` property changes this color mapping. `image(C)` specifies the colors of cells and directly indexes into the colormap without scaling. Additionally, `pcolor(X,Y,C)` can produce parametric grids, which is not possible with `image`.

See Also

`caxis`, `image`, `mesh`, `shading`, `surf`, `view`

Purpose

Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D

Syntax

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
```

Arguments

`m`	A parameter corresponding to the symmetry of the problem. `m` can be slab = `0`, cylindrical = `1`, or spherical = `2`.
`pdefun`	A handle to a function that defines the components of the PDE.
`icfun`	A handle to a function that defines the initial conditions.
`bcfun`	A handle to a function that defines the boundary conditions.
`xmesh`	A vector [`x0`, `x1`, ..., `xn`] specifying the points at which a numerical solution is requested for every value in `tspan`. The elements of `xmesh` must satisfy `x0 < x1 < ... < xn`. The length of `xmesh` must be `>= 3`.
`tspan`	A vector [`t0`, `t1`, ..., `tf`] specifying the points at which a solution is requested for every value in `xmesh`. The elements of `tspan` must satisfy `t0 < t1 < ... < tf`. The length of `tspan` must be `>= 3`.
`options`	Some options of the underlying ODE solver are available in `pdepe`: `RelTol`, `AbsTol`, `NormControl`, `InitialStep`, and `MaxStep`. In most cases, default values for these options provide satisfactory solutions. See `odeset` for details.

Description

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)` solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t. `pdefun`, `icfun`, and

bcfun are function handles. See “Function Handles” in the MATLAB Programming documentation for more information. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at times specified in tspan. The pdepe function returns values of the solution on a mesh provided in xmesh.

“Parameterizing Functions Called by Function Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the functions pdefun, icfun, or bcfun, if necessary.

pdepe solves PDEs of the form:

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x, t, u, \frac{\partial u}{\partial x}\right)\right) + s\left(x, t, u, \frac{\partial u}{\partial x}\right) \quad (2\text{-}2)$$

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. The interval $[a, b]$ must be finite. m can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m > 0$, then a must be >= 0.

In Equation 2-2, $f(x, t, u, \partial u/\partial x)$ is a flux term and $s(x, t, u, \partial u/\partial x)$ is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c(x, t, u, \partial u/\partial x)$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if those values of x are mesh points. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

For $t = t_0$ and all x, the solution components satisfy initial conditions of the form

$$u(x, t_0) = u_0(x)$$

(2-3)

For all t and either $x = a$ or $x = b$, the solution components satisfy a boundary condition of the form

$$p(x, t, u) + q(x, t) f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0 \qquad \textbf{(2-4)}$$

Elements of q are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux f rather than $\partial u/\partial x$. Also, of the two coefficients, only p can depend on u.

In the call `sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)`:

- `m` corresponds to m.
- `xmesh(1)` and `xmesh(end)` correspond to a and b.
- `tspan(1)` and `tspan(end)` correspond to t_0 and t_f.
- `pdefun` computes the terms c, f, and s (Equation 2-2). It has the form

  ```
  [c,f,s] = pdefun(x,t,u,dudx)
  ```

  The input arguments are scalars `x` and `t` and vectors `u` and `dudx` that approximate the solution u and its partial derivative with respect to x, respectively. `c`, `f`, and `s` are column vectors. `c` stores the diagonal elements of the matrix c (Equation 2-2).

- `icfun` evaluates the initial conditions. It has the form

  ```
  u = icfun(x)
  ```

  When called with an argument `x`, `icfun` evaluates and returns the initial values of the solution components at `x` in the column vector `u`.

- `bcfun` evaluates the terms p and q of the boundary conditions (Equation 2-4). It has the form

  ```
  [pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
  ```

ul is the approximate solution at the left boundary xl = a and ur is the approximate solution at the right boundary xr = b. pl and ql are column vectors corresponding to p and q evaluated at xl, similarly pr and qr correspond to xr. When $m > 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. pdepe imposes this boundary condition automatically and it ignores values returned in pl and ql.

pdepe returns the solution as a multidimensional array sol. u_i = ui = sol(:,:,i) is an approximation to the ith component of the solution vector u. The element ui(j,k) = sol(j,k,i) approximates u_i at (t, x) = (tspan(j),xmesh(k)).

ui = sol(j,:,i) approximates component i of the solution at time tspan(j) and mesh points xmesh(:). Use pdeval to compute the approximation and its partial derivative $\partial u_i / \partial x$ at points not included in xmesh. See pdeval for details.

sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options) solves as above with default integration parameters replaced by values in options, an argument created with the odeset function. Only some of the options of the underlying ODE solver are available in pdepe: RelTol, AbsTol, NormControl, InitialStep, and MaxStep. The defaults obtained by leaving off the input argument options will generally be satisfactory. See odeset for details.

Remarks

- The arrays xmesh and tspan play different roles in pdepe.

  tspan – The pdepe function performs the time integration with an ODE solver that selects both the time step and formula dynamically. The elements of tspan merely specify where you want answers and the cost depends weakly on the length of tspan.

  xmesh – Second order approximations to the solution are made on the mesh specified in xmesh. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. pdepe does *not* select the mesh in x automatically. You must provide an appropriate fixed mesh in xmesh. The cost depends strongly on the length of

xmesh. When $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

- The time integration is done with ode15s. pdepe exploits the capabilities of ode15s for solving the differential-algebraic equations that arise when Equation 2-2 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.
- After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not "consistent" with the discretization, pdepe tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, pdepe can find consistent initial conditions close to the given ones. If pdepe displays a message that it has difficulty finding consistent initial conditions, try refining the mesh.

  No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

Examples

Example 1. This example illustrates the straightforward formulation, computation, and plotting of the solution of a single PDE.

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial condition

$$u(x, 0) = \sin \pi x$$

and boundary conditions

$$u(0, t) \equiv 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$

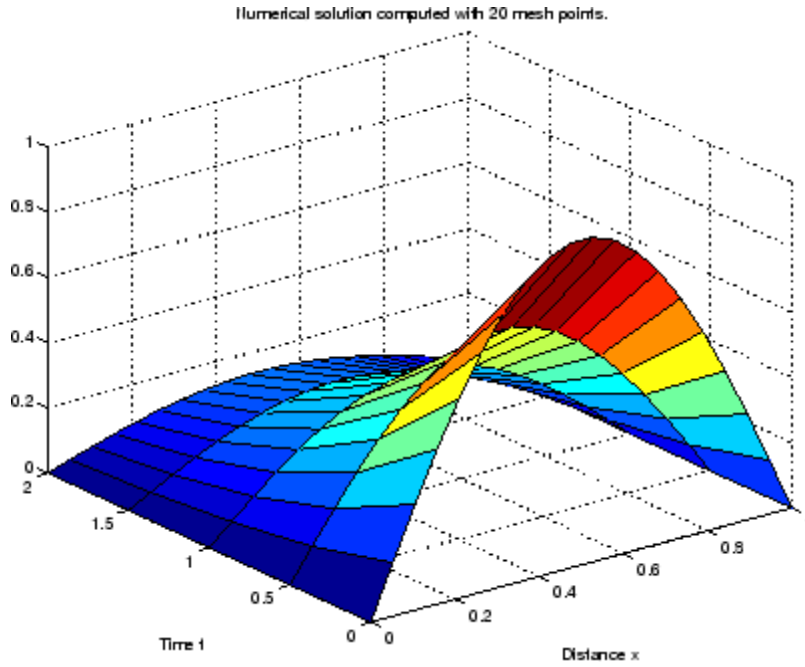
It is convenient to use subfunctions to place all the functions required by pdepe in a single M-file.

```
function pdex1

m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);

sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
% Extract the first solution component as u.
u = sol(:,:,1);

% A surface plot is often a good way to study a solution.
surf(x,t,u)
title('Numerical solution computed with 20 mesh points.')
xlabel('Distance x')
ylabel('Time t')

% A solution profile can also be illuminating.
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
% --------------------------------------------------------------
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
% --------------------------------------------------------------
function u0 = pdex1ic(x)
u0 = sin(pi*x);
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
```

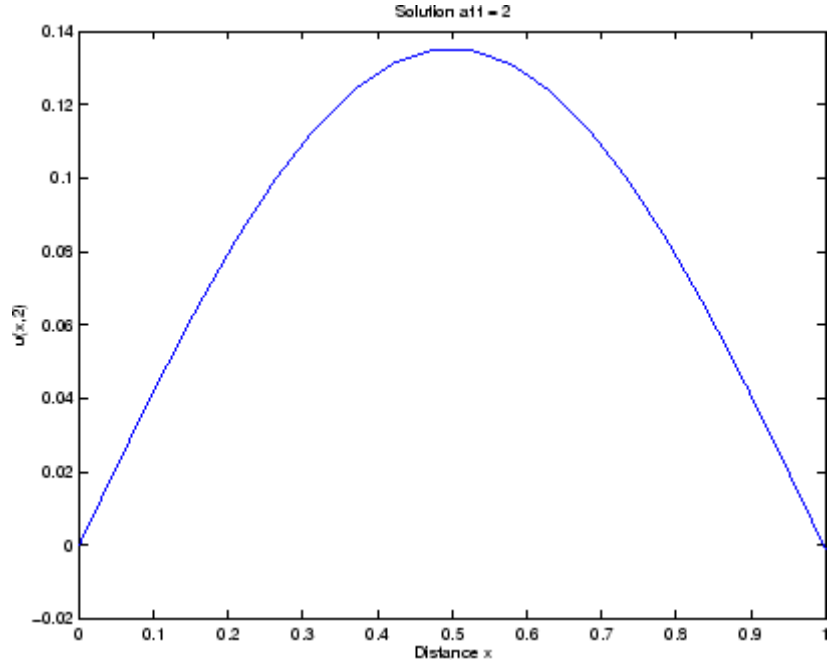
```
pr = pi * exp(-t);
qr = 1;
```

In this example, the PDE, initial condition, and boundary conditions are coded in subfunctions `pdex1pde`, `pdex1ic`, and `pdex1bc`.

The surface plot shows the behavior of the solution.



The following plot shows the solution profile at the final value of `t` (i.e., `t = 2`).



Example 2. This example illustrates the solution of a system of PDEs. The problem has boundary layers at both ends of the interval. The solution changes rapidly for small t.

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024 \frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$

$$\frac{\partial u_2}{\partial t} = 0.170 \frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$.

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial conditions

$$u_1(x, 0) \equiv 1$$

$$u_2(x, 0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0, t) \equiv 0$$

$$u_2(0, t) \equiv 0$$

$$u_1(1, t) \equiv 1$$

$$\frac{\partial u_2}{\partial x}(1, t) \equiv 0$$

In the form expected by `pdepe`, the equations are

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} .* \frac{\partial}{\partial t}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} + \begin{bmatrix} -F(u_1 - u_2) \\ F(u_1 - u_2) \end{bmatrix}$$

The boundary conditions on the partial derivatives of u have to be written in terms of the flux. In the form expected by `pdepe`, the left boundary condition is

$$\begin{bmatrix} 0 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} .* \begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and the right boundary condition is

$$\begin{bmatrix} u_1 - 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} .* \begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

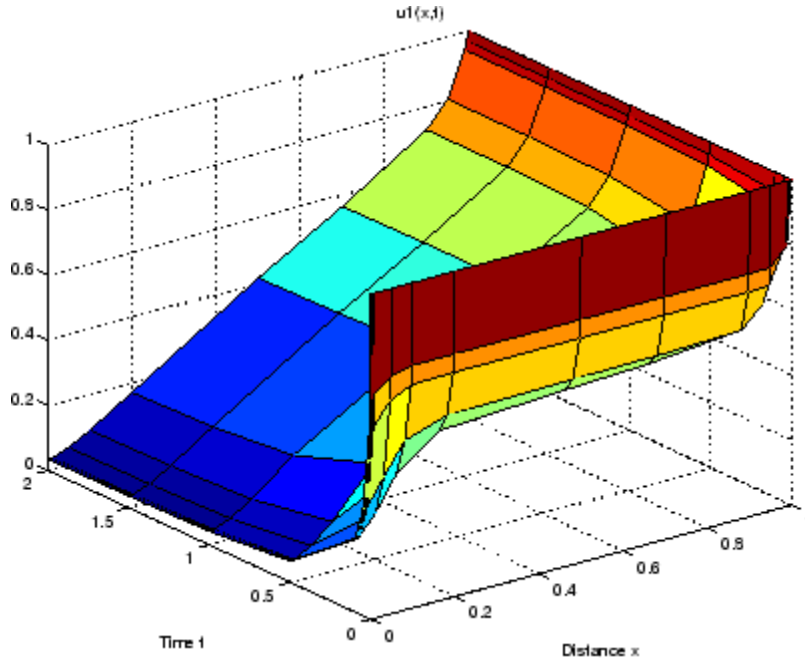
The solution changes rapidly for small t. The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, the example must select the output times accordingly. There are boundary layers in the solution at both ends of [0,1], so the example places mesh points near `0` and `1` to resolve these sharp changes. Often some experimentation is needed to select a mesh that reveals the behavior of the solution.

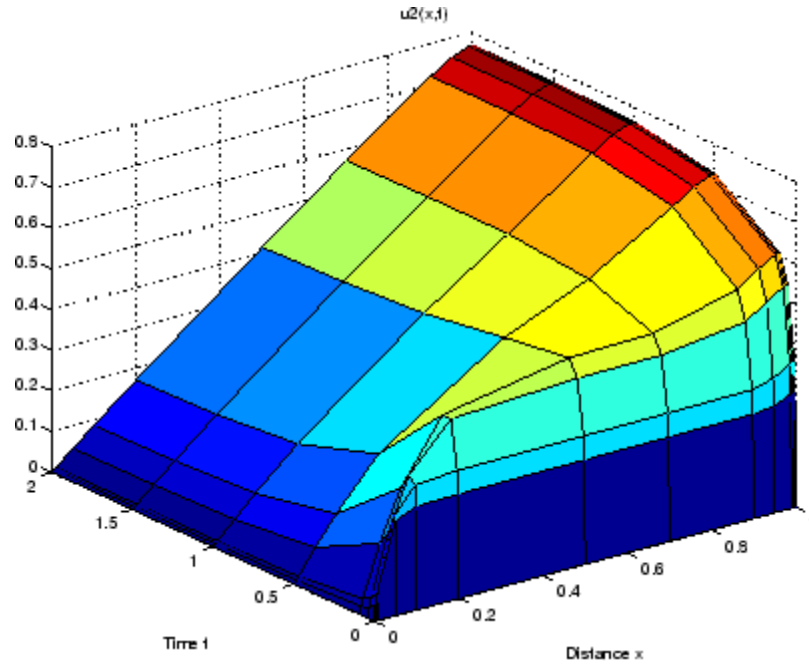
```
function pdex4
m = 0;
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
u1 = sol(:,:,1);
u2 = sol(:,:,2);

figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')

figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
% --------------------------------------------------------------
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
```

```
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
% --------------------------------------------------------------
function u0 = pdex4ic(x);
u0 = [1; 0];
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];
```

In this example, the PDEs, initial conditions, and boundary conditions are coded in subfunctions `pdex4pde`, `pdex4ic`, and `pdex4bc`.

The surface plots show the behavior of the solution components.





See Also

`function_handle (@)`, `pdeval`, `ode15s`, `odeset`, `odeget`

References

[1] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp.1-32.

Purpose

Evaluate numerical solution of PDE using output of pdepe

Syntax

```
[uout,duoutdx] = pdeval(m,x,ui,xout)
```

Arguments

m	Symmetry of the problem: slab = 0, cylindrical = 1, spherical = 2. This is the first input argument used in the call to pdepe.
xmesh	A vector [x0, x1, ..., xn] specifying the points at which the elements of ui were computed. This is the same vector with which pdepe was called.
ui	A vector sol(j,:,i) that approximates component i of the solution at time t_f and mesh points xmesh, where sol is the solution returned by pdepe.
xout	A vector of points from the interval [x0,xn] at which the interpolated solution is requested.

Description

`[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution u_i and its partial derivative $\partial u_i/\partial x$ at points from the interval [x0,xn]. The `pdeval` function returns the computed values in `uout` and `duoutdx`, respectively.

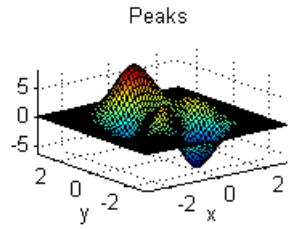
Note `pdeval` evaluates the partial derivative $\partial u_i/\partial x$ rather than the flux f. Although the flux is continuous, the partial derivative may have a jump at a material interface.

See Also

pdepe

Purpose

Example function of two variables



Syntax

```
Z = peaks;
Z = peaks(n);
Z = peaks(V);
Z = peaks(X,Y);

peaks;
peaks(N);
peaks(V);
peaks(X,Y);

X,Y,Z] = peaks;
[X,Y,Z] = peaks(n);
[X,Y,Z] = peaks(V);
```

Description

`peaks` is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating `mesh`, `surf`, `pcolor`, `contour`, and so on.

`Z = peaks;` returns a 49-by-49 matrix.

`Z = peaks(n);` returns an `n`-by-`n` matrix.

`Z = peaks(V);` returns an `n`-by-`n` matrix, where `n = length(V)`.

`Z = peaks(X,Y);` evaluates `peaks` at the given `X` and `Y` (which must be the same size) and returns a matrix the same size.

`peaks(...)` (with no output argument) plots the peaks function with `surf`.

`[X,Y,Z] = peaks(...);` returns two additional matrices, `X` and `Y`, for parametric plots, for example, `surf(X,Y,Z,del2(Z))`. If not given as input, the underlying matrices `X` and `Y` are

```
[X,Y] = meshgrid(V,V)
```

where `V` is a given vector, or `V` is a vector of length `n` with elements equally spaced from -3 to 3. If no input argument is given, the default `n` is 49.

See Also

`meshgrid`, `surf`

Purpose

Call Perl script using appropriate operating system executable

Syntax

```
perl('perlfile')
perl('perlfile',arg1,arg2,...)
result = perl(...)
```

Description

`perl('perlfile')` calls the Perl script `perlfile`, using the appropriate operating system Perl executable. Perl is included with MATLAB on Windows systems, and thus MATLAB users can run M-files containing the `perl` function. On Unix systems, MATLAB just calls the Perl interpreter that's available with the OS

`perl('perlfile',arg1,arg2,...)` calls the Perl script `perlfile`, using the appropriate operating system Perl executable, and passes the arguments `arg1`, `arg2`, and so on, to `perlfile`.

`result = perl(...)` returns the results of attempted Perl call to `result`.

Examples

Given the Perl script, `hello.pl`

```
$input = $ARGV[0];
print "Hello $input.";
```

run the following statement in MATLAB

```
perl('hello.pl','World')
```

MATLAB returns

```
ans =
Hello World.
```

It is sometimes beneficial to use Perl scripts instead of MATLAB code. The `perl` function allows you to run those scripts from within MATLAB. Specific examples where you might choose to use a Perl script include

- Perl script already exists

- Perl script preprocesses data quickly, formatting it in a way more easily read by MATLAB
- Perl has features not supported by MATLAB

See Also

`!` (exclamation point), `dos`, `regexp`, `system`, `unix`

perms

Purpose

All possible permutations

Syntax

```
P = perms(v)
```

Description

`P = perms(v)`, where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible permutations of the `n` elements of `v`. Matrix `P` contains `n!` rows and `n` columns.

Examples

The command `perms(2:2:6)` returns *all* the permutations of the numbers 2, 4, and 6:

```
6     4     2
6     2     4
4     6     2
4     2     6
2     4     6
2     6     4
```

Limitations

This function is only practical for situations where `n` is less than about 15.

See Also

`nchoosek`, `permute`, `randperm`

Purpose

Rearrange dimensions of N-D array

Syntax

```
B = permute(A,order)
```

Description

`B = permute(A,order)` rearranges the dimensions of `A` so that they are in the order specified by the vector `order`. `B` has the same values of `A` but the order of the subscripts needed to access any particular element is rearranged as specified by `order`. All the elements of `order` must be unique.

Remarks

`permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

Examples

Given any matrix `A`, the statement

```
permute(A,[2 1])
```

is the same as `A'`.

For example:

```
A = [1 2; 3 4]; permute(A,[2 1])
ans =
     1     3
     2     4
```

The following code permutes a three-dimensional array:

```
X = rand(12,13,14);
Y = permute(X,[2 3 1]);
size(Y)
ans =
    13    14    12
```

See Also

`ipermute`, `circshift`

persistent

Purpose

Define persistent variable

Syntax

```
persistent X Y Z
```

Description

`persistent X Y Z` defines `X`, `Y`, and `Z` as variables that are local to the function in which they are declared; yet their values are retained in memory between calls to the function. Persistent variables are similar to global variables because MATLAB creates permanent storage for both. They differ from global variables in that persistent variables are known only to the function in which they are declared. This prevents persistent variables from being changed by other functions or from the MATLAB command line.

Persistent variables are cleared when the M-file is cleared from memory or when the M-file is changed. To keep an M-file in memory until MATLAB quits, use `mlock`.

If the persistent variable does not exist the first time you issue the `persistent` statement, it is initialized to the empty matrix.

It is an error to declare a variable persistent if a variable with the same name exists in the current workspace. MATLAB also errors if you declare any of a function's input or output arguments as persistent within that same function. For example, the following persistent declaration is invalid:

```
function myfun(argA, argB, argC)
persistent argB
```

Remarks

There is no function form of the `persistent` command (i.e., you cannot use parentheses and quote the variable names).

Example

This function prompts a user to enter a directory name to use in locating one or more files. If the user has already entered this information, and it requires no modification, they do not need to enter it again. This is because the function stores it in a persistent variable (`lastDir`), and offers it as the default selection. Here is the function definition:

```
function find_file(file)
persistent lastDir

if isempty(lastDir)
    prompt = 'Enter directory:  ';
else
    prompt = ['Enter directory[' lastDir ']:  '];
end
response = input(prompt, 's');

if ~isempty(response)
    dirName = response;
else
    dirName = lastDir;
end

dir(strcat(dirName, file))
lastDir = dirName;
```

Execute the function twice. The first time, it prompts you to enter the information and does not offer a default:

```
cd(matlabroot)

find_file('is*.m')
Enter directory:  toolbox/matlab/strfun/

iscellstr.m   ischar.m   isletter.m   isspace.m   isstr.m
isstrprop.m
```

The second time, it does offer a default taken from the persistent variable `dirName`:

```
find_file('is*.m')
Enter directory[toolbox/matlab/strfun/]:
toolbox/matlab/elmat/
```

```
isempty.m             isfinite.m           isscalar.m
isequal.m             isinf.m              isvector.m
isequalwithequalnans.m  isnan.m
```

See Also

global, clear, mislocked, mlock, munlock, isempty

Purpose

Ratio of circle's circumference to its diameter, π

Syntax

```
pi
```

Description

`pi` returns the floating-point number nearest the value of π. The expressions `4*atan(1)` and `imag(log(-1))` provide the same value.

Examples

The expression `sin(pi)` is not exactly zero because `pi` is not exactly π.

```
sin(pi)

ans =

    1.2246e-16
```

See Also

`ans`, `eps`, `i`, `Inf`, `j`, `NaN`

pie

Purpose

Pie chart

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
pie(X)
pie(X,explode)
pie(...,labels)
pie(axes_handle,...)
h = pie(...)
```

Description

`pie(X)` draws a pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie(X,explode)` offsets a slice from the pie. `explode` is a vector or matrix of zeros and nonzeros that correspond to `X`. A nonzero value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`. For example,

```
pie(1:3,{'Taxes','Expenses','Profit'})
```

`pie(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = pie(...)` returns a vector of handles to patch and text graphics objects.

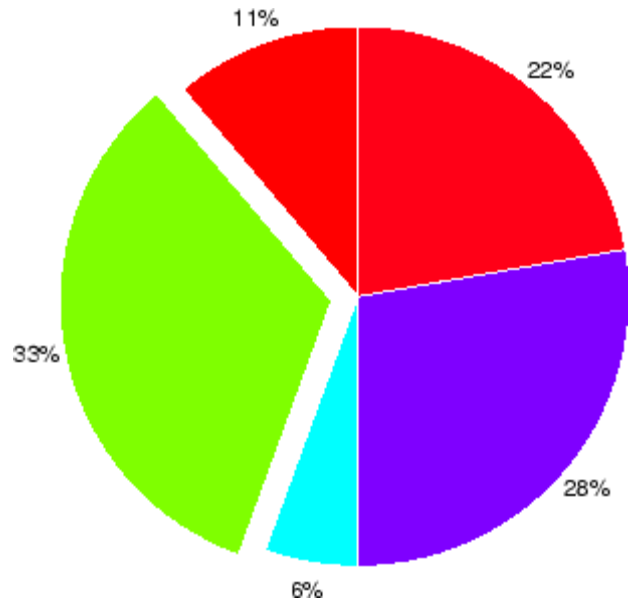
Remarks

The values in `X` are normalized via `X/sum(X)` to determine the area of each slice of the pie. If `sum(X) ≤ 1`, the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if `sum(X) < 1`.

Examples

Emphasize the second slice in the chart by setting its corresponding `explode` element to 1.

```
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie(x,explode)
colormap jet
```



See Also

`pie3`

pie3

Purpose

3-D pie chart

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
pie3(X)
pie3(X,explode)
pie3(...,labels)
pie3(axes_handle,...)
h = pie3(...)
```

Description

`pie3(X)` draws a three-dimensional pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie3(X,explode)` specifies whether to offset a slice from the center of the pie chart. `X(i,j)` is offset from the center of the pie chart if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie3(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`. For example,

```
pie3(1:3,{'Taxes','Expenses','Profit'})
```

`pie3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = pie3(...)` returns a vector of handles to patch, surface, and text graphics objects.

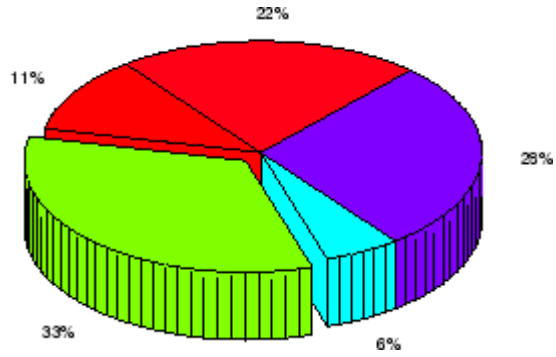
Remarks

The values in `X` are normalized via `X/sum(X)` to determine the area of each slice of the pie. If `sum(X) ≤ 1`, the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if `sum(X) < 1`.

Examples

Offset a slice in the pie chart by setting the corresponding `explode` element to `1`:

```
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie3(x,explode)
colormap hsv
```



See Also

`pie`

Purpose

Moore-Penrose pseudoinverse of matrix

Syntax

```
B = pinv(A)
B = pinv(A,tol)
```

Definition

The Moore-Penrose pseudoinverse is a matrix `B` of the same dimensions as `A'` satisfying four conditions:

```
A*B*A = A
B*A*B = B
A*B is Hermitian
B*A is Hermitian
```

The computation is based on `svd(A)` and any singular values less than `tol` are treated as zero.

Description

`B = pinv(A)` returns the Moore-Penrose pseudoinverse of `A`.

`B = pinv(A,tol)` returns the Moore-Penrose pseudoinverse and overrides the default tolerance, `max(size(A))*norm(A)*eps`.

Examples

If `A` is square and not singular, then `pinv(A)` is an expensive way to compute `inv(A)`. If `A` is not square, or is square and singular, then `inv(A)` does not exist. In these cases, `pinv(A)` has some of, but not all, the properties of `inv(A)`.

If `A` has more rows than columns and is not of full rank, then the overdetermined least squares problem

```
minimize norm(A*x-b)
```

does not have a unique solution. Two of the infinitely many solutions are

```
x = pinv(A)*b
```

and

```
y = A\b
```

These two are distinguished by the facts that `norm(x)` is smaller than the norm of any other solution and that `y` has the fewest possible nonzero components.

For example, the matrix generated by

```
A = magic(8); A = A(:,1:6)
```

is an 8-by-6 matrix that happens to have `rank(A) = 3`.

```
A =
    64     2     3    61    60     6
     9    55    54    12    13    51
    17    47    46    20    21    43
    40    26    27    37    36    30
    32    34    35    29    28    38
    41    23    22    44    45    19
    49    15    14    52    53    11
     8    58    59     5     4    62
```

The right-hand side is `b = 260*ones(8,1)`,

```
b =
     260
     260
     260
     260
     260
     260
     260
     260
```

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to `A*x = b` would be a vector of all `1`'s. With only six columns, the equations are still consistent, so a solution exists, but it is not all `1`'s. Since the matrix is rank deficient, there are infinitely many solutions. Two of them are

```
x = pinv(A)*b
```

which is

```
x =
    1.1538
    1.4615
    1.3846
    1.3846
    1.4615
    1.1538
```

and

```
y = A\b
```

which produces this result.

```
Warning: Rank deficient, rank = 3  tol =   1.8829e-013.
y =
    4.0000
    5.0000
         0
         0
         0
   -1.0000
```

Both of these are exact solutions in the sense that `norm(A*x-b)` and `norm(A*y-b)` are on the order of roundoff error. The solution `x` is special because

```
norm(x) = 3.2817
```

is smaller than the norm of any other solution, including

```
norm(y) = 6.4807
```

On the other hand, the solution `y` is special because it has only three nonzero components.

See Also

`inv`, `qr`, `rank`, `svd`

Purpose

Givens plane rotation

Syntax

```
[G,y] = planerot(x)
```

Description

`[G,y] = planerot(x)` where `x` is a 2-component column vector, returns a 2-by-2 orthogonal matrix G so that `y = G*x` has `y(2) = 0`.

Examples

```
x = [3 4];
[G,y] = planerot(x')

G  =
    0.6000    0.8000
   -0.8000    0.6000

y =
     5
     0
```

See Also

`qrdelete`, `qrinsert`

playshow

Purpose

Run M-file demo (deprecated; use `echodemo` instead)

Syntax

```
playshow filename
```

Description

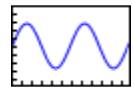
`playshow filename` runs `filename`, which is a demo. Replace `playshow filename` with `echodemo filename`. Note that other arguments supported by `playshow` are not supported by `echodemo`.

See Also

`demo`, `echodemo`, `helpbrowser`

Purpose

2-D line plot



Contents

GUI Alternatives

Use the Plot Selector  to graph selected variables in the Workspace Browser and the Plot Catalog, accessed from the Figure Palette. Directly manipulate graphs in *plot edit* mode, and modify them using the Property Editor. For details, see Using Plot Edit Mode, and The Figure Palette in the MATLAB Graphics documentation, and also Creating Graphics from the Workspace Browser in the MATLAB Desktop documentation.

Syntax

```
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineSpec,...)
plot(...,'PropertyName',PropertyValue,...)
plot(axes_handle,...)
h = plot(...)
```

```
hlines = plot('v6',...)
```

Description

`plot(Y)` plots the columns of `Y` versus their index if `Y` is a real number. If `Y` is complex, `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`. In all other uses of `plot`, the imaginary component is ignored.

`plot(X1,Y1,...)` plots all lines defined by `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, the vector is plotted versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix. If `Xn` is a scalar and `Yn` is a vector, disconnected line objects are created and plotted as discrete points vertically at `Xn`.

`plot(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` is a line specification that determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs: `plot(X1,Y1,X2,Y2,LineSpec,X3,Y3)`.

Note See `LineSpec` for a list of line style, marker, and color specifiers.

`plot(...,'PropertyName',PropertyValue,...)` sets properties to the specified property values for all lineseries graphics objects created by `plot`. (See the "Examples" on page 2-2490 section for examples.)

`plot(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = plot(...)` returns a column vector of handles to lineseries graphics objects, one handle per line.

Backward-Compatible Version

`hlines = plot('v6',...)` returns the handles to line objects instead of lineseries objects.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

Cycling Through Line Colors and Styles

If you do not specify a color when plotting more than one line, `plot` automatically cycles through the colors in the order specified by the current axes `ColorOrder` property. After cycling through all the colors defined by `ColorOrder`, `plot` then cycles through the line styles defined in the axes `LineStyleOrder` property.

The default `LineStyleOrder` property has a single entry (a solid line with no marker).

By default, MATLAB resets the `ColorOrder` and `LineStyleOrder` properties each time you call `plot`. If you want the changes you make to these properties to persist, you must define these changes as default values. For example,

```
set(0,'DefaultAxesColorOrder',[0 0 0],...
      'DefaultAxesLineStyleOrder','-|-.|--|:')
```

sets the default `ColorOrder` to use only the color black and sets the `LineStyleOrder` to use solid, dash-dot, dash-dash, and dotted line styles.

Prevent Resetting of Color and Styles with hold all

The `all` option to the `hold` command prevents the `ColorOrder` and `LineStyleOrder` from being reset in subsequent `plot` commands. In the following sequence of commands, MATLAB continues to cycle through the colors defined by the axes `ColorOrder` property (see above).

```
plot(rand(12,2))
hold all
```

```
plot(randn(12,2))
```

Additional Information

- See Creating Line Plots and Annotating Graphs for more information on plotting.
- See `LineSpec` for more information on specifying line styles and colors.

Examples

Specifying the Color and Size of Markers

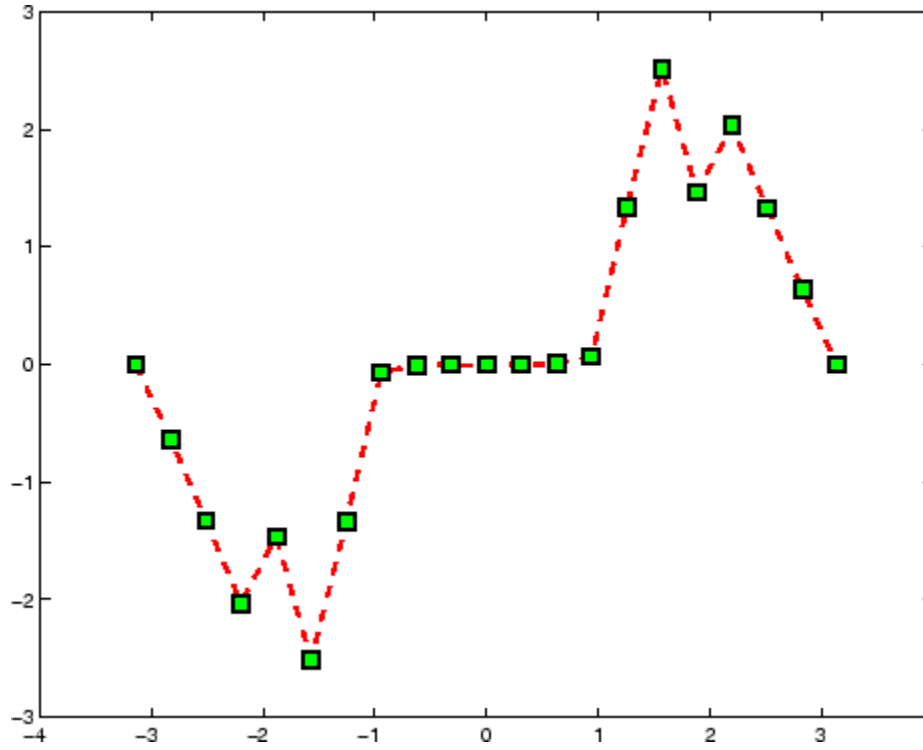
You can also specify other line characteristics using graphics properties (see `line` for a description of these properties):

- `LineWidth` — Specifies the width (in points) of the line.
- `MarkerEdgeColor` — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` — Specifies the color of the face of filled markers.
- `MarkerSize` — Specifies the size of the marker in units of points.

For example, these statements,

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor','g',...
                'MarkerSize',10)
```

produce this graph.

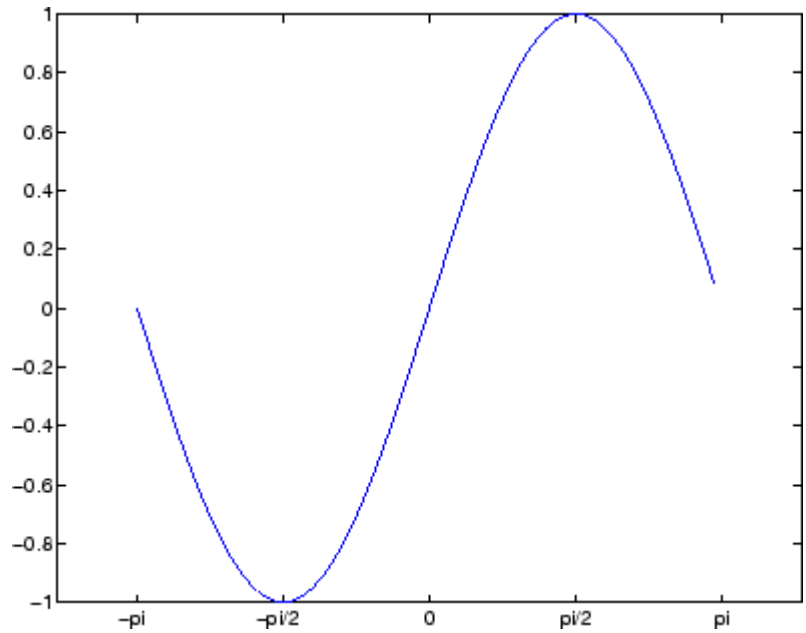


Specifying Tick-Mark Location and Labeling

You can adjust the axis tick-mark locations and the labels appearing at each tick. For example, this plot of the sine function relabels the *x*-axis with more meaningful values:

```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

Now add axis labels and annotate the point -pi/4, sin(-pi/4).



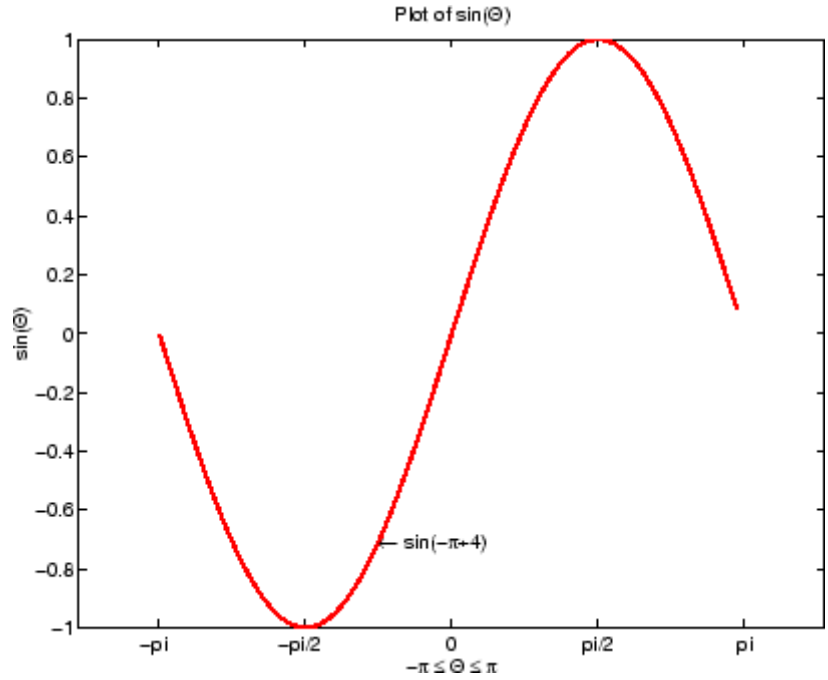
Adding Titles, Axis Labels, and Annotations

MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add an *x*- and *y*-axis label:

```
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
     'HorizontalAlignment','left')
```

Now change the line color to red by first finding the handle of the line object created by `plot` and then setting its `Color` property. In the same statement, set the `LineWidth` property to 2 points.

```
set(findobj(gca,'Type','line','Color',[0 0 1]),...
    'Color','red',...
    'LineWidth',2)
```



See Also

`axis`, `bar`, `grid`, `hold`, `legend`, `line`, `LineSpec`, `loglog`, `plot3`, `plotyy`, `semilogx`, `semilogy`, `subplot`, `title`, `xlabel`, `xlim`, `ylabel`, `ylim`, `zlabel`, `zlim`, `stem`

See the `text` `String` property for a list of symbols and how to display them.

See the Plot Editor for information on plot annotation tools in the figure window toolbar.

See “Basic Plots and Graphs” on page 1-86 for related functions.

plot (timeseries)

Purpose

Plot time series

Syntax

```
plot(ts)
plot(tsc.tsname)
plot(function)
```

Description

`plot(ts)` plots the time-series data against time and interpolates values between samples by using either zero-order-hold (`'zoh'`) or linear interpolation.

`plot(tsc.tsname)` plots the `timeseries` object `tsname` that is part of the `tscollection` `tsc`.

`plot(function)` accepts the modifiers used by the MATLAB plotting utility for numerical arrays. These modifiers can be specified as auxiliary inputs for modifying the appearance of the plot. See Examples below.

Remarks

Time-series events, when defined, are marked in the plot by a red circular marker.

Examples

`plot(ts,'-r*')` uses a regular line with the color red and marker `'*'` to render the plot.

`plot(ts,'ko','MarkerSize',3)` uses black circular markers of size `3` to render the plot.

Purpose

3-D line plot

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)
```

Description

The `plot3` function displays a three-dimensional plot of a set of data points.

`plot3(X1,Y1,Z1,...)`, where `X1`, `Y1`, `Z1` are vectors or matrices, plots one or more lines in three-dimensional space through the points whose coordinates are the elements of `X1`, `Y1`, and `Z1`.

`plot3(X1,Y1,Z1,LineSpec,...)` creates and displays all lines defined by the `Xn,Yn,Zn,LineSpec` quads, where `LineSpec` is a line specification that determines line style, marker symbol, and color of the plotted lines.

`plot3(...,'PropertyName',PropertyValue,...)` sets properties to the specified property values for all line graphics objects created by `plot3`.

`h = plot3(...)` returns a column vector of handles to lineseries graphics objects, with one handle per object.

Remarks

If one or more of `X1`, `Y1`, `Z1` is a vector, the vectors are plotted versus the rows or columns of the matrix, depending whether the vectors' lengths equal the number of rows or the number of columns.

You can mix `Xn,Yn,Zn` triples with `Xn,Yn,Zn,`*`LineSpec`* quads, for example,

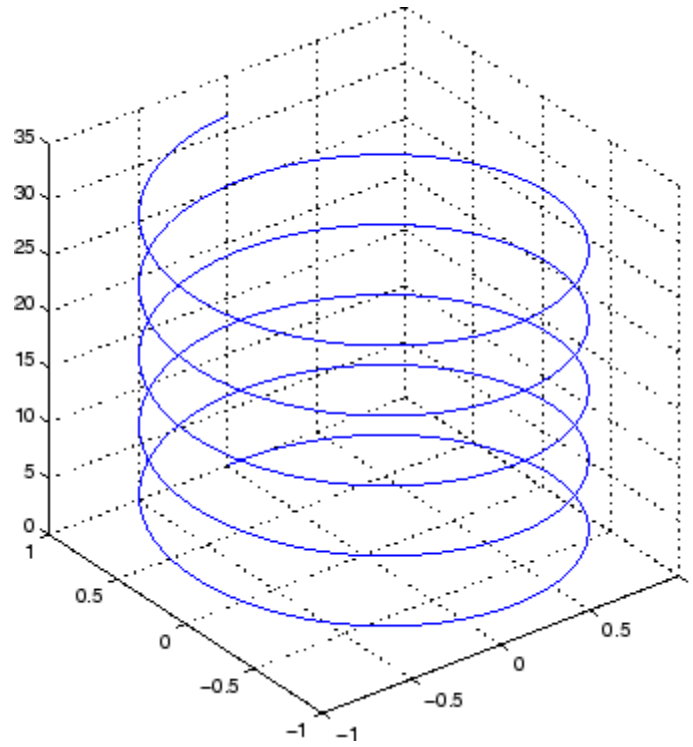
```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

See `LineSpec` and `plot` for information on line types and markers.

Examples

Plot a three-dimensional helix.

```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
grid on
axis square
```

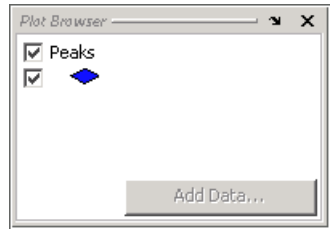


See Also

axis, bar3, grid, line, LineSpec, loglog, plot, semilogx, semilogy, subplot

Purpose

Show or hide figure plot browser



GUI Alternatives

Click the larger **Plotting Tools** icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Open or close the **Plot Browser** tool from the figure's **View** menu. For details, see "The Plot Browser" in the MATLAB Graphics documentation.

Syntax

```
plotbrowser('on')
plotbrowser('off')
plotbrowser('toggle')
plotbrowser
plotbrowser(figure_handle,...)
```

Description

`plotbrowser('on')` displays the Plot Browser on the current figure.

`plotbrowser('off')` hides the Plot Browser on the current figure.

`plotbrowser('toggle')` or `plotbrowser` toggles the visibility of the Plot Browser on the current figure.

`plotbrowser(figure_handle,...)` shows or hides the Plot Browser on the figure specified by `figure_handle`.

See Also

`plottools`, `figurepalette`, `propertyeditor`

Purpose

Interactively edit and annotate plots

Syntax

```
plotedit on
plotedit off
plotedit
plotedit(h)
plotedit('state')
plotedit(h,'state')
```

Description

`plotedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, change line styles, and add text, line, and arrow annotations.

`plotedit off` ends plot mode for the current figure.

`plotedit` toggles the plot edit mode for the current figure.

`plotedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plotedit('state')` specifies the `plotedit` state for the current figure. Values for `state` can be as shown.

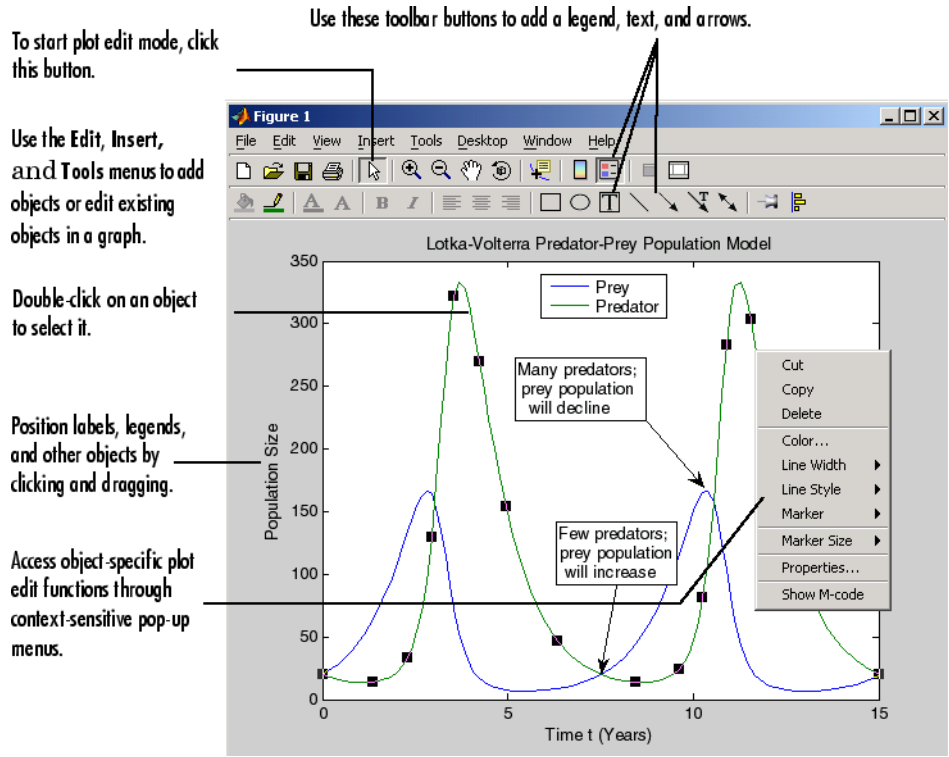
Value for *state*	Description
`on`	Starts plot edit mode
`off`	Ends plot edit mode
`showtoolsmenu`	Displays the **Tools** menu in the menu bar
`hidetoolsmenu`	Removes the **Tools** menu from the menu bar

Note `hidetoolsmenu` is intended for GUI developers who do not want the **Tools** menu to appear in applications that use the figure window.

`plotedit(h,'state')` specifies the `plotedit` `state` for figure handle `h`.

Remarks

Plot Editing Mode Graphical Interface Components



Examples

Start plot edit mode for figure 2.

```
plotedit(2)
```

End plot edit mode for figure 2.

```
plotedit(2, 'off')
```

Hide the **Tools** menu for the current figure:

```
plotedit('hidetoolsmenu')
```

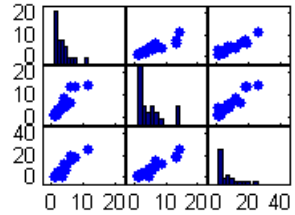
See Also

axes, line, open, plot, print, saveas, text, propedit

plotmatrix

Purpose

Scatter plot matrix



Syntax

```
plotmatrix(X,Y)
plotmatrix(...,'LineSpec')
[H,AX,BigAx,P] = plotmatrix(...)
```

Description

`plotmatrix(X,Y)` scatter plots the columns of `X` against the columns of `Y`. If `X` is *p*-by-*m* and `Y` is *p*-by-*n*, `plotmatrix` produces an *n*-by-*m* matrix of axes. `plotmatrix(Y)` is the same as `plotmatrix(Y,Y)` except that the diagonal is replaced by `hist(Y(:,i))`.

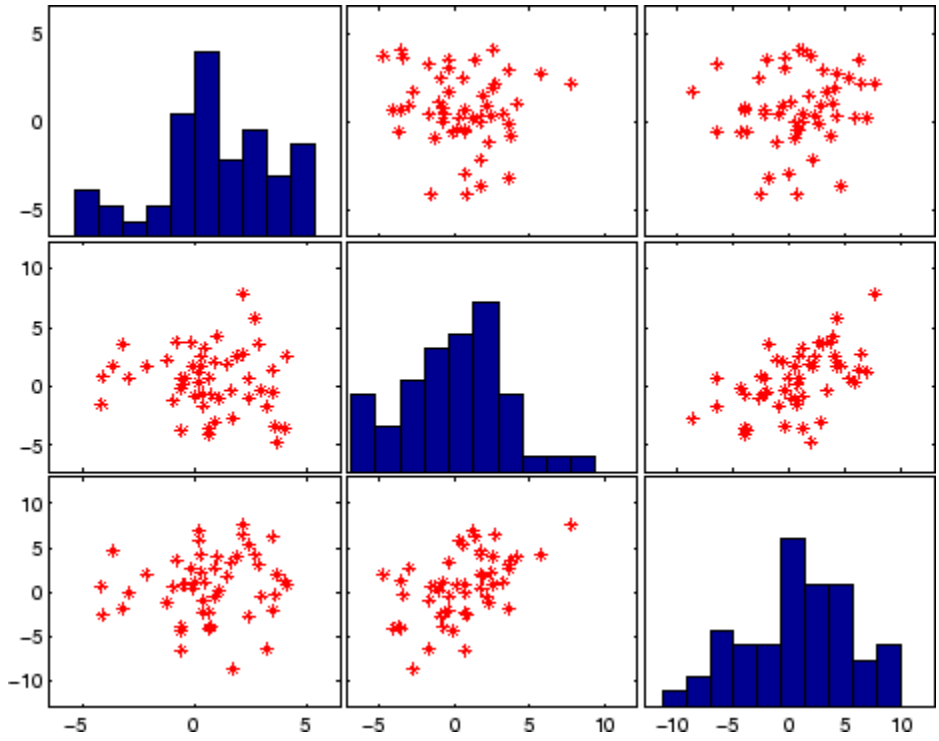
`plotmatrix(...,'LineSpec')` uses a `LineSpec` to create the scatter plot. The default is `'.'`.

`[H,AX,BigAx,P] = plotmatrix(...)` returns a matrix of handles to the objects created in `H`, a matrix of handles to the individual subaxes in `AX`, a handle to a big (invisible) axes that frames the subaxes in `BigAx`, and a matrix of handles for the histogram plots in `P`. `BigAx` is left as the current axes so that a subsequent `title`, `xlabel`, or `ylabel` command is centered with respect to the matrix of axes.

Examples

Generate plots of random data.

```
x = randn(50,3); y = x*[-1 2 1;2 0 1;1 -2 3;]';
plotmatrix(y,'*r')
```

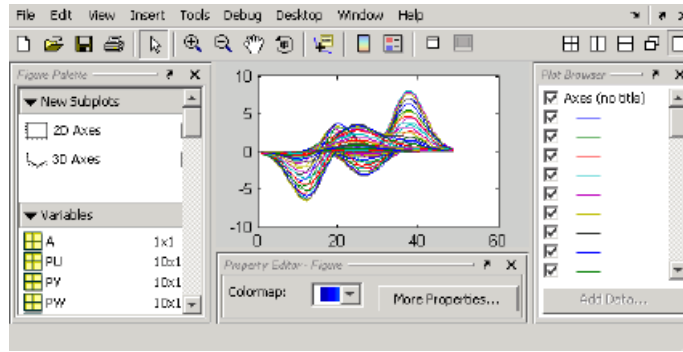


See Also

`scatter`, `scatter3`

Purpose

Show or hide plot tools



GUI Alternatives

Click the larger **Plotting Tools** icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Individually select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation.

Syntax

```
plottools('on')
plottools('off')
plottools
plottools(figure_handle,...)
plottools(...,'tool')
```

Description

`plottools('on')` displays the Figure Palette, Plot Browser, and Property Editor on the current figure, configured as you last used them.

`plottools('off')` hides the Figure Palette, Plot Browser, and Property Editor on the current figure.

`plottools` with no arguments, is the same as `plottools('on')`

`plottools(figure_handle,...)` displays or hides the plot tools on the specified figure instead of on the current figure.

`plottools(...,'tool')` operates on the specified tool only. *`tool`* can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

Note The first time you open the plotting tools, all three of them appear, grouped around the current figure as shown above. If you close, move, or undock any of the tools, MATLAB remembers the configuration you left them in and restores it when you invoke the tools for subsequent figures, both within and across MATLAB sessions.

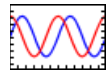
See Also

`figurepalette`, `plotbrowser`, `propertyeditor`

plotyy

Purpose

2-D line plots with y-axes on both left and right side



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation and "Creating Plots from the Workspace Browser" in the MATLAB Desktop Tools documentation.

Syntax

```
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,function)
plotyy(X1,Y1,X2,Y2,'function1','function2')
[AX,H1,H2] = plotyy(...)
```

Description

`plotyy(X1,Y1,X2,Y2)` plots `X1` versus `Y1` with *y*-axis labeling on the left and plots `X2` versus `Y2` with *y*-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,function)` uses the specified plotting function to produce the graph.

`function` can be either a function handle or a string specifying `plot`, `semilogx`, `semilogy`, `loglog`, `stem`, or any MATLAB function that accepts the syntax

```
h = function(x,y)
```

For example,

```
plotyy(x1,y1,x2,y2,@loglog) % function handle
plotyy(x1,y1,x2,y2,'loglog') % string
```

Function handles enable you to access user-defined subfunctions and can provide other advantages. See @ for more information on using function handles.

`plotyy(X1,Y1,X2,Y2,'function1','function2')` uses `function1(X1,Y1)` to plot the data for the left axis and `function2(X2,Y2)` to plot the data for the right axis.

`[AX,H1,H2] = plotyy(...)` returns the handles of the two axes created in `AX` and the handles of the graphics objects from each plot in `H1` and `H2`. `AX(1)` is the left axes and `AX(2)` is the right axes.

Examples

This example graphs two mathematical functions using `plot` as the plotting function. The two *y*-axes enable you to display both sets of data on one graph even though relative values of the data are quite different.

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
[AX,H1,H2] = plotyy(x,y1,x,y2,'plot');
```

You can use the handles returned by `plotyy` to label the axes and set the line styles used for plotting. With the axes handles you can specify the `YLabel` properties of the left- and right-side *y*-axis:

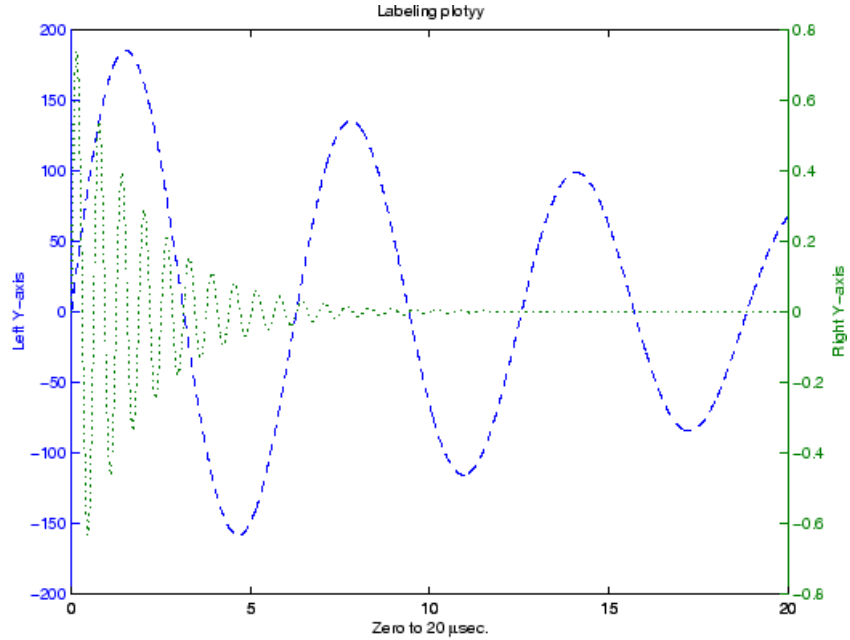
```
set(get(AX(1),'Ylabel'),'String','Slow Decay')
set(get(AX(2),'Ylabel'),'String','Fast Decay')
```

Use the `xlabel` and `title` commands to label the *x*-axis and add a title:

```
xlabel('Time (\musec)')
title('Multiple Decay Rates')
```

Use the line handles to set the `LineStyle` properties of the left- and right-side plots:

```
set(H1,'LineStyle','--')
set(H2,'LineStyle',':')
```



See Also

`plot`, `loglog`, `semilogx`, `semilogy`, axes properties `XAxisLocation`, `YAxisLocation`

See “Using Multiple X- and Y-Axes” for more information.

Purpose

Transform polar or cylindrical coordinates to Cartesian

Syntax

```
[X,Y] = pol2cart(THETA,RHO)
[X,Y,Z] = pol2cart(THETA,RHO,Z)
```

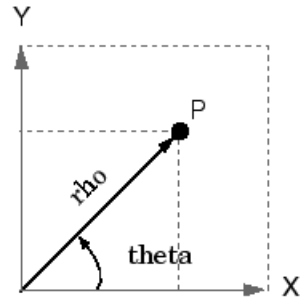
Description

`[X,Y] = pol2cart(THETA,RHO)` transforms the polar coordinate data stored in corresponding elements of `THETA` and `RHO` to two-dimensional Cartesian, or *xy*, coordinates. The arrays `THETA` and `RHO` must be the same size (or either can be scalar). The values in `THETA` must be in radians.

xyz, `[X,Y,Z] = pol2cart(THETA,RHO,Z)` transforms the cylindrical coordinate data stored in corresponding elements of `THETA`, `RHO`, and `Z` to three-dimensional Cartesian, or coordinates. The arrays `THETA`, `RHO`, and `Z` must be the same size (or any can be scalar). The values in `THETA` must be in radians.

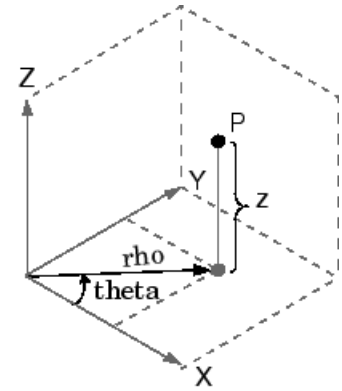
Algorithm

The mapping from polar and cylindrical coordinates to Cartesian coordinates is:



Polar to Cartesian Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
```



Cylindrical to Cartesian Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
```

See Also

cart2pol, cart2sph, sph2cart

Purpose

Polar coordinate plot

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axes_handle,...)
h = polar(...)
```

Description

The `polar` function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

`polar(theta,rho)` creates a polar coordinate plot of the angle `theta` versus the radius `rho`. `theta` is the angle from the *x*-axis to the radius vector specified in radians; `rho` is the length of the radius vector specified in dataspace units.

`polar(theta,rho,LineSpec)` `LineSpec` specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

`polar(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = polar(...)` returns the handle of a line object in `h`.

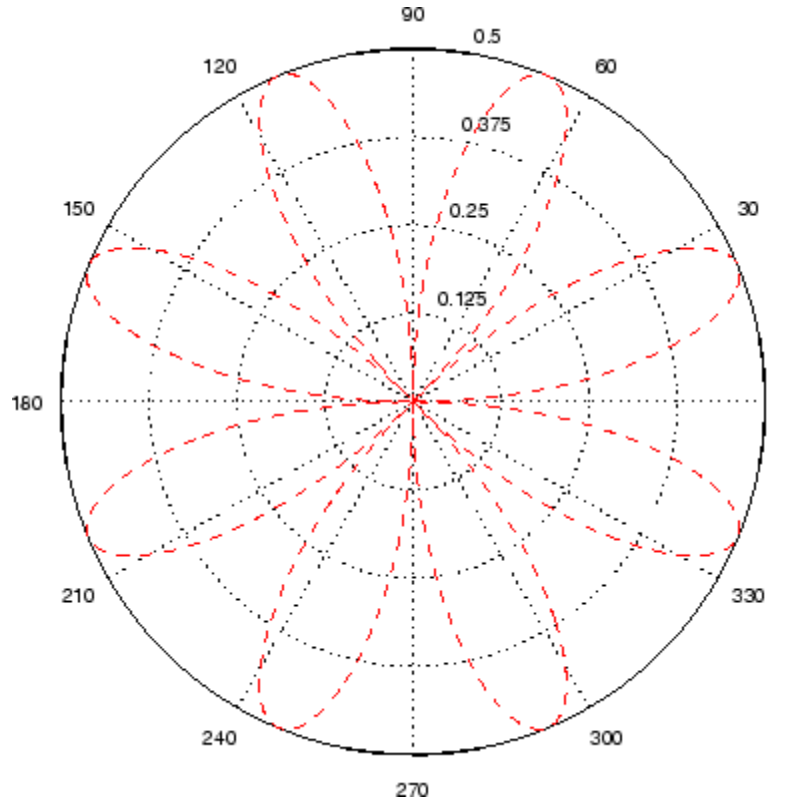
Remarks

Negative `r` values reflect through the origin, rotating by `pi` (since `(theta,r)` transforms to `(r*cos(theta), r*sin(theta))`). If you want different behavior, you can manipulate `r` prior to plotting. For example, you can make `r` equal to `max(0,r)` or `abs(r)`.

Examples

Create a simple polar plot using a dashed red line:

```
t = 0:.01:2*pi;
polar(t,sin(2*t).*cos(2*t),'--r')
```



See Also

`cart2pol`, `compass`, `LineSpec`, `plot`, `pol2cart`, `rose`

Purpose

Polynomial with specified roots

Syntax

```
p = poly(A)
p = poly(r)
```

Description

`p = poly(A)` where `A` is an `n`-by-`n` matrix returns an `n+1` element row vector whose elements are the coefficients of the characteristic polynomial, $\det(sI - A)$. The coefficients are ordered in descending powers: if a vector `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$

`p = poly(r)` where `r` is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of `r`.

Remarks

Note the relationship of this command to

```
r = roots(p)
```

which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector `p`. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

```
A =

     1     2     3
     4     5     6
     7     8     0
```

is returned in a row vector by `poly`:

```
p = poly(A)

p =
```

```
     1    -6    -72    -27
```

The roots of this polynomial (eigenvalues of matrix `A`) are returned in a column vector by `roots`:

```
r = roots(p)

r =

    12.1229
    -5.7345
    -0.3884
```

Algorithm

The algorithms employed for `poly` and `roots` illustrate an interesting aspect of the modern approach to eigenvalue computation. `poly(A)` generates the characteristic polynomial of `A`, and `roots(poly(A))` finds the roots of that polynomial, which are the eigenvalues of `A`. But both `poly` and `roots` use `eig`, which is based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If `A` is an `n`-by-`n` matrix, `poly(A)` produces the coefficients `c(1)` through `c(n+1)`, with `c(1) = 1`, in

$$det(\lambda I - A) = c_1\lambda^n + \ldots + c_n\lambda + c_{n+1}$$

The algorithm is

```
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
for j = 1:n
    c(2:j+1) = c(2:j+1)-z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product.

$$(\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n)$$

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of `A`. This is true even if the eigenvalues of `A` are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.

See Also

`conv`, `polyval`, `residue`, `roots`

polyarea

Purpose

Area of polygon

Syntax

```
A = polyarea(X,Y)
A = polyarea(X,Y,dim)
```

Description

`A = polyarea(X,Y)` returns the area of the polygon specified by the vertices in the vectors `X` and `Y`.

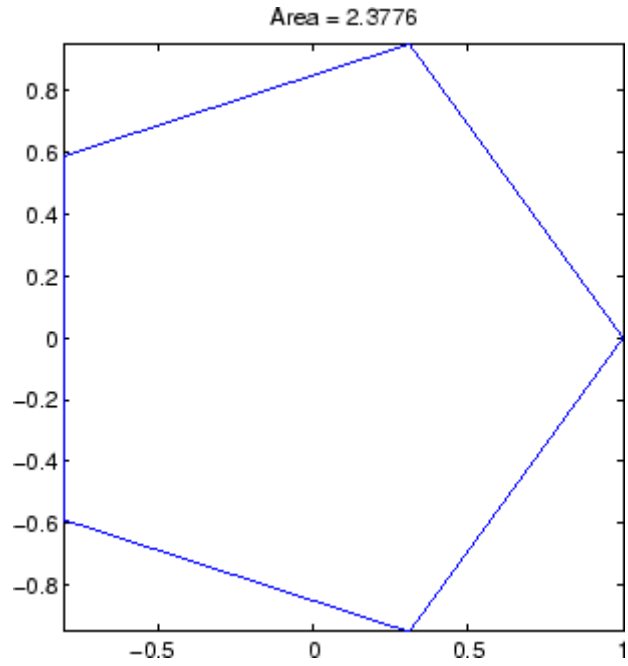
If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

If `X` and `Y` are multidimensional arrays, `polyarea` returns the area of the polygons in the first nonsingleton dimension of `X` and `Y`.

`A = polyarea(X,Y,dim)` operates along the dimension specified by scalar `dim`.

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
A = polyarea(xv,yv);
plot(xv,yv); title(['Area = ' num2str(A)]); axis image
```



See Also

`convhull`, `inpolygon`, `rectint`

polyder

Purpose

Polynomial derivative

Syntax

```
k = polyder(p)
k = polyder(a,b)
[q,d] = polyder(b,a)
```

Description

The `polyder` function calculates the derivative of polynomials, polynomial products, and polynomial quotients. The operands `a`, `b`, and `p` are vectors whose elements are the coefficients of a polynomial in descending powers.

`k = polyder(p)` returns the derivative of the polynomial `p`.

`k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`.

`[q,d] = polyder(b,a)` returns the numerator `q` and denominator `d` of the derivative of the polynomial quotient `b/a`.

Examples

The derivative of the product

$$(3x^2 + 6x + 9)(x^2 + 2x)$$

is obtained with

```
a = [3 6 9];
b = [1 2 0];
k = polyder(a,b)
k =
    12    36    42    18
```

This result represents the polynomial

$$12x^3 + 36x^2 + 42x + 18$$

See Also

`conv`, `deconv`

Purpose

Polynomial eigenvalue problem

Syntax

```
[X,e] = polyeig(A0,A1,...Ap)
e = polyeig(A0,A1,..,Ap)
[X, e, s] = polyeig(A0,A1,..,AP)
```

Description

`[X,e] = polyeig(A0,A1,...Ap)` solves the polynomial eigenvalue problem of degree `p`

$$(A_0 + \lambda A_1 + \ldots + \lambda^P A_p)x = 0$$

where polynomial degree `p` is a non-negative integer, and `A0,A1,...Ap` are input matrices of order `n`. The output consists of a matrix `X` of size `n`-by-`n*p` whose columns are the eigenvectors, and a vector `e` of length `n*p` containing the eigenvalues.

If `lambda` is the `j`th eigenvalue in `e`, and `x` is the `j`th column of eigenvectors in `X`, then `(A0 + lambda*A1 + ... + lambda^p*Ap)*x` is approximately `0`.

`e = polyeig(A0,A1,..,Ap)` is a vector of length `n*p` whose elements are the eigenvalues of the polynomial eigenvalue problem.

`[X, e, s] = polyeig(A0,A1,..,AP)` also returns a vector `s` of length `p*n` containing condition numbers for the eigenvalues. At least one of `A0` and `AP` must be nonsingular. Large condition numbers imply that the problem is close to a problem with multiple eigenvalues.

Remarks

Based on the values of `p` and `n`, `polyeig` handles several special cases:

- `p = 0`, or `polyeig(A)` is the standard eigenvalue problem: `eig(A)`.
- `p = 1`, or `polyeig(A,B)` is the generalized eigenvalue problem: `eig(A,-B)`.
- `n = 1`, or `polyeig(a0,a1,...ap)` for scalars `a0`, `a1` ..., `ap` is the standard polynomial problem: `roots([ap ... a1 a0])`.

If both A0 and Ap are singular the problem is potentially ill-posed. Theoretically, the solutions might not exist or might not be unique. Computationally, the computed solutions might be inaccurate. If one, but not both, of A0 and Ap is singular, the problem is well posed, but some of the eigenvalues might be zero or infinite.

Note that scaling A0,A1,..,Ap to have norm(Ai) roughly equal 1 may increase the accuracy of polyeig. In general, however, this cannot be achieved. (See Tisseur [3] for more detail.)

Algorithm

The polyeig function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of eig and qz for more on this.

See Also

condeig, eig, qz

References

[1] Dedieu, Jean-Pierre Dedieu and Francoise Tisseur, “Perturbation theory for homogeneous polynomial eigenvalue problems,” *Linear Algebra Appl.*, Vol. 358, pp. 71-94, 2003.

[2] Tisseur, Francoise and Karl Meerbergen, “The quadratic eigenvalue problem,” *SIAM Rev.*, Vol. 43, Number 2, pp. 235-286, 2001.

[3] Francoise Tisseur, “Backward error and condition of polynomial eigenvalue problems” *Linear Algebra Appl.*, Vol. 309, pp. 339-361, 2000.

Purpose

Polynomial curve fitting

Syntax

```
p = polyfit(x,y,n)
[p,S] = polyfit(x,y,n)
[p,S,mu] = polyfit(x,y,n)
```

Description

`p = polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the data, `p(x(i))` to `y(i)`, in a least squares sense. The result `p` is a row vector of length `n+1` containing the polynomial coefficients in descending powers

$$p(x) = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_{n+1}$$

`[p,S] = polyfit(x,y,n)` returns the polynomial coefficients `p` and a structure `S` for use with `polyval` to obtain error estimates or predictions. Structure `S` contains fields `R`, `df`, and `normr`, for the triangular factor from a QR decomposition of the Vandermonde matrix of `X`, the degrees of freedom, and the norm of the residuals, respectively. If the data `Y` are random, an estimate of the covariance matrix of `P` is `(Rinv*Rinv')*normr^2/df`, where `Rinv` is the inverse of `R`. If the errors in the data `y` are independent normal with constant variance, `polyval` produces error bounds that contain at least 50% of the predictions.

`[p,S,mu] = polyfit(x,y,n)` finds the coefficients of a polynomial in

$$\hat{x} = \frac{x - \mu_1}{\mu_2}$$

where $\mu_1 = \mathrm{mean}(x)$ and $\mu_2 = \mathrm{std}(x)$. `mu` is the two-element vector $[\mu_1, \mu_2]$. This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

Examples

This example involves fitting the error function, `erf(x)`, by a polynomial in `x`. This is a risky project because `erf(x)` is a bounded function, while polynomials are unbounded, so the fit might not be very good.

First generate a vector of x points, equally spaced in the interval $[0, 2.5]$; then evaluate `erf(x)` at those points.

```
x = (0: 0.1: 2.5)';
y = erf(x);
```

The coefficients in the approximating polynomial of degree 6 are

```
p = polyfit(x,y,6)

p =

  0.0084   -0.0983    0.4217   -0.7435   0.1471    1.1064   0.0004
```

There are seven coefficients and the polynomial is

$$0.0084x^6 - 0.0983x^5 + 0.4217x^4 - 0.7435x^3 + 0.1471x^2 + 1.1064x + 0.0004$$

To see how good the fit is, evaluate the polynomial at the data points with

```
f = polyval(p,x);
```

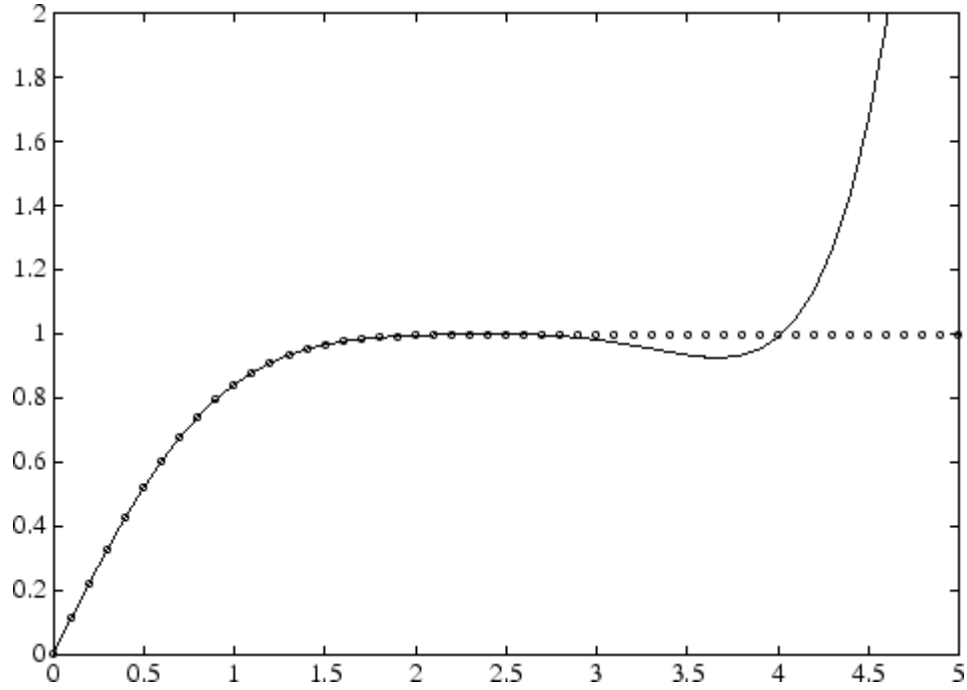
A table showing the data, fit, and error is

```
table = [x y f y-f]

table =

   0          0          0.0004    -0.0004
   0.1000     0.1125     0.1119     0.0006
   0.2000     0.2227     0.2223     0.0004
   0.3000     0.3286     0.3287    -0.0001
   0.4000     0.4284     0.4288    -0.0004
   ...
   2.1000     0.9970     0.9969     0.0001
   2.2000     0.9981     0.9982    -0.0001
   2.3000     0.9989     0.9991    -0.0003
   2.4000     0.9993     0.9995    -0.0002
```

```
    2.5000    0.9996    0.9994    0.0002
```

So, on this interval, the fit is good to between three and four digits. Beyond this interval the graph shows that the polynomial behavior takes over and the approximation quickly deteriorates.

```
x = (0: 0.1: 5)';
y = erf(x);
f = polyval(p,x);
plot(x,y,'o',x,f,'-')
axis([0  5  0  2])
```



Algorithm

The `polyfit` M-file forms the Vandermonde matrix, V, whose elements are powers of x.

$$v_{i,j} = x_i^{n-j}$$

It then uses the backslash operator, `\`, to solve the least squares problem

$$Vp \cong y$$

You can modify the M-file to use other functions of x as the basis functions.

See Also

`poly`, `polyval`, `roots`

Purpose

Integrate polynomial analytically

Syntax

```
polyint(p,k)
polyint(p)
```

Description

`polyint(p,k)` returns a polynomial representing the integral of polynomial `p`, using a scalar constant of integration `k`.

`polyint(p)` assumes a constant of integration `k=0`.

See Also

`polyder`, `polyval`, `polyvalm`, `polyfit`

polyval

Purpose

Polynomial evaluation

Syntax

```
y = polyval(p,x)
y = polyval(p,x,[],mu)
[y,delta] = polyval(p,x,S)
[y,delta] = polyval(p,x,S,mu)
```

Description

`y = polyval(p,x)` returns the value of a polynomial of degree `n` evaluated at `x`. The input argument `p` is a vector of length `n+1` whose elements are the coefficients in descending powers of the polynomial to be evaluated.

$$y = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_{n+1}$$

`x` can be a matrix or a vector. In either case, `polyval` evaluates `p` at each element of `x`.

`y = polyval(p,x,[],mu)` uses $\hat{x} = (x - \mu_1)/\mu_2$ in place of x. In this equation, $\mu_1 = \text{mean}(x)$ and $\mu_2 = \text{std}(x)$. The centering and scaling parameters `mu` = $[\mu_1, \mu_2]$ are optional output computed by `polyfit`.

`[y,delta] = polyval(p,x,S)` and `[y,delta] = polyval(p,x,S,mu)` use the optional output structure `S` generated by `polyfit` to generate error estimates, `y±delta`. If the errors in the data input to `polyfit` are independent normal with constant variance, `y±delta` contains at least 50% of the predictions.

Remarks

The `polyvalm(p,x)` function, with `x` a matrix, evaluates the polynomial in a matrix sense. See `polyvalm` for more information.

Examples

The polynomial $p(x) = 3x^2 + 2x + 1$ is evaluated at x = 5, 7, and 9 with

```
p = [3 2 1];
polyval(p,[5 7 9])
```

which results in

```
ans =

    86   162   262
```

For another example, see `polyfit`.

See Also

`polyfit`, `polyvalm`

polyvalm

Purpose

Matrix polynomial evaluation

Syntax

```
Y = polyvalm(p,X)
```

Description

`Y = polyvalm(p,X)` evaluates a polynomial in a matrix sense. This is the same as substituting matrix `X` in the polynomial `p`.

Polynomial `p` is a vector whose elements are the coefficients of a polynomial in descending powers, and `X` must be a square matrix.

Examples

The Pascal matrices are formed from Pascal's triangle of binomial coefficients. Here is the Pascal matrix of order 4.

```
X = pascal(4)
X =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

Its characteristic polynomial can be generated with the `poly` function.

```
p = poly(X)
p =
     1   -29    72   -29     1
```

This represents the polynomial $x^4 - 29x^3 + 72x^2 - 29x + 1$.

Pascal matrices have the curious property that the vector of coefficients of the characteristic polynomial is palindromic; it is the same forward and backward.

Evaluating this polynomial at each element is not very interesting.

```
polyval(p,X)
ans =
    16      16      16      16
    16      15    -140    -563
    16    -140   -2549  -12089
```

```
    16    -563   -12089   -43779
```

But evaluating it in a matrix sense is interesting.

```
polyvalm(p,X)
ans =
   0    0    0    0
   0    0    0    0
   0    0    0    0
   0    0    0    0
```

The result is the zero matrix. This is an instance of the Cayley-Hamilton theorem: a matrix satisfies its own characteristic equation.

See Also

`polyfit`, `polyval`

pow2

Purpose

Base 2 power and scale floating-point numbers

Syntax

```
X = pow2(Y)
X = pow2(F,E)
```

Description

`X = pow2(Y)` returns an array `X` whose elements are 2 raised to the power `Y`.

`X = pow2(F,E)` computes $x = f * 2^e$ for corresponding elements of `F` and `E`. The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. Arguments `F` and `E` are real and integer arrays, respectively.

Remarks

This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

Examples

For IEEE arithmetic, the statement `X = pow2(F,E)` yields the values:

```
F          E         X
1/2        1         1
pi/4       2         pi
-3/4       2         -3
1/2        -51       eps
1-eps/2    1024      realmax
1/2        -1021     realmin
```

See Also

`log2`, `exp`, `hex2num`, `realmax`, `realmin`

The arithmetic operators `^` and `.^`

Purpose

Array power

Syntax

```
Z = X.^Y
```

Description

`Z = X.^Y` denotes element-by-element powers. `X` and `Y` must have the same dimensions unless one is a scalar. A scalar is expanded to an array of the same size as the other input.

`C = power(A,B)` is called for the syntax `'A .^ B'` when `A` or `B` is an object.

Note that for a negative value `X` and a non-integer value `Y`, if the `abs(Y)` is less than one, the `power` function returns the complex roots. To obtain the remaining real roots, use the `nthroot` function.

See Also

`nthroot`, `realpow`

Purpose

Evaluate piecewise polynomial

Syntax

```
v = ppval(pp,xx)
```

Description

`v = ppval(pp,xx)` returns the value of the piecewise polynomial *f*, contained in `pp`, at the entries of `xx`. You can construct `pp` using the functions `interp1`, `pchip`, `spline`, or the spline utility `mkpp`.

`v` is obtained by replacing each entry of `xx` by the value of *f* there. If *f* is scalar-valued, `v` is of the same size as `xx`. `xx` may be N-dimensional.

If `pp` was constructed by `pchip`, `spline`, or `mkpp` using the orientation of non-scalar function values specified for those functions, then:

If `f` is `[D1,..,Dr]`-valued, and `xx` is a vector of length `N`, then `V` has size `[D1,...,Dr, N]`, with `V(:,...,:,J)` the value of `f` at `xx(J)`.

If `f` is `[D1,..,Dr]`-valued, and `xx` has size `[N1,...,Ns]`, then `V` has size `[D1,...,Dr, N1,...,Ns]`, with `V(:,...,:, J1,...,Js)` the value of `f` at `xx(J1,...,Js)`.

If `pp` was constructed by `interp1` using the orienatation of non-scalar function values specified for that function, then:

If `f` is `[D1,..,Dr]`-valued, and `xx` is a vector of length `N`, then `V` has size `[N,D1,...,Dr]`, with `V(J,:,...,:)` the value of `f` at `xx(J)`.

If `f` is `[D1,..,Dr]`-valued, and `xx` has size `[N1,...,Ns]`, then `V` has size `[N1,...,Ns,D1,...,Dr]`, with `V(J1,...,Js,:,...,:)` the value of `f` at `xx(J1,...,Js)`.

Examples

Compare the results of integrating the function `cos`

```
a = 0; b = 10;
int1 = quad(@cos,a,b)

int1 =
   -0.5440
```

with the results of integrating the piecewise polynomial `pp` that approximates the cosine function by interpolating the computed values `x` and `y`.

```
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = quad(@(x)ppval(pp,x),a,b)

int2 =
   -0.5485
```

`int1` provides the integral of the cosine function over the interval `[a,b]`, while `int2` provides the integral over the same interval of the piecewise polynomial `pp`.

See Also

`mkpp`, `spline`, `unmkpp`

prefdir

Purpose

Directory containing preferences, history, and layout files

Syntax

```
prefdir
d = prefdir
d = prefdir(1)
```

Description

`prefdir` returns the directory that contains

- Preferences for MATLAB and related products (`matlab.prf`)
- Command history file (`history.m`)
- MATLAB shortcuts (`shortcuts.xml`)
- MATLAB desktop layout files (`MATLABDesktop.xml` and `Your_Saved_LayoutMATLABLayout.xml`)
- Other related files

The directory might be in a hidden folder, for example, `myname/.matlab/R2007b`. How to access hidden folders depends on your platform:

- On Windows, in any folder window, select **Tools > Folder Options**. Click the **View** tab, and under **Advanced** settings, select **Show hidden files and folders**. Then you should be able to see the folder returned by `prefdir`.
- On Macintosh platforms, in the Finder, select **Go -> Go to Folder**. In the resulting dialog box, type the path returned by `prefdir` and press **Enter**.

`d = prefdir` assigns to `d` the name of the directory containing preferences and related files.

`d = prefdir(1)` creates a directory for preferences and related files if one does not exist. If the directory does exist, the name is assigned to `d`.

Remarks

The preferences directory MATLAB uses depends on the release. The preference directory naming and preference migration practice used from R13 through R14SP2 was changed starting in R14SP3 to address backwards compatibility problems. The differences are relevant primarily if you run multiple versions of MATLAB, and especially if one version is prior to R14SP3:

- For R2007b back through and including R2006a, and R14SP3, MATLAB uses the name of the release for the preference directory. For example, it uses `R2007b`, `R2007a`, ... through `R14SP3`. When you install R2007b, MATLAB migrates the files in the `R2007a` preferences directory to the `R2007b` preferences directory. While running R2007b through R14SP3, any changes made to files in those preferences directories (`R2007b` through `R14SP3`) are used only in their respective versions. As an example, commands you run in R2007b will *not* appear in the Command History when you run R2007a, and so on. The converse is also true.
- The R14 through R14SP2 releases all share the `R14` preferences directory. While running R14SP1, for example, any changes made to files in the preferences directory, `R14`, are used when you run R14SP2 and R14. As another example, commands you run in R14 appear in the Command History when you run R14SP2, and the converse is also true. The preferences are not used when you run R14SP3 or later versions because those versions each use their own preferences directories.
- All R13 releases use the `R13` preferences directory. While running R13SP1, for example, any changes made to files in the preferences directory, `R13`, are used when you run R13. As an example, commands you run in R13 will appear in the Command History when you run R13SP1, and the converse is true. The preferences are not used when you run any R14 or later releases because R14 and later releases use different preferences directories, and the converse is true.
- Upon startup, MATLAB 7.5 (R2007b) looks for and if found, uses the `R2007b` preferences directory. If not found, MATLAB creates an `R2007b` preferences directory. This happens when the R2007b

preferences directory is deleted. MATLAB then looks for the `R2007a` preferences directory, and if found, migrates the R2007a preferences to the R2007b preferences. If it does not find the `R2007a` preferences directory, it uses the default preferences for R2007b. This process also applies when starting MATLAB 7.3 through 7.1.

- If you want to use default preferences for R2007b, and do not want MATLAB to migrate preferences from R2007a, the `R2007b` preferences directory *must exist but be empty* when you start MATLAB. If you want to maintain some of your R2007b preferences, but restore the defaults for others, in the `R2007b` preferences directory, delete the files for which you want the defaults to be restored. One file you might want to maintain is `history.m`—for more information about the file, see "Viewing Statements in the Command History Window" in the MATLAB Desktop Tools and Development Environment documentation.

Examples

Run

```
prefdir
```

MATLAB returns

```
ans =

C:\WINNT\Profiles\my_user_name\MATHWORKS\Application Data\MathWorks\MATLAB\R2007b
```

Running `dir` for the directory shows these files and others for MathWorks products:

```
.                 history.m
..                matlab.prf
cwdhistory.m      MATLABDesktop.xml
shortcuts.xml     MATLAB EditorDesktop.xml
```

In MATLAB, run `cd(prefdir)` to change to that directory.

On Windows platforms, go directly to the preferences directory in Explorer by running `winopen(prefdir)`.

See Also

`preferences`, `winopen`

Fonts, Colors, and Other Preferences in the MATLAB Desktop Tools and Development Environment documentation

preferences

Purpose

Open Preferences dialog box for MATLAB and related products

GUI Alternatives

As an alternative to the `preferences` function, select **File > Preferences** in the MATLAB desktop or any desktop tool.

Syntax

```
preferences
```

Description

`preferences` displays the Preferences dialog box, from which you can make changes to options for MATLAB and related products.

See Also

`prefdir`

Fonts, Colors, and Other Preferences in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Generate list of prime numbers

Syntax

```
p = primes(n)
```

Description

`p = primes(n)` returns a row vector of the prime numbers less than or equal to `n`. A prime number is one that has no factors other than `1` and itself.

Examples

```
p = primes(37)

p = 2  3  5  7  11  13  17  19  23  29  31  37
```

See Also

`factor`

print, printopt

Purpose

Print figure or save to file and configure printer defaults

Contents

GUI Alternative

Use **File ⇒ Print** on the figure window menu to access the Print dialog and **File ⇒ Print Preview** to access the Print Preview GUI. For details, see How to Print or Export in the MATLAB Graphics documentation.

Syntax

```
print
print filename
print -ddriver
print -dformat
print -dformat filename
print -smodelname
print -options
print(...)
[pcmd,dev] = printopt
```

Description

`print` and `printopt` produce hardcopy output. All arguments to the `print` command are optional. You can use them in any combination or order.

`print` sends the contents of the current figure, including bitmap representations of any user interface controls, to the printer using the device and system printing command defined by `printopt`.

`print filename` directs the output to the PostScript file designated by `filename`. If `filename` does not include an extension, `print` appends an appropriate extension.

`print -d`*`driver`* prints the figure using the specified printer *`driver`*, (such as color PostScript). If you omit `-d`*`driver`*, `print` uses the default value stored in `printopt.m`. The "Printer Drivers" on page 2-2542 table lists all supported device types.

`print -d`*`format`* copies the figure to the system clipboard (Windows only). To be valid, the *`format`* for this operation must be either `-dmeta` (Windows Enhanced Metafile) or `-dbitmap` (Windows Bitmap).

`print -d`*`format`* `filename` exports the figure to the specified file using the specified graphics *`format`*, (such as TIFF). The table of "Graphics Format Files" on page 2-2546 lists all supported graphics file formats.

`print -s`*`modelname`* prints the current Simulink model *`modelname`*.

`print -`*`options`* specifies print options that modify the action of the `print` command. (For example, the `-noui` option suppresses printing of user interface controls.) The Options section lists available options.

`print(...)` is the function form of `print`. It enables you to pass variables for any input arguments. This form is useful for passing filenames and handles. See Batch Processing for an example.

`[pcmd,dev] = printopt` returns strings containing the current system-dependent printing command and output device. `printopt` is an M-file used by `print` to produce the hardcopy output. You can edit the M-file `printopt.m` to set your default printer type and destination.

`pcmd` and `dev` are platform-dependent strings. `pcmd` contains the command that `print` uses to send a file to the printer. `dev` contains the printer driver or graphics format option for the `print` command. Their defaults are platform dependent.

Platform	System Printing Command	Driver or Format
MAC and UNIX	`lpr -r`	`dps2`
Windows	COPY /B %s LPT1:	`dwin`

Printer Drivers

The table below shows the more widely used printer drivers supported by MATLAB. If you do not specify a driver, MATLAB uses the default setting shown in the previous table. For a list of all supported printer drivers, type

```
print -d
```

at the MATLAB prompt. Some things to remember:

- As indicated in the "Description" on page 2-2540 section, the `-d` switch either specifies a printer driver or a graphics file format:
  - Specifying a printer driver without a filename or printer name (the `-P` option) sends the output formatted by the specified driver to your default printer, which may not be what you want to do.

    Note On Windows, when you use the `-P` option to identify a printer to use, if you specify any driver other than `-dwin` or `-dwinc`, MATLAB writes the output to a file with an appropriate extension but does not send it to the printer; you can then copy that file to a printer.

  - Specifying a `-dmeta` or a `-dbitmap` graphics format without a filename places the graphic on the system clipboard, if possible (Windows only).
  - Specifying any other graphics format without a filename creates a file in the current directory with a name such as `figureN.`*`fmt`*,

where `N` is 1, 2, 3, ... and *`fmt`* indicates the format type, for example `eps` or `png`.

- Several of the drivers come from a product called Ghostscript, which is shipped with MATLAB. The last column indicates when Ghostscript is used.
- Some drivers are not available on all platforms. This is noted in the first column of the table.
- If you specify a particular printer with the `-P` option and do not specify a driver, a default driver for that printer is selected, either by the operating system or by MATLAB, depending on the platform:
  - On Windows, the driver associated with this particular printing device is used
  - On MAC and UNIX, the driver specified in `printop.m` is used

See Selecting the Printer in the Graphics documentation for more information.

Printer Driver	`print` Command Option String	Ghostscript
Canon BubbleJet BJ10e	`-dbj10e`	Yes
Canon BubbleJet BJ200 color	`-dbj200`	Yes
Canon Color BubbleJet BJC-70/BJC-600/BJC-4000	`-dbjc600`	Yes
Canon Color BubbleJet BJC-800	`-dbjc800`	Yes
Epson and compatible 9- or 24-pin dot matrix print drivers	`-depson`	Yes

Printer Driver	`print` Command Option String	Ghostscript
Epson and compatible 9-pin with interleaved lines (triple resolution)	`-deps9high`	Yes
Epson LQ-2550 and compatible; color (not supported on HP-700)	`-depsonc`	Yes
Fujitsu 3400/2400/1200	`-depsonc`	Yes
HP DesignJet 650C color (not supported on Windows)	`-ddnj650c`	Yes
HP DeskJet 500	`-ddjet500`	Yes
HP DeskJet 500C (creates black and white output)	`-dcdjmono`	Yes
HP DeskJet 500C (with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows)	`-dcdjcolor`	Yes
HP DeskJet 500C/540C color (not supported on Windows)	`-dcdj500`	Yes
HP Deskjet 550C color (not supported on Windows)	`-dcdj550`	Yes
HP DeskJet and **DeskJet Plus**	`-ddeskjet`	Yes
HP LaserJet	`-dlaserjet`	Yes

Printer Driver	`print` Command Option String	Ghostscript
HP LaserJet+	`-dljetplus`	Yes
HP LaserJet IIP	`-dljet2p`	Yes
HP LaserJet III	`-dljet3`	Yes
HP LaserJet 4.5L and 5P	`-dljet4`	Yes
HP LaserJet 5 and 6	`-dpxlmono`	Yes
HP PaintJet color	`-dpaintjet`	Yes
HP PaintJet XL color	`-dpjxl`	Yes
HP PaintJet XL color	`-dpjetxl`	Yes
HP PaintJet XL300 color (not supported on Windows)	`-dpjxl300`	Yes
HPGL for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)	`-dhpgl`	No
IBM 9-pin Proprinter	`-dibmpro`	Yes
PostScript black and white	`-dps`	No
PostScript color	`-dpsc`	No
PostScript Level 2 black and white	`-dps2`	No
PostScript Level 2 color	`-dpsc2`	No
Windows color (Windows only)	`-dwinc`	No
Windows monochrome (Windows only)	`-dwin`	No

Note Generally, Level 2 PostScript files are smaller and are rendered more quickly when printing than Level 1 PostScript files. However, not all PostScript printers support Level 2, so determine the capabilities of your printer before using those drivers. Level 2 PostScript is the default for UNIX. You can change this default by editing the `printopt.m` file. Likewise, if you want color PostScript to be the default instead of black-and-white PostScript, edit the line in the `printopt.m` file that reads `dev = '-dps2';` to be `dev = '-dpsc2';`.

Graphics Format Files

To save your figure as a graphics-format file, specify a format switch and filename. To set the resolution of the output file for a built-in MATLAB format, use the `-r` switch. (For example, `-r300` sets the output resolution to 300 dots per inch.) The `-r` switch is also supported for Windows Enhanced Metafiles, JPEG, TIFF and PNG files, but is not supported for Ghostscript formats. For more information, see "Printing and Exporting without a Display" on page 2-2549.

The table below shows the supported output formats for exporting from MATLAB and the switch settings to use. In some cases, a format is available both as a MATLAB output filter and as a Ghostscript output filter. All formats except for EMF are supported on both the PC and UNIX platforms.

Graphics Format	Bitmap or Vector	`print` Command Option String	MATLAB or Ghostscript
BMP monochrome BMP	Bitmap	`-dbmpmono`	Ghostscript
BMP 24-bit BMP	Bitmap	`-dbmp16m`	Ghostscript
BMP 8-bit (256-color) BMP (this format uses a fixed colormap)	Bitmap	`-dbmp256`	Ghostscript

Graphics Format	Bitmap or Vector	`print` Command Option String	MATLAB or Ghostscript
BMP 24-bit	Bitmap	`-dbmp`	MATLAB
EMF	Vector	`-dmeta`	MATLAB
EPS black and white	Vector	`-deps`	MATLAB
EPS color	Vector	`-depsc`	MATLAB
EPS Level 2 black and white	Vector	`-deps2`	MATLAB
EPS Level 2 color	Vector	`-depsc2`	MATLAB
HDF 24-bit	Bitmap	`-dhdf`	MATLAB
ILL (Adobe Illustrator)	Vector	`-dill`	MATLAB
JPEG 24-bit	Bitmap	`-djpeg`	MATLAB
PBM (plain format) 1-bit	Bitmap	`-dpbm`	Ghostscript
PBM (raw format) 1-bit	Bitmap	`-dpbmraw`	Ghostscript
PCX 1-bit	Bitmap	`-dpcxmono`	Ghostscript
PCX 24-bit color PCX file format, three 8-bit planes	Bitmap	`-dpcx24b`	Ghostscript
PCX 8-bit newer color PCX file format (256-color)	Bitmap	`-dpcx256`	Ghostscript

Graphics Format	Bitmap or Vector	`print` Command Option String	MATLAB or Ghostscript
PCX Older color PCX file format (EGA/VGA, 16-color)	Bitmap	`-dpcx16`	Ghostscript
PDF Color PDF file format	Vector	`-dpdf`	Ghostscript
PGM Portable Graymap (plain format)	Bitmap	`-dpgm`	Ghostscript
PGM Portable Graymap (raw format)	Bitmap	`-dpgmraw`	Ghostscript
PNG 24-bit	Bitmap	`-dpng`	MATLAB
PPM Portable Pixmap (plain format)	Bitmap	`-dppm`	Ghostscript
PPM Portable Pixmap (raw format)	Bitmap	`-dppmraw`	Ghostscript
TIFF 24-bit	Bitmap	`-dtiff` or `-dtiffn`	MATLAB
TIFF preview for EPS files	Bitmap	`-tiff`	

The TIFF image format is supported on all platforms by almost all word processors for importing images. The `-dtiffn` variant writes an uncompressed TIFF. JPEG is a lossy, highly compressed format that is supported on all platforms for image processing and for inclusion into HTML documents on the World Wide Web. To create these formats, MATLAB renders the figure using the Z-buffer rendering method and the resulting bitmap is then saved to the specified file.

Printing and Exporting without a Display

On a UNIX platform (including Macintosh), where you can start MATLAB in `nodisplay` mode (`matlab -nodisplay`), you can print using most of the drivers you can use with a display and export to most of the same file formats. The PostScript and Ghostscript devices all function in `nodisplay` mode on UNIX. The graphic devices `-djpeg`, `-dpng`, `-dtiff` (compressed TIFF bitmaps) and `-tiff` (EPS with TIFF preview) work as well, but under `nodisplay` they use Ghostscript to generate output instead of using the drivers built into MATLAB. However, Ghostscript ignores the `-r` option when generating `-djpeg`, `-dpng`, `-dtiff` and `-tiff` image files. This means that you cannot vary the resolution of image files when running in `nodisplay` mode.

Naturally, the Windows-only `-dwin` and `-dwinc` output formats cannot be used on UNIX or MAC with or without a display.

The same holds true on Windows with the `-noFigureWindows` startup option. The `-dwin`, `-dwinc`, and `-dsetup` options operate as usual under `-noFigureWindows`. However, the `printpreview` GUI does not function in this mode.

The formats which you cannot generate in `nodisplay` mode on UNIX and MAC are

- bitmap (`-dbitmap`) — Windows bitmap file (except for Simulink models)
- bmp (`-dbmp...`) — Monochrome and color bitmaps
- hdf (`-dhdf`) — Hierarchical Data Format
- svg (`-dsvg`) — Scalable Vector Graphics file (except for Simulink models)
- tiffn (`-dtiffn`) — TIFF image file, no compression

Printing Options

This table summarizes options that you can specify for `print`. The second column also shows which tutorial sections contain more detailed information. The sections listed are located under Printing and Exporting Figures with MATLAB.

Option	Description
`-adobecset`	PostScript only. Use PostScript default character set encoding. See Early PostScript 1 Printers.
`-append`	PostScript only. Append figure to existing PostScript file. See Settings That Are Driver Specific.
`-cmyk`	PostScript only. Print with CMYK colors instead of RGB. See Setting CMYK Color.
`-ddriver`	Printing only. Printer driver to use. See Drivers table.
`-dformat`	Exporting only. Graphics format to use. See Graphics Format Files table.
`-dsetup`	Windows only. Display the (platform-specific) Print Setup dialog. Settings you make in it are saved, but nothing is printed.
`-fhandle`	Handle of figure to print. Note that you cannot specify both this option and the `-s`*`windowtitle`* option. See Which Figure Is Printed.
`-loose`	PostScript and Ghostscript only. Use loose bounding box for PostScript. See Producing Uncropped Figures.
`-noui`	Suppress printing of user interface controls. See Excluding User Interface Controls.
`-opengl`	Render using the OpenGL algorithm. Note that you cannot specify this method in conjunction with `-zbuffer` or `-painters`. See Selecting a Renderer.
`-painters`	Render using the Painter's algorithm. Note that you cannot specify this method in conjunction with `-zbuffer` or `-opengl`. See Selecting a Renderer.

Option	Description
`-Pprinter`	Specify name of printer to use. See Selecting the Printer.
`-rnumber`	PostScript, JPEG, PNG, and Ghostscript only. Specify resolution in dots per inch. Defaults to `90` for Simulink, `150` for figures in image formats and when printing in Z-buffer or OpenGL mode, screen resolution for metafiles, and `864` otherwise. Use `-r0` to specify screen resolution. See Setting the Resolution.
`-s`*`windowtitle`*	Specify name of Simulink system window to print. Note that you cannot specify both this option and the `-f`*`handle`* option. See Which Figure Is Printed.
`-v`	Windows only. Display the Windows Print dialog box. The `v` stands for "verbose mode."
`-zbuffer`	Render using the Z-buffer algorithm. Note that you cannot specify this method in conjunction with `-opengl` or `-painters`. See Selecting a Renderer.

Paper Sizes

MATLAB supports a number of standard paper sizes. You can select from the following list by setting the `PaperType` property of the figure or selecting a supported paper size from the Print dialog box.

Property Value	Size (Width by Height)
`usletter`	8.5 by 11 inches
`uslegal`	11 by 14 inches
`tabloid`	11 by 17 inches
`A0`	841 by 1189 mm
`A1`	594 by 841 mm

Property Value	Size (Width by Height)
A2	420 by 594 mm
A3	297 by 420 mm
A4	210 by 297 mm
A5	148 by 210 mm
B0	1029 by 1456 mm
B1	728 by 1028 mm
B2	514 by 728 mm
B3	364 by 514 mm
B4	257 by 364 mm
B5	182 by 257 mm
arch-A	9 by 12 inches
arch-B	12 by 18 inches
arch-C	18 by 24 inches
arch-D	24 by 36 inches
arch-E	36 by 48 inches
A	8.5 by 11 inches
B	11 by 17 inches
C	17 by 22 inches
D	22 by 34 inches
E	34 by 43 inches

Printing Tips

This section includes information about specific printing issues.

Figures with Resize Functions

The `print` command produces a warning when you print a figure having a callback routine defined for the figure `ResizeFcn`. To avoid the

warning, set the figure PaperPositionMode property to auto or select **Match Figure Screen Size** in the **File⇒Page Setup** dialog box.

Troubleshooting MS Windows Printing

If you encounter problems such as segmentation violations, general protection faults, or application errors, or the output does not appear as you expect when using MS-Windows printer drivers, try the following:

- If your printer is PostScript compatible, print with one of the MATLAB built-in PostScript drivers. There are various PostScript device options that you can use with the print command: they all start with -dps.
- The behavior you are experiencing might occur only with certain versions of the print driver. Contact the print driver vendor for information on how to obtain and install a different driver.
- Try printing with one of the MATLAB built-in Ghostscript devices. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet, PCX, Canon BubbleJet, and so on.
- Copy the figure as a Windows Enhanced Metafile using the **Edit**-⇒**Copy Figure** menu item on the figure window menu or the print -dmeta option at the command line. You can then import the file into another application for printing.

  You can set copy options in the figure's **File⇒Preferences⇒Copying Options** dialog box. The Windows Enhanced Metafile clipboard format produces a better quality image than Windows Bitmap.

Printing MATLAB GUIs

You can generally obtain better results when printing a figure window that contains MATLAB uicontrols by setting these key properties:

- Set the figure PaperPositionMode property to auto. This ensures that the printed version is the same size as the onscreen version. With PaperPositionMode set to auto MATLAB does not resize the figure to fit the current value of the PaperPosition. This is particularly important if you have specified a figure ResizeFcn,

because if MATLAB resizes the figure during the print operation, `ResizeFcn` is automatically called.

To set `PaperPositionMode` on the current figure, use the command

```
set(gcf,'PaperPositionMode','auto')
```

- Set the figure `InvertHardcopy` property to `off`. By default, MATLAB changes the figure background color of printed output to white, but does not change the color of uicontrols. If you have set the background color, for example, to match the gray of the GUI devices, you must set `InvertHardcopy` to `off` to preserve the color scheme.

  To set `InvertHardcopy` on the current figure, use the command

  ```
  set(gcf,'InvertHardcopy','off')
  ```

- Use a color device if you want lines and text that are in color on the screen to be written to the output file as colored objects. Black and white devices convert colored lines and text to black or white to provide the best contrast with the background and to avoid dithering.
- Use the `print` command's `-loose` option to prevent MATLAB from using a bounding box that is tightly wrapped around objects contained in the figure. This is important if you have intentionally used space between uicontrols or axes and the edge of the figure and you want to maintain this appearance in the printed output.

If you run code that adds uicontrols to a figure when the figure is invisible, the controls will not print until the figure is made visible.

Notes on Printing Interpolated Shading with PostScript Drivers

MATLAB can print surface objects (such as graphs created with `surf` or `mesh`) using interpolated colors. However, only patch objects that are composed of triangular faces can be printed using interpolated shading.

Printed output is always interpolated in RGB space, not in the colormap colors. This means that if you are using indexed color and interpolated

face coloring, the printed output can look different from what is displayed on screen.

PostScript files generated for interpolated shading contain the color information of the graphics object's vertices and require the printer to perform the interpolation calculations. This can take an excessive amount of time and in some cases, printers might time out before finishing the print job. One solution to this problem is to interpolate the data and generate a greater number of faces, which can then be flat shaded.

To ensure that the printed output matches what you see on the screen, print using the `-zbuffer` option. To obtain higher resolution (for example, to make text look better), use the `-r` option to increase the resolution. There is, however, a tradeoff between the resolution and the size of the created PostScript file, which can be quite large at higher resolutions. The default resolution of 150 dpi generally produces good results. You can reduce the size of the output file by making the figure smaller before printing it and setting the figure `PaperPositionMode` to `auto`, or by just setting the `PaperPosition` property to a smaller size.

Examples

Specifying the Figure to Print

You can print a noncurrent figure by specifying the figure's handle. If a figure has the title "Figure 2", its handle is 2. The syntax is

```
print -fhandle
```

This example prints the figure whose handle is 2, regardless of which figure is the current figure.

```
print -f2
```

Note You must use the `-f` option if the figure's handle is hidden (i.e., its `HandleVisibility` property is set to `off`).

This example saves the figure with the handle `-f2` to a PostScript file named `Figure2`, which can be printed later.

```
print -f2 -dps 'Figure2.ps'
```

If the figure uses noninteger handles, use the `figure` command to get its value, and then pass it in as the first argument.

```
h = figure('IntegerHandle','off')
print h -depson
```

You can also pass a figure handle as a variable to the function form of `print`. For example,

```
h = figure; plot(1:4,5:8)
print(h)
```

This example uses the function form of `print` to enable a filename to be passed in as a variable.

```
filename = 'mydata';
print('-f3', '-dpsc', filename);
```

(Because a filename is specified, the figure will be printed to a file.)

Specifying the Model to Print

To print a noncurrent Simulink model, use the `-s` option with the title of the window. For example, this command prints the Simulink window titled `f14`.

```
print -sf14
```

If the window title includes any spaces, you must call the function form rather than the command form of `print`. For example, this command saves Simulink window title `Thruster Control`.

```
print('-sThruster Control')
```

To print the current system, use

```
print -s
```

For information about issues specific to printing Simulink windows, see the Simulink documentation.

Printing Figures at Screen Size

This example prints a surface plot with interpolated shading. Setting the current figure's (`gcf`) `PaperPositionMode` to `auto` enables you to resize the figure window and print it at the size you see on the screen. See Options and the previous section for information on the `-zbuffer` and `-r200` options.

```
surf(peaks)
shading interp
set(gcf,'PaperPositionMode','auto')
print -dpsc2 -zbuffer -r200
```

For additional details, see Printing Images in the MATLAB Graphics documentation.

Batch Processing

You can use the function form of `print` to pass variables containing file names. For example, this `for` loop uses filenames stored in a cell array to create a series of graphs and prints each one with a different file name.

```
fnames = {'file1', 'file2', 'file3'};
for k=1:length(fnames)
    surf(peaks)
    print('-dtiff','-r200',fnames{k})
end
```

Tiff Preview

The command

```
print -depsc -tiff -r300 picture1
```

saves the current figure at 300 dpi, in a color Encapsulated PostScript file named `picture1.eps`. The `-tiff` option creates a 72 dpi TIFF preview, which many word processor applications can display on screen after you import the EPS file. This enables you to view the picture on screen within your word processor and print the document to a PostScript printer using a resolution of 300 dpi.

See Also

`orient`, `figure`

Purpose

Print dialog box

Syntax

```
printdlg
printdlg(fig)
printdlg('-crossplatform',fig)
printdlg('-setup',fig)
```

Description

`printdlg` prints the current figure.

`printdlg(fig)` creates a modal dialog box from which you can print the figure window identified by the handle `fig`. Note that uimenus do not print.

`printdlg('-crossplatform',fig)` displays the standard cross-platform MATLAB printing dialog rather than the built-in printing dialog box for Microsoft Windows computers. Insert this option before the `fig` argument.

`printdlg('-setup',fig)` forces the printing dialog to appear in a setup mode. Here one can set the default printing options without actually printing.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

See Also

`pagesetupdlg`, `printpreview`

printpreview

Purpose

Preview figure to print

Contents

GUI Alternative

Use **File > Print Preview** on the figure window menu to access the Print Preview dialog box, described below. For details, see "Using Print Preview" in the MATLAB Graphics documentation.

Syntax

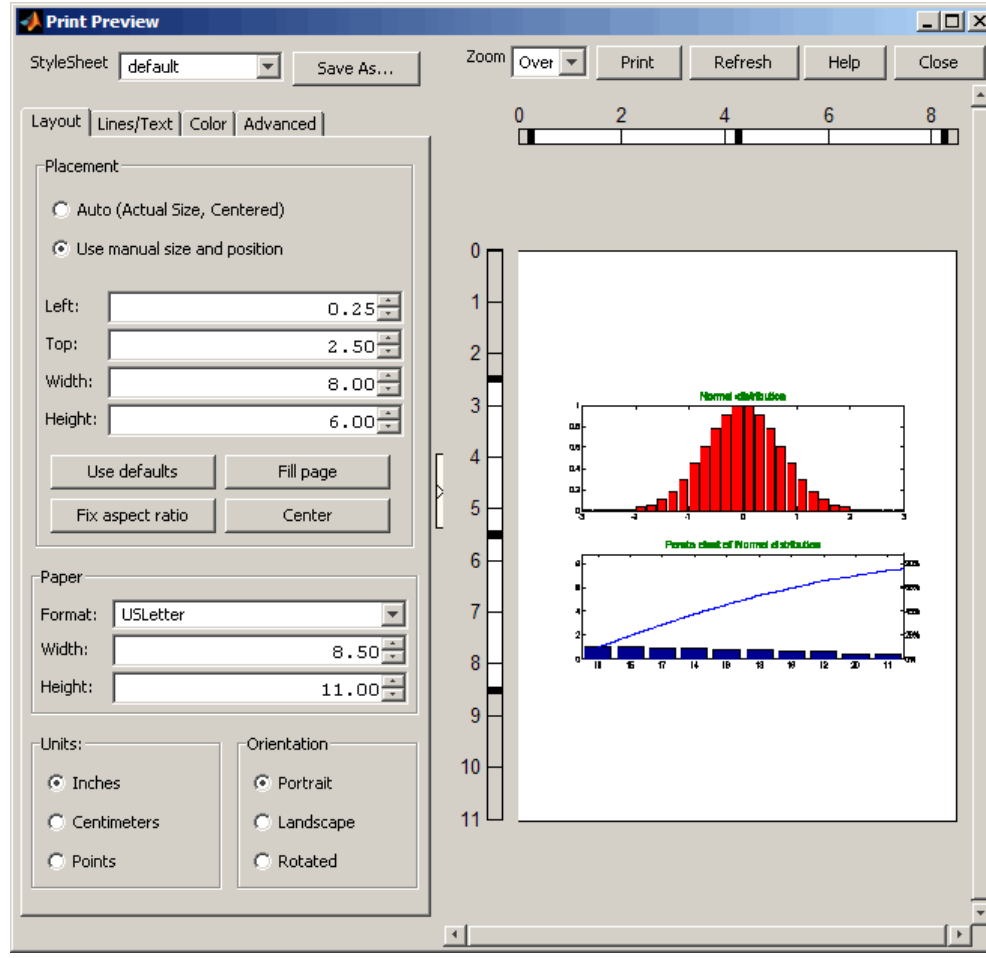
```
printpreview
printpreview(f)
```

Description

`printpreview` displays a dialog box showing the figure in the currently active figure window as it will print. A scaled version of the figure displays in the right-hand pane of the GUI.

`printpreview(f)` displays a dialog box showing the figure having the handle `f` as it will print.

Use the Print Preview dialog box, shown below, to control the layout and appearance of figures before sending them to a printer or print file. Controls are grouped into four tabbed panes: **Layout**, **Lines/Text**, **Color**, and **Advanced**.



Right Pane Controls

You can position and scale plots on the printed page using the rulers in the right-hand pane of the Print Preview dialog. Use the outer ruler handlebars to change margins. Moving them changes plot proportions. Use the center ruler handlebars to change the position of the plot on the page. Plot proportions do not change, but you can move portions of

the plot off the paper. The buttons on that pane let you refresh the plot, close the dialog (preserving all current settings), print the page immediately, or obtain context-sensitive help. Use the **Zoom** box and scroll bars to view and position page elements more precisely.

The Layout Tab

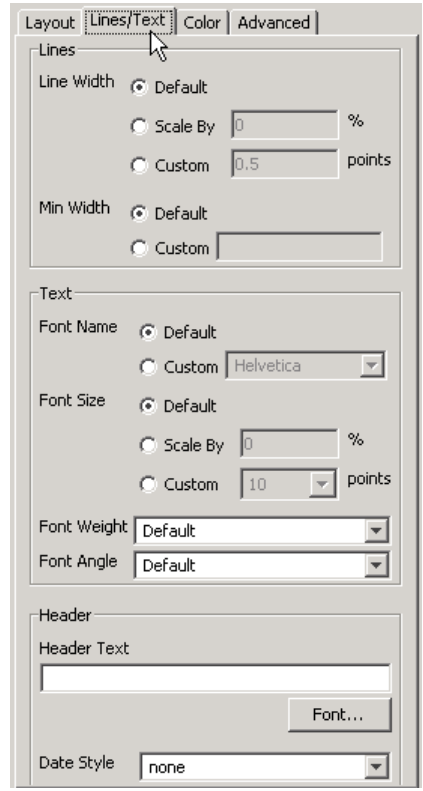
Use the **Layout** tab, shown above, to control the paper format and placement of the plot on printed pages. The following table summarizes the **Layout** options:

Group	Option	Description
Placement	**Auto**	Let MATLAB decide placement of plot on page
	Use manual...	Specify position parameters for plot on page
	Top, Left, Width, Height	Standard position parameters in current units
	Use defaults	Revert to default position
	Fill page	Expand figure to fill printable area
	Fix aspect ratio	Correct height/width ratio
	Center	Center plot on printed page
Paper	**Format**	U.S. and ISO sheet size selector
	Width, Height	Sheet size in current units
Units	**Inches**	Use inches as units for dimensions and positions
	Centimeters	Use centimeters as units for dimensions and positions
	Points	Use points as units for dimensions and positions
Orientation	**Portrait**	Upright paper orientation

Group	Option	Description
	Landscape	Sideways paper orientation
	Rotated	Currently the same as **Landscape**

The Lines/Text Tab

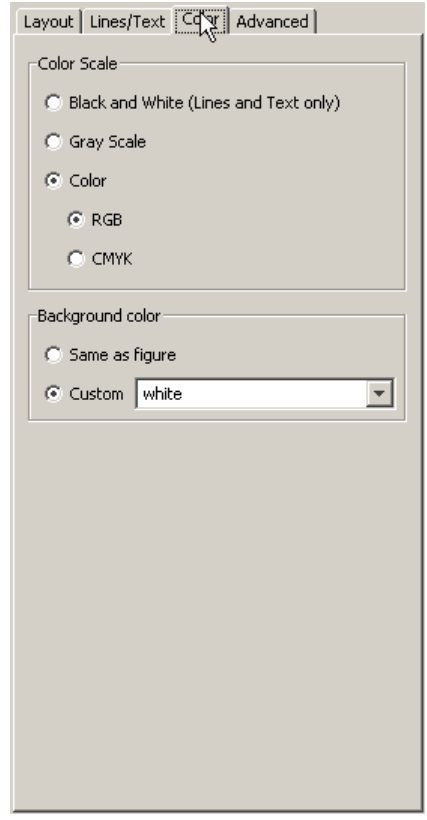
Use the **Lines/Text** tab, shown below, to control the line weights, font characteristics, and headers for printed pages. The following table summarizes the **Lines/Text** options:



Group	Option	Description
Lines	**Line Width**	Scale all lines by a percentage from 0 upward (100 being no change), print lines at a specified point size, or default line widths used on the plot
	Min Width	Smallest line width (in points) to use when printing; defaults to 0.5 point
Text	**Font Name**	Select a system font for all text on plot, or default to fonts currently used on the plot
	Font Size	Scale all text by a percentage from 0 upward (100 being no change), print text at a specified point size, or default to this
	Font Weight	Select `Normal` ... `Bold` font styling for all text from drop-down menu or default to the font weights used on the plot
	Font Angle	Select `Normal`, `Italic` or `Oblique` font styling for all text from drop-down menu or default to the font angles used on the plot
Header	**Header Text**	Type the text to appear on the header at the upper left of printed pages, or leave blank for no header
	Date Style	Select a date format to have today's date appear at the upper left of printed pages, or `none` for no date

The Color Tab

Use the **Color** tab, shown below, to control how colors are printed for lines and backgrounds. The following table summarizes the **Color** options:

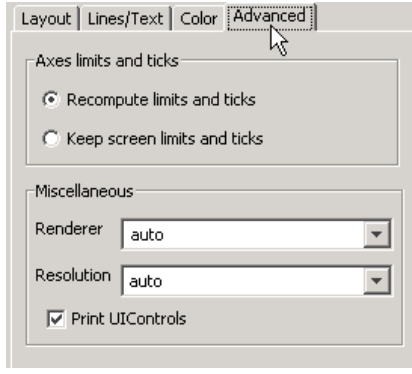


Group	Option	Description
Color Scale	**Black and White**	Select to print lines and text in black and white, but use color for patches and other objects
	Gray Scale	Convert colors to shades of gray on printed pages

Group	Option	Description
	Color	Print everything in color, matching colors on plot; select `RGB` (default) or `CMYK` color model for printing
Background Color	**Same as figure**	Print the figure's background color as it is
	Custom	Select a color name, or type a colorspec for the background; `white` (default) implies no background color, even on colored paper.

The Advanced Tab

Use the **Advanced** tab, shown below, to control finer details of printing, such as limits and ticks, renderer, resolution, and the printing of UIControls. The following table summarizes the **Advanced** options:



Group	Option	Description
Axes limits and ticks	**Recompute limits and ticks**	Redraw *x*- and *y*-axes ticks and limits based on printed plot size (default)

Group	Option	Description
	Keep limits and ticks	Use the *x*- and *y*-axes ticks and limits shown on the plot when printing the previewed figure
Miscellaneous	**Renderer**	Select a rendering algorithm for printing: `painters`, `zbuffer`, `opengl`, or `auto` (default)
	Resolution	Select resolution to print at in dots per inch: `150`, `300`, `600`, or `auto` (default), or type in any other positive value
	Print UIControls	Print all visible UIControls in the figure (default), or uncheck to exclude them from being printed

See Also

`printdlg`, `pagesetupdlg`

For more information, see How to Print or Export in the MATLAB Graphics documentation.

Purpose

Product of array elements

Syntax

```
B = prod(A)
B = prod(A,dim)
```

Description

`B = prod(A)` returns the products along different dimensions of an array.

If `A` is a vector, `prod(A)` returns the product of the elements.

If `A` is a matrix, `prod(A)` treats the columns of `A` as vectors, returning a row vector of the products of each column.

If `A` is a multidimensional array, `prod(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = prod(A,dim)` takes the products along the dimension of `A` specified by scalar `dim`.

Examples

The magic square of order 3 is

```
M = magic(3)

M =
     8     1     6
     3     5     7
     4     9     2
```

The product of the elements in each column is

```
prod(M) =

    96    45    84
```

The product of the elements in each row can be obtained by:

```
prod(M,2) =

     48
```

```
   105
    72
```

See Also

cumprod, diff, sum

profile

Purpose

Profile execution time for function

GUI Alternatives

As an alternative to the `profile` function, select **Desktop > Profiler** to open the Profiler.

Syntax

```
profile on
profile on -detail level
profile on -history
profile on -nohistory
profile on -timer clock
profile on -detail level -history -timer clock
profile off
profile resume
profile clear
profile viewer
S = profile('status')
stats = profile('info')
```

Description

The `profile` function helps you debug and optimize M-files by tracking their execution time. For each function in the M-file, `profile` records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Some people use `profile` simply to see the child functions; see also `depfun` for that purpose. To open the Profiler graphical user interface, use the `profile viewer` syntax. Profile time is CPU time. The total time reported by the Profiler is not the same as the time reported using the `tic` and `toc` functions or the time you would observe using a stopwatch. To change options, stop profiling and then start or resume profiling with new options.

`profile on` starts the Profiler, clearing previously recorded profile statistics.

`profile on -detail level` starts the Profiler, clearing previously recorded profile statistics, and specifies the set of functions you want to profile. The level applies to subsequent uses of `profile` or the Profiler, until you change it. Allowable values for *`level`* are

- `'builtin'`—Gathers information about M-functions, M-subfunctions, and MEX-functions, plus built-in functions, such as `eig`.
- `'mmex'`—Gathers information about M-functions, M-subfunctions, and MEX-functions. This is the default value.

`profile` **`on -history`** starts the Profiler, clearing previously recorded profile statistics, and records the exact sequence of function calls. The `profile` function records up to 10,000 function entry and exit events. For more than 10,000 events, `profile` continues to record other profile statistics, but not the sequence of calls. By default, the `history` option is not enabled.

`profile` **`on -nohistory`** starts the Profiler, clearing previously recorded profile statistics, and disables further recording of the history (exact sequence of function calls). Use the `-nohistory` option after having previously set the `-history` option. All other profiling statistics continue to accumulate.

`profile` **`on -timer`** *`clock`* starts the Profiler, clearing previously recorded profile statistics, and specifies the type of time to use. Allowable values for *`clock`* are

- `'cpu'`—The Profiler uses compute time (the default).
- `'real'`—The Profiler uses wall-clock time.

For example, `cpu` time for the `pause` function would be small, but `real` time would account for the actual time paused.

`profile` **`on -detail`** *`level`* **`-history -timer`** *`clock`* starts the Profiler using all of these specified options. Any order is acceptable, as is a subset.

`profile` **`off`** stops the Profiler.

`profile` **`resume`** restarts the Profiler without clearing previously recorded statistics.

`profile` **`clear`** clears the statistics recorded by `profile`.

`profile viewer` stops the Profiler and displays the results in the Profiler window. For more information, see Profiling for Improving Performance in the Desktop Tools and Development Environment documentation.

`S = profile('status')` returns a structure containing information about the current status of the Profiler. The table lists the fields in the order they appear in the structure.

Field	Values
`ProfilerStatus`	`'on'` or `'off'`
`DetailLevel`	`'mmex'` or `'builtin'`
`Timer`	`'cpu'` or `'real'`
`HistoryTracking`	`'on'` or `'off'`

`stats = profile('info')` stops the Profiler and displays a structure containing the results. Use this function to access the data generated by `profile`. The table lists the fields in the order they appear in the structure.

Field	Description
`FunctionTable`	Structure array containing statistics about each function called
`FunctionHistory`	Array containing function call history
`ClockPrecision`	Precision of `profile`'s time measurement
`ClockSpeed`	Estimated clock speed of the CPU
`Name`	Name of the profiler

The `FunctionTable` field is an array of structures, where each structure contains information about one of the functions or subfunctions called during execution. The following table lists these fields in the order they appear in the structure.

Field	Description
`CompleteName`	Full path to `FunctionName`, including subfunctions
`FunctionName`	Function name; includes subfunctions
`FileName`	Full path to `FunctionName`, with file extension, excluding subfunctions
`Type`	M-functions, MEX-functions, and many other types of functions including M-subfunctions, nested functions, and anonymous functions
`NumCalls`	Number of times the function was called
`TotalTime`	Total time spent in the function and its child functions
`TotalRecursiveTime`	No longer used.
`Children`	`FunctionTable` indices to child functions
`Parents`	`FunctionTable` indices to parent functions
`ExecutedLines`	Array containing line-by-line details for the function being profiled. Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix. Column 2: Number of times the line was executed Column 3: Total time spent on that line. Note: The sum of Column 3 entries does not necessarily add up to the function's `TotalTime`.

Field	Description
IsRecursive	BOOLEAN value: Logical 1 (true) if recursive, otherwise logical 0 (false)
PartialData	BOOLEAN value: Logical 1 (true) if function was modified during profiling, for example by being edited or cleared. In that event, data was collected only up until the point when the function was modified.

Examples

Profile and Display Results

This example profiles the MATLAB `magic` command and then displays the results in the Profiler window. The example then retrieves the profile data on which the HTML display is based and uses the `profsave` command to save the profile data in HTML form.

```
profile on
plot(magic(35))
profile viewer
p = profile('info');
profsave(p,'profile_results')
```

Profile and Save Results

Another way to save profile data is to store it in a MAT-file. This example stores the profile data in a MAT-file, clears the profile data from memory, and then loads the profile data from the MAT-file. This example also shows a way to bring the reloaded profile data into the Profiler graphical interface as live profile data, not as a static HTML page.

```
p = profile('info');
save myprofiledata p
clear p
load myprofiledata
profview(0,p)
```

Profile and Show Results Including History

This example illustrates an effective way to view the results of profiling when the `history` option is enabled. The history data describes the sequence of functions entered and exited during execution. The `profile` command returns history data in the `FunctionHistory` field of the structure it returns. The history data is a 2-by-n array. The first row contains Boolean values, where `0` means entrance into a function and `1` means exit from a function. The second row identifies the function being entered or exited by its index in the `FunctionTable` field. This example reads the history data and displays it in the MATLAB Command Window.

```
profile on -history
plot(magic(4));
p = profile('info');

for n = 1:size(p.FunctionHistory,2)
 if p.FunctionHistory(1,n)==0
        str = 'entering function: ';
 else
        str = 'exiting function: ';
 end
 disp([str p.FunctionTable(p.FunctionHistory(2,n)).FunctionName])
end
```

See Also

`depdir`, `depfun`, `mlint`, `profsave`

Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation

profsave

Purpose

Save profile report in HTML format

Syntax

```
profsave
profsave(profinfo)
profsave(profinfo,dirname)
```

Description

`profsave` executes the `profile('info')` function and saves the results in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of the structure returned by `profile`. By default, `profsave` stores the HTML files in a subdirectory of the current directory named `profile_results`.

`profsave(profinfo)` saves the profiling results, `profinfo`, in HTML format. `profinfo` is a structure of profiling information returned by the `profile('info')` function.

`profsave(profinfo,dirname)` saves the profiling results, `profinfo`, in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of `profinfo` and stores them in the directory specified by `dirname`.

Examples

Run `profile` and save the results.

```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

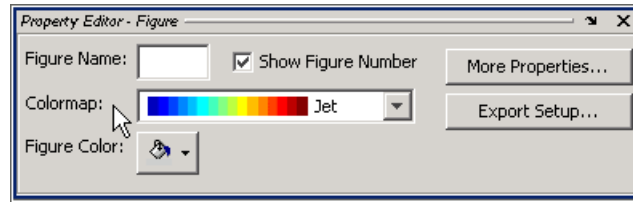
See Also

`profile`

Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Open Property Editor



Syntax

```
propedit
propedit(handle_list)
```

Description

`propedit` starts the Property Editor, a graphical user interface to the properties of graphics objects. If no current figure exists, `propedit` will create one.

`propedit(handle_list)` edits the properties for the object (or objects) in `handle_list`.

Starting the Property Editor enables plot editing mode for the figure.

See Also

`inspect`, `plotedit`, `propertyeditor`

propedit (COM)

Purpose

Open built-in property page for control

Syntax

```
h.propedit
propedit(h)
```

Description

`h.propedit` requests the control to display its built-in property page. Note that some controls do not have a built-in property page. For those controls, this command fails.

`propedit(h)` is an alternate syntax for the same operation.

Examples

Create a Microsoft Calendar control and display its property page:

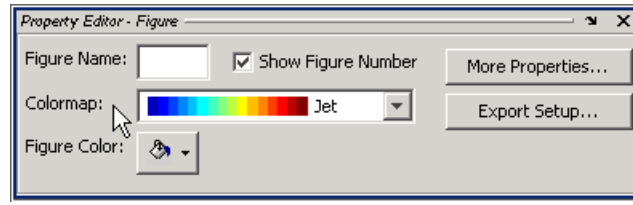
```
cal = actxcontrol('mscal.calendar', [0 0 500 500]);
cal.propedit
```

See Also

`inspect`, `get`

Purpose

Show or hide property editor



GUI Alternatives

Click the larger **Plotting Tools** icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Open or close the **Property Editor** tool from the figure's **View** menu. For details, see "The Property Editor" in the MATLAB Graphics documentation.

Syntax

```
propertyeditor('on')
propertyeditor('off')
propertyeditor('toggle')
propertyeditor
propertyeditor(figure_handle,...)
```

Description

`propertyeditor('on')` displays the Property Editor on the current figure.

`propertyeditor('off')` hides the Property Editor on the current figure.

`propertyeditor('toggle')` or `propertyeditor` toggles the visibility of the property editor on the current figure.

`propertyeditor(figure_handle,...)` displays or hides the Property Editor on the figure specified by `figure_handle`.

See Also

`plottools`, `plotbrowser`, `figurepalette`, `inspect`

Purpose

Psi (polygamma) function

Syntax

```
Y = psi(X)
Y = psi(k,X)
Y = psi(k0:k1,X)
```

Description

`Y = psi(X)` evaluates the Ψ function for each element of array `X`. `X` must be real and nonnegative. The Ψ function, also known as the digamma function, is the logarithmic derivative of the gamma function

$$\begin{aligned}\psi(x) &= \mathbf{digamma}(x) \\ &= \frac{d(\log(\Gamma(x)))}{dx} \\ &= \frac{d(\Gamma(x))/dx}{\Gamma(x)}\end{aligned}$$

`Y = psi(k,X)` evaluates the `k`th derivative of Ψ at the elements of `X`. `psi(0,X)` is the digamma function, `psi(1,X)` is the trigamma function, `psi(2,X)` is the tetragamma function, etc.

`Y = psi(k0:k1,X)` evaluates derivatives of order `k0` through `k1` at `X`. `Y(k,j)` is the `(k-1+k0)`th derivative of Ψ, evaluated at `X(j)`.

Examples

Example 1

Use the `psi` function to calculate Euler's constant, γ.

```
format long
-psi(1)
ans =
   0.57721566490153

-psi(0,1)
ans =
   0.57721566490153
```

Example 2

The trigamma function of 2, `psi(1,2)`, is the same as $(\pi^2/6) - 1$.

```
format long
psi(1,2)
ans =
   0.64493406684823

pi^2/6 - 1
ans =
   0.64493406684823
```

Example 3

This code produces the first page of Table 6.1 in Abramowitz and Stegun [1].

```
x = (1:.005:1.250)';
[x gamma(x) gammaln(x) psi(0:1,x)' x-1]
```

Example 4

This code produces a portion of Table 6.2 in [1].

```
psi(2:3,1:.01:2)'
```

See Also

`gamma`, `gammainc`, `gammaln`

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Sections 6.3 and 6.4.

publish

Purpose

Publish M-file containing cells, saving output to file of specified type

GUI Alternatives

As an alternative to the `publish` function, use the **File > Publish To** menu items in the Editor/Debugger.

Syntax

```
publish('script')
publish('script','format')
publish('script', options)
publish('function', options)
```

Description

`publish('script')` runs the M-file script named `script` in the base workspace one cell at a time, and saves the code, comments, and results to an HTML output file. The output file is named `script.html` and is stored, along with other supporting output files, in an `html` subdirectory in `script`'s directory.

`publish('script','format')` runs the M-file script named `script`, one cell at a time in the base workspace, and publishes the code, comments, and results to an output file using the specified *`format`*. Allowable values for *`format`* are `html` (the default), `xml`, `latex` for LaTeX, `doc` for Microsoft Word documents, and `ppt` for Microsoft PowerPoint documents. The output file is named `script.format` and is stored, along with other supporting output files, in an `html` subdirectory in `script`'s directory. The `doc` format requires the Microsoft Word application, and the `ppt` format requires PowerPoint application. When publishing to HTML, the M-file code is included at the end of published HTML file as comments, even when the `showCode` option is set to `false`. Because it is included as comments, it does not display in a Web browser. Use the `grabcode` function to extract the code from the HTML file.

`publish('script', options)` publishes using the structure *`options`*, which can contain any of the fields and corresponding value for each field as shown in Options for publish on page 2-2583. Create and save structures for the options you use regularly. For details about the values, see and Publishing Images preferences in the online documentation for MATLAB.

`publish('function', options)` publishes an M-file function using the structure *`options`*. The `evalCode` field must be set to `false` to publish a function. Publishing an M-file function essentially saves the M-file to another format, such as HTML, which allows display with formatting in a Web browser.

Options for publish

Field	Allowable Values
`format`	`'doc'`,`'html'` (default), `'latex'`, `'ppt'`, `'xml'`
`stylesheet`	`''` (default), XSL filename (used only when `format` is `html`, `latex`, or `xml`)
`outputDir`	`''` (default, a subfolder named `html`), full pathname
`imageFormat`	`'png'` (default unless `format` is `latex`), `'epsc2'` (default when `format` is `latex`), any format supported by `print` when `figureSnapMethod` is `print`, any format supported by `imwrite` functions when `figureSnapMethod` is `getframe`.
`figureSnapMethod`	`'print'` (default),`'getframe'`
`useNewFigure`	`true` (default), `false`
`maxHeight`	`[]` (default), positive integer specifying number of pixels
`maxWidth`	`[]` (default), positive integer specifying number of pixels
`showCode`	`true` (default), `false`
`evalCode`	`true` (default), `false`
`catchError`	`true` (default, continues publishing and includes the error in the published file), `false` (displays the error and publishing ends)
`stopOnError`	`true` (default), `false`

Options for publish (Continued)

Field	Allowable Values
`createThumbnail`	`true` (default), `false`
`maxOutputLines`	`Inf` (default), nonnegative integer specifying the maximum number of output lines before truncation of output

Examples

Publish to HTML Format

To publish the M-file script `d:/mymfiles/sine_wave.m` to HTML, run

```
publish('d:/mymfiles/sine_wave.m', 'html')
```

MATLAB runs the file and saves the code, comments, and results to `d:/mymfiles/html/sine_wave.html`. Open that file in the Web browser to view the published document.

Publish with Options

This example defines the structure `options_doc_nocode`, publishes `sine_wave.m` using the defined options, and displays the resulting file. The resulting file is a Word document, `d:/nocode_output/sine_wave.doc` and includes results, but not MATLAB code.

```
options_doc_nocode.format='doc'
options_doc_nocode.outputDir='d:/nocode_output'
options_doc_nocode.showCode=false
publish('d:/mymfiles/sine_wave.m',options_doc_nocode)
winopen('d:/nocode_output/sine_wave.doc')
```

Publish Function M-File (Save M-File as HTML)

This example defines the structure `function_options`, publishes the function `d:/collatzplot.m`, and displays the resulting file, an HTML document, `d:/html/collatzplot.html`.

```
function_options.format='html'
function_options.evalCode=false
publish('d:/collatzplot.m',function_options)
web('d:/html/collatzplot.html')
```

See Also

`grabcode`, `notebook`, `web`, `winopen`

MATLAB Desktop Tools and Development Environment documentation, specifically

- Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells
- Defining Cells

PutCharArray

Purpose

Store character array in server

Syntax

MATLAB Client

```
h.PutCharArray('varname', 'workspace', 'string')
PutCharArray(h, 'varname', 'workspace', 'string')
invoke(h, 'PutCharArray', 'varname', 'workspace', 'string')
```

Method Signature

```
PutCharArray([in] BSTR varname, [in] BSTR workspace,
[in] BSTR string)
```

Visual Basic Client

```
PutCharArray(varname As String, workspace As String,
string As String)
```

Description

`PutCharArray` stores the character array in `string` in the specified `workspace` of the server attached to handle `h`, assigning to it the variable `varname`. The `workspace` argument can be either `base` or `global`.

Remarks

The character array specified in the `string` argument can have any dimensions. However, `PutCharArray` changes the dimensions to a 1–by-n column-wise representation, where n is the number of characters in the array. Executing the following commands in MATLAB illustrates this behavior:

```
h = actxserver('matlab.application');
chArr = ['abc'; 'def'; 'ghk']
chArr =
abc
def
ghk

h.PutCharArray('Foo', 'base', chArr)
tstArr = h.GetCharArray('Foo', 'base')
tstArr =
adgbehcfk
```

Server function names, like `PutCharArray`, are case sensitive when using the dot notation syntax shown in the Syntax section.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Examples

Store string `str` in the base workspace of the server using `PutCharArray`.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutCharArray('str', 'base', ...
   'He jests at scars that never felt a wound.')

S = h.GetCharArray('str', 'base')
S =
    He jests at scars that never felt a wound.
```

Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab As Object
Try
    Matlab = GetObject(, "matlab.application")
Catch e As Exception
    Matlab = CreateObject("matlab.application")
End Try
MsgBox("MATLAB window created; now open it...")
```

Open the MATLAB window, then click **Ok**.

```
Matlab.PutCharArray("str", "base", _
       "He jests at scars that never felt a wound.")
MsgBox("In MATLAB, type" & vbCrLf _
  & "str")
```

In the MATLAB window type `str`; MATLAB displays

```
str =
He jests at scars that never felt a wound.
```

Click **Ok**.

```
MsgBox("closing MATLAB window...")
```

Click **Ok** to close and terminate MATLAB.

```
Matlab.Quit()
```

See Also

`GetCharArray`, `PutWorkspaceData`, `GetWorkspaceData`, `Execute`

Purpose

Store matrix in server

Syntax

MATLAB Client

```
h.PutFullMatrix('varname', 'workspace', xreal, ximag)
PutFullMatrix(h, 'varname', 'workspace', xreal, ximag)
invoke(h, 'PutFullMatrix', 'varname', 'workspace',
xreal, ximag)
```

Method Signature

```
PutFullMatrix([in] BSTR varname, [in] BSTR workspace,
   [in] SAFEARRAY(double) xreal, [in] SAFEARRAY(double) ximag)
```

Visual Basic Client

```
PutFullMatrix([in] varname As String, [in] workspace As String,
   [in] xreal As Double, [in] ximag As Double)
```

Description

`PutFullMatrix` stores a matrix in the specified `workspace` of the server attached to handle `h`, assigning to it the variable `varname`. Enter the real and imaginary parts of the matrix in the `xreal` and `ximag` input arguments. The `workspace` argument can be either `base` or `global`.

Remarks

The matrix specified in the `xreal` and `ximag` arguments cannot be scalar, an empty array, or have more than two dimensions.

Server function names, like `PutFullMatrix`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of `safearray` which is not supported by VBScript.

Examples

Writing to the Base Workspace Example

Assign a 5-by-5 real matrix to the variable `M` in the base workspace of the server, and then read it back with `GetFullMatrix`. The real and imaginary parts are passed in through separate arrays of doubles.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutFullMatrix('M', 'base', rand(5), zeros(5))
% One output returns real, use two for real and imag
xreal = h.GetFullMatrix('M', 'base', zeros(5), zeros(5))
xreal =
    0.9501    0.7621    0.6154    0.4057    0.0579
    0.2311    0.4565    0.7919    0.9355    0.3529
    0.6068    0.0185    0.9218    0.9169    0.8132
    0.4860    0.8214    0.7382    0.4103    0.0099
    0.8913    0.4447    0.1763    0.8936    0.1389
```

Visual Basic .NET Client

```
Dim MatLab As Object
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim ZReal(4, 4) As Double
Dim ZImag(4, 4) As Double
Dim i, j As Integer

For i = 0 To 4
   For j = 0 To 4
   XReal(i, j)  = Rnd() * 6
   XImag(i, j) = 0
   Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("M", "base", XReal, XImag)
MatLab.GetFullMatrix("M", "base", ZReal, ZImag)
```

Writing to the Global Workspace Example

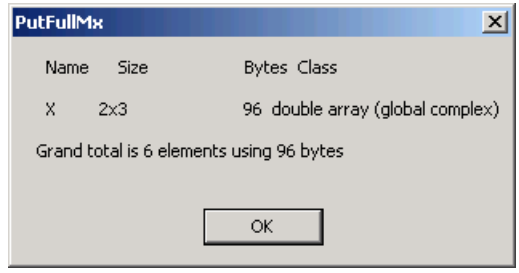
Write a matrix to the global workspace of the server and then examine the server's global workspace from the client.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutFullMatrix('X', 'global', [1 3 5; 2 4 6], ...
                [1 1 1; 1 1 1])
h.invoke('Execute', 'whos global')
ans =
  Name    Size        Bytes  Class
  X       2x3         96  double array (global complex)
Grand total is 6 elements using 96 bytes
```

Visual Basic .NET Client

```
Dim MatLab As Object
Dim XReal(1, 2) As Double
Dim XImag(1, 2) As Double
Dim result As String
Dim i, j As Integer

For i = 0 To 1
 For j = 0 To 2
  XReal(i, j) = (j * 2 + 1) + i
  XImag(i, j) = 1
 Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("X", "global", XReal, XImag)
result = Matlab.Execute("whos global")
MsgBox(result)
```



See Also

`GetFullMatrix`, `PutWorkspaceData`, , `GetWorkspaceDataExecute`

Purpose

Store data in server workspace

Syntax

MATLAB Client

```
h.PutWorkspaceData('varname', 'workspace', data)
PutWorkspaceData(h, 'varname', 'workspace', data)
invoke(h, 'PutWorkspaceData', 'varname', 'workspace', data)
```

Method Signature

```
PutWorkspaceData([in] BSTR varname, [in] BSTR workspace,
[in] VARIANT data)
```

Visual Basic Client

```
PutWorkspaceData(varname As String, workspace As String,
data As Object)
```

Description

`PutWorkspaceData` stores `data` in the specified `workspace` of the server attached to handle `h`, assigning to it the variable `varname`. The `workspace` argument can be either `base` or `global`.

Note `PutWorkspaceData` works on all MATLAB data types except sparse arrays, structure arrays, and function handles. Use the `Execute` method for these data types.

Passing Character Arrays

MATLAB enables you to define 2-D character arrays such as the following:

```
chArr = ['abc';'def';'ghk']
chArr =
abc
def
ghk

size(chArr)
ans =
     3     3
```

However, PutWorkspaceData does not preserve the dimensions of character arrays when passing them to a COM server. 2-D arrays are converted to 1-by-n arrays of characters, where n equals the number of characters in the original array plus one newline character for each row in the original array. This means that chArr above is converted to a 1-by-12 array, but the newline characters make it display with three rows in the MATLAB command window. For example,

```
h = actxserver('matlab.application');
h.PutWorkspaceData('Foo','base',chArr);
tstArr = h.GetWorkspaceData('Foo','base')
tstArr =
abc
def
ghk

size(tstArr)
ans =
     1    12
```

Remarks

You can use PutWorkspaceData in place of PutFullMatrix and PutCharArray to pass numeric and character array data respectively to the server.

Server function names, like PutWorkspaceData, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

The GetWorkspaceData and PutWorkspaceData functions pass numeric data as a variant data type. These functions are especially useful for VBScript clients as VBScript does not support the safearray data type used by GetFullMatrix and PutFullMatrix.

Examples

Create an array in the client and assign it to variable A in the base workspace of the server:

MATLAB Client

```
h = actxserver('matlab.application');
for i = 0:6
   data(i+1)  = i * 15;
end
h.PutWorkspaceData('A', 'base', data)
```

Visual Basic .NET Client

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab As Object
Dim data(6) As Double
Dim i As Integer
MatLab = CreateObject("matlab.application")
For i = 0 To 6
   data(i)  = i * 15
Next i
MatLab.PutWorkspaceData("A", "base", data)
MsgBox("In MATLAB, type" & vbCrLf & "A")
```

Open the MATLAB window and type `A`. MATLAB displays

```
A =
     0    15    30    45    60    75    90
```

Click **Ok** to close and terminate MATLAB.

See Also

`GetWorkspaceData`, `PutFullMatrix`, , `GetFullMatrix`, `PutCharArray`, `GetCharArrayExecute`

See “Introduction” for more examples.

pwd

Purpose

Identify current directory

Graphical Interface

As an alternative to the `pwd` function, use the "Current Directory Field" 

in the MATLAB desktop toolbar.

Syntax

```
pwd
s = pwd
```

Description

`pwd` displays the current working directory.

`s = pwd` returns the current directory to the variable `s`.

On Windows platforms, go directly to the current working directory using

```
winopen(pwd)
```

See Also

`cd`, `dir`, `fileparts`, `mfilename`, `path`, `what`, `winopen`

Purpose

Quasi-minimal residual method

Syntax

```
x = qmr(A,b)
qmr(A,b,tol)
qmr(A,b,tol,maxit)
qmr(A,b,tol,maxit,M)
qmr(A,b,tol,maxit,M1,M2)
qmr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = qmr(A,b,...)
[x,flag,relres] = qmr(A,b,...)
[x,flag,relres,iter] = qmr(A,b,...)
[x,flag,relres,iter,resvec] = qmr(A,b,...)
```

Description

`x = qmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `qmr` converges, a message to that effect is displayed. If `qmr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`qmr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `qmr` uses the default, `1e-6`.

`qmr(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `qmr` uses the default, `min(n,20)`.

`qmr(A,b,tol,maxit,M)` and `qmr(A,b,tol,maxit,M1,M2)` use preconditioners `M` or `M = M1*M2` and effectively solve the system

`inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `qmr` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `qmr` uses the default, an all zero vector.

`[x,flag] = qmr(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`qmr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`qmr` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	The method stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `qmr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = qmr(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = qmr(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = qmr(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Example 1

```
n = 100;
on = ones(n,1);
```

```
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8; maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = qmr(A,b,tol,maxit,M1,M2);
```

displays the message

```
qmr converged at iteration 9 to a solution...
with relative residual
5.6e-009
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_qmr` that

- Calls `qmr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_qmr` are available to `afun`.

The following shows the code for `run_qmr`:

```
function x1 = run_qmr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = qmr(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
```

```
            y = 4 * x;
            y(1:n-1) = y(1:n-1) - 2 * x(2:n);
            y(2:n) = y(2:n) - x(1:n-1);
         elseif strcmp(transp_flag,'notransp') % y = A*x
            y = 4 * x;
            y(2:n) = y(2:n) - 2 * x(1:n-1);
            y(1:n-1) = y(1:n-1) - x(2:n);
         end
      end
   end
```

When you enter

```
x1=run_qmr;
```

MATLAB displays the message

```
qmr converged at iteration 9 to a solution with relative residual
5.6e-009
```

Example 3

```
load west0479;
A = west0479;
b = sum(A,2);
[x,flag] = qmr(A,b)
```

`flag` is `1` because `qmr` does not converge to the default tolerance `1e-6` within the default 20 iterations.

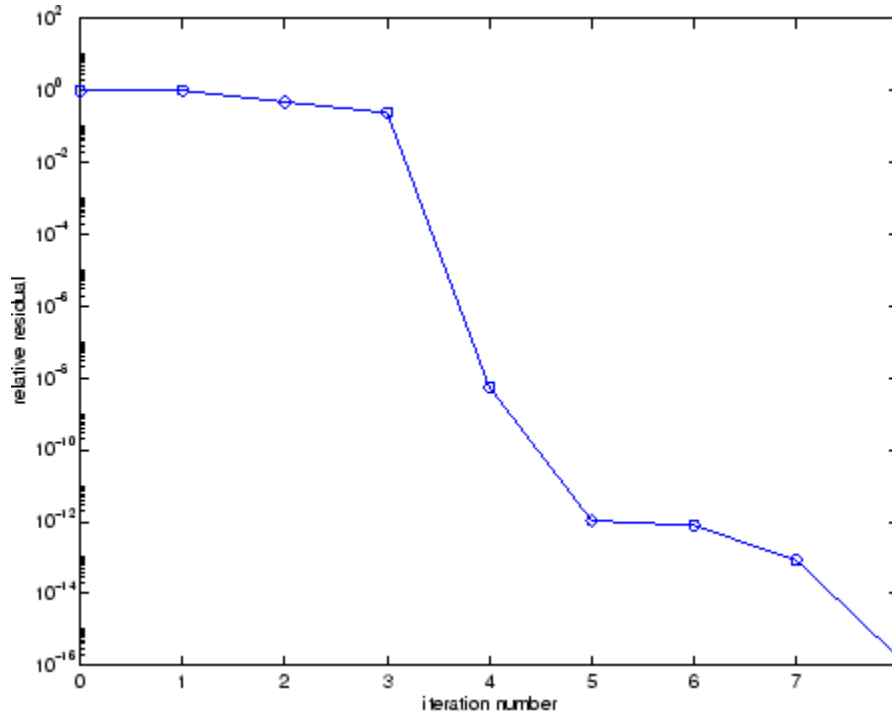
```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = qmr(A,b,1e-6,20,L1,U1)
```

`flag1` is `2` because the upper triangular `U1` has a zero on its diagonal, and `qmr` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` using backslash.

```
[L2,U2] = luinc(A,1e-6);
[x2,flag2,relres2,iter2,resvec2] = qmr(A,b,1e-15,10,L2,U2)
```

`flag2` is `0` because `qmr` converges to the tolerance of `1.6571e-016` (the value of `relres2`) at the eighth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. `resvec2(1) = norm(b)` and `resvec2(9) = norm(b-A*x2)`. You can follow the progress of `qmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number `0`).

```
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```



See Also

`bicg`, `bicgstab`, `cgs`, `gmres`, `lsqr`, `luinc`, `minres`, `pcg`, `symmlq`, `function_handle` (`@`), `mldivide` (`\`)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Freund, Roland W. and Nöel M. Nachtigal, "QMR: A quasi-minimal residual method for non-Hermitian linear systems," *SIAM Journal: Numer. Math.* 60, 1991, pp. 315-339.

Purpose

Orthogonal-triangular decomposition

Syntax

```
[Q,R] = qr(A)       (full and sparse matrices)
[Q,R] = qr(A,0)     (full and sparse matrices)
[Q,R,E] = qr(A)     (full matrices)
[Q,R,E] = qr(A,0)   (full matrices)
X = qr(A)           (full matrices)
R = qr(A)           (sparse matrices)
[C,R] = qr(A,B)     (sparse matrices)
R = qr(A,0)         (sparse matrices)
[C,R] = qr(A,B,0)   (sparse matrices)
```

Description

The `qr` function performs the orthogonal-triangular decomposition of a matrix. This factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of a real complex unitary matrix and an upper triangular matrix.

`[Q,R] = qr(A)` produces an upper triangular matrix `R` of the same dimension as `A` and a unitary matrix `Q` so that `A = Q*R`. For sparse matrices, `Q` is often nearly full. If `[m n] = size(A)`, then `Q` is `m`-by-`m` and `R` is `m`-by-`n`.

`[Q,R] = qr(A,0)` produces an "economy-size" decomposition. If `[m n] = size(A)`, and `m > n`, then `qr` computes only the first `n` columns of `Q` and `R` is `n`-by-`n`. If `m <= n`, it is the same as `[Q,R] = qr(A)`.

`[Q,R,E] = qr(A)` for full matrix `A`, produces a permutation matrix `E`, an upper triangular matrix `R` with decreasing diagonal elements, and a unitary matrix `Q` so that `A*E = Q*R`. The column permutation `E` is chosen so that `abs(diag(R))` is decreasing.

`[Q,R,E] = qr(A,0)` for full matrix `A`, produces an "economy-size" decomposition in which `E` is a permutation vector, so that `A(:,E) = Q*R`. The column permutation `E` is chosen so that `abs(diag(R))` is decreasing.

`X = qr(A)` for full matrix `A`, returns the output of the LAPACK subroutine `DGEQRF` or `ZGEQRF`. `triu(qr(A))` is `R`.

`R = qr(A)` for sparse matrix `A`, produces only an upper triangular matrix, `R`. The matrix `R` provides a Cholesky factorization for the matrix associated with the normal equations,

```
R'*R = A'*A
```

This approach avoids the loss of numerical information inherent in the computation of `A'*A`. It may be preferred to `[Q,R] = qr(A)` since `Q` is always nearly full.

`[C,R] = qr(A,B)` for sparse matrix `A`, applies the orthogonal transformations to `B`, producing `C = Q'*B` without computing `Q`. `B` and `A` must have the same number of rows.

`R = qr(A,0)` and `[C,R] = qr(A,B,0)` for sparse matrix `A`, produce "economy-size" results.

For sparse matrices, the Q-less QR factorization allows the solution of sparse least squares problems

$$\text{minimize}\|Ax - b\|$$

with two steps

```
[C,R] = qr(A,b)
x = R\c
```

If `A` is sparse but not square, MATLAB uses the two steps above for the linear equation solving backslash operator, i.e., `x = A\b`.

Examples

Example 1

Start with

```
A =  [ 1     2     3
       4     5     6
       7     8     9
      10    11    12 ]
```

This is a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization:

```
[Q,R] = qr(A)

Q =

    -0.0776    -0.8331     0.5444     0.0605
    -0.3105    -0.4512    -0.7709     0.3251
    -0.5433    -0.0694    -0.0913    -0.8317
    -0.7762     0.3124     0.3178     0.4461

R =

    -12.8841    -14.5916    -16.2992
           0     -1.0413     -2.0826
           0           0      0.0000
           0           0           0
```

The triangular structure of `R` gives it zeros below the diagonal; the zero on the diagonal in `R(3,3)` implies that `R`, and consequently `A`, does not have full rank.

Example 2

This examples uses matrix `A` from the first example. The QR factorization is used to solve linear systems with more equations than unknowns. For example, let

```
b = [1;3;5;7]
```

The linear system $Ax = b$ represents four equations in only three unknowns. The best solution in a least squares sense is computed by

```
x = A\b
```

which produces

```
Warning: Rank deficient, rank = 2, tol = 1.4594E-014
```

```
x =
    0.5000
         0
    0.1667
```

The quantity `tol` is a tolerance used to decide if a diagonal element of `R` is negligible. If `[Q,R,E] = qr(A)`, then

```
tol = max(size(A))*eps*abs(R(1,1))
```

The solution `x` was computed using the factorization and the two steps

```
y = Q'*b;
x = R\y
```

The computed solution can be checked by forming Ax. This equals b to within roundoff error, which indicates that even though the simultaneous equations $Ax = b$ are overdetermined and rank deficient, they happen to be consistent. There are infinitely many solution vectors `x`; the QR factorization has found just one of them.

Algorithm

Inputs of Type Double

For inputs of type `double`, `qr` uses the LAPACK routines listed in the following table to compute the QR decomposition.

Syntax	Real	Complex
`X = qr(A)` `X = qr(A,0)`	`DGEQRF`	`ZGEQRF`
`[Q,R] = qr(A)` `[Q,R] = qr(A,0)`	`DGEQRF`, `DORGQR`	`ZGEQRF`, `ZUNGQR`
`[Q,R,e] = qr(A)` `[Q,R,e] = qr(A,0)`	`DGEQP3`, `DORGQR`	`ZGEQP3`, `ZUNGQR`

Inputs of Type Single

For inputs of type `single`, `qr` uses the LAPACK routines listed in the following table to compute the QR decomposition.

Syntax	Real	Complex
`R = qr(A)` `R = qr(A,0)`	`SGEQRF`	`CGEQRF`
`[Q,R] = qr(A)` `[Q,R] = qr(A,0)`	`SGEQRF`, `SORGQR`	`CGEQRF`, `CUNGQR`
`[Q,R,e] = qr(A)` `[Q,R,e] = qr(A,0)`	`SGEQP3`, `SORGQR`	`CGEQP3`, `CUNGQR`

See Also

`lu`, `null`, `orth`, `qrdelete`, `qrinsert`, `qrupdate`

The arithmetic operators `\` and `/`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

qrdelete

Purpose

Remove column or row from QR factorization

Syntax

```
[Q1,R1] = qrdelete(Q,R,j)
[Q1,R1] = qrdelete(Q,R,j,'col')
[Q1,R1] = qrdelete(Q,R,j,'row')
```

Description

`[Q1,R1] = qrdelete(Q,R,j)` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the column `A(:,j)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

`[Q1,R1] = qrdelete(Q,R,j,'col')` is the same as `qrdelete(Q,R,j)`.

`[Q1,R1] = qrdelete(Q,R,j,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the row `A(j,:)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
[Q1,R1] = qrdelete(Q,R,j,'row');

Q1 =
    0.5274   -0.5197   -0.6697   -0.0578
    0.7135    0.6911    0.0158    0.1142
    0.3102   -0.1982    0.4675   -0.8037
    0.3413   -0.4616    0.5768    0.5811

R1 =
   32.2335   26.0908   19.9482   21.4063   23.3297
         0  -19.7045  -10.9891    0.4318   -1.4873
         0         0   22.7444    5.8357   -3.1977
         0         0         0  -14.5784    3.7796
```

returns a valid QR factorization, although possibly different from

```
A2 = A;
A2(j,:) = [];
[Q2,R2] = qr(A2)
```

```
Q2 =
    -0.5274    0.5197    0.6697   -0.0578
    -0.7135   -0.6911   -0.0158    0.1142
    -0.3102    0.1982   -0.4675   -0.8037
    -0.3413    0.4616   -0.5768    0.5811

R2 =
   -32.2335  -26.0908  -19.9482  -21.4063  -23.3297
          0   19.7045   10.9891   -0.4318    1.4873
          0         0  -22.7444   -5.8357    3.1977
          0         0         0  -14.5784    3.7796
```

Algorithm

The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

See Also

`planerot`, `qr`, `qrinsert`

Purpose

Insert column or row into QR factorization

Syntax

```
[Q1,R1] = qrinsert(Q,R,j,x)
[Q1,R1] = qrinsert(Q,R,j,x,'col')
[Q1,R1] = qrinsert(Q,R,j,x,'row')
```

Description

`[Q1,R1] = qrinsert(Q,R,j,x)` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with the column `x` inserted before `A(:,j)`. If `A` has `n` columns and `j = n+1`, then `x` is inserted after the last column of `A`.

`[Q1,R1] = qrinsert(Q,R,j,x,'col')` is the same as `qrinsert(Q,R,j,x)`.

`[Q1,R1] = qrinsert(Q,R,j,x,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with an extra row, `x`, inserted before `A(j,:)`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
x = 1:5;
[Q1,R1] = qrinsert(Q,R,j,x,'row')

Q1 =
    0.5231    0.5039   -0.6750    0.1205    0.0411    0.0225
    0.7078   -0.6966    0.0190   -0.0788    0.0833   -0.0150
    0.0308    0.0592    0.0656    0.1169    0.1527   -0.9769
    0.1231    0.1363    0.3542    0.6222    0.6398    0.2104
    0.3077    0.1902    0.4100    0.4161   -0.7264   -0.0150
    0.3385    0.4500    0.4961   -0.6366    0.1761    0.0225

R1 =
   32.4962   26.6801   21.4795   23.8182   26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0   24.4514   11.8132    3.9931
         0         0         0   20.2382   10.3392
```

```
     0     0     0     0  16.1948
     0     0     0     0        0
```

returns a valid QR factorization, although possibly different from

```
A2 = [A(1:j-1,:); x; A(j:end,:)];
[Q2,R2] = qr(A2)

Q2 =
   -0.5231    0.5039    0.6750   -0.1205    0.0411    0.0225
   -0.7078   -0.6966   -0.0190    0.0788    0.0833   -0.0150
   -0.0308    0.0592   -0.0656   -0.1169    0.1527   -0.9769
   -0.1231    0.1363   -0.3542   -0.6222    0.6398    0.2104
   -0.3077    0.1902   -0.4100   -0.4161   -0.7264   -0.0150
   -0.3385    0.4500   -0.4961    0.6366    0.1761    0.0225

R2 =
  -32.4962  -26.6801  -21.4795  -23.8182  -26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0  -24.4514  -11.8132   -3.9931
         0         0         0  -20.2382  -10.3392
         0         0         0         0   16.1948
         0         0         0         0         0
```

Algorithm

The `qrinsert` function inserts the values of `x` into the `j`th column (row) of `R`. It then uses a series of Givens rotations to zero out the nonzero elements of `R` on and below the diagonal in the `j`th column (row).

See Also

`planerot`, `qr`, `qrdelete`

Description

Rank 1 update to QR factorization

Syntax

```
[Q1,R1] = qrupdate(Q,R,u,v)
```

Description

`[Q1,R1] = qrupdate(Q,R,u,v)` when `[Q,R] = qr(A)` is the original QR factorization of `A`, returns the QR factorization of `A + u*v'`, where `u` and `v` are column vectors of appropriate lengths.

Remarks

`qrupdate` works only for full matrices.

Examples

The matrix

```
mu = sqrt(eps)

mu =

   1.4901e-08

A = [ones(1,4); mu*eye(4)];
```

is a well-known example in least squares that indicates the dangers of forming `A'*A`. Instead, we work with the QR factorization – orthonormal Q and upper triangular R.

```
 [Q,R] = qr(A);
```

As we expect, `R` is upper triangular.

```
R =

   -1.0000   -1.0000   -1.0000   -1.0000
         0    0.0000    0.0000    0.0000
         0         0    0.0000    0.0000
         0         0         0    0.0000
         0         0         0         0
```

In this case, the upper triangular entries of `R`, excluding the first row, are on the order of `sqrt(eps)`.

Consider the update vectors

```
u = [-1 0 0 0 0]'; v = ones(4,1);
```

Instead of computing the rather trivial QR factorization of this rank one update to `A` from scratch with

```
[QT,RT] = qr(A + u*v')

QT =

     0     0     0     0     1
    -1     0     0     0     0
     0    -1     0     0     0
     0     0    -1     0     0
     0     0     0    -1     0

RT =

  1.0e-007 *

   -0.1490         0         0         0
         0   -0.1490         0         0
         0         0   -0.1490         0
         0         0         0   -0.1490
         0         0         0         0
```

we may use `qrupdate`.

```
[Q1,R1] = qrupdate(Q,R,u,v)

Q1 =

   -0.0000   -0.0000   -0.0000   -0.0000    1.0000
    1.0000   -0.0000   -0.0000   -0.0000    0.0000
```

```
     0.0000    1.0000   -0.0000   -0.0000    0.0000
     0.0000    0.0000    1.0000   -0.0000    0.0000
    -0.0000   -0.0000   -0.0000    1.0000    0.0000

R1 =

   1.0e-007 *
    0.1490    0.0000    0.0000    0.0000
         0    0.1490    0.0000    0.0000
         0         0    0.1490    0.0000
         0         0         0    0.1490
         0         0         0         0
```

Note that both factorizations are correct, even though they are different.

Algorithm

`qrupdate` uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. `qrupdate` is useful since, if we take `N = max(m,n)`, then computing the new QR factorization from scratch is roughly an $O(N^3)$ algorithm, while simply updating the existing factors in this way is an $O(N^2)$ algorithm.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

`cholupdate`, `qr`

Purpose

Numerically evaluate integral, adaptive Simpson quadrature

Syntax

```
q = quad(fun,a,b)
q = quad(fun,a,b,tol)
q = quad(fun,a,b,tol,trace)
[q,fcnt] = quad(...)
```

Description

Quadrature is a numerical method used to find the area under the graph of a function, that is, to compute a definite integral.

$$q = \int_a^b f(x)\,dx$$

`q = quad(fun,a,b)` tries to approximate the integral of function `fun` from `a` to `b` to within an error of `1e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. Limits `a` and `b` must be finite. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`, the integrand evaluated at each element of `x`.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`q = quad(fun,a,b,tol)` uses an absolute error tolerance `tol` instead of the default which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results. In MATLAB version 5.3 and earlier, the `quad` function used a less reliable algorithm and a default relative tolerance of `1.0e-3`.

`q = quad(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q]` during the recursion.

`[q,fcnt] = quad(...)` returns the number of function evaluations.

The function `quadl` may be more efficient with high accuracies and smooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quadl function may be more efficient than quad at higher accuracies with smooth integrands.
- The quadgk function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, [a,Inf), then for the integral of fun(x) to exist, fun(x) must decay as x approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if fun(x) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint c like log|x-c| or |x-c|^p for p >= -1/2. If the function is singular at points inside (a,b), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

Example

To compute the integral

$$\int_0^2 \frac{1}{x^3 - 2x - 5}\,dx$$

write an M-file function myfun that computes the integrand:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Then pass `@myfun`, a function handle to `myfun`, to `quad`, along with the limits of integration, `0` to `2`:

```
Q = quad(@myfun,0,2)

Q =

   -0.4605
```

Alternatively, you can pass the integrand to `quad` as an anonymous function handle `F`:

```
F = @(x)1./(x.^3-2*x-5);
Q = quad(F,0,2);
```

Algorithm

`quad` implements a low order method using an adaptive recursive Simpson's rule.

Diagnostics

`quad` may issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

See Also

`dblquad`, `quadgk`, `quadl`, `quadv`, `trapz`, `triplequad`, `function_handle` (`@`), "Anonymous Functions"

References

[1] Gander, W. and W. Gautschi, "Adaptive Quadrature – Revisited," BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `http://www.inf.ethz.ch/personal/gander`.

Purpose

Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Syntax

```
q = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b,tol)
[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)
```

Description

`q = quadgk(fun,a,b)` attempts to approximate the integral of a scalar-valued function `fun` from `a` to `b` using high-order global adaptive quadrature and default error tolerances. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`. The integrand evaluated at each element of `x`. `fun` must be a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. Limits `a` and `b` can be `-Inf` or `Inf`. If both are finite, they can be complex. If at least one is complex, the integral is approximated over a straight line path from `a` to `b` in the complex plane.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`[q,errbnd] = quadgk(fun,a,b,tol)` returns an approximate bound on the absolute error, `|Q - I|`, where `I` denotes the exact value of the integral.

`[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)` performs the integration with specified values of optional parameters. The available parameters are

Parameter	Description	
'AbsTol'	Absolute error tolerance. The default value of 'AbsTol' is 1.e-10 (double), 1.e-5 (single).	quadgk attempts to satisfy errbnd <= max(AbsTol,RelTol*|Q|). This is absolute error control when |Q| is sufficiently small and relative error control when |Q| is larger. For pure absolute error control use 'AbsTol' > 0 and 'RelTol'= 0. For pure relative error control use 'AbsTol' = 0. Except when using pure absolute error control, the minimum relative tolerance is 'RelTol' >= 100*eps(class(Q)).
'RelTol'	Relative error tolerance. The default value of 'RelTol' is 1.e-6 (double), 1.e-4 (single).	

Parameter	Description	
`'Waypoints'`	Vector of integration waypoints.	If `fun(x)` has discontinuities in the interval of integration, the locations should be supplied as a `'Waypoints'` vector. When `a`, `b`, and the waypoints are all real, the waypoints must be supplied in strictly increasing or strictly decreasing order, and only the waypoints between `a` and `b` are used. Waypoints are not intended for singularities in `fun(x)`. Singular points should be handled by making them endpoints of separate integrations and adding the results. If `a`, `b`, or any entry of the waypoints vector is complex, the integration is performed over a sequence of straight line paths in the complex plane, from `a` to the first waypoint, from the first waypoint to the second, and so forth, and finally from the last waypoint to `b`.
`'MaxIntervalCount'`	Maximum number of intervals allowed. The default value is `650`.	The `'MaxIntervalCount'` parameter limits the number of intervals that `quadgk` uses at any one time after the first iteration. A warning is issued if `quadgk` returns early because

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quadl function may be more efficient than quad at higher accuracies with smooth integrands.
- The quadgk function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, [a,Inf), then for the integral of fun(x) to exist, fun(x) must decay as x approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if fun(x) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint c like log|x-c| or $|x-c|^p$ for p >= -1/2. If the function is singular at points inside (a,b), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

Examples

Integrand with a singularity at an integration end point

Write an M-file function myfun that computes the integrand:

```
function y = myfun(x)
y = exp(x).*log(x);
```

Then pass `@myfun`, a function handle to `myfun`, to `quadgk`, along with the limits of integration, `0` to `1`:

```
Q = quadgk(@myfun,0,1)

Q =

   -1.3179
```

Alternatively, you can pass the integrand to `quadgk` as an anonymous function handle `F`:

```
F = (@(x)exp(x).*log(x));
Q = quadgk(F,0,1);
```

Oscillatory integrand on a semi-infinite interval

Integrate over a semi-infinite interval with specified tolerances, and return the approximate error bound:

```
[q,errbnd] = quadgk(@(x)x.^5.*exp(-x).*sin(x),0,inf,'RelTol',1e-8,'

q =

  -15.0000


errbnd =

  9.4386e-009
```

Contour integration around a pole

Use `Waypoints` to integrate around a pole using a piecewise linear contour:

```
Q = quadgk(@(z)1./(2*z - 1),-1-i,-1-i,'Waypoints',[1-i,1+i,-1+i])

Q =
```

```
0.0000 + 3.1416i
```

Algorithm

`quadgk` implements adaptive quadrature based on a Gauss-Kronrod pair (15^{th} and 7^{th} order formulas).

Diagnostics

`quadgk` may issue one of the following warnings:

`'Minimum step size reached'` indicates that interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Reached the limit on the maximum number of intervals in use'` indicates that the integration was terminated before meeting the tolerance requirements and that continuing the integration would require more than `MaxIntervalCount` subintervals. The integral may not exist, or it may be difficult to approximate numerically. Increasing `MaxIntervalCount` usually does not help unless the tolerance requirements were nearly met when the integration was previously terminated.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

References

[1] L.F. Shampine "Vectorized Adaptive Quadrature in MATLAB," *Journal of Computational and Applied Mathematics*, to appear.

See Also

`dblquad`, `quadquadl`, `quadv`, `triplequad`, `function_handle` (`@`), "Anonymous Functions"

Purpose

Numerically evaluate integral, adaptive Lobatto quadrature

Syntax

```
q = quadl(fun,a,b)
q = quadl(fun,a,b,tol)
quadl(fun,a,b,tol,trace)
[q,fcnt] = quadl(...)
```

Description

`q = quadl(fun,a,b)` approximates the integral of function `fun` from `a` to `b`, to within an error of 10^{-6} using recursive adaptive Lobatto quadrature. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. `fun` accepts a vector `x` and returns a vector `y`, the function `fun` evaluated at each element of `x`. Limits `a` and `b` must be finite.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`q = quadl(fun,a,b,tol)` uses an absolute error tolerance of `tol` instead of the default, which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results.

`quadl(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a q]` during the recursion.

`[q,fcnt] = quadl(...)` returns the number of function evaluations.

Use array operators `.*`, `./` and `.^` in the definition of `fun` so that it can be evaluated with a vector argument.

The function `quad` may be more efficient with low accuracies or nonsmooth integrands.

The list below contains information to help you determine which quadrature function in `MATLAB` to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

Pass M-file function handle `@myfun` to `quadl`:

```
Q = quadl(@myfun,0,2);
```

where the M-file `myfun.m` is

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Pass anonymous function handle `F` to `quadl`:

```
F = @(x) 1./(x.^3-2*x-5);
Q = quadl(F,0,2);
```

Algorithm

`quadl` implements a high order method using an adaptive Gauss/Lobatto quadrature rule.

Diagnostics

`quadl` may issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

See Also

`dblquad`, `quad`, `quadgk`, `triplequad`, `function_handle` (`@`), "Anonymous Functions"

References

[1] Gander, W. and W. Gautschi, "Adaptive Quadrature – Revisited," BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `http://www.inf.ethz.ch/personal/gander`.

Purpose

Vectorized quadrature

Syntax

```
Q = quadv(fun,a,b)
Q = quadv(fun,a,b,tol)
Q = quadv(fun,a,b,tol,trace)
[Q,fcnt] = quadv(...)
```

Description

`Q = quadv(fun,a,b)` approximates the integral of the complex array-valued function `fun` from `a` to `b` to within an error of `1.e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information. The function `Y = fun(x)` should accept a scalar argument `x` and return an array result `Y`, whose components are the integrands evaluated at `x`. Limits `a` and `b` must be finite.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide addition parameters to the function `fun`, if necessary.

`Q = quadv(fun,a,b,tol)` uses the absolute error tolerance `tol` for all the integrals instead of the default, which is `1.e-6`.

Note The same tolerance is used for all components, so the results obtained with `quadv` are usually not the same as those obtained with `quad` on the individual components.

`Q = quadv(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q(1)]` during the recursion.

`[Q,fcnt] = quadv(...)` returns the number of function evaluations.

The list below contains information to help you determine which quadrature function in `MATLAB` to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Example

For the parameterized array-valued function `myarrayfun`, defined by

```
function Y = myarrayfun(x,n)
Y = 1./((1:n)+x);
```

the following command integrates `myarrayfun`, for the parameter value n = 10 between a = 0 and b = 1:

```
Qv = quadv(@(x)myarrayfun(x,10),0,1);
```

The resulting array `Qv` has 10 elements estimating `Q(k) = log((k+1)./(k))`, for `k = 1:10`.

The entries in `Qv` are slightly different than if you compute the integrals using `quad` in a loop:

```
for k = 1:10
 Qs(k) = quadv(@(x)myscalarfun(x,k),0,1);
end
```

where `myscalarfun` is:

```
function y = myscalarfun(x,k)
y = 1./(k+x);
```

See Also

`quad`, `quadgk`, `quadl`, `dblquad`, `triplequad`, `function_handle` (`@`)

Purpose

Create and open question dialog box

Syntax

```
button = questdlg('qstring')
button = questdlg('qstring','title')
button = questdlg('qstring','title','default')
button = questdlg('qstring','title','str1','str2','default')
button =  questdlg('qstring','title','str1','str2','str3',
   'default')
```

Description

`button = questdlg('qstring')` displays a modal dialog box presenting the question `'qstring'`. The dialog has three default buttons, **Yes**, **No**, and **Cancel**. If the user presses one of these three buttons, `button` is set to the name of the button pressed. If the user presses the close button on the dialog, `button` is set to the empty string. If the user presses the **Return** key, button is set to `'Yes'`. `'qstring'` is a cell array or a string that automatically wraps to fit within the dialog box.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`button = questdlg('qstring','title')` displays a question dialog with `'title'` displayed in the dialog's title bar.

`button = questdlg('qstring','title','default')` specifies which push button is the default in the event that the **Return** key is pressed. `'default'` must be `'Yes'`, `'No'`, or `'Cancel'`.

`button = questdlg('qstring','title','str1','str2','default')` creates a question dialog box with two push buttons labeled `'str1'` and `'str2'`. `'default'` specifies the default button selection and must be `'str1'` or `'str2'`.

```
button =
questdlg('qstring','title','str1','str2','str3','default')
```
creates a question dialog box with three push buttons labeled `'str1'`, `'str2'`, and `'str3'`. `'default'` specifies the default button selection and must be `'str1'`, `'str2'`, or `'str3'`.

In all cases where `'default'` is specified, if `'default'` is not set to one of the button names, pressing the **Enter** key displays a warning and the dialog remains open.

See Also

`dialog`, `errordlg`, `helpdlg`, `inputdlg`, `listdlg`, `msgbox`, `warndlg`

`figure`, `textwrap`, `uiwait`, `uiresume`

“Predefined Dialog Boxes” on page 1-104 for related functions

Purpose

Terminate MATLAB

GUI Alternatives

As an alternative to the quit function, use the Close box or select **File > Exit MATLAB** in the MATLAB desktop.

Syntax

```
quit
quit cancel
quit force
```

Description

`quit` displays a confirmation dialog box if the confirm upon quitting preference is selected, and if confirmed or if the confirmation preference is not selected, terminates MATLAB after running `finish.m`, if `finish.m` exists. The workspace is not automatically saved by `quit`. To save the workspace or perform other actions when quitting, create a `finish.m` file to perform those actions. For example, you can display a custom dialog box to confirm quitting using a `finish.m` file—see the following examples for details. If an error occurs while `finish.m` is running, `quit` is canceled so that you can correct your `finish.m` file without losing your workspace.

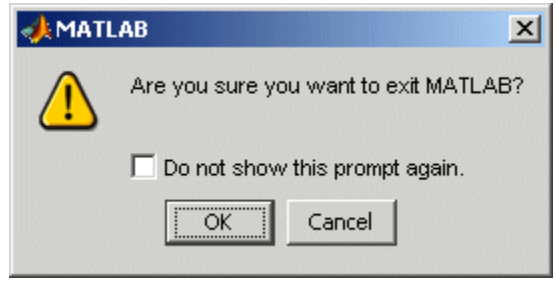
`quit cancel` is for use in `finish.m` and cancels quitting. It has no effect anywhere else.

`quit force` bypasses `finish.m` and terminates MATLAB. Use this to override `finish.m`, for example, if an errant `finish.m` will not let you quit.

Remarks

When using Handle Graphics in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

If you want MATLAB to display the following confirmation dialog box after running quit, select **File > Preferences > General > Confirmation Dialogs**. Then select the check box for Confirm before exiting MATLAB, and click **OK**.



Examples

Two sample `finish.m` files are included with MATLAB. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finishsav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m`—Displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code:

```
button = questdlg('Ready to quit?', ...
        'Exit Dialog','Yes','No','No');
switch button
            case 'Yes',
              disp('Exiting MATLAB');
              %Save variables to matlab.mat
              save
            case 'No',
              quit cancel;
end
```

See Also

`exit`, `finish`, `save`, `startup`

Purpose

Terminate MATLAB server

Syntax

MATLAB Client

```
h.Quit
Quit(h)
invoke(h, 'Quit')
```

Method Signature

```
void Quit(void)
```

Visual Basic Client

```
Quit
```

Description

`Quit` terminates the MATLAB server session to which handle `h` is attached.

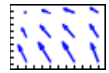
Remarks

Server function names, like `Quit`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Purpose

Quiver or velocity plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
quiver(x,y,u,v)
quiver(u,v)
quiver(...,scale)
quiver(...,LineSpec)
quiver(...,LineSpec,'filled')
quiver(axes_handle,...)
h = quiver(...)
hlines = quiver('v6',...)
```

Description

A quiver plot displays velocity vectors as arrows with components `(u,v)` at the points `(x,y)`.

For example, the first vector is defined by components `u(1),v(1)` and is displayed at the point `x(1),y(1)`.

`quiver(x,y,u,v)` plots vectors as arrows at the coordinates specified in each corresponding pair of elements in `x` and `y`. The matrices `x`, `y`, `u`, and `v` must all be the same size and contain corresponding position and velocity components. However, `x` and `y` can also be vectors, as explained in the next section. By default, the arrows are scaled to just not overlap, but you can scale them to be longer or shorter if you want.

Expanding x- and y-Coordinates

MATLAB expands `x` and `y` if they are not matrices. This expansion is equivalent to calling `meshgrid` to generate matrices from vectors:

```
[x,y] = meshgrid(x,y);
quiver(x,y,u,v)
```

In this case, the following must be true:

`length(x)` = `n` and `length(y)` = `m`, where `[m,n]` = `size(u)` = `size(v)`.

The vector `x` corresponds to the columns of `u` and `v`, and vector `y` corresponds to the rows of `u` and `v`.

`quiver(u,v)` draws vectors specified by `u` and `v` at equally spaced points in the *x-y* plane.

`quiver(...,scale)` automatically scales the arrows to fit within the grid and then stretches them by the factor `scale`. `scale` = `2` doubles their relative length, and `scale` = `0.5` halves the length. Use `scale = 0` to plot the velocity vectors without automatic scaling. You can also tune the length of arrows after they have been drawn by choosing the **Plot Edit**  tool, selecting the `quivergroup` object, opening the Property Editor, and adjusting the **Length** slider.

`quiver(...,LineSpec)` specifies line style, marker symbol, and color using any valid `LineSpec`. `quiver` draws the markers at the origin of the vectors.

`quiver(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

`quiver(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = quiver(...)` returns the handle to the quivergroup object.

Backward-Compatible Version

`hlines = quiver('v6',...)` returns the handles of line objects instead of quivergroup objects for compatibility with MATLAB 6.5 and earlier.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

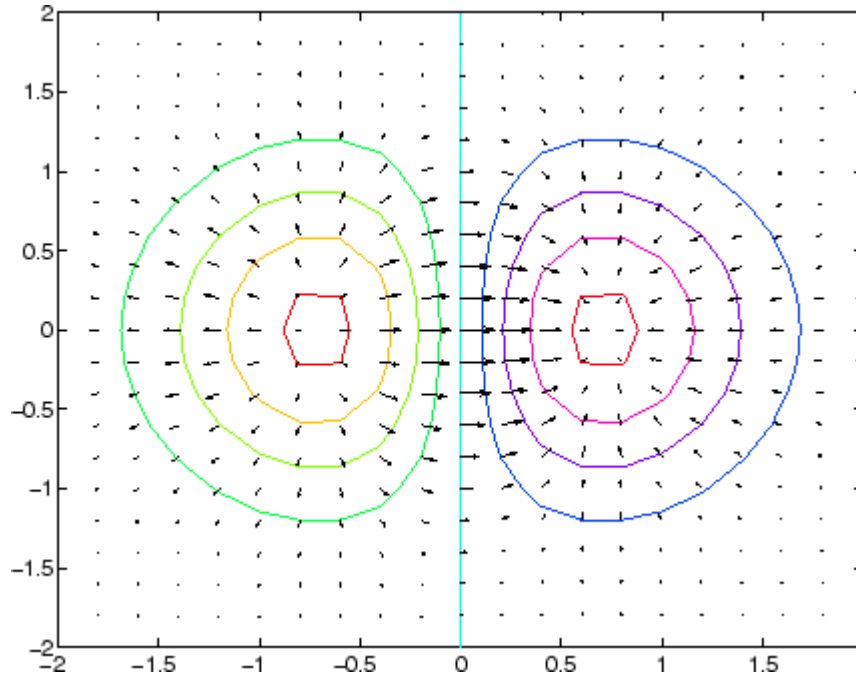
See Plot Objects and Backward Compatibility for more information.

Examples

Showing the Gradient with Quiver Plots

Plot the gradient field of the function $z = xe^{(-x^2 - y^2)}$:

```
[X,Y] = meshgrid(-2:.2:2);
Z = X.*exp(-X.^2 - Y.^2);
[DX,DY] = gradient(Z,.2,.2);
contour(X,Y,Z)
hold on
quiver(X,Y,DX,DY)
colormap hsv
hold off
```



See Also

`contour`, `LineSpec`, `plot`, `quiver3`

"Direction and Velocity Plots" on page 1-89 for related functions

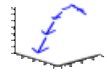
Two-Dimensional Quiver Plots for more examples

`Quivergroup Properties` for property descriptions

quiver3

Purpose

3-D quiver or velocity plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
quiver3(x,y,z,u,v,w)
quiver3(z,u,v,w)
quiver3(...,scale)
quiver3(...,LineSpec)
quiver3(...,LineSpec,'filled')
quiver3(axes_handle,...)
h = quiver3(...)
```

Description

A three-dimensional quiver plot displays vectors with components `(u,v,w)` at the points `(x,y,z)`.

`quiver3(x,y,z,u,v,w)` plots vectors with components `(u,v,w)` at the points `(x,y,z)`. The matrices `x,y,z,u,v,w` must all be the same size and contain the corresponding position and vector components.

`quiver3(z,u,v,w)` plots the vectors at the equally spaced surface points specified by matrix `z`. `quiver3` automatically scales the vectors based on the distance between them to prevent them from overlapping.

`quiver3(...,scale)` automatically scales the vectors to prevent them from overlapping, and then multiplies them by `scale`. `scale = 2` doubles their relative length, and `scale = 0.5` halves them. Use `scale = 0` to plot the vectors without the automatic scaling.

`quiver3(...,LineSpec)` specifies line type and color using any valid `LineSpec`.

`quiver3(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

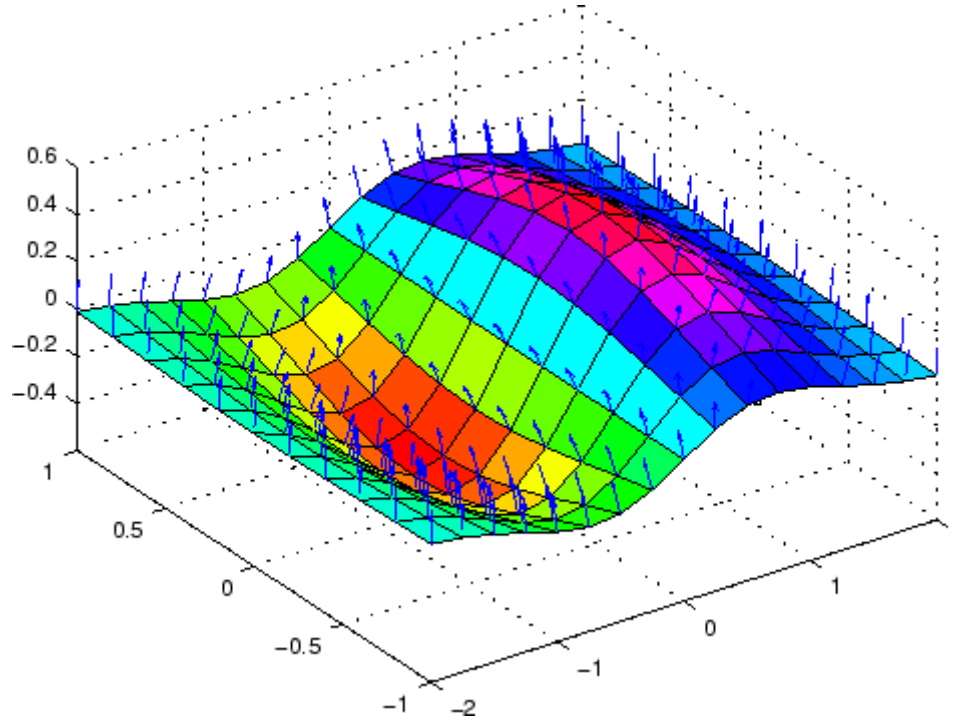
`quiver3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = quiver3(...)` returns a vector of line handles.

Examples

Plot the surface normals of the function $z = xe^{(-x^2 - y^2)}$.

```
[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);
quiver3(X,Y,Z,U,V,W,0.5);
hold on
surf(X,Y,Z);
colormap hsv
view(-35,45)
axis ([-2 2 -1 1 -.6 .6])
hold off
```



See Also

`axis`, `contour`, `LineSpec`, `plot`, `plot3`, `quiver`, `surfnorm`, `view`

"Direction and Velocity Plots" on page 1-89 for related functions

Three-Dimensional Quiver Plots for more examples

Purpose

Define quivergroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default properties for areaseries objects.

See Plot Objects for more information on quivergroup objects.

Quivergroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of quivergroup objects in legends. The `Annotation` property enables you to specify whether this quivergroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the quivergroup object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the quivergroup object in a legend as one entry, but not its children objects
`off`	Do not include the quivergroup or its children in a legend (default)
`children`	Include only the children of the quivergroup as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`AutoScale`
`{on} | off`

Autoscale arrow length. Based on average spacing in the *x* and *y* directions, `AutoScale` scales the arrow length to fit within the grid-defined coordinate data and keeps the arrows from overlapping. After autoscaling, quiver applies the `AutoScaleFactor` to the arrow length.

`AutoScaleFactor`
scalar (default = 0.9)

User-specified scale factor. When `AutoScale` is on, the `quiver` function applies this user-specified autoscale factor to the arrow length. A value of `2` doubles the length of the arrows; `0.5` halves the length.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called

(see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's BeingDeleted property before acting.

BusyAction

cancel | {queue}

Callback routine interruption. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- cancel — Discard the event that attempted to execute a second callback routine.
- queue — Queue the event that attempted to execute a second callback routine until the current callback finishes.

ButtonDownFcn

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.

See the figure's SelectionType property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See "Function Handle Callbacks" for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue

a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

String used by legend for this quivergroup object. The `legend` function uses the string defined by the `DisplayName` property to label this quivergroup object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this quivergroup object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`
`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the

`Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`MaxHeadSize`
scalar (default = 0.2

Maximum size of arrowhead. A value determining the maximum size of the arrowhead relative to the length of the arrow.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this

property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

SelectionHighlight
{on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

ShowArrowHead
{on} | off

Display arrowheads on vectors. When this property is on, MATLAB draws arrowheads on the vectors displayed by quiver. When you set this property to off, quiver draws the vectors as lines without arrowheads.

Tag
string

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define Tag as any string.

For example, you might create an areaseries object and set the Tag property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use findobj to find the object's handle. The following statement changes the FaceColor property of the object whose Tag is area1.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For stem objects, `Type` is `'hggroup'`. This statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`UData`
matrix

One dimension of 2-D or 3-D vector components. `UData`, `VData`, and `WData`, together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components `UData(1)`,`VData(1)`,`WData(1)`.

`UDataSource`
string (MATLAB variable)

Link `UData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `UData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `UData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`VData`
matrix

One dimension of 2-D or 3-D vector components. `UData`, `VData` and `WData` (for 3-D) together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components `UData(1)`,`VData(1)`,`WData(1)`.

`VDataSource`
string (MATLAB variable)

Link `VData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `VData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `VData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`WData`
matrix

One dimension of 2-D or 3-D vector components. `UData`, `VData` and `WData` (for 3-D) together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components `UData(1)`,`VData(1)`,`WData(1)`.

`WDataSource`
string (MATLAB variable)

Link `WData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `WData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `WData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`XData`
vector or matrix

X-axis coordinates of arrows. The `quiver` function draws an individual arrow at each *x*-axis location in the `XData` array.`XData` can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of columns in `UData` or `VData`. That is, `length(XData) == size(UData,2)`.

If you do not specify `XData` (i.e., the input argument `X`), the `quiver` function uses the indices of `UData` to create the quiver graph. See the `XDataMode` property for related information.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the input argument `X`), the `quiver` function sets this property to `manual`.

If you set `XDataMode` to `auto` after having specified `XData`, the `quiver` function resets the *x* tick-mark labels to the indices of the `U`, `V`, and `W` data, overwriting any previous values.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
vector or matrix

Y-axis coordinates of arrows. The quiver function draws an individual arrow at each *y*-axis location in the `YData` array. `YData` can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of rows in `UData` or `VData`. That is, `length(YData) == size(UData,1)`.

If you do not specify `YData` (i.e., the input argument `Y`), the `quiver` function uses the indices of `VData` to create the quiver graph. See the `YDataMode` property for related information.

The input argument `y` in the `quiver` function calling syntax assigns values to `YData`.

`YDataMode`
{auto} | manual

Use automatic or user-specified y-axis values. If you specify `YData` (by setting the `YData` property or specifying the input argument `Y`), MATLAB sets this property to `manual`.

If you set `YDataMode` to auto after having specified `YData`, MATLAB resets the *y* tick-mark labels to the indices of the `U`, `V`, and `W` data, overwriting any previous values.

`YDataSource`
string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`

vector or matrix

Z-axis coordinates of arrows. The quiver function draws an individual arrow at each *z*-axis location in the `ZData` array. `ZData` must be a matrix equal in size to `XData` and `YData`.

The input argument `z` in the `quiver3` function calling syntax assigns values to `ZData`.

Purpose

QZ factorization for generalized eigenvalues

Syntax

```
[AA,BB,Q,Z] = qz(A,B)
[AA,BB,Q,Z,V,W] = qz(A,B)
qz(A,B,flag)
```

Description

The `qz` function gives access to intermediate results in the computation of generalized eigenvalues.

`[AA,BB,Q,Z] = qz(A,B)` for square matrices `A` and `B`, produces upper quasitriangular matrices `AA` and `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA`, and `Q*B*Z = BB`. For complex matrices, `AA` and `BB` are triangular.

`[AA,BB,Q,Z,V,W] = qz(A,B)` also produces matrices `V` and `W` whose columns are generalized eigenvectors.

`qz(A,B,flag)` for real matrices `A` and `B`, produces one of two decompositions depending on the value of `flag`:

`'complex'`	Produces a possibly complex decomposition with a triangular `AA`. For compatibility with earlier versions, `'complex'` is the default.
`'real'`	Produces a real decomposition with a quasitriangular `AA`, containing 1-by-1 and 2-by-2 blocks on its diagonal.

If `AA` is triangular, the diagonal elements of `AA` and `BB`, $\alpha = \mathrm{diag}(AA)$ and $\beta = \mathrm{diag}(BB)$, are the generalized eigenvalues that satisfy

$$A*V*\beta = B*V*\alpha$$
$$\beta*W'*A = \alpha*W'*B$$

The eigenvalues produced by

$$\lambda = \mathrm{eig}(A, B)$$

are the ratios of the αs and βs.

$$\lambda = \alpha./\beta$$

If `AA` is triangular, the diagonal elements of `AA` and `BB`,

```
alpha = diag(AA)
beta = diag(BB)
```

are the generalized eigenvalues that satisfy

```
A*V*diag(beta) = B*V*diag(alpha)
diag(beta)*W'*A = diag(alpha)*W'*B
```

The eigenvalues produced by

```
lambda = eig(A,B)
```

are the element-wise ratios of `alpha` and `beta`.

```
lambda = alpha ./ beta
```

If `AA` is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

Algorithm

For full matrices `A` and `B`, `qz` uses the LAPACK routines listed in the following table.

	A and B Real	**A or B Complex**
`A` and `B` `double`	DGGES, `DTGEVC` (if you request the fifth output `V`)	ZGGES, `ZTGEVC` (if you request the fifth output `V`)
`A` or `B` `single`	SGGES, `STGEVC` (if you request the fifth output `V`)	CGGES, `CTGEVC` (if you request the fifth output `V`)

See Also

`eig`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Uniformly distributed pseudorandom numbers

Syntax

```
Y = rand
Y = rand(n)
Y = rand(m,n)
Y = rand([m n])
Y = rand(m,n,p,...)
Y = rand([m n p...])
Y = rand(size(A))
rand(method,s)
s = rand(method)
```

Description

`Y = rand` returns a pseudorandom, scalar value drawn from a uniform distribution on the unit interval.

`Y = rand(n)` returns an `n`-by-`n` matrix of values derived as described above.

`Y = rand(m,n)` or `Y = rand([m n])` returns an `m`-by-`n` matrix of the same.

`Y = rand(m,n,p,...)` or `Y = rand([m n p...])` generates an `m`-by-`n`-by-`p`-by-`...` array of the same.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`Y = rand(size(A))` returns an array that is the same size as `A`.

`rand(method,s)` causes `rand` to use the generator determined by `method`, and initializes the state of that generator using the value of `s`.

The value of `s` is dependent upon which `method` is selected. If `method` is set to `'state'` or `'twister'`, then `s` must be either a scalar integer value from `0` to `2^32-1` or the output of `rand(method)`. If `method` is set to `'seed'`, then `s` must be either a scalar integer value from `0` to `2^31-2` or the output of `rand(method)`.

The rand and randn generators each maintain their own internal state information. Initializing the state of one has no effect on the other.

Input argument method can be any of the strings shown in the table below:

method	Description
'twister'	Use the Mersenne Twister algorithm by Nishimura and Matsumoto (the default in MATLAB Versions 7.4 and later). This method generates double-precision values in the closed interval [2^(-53), 1-2^(-53)], with a period of (2^19937-1)/2.
'state'	Use a modified version of Marsaglia's *subtract with borrow* algorithm (the default in MATLAB versions 5 through 7.3). This method can generate all the double-precision values in the closed interval [2^(-53), 1-2^(-53)]. It theoretically can generate over 2^1492 values before repeating itself.
'seed'	Use a multiplicative congruential algorithm (the default in MATLAB version 4). This method generates double-precision values in the closed interval [1/(2^31-1), 1-1/(2^31-1)], with a period of 2^31-2.

For a full description of the Mersenne twister algorithm, see

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

s = rand(method) returns in s the current internal state of the generator selected by method. It does not change the generator being used.

Remarks

The sequence of numbers produced by rand is determined by the internal state of the generator. Setting the generator to the same fixed state enables you to repeat computations. Setting the generator to different states leads to unique computations. It does not, however, improve statistical properties.

Because MATLAB resets the `rand` state at startup, `rand` generates the same sequence of numbers in each session unless you change the value of the `state` input.

Examples

Example 1

Make a random choice between two equally probable alternatives:

```
if rand < .5
   'heads'
else
   'tails'
end
```

Example 2

Generate a 3-by-4 pseudorandom matrix:

```
R = rand(3,4)
R =

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

Example 3

Set `rand` to its default initial state:

```
rand('twister', 5489);
```

Initialize `rand` to a different state each time:

```
rand('twister', sum(100*clock));
```

Save the current state, generate 100 values, reset the state, and repeat the sequence:

```
s = rand('twister');
u1 = rand(100);
rand('twister',s);
```

```
u2 = rand(100); % contains exactly the same values as u1
```

Example 4

Generate uniform integers on the set `1:n`:

```
n = 75;
f = ceil(n.*rand(100,1));

f(1:10)
ans =

    72
    37
    61
    11
    32
    69
    60
    72
    50
     3
```

Example 5

Generate a uniform distribution of random numbers on a specified interval `[a,b]`. To do this, multiply the output of `rand` by `(b-a)`, then add `a`. For example, to generate a 5-by-5 array of uniformly distributed random numbers on the interval `[10,50]`,

```
a = 10; b = 50;
x = a + (b-a) * rand(5)
x =
   19.1591   49.8454   10.1854   25.9913   17.2739
   46.5335   13.1270   40.9964   20.3948   20.5521
   16.0951   27.7071   42.6921   42.0027   15.8216
   43.0327   14.2661   44.7478   27.2566   15.4427
   31.5337   48.4759   13.3774   46.4259   44.7717
```

References

[1] Moler, C.B., "Numerical Computing with MATLAB," SIAM, (2004), 336 pp. Available online at `http://www.mathworks.com/moler`.

[2] G. Marsaglia and A. Zaman "A New Class of Random Number Generators," Annals of Applied Probability, (1991), 3:462-480.

[3] Matsumoto, M. and Nishimura, T. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator," ACM Transactions on Modeling and Computer Simulation, (1998), 8(1):3-30.

[4] Park, S.K. and Miller, K.W. "Random Number Generators: Good Ones Are Hard to Find," Communications of the ACM, (1988), 31(10):1192-1201

See Also

`randn`, `randperm`, `sprand`, `sprandn`

Purpose

Normally distributed random numbers

Syntax

```
Y = randn
Y = randn(n)
Y = randn(m,n)
Y = randn([m n])
Y = randn(m,n,p,...)
Y = randn([m n p...])
Y = randn(size(A))
randn(method,s)
s = randn(method)
```

Description

`Y = randn` returns a pseudorandom, scalar value drawn from a normal distribution with mean 0 and standard deviation 1.

`Y = randn(n)` returns an `n`-by-`n` matrix of values derived as described above.

`Y = randn(m,n)` or `Y = randn([m n])` returns an `m`-by-`n` matrix of the same.

`Y = randn(m,n,p,...)` or `Y = randn([m n p...])` generates an `m`-by-`n`-by-`p`-by-`...` array of the same.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`Y = randn(size(A))` returns an array that is the same size as `A`.

`randn(method,s)` causes `randn` to use the generator determined by `method`, and initializes the state of that generator using the value of `s`.

The value of `s` is dependent upon which `method` is selected. If `method` is set to `'state'`, then `s` must be either a scalar integer value from `0` to `2^32-1` or the output of `rand(method)`. If `method` is set to `'seed'`, then `s` must be either a scalar integer value from `0` to `2^31-2` or the

output of `rand(method)`. To set the generator to its default initial state, set `s` equal to zero.

The `randn` and `rand` generators each maintain their own internal state information. Initializing the state of one has no effect on the other.

Input argument `method` can be either of the strings shown in the table below:

method	Description
`'state'`	Use Marsaglia's ziggurat algorithm (the default in MATLAB versions 5 and later). The period is approximately 2^64.
`'seed'`	Use the polar algorithm (the default in MATLAB version 4). The period is approximately `(2^31-1)*(pi/8)`.

`s = randn(method)` returns in `s` the current internal state of the generator selected by `method`. It does not change the generator being used.

Examples

Example 1

`R = randn(3,4)` might produce

```
R =
    1.1650    0.3516    0.0591    0.8717
    0.6268   -0.6965    1.7971   -1.4462
    0.0751    1.6961    0.2641   -0.7012
```

For a histogram of the `randn` distribution, see `hist`.

Example 2

Set `randn` to its default initial state:

```
randn('state', 0);
```

Initialize `randn` to a different state each time:

```
randn('state', sum(100*clock));
```

Save the current state, generate 100 values, reset the state, and repeat the sequence:

```
s = randn('state');
u1 = randn(100);
randn('state',s);
u2 = randn(100);      % Contains exactly the same values as u1.
```

Example 3

Generate a random distribution with a specific mean and variance σ^2. To do this, multiply the output of `randn` by the standard deviation σ, and then add the desired mean. For example, to generate a 5-by-5 array of random numbers with a mean of .6 that are distributed with a variance of 0.1,

```
x = .6 + sqrt(0.1) * randn(5)
x =

    0.8713    0.4735    0.8114    0.0927    0.7672
    0.9966    0.8182    0.9766    0.6814    0.6694
    0.0960    0.8579    0.2197    0.2659    0.3085
    0.1443    0.8251    0.5937    1.0475   -0.0864
    0.7806    1.0080    0.5504    0.3454    0.5813
```

References

[1] Moler, C.B., "Numerical Computing with MATLAB," SIAM, (2004), 336 pp. Available online at `http://www.mathworks.com/moler`.

[2] Marsaglia, G. and Tsang, W.W., The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, (2000), 5(8). Available online at `http://www.jstatsoft.org/v05/i08/`.

[3] Marsaglia, G. and Tsang, W.W., "A Fast, Easily Implemented Method for Sampling from Decreasing or Symmetric Unimodal Density Functions," *SIAM Journal of Scientific and Statistical Computing*, (1984), 5(2):349-359.

[4] Knuth, D.E., “Seminumerical Algorithms,” Volume 2 of *The Art of Computer Programming*, 3rd edition Addison-Wesley (1998).

See Also

`rand`, `randperm`, `sprand`, `sprandn`

randperm

Purpose

Random permutation

Syntax

```
p = randperm(n)
```

Description

`p = randperm(n)` returns a random permutation of the integers `1:n`.

Remarks

The `randperm` function calls `rand` and therefore, changes `rand`'s state.

Examples

`randperm(6)` might be the vector

```
   [3  2  6  4  1  5]
```

or it might be some other permutation of `1:6`.

See Also

`permute`

Purpose

Rank of matrix

Syntax

```
k = rank(A)
k = rank(A,tol)
```

Description

The `rank` function provides an estimate of the number of linearly independent rows or columns of a full matrix.

`k = rank(A)` returns the number of singular values of `A` that are larger than the default tolerance, `max(size(A))*eps(norm(A))`.

`k = rank(A,tol)` returns the number of singular values of `A` that are larger than `tol`.

Remark

Use `sprank` to determine the structural rank of a sparse matrix.

Algorithm

There are a number of ways to compute the rank of a matrix. MATLAB uses the method based on the singular value decomposition, or SVD. The SVD algorithm is the most time consuming, but also the most reliable.

The `rank` algorithm is

```
s = svd(A);
tol = max(size(A))*eps(max(s));
r = sum(s > tol);
```

See Also

`sprank`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Rational fraction approximation

Syntax

```
[N,D] = rat(X)
[N,D] = rat(X,tol)
rat(X)
S = rats(X,strlen)
S = rats(X)
```

Description

Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. The `rat` function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The `rats` function calls `rat`, and returns strings.

`[N,D] = rat(X)` returns arrays `N` and `D` so that `N./D` approximates `X` to within the default tolerance, `1.e-6*norm(X(:),1)`.

`[N,D] = rat(X,tol)` returns `N./D` approximating `X` to within `tol`.

`rat(X)`, with no output arguments, simply displays the continued fraction.

`S = rats(X,strlen)` returns a string containing simple rational approximations to the elements of `X`. Asterisks are used for elements that cannot be printed in the allotted space, but are not negligible compared to the other elements in `X`. `strlen` is the length of each string element returned by the `rats` function. The default is `strlen = 13`, which allows 6 elements in 78 spaces.

`S = rats(X)` returns the same results as those printed by MATLAB with `format rat`.

Examples

Ordinarily, the statement

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7
```

produces

```
s =
```

```
    0.7595
```

However, with

```
format rat
```

or with

```
rats(s)
```

the printed result is

```
s =
   319/420
```

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity `s` is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement `rat(s)` produces

```
1 + 1/(-4 + 1/(-6 + 1/(-3 + 1/(-5))))
```

And the statement

```
[n,d] = rat(s)
```

produces

```
n = 319, d = 420
```

The mathematical quantity π is certainly not a rational number, but the MATLAB quantity `pi` that approximates it is a rational number. `pi` is the ratio of a large integer and 2^{52}:

```
14148475504056880/4503599627370496
```

However, this is not a simple rational number. The value printed for `pi` with `format rat`, or with `rats(pi)`, is

```
355/113
```

This approximation was known in Euclid's time. Its decimal representation is

```
3.14159292035398
```

and so it agrees with `pi` to seven significant figures. The statement

```
rat(pi)
```

produces

```
3 + 1/(7 + 1/(16))
```

This shows how the `355/113` was obtained. The less accurate, but more familiar approximation `22/7` is obtained from the first two terms of this continued fraction.

Algorithm

The `rat(X)` function approximates each element of `X` by a continued fraction of the form

$$\frac{n}{d} = d_1 + \frac{1}{d_2 + \frac{1}{\left(d_3 + \ldots + \frac{1}{d_k}\right)}}$$

The ds are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when `X = sqrt(2)`. For `x = sqrt(2)` , the error with `k` terms is about `2.68*(.173)^k`, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

See Also

format

rbbox

Purpose

Create rubberband box for area selection

Syntax

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

Description

`rbbox` initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure's `CurrentPoint`, and begins tracking from this point.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` figure units, `rbbox` redraws the rubberband box. The default stepsize is `1`.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the *x* and *y* components of the lower left corner of the box, and `width` and `height` are the dimensions of the box.

Remarks

`rbbox` is useful for defining and resizing a rectangular region:

- For box definition, `initialRect` is `[x y 0 0]`, where `(x,y)` is the figure's `CurrentPoint`.
- For box resizing, `initialRect` defines the rectangular region that you resize (e.g., a legend). `fixedPoint` is the corner diametrically opposite the tracking point.

`rbbox` returns immediately if a button is not currently pressed. Therefore, you use `rbbox` with `waitforbuttonpress` so that the mouse button is down when `rbbox` is called. `rbbox` returns when you release the mouse button.

Examples

Assuming the current view is `view(2)`, use the current axes' `CurrentPoint` property to determine the extent of the rectangle in dataspace units:

```
k = waitforbuttonpress;
point1 = get(gca,'CurrentPoint');    % button down detected
finalRect = rbbox;                   % return figure units
point2 = get(gca,'CurrentPoint');    % button up detected
point1 = point1(1,1:2);              % extract x and y
point2 = point2(1,1:2);
p1 = min(point1,point2);             % calculate locations
offset = abs(point1-point2);         % and dimensions
x = [p1(1) p1(1)+offset(1) p1(1)+offset(1) p1(1) p1(1)];
y = [p1(2) p1(2) p1(2)+offset(2) p1(2)+offset(2) p1(2)];
hold on
axis manual
plot(x,y)                            % redraw in dataspace units
```

See Also

`axis`, `dragrect`, `waitforbuttonpress`

"View Control" on page 1-99 for related functions

Purpose

Matrix reciprocal condition number estimate

Syntax

```
c = rcond(A)
```

Description

`c = rcond(A)` returns an estimate for the reciprocal of the condition of `A` in 1-norm using the LAPACK condition estimator. If `A` is well conditioned, `rcond(A)` is near 1.0. If `A` is badly conditioned, `rcond(A)` is near 0.0. Compared to `cond`, `rcond` is a more efficient, but less reliable, method of estimating the condition of a matrix.

Algorithm

For full matrices `A`, `rcond` uses the LAPACK routines listed in the following table to compute the estimate of the reciprocal condition number.

	Real	Complex
`A double`	DLANGE, DGETRF, DGECON	ZLANGE, ZGETRF, ZGECON
`A single`	SLANGE, SGETRF, SGECON	CLANGE, CGETRF, CGECON

See Also

`cond`, `condest`, `norm`, `normest`, `rank`, `svd`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Read video frame data from multimedia reader object

Syntax

```
video = read(obj)
video = read(obj, index)
```

Description

`video = read(obj)` reads in video frames from the associated file. `video` is an H-by-W-by-B-by-F matrix where H is the image frame height, W is the image frame width, B is the number of bands in the image (e.g., 3 for RGB), and F is the number of frames read in. The default behavior is to read in all frames unless an index is specified. The type of data returned is always `UINT8` data representing RGB24 video frames.

`video = read(obj, index)` performs the same operation, but reads only the frame(s) specified by `index`, where the first frame number is 1. `index` can be a single index, or a two-element array representing an index range of the video stream.

For example, read only the first frame:

```
video = read(obj, 1);
```

Read the first 10 frames:

```
video = read(obj, [1 10]);
```

You can use `Inf` to represent the last frame in the file:

```
video = read(obj, Inf);
```

Read from frame 50 through the end of the file:

```
video = read(obj, [50 Inf]);
```

If an invalid `index` is specified, MATLAB throws an error.

Examples

Construct a multimedia reader object associated with file `xylophone.mpg` and with the user `tag` property set to `'myreader1'`.

```
readerobj = mmreader('xylophone.mpg', 'tag', 'myreader1');
```

Read in all video frames from the file.

```
vidFrames = read(readerobj);
```

Determine the number of frames in the file.

```
numFrames = get(readerobj, 'NumberOfFrames');
```

Create a MATLAB movie struct from the video frames.

```
for k = 1 : numFrames
   mov(k).cdata = vidFrames(:,:,:,k);
   mov(k).colormap = [];
end
```

Create a figure.

```
hf = figure;
```

Resize the figure based on the video's width and height.

```
set(hf, 'position', [150 150 readerobj.Width readerobj.Height])
```

Play back the movie once at the video's frame rate.

```
movie(hf, mov, 1, readerobj.FrameRate);
```

See Also

`get`, `mmreader`, `movie`, `set`

Purpose

Read data asynchronously from device

Syntax

```
readasync(obj)
readasync(obj,size)
```

Arguments

`obj`	A serial port object.
`size`	The number of bytes to read from the device.

Description

`readasync(obj)` initiates an asynchronous read operation.

`readasync(obj,size)` asynchronously reads, at most, the number of bytes given by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Remarks

Before you can read data, you must connect `obj` to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

You should use `readasync` only when you configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute an M-file callback function when the terminator or the specified amount of data is read.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled (if `size` is not specified).

Because `readasync` checks for the terminator, this function can be slow. To increase speed, you might want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the device.

Example

This example creates the serial port object `s`, connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
ans =
     15
out = fscanf(s)
```

```
out =
2.0399999619E0
fclose(s)
```

See Also

Functions

fopen, stopasync

Properties

BytesAvailable, BytesAvailableFcn, ReadAsyncMode, Status, TransferStatus

real

Purpose

Real part of complex number

Syntax

```
X = real(Z)
```

Description

`X = real(Z)` returns the real part of the elements of the complex array `Z`.

Examples

`real(2+3*i)` is 2.

See Also

`abs`, `angle`, `conj`, `i`, `j`, `imag`

Purpose

Natural logarithm for nonnegative real arrays

Syntax

```
Y = reallog(X)
```

Description

`Y = reallog(X)` returns the natural logarithm of each element in array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

```
M = magic(4)

M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

reallog(M)

ans =
    2.7726    0.6931    1.0986    2.5649
    1.6094    2.3979    2.3026    2.0794
    2.1972    1.9459    1.7918    2.4849
    1.3863    2.6391    2.7081         0
```

See Also

`log`, `realpow`, `realsqrt`

realmax

Purpose

Largest positive floating-point number

Syntax

```
n = realmax
```

Description

`n = realmax` returns the largest floating-point number representable on your computer. Anything larger overflows.

`realmax('double')` is the same as `realmax` with no arguments.

`realmax('single')` is the largest single precision floating point number representable on your computer. Anything larger overflows to `single(Inf)`.

Examples

`realmax` is one bit less than 2^{1024} or about `1.7977e+308`.

Algorithm

The `realmax` function is equivalent to `pow2(2-eps,maxexp)`, where `maxexp` is the largest possible floating-point exponent.

Execute `type realmax` to see `maxexp` for various computers.

See Also

`eps`, `realmin`, `intmax`

Purpose

Smallest positive normalized floating-point number

Syntax

```
n = realmin
```

Description

`n = realmin` returns the smallest positive normalized floating-point number on your computer. Anything smaller underflows or is an IEEE "denormal."

`REALMIN('double')` is the same as `REALMIN` with no arguments.

`REALMIN('single')` is the smallest positive normalized single precision floating point number on your computer.

Examples

`realmin` is `2^(-1022)` or about `2.2251e-308`.

Algorithm

The `realmin` function is equivalent to `pow2(1,minexp)` where `minexp` is the smallest possible floating-point exponent.

Execute `type realmin` to see `minexp` for various computers.

See Also

`eps`, `realmax`, `intmin`

realpow

Purpose

Array power for real-only output

Syntax

```
Z = realpow(X,Y)
```

Description

`Z = realpow(X,Y)` raises each element of array `X` to the power of its corresponding element in array `Y`. Arrays `X` and `Y` must be the same size. The range of `realpow` is the set of all real numbers, i.e., all elements of the output array `Z` must be real.

Examples

```
X = -2*ones(3,3)

X =
     -2     -2     -2
     -2     -2     -2
     -2     -2     -2

Y = pascal(3)

ans =
      1      1      1
      1      2      3
      1      3      6

realpow(X,Y)

ans =
     -2     -2     -2
     -2      4     -8
     -2     -8     64
```

See Also

`reallog`, `realsqrt`, `.^` (array power operator)

Purpose

Square root for nonnegative real arrays

Syntax

```
Y = realsqrt(X)
```

Description

`Y = realsqrt(X)` returns the square root of each element of array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

```
M = magic(4)

M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

realsqrt(M)

ans =
    4.0000    1.4142    1.7321    3.6056
    2.2361    3.3166    3.1623    2.8284
    3.0000    2.6458    2.4495    3.4641
    2.0000    3.7417    3.8730    1.0000
```

See Also

`reallog`, `realpow`, `sqrt`, `sqrtm`

record

Purpose

Record data and event information to file

Syntax

```
record(obj)
record(obj,'switch')
```

Arguments

`obj`	A serial port object.
`'switch'`	Switch recording capabilities on or off.

Description

`record(obj)` toggles the recording state for `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Remarks

Before you can record information to disk, `obj` must be connected to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to Debugging: Recording Information to Disk.

Example

This example creates the serial port object `s`, connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
```

```
s.RecordName = 'MySerialFile.txt';
record(s,'on')
fprintf(s,'*IDN?')
out = fscanf(s);
record(s,'off')
fclose(s)
```

See Also

Functions

`fclose`, `fopen`

Properties

`RecordDetail`, `RecordMode`, `RecordName`, `RecordStatus`, `Status`

rectangle

Purpose

Create 2-D rectangle object

Syntax

Description

`rectangle` draws a rectangle with `Position [0,0,1,1]` and `Curvature [0,0]` (i.e., no curvature).

`rectangle('Position',[x,y,w,h])` draws the rectangle from the point `x,y` and having a width of `w` and a height of `h`. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the x and y axes. You can do this with the command `axis equal` or `daspect([1,1,1])`.

`rectangle(...,'Curvature',[x,y])` specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`h = rectangle(...)` returns the handle of the rectangle object created.

Remarks

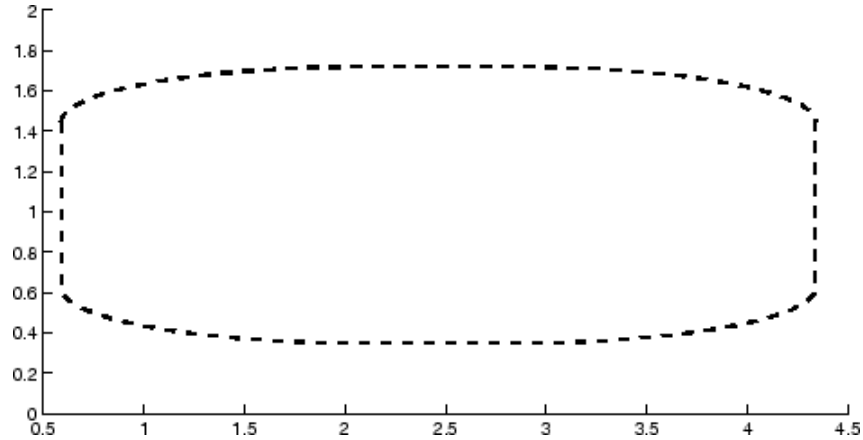
Rectangle objects are 2-D and can be drawn in an axes only if the view is `[0 90]` (i.e., `view(2)`). Rectangles are children of axes and are defined in coordinates of the axes data.

Examples

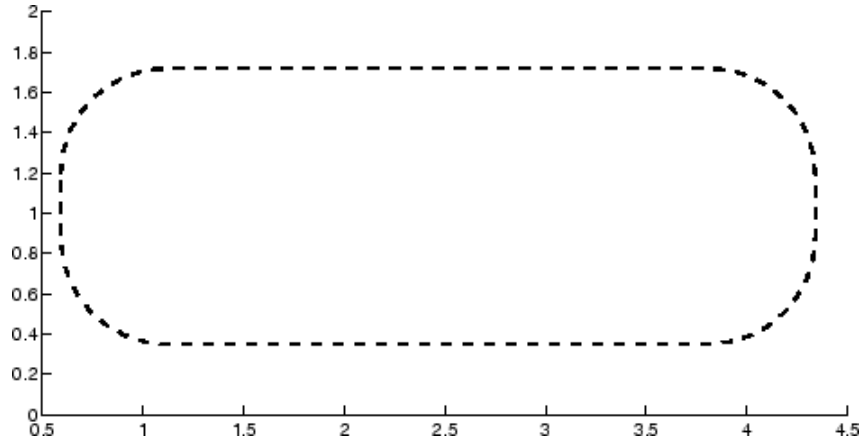
This example sets the data aspect ratio to `[1,1,1]` so that the rectangle is displayed in the specified proportions (`daspect`). Note that the

horizontal and vertical curvature can be different. Also, note the effects of using a single value for `Curvature`.

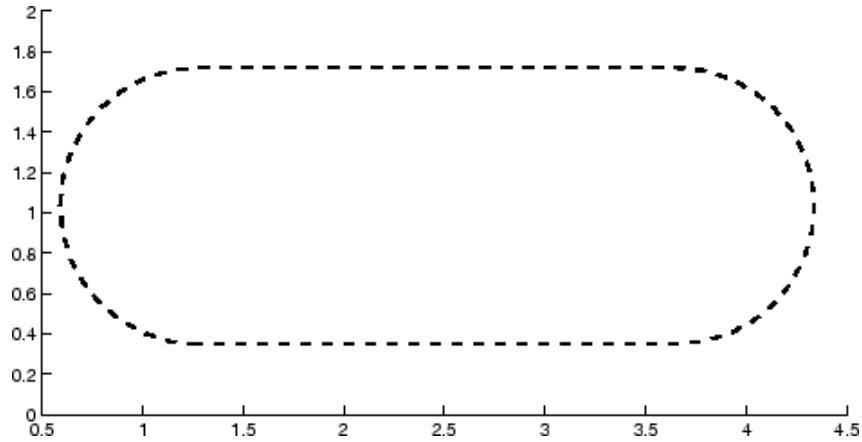
```
rectangle('Position',[0.59,0.35,3.75,1.37],...
          'Curvature',[0.8,0.4],...
         'LineWidth',2,'LineStyle','--')
daspect([1,1,1])
```



Specifying a single value of `[0.4]` for `Curvature` produces



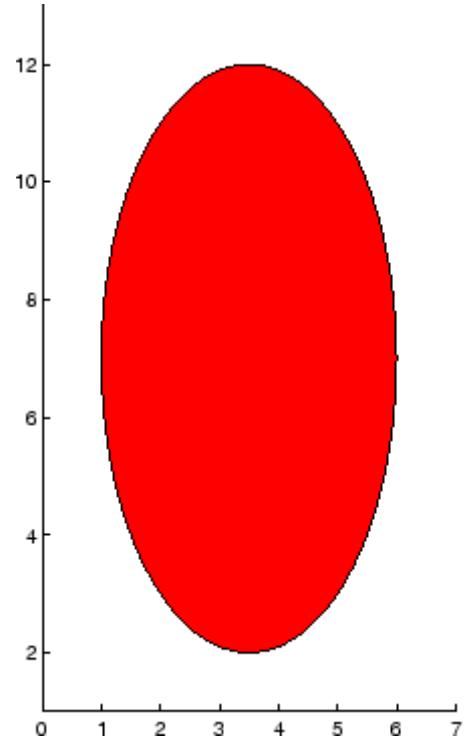
A `Curvature` of `[1]` produces a rectangle with the shortest side completely round:



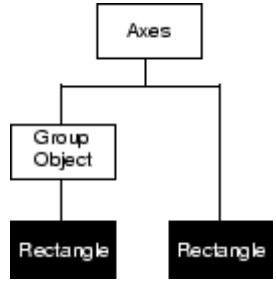
This example creates an ellipse and colors the face red.

```
rectangle('Position',[1,2,5,10],'Curvature',[1,1],...
          'FaceColor','r')
daspect([1,1,1])
xlim([0,7])
```

```
ylim([1,13])
```



Object Hierarchy



Setting Default Properties

You can set default rectangle properties on the axes, figure, and root levels:

```
set(0,'DefaultRectangleProperty',PropertyValue...)
set(gcf,'DefaultRectangleProperty',PropertyValue...)
set(gca,'DefaultRectangleProperty',PropertyValue...)
```

where *Property* is the name of the rectangle property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.

See Also

`line`, `patch`, rectangle properties

"Object Creation Functions" on page 1-94 for related functions

See the `annotation` function for information about the rectangle annotation object.

`Rectangle Properties` for property descriptions

Purpose

Define rectangle properties

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Rectangle Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of rectangle objects in legends. The `Annotation` property enables you to specify whether this rectangle object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the rectangle object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this rectangle object in a legend (default)
`off`	Do not include this rectangle object in a legend
`children`	Same as `on` because rectangle objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` read only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted,

and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the rectangle object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property)

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a rectangle object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`Children`

vector of handles

The empty matrix; rectangle objects have no children.

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips rectangles to the axes plot box by default. If you set `Clipping` to `off`, rectangles are displayed outside the axes plot box. This can occur if you create a rectangle,

set `hold` to `on`, freeze axis scaling (`axis` set to `manual`), and then create a larger rectangle.

`CreateFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during object creation. This property defines a callback function that executes when MATLAB creates a rectangle object. You must define this property as a default value for rectangles or in a call to the `rectangle` function to create a new rectangle object. For example, the statement

```
set(0,'DefaultRectangleCreateFcn',@rect_create)
```

defines a default value for the rectangle `CreateFcn` property on the root level that sets the axes `DataAspectRatio` whenever you create a rectangle object. The callback function must be on your MATLAB path when you execute the above statement.

```
function rect_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 axh = get(src,'Parent');
 set(axh,'DataAspectRatio',[1,1,1]))
end
```

MATLAB executes this function after setting all rectangle properties. Setting this property on an existing rectangle object has no effect. The function must define at least two input arguments (handle of object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`Curvature`

one- or two-element vector `[x,y]`

Amount of horizontal and vertical curvature. This property specifies the curvature of the rectangle sides, which enables the shape of the rectangle to vary from rectangular to ellipsoidal. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`DeleteFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Delete rectangle callback function. A callback function that executes when you delete the rectangle object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
```

```
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DisplayName`

string (default is empty string)

String used by legend for this rectangle object. The `legend` function uses the string defined by the `DisplayName` property to label this rectangle object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this rectangle object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.

- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

`EdgeColor`
`{ColorSpec} | none`

Color of the rectangle edges. This property specifies the color of the rectangle edges as a color or specifies that no edges be drawn.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase rectangle objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the rectangle when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the rectangle by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the rectangle. However, the rectangle’s color depends on the color of whatever is beneath it on the display.
- `background` — Erase the rectangle by drawing it in the axes background `Color`, or the figure background `Color` if the axes

`Color` is set to `none`. This damages objects that are behind the erased rectangle, but rectangles are always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing nonnormal mode objects.

`FaceColor`

`ColorSpec | {none}`

Color of rectangle face. This property specifies the color of the rectangle face, which is not colored by default.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from

the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the Root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

`HitTest`

```
{on} | off
```

Selectable by mouse click. `HitTest` determines if the rectangle can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on

the rectangle. If `HitTest` is `off`, clicking the rectangle selects the object below it (which may be the axes containing it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a rectangle callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine.

`LineStyle`
`{-} | -- | :  | -.  | none`

Line style of rectangle edge. This property specifies the line style of the edges. The available line styles are

Symbol	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

`LineWidth`
scalar

The width of the rectangle edge line. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of rectangle object. This property contains the handle of the rectangle object's parent. The parent of a rectangle object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Position`
four-element vector `[x,y,width,height]`

Location and size of rectangle. This property specifies the location and size of the rectangle in the data units of the axes. The point defined by `x`, `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the *x*-and *y*-axes respectively.

`Selected`
`on | off`

Is object selected? When this property is `on` MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as

global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`
string (read only)

Class of graphics object. For rectangle objects, `Type` is always the string `'rectangle'`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the rectangle. Assign this property the handle of a uicontextmenu object created in the same figure as the rectangle. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the rectangle.

`UserData`
matrix

User-specified data. Any data you want to associate with the rectangle object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Rectangle visibility. By default, all rectangles are visible. When set to `off`, the rectangle is not visible, but still exists, and you can `get` and `set` its properties.

rectint

Purpose

Rectangle intersection area

Syntax

```
area = rectint(A,B)
```

Description

`area = rectint(A,B)` returns the area of intersection of the rectangles specified by position vectors `A` and `B`.

If `A` and `B` each specify one rectangle, the output `area` is a scalar.

`A` and `B` can also be matrices, where each row is a position vector. `area` is then a matrix giving the intersection of all rectangles specified by `A` with all the rectangles specified by `B`. That is, if `A` is `n`-by-`4` and `B` is `m`-by-`4`, then `area` is an `n`-by-`m` matrix where `area(i,j)` is the intersection area of the rectangles specified by the `i`th row of `A` and the `j`th row of `B`.

Note A position vector is a four-element vector `[x,y,width,height]`, where the point defined by `x` and `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the `x` and `y` axes respectively.

See Also

`polyarea`

Purpose

Set option to move deleted files to recycle folder

Syntax

```
S = recycle
S = recycle state
S = recycle('state')
```

Description

`S = recycle` returns a character array `S` that shows the current state of the MATLAB file recycling option. This state can be either `on` or `off`. When file recycling is `on`, MATLAB moves all files that you delete with the `delete` function to either the recycle bin on the PC or Macintosh, or a temporary directory on UNIX. (To locate this directory on UNIX, see the Remarks section below.) When file recycling is `off`, any files you delete are actually removed from the system.

The default recycle state is `off`. You can turn recycling on for all of your MATLAB sessions using the Preferences dialog box (Select **File > Preferences > General**). Under the heading **Default behavior of the delete function** select **Move files to the Recycle Bin**.

`S = recycle state` sets the MATLAB recycle option to the given `state`, either `on` or `off`. Return value `S` shows the previous recycle state.

`S = recycle('state')` is the function format for this command.

Remarks

On UNIX systems, you can locate the system temporary directory by entering the MATLAB function `tempdir`. The recycle directory is a subdirectory of this temporary directory, and is named according to the format

```
MATLAB_Files_<day>-<mo>-<yr>_<hr>_<min>_<sec>
```

For example, files recycled on a UNIX system at 2:09:28 in the afternoon of November 9, 2004 would be copied to a directory named

```
/tmp/MATLAB_Files_09-Nov-2004_14_09_28
```

To set the recycle state for all MATLAB sessions, use the **Preferences** dialog box. Open the **Preferences** dialog and select **General**. To

enable or disable recycling, click **Move files to the recycle bin** or **Delete files permanently**. See "General Preferences for MATLAB" in the Desktop Tools and Development Environment documentation for more information.

You can recycle files that are stored on your local computer system, but not files that you access over a network connection. On Windows systems, when you use the `delete` function on files accessed over a network, MATLAB removes the file entirely.

Examples

Start from a state where file recycling has been turned off. Check the current recycle state:

```
recycle
ans =
   off
```

Turn file recycling on. Delete a file and verify that it has been transferred to the recycle bin or temporary folder:

```
recycle on;
delete myfile.txt
```

See Also

`delete`, `dir`, `ls`, `fileparts`, `mkdir`, `rmdir`

Purpose

Reduce number of patch faces

Syntax

```
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
nfv = reducepatch(p) or nfv = reducepatch(fv)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)
```

Description

`reducepatch(p,r)` reduces the number of faces of the patch identified by handle `p`, while attempting to preserve the overall shape of the original object. MATLAB interprets the reduction factor `r` in one of two ways depending on its value:

- If `r` is less than 1, `r` is interpreted as a fraction of the original number of faces. For example, if you specify `r` as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If `r` is greater than or equal to 1, then `r` is the target number of faces. For example, if you specify `r` as 400, then the number of faces is reduced until there are 400 faces remaining.

`nfv = reducepatch(p,r)` returns the reduced set of faces and vertices but does not set the `Faces` and `Vertices` properties of patch `p`. The struct `nfv` contains the faces and vertices after reduction.

`nfv = reducepatch(fv,r)` performs the reduction on the faces and vertices in the struct `fv`.

`nfv = reducepatch(p) or nfv = reducepatch(fv)` uses a reduction value of `0.5`.

`reducepatch(...,'fast')` assumes the vertices are unique and does not compute shared vertices.

`reducepatch(...,'verbose')` prints progress messages to the command window as the computation progresses.

`nfv = reducepatch(f,v,r)` performs the reduction on the faces in `f` and the vertices in `v`.

`[nf,nv] = reducepatch(...)` returns the faces and vertices in the arrays `nf` and `nv`.

Remarks

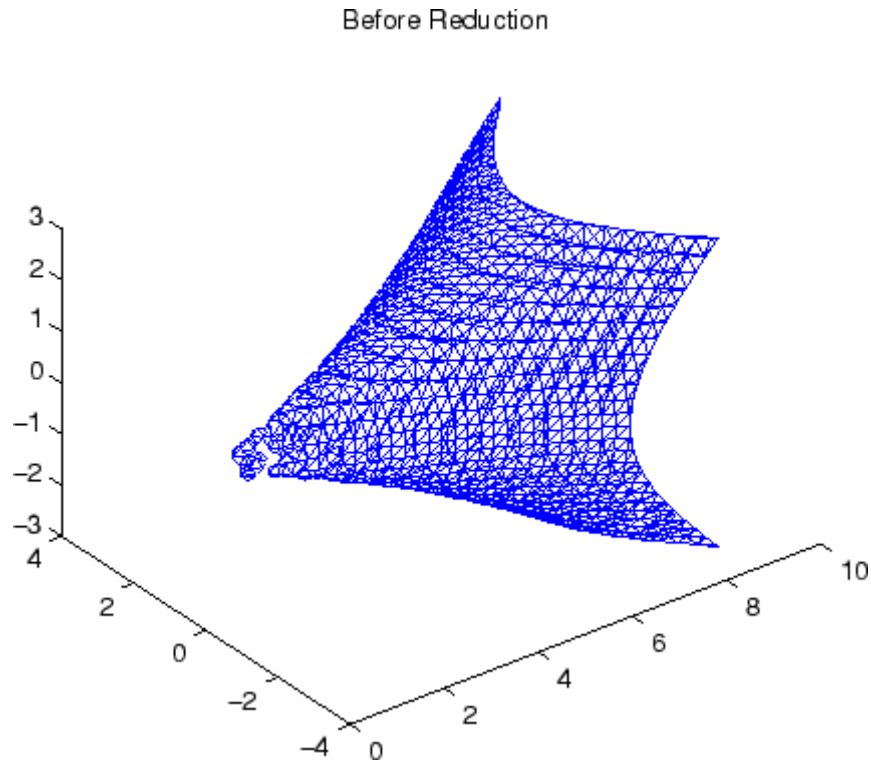
If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

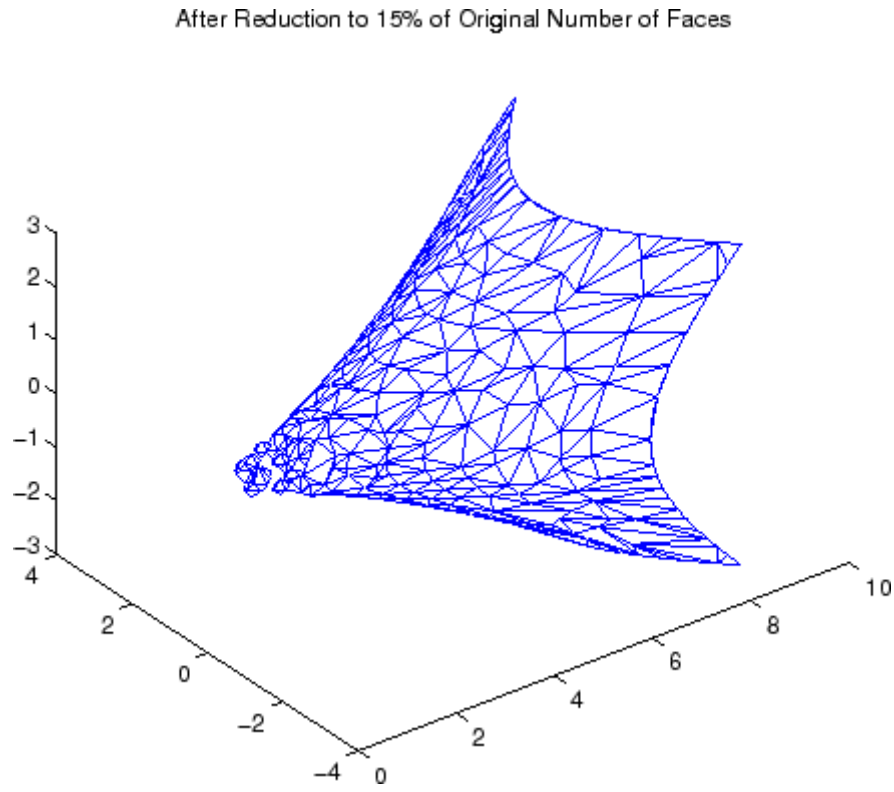
The number of output triangles may not be exactly the number specified with the reduction factor argument (`r`), particularly if the faces of the original patch are not triangles.

Examples

This example illustrates the effect of reducing the number of faces to only 15% of the original value.

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
set(p,'facecolor','w','EdgeColor','b');
daspect([1,1,1])
view(3)
figure;
h = axes;
p2 = copyobj(p,h);
reducepatch(p2,0.15)
daspect([1,1,1])
view(3)
```


Before Reduction
3
2
1
0
−1
−2
−3
4
2
0
−2
−4
0
2
4
6
8
10



See Also

`isosurface`, `isocaps`, `isonormals`, `smooth3`, `subvolume`, `reducevolume`

"Volume Visualization" on page 1-102 for related functions

Vector Field Displayed with Cone Plots for another example

Purpose

Reduce number of elements in volume data set

Syntax

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])
[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])
nv = reducevolume(...)
```

Description

`[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])` reduces the number of elements in the volume by retaining every Rx^{th} element in the x direction, every Ry^{th} element in the y direction, and every Rz^{th} element in the z direction. If a scalar `R` is used to indicate the amount or reduction instead of a three-element vector, MATLAB assumes the reduction to be `[R R R]`.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The reduced volume is returned in `nv`, and the coordinates of the reduced volume are returned in `nx`, `ny`, and `nz`.

`[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p] = size(V)`.

`nv = reducevolume(...)` returns only the reduced volume.

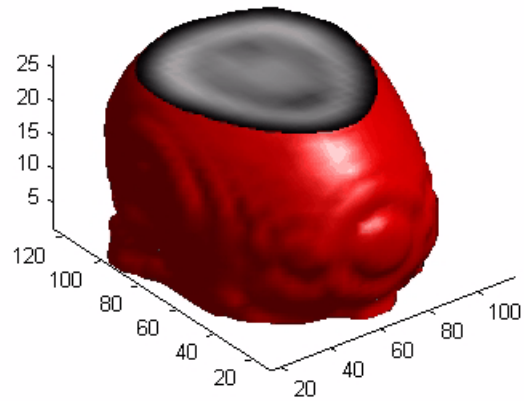
Examples

This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every fourth element in the x and y directions and every element in the z direction.
- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).

- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).
- Adding a light to the right of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = reducevolume(D,[4,4,1]);
D = smooth3(D);
p1 = patch(isosurface(x,y,z,D, 5,'verbose'),...
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
    'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight; lighting gouraud
```



See Also

`isosurface`, `isocaps`, `isonormals`, `smooth3`, `subvolume`, `reducepatch`

“Volume Visualization” on page 1-102 for related functions

refresh

Purpose

Redraw current figure

Syntax

```
refresh
refresh(h)
```

Description

`refresh` erases and redraws the current figure.

`refresh(h)` redraws the figure identified by `h`.

See Also

"Figure Windows" on page 1-95 for related functions

Purpose

Refresh data in graph when data source is specified

Syntax

```
refreshdata
refreshdata(figure_handle)
refreshdata(object_handles)
refreshdata(object_handles,'workspace')
```

Description

`refreshdata` evaluates any data source properties (`XDataSource`, `YDataSource`, or `ZDataSource`) on all objects in graphs in the current figure. If the specified data source has changed, MATLAB updates the graph to reflect this change.

Note that the variable assigned to the data source property must be in the base workspace.

`refreshdata(figure_handle)` refreshes the data of the objects in the specified figure.

`refreshdata(object_handles)` refreshes the data of the objects specified in `object_handles` or the children of those objects. Therefore, `object_handles` can contain figure, axes, or plot object handles.

`refreshdata(object_handles,'workspace')` enables you to specify whether the data source properties are evaluated in the base workspace or the workspace of the function in which `refreshdata` was called. *`workspace`* is a string that can be

- `base` — Evaluate the data source properties in the base workspace.
- `caller` — Evaluate the data source properties in the workspace of the function that called refreshdata.

Examples

This example creates a contour plot and changes its data source. The call to `refreshdata` causes the graph to update.

```
z = peaks(5);
[c h] = contour(z,'ZDataSource','z');
drawnow
pause(3) % Wait 3 seconds and the graph will update
```

```
z = peaks(20);
refreshdata(h)
```

See Also

The `[X,Y,Z]DataSource` properties of plot objects.

Purpose

Match regular expression

Syntax

```
regexp('str', 'expr')
[start_idx, end_idx, extents, matches, tokens, names,
   splits] = regexp('str', 'expr')
[v1, v2, ...] = regexp('str', 'expr', q1, q2, ...)
[v1 v2 ...] = regexp('str', 'expr', ..., options)
```

Each of these syntaxes apply to both `regexp` and `regexpi`. The `regexp` function is case sensitive in matching regular expressions to a string, and `regexpi` is case insensitive.

Description

The following descriptions apply to both `regexp` and `regexpi`:

`regexp('str', 'expr')` returns a row vector containing the starting index of each substring of `str` that matches the regular expression string `expr`. If no matches are found, `regexp` returns an empty array. The `str` and `expr` arguments can also be cell arrays of strings. See "Regular Expressions" in the MATLAB Programming documentation for more information.

To specify more than one string to parse or more than one expression to match, see the guidelines listed below under "Multiple Strings or Expressions" on page 2-2733.

`[start_idx, end_idx, extents, matches, tokens, names, splits] = regexp('str', 'expr')` returns up to six values, one for each output variable you specify, and in the default order (as shown in the table below).

Note The `str` and `expr` inputs are required and must be entered as the first and second arguments, respectively. Any other input arguments (all are described below) are optional and can be entered following the two required inputs in any order.

`[v1, v2, ...] = regexp('str', 'expr', q1, q2, ...)` returns up to six values, one for each output variable you specify, and ordered according to the order of the qualifier arguments, `q1`, `q2`, etc.

Return Values for Regular Expressions

Default Order	Description	Qualifier
1	Row vector containing the starting index of each substring of `str` that matches `expr`.	**`start`**
2	Row vector containing the ending index of each substring of `str` that matches `expr`.	**`end`**
3	Cell array containing the starting and ending indices of each substring of `str` that matches a token in `expr`. (This is a `double` array when used with `'once'`.)	**`tokenExtents`**
4	Cell array containing the text of each substring of `str` that matches `expr`. (This is a string when used with `'once'`.)	**`match`**
5	Cell array of cell arrays of strings containing the text of each token captured by `regexp`. (This is a cell array of strings when used with `'once'`.)	**`tokens`**
6	Structure array containing the name and text of each *named* token captured by `regexp`. If there are no named tokens in `expr`, `regexp` returns a structure array with no fields. Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression `(?<tokenname>)`.	**`names`**
7	Cell array containing those parts of the input string that are delimited by substrings returned when using the `regexp 'match'` option.	**`split`**

Tip When using the `split` option, `regexp` always returns one more string than it does with the `match` option. Also, you can always put the original input string back together from the substrings obtained from both `split` and `match`. See "Example 4 — Splitting the Input String" on page 2-2735.

`[v1 v2 ...] = regexp('str', 'expr', ..., options)` calls `regexp` with one or more of the nondefault options listed in the following table. These options must follow `str` and `expr` in the input argument list.

Option	Description
`mode`	See the section on "Modes" on page 2-2731 below.
'**`once`**'	Return only the first match found.
'**`warnings`**'	Display any hidden warning messages issued by MATLAB during the execution of the command. This option only enables warnings for the one command being executed. See Example 10.

Modes

You can specify one or more of the following modes with the `regexp`, `regexpi`, and `regexprep` functions. You can enable or disable any of these modes using the mode specifier keyword (e.g., `'lineanchors'`) or the mode flag (e.g., `(?m)`). Both are shown in the tables that follow. Use the keyword to enable or disable the mode for the entire string being parsed. Use the flag to both enable and disable the mode for selected pieces of the string.

Case-Sensitivity Mode

Use the Case-Sensitivity mode to control whether or not MATLAB considers letter case when matching an expression to a string. Example 6 illustrates the this mode.

Mode Keyword	Flag	Description
'**matchcase**'	`(?-i)`	Letter case must match when matching patterns to a string. (The default for `regexp`).
'**ignorecase**'	`(?i)`	Do not consider letter case when matching patterns to a string. (The default for `regexpi`).

Dot Matching Mode

Use the Dot Matching mode to control whether or not MATLAB includes the newline (`\n`) character when matching the dot (`.`) metacharacter in a regular expression. Example 7 illustrates the Dot Matching mode.

Mode Keyword	Flag	Description
'**dotall**'	`(?s)`	Match dot ('.') in the pattern string with any character. (This is the default).
'**dotexceptnewline**'	`(?-s)`	Match dot in the pattern with any character that is not a newline.

Anchor Type Mode

Use the Anchor Type mode to control whether MATLAB considers the ^ and $ metacharacters to represent the beginning and end of a string or the beginning and end of a line. Example 8 illustrates the Anchor mode.

Mode Keyword	Flag	Description
'**stringanchors**'	(?-m)	Match the ^ and $ metacharacters at the beginning and end of a string. (This is the default).
'**lineanchors**'	(?m)	Match the ^ and $ metacharacters at the beginning and end of a line.

Spacing Mode

Use the Spacing mode to control how MATLAB interprets space characters and comments within the string being parsed. Example 9 illustrates the Spacing mode.

Mode Keyword	Flag	Description
'**literalspacing**'	(?-x)	Parse space characters and comments (the # character and any text to the right of it) in the same way as any other characters in the string. (This is the default).
'**freespacing**'	(?x)	Ignore spaces and comments when parsing the string. (You must use '\ ' and '\#' to match space and # characters.)

Remarks

See "Regular Expressions" in the MATLAB Programming documentation for a listing of all regular expression elements supported by MATLAB.

Multiple Strings or Expressions

Either the str or expr argument, or both, can be a cell array of strings, according to the following guidelines:

- If `str` is a cell array of strings, then each of the `regexp` outputs is a cell array having the same dimensions as `str`.
- If `str` is a single string but `expr` is a cell array of strings, then each of the `regexp` outputs is a cell array having the same dimensions as `expr`.
- If both `str` and `expr` are cell arrays of strings, these two cell arrays must contain the same number of elements.

Examples

Example 1 — Matching a Simple Pattern

Return a row vector of indices that match words that start with `c`, end with `t`, and contain one or more vowels between them. Make the matches insensitive to letter case (by using `regexpi`):

```
str = 'bat cat can car COAT court cut ct CAT-scan';
regexpi(str, 'c[aeiou]+t')
ans =
     5    17    28    35
```

Example 2 — Parsing Multiple Input Strings

Return a cell array of row vectors of indices that match capital letters and white spaces in the cell array of strings `str`:

```
str = {'Madrid, Spain' 'Romeo and Juliet' 'MATLAB is great'};
s1 = regexp(str, '[A-Z]');
s2 = regexp(str, '\s');
```

Capital letters, `'[A-Z]'`, were found at these `str` indices:

```
s1{:}
ans =
     1     9
ans =
     1    11
ans =
     1     2     3     4     5     6
```

Space characters, `'\s'`, were found at these `str` indices:

```
s2{:}
ans =
     8
ans =
     6    10
ans =
     7    10
```

Example 3 — Selecting Return Values

Return the text and the starting and ending indices of words containing the letter `x`:

```
str = 'regexp helps you relax';
[m s e] = regexp(str, '\w*x\w*', 'match', 'start', 'end')
m =
    'regexp'    'relax'
s =
     1    18
e =
     6    22
```

Example 4 — Splitting the Input String

Find the substrings delimited by the `^` character:

```
s1 = ['Use REGEXP to split ^this string into ' ...
      'several ^individual pieces'];

s2 = regexp(s1, '\^', 'split');

s2(:)
ans =
    'Use REGEXP to split '
    'this string into several '
    'individual pieces'
```

The `split` option returns those parts of the input string that are not returned when using the `'match'` option. Note that when you match the beginning or ending characters in a string (as is done in this example), the first (or last) return value is always an empty string:

```
str = 'She sells sea shells by the seashore.';

[matchstr splitstr] = regexp(str, '[Ss]h.', 'match', 'split')
matchstr =
    'She'    'she'    'sho'
splitstr =
     ''    ' sells sea '    'lls by the sea'    're.'
```

For any string that has been split, you can reassemble the pieces into the initial string using the command

```
j = [splitstr; [matchstr {''}]]; [j{:}]

ans =
   She sells sea shells by the seashore.
```

Example 5 — Using Tokens

Search a string for opening and closing HTML tags. Use the expression `<(\w+)` to find the opening tag (e.g., `'<tagname'`) and to create a token for it. Use the expression `</\1>` to find another occurrence of the same token, but formatted as a closing tag (e.g., `'</tagname>'`):

```
str = ['if <code>A</code> == x<sup>2</sup>, ' ...
       '<em>disp(x)</em>']
str =
if <code>A</code> == x<sup>2</sup>, <em>disp(x)</em>

expr = '<(\w+).*?>.*?</\1>';

[tok mat] = regexp(str, expr, 'tokens', 'match');

tok{:}
ans =
```

```
    'code'
ans =
    'sup'
ans =
    'em'

mat{:}
ans =
    <code>A</code>
ans =
    <sup>2</sup>
ans =
    <em>disp(x)</em>
```

See “Tokens” in the MATLAB Programming documentation for information on using tokens.

Example 6 — Using Named Capture

Enter a string containing two names, the first and last names being in a different order:

```
str = sprintf('John Davis\nRogers, James')
str =
    John Davis
    Rogers, James
```

Create an expression that generates first and last name tokens, assigning the names `first` and `last` to the tokens. Call `regexp` to get the text and names of each token found:

```
expr = ...
   '(?<first>\w+)\s+(?<last>\w+)|(?<last>\w+),\s+(?<first>\w+)';

[tokens names] = regexp(str, expr, 'tokens', 'names');
```

Examine the `tokens` cell array that was returned. The first and last name tokens appear in the order in which they were generated: first name–last name, then last name–first name:

```
tokens{:}
ans =
    'John'    'Davis'
ans =
    'Rogers'    'James'
```

Now examine the `names` structure that was returned. First and last names appear in a more usable order:

```
names(:,1)
ans =
    first: 'John'
     last: 'Davis'

names(:,2)
ans =
    first: 'James'
     last: 'Rogers'
```

Example 7 — Using the Case-Sensitive Mode

Given a string that has both uppercase and lowercase letters,

```
str = 'A string with UPPERCASE and lowercase text.';
```

Use the `regexp` default mode (case-sensitive) to locate only the lowercase instance of the word `case`:

```
regexp(str, 'case', 'match')
ans =
    'case'
```

Now disable case-sensitive matching to find both instances of `case`:

```
regexp(str, 'case', 'ignorecase', 'match')
ans =
    'CASE'    'case'
```

Match 5 letters that are followed by '`CASE`'. Use the `(?-i)` flag to turn on case-sensitivity for the first match and `(?i)` to turn it off for the second:

```
M = regexp(str, {'(?-i)\w{5}(?=CASE)', ...
                 '(?i)\w{5}(?=CASE)'}, 'match');

M{:}
ans =
    'UPPER'
ans =
    'UPPER'    'lower'
```

Example 8 — Using the Dot Matching Mode

Parse the following string that contains a newline (`\n`) character:

```
str = sprintf('abc\ndef')
str =
   abc
   def
```

When you use the default mode, `dotall`, MATLAB includes the newline in the characters matched:

```
regexp(str, '.', 'match')
ans =
   'a'   'b'   'c'   [1x1 char]   'd'   'e'   'f'
```

When you use the `dotexceptnewline` mode, MATLAB skips the newline character:

```
regexp(str, '.', 'match', 'dotexceptnewline')
ans =
   'a'   'b'   'c'   'd'   'e'   'f'
```

Example 9 — Using the Anchor Type Mode

Given the following two-line string,

```
str = sprintf('%s\n%s', 'Here is the first line', ...
              'followed by the second line')
str =
   Here is the first line
```

```
    followed by the second line
```

In `stringanchors` mode, MATLAB interprets the `$` metacharacter as an end-of-string specifier, and thus finds the last two words of the entire *string*:

```
regexp(str, '\w+\W\w+$', 'match', 'stringanchors')
ans =
    'second line'
```

While in `lineanchors` mode, MATLAB interprets `$` as an end-of-line specifier, and finds the last two words of each *line*:

```
regexp(str, '\w+\W\w+$', 'match', 'lineanchors')
ans =
    'first line'    'second line'
```

Example 10 — Using the Freespacing Mode

Create a file called `regexp_str.txt` containing the following text. Because the first line enables freespacing mode, MATLAB ignores all spaces and comments that appear in the file:

```
(?x)    # turn on freespacing.

# This pattern matches a string with a repeated letter.

\w*     # First, match any number of preceding word characters.

(       # Mark a token.
    \w  # Match a word character.
)       # Finish capturing said token.
\1      # Backreference to match what token #1 matched.

\w*     # Finally, match the remainder of the word.
```

Here is the string to parse:

```
str = ['Looking for words with letters that ' ...
```

```
          'appear twice in succession.'];
```

Use the pattern expression read from the file to find those words that have consecutive matching letters:

```
patt = fileread('regexp_str.txt');
regexp(str, patt, 'match')
ans =
    'Looking'    'letters'    'appear'    'succession'
```

Example 11 — Displaying Parsing Warnings

To help debug problems in parsing a string with `regexp`, `regexpi`, or `regexprep`, use the 'warnings' option to view all warning messages:

```
regexp('$.', '[a-]','warnings')
Warning: Unbound range.
 [a-]
   |
```

See Also

`regexprep`, `regexptranslate`, `strfind`, `findstr`, `strmatch`, `strcmp`, `strcmpi`, `strncmp`, `strncmpi`

Purpose

Replace string using regular expression

Syntax

```
s = regexprep('str', 'expr', 'repstr')
s = regexprep('str', 'expr', 'repstr' options)
```

Description

`s = regexprep('str', 'expr', 'repstr')` replaces all occurrences of the regular expression `expr` in string `str` with the string `repstr`. The new string is returned in `s`. If no matches are found, return string `s` is the same as input string `str`. You can use character representations (e.g., `'\t'` for tab, or `'\n'` for newline) in replacement string `repstr`. See "Regular Expressions" in the MATLAB Programming documentation for more information.

If `str` is a cell array of strings, then the `regexprep` return value `s` is always a cell array of strings having the same dimensions as `str`.

To specify more than one expression to match or more than one replacement string, see the guidelines listed below under "Multiple Expressions or Replacement Strings" on page 2-2743.

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the `(...)` operator. Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See "Tokens" and the example "Using Tokens in a Replacement String" in the MATLAB Programming documentation for information on using tokens.)

`s = regexprep('str', 'expr', 'repstr' options)` By default, `regexprep` replaces all matches and is case sensitive. You can use one or more of the following options with `regexprep`.

Option	Description
`mode`	See `mode` descriptions on the `regexp` reference page.
`N`	Replace only the `N`th occurrence of `expr` in `str`.
'**once**'	Replace only the first occurrence of `expr` in `str`.

Option	Description
'**ignorecase**'	Ignore case when matching and when replacing.
'**preservecase**'	Ignore case when matching (as with `'ignorecase'`), but override the case of `replace` characters with the case of corresponding characters in `str` when replacing.
'**warnings**'	Display any hidden warning messages issued by MATLAB during the execution of the command. This option only enables warnings for the one command being executed.

Remarks

See "Regular Expressions" in the MATLAB Programming documentation for a listing of all regular expression metacharacters supported by MATLAB.

Multiple Expressions or Replacement Strings

In the case of multiple expressions and/or replacement strings, `regexprep` attempts to make all matches and replacements. The first match is against the initial input string. Successive matches are against the string resulting from the previous replacement.

The `expr` and `repstr` inputs follow these rules:

- If `expr` is a cell array of strings and `repstr` is a single string, `regexprep` uses the same replacement string on each expression in `expr`.
- If `expr` is a single string and `repstr` is a cell array of N strings, `regexprep` attempts to make N matches and replacements.
- If both `expr` and `repstr` are cell arrays of strings, then `expr` and `repstr` must contain the same number of elements, and `regexprep` pairs each `repstr` element with its matching element in `expr`.

Examples

Example 1 — Making a Case-Sensitive Replacement

Perform a case-sensitive replacement on words starting with `m` and ending with `y`:

```
str = 'My flowers may bloom in May';
pat = 'm(\w*)y';
regexprep(str, pat, 'April')
ans =
   My flowers April bloom in May
```

Replace all words starting with `m` and ending with `y`, regardless of case, but maintain the original case in the replacement strings:

```
regexprep(str, pat, 'April', 'preservecase')
ans =
   April flowers april bloom in April
```

Example 2 — Using Tokens In the Replacement String

Replace all variations of the words `'walk up'` using the letters following `walk` as a token. In the replacement string

```
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';
regexprep(str, pat, 'ascend$1')
ans =
   I ascend, they ascended, we are ascending.
```

Example 3 — Operating on Multiple Strings

This example operates on a cell array of strings. It searches for consecutive matching letters (e.g., `'oo'`) and uses a common replacement value (`'--'`) for all matches. The function returns a cell array of strings having the same dimensions as the input cell array:

```
str = {                                  ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};
```

```
a = regexprep(str, '(.)\1', '--', 'ignorecase')
a =
    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

See Also

regexp, regexpi, regexptranslate, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi

regexptranslate

Purpose

Translate string into regular expression

Syntax

```
s2 = regexptranslate(type, s1)
```

Description

`s2 = regexptranslate(type, s1)` translates string `s1` into a regular expression string `s2` that you can then use as input into one of the MATLAB regular expression functions such as `regexp`. The `type` input can be either one of the following strings that define the type of translation to be performed. See "Regular Expressions" in the MATLAB Programming documentation for more information.

Type	Description
`'escape'`	Translate all special characters (e.g., '$', '.', '?', '[') in string `s1` so that they are treated as literal characters when used in the `regexp` and `regexprep` functions. The translation inserts an escape character ('\') before each special character in `s1`. Return the new string in `s2`.
`'wildcard'`	Translate all wildcard and '.' characters in string `s1` so that they are treated as literal wildcards and periods when used in the `regexp` and `regexprep` functions. The translation replaces all instances of '*' with '.*', all instances of '?' with '.', and all instances of '.' with '\.'. Return the new string in `s2`.

Examples

Example 1 — Using the 'escape' Option

Because `regexp` interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\' and 'n' in this string:

```
str = 'The sequence \n generates a new line';
pat = '\n';

regexp(str, pat)
ans =
     []
```

To have regexp interpret the expression expr as the characters '\' and 'n', first translate the expression using regexptranslate:

```
pat2 = regexptranslate('escape', pat)
pat2 =
   \\n

regexp(str, pat2)
ans =
    14
```

Example 2 — Using 'escape' In a Replacement String

Replace the word 'walk' with 'ascend' in this string, treating the characters '$1' as a token designator:

```
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';

regexprep(str, pat, 'ascend$1')
ans =
   I ascend, they ascended, we are ascending.
```

Make another replacement on the same string, this time treating the '$1' as literal characters:

```
regexprep(str, pat, regexptranslate('escape', 'ascend$1'))
ans =
   I ascend$1, they ascend$1, we are ascend$1.
```

Example 3 — Using the 'wildcard' Option

Given the following string of filenames, pick out just the MAT-files. Use regexptranslate to interpret the '*' wildcard as '\w+' instead of as a regular expression quantifier:

```
files = ['test1.mat, myfile.mat, newfile.txt, ' ...
         'jan30.mat, table3.xls'];
regexp(str, regexptranslate('wildcard', '*.mat'), 'match')
ans =
```

```
    'test1.mat' 'myfile.mat' 'jan30.mat'
```

To see the translation, you can type

```
regexptranslate('wildcard','*.mat')
ans =
    \w+\.mat
```

See Also

regexp, regexpi, regexprep

Purpose

Register event handler with control's event

Syntax

```
h.registerevent(event_handler)
registerevent(h, event_handler)
```

Description

`h.registerevent(event_handler)` registers certain event handler routines with their corresponding events. Once an event is registered, the control responds to the occurrence of that event by invoking its event handler routine. The `event_handler` argument can be either a string that specifies the name of the event handler function, or a function handle that maps to that function. Strings used in the `event_handler` argument are not case sensitive.

`registerevent(h, event_handler)` is an alternate syntax for the same operation.

You can either register events at the time you create the control (using `actxcontrol`), or register them dynamically at any time after the control has been created (using `registerevent`). The `event_handler` argument specifies both events and event handlers (see "Writing Event Handlers" in the External Interfaces documentation).

Examples

Register Events Using Function Name Example

Create an `mwsamp` control and list all events associated with the control:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

h.events
ans =
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
      Variant x, Variant y)
```

Register all events with the same event handler routine, `sampev`. Use `eventlisteners` to see the event handler used by each event:

```
h.registerevent('sampev');
h.eventlisteners
ans =
    'click'        'sampev'
    'dblclick'     'sampev'
    'mousedown'    'sampev'

h.unregisterallevents;
```

Register the `Click` and `DblClick` events with the event handlers `myclick` and `my2click`, respectively. Note that the strings in the argument list are not case sensitive.

```
h.registerevent({'click' 'myclick'; ...
                 'dblclick' 'my2click'});
h.eventlisteners
ans =
    'click'       'myclick'
    'dblclick'    'my2click'
```

Register Events Using Function Handle Example

Register all events with the same event handler routine, `sampev`, but use a function handle (`@sampev`) instead of the function name:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200]);
registerevent(h, @sampev);
```

Register Excel Workbook Events Example

Create an Excel Workbook object.

```
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
```

Register all events with the same event handler routine, `AllEventHandler`.

```
wb.registerevent('AllEventHandler')
wb.eventlisteners
```

MATLAB displays the list of all Workbook events, registered with `AllEventHandler`.

```
ans =

    'Open'                'AllEventHandler'
    'Activate'            'AllEventHandler'
    'Deactivate'          'AllEventHandler'
    'BeforeClose'         'AllEventHandler'
                  .
                  .
```

See Also

`events`, `eventlisteners`, `unregisterevent`, `unregisterallevents`, `isevent`

rehash

Purpose

Refresh function and file system path caches

Syntax

```
rehash
rehash path
rehash toolbox
rehash pathreset
rehash toolboxreset
rehash toolboxcache
```

Description

`rehash` with no arguments updates the MATLAB list of known files and classes for directories on the search path that are not in *`matlabroot`*`/toolbox`. It compares the timestamps for loaded shadowed functions (functions that have been called but not cleared in the current session) against their timestamps on disk. It clears loaded functions if the files on disk are newer. All of this normally happens each time MATLAB displays the Command Window prompt. Therefore, use `rehash` with no arguments only when you run an M-file that updates another M-file, and the calling file needs to reuse the updated version before it has finished running.

`rehash` **`path`** performs the same updates as `rehash`, but uses a different technique for detecting the files and directories that require updates. If you receive a warning during MATLAB startup notifying you that MATLAB could not tell if a directory has changed and you encounter problems with MATLAB using the most current versions of your M-files, run `rehash path`.

`rehash` **`toolbox`** updates all directories in *`matlabroot`*`/toolbox`. Run this when you add or remove files in *`matlabroot`*`/toolbox` during a session by some means other than MATLAB tools.

`rehash` **`pathreset`** performs the same updates as `rehash` **`path`**, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash` **`toolboxreset`** performs the same updates as `rehash` **`toolbox`**, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash` **`toolboxcache`** performs the same updates as `rehash` **`toolbox`**, and also updates the cache file. This is the equivalent of clicking the **Update Toolbox Path Cache** button in **Preferences > General**.

See Also

`addpath`, `clear`, `path`, `rmpath`

"Toolbox Path Caching in MATLAB" in the MATLAB Desktop Tools and Development Environment documentation

release

Purpose

Release interface

Syntax

```
h.release
release(h)
```

Description

`h.release` releases the interface and all resources used by the interface. Each interface handle must be released when you are finished manipulating its properties and invoking its methods. Once an interface has been released, it is no longer valid. Subsequent operations on the MATLAB object that represents that interface will result in errors.

`release(h)` is an alternate syntax for the same operation.

Note Releasing the interface does not delete the control itself (see `delete`), since other interfaces on that object may still be active. See Releasing Interfaces in the External Interfaces documentation for more information.

Examples

Create a Microsoft Calender application. Then create a `TitleFont` interface and use it to change the appearance of the font of the calendar's title:

```
f = figure('position',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);

TFont = cal.TitleFont
TFont =
   Interface.Standard_OLE_Types.Font

TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

When you're finished working with the title font, release the `TitleFont` interface:

```
TFont.release;
```

Now create a `GridFont` interface and use it to modify the size of the calendar's date numerals:

```
GFont = cal.GridFont
GFont =
   Interface.Standard_OLE_Types.Font

GFont.Size = 16;
```

When you're done, delete the `cal` object and the figure window:

```
cal.delete;
delete(f);
clear f;
```

See Also

`delete`, `save`, `load`, `actxcontrol`, `actxserver`

Purpose

Remainder after division

Syntax

```
R = rem(X,Y)
```

Description

`R = rem(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = fix(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

The following are true by convention:

- `rem(X,0)` is `NaN`
- `rem(X,X)` for `X~=0` is `0`
- `rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

Remarks

`mod(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `Y`.

`rem(X,Y)` and `mod(X,Y)` are equal if `X` and `Y` have the same sign, but differ by `Y` if `X` and `Y` have different signs.

The `rem` function returns a result that is between 0 and `sign(X)*abs(Y)`. If `Y` is zero, `rem` returns `NaN`.

See Also

`mod`

Purpose

Remove timeseries objects from tscollection object

Syntax

```
tsc = removets(tsc,Name)
```

Description

`tsc = removets(tsc,Name)` removes one or more `timeseries` objects with the name specified in `Name` from the `tscollection` object tsc. `Name` can either be a string or a cell array of strings.

Examples

The following example shows how to remove a time series from a `tscollection`.

1 Create two `timeseries` objects, `ts1` and `ts2`.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
```

2 Create a `tscollection` object `tsc`, which includes `ts1` and `ts2`.

```
tsc=tscollection({ts1 ts2});
```

3 To view the members of `tsc`, type the following at the MATLAB prompt:

```
tsc
```

MATLAB responds with

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
```

```
      acceleration
      speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of `ts1` and `ts2`, respectively.

4 Remove `ts2` from `tsc`.

```
tsc=removets(tsc,'speed');
```

5 To view the current members of `tsc`, type the following at the MATLAB prompt:

```
tsc
```

MATLAB responds with

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
      acceleration
```

The remaining member of `tsc` is `acceleration`. The `timeseries` `speed` has been removed.

See Also

`addts`, `tscollection`

Purpose

Rename file on FTP server

Syntax

```
rename(f,'oldname','newname')
```

Description

`rename(f,'oldname','newname')` changes the name of the file `oldname` to `newname` in the current directory of the FTP server `f`, where `f` was created using `ftp`.

Examples

Connect to server `testsite`, view the contents, and change the name of `testfile.m` to `showresults.m`.

```
test=ftp('ftp.testsite.com');
dir(test)
.               ..       testfile.m
rename(test,'testfile.m','showresults.m')
dir(test)
.               ..       showresults.m
```

See Also

`dir (ftp)`, `delete (ftp)`, `ftp`, `mget`, `mput`

Purpose

Replicate and tile array

Syntax

```
B = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p...])
```

Description

`B = repmat(A,m,n)` creates a large matrix `B` consisting of an `m`-by-`n` tiling of copies of `A`. The size of `B` is `[size(A,1)*m, (size(A,2)*n]`. The statement `repmat(A,n)` creates an `n`-by-`n` tiling.

`B = repmat(A,[m n])` accomplishes the same result as `repmat(A,m,n)`.

`B = repmat(A,[m n p...])` produces a multidimensional array `B` composed of copies of `A`. The size of `B` is `[size(A,1)*m, size(A,2)*n, size(A,3)*p, ...]`.

Remarks

`repmat(A,m,n)`, when `A` is a scalar, produces an `m`-by-`n` matrix filled with `A`'s value and having `A`'s `class`. For certain values, you can achieve the same results using other functions, as shown by the following examples:

- `repmat(NaN,m,n)` returns the same result as `NaN(m,n)`.
- `repmat(single(inf),m,n)` is the same as `inf(m,n,'single')`.
- `repmat(int8(0),m,n)` is the same as `zeros(m,n,'int8')`.
- `repmat(uint32(1),m,n)` is the same as `ones(m,n,'uint32')`.
- `repmat(eps,m,n)` is the same as `eps(ones(m,n))`.

Examples

In this example, `repmat` replicates 12 copies of the second-order identity matrix, resulting in a "checkerboard" pattern.

```
B = repmat(eye(2),3,4)

B =
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
```

```
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
```

The statement `N = repmat(NaN,[2 3])` creates a 2-by-3 matrix of `NaN`s.

See Also

`bsxfun`, `NaN`, `Inf`, `ones`, `zeros`

resample (timeseries)

Purpose

Select or interpolate `timeseries` data using new time vector

Syntax

```
ts = resample(ts,Time)
ts = resample(ts,Time,interp_method)
ts = resample(ts,Time,interp_method,code)
```

Description

`ts = resample(ts,Time)` resamples the `timeseries` object ts using the new Time vector. When `ts` uses date strings and Time is numeric, Time is treated as specified relative to the `ts.TimeInfo.StartDate` property and in the same units that ts uses. The `resample` operation uses the default interpolation method, which you can view by using the `getinterpmethod(ts)` syntax.

`ts = resample(ts,Time,interp_method)` resamples the `timeseries` object ts using the interpolation method given by the string interp_method. Valid interpolation methods include `'linear'` and `'zoh'` (zero-order hold).

`ts = resample(ts,Time,interp_method,code)` resamples the `timeseries` object ts using the interpolation method given by the string interp_method. The integer code is a user-defined `Quality` code for resampling, applied to all samples.

Examples

The following example shows how to resample a `timeseries` object.

1 Create a `timeseries` object.

```
ts=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'Name','speed');
```

2 Transpose `ts` to make the data columnwise.

```
ts=transpose(ts)
```

MATLAB displays

```
Time Series Object: speed

Time vector characteristics
```

```
      Length                5
      Start time            1 seconds
      End time              5 seconds

Data characteristics

      Interpolation method  linear
      Size                  [5  1]
      Data type             double

Time              Data              Quality
-------------------------------------------
1                 1.1
2                 2.9
3                 3.7
4                 4
5                 3
```

Note that the interpolation method is set to `linear`, by default.

3 Resample `ts` using its default interpolation method.

```
res_ts=resample(ts,[1 1.5 3.5 4.5 4.9])
```

MATLAB displays the resampled time series as follows:

```
Time Series Object: speed

Time vector characteristics

      Length                5
      Start time            1 seconds
      End time              4.900000e+000 seconds
```

```
Data characteristics

      Interpolation method  linear
      Size                  [5  1]
      Data type             double

Time              Data              Quality
-----------------------------------------
1                 1.1
1.5               2
3.5               3.85
4.5               3.5
4.9               3.1
```

See Also

getinterpmethod, setinterpmethod, synchronize, timeseries

Purpose

Select or interpolate data in tscollection using new time vector

Syntax

```
tsc = resample(tsc,Time)
tsc = resample(tsc,Time,interp_method)
tsc = resample(tsc,Time,interp_method,code)
```

Description

`tsc = resample(tsc,Time)` resamples the tscollection object tsc on the new Time vector. When tsc uses date strings and Time is numeric, Time is treated as numerical specified relative to the `tsc.TimeInfo.StartDate` property and in the same units that tsc uses. The resample method uses the default interpolation method for each time series member.

`tsc = resample(tsc,Time,interp_method)` resamples the tscollection object tsc using the interpolation method given by the string interp_method. Valid interpolation methods include `'linear'` and `'zoh'` (zero-order hold).

`tsc = resample(tsc,Time,interp_method,code)` resamples the tscollection object tsc using the interpolation method given by the string interp_method. The integer code is a user-defined quality code for resampling, applied to all samples.

Examples

The following example shows how to resample a tscollection that consists of two timeseries members.

1 Create two timeseries objects.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
```

2 Create a tscollection tsc.

```
tsc=tscollection({ts1 ts2});
```

The time vector of the collection tsc is [1:5], which is the same as for ts1 and ts2 (individually).

3 Get the interpolation method for `acceleration` by typing

```
tsc.acceleration
```

MATLAB responds with

```
Time Series Object: acceleration

Time vector characteristics

      Length               5
      Start time           1 seconds
      End time             5 seconds

Data characteristics

      Interpolation method  linear
      Size                  [1  1  5]
      Data type             double
```

4 Set the interpolation method for `speed` to zero-order hold by typing

```
setinterpmethod(tsc.speed,'zoh')
```

MATLAB responds with

```
Time Series Object: acceleration

Time vector characteristics

      Length               5
      Start time           1 seconds
      End time             5 seconds
```

```
Data characteristics

      Interpolation method  zoh
      Size                  [1  1  5]
      Data type             double
```

5 Resample the time-series collection `tsc` by individually resampling each time-series member of the collection and using its interpolation method.

```
res_tsc=resample(tsc,[1 1.5 3.5 4.5 4.9])
```

See Also

`getinterpmethod`, `setinterpmethod`, `tscollection`

reset

Purpose

Reset graphics object properties to their defaults

Syntax

```
reset(h)
```

Description

`reset(h)` resets all properties having factory defaults on the object identified by `h`. To see the list of factory defaults, use the statement

```
get(0,'factory')
```

If `h` is a `figure`, MATLAB does not reset `Position`, `Units`, `Windowstyle`, or `PaperUnits`. If `h` is an `axes`, MATLAB does not reset `Position` and `Units`.

Examples

`reset(gca)` resets the properties of the current axes.

`reset(gcf)` resets the properties of the current figure.

See Also

`cla`, `clf`, `gca`, `gcf`, `hold`

"Object Manipulation" on page 1-100 for related functions

Purpose

Reshape array

Syntax

```
B = reshape(A,m,n)
B = reshape(A,m,n,p,...)
B = reshape(A,[m n p ...])
B = reshape(A,...,[],...)
B = reshape(A,siz)
```

Description

`B = reshape(A,m,n)` returns the `m`-by-`n` matrix `B` whose elements are taken column-wise from `A`. An error results if `A` does not have `m*n` elements.

`B = reshape(A,m,n,p,...)` or `B = reshape(A,[m n p ...])` returns an n-dimensional array with the same elements as `A` but reshaped to have the size `m`-by-`n`-by-`p`-by-`...`. The product of the specified dimensions, `m*n*p*...`, must be the same as `prod(size(A))`.

`B = reshape(A,...,[],...)` calculates the length of the dimension represented by the placeholder `[]`, such that the product of the dimensions equals `prod(size(A))`. The value of `prod(size(A))` must be evenly divisible by the product of the specified dimensions. You can use only one occurrence of `[]`.

`B = reshape(A,siz)` returns an n-dimensional array with the same elements as `A`, but reshaped to `siz`, a vector representing the dimensions of the reshaped array. The quantity `prod(siz)` must be the same as `prod(size(A))`.

Examples

Reshape a 3-by-4 matrix into a 2-by-6 matrix.

```
A =
     1     4     7    10
     2     5     8    11
     3     6     9    12

B = reshape(A,2,6)

B =
```

```
     1     3     5     7     9    11
     2     4     6     8    10    12
B = reshape(A,2,[])

B =
     1     3     5     7     9    11
     2     4     6     8    10    12
```

See Also

`shiftdim`, `squeeze`

The colon operator `:`

Purpose

Convert between partial fraction expansion and polynomial coefficients

Syntax

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

Description

The `residue` function converts a quotient of polynomials to pole-residue representation, and back again.

`[r,p,k] = residue(b,a)` finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, $b(s)$ and $a(s)$, of the form

$$\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + b_3 s^{m-2} + \ldots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + a_3 s^{n-2} + \ldots + a_{n+1}}$$

where b_j and a_j are the `j`th elements of the input vectors `b` and `a`.

`[b,a] = residue(r,p,k)` converts the partial fraction expansion back to the polynomials with coefficients in `b` and `a`.

Definition

If there are no multiple roots, then

$$\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \ldots + \frac{r_n}{s-p_n} + k(s)$$

The number of poles `n` is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b)-length(a)+1
```

If `p(j) = ... = p(j+m-1)` is a pole of multiplicity `m`, then the expansion includes terms of the form

$$\frac{r_j}{s-p_j}+\frac{r_{j+1}}{(s-p_j)^2}+\ldots+\frac{r_{j+m-1}}{(s-p_j)^m}$$

Arguments

`b,a`	Vectors that specify the coefficients of the polynomials in descending powers of s
`r`	Column vector of residues
`p`	Column vector of poles
`k`	Row vector of direct terms

Algorithm

It first obtains the poles with `roots`. Next, if the fraction is nonproper, the direct term `k` is found using `deconv`, which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed. For repeated roots, `resi2` computes the residues at the repeated root locations.

Limitations

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, $a(s)$, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

Examples

If the ratio of two polynomials is expressed as

$$\frac{b(s)}{a(s)}=\frac{5s^3+3s^2-2s+7}{-4s^3+8s+3}$$

then

```
b = [ 5 3 -2 7]
a = [-4 0 8 3]
```

and you can calculate the partial fraction expansion as

```
[r, p, k] = residue(b,a)

r =
   -1.4167
   -0.6653
    1.3320

p =
    1.5737
   -1.1644
   -0.4093

k =
   -1.2500
```

Now, convert the partial fraction expansion back to polynomial coefficients.

```
[b,a] = residue(r,p,k)

b =
   -1.2500   -0.7500    0.5000   -1.7500

a =
    1.0000   -0.0000   -2.0000   -0.7500
```

The result can be expressed as

$$\frac{b(s)}{a(s)} = \frac{-1.25s^3 - 0.75s^2 + 0.50s - 1.75}{s^3 - 2.00s - 0.75}$$

Note that the result is normalized for the leading coefficient in the denominator.

See Also

`deconv`, `poly`, `roots`

References

[1] Oppenheim, A.V. and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975, p. 56.

Purpose

Restore default MATLAB search path

Syntax

```
restoredefaultpath
restoredefaultpath; matlabrc
```

Description

`restoredefaultpath` sets the search path to include only installed products from The MathWorks. Run `restoredefaultpath` if you are having problems with the search path. If `restoredefaultpath` seems to correct the problem, run `savepath`. Start MATLAB again to be sure the problem does not reappear.

`restoredefaultpath; matlabrc` sets the search path to include only installed products from The MathWorks and corrects path problems encountered during startup. Run `restoredefaultpath; matlabrc` if you are having problems with the search path and `restoredefaultpath` by itself does not correct the problem. After the problem seems to be resolved, run `savepath`. Start MATLAB again to be sure the problem does not reappear.

See Also

`addpath`, `path`, `pathdef`, `rmpath`, `savepath`

Search Path in the MATLAB Desktop Tools and Development Environment documentation

rethrow

Purpose

Reissue error

Syntax

```
rethrow(err)
```

Description

`rethrow(err)` reissues the error specified by `err`. The currently running M-file terminates and control returns to the keyboard (or to any enclosing `catch` block). The `err` argument must be a MATLAB structure containing at least one of the following fields.

Fieldname	Description
`message`	Text of the error message
`identifier`	Message identifier of the error message
`stack`	Information about the error from the program stack

See "Message Identifiers" in the MATLAB documentation for more information on the syntax and usage of message identifiers.

A convenient way to get a valid `err` structure for the last error issued is by using the `lasterror` function.

Remarks

The `err` input can contain the field `stack`, identical in format to the output of the `dbstack` command. If the `stack` field is present, the stack of the rethrown error will be set to that value. Otherwise, the stack will be set to the line at which the rethrow occurs.

Examples

`rethrow` is usually used in conjunction with `try-catch` statements to reissue an error from a `catch` block after performing `catch`-related operations. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(lasterror)
end
```

See Also

`error`, `lasterror`, `try`, `catch`, `dbstop`

rethrow (MException)

Purpose

Reissue existing exception

Syntax

```
rethrow(ME)
```

Description

`rethrow(ME)` terminates the currently running function, reissues an exception that is based on `MException` object `ME` that has been caught within a `try-catch` block, and returns control to the keyboard or to any enclosing `catch` block.

`rethrow` differs from the `throw` and `throwAsCaller` methods in that it does not modify the `stack` field. Call stack information in the `ME` object is kept as it was when the exception was first thrown.

`rethrow` can only issue a previously caught exception. If an exception that was not previously thrown is passed to the `rethrow` method, MATLAB generates a new exception.

You might use `rethrow` from the `catch` part of a `try-catch` block, for example, after performing some required cleanup tasks following an error.

Examples

This variation of the MATLAB `surf` function catches an error in the input arguments, gives the user the opportunity to correct the error, and rethrows the error if the user does not use that opportunity:

```
function surf2(varargin)
try
   surf(varargin{:})
catch ME
   ME.message      % Display the error.
   % Give user another try to enter input arguments.
   newargs = input('\nEnter argument list: ','s');
   if ~isempty(newargs)
      surf(eval(newargs));
   else
   % If no response from user, rethrow the error.
      rethrow(ME);
   end
```

```
end
```

When asked to correct the error, the user presses **Enter**. MATLAB rethrows the original error:

```
surf2
ans =
Not enough input arguments.

Enter argument list:
??? Error using ==> surf at 54
Not enough input arguments.

Error in ==> surf2 at 3
   surf(varargin{:});
```

This time, the user enters valid input and MATLAB successfully displays the output plot:

```
surf2
ans =
Not enough input arguments.

Enter argument list: peaks(30)
```

See Also

`try`, `catch`, `error`, `assert`, `MException`, `throw(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `getReport(MException)`, `disp(MException)`, `isequal(MException)`, `eq(MException)`, `ne(MException)`, `last(MException)`

Purpose

Return to invoking function

Syntax

```
return
```

Description

`return` causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.

Examples

If the determinant function were an M-file, it might use a `return` statement in handling the special case of an empty matrix, as follows:

```
function d = det(A)
%DET det(A) is the determinant of A.
if isempty(A)
    d = 1;
    return
else
    ...
end
```

See Also

`break`, `continue`, `disp`, `end`, `error`, `for`, `if`, `keyboard`, `switch`, `while`

Purpose

Convert RGB colormap to HSV colormap

Syntax

```
cmap = rgb2hsv(M)
hsv_image = rgb2hsv(rgb_image)
```

Description

`cmap = rgb2hsv(M)` converts an RGB colormap `M` to an HSV colormap `cmap`. Both colormaps are *m*-by-3 matrices. The elements of both colormaps are in the range 0 to 1.

The columns of the input matrix `M` represent intensities of red, green, and blue, respectively. The columns of the output matrix `cmap` represent hue, saturation, and value, respectively.

`hsv_image = rgb2hsv(rgb_image)` converts the RGB image to the equivalent HSV image. RGB is an m-by-n-by-3 image array whose three planes contain the red, green, and blue components for the image. HSV is returned as an m-by-n-by-3 image array whose three planes contain the hue, saturation, and value components for the image.

See Also

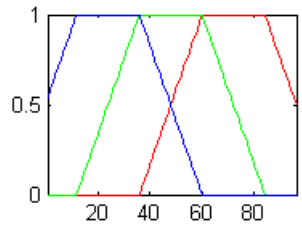
`brighten`, `colormap`, `hsv2rgb`, `rgbplot`

"Color Operations" on page 1-98 for related functions

rgbplot

Purpose

Plot colormap



Syntax

```
rgbplot(cmap)
```

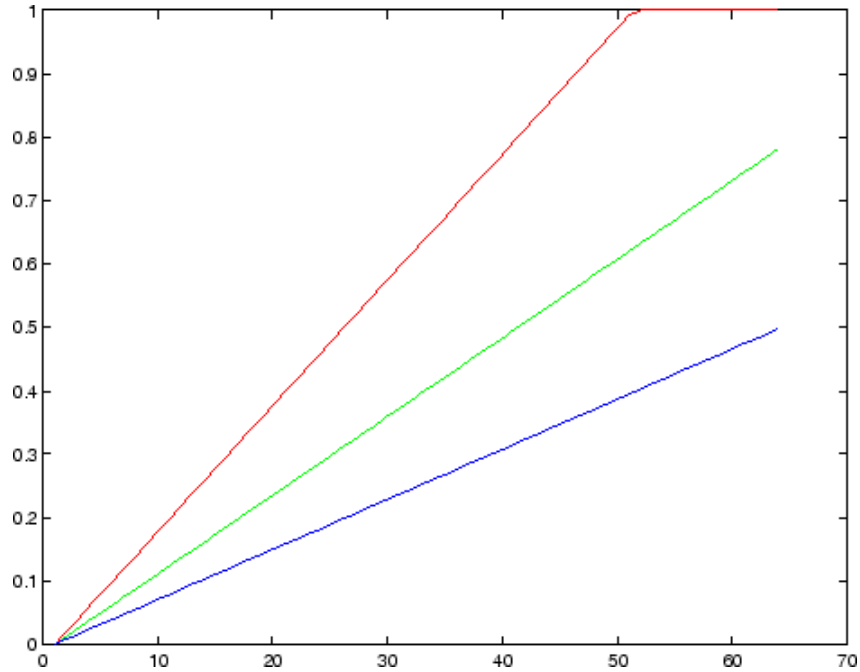
Description

`rgbplot(cmap)` plots the three columns of `cmap`, where `cmap` is an *m*-by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

Examples

Plot the RGB values of the `copper` colormap.

```
rgbplot(copper)
```



See Also

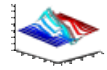
`colormap`

"Color Operations" on page 1-98 for related functions

ribbon

Purpose

Ribbon plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
ribbon(axes_handle,...)
h = ribbon(...)
```

Description

`ribbon(Y)` plots the columns of `Y` as separate three-dimensional ribbons using `X = 1:size(Y,1)`.

`ribbon(X,Y)` plots `X` versus the columns of `Y` as three-dimensional strips. `X` and `Y` are vectors of the same size or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows.

`ribbon(X,Y,width)` specifies the width of the ribbons. The default is `0.75`.

`ribbon(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

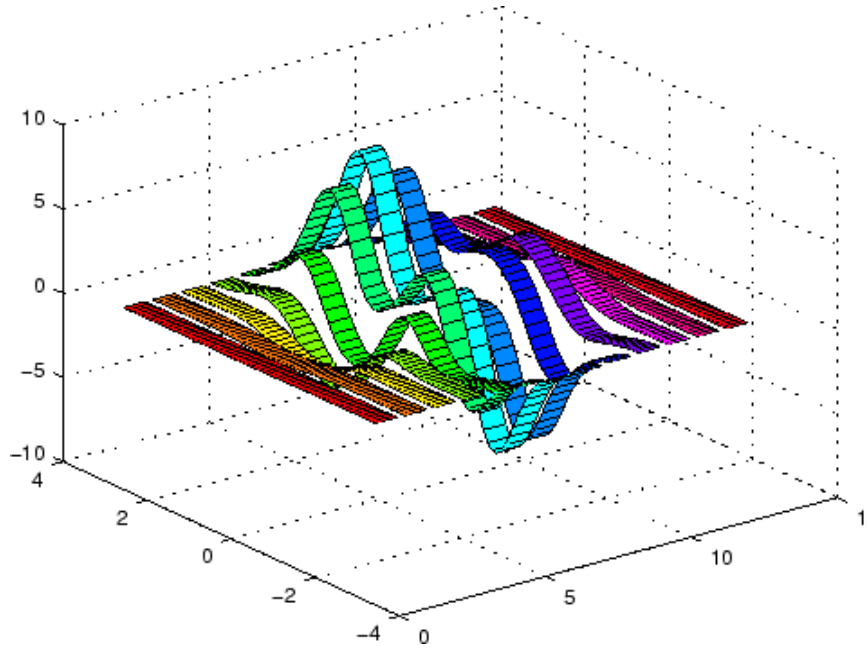
`h = ribbon(...)` returns a vector of handles to surface graphics objects. `ribbon` returns one handle per strip.

Examples

Create a ribbon plot of the `peaks` function.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
```

```
z = peaks(x,y);
ribbon(y,z)
colormap hsv
```



See Also

`plot`, `plot3`, `surface`, `waterfall`

“Polygons and Surfaces” on page 1-90 for related functions

rmappdata

Purpose

Remove application-defined data

Syntax

```
rmappdata(h,name)
```

Description

`rmappdata(h,name)` removes the application-defined data `name` from the object specified by handle `h`.

See Also

`getappdata`, `isappdata`, `setappdata`

Purpose

Remove directory

Graphical Interface

As an alternative to the `rmdir` function, use the delete feature in the "Current Directory Browser".

Syntax

```
rmdir('dirname')
rmdir('dirname','s')
[status, message, messageid] = rmdir('dirname','s')
```

Description

`rmdir('dirname')` removes the directory `dirname` from the current directory. If the directory is not empty, you must use the **`s`** argument. If `dirname` is not in the current directory, specify the relative path to the current directory or the full path for `dirname`.

`rmdir('dirname','s')` removes the directory `dirname` and its contents from the current directory. This removes all subdirectories and files in the current directory regardless of their write permissions.

`[status, message, messageid] = rmdir('dirname','s')` removes the directory `dirname` and its contents from the current directory, returning the status, a message, and the MATLAB error message ID (see `error` and `lasterror`). Here, `status` is `1` for success and is `0` for error, and `message`, `messageid`, and the **`s`** input argument are optional.

Remarks

When attempting to remove multiple directories, either by including a wildcard in the directory name or by specifying the `'s'` flag in the `rmdir` command, MATLAB throws an error if it is unable remove all directories to which the command applies. The error message contains a listing of those directories and files that MATLAB could not remove.

Examples

Remove Empty Directory

To remove `myfiles` from the current directory, where `myfiles` is empty, type

```
rmdir('myfiles')
```

If the current directory is `matlabr13/work`, and `myfiles` is in `d:/matlabr13/work/project/`, use the relative path to `myfiles`

```
rmdir('project/myfiles')
```

or the full path to `myfiles`

```
rmdir('d:/matlabr13/work/project/myfiles')
```

Remove Directory and All Contents

To remove `myfiles`, its subdirectories, and all files in the directories, assuming `myfiles` is in the current directory, type

```
rmdir('myfiles','s')
```

Remove Directory and Return Results

To remove `myfiles` from the current directory, type

```
[stat, mess, id]=rmdir('myfiles')
```

MATLAB returns

```
stat =
     0

mess =

The directory is not empty.

id =

MATLAB:RMDIR:OSError
```

indicating the directory `myfiles` is not empty.

To remove `myfiles` and its contents, run

```
[stat, mess]=rmdir('myfiles','s')
```

and MATLAB returns

```
stat =
     1

mess =

     ''
```

indicating `myfiles` and its contents were removed.

See Also

`cd`, `copyfile`, `delete`, `dir`, `error`, `fileattrib`, `filebrowser`, `lasterror`, `mkdir`, `movefile`

rmdir (ftp)

Purpose

Remove directory on FTP server

Syntax

```
rmdir(f,'dirname')
```

Description

`rmdir(f,'dirname')` removes the directory `dirname` from the current directory of the FTP server `f`, where `f` was created using `ftp`.

Examples

Connect to server `testsite`, view the contents of `testdir`, and remove the directory `newdir` from the directory `testdir`.

```
test=ftp('ftp.testsite.com');
cd(test,'testdir');
dir(test)
.               ..       newdir
dir(test,'newdir')
.               ..
rmdir(test,'newdir');
dir(test,'testdir')
.               ..
```

See Also

`cd (ftp)`, `delete (ftp)`, `dir (ftp)`, `ftp`, `mkdir (ftp)`

Purpose

Remove fields from structure

Syntax

```
s = rmfield(s, 'fieldname')
s = rmfield(s, fields)
```

Description

`s = rmfield(s, 'fieldname')` removes the specified field from the structure array `s`.

`s = rmfield(s, fields)` removes more than one field at a time. `fields` is a character array of field names or cell array of strings.

See Also

`fieldnames`, `setfield`, `getfield`, `isfield`, `orderfields`, "Using Dynamic Field Names"

rmpath

Purpose

Remove directories from MATLAB search path

GUI Alternatives

As an alternative to the `rmpath` function, use the Set Path dialog box. To open it, select **File > Set Path** in the MATLAB desktop.

Syntax

```
rmpath('directory')
rmpath directory
```

Description

`rmpath('directory')` removes the specified directory from the current MATLAB search path. Use the full pathname for `directory`.

`rmpath directory` is the command form of the syntax.

Examples

Remove `/usr/local/matlab/mytools` from the search path.

```
rmpath /usr/local/matlab/mytools
```

See Also

`addpath`, `cd`, `dir`, `genpath`, `matlabroot`, `partialpath`, `path`, `pathdef`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `savepath`, `what`

Search Path in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Remove preference

Syntax

```
rmpref('group','pref')
rmpref('group',{'pref1','pref2',...'prefn'})
rmpref('group')
```

Description

`rmpref('group','pref')` removes the preference specified by `group` and `pref`. It is an error to remove a preference that does not exist.

`rmpref('group',{'pref1','pref2',...'prefn'})` removes each preference specified in the cell array of preference names. It is an error if any of the preferences do not exist.

`rmpref('group')` removes all the preferences for the specified `group`. It is an error to remove a group that does not exist.

Examples

```
addpref('mytoolbox','version','1.0')
rmpref('mytoolbox')
```

See Also

`addpref`, `getpref`, `ispref`, `setpref`, `uigetpref`, `uisetpref`

root object

Purpose

Root object properties

Description

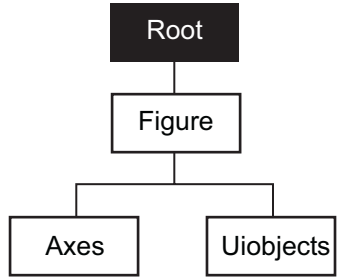
The root is a graphics object that corresponds to the computer screen. There is only one root object and it has no parent. The children of the root object are figures.

The root object exists when you start MATLAB; you never have to create it and you cannot destroy it. Use `set` and `get` to access the root properties.

See Also

`diary`, `echo`, `figure`, `format`, `gcf`, `get`, `set`

Object Hierarchy



Purpose

Root properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

Root Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`BusyAction`
`cancel | {queue}`

Not used by the root object.

`ButtonDownFcn`
string

Not used by the root object.

`CallbackObject`
handle (read only)

Handle of current callback's object. This property contains the handle of the object whose callback routine is currently executing. If no callback routines are executing, this property contains the empty matrix [ ]. See also the `gco` command.

`CaptureMatrix`
(obsolete)

This property has been superseded by the `getframe` command.

`CaptureRect`
(obsolete)

This property has been superseded by the `getframe` command.

`Children`
vector of handles

Handles of child objects. A vector containing the handles of all nonhidden figure objects (see `HandleVisibility` for more information). You can change the order of the handles and thereby change the stacking order of the figures on the display.

`Clipping`
`{on} | off`

Clipping has no effect on the root object.

`CommandWindowSize`
`[columns rows]`

Current size of command window. This property contains the size of the MATLAB command window in a two-element vector. The first element is the number of columns wide and the second element is the number of rows tall.

`CreateFcn`
The root does not use this property.

`CurrentFigure`
figure handle

Handle of the current figure window, which is the one most recently created, clicked in, or made current with the statement

```
figure(h)
```

which restacks the figure to the top of the screen, or

```
set(0,'CurrentFigure',h)
```

which does not restack the figures. In these statements, `h` is the handle of an existing figure. If there are no figure objects,

```
get(0,'CurrentFigure')
```

returns the empty matrix. Note, however, that `gcf` always returns a figure handle, and creates one if there are no figure objects.

`DeleteFcn`
string

This property is not used, because you cannot delete the root object.

`Diary`
`on | {off}`

Diary file mode. When this property is `on`, MATLAB maintains a file (whose name is specified by the `DiaryFile` property) that saves a copy of all keyboard input and most of the resulting output. See also the `diary` command.

`DiaryFile`
string

Diary filename. The name of the diary file. The default name is `diary`.

`Echo`
`on | {off}`

Script echoing mode. When `Echo` is `on`, MATLAB displays each line of a script file as it executes. See also the `echo` command.

`ErrorMessage`
string

Text of last error message. This property contains the last error message issued by MATLAB.

`FixedWidthFontName`
font name

Fixed-width font to use for axes, text, and uicontrols whose `FontName` *is set to* `FixedWidth`. MATLAB uses the font name specified for this property as the value for axes, text, and uicontrol `FontName` properties when their `FontName` property is set to `FixedWidth`. Specifying the font name with this property eliminates the need to hardcode font names in MATLAB applications and thereby enables these applications to run without modification in locales where non-ASCII character sets are required. In these cases, MATLAB attempts to set the value of `FixedWidthFontName` to the correct value for a given locale.

MATLAB application developers should not change this property, but should create axes, text, and uicontrols with `FontName` properties set to `FixedWidth` when they want to use a fixed-width font for these objects.

MATLAB end users can set this property if they do not want to use the preselected value. In locales where Latin-based characters are used, Courier is the default.

`Format`
`short | {shortE} | long | longE | bank | hex | + | rat`

Output format mode. This property sets the format used to display numbers. See also the `format` command.

- `short` — Fixed-point format with 5 digits
- `shortE` — Floating-point format with 5 digits
- `shortG` — Fixed- or floating-point format displaying as many significant figures as possible with 5 digits
- `long` — Scaled fixed-point format with 15 digits
- `longE` — Floating-point format with 15 digits

- `longG` — Fixed- or floating-point format displaying as many significant figures as possible with 15 digits
- `bank` — Fixed-format of dollars and cents
- `hex` — Hexadecimal format
- `+` — Displays + and – symbols
- `rat` — Approximation by ratio of small integers

`FormatSpacing`
`compact | {loose}`

Output format spacing (see also `format` command).

- `compact` — Suppress extra line feeds for more compact display.
- `loose` — Display extra line feeds for a more readable display.

`HandleVisibility`
`{on} | callback | off`

This property is not useful on the root object.

`HitTest`
`{on} | off`

This property is not useful on the root object.

`Interruptible`
`{on} | off`

This property is not useful on the root object.

`Language`
string

System environment setting.

`MonitorPosition`
`[x y width height;x y width height]`

Width and height of primary and secondary monitors, in pixels. This property contains the width and height of each monitor connnected to your computer. The `x` and `y` values for the primary monitor are `0`, `0` and the width and height of the monitor are specified in pixels.

The secondary monitor position is specified as

```
x = primary monitor width + 1
y = primary monitor height + 1
```

Querying the value of the figure `MonitorPosition` on a multiheaded system returns the position for each monitor on a separate line.

```
v = get(0,'MonitorPosition')
v =
 x y width height % Primary monitor
 x y width height % Secondary monitor
```

Note that MATLAB sets the value of the ScreenSize property to the combined size of the monitors.

`Parent`
handle

Handle of parent object. This property always contains the empty matrix, because the root object has no parent.

`PointerLocation`
`[x,y]`

Current location of pointer. A vector containing the *x*- and *y*-coordinates of the pointer position, measured from the lower left corner of the screen. You can move the pointer by changing the values of this property. The `Units` property determines the units of this measurement.

This property always contains the current pointer location, even if the pointer is not in a MATLAB window. A callback routine querying the `PointerLocation` can get a value different from the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

On Macintosh platforms, you cannot change the pointer location using the `set` command.

`PointerWindow`
handle (read only)

Handle of window containing the pointer. MATLAB sets this property to the handle of the figure window containing the pointer. If the pointer is not in a MATLAB window, the value of this property is 0. A callback routine querying the `PointerWindow` can get the wrong window handle if you move the pointer to another window before the callback executes. This error results from delays in callback execution caused by competition for system resources.

`RecursionLimit`
integer

Number of nested M-file calls. This property sets a limit to the number of nested calls to M-files MATLAB will make before stopping (or potentially running out of memory). By default the value is set to a large value. Setting this property to a smaller value (something like 150, for example) should prevent MATLAB from running out of memory and will instead cause MATLAB to issue an error when the limit is reached.

`ScreenDepth`
bits per pixel

Screen depth. The depth of the display bitmap (i.e., the number of bits per pixel). The maximum number of simultaneously displayed colors on the current graphics device is 2 raised to this power.

`ScreenDepth` supersedes the `BlackAndWhite` property. To override automatic hardware checking, set this property to `1`. This value causes MATLAB to assume the display is monochrome. This is useful if MATLAB is running on color hardware but is being displayed on a monochrome terminal. Such a situation can cause MATLAB to determine erroneously that the display is color.

`ScreenPixelsPerInch`
Display resolution

DPI setting for your display. This property contains the setting of your display resolution specified in your system preferences.

`ScreenSize`
four-element rectangle vector (read only)

Screen size. A four-element vector,

```
[left,bottom,width,height]
```

that defines the display size. `left` and `bottom` are `0` for all `Units` except `pixels`, in which case `left` and `bottom` are `1`. `width` and `height` are the screen dimensions in units specified by the `Units` property.

Determining Screen Size

Note that the screen size in absolute units (e.g., inches) is determined by dividing the number of pixels in width and height by the screen DPI (see the `ScreenPixelPerInch` property). This value is approximate and might not represent the actual size of the screen.

Note that the `ScreenSize` property is static. Its values are read only at MATLAB startup and not updated if system display settings change. Also, the values returned might not represent the usable screen size for application developers due to the presence of other GUIs, such as the Windows task bar.

`Selected`
`on | off`

This property has no effect on the root level.

`SelectionHighlight`
`{on} | off`

This property has no effect on the root level.

`ShowHiddenHandles`
`on | {off}`

Show or hide handles marked as hidden. When set to `on`, this property disables handle hiding and exposes all object handles regardless of the setting of an object's `HandleVisibility` property. When set to `off`, all objects so marked remain hidden within the graphics hierarchy.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. While it is not necessary to identify the root object with a tag (since its handle is always `0`), you can use this property to store any string value that you can later retrieve using `set`.

`Type`
string (read only)

Class of graphics object. For the root object, `Type` is always `'root'`.

`UIContextMenu`
handle

This property has no effect on the root level.

`Units`

```
{pixels} | normalized | inches | centimeters
  | points | characters
```

Unit of measurement. This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower left corner of the screen. Normalized units map the lower left corner of the screen to (0,0) and the upper right corner to (1.0,1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch). Characters are units defined by characters from the default system font; the width of one unit is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

This property affects the `PointerLocation` and `ScreenSize` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your operation, so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. This property can be any data you want to associate with the root object. MATLAB does not use this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Object visibility. This property has no effect on the root object.

Purpose

Polynomial roots

Syntax

```
r = roots(c)
```

Description

`r = roots(c)` returns a column vector whose elements are the roots of the polynomial `c`.

Row vector `c` contains the coefficients of a polynomial, ordered in descending powers. If `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$.

Remarks

Note the relationship of this function to `p = poly(r)`, which returns a row vector whose elements are the coefficients of the polynomial. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

The polynomial $s^3 - 6s^2 - 72s - 27$ is represented in MATLAB as

```
p = [1 -6 -72 -27]
```

The roots of this polynomial are returned in a column vector by

```
r = roots(p)

r =
    12.1229
    -5.7345
    -0.3884
```

Algorithm

The algorithm simply involves computing the eigenvalues of the companion matrix:

```
A = diag(ones(n-1,1),-1);
A(1,:) = -c(2:n+1)./c(1);
eig(A)
```

It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix `A`, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in `c`.

See Also

`fzero`, `poly`, `residue`

Purpose

Angle histogram plot

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
rose(theta)
rose(theta,x)
rose(theta,nbins)
rose(axes_handle,...)
h = rose(...)
[tout,rout] = rose(...)
```

Description

`rose(theta)` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range, showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle of each bin from the origin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0, 2*pi]`. The default is `20`.

`rose(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

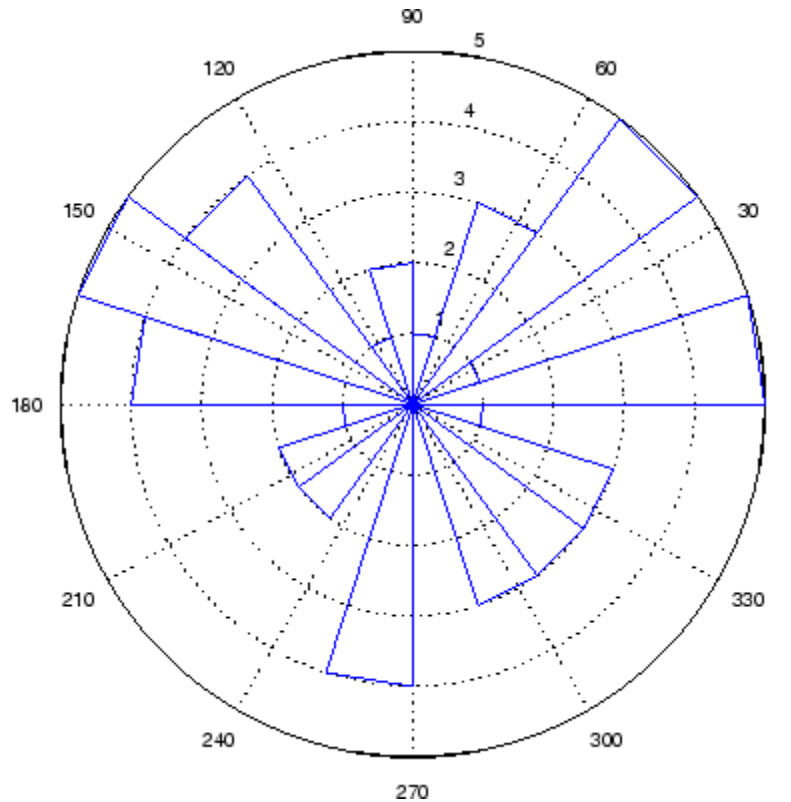
`h = rose(...)` returns the handles of the line objects used to create the graph.

`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

Example

Create a rose plot showing the distribution of 50 random numbers.

```
theta = 2*pi*rand(1,50);
rose(theta)
```



See Also

`compass`, `feather`, `hist`, `line`, `polar`

"Histograms" on page 1-90 for related functions

Histograms in Polar Coordinates for another example

Purpose

Classic symmetric eigenvalue test problem

Syntax

```
A = rosser
```

Description

`A = rosser` returns the Rosser matrix. This matrix was a challenge for many matrix eigenvalue algorithms. But LAPACK's `DSYEV` routine used in MATLAB has no trouble with it. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

Examples

```
rosser

ans =

   611   196  -192   407    -8   -52   -49    29
   196   899   113  -192   -71   -43    -8   -44
  -192   113   899   196    61    49     8    52
   407  -192   196   611     8    44    59   -23
    -8   -71    61     8   411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99
```

Purpose

Rotate matrix 90 degrees

Syntax

```
B = rot90(A)
B = rot90(A,k)
```

Description

`B = rot90(A)` rotates matrix `A` counterclockwise by 90 degrees.

`B = rot90(A,k)` rotates matrix `A` counterclockwise by `k*90` degrees, where `k` is an integer.

Examples

The matrix

```
X =
     1     2     3
     4     5     6
     7     8     9
```

rotated by 90 degrees is

```
Y = rot90(X)
Y =
     3     6     9
     2     5     8
     1     4     7
```

See Also

`flipdim`, `fliplr`, `flipud`

Purpose

Rotate object in specified direction

Syntax

```
rotate(h,direction,alpha)
rotate(...,origin)
```

Description

The `rotate` function rotates a graphics object in three-dimensional space, according to the right-hand rule.

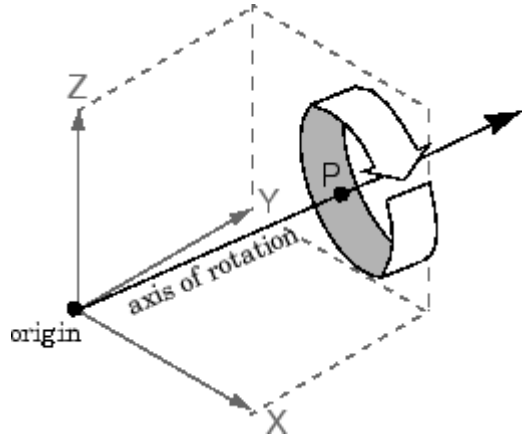
`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector. The default origin is the center of the plot box.

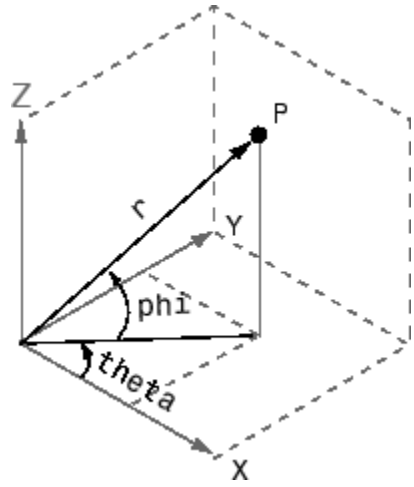
Remarks

The graphics object you want rotated must be a child of the same axes. The object's data is modified by the rotation transformation. This is in contrast to `view` and `rotate3d`, which only modify the viewpoint.

The axis of rotation is defined by an origin and a point *P* relative to the origin. *P* is expressed as the spherical coordinates `[theta phi]` or as Cartesian coordinates.



The two-element form for `direction` specifies the axis direction using the spherical coordinates `[theta phi]`. `theta` is the angle in the *x*-*y* plane counterclockwise from the positive *x*-axis. `phi` is the elevation of the direction vector from the *x*-*y* plane.



The three-element form for `direction` specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin to (X,Y,Z).

Examples

Rotate a graphics object 180º about the *x*-axis.

```
h = surf(peaks(20));
rotate(h,[1 0 0],180)
```

Rotate a surface graphics object 45º about its center in the *z* direction.

```
h = surf(peaks(20));
zdir = [0 0 1];
center = [10 10 0];
rotate(h,zdir,45,center)
```

Remarks

`rotate` changes the `Xdata`, `Ydata`, and `Zdata` properties of the appropriate graphics object.

See Also

`rotate3d`, `sph2cart`, `view`

The axes `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`

“Object Manipulation” on page 1-100 for related functions

Purpose

Rotate 3-D view using mouse

GUI Alternatives

Use the Rotate3D tool  on the figure toolbar to enable and disable rotate3D mode on a plot, or select **Rotate 3D** from the figure's **Tools** menu. For details, see "Rotate 3D — Interactive Rotation of 3-D Views" in the MATLAB Graphics documentation.

Syntax

```
rotate3d
rotate3d
rotate3d
rotate3d(figure_handle,...)
rotate3d(axes_handle,...)
h = rotate3d(figure_handle)
```

Description

`rotate3d on` enables mouse-base rotation on all axes within the current figure.

`rotate3d off` disables interactive axes rotation in the current figure.

`rotate3d` toggles interactive axes rotation in the current figure.

`rotate3d(figure_handle,...)` enables rotation within the specified figure instead of the current figure.

`rotate3d(axes_handle,...)` enables rotation only in the specified axes.

`h = rotate3d(figure_handle)` returns a rotate3d *mode object* for figure figure_handle for you to customize the mode's behavior.

Using Rotate Mode Objects

You access the following properties of rotate mode objects via `get` and modify some of them using `set`:

`FigureHandle <handle>`

The associated figure handle. This read-only property cannot be set.

`Enable` `'on'|'off'`

Specifies whether this figure mode is currently enabled on the figure.

RotateStyle `'orbit'|'box'`

Sets the method of rotation. `'orbit'` rotates the entire axes; `'box'` rotates a plot-box outline of the axes.

`ButtonDownFilter <function_handle>`

The application can inhibit the rotate operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function [res] = myfunction(obj,event_obj)
% OBJ        handle to the object that has been clicked on.
% EVENT_OBJ  handle to event object (empty in this release).
% RES        a logical flag to determine whether the rotate
             operation should take place or the
             'ButtonDownFcn' property of the object should
             take precedence.
```

`ActionPreCallback <function_handle>`

Set this callback to listen to when a rotate operation will start. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function myfunction(obj,event_obj)
% obj        handle to the figure that has been clicked on.
% event_obj  handle to event object.
```

The event object has the following read-only property:

`Axes`	The handle of the axes that is being rotated.

`ActionPostCallback <function_handle>`

Set this callback to listen to when a rotate operation has finished. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on.
% event_obj   handle to event object. The object has the same
              properties as the EVENT_OBJ of the
              'ActionPreCallback' callback.
```

```
flags = isAllowAxesRotate(h,axes)
```

Calling the function `isAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a rotate operation is permitted on the axes objects.

```
setAllowAxesRotate(h,axes,flag)
```

Calling the function `setAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a rotate operation on the axes objects.

Examples

Example 1

Simple 3-D rotation

```
surf(peaks);
rotate3d on
% rotate the plot using the mouse pointer.
```

Example 2

Rotate the plot using the "Plot Box" rotate style:

```
surf(peaks);
h = rotate3d;
set(h,'RotateStyle','box','Enable','on');
% Rotate the plot.
```

Example 3

Create two axes as subplots and then prevent one from rotating:

```
ax1 = subplot(1,2,1);
surf(peaks);
h = rotate3d;
ax2 = subplot(1,2,2);
surf(membrane);
setAllowAxesRotate(h,ax2,false);
% rotate the plots.
```

Example 4

Create a buttonDown callback for rotate mode objects to trigger. Copy the following code to a new M-file, execute it, and observe rotation behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = rotate3d;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse-click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5

Create callbacks for pre- and post-buttonDown events for rotate3D mode objects to trigger. Copy the following code to a new M-file, execute it, and observe rotation behavior:

```
function demo
% Listen to rotate events
surf(peaks);
h = rotate3d;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A rotation is about to occur.');
%
function mypostcallback(obj,evd)
newView = round(get(evd.Axes,'View'));
msgbox(sprintf('The new view is [%d %d].',newView));
```

Remarks

When enabled, `rotate3d` provides continuous rotation of axes and the objects it contains through mouse movement. A numeric readout appears in the lower left corner of the figure during rotation, showing the current azimuth and elevation of the axes. Releasing the mouse button removes the animated box and the readout.

You can also enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

You can create a rotate3D mode object once and use it to customize the behavior of different axes, as example 3 illustrates. You can also change its callback functions on the fly.

When you assign different 3-D rotation behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse will carry over

to the linked axes, regardless of the behavior you previously set for the other axes.

See Also

`camorbit`, `pan`, `rotate`, `view`, `zoom`

Object Manipulation for related functions

Purpose

Round to nearest integer

Syntax

```
Y = round(X)
```

Description

`Y = round(X)` rounds the elements of `X` to the nearest integers. For complex `X`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000            -0.2000             3.4000             5.6000
  Columns 5 through 6
   7.0000             2.4000 + 3.6000i

round(a)

ans =
  Columns 1 through 4
  -2.0000                  0             3.0000             6.0000
  Columns 5 through 6
   7.0000             2.0000 + 4.0000i
```

See Also

`ceil`, `fix`, `floor`

Purpose

Reduced row echelon form

Syntax

```
R = rref(A)
[R,jb] = rref(A)
[R,jb] = rref(A,tol)
```

Description

`R = rref(A)` produces the reduced row echelon form of `A` using Gauss Jordan elimination with partial pivoting. A default tolerance of `(max(size(A))*eps *norm(A,inf))` tests for negligible column elements.

`[R,jb] = rref(A)` also returns a vector `jb` such that:

- `r = length(jb)` is this algorithm's idea of the rank of `A`.
- `x(jb)` are the pivot variables in a linear system `Ax = b`.
- `A(:,jb)` is a basis for the range of `A`.
- `R(1:r,jb)` is the `r`-by-`r` identity matrix.

`[R,jb] = rref(A,tol)` uses the given tolerance in the rank tests.

Roundoff errors may cause this algorithm to compute a different value for the rank than `rank`, `orth` and `null`.

Examples

Use `rref` on a rank-deficient magic square:

```
A = magic(4), R = rref(A)

A =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1

R =
    1    0    0    1
    0    1    0    3
```

```
     0     0     1    -3
     0     0     0     0
```

See Also

inv, lu, rank

Purpose

Convert real Schur form to complex Schur form

Syntax

```
[U,T] = rsf2csf(U,T)
```

Description

The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

`[U,T] = rsf2csf(U,T)` converts the real Schur form to the complex form.

Arguments `U` and `T` represent the unitary and Schur forms of a matrix `A`, respectively, that satisfy the relationships: `A = U*T*U'` and `U'*U = eye(size(A))`. See `schur` for details.

Examples

Given matrix `A`,

```
 1     1     1     3
 1     2     1     1
 1     1     3     1
-2     1     1     4
```

with the eigenvalues

```
4.8121    1.9202 + 1.4742i    1.9202 + 1.4742i    1.3474
```

Generating the Schur form of `A` and converting to the complex Schur form

```
[u,t] = schur(A);
[U,T] = rsf2csf(u,t)
```

yields a triangular matrix `T` whose diagonal (underlined here for readability) consists of the eigenvalues of `A`.

```
U =
```

```
-0.4916       -0.2756 - 0.4411i    0.2133 + 0.5699i   -0.3428
-0.4980       -0.1012 + 0.2163i   -0.1046 + 0.2093i    0.8001
-0.6751        0.1842 + 0.3860i   -0.1867 - 0.3808i   -0.4260
-0.2337        0.2635 - 0.6481i    0.3134 - 0.5448i    0.2466

T =

4.8121       -0.9697 + 1.0778i   -0.5212 + 2.0051i   -1.0067
     0        1.9202 + 1.4742i    2.3355              0.1117 + 1.6547i
     0             0              1.9202 - 1.4742i    0.8002 + 0.2310i
     0             0                   0              1.3474
```

See Also

schur

run

Purpose

Run script that is not on current path

Syntax

```
run scriptname
```

Description

`run scriptname` runs the MATLAB script specified by `scriptname`. If `scriptname` contains the full pathname to the script file, then `run` changes the current directory to be the one in which the script file resides, executes the script, and sets the current directory back to what it was. The script is run within the caller's workspace.

`run` is a convenience function that runs scripts that are not currently on the path. Typically, you just type the name of a script at the MATLAB prompt to execute it. This works when the script is on your path. Use the `cd` or `addpath` function to make a script executable by entering the script name alone.

See Also

`cd`, `addpath`

Purpose

Save workspace variables to disk

Graphical Interface

As an alternative to the `save` function, select **Save Workspace As** from the **File** menu in the MATLAB desktop, or use the Workspace browser.

Syntax

```
save
save filename
save filename content
save filename options
save filename content options
save('filename', 'var1', 'var2', ...)
```

Description

`save` stores all variables from the current MATLAB workspace in a MATLAB-formatted file (MAT-file) named `matlab.mat` that resides in the current working directory. Use the `load` function to retrieve data stored in MAT-files. By default, MAT-files are double-precision, binary files. You can create a MAT-file on one machine and then load it on another machine using a different floating-point format, and retaining as much accuracy and range as the different formats allow. MAT-files can also be manipulated by other programs external to MATLAB.

`save filename` stores all variables in the current workspace in the file `filename`. If you do not specify an extension to the filename, MATLAB uses `.mat`. The file must be writable. To save to another directory, use a full pathname for the `filename`.

`save filename content` stores only those variables specified by `content` in file `filename`. If `filename` is not specified, MATLAB stores the data in a file called `matlab.mat`. See the following table.

Values for *content*	Description
`varlist`	Save only those variables that are in `varlist`. You can use the `*` wildcard to save only those variables that match the specified pattern. For example, `save('A*')` saves all variables that start with `A`.
`-regexp` `exprlist`	Save those variables that match any of the regular expressions in `exprlist`.
`-struct` `s`	Save as individual variables all fields of the scalar structure `s`.
`-struct` `s fieldlist`	Save as individual variables only the specified fields of structure `s`.

In this table, the terms `varlist`, `exprlist`, and `fieldlist` refer to one or more variable names, regular expressions, or structure field names separated by either spaces or commas, depending on whether you are using the MATLAB command or function format. See the examples below:

Command format:

```
save firstname lastname street town
```

Function format:

```
save('firstname', 'lastname', 'street', 'town')
```

`save filename` *`options`* stores all variables from the MATLAB workspace in file `filename` according to one or more of the following options. If `filename` is not specified, MATLAB stores the data in a file called `matlab.mat`.

Values for *options*	Description
-append	Add new variables to those already stored in an existing MAT-file.
-format	Save using the specified binary or ASCII format. See the section on, "MAT-File Format Options" on page 2-2829, below.
-version	Save in a format that can be loaded into an earlier version of MATLAB. See the section on "Version Compatibility Options" on page 2-2830, below.

`save filename` *`content options`* stores only those variables specified by *`content`* in file `filename`, also applying the specified *`options`*. If `filename` is not specified, MATLAB stores the data in a file called `matlab.mat`.

`save('filename', 'var1', 'var2', ...)` is the function form of the syntax.

MAT-File Format Options

The following table lists the valid *`MAT-file format`* options.

MAT-file format Options	How Data Is Stored
`-ascii`	Save data in 8-digit ASCII format.
`-ascii -tabs`	Save data in 8-digit ASCII format delimited with tabs.
`-ascii -double`	Save data in 16-digit ASCII format.
`-ascii -double -tabs`	Save data in 16-digit ASCII format delimited with tabs.
`-mat`	Binary MAT-file form (default).

Version Compatibility Options

The following table lists version compatibility options. These options enable you to save your workspace data to a MAT-file that can then be loaded into an earlier version of MATLAB. The resulting MAT-file supports only those data items and features that were available in this earlier version of MATLAB. (See the second table below for what is supported in each version.)

version Option	Use When Running ...	To Save a MAT-File That You Can Load In ...
`-v7.3`	Version 7.3 or later	Version 7.3 or later
`-v7`	Version 7.3 or later	Versions 7.0 through 7.2 (or later)
`-v6`	Version 7 or later	Versions 5 and 6 (or later)
`-v4`	Version 5 or later	Versions 1 through 4 (or later)

The default version option is the value specified in the **Preferences** dialog box. Select **File > Preferences** in the Command Window, click **General**, and then **MAT-Files** to view or change the default.

The next table shows what data items and features are supported in different versions of MATLAB. You can use this information to determine which of the version compatibility options shown above to use.

MATLAB Versions	Data Items or Features Supported
4 and earlier	Support for 2D double, character, and sparse
5 and 6	Version 4 capability plus support for ND arrays, structs, and cells

MATLAB Versions	Data Items or Features Supported
7.0 through 7.2	Version 6 capability plus support for data compression and Unicode character encoding
7.3 and later	Version 7.2 capability plus support for data items greater than or equal to 2GB

Remarks

When working on 64-bit platforms, you can have data items in your workspace that occupy more than 2 GB. To save data of this size, you must use the HDF5-based version of the MATLAB MAT-file. Use the `v7.3` option to do this:

```
save -v7.3 myfile v1 v2
```

If you are running MATLAB on a 64-bit computer system and you attempt to save a variable that is too large for a version 7 (or earlier) MAT-file, that is, you save without using the `-v7.3` option, MATLAB skips that variable during the save operation and issues a warning message to that effect.

If you are running MATLAB on a 32-bit computer system and attempt to load a variable from a `-v7.3` MAT-file that is too large to fit in 32–bit address space, MATLAB skips that variable and issues a warning message to that effect.

MAT-files saved with compression and Unicode encoding cannot be loaded into versions of MATLAB prior to MATLAB Version 7.0. If you save data to a MAT-file that you intend to load using MATLAB Version 6 or earlier, you must specify the `-v6` option when saving. This disables compression and Unicode encoding for that particular `save` operation.

If you want to save to a file that you can then load into a Version 4 MATLAB session, you must use the `-v4` option when saving. When you use this option, variables that are incompatible with MATLAB Version 4 are not saved to the MAT-file. For example, ND arrays, structs, cells, etc. cannot be saved to a MATLAB Version 4 MAT-file. Also, variables with names that are longer than 19 characters cannot be saved to a MATLAB Version 4 MAT-file.

For information on any of the following topics related to saving to MAT-files, see in the MATLAB Programming documentation:

- Appending variables to an existing MAT-file
- Compressing data in the MAT-file
- Saving in ASCII format
- Saving in MATLAB Version 4 format
- Saving with Unicode character encoding
- Data storage requirements
- Saving from external programs

For information on saving figures, see the documentation for `hgsave` and `saveas`. For information on exporting figures to other graphics formats, see the documentation for `print`.

Examples

Example 1

Save all variables from the workspace in binary MAT-file `test.mat`:

```
save test.mat
```

Example 2

Save variables `p` and `q` in binary MAT-file `test.mat`.

In this example, the file name is stored in a variable, `savefile`. You must call `save` using the function syntax of the command if you intend to reference the file name through a variable.

```
savefile = 'test.mat';
p = rand(1, 10);
q = ones(10);
save(savefile, 'p', 'q')
```

Example 3

Save the variables `vol` and `temp` in ASCII format to a file named `june10`:

```
save('d:\mymfiles\june10','vol','temp','-ASCII')
```

Example 4

Save the fields of structure `s1` as individual variables rather than as an entire structure.

```
s1.a = 12.7;  s1.b = {'abc', [4 5; 6 7]};  s1.c = 'Hello!';
save newstruct.mat -struct s1;
clear
```

Check what was saved to `newstruct.mat`:

```
whos -file newstruct.mat
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  b         1x2                       158  cell array
  c         1x6                        12  char array

Grand total is 16 elements using 178 bytes
```

Read only the `b` field into the MATLAB workspace.

```
str = load('newstruct.mat', 'b')
str =
    b: {'abc'  [2x2 double]}
```

Example 5

Using regular expressions, save in MAT-file `mydata.mat` those variables with names that begin with `Mon`, `Tue`, or `Wed`:

```
save('mydata', '-regexp', '^Mon|^Tue|^Wed');
```

Here is another way of doing the same thing. In this case, there are three separate expression arguments:

```
save('mydata', '-regexp', '^Mon', '^Tue', '^Wed');
```

Example 6

Save a 3000-by-3000 matrix uncompressed to file `c1.mat`, and compressed to file `c2.mat`. The compressed file uses about one quarter the disk space required to store the uncompressed data:

```
x = ones(3000);
y = uint32(rand(3000) * 100);

save -v6 c1 x y     % Save without compression
save -v7 c2 x y     % Save with compression

d1 = dir('c1.mat');
d2 = dir('c2.mat');

d1.bytes
ans =
    45000240          % Size of the uncompressed data in bytes.
d2.bytes
ans =
    11985283          % Size of the compressed data in bytes.

d2.bytes/d1.bytes
ans =
    0.2663            % Ratio of compressed to uncompressed
```

See Also

`load`, `clear`, `diary`, `fprintf`, `fwrite`, `genvarname`, `who`, `whos`, `workspace`, `regexp`

Purpose

Serialize control object to file

Syntax

```
h.save('filename')
save(h, 'filename')
```

Description

`h.save('filename')` saves the COM control object, `h`, to the file specified in the string, `filename`.

`save(h, 'filename')` is an alternate syntax for the same operation.

Note The COM `save` function is only supported for controls at this time.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.save('mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```
h.load('mwsample');
h.get
ans =
     Label: 'Label'
    Radius: 20
```

See Also

`load`, `actxcontrol`, `actxserver`, `release`, `delete`

save (serial)

Purpose

Save serial port objects and variables to MAT-file

Syntax

```
save filename
save filename obj1 obj2...
```

Arguments

`filename`	The MAT-file name.
`obj1` `obj2...`	Serial port objects or arrays of serial port objects.

Description

`save filename` saves all MATLAB variables to the MAT-file `filename`. If an extension is not specified for `filename`, then the `.mat` extension is used.

`save filename obj1 obj2...` saves the serial port objects `obj1` `obj2...` to the MAT-file `filename`.

Remarks

You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and serial port objects as strings. For example. to save the serial port object `s` to the file `MySerial.mat`

```
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the serial port object is not automatically stored in the MAT-file. For example, suppose there is data in the input buffer for `obj`. To save that data to a MAT-file, you must bring it into the MATLAB workspace using one of the synchronous read functions, and then save to the MAT-file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

Example

This example illustrates how to use the command and functional form of save.

```
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

See Also

Functions

load, record

Properties

Status

Purpose

Save figure or Simulink block diagram using specified format

GUI Alternative

Use **File —> Save As** on the figure window menu to access the Save As dialog, in which you can select a graphics format. For details, see "Exporting in a Specific Graphics Format" in the MATLAB Graphics documentation. Note that sizes of files written to image formats by this GUI and by `saveas` can differ, due to disparate resolution settings.

Syntax

```
saveas(h,'filename.ext')
saveas(h,'filename','format')
```

Description

`saveas(h,'filename.ext')` saves the figure or Simulink block diagram with the handle `h` to the file `filename.ext`. The format of the file is determined by the extension, `ext`. Allowable values for `ext` are listed in this table.

You can pass the handle of any Handle Graphics object to `saveas`, which then saves the parent figure to the object you specified should `h` not be a figure handle. This means that `saveas` cannot save a subplot without also saving all subplots in its parent figure.

ext Value	Format
`ai`	Adobe Illustrator '88
`bmp`	Windows bitmap
`emf`	Enhanced metafile
`eps`	EPS Level 1
`fig`	MATLAB figure (invalid for Simulink block diagrams)
`jpg`	JPEG image (invalid for Simulink block diagrams)
`m`	MATLAB M-file (invalid for Simulink block diagrams)
`pbm`	Portable bitmap

ext Value	Format
pcx	Paintbrush 24-bit
pdf	Portable Document Format
pgm	Portable Graymap
png	Portable Network Graphics
ppm	Portable Pixmap
tif	TIFF image, compressed

`saveas(h,'filename','format')` saves the figure or Simulink block diagram with the handle `h` to the file called `filename` using the specified `format`. The filename can have an extension, but the extension is not used to define the file format. If no extension is specified, the standard extension corresponding to the specified format is automatically appended to the filename.

Allowable values for `format` are the extensions in the table above and the device drivers and graphic formats supported by `print`. The drivers and graphic formats supported by `print` include additional file formats not listed in the table above. When using a `print` device type to specify format for `saveas`, do not prefix it with `-d`.

Remarks

You can use `open` to open files saved using `saveas` with an `m` or `fig` extension. Other `saveas` and `print` formats are not supported by `open`. Both the **Save As** and **Export** dialog boxes that you access from a figure's **File** menu use `saveas` with the `format` argument, and support all device and file types listed above.

If you want to control the size or resolution of figures saved in image (bitmapped) formats (such as `BMP` or `JPG`), use the `print` command and specify dots-per-inch resolution with the `r` switch.

Examples

Example 1: Specify File Extension

Save the current figure that you annotated using the Plot Editor to a file named `pred_prey` using the MATLAB `fig` format. This allows you to open the file `pred_prey.fig` at a later time and continue editing it with the Plot Editor.

```
saveas(gcf,'pred_prey.fig')
```

Example 2: Specify File Format but No Extension

Save the current figure, using Adobe Illustrator format, to the file `logo`. Use the `ai` extension from the above table to specify the format. The file created is `logo.ai`.

```
saveas(gcf,'logo', 'ai')
```

This is the same as using the Adobe Illustrator format from the print devices table, which is `-dill`; use `doc print` or `help print` to see the table for print device types. The file created is `logo.ai`. MATLAB automatically appends the `ai` extension for an Illustrator format file because no extension was specified.

```
saveas(gcf,'logo', 'ill')
```

Example 3: Specify File Format and Extension

Save the current figure to the file `star.eps` using the Level 2 Color PostScript format. If you use `doc print` or `help print`, you can see from the table for print device types that the device type for this format is `-dpsc2`. The file created is `star.eps`.

```
saveas(gcf,'star.eps', 'psc2')
```

In another example, save the current Simulink block diagram to the file `trans.tiff` using the TIFF format with no compression. From the table for print device types, you can see that the device type for this format is `-dtiffn`. The file created is `trans.tiff`.

```
saveas(gcf,'trans.tiff', 'tiffn')
```

See Also

`hgsave`, `open`, `print`

"Printing" on page 1-92 for related functions

Simulink users, see also `save_system`

saveobj

Purpose

User-defined extension of save function for user objects

Syntax

```
B = saveobj(A)
```

Description

`B = saveobj(A)` is called by the MATLAB `save` function when object `A` is saved to a MAT-file. This call executes the `saveobj` method for the object's class, if such a method exists. The return value `B` is subsequently used by `save` to populate the MAT-file.

When you issue a `save` command on an object, MATLAB looks for a method called `saveobj` in the class directory. You can overload this method to modify the object before the save operation. For example, you could define a `saveobj` method that saves related data along with the object.

Remarks

`saveobj` can be overloaded only for user objects. `save` will not call `saveobj` for a built-in datatype, such as `double`, even if `@double/saveobj` exists.

`saveobj` will be separately invoked for each object to be saved.

A child object does not inherit the `saveobj` method of its parent class. To implement `saveobj` for any class, including a class that inherits from a parent, you must define a `saveobj` method within that class directory.

Examples

The following example shows a `saveobj` method written for the `portfolio` class. The method determines if a `portfolio` object has already been assigned an account number from a previous `save` operation. If not, `saveobj` calls `getAccountNumber` to obtain the number and assigns it to the `account_number` field. The contents of `b` is saved to the MAT-file.

```
function b = saveobj(a)
if isempty(a.account_number)
    a.account_number = getAccountNumber(a);
end
b = a;
```

See Also

save, load, loadobj

savepath

Purpose

Save current MATLAB search path to `pathdef.m` file

GUI Alternatives

As an alternative to the `savepath` function, use the Set Path dialog box. To open it, select **File > Set Path** in the MATLAB desktop.

Syntax

```
savepath
savepath newfile
```

Description

`savepath` saves the current MATLAB search path to `pathdef.m`. It returns

`0`	If the file was saved successfully
`1`	If the save failed

`savepath newfile` saves the current MATLAB search path to `newfile`, where `newfile` is in the current directory or is a relative or absolute path.

Examples

The statement

```
savepath myfiles/pathdef.m
```

saves the current search path to the file `pathdef.m`, which is located in the `myfiles` directory in the MATLAB current directory.

Consider using `savepath` in your MATLAB `finish.m` file to save the path when you exit MATLAB.

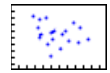
See Also

`addpath`, `cd`, `dir`, `finish`, `genpath`, `matlabroot`, `partialpath`, `pathdef`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`, `startup`, `what`

Search Path in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Scatter plot



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
scatter(X,Y,S,C)
scatter(X,Y)
scatter(X,Y,S)
scatter(...,markertype)
scatter(...,'filled')
scatter(...,'PropertyName',propertyvalue)
scatter(axes_handles,...)
h = scatter(...)
hpatch = scatter('v6',...)
```

Description

`scatter(X,Y,S,C)` displays colored circles at the locations specified by the vectors `X` and `Y` (which must be the same size).

`S` determines the area of each marker (specified in points^2). `S` can be a vector the same length as `X` and `Y` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size. If `S` is empty, the default size is used.

`C` determines the color of each marker. When `C` is a vector the same length as `X` and `Y`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a `length(X)`-by-3 matrix, it specifies the colors of the markers as RGB values. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers).

`scatter(X,Y)` draws the markers in the default size and color.

`scatter(X,Y,S)` draws the markers at the specified sizes (`S`) with a single color. This type of graph is also known as a bubble plot.

`scatter(...,markertype)` uses the marker type specified instead of `'o'` (see `LineSpec` for a list of marker specifiers).

`scatter(...,'filled')` fills the markers.

`scatter(...,'PropertyName',propertyvalue)` creates the scatter graph, applying the specified property settings. See `scattergroup` properties for a description of properties.

`scatter(axes_handles,...)` plots into the axes object with handle `axes_handle` instead of the current axes object (`gca`).

`h = scatter(...)` returns the handle of the scattergroup object created.

Backward-Compatible Version

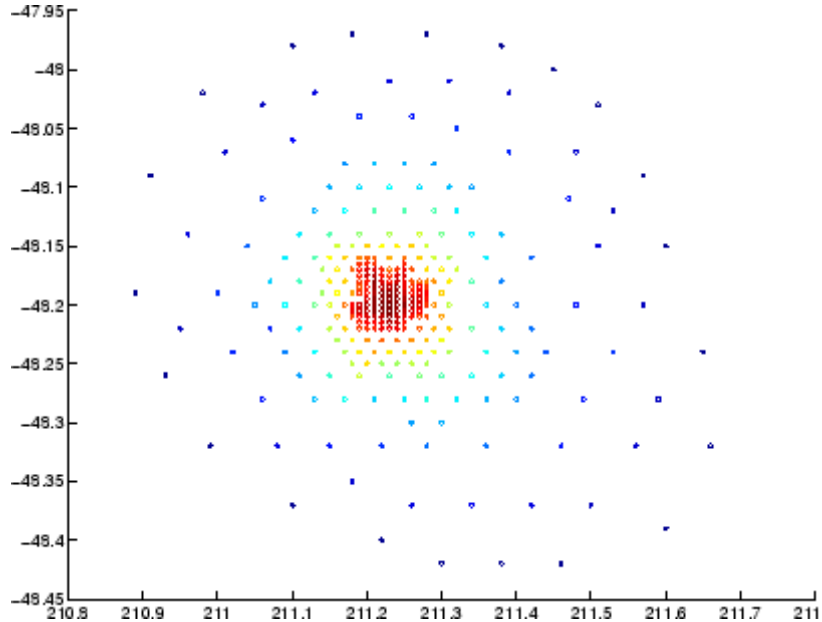
`hpatch = scatter('v6',...)` returns the handles to the patch objects created by `scatter` (see `Patch` Properties for a list of properties you can specify using the object handles and `set`).

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Example

```
load seamount
scatter(x,y,5,z)
```



See Also

`scatter3`, `plot3`

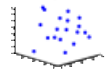
"Scatter/Bubble Plots" on page 1-91 for related functions

See Triangulation and Interpolation of Scatter Data for related information.

See `Scattergroup Properties` for property descriptions.

Purpose

3-D scatter plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
scatter3(X,Y,Z,S,C)
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(...,markertype)
scatter3(...,'filled')
scatter3(...,'PropertyName',propertyvalue)
h = scatter3(...)
hpatch = scatter3('v6',...)
```

Description

`scatter3(X,Y,Z,S,C)` displays colored circles at the locations specified by the vectors `X`, `Y`, and `Z` (which must all be the same size).

`S` determines the size of each marker (specified in points). `S` can be a vector the same length as `X`, `Y`, and `Z` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size.

`C` determines the colors of each marker. When `C` is a vector the same length as `X`, `Y`, and `Z`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a `length(X)`-by-3 matrix, it specifies the colors of the markers as RGB values. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers).

`scatter3(X,Y,Z)` draws the markers in the default size and color.

`scatter3(X,Y,Z,S)` draws markers at the specified sizes (`S`) in a single color.

`scatter3(...,markertype)` uses the marker type specified instead of 'o' (see `LineSpec` for a list of marker specifiers).

`scatter3(...,'filled')` fills the markers.

`scatter3(...,'PropertyName',propertyvalue)` creates the scatter graph, applying the specified property settings. See `scattergroup` properties for a description of properties.

`h = scatter3(...)` returns handles to the scattergroup objects created by `scatter3`. See `Scattergroup Properties` for property descriptions.

Backward-Compatible Version

`hpatch = scatter3('v6',...)` returns the handles to the patch objects created by `scatter3` (see `Patch` for a list of properties you can specify using the object handles and `set`).

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

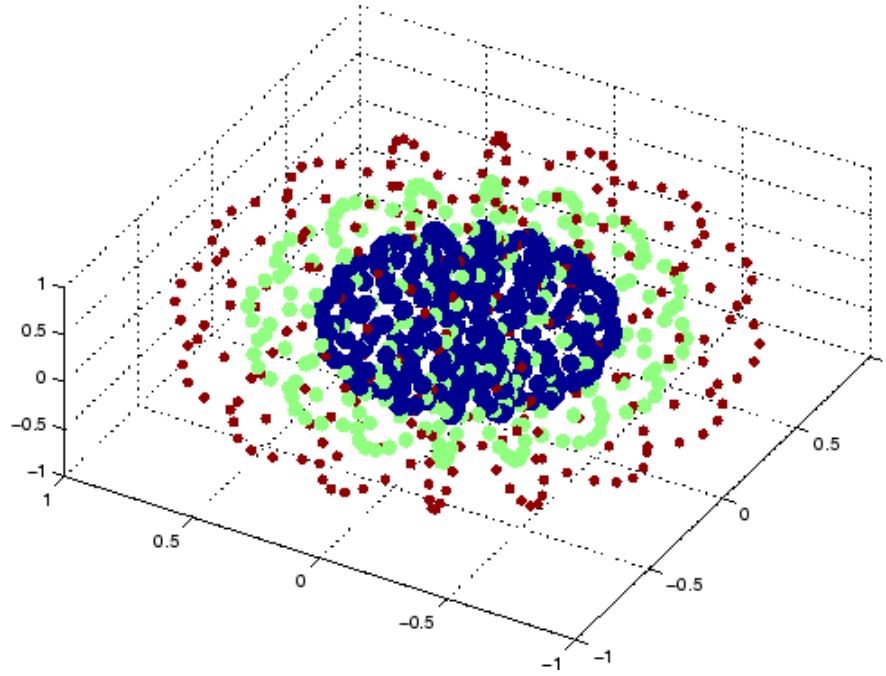
See Plot Objects and Backward Compatibility for more information.

Remarks

Use `plot3` for single color, single marker size 3-D scatter plots.

Examples

```
[x,y,z] = sphere(16);
X = [x(:)*.5 x(:)*.75 x(:)];
Y = [y(:)*.5 y(:)*.75 y(:)];
Z = [z(:)*.5 z(:)*.75 z(:)];
S = repmat([1 .75 .5]*10,prod(size(x)),1);
C = repmat([1 2 3],prod(size(x)),1);
scatter3(X(:),Y(:),Z(:),S(:),C(:),'filled'), view(-60,60)
```



See Also

`scatter`, `plot3`

See `Scattergroup Properties` for property descriptions

“Scatter/Bubble Plots” on page 1-91 for related functions

Purpose

Define scattergroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default property values for scattergroup objects.

See Plot Objects for information on scattergroup objects.

Scattergroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of scattergroup objects in legends. The `Annotation` property enables you to specify whether this scattergroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the scattergroup object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the scattergroup object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
`off`	Do not include the scattergroup or its children in a legend (default)
`children`	Include only the children of the scattergroup as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object’s `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file

- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

`CData`
vector, m-by-3 matrix, `ColorSpec`

Color of markers. When `CData` is a vector the same length as `XData` and `YData`, the values in `CData` are linearly mapped to the colors in the current colormap. When `CData` is a `length(XData)`-by-3 matrix, it specifies the colors of the markers as RGB values.

`CDataSource`
string (MATLAB variable)

Link YData to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `CData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `CData`.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this scattergroup object. The `legend` function uses the string defined by the `DisplayName` property to label this scattergroup object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this scattergroup object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EraseMode`

```
{normal} | none | xor | background
```

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in

the figure's CurrentObject property, and axes do not appear in their parent's CurrentAxes property.

Overriding Handle Visibility

You can set the root ShowHiddenHandles property to on to make all handles visible regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties). See also findall.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

HitTest

{on} | off

Selectable by mouse click. HitTest determines whether this object can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click on the objects that compose the area graph. If HitTest is off, clicking this object selects the object below it (which is usually the axes containing it).

HitTestArea

on | {off}

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none`

specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`SizeData`

square points

Size of markers in square points. This property specifies the area of the marker in the scatter graph in units of points. Since there are 72 points to one inch, to specify a marker that has an area of one square inch you would use a value of `72^2`.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For stemseries objects, `Type` is 'hggroup'. The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
array

X-coordinates of scatter markers. The scatter function draws individual markers at each *x*-axis location in the `XData` array. The

input argument `x` in the `scatter` function calling syntax assigns values to `XData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
scalar, vector, or matrix

Y-coordinates of scatter markers. The scatter function draws individual markers at each *y*-axis location in the `YData` array.

The input argument `y` in the `scatter` function calling syntax assigns values to `YData`.

`YDataSource`
string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`
vector of coordinates

Z-coordinates. A vector defining the *z*-coordinates for the graph. `XData` and `YData` must be the same length and have the same number of rows.

`ZDataSource`
string (MATLAB variable)

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `ZData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Schur decomposition

Syntax

```
T = schur(A)
T = schur(A,flag)
[U,T] = schur(A,...)
```

Description

The `schur` command computes the Schur form of a matrix.

`T = schur(A)` returns the Schur matrix `T`.

`T = schur(A,flag)` for real matrix A, returns a Schur matrix `T` in one of two forms depending on the value of `flag`:

`'complex'`	`T` is triangular and is complex if `A` has complex eigenvalues.
`'real'`	`T` has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. `'real'` is the default.

If `A` is complex, `schur` returns the complex Schur form in matrix `T`. The complex Schur form is upper triangular with the eigenvalues of `A` on the diagonal.

The function `rsf2csf` converts the real Schur form to the complex Schur form.

`[U,T] = schur(A,...)` also returns a unitary matrix `U` so that `A = U*T*U'` and `U'*U = eye(size(A))`.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H = [ -149    -50   -154
       537    180    546
       -27     -9    -25 ]
```

Its Schur form is

```
schur(H)
```

```
ans =
     1.0000   -7.1119 -815.8706
          0    2.0000  -55.0236
          0         0    3.0000
```

The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

Algorithm

Input of Type Double

If `A` has type `double`, `schur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

Matrix A	Routine
Real symmetric	`DSYTRD`, `DSTEQR` `DSYTRD`, `DORGTR`, `DSTEQR` (with output `U`)
Real nonsymmetric	`DGEHRD`, `DHSEQR` `DGEHRD`, `DORGHR`, `DHSEQR` (with output `U`)
Complex Hermitian	`ZHETRD`, `ZSTEQR` `ZHETRD`, `ZUNGTR`, `ZSTEQR` (with output `U`)
Non-Hermitian	`ZGEHRD`, `ZHSEQR` `ZGEHRD`, `ZUNGHR`, `ZHSEQR` (with output `U`)

Input of Type Single

If `A` has type `single`, `schur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

Matrix A	Routine
Real symmetric	SSYTRD, SSTEQR
	SSYTRD, SORGTR, SSTEQR (with output U)
Real nonsymmetric	SGEHRD, SHSEQR
	SGEHRD, SORGHR, SHSEQR (with output U)
Complex Hermitian	CHETRD, CSTEQR
	CHETRD, CUNGTR, CSTEQR (with output U)
Non-Hermitian	CGEHRD, CHSEQR
	CGEHRD, CUNGHR, CHSEQR (with output U)

See Also

eig, hess, qz, rsf2csf

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (http://www.netlib.org/lapack/lug/lapack_lug.html), Third Edition, SIAM, Philadelphia, 1999.

script

Purpose

Script M-file description

Description

A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, you can obtain subsequent MATLAB input from the file. Script files have a filename extension of `.m` and are often called M-files.

Scripts are the simplest kind of M-file. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, so you can use them in further computations. In addition, scripts can produce graphical output using commands like `plot`.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any M-file, scripts can contain comments. Any text following a percent sign (`%`) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also

`echo`, `function`, `type`

Purpose

Secant of argument in radians

Syntax

```
Y = sec(X)
```

Description

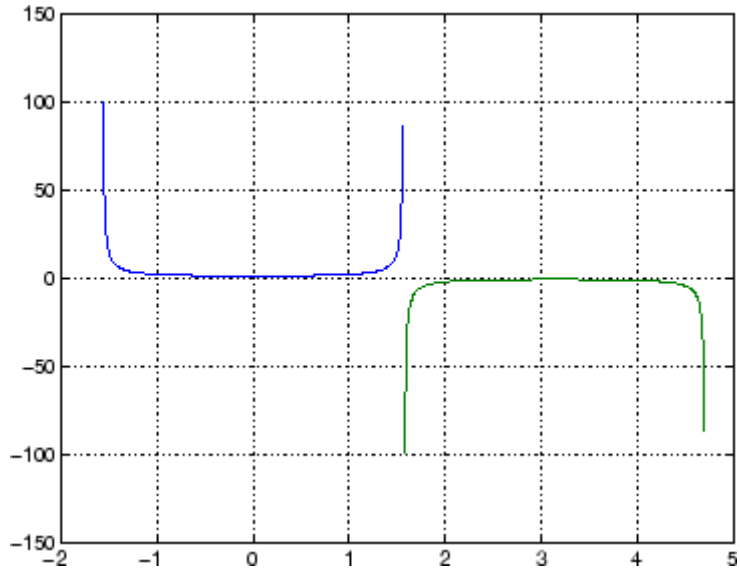
The `sec` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sec(X)` returns an array the same size as `X` containing the secant of the elements of `X`.

Examples

Graph the secant over the domains $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$.

```
x1 = -pi/2+0.01:0.01:pi/2-0.01;
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;
plot(x1,sec(x1),x2,sec(x2)), grid on
```



The expression `sec(pi/2)` does not evaluate as infinite but as the reciprocal of the floating-point accuracy `eps`, because `pi` is a floating-point approximation to the exact value of π.

Definition

The secant can be defined as

$$\sec(z) = \frac{1}{\cos(z)}$$

Algorithm

`sec` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`secd`, `sech`, `asec`, `asecd`, `asech`

Purpose

Secant of argument in degrees

Syntax

```
Y = secd(X)
```

Description

`Y = secd(X)` is the secant of the elements of `X`, expressed in degrees. For odd integers `n`, `secd(n*90)` is infinite, whereas `sec(n*pi/2)` is large but finite, reflecting the accuracy of the floating point value of `pi`.

See Also

`sec`, `sech`, `asec`, `asecd`, `asech`

Purpose

Hyperbolic secant

Syntax

```
Y = sech(X)
```

Description

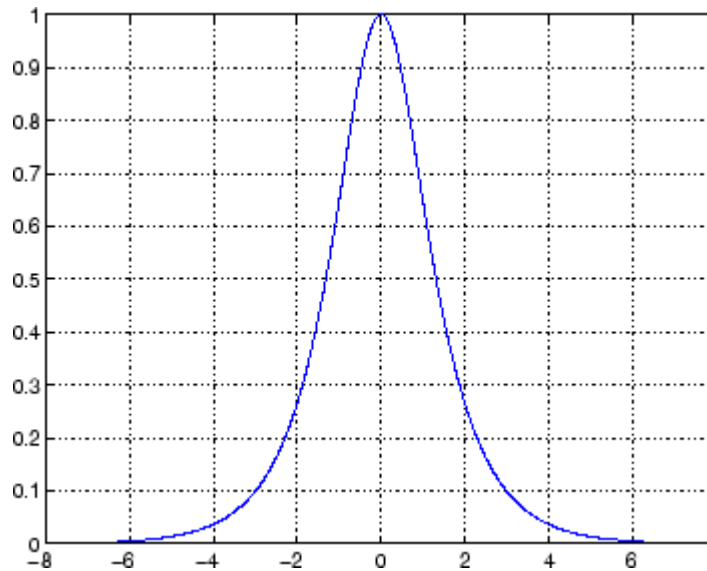
The `sech` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sech(X)` returns an array the same size as `X` containing the hyperbolic secant of the elements of `X`.

Examples

Graph the hyperbolic secant over the domain $-2\pi \le x \le 2\pi$.

```
x = -2*pi:0.01:2*pi;
plot(x,sech(x)), grid on
```



Algorithm

`sech` uses this algorithm.

$$\operatorname{sech}(z) = \frac{1}{\cosh(z)}$$

Definition

The secant can be defined as

$$\operatorname{sech}(z) = \frac{1}{\cosh(z)}$$

Algorithm

`sec` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`asec`, `asech`, `sec`

selectmoveresize

Purpose

Select, move, resize, or copy axes and uicontrol graphics objects

Syntax

```
A = selectmoveresize
set(gca,'ButtonDownFcn','selectmoveresize')
```

Description

`selectmoveresize` is useful as the callback routine for axes and uicontrol button down functions. When executed, it selects the object and allows you to move, resize, and copy it.

`A = selectmoveresize` returns a structure array containing

- `A.Type`: a string containing the action type, which can be `Select`, `Move`, `Resize`, or `Copy`
- `A.Handles`: a list of the selected handles, or, for a `Copy`, an m-by-2 matrix containing the original handles in the first column and the new handles in the second column

`set(gca,'ButtonDownFcn','selectmoveresize')` sets the `ButtonDownFcn` property of the current axes to selectmoveresize:

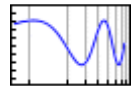
See Also

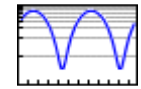
The `ButtonDownFcn` property of `axes` and `uicontrol` objects

"Object Manipulation" on page 1-100 for related functions

Purpose

Semilogarithmic plots





GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
semilogx(Y)
semilogy(...)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)
h = semilogy(...)
hlines = semilogx('v6',...)
```

Description

`semilogx` and `semilogy` plot data as logarithmic scales for the *x*- and *y*-axis, respectively.

`semilogx(Y)` creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of `Y` versus their index if `Y` contains real numbers. `semilogx(Y)` is equivalent to `semilogx(real(Y), imag(Y))` if `Y` contains complex numbers. `semilogx` ignores the imaginary component in all other uses of this function.

`semilogy(...)` creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis.

`semilogx(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `semilogx` plots the vector argument versus the rows or

columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`semilogx(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'PropertyName',PropertyValue,...)` sets property values for all `lineseries` graphics objects created by `semilogx`.

`h = semilogx(...)` and `h = semilogy(...)` return a vector of handles to lineseries graphics objects, one handle per line.

Backward-Compatible Version

`hlines = semilogx('v6',...)` and `hlines = semilogy('v6',...)` return the handles to line objects instead of lineseries objects.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

If you do not specify a color when plotting more than one line, `semilogx` and `semilogy` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,LineSpec` triples; for example,

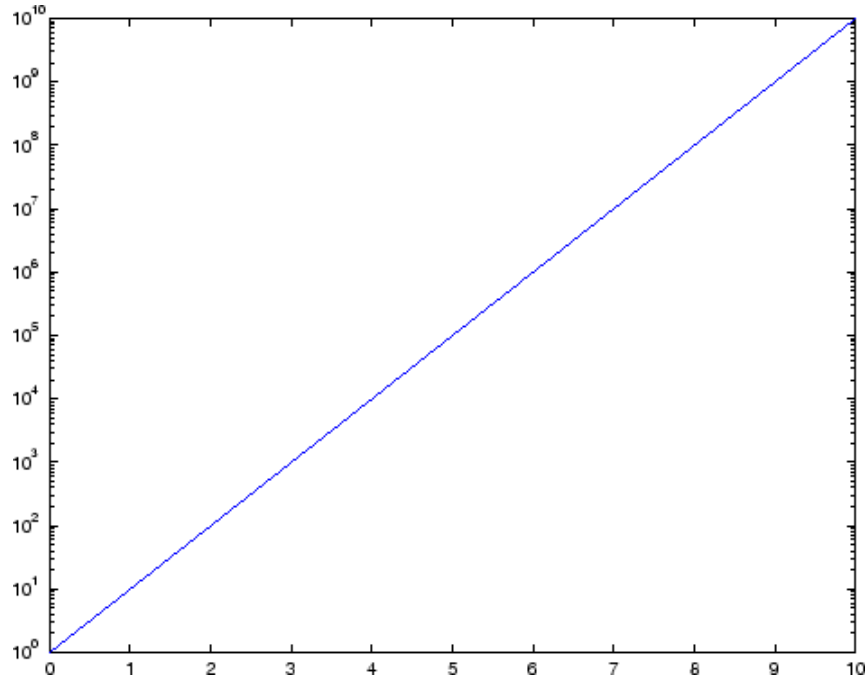
```
semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode will remain as it is and the new data will plot as linear.

Examples

Create a simple `semilogy` plot.

```
x = 0:.1:10;
semilogy(x,10.^x)
```



See Also

`line`, `LineSpec`, `loglog`, `plot`

"Basic Plots and Graphs" on page 1-86 for related functions

sendmail

Purpose

Send e-mail message to address list

Syntax

```
sendmail('recipients','subject')
sendmail('recipients','subject','message','attachments')
```

Description

`sendmail('recipients','subject')` sends e-mail to `recipients` with the specified `subject`. For `recipients`, use a string for a single address, or a cell array of strings for multiple addresses.

`sendmail('recipients','subject','message','attachments')` sends `message` to `recipients` with the specified `subject`. For `recipients`, use a string for a single address, or a cell array of strings for multiple addresses. For `message`, use a string or cell array. When `message` is a string, the text automatically wraps at 75 characters. When `message` is a cell array, it does not wrap but rather each cell is a new line. To force text to start on a new line in strings or cells, use `10`, as shown in the "Example of sendmail with New Lines Specified" on page 2-2883. Specify `attachments` as a cell array of files to send along with `message`.

To use `sendmail`, you must set the preferences for your e-mail server (Internet SMTP server) and your e-mail address must be set. MATLAB tries to read the SMTP mail server from your system registry, but if it cannot, it results in an error. In this event, identify the outgoing mail server for your electronic mail application, which is usually listed in the application's preferences, or, consult your e-mail system administrator. Then provide the information to MATLAB using

```
setpref('Internet','SMTP_Server','myserver.myhost.com');
```

If you cannot easily determine your e-mail server, try using `mail`, as in

```
setpref('Internet','SMTP_Server','mail');
```

which might work because `mail` is often a default for mail systems.

Similarly, if MATLAB cannot determine your e-mail address and produces an error, specify your e-mail address using

```
setpref('Internet','E_mail','myaddress@example.com');
```

Note The `sendmail` function does not support e-mail servers that require authentication.

Examples

Example of sendmail with Two Attachments

```
sendmail('user@otherdomain.com',...
         'Test subject','Test message',...
         {'directory/attach1.html','attach2.doc'});
```

Example of sendmail with New Lines Specified

This mail message forces the message to start new lines after each `10`.

```
sendmail('user@otherdomain.com','New subject', ...
['Line1 of message' 10 'Line2 of message' 10 ...
 'Line3 of message' 10 'Line4 of message']);
```

The resulting message is

```
Line1 of message
Line2 of message
Line3 of message
Line4 of message
```

See Also

`getpref`, `setpref`

Purpose

Create serial port object

Syntax

```
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)
```

Arguments

`'port'`	The serial port name.
`'PropertyName'`	A serial port property name.
`PropertyValue`	A property value supported by *`PropertyName`*.
`obj`	The serial port object.

Description

`obj = serial('port')` creates a serial port object associated with the serial port specified by `port`. If `port` does not exist, or if it is in use, you will not be able to connect the serial port object to the device.

`obj = serial('port','PropertyName',PropertyValue,...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

Remarks

When you create a serial port object, these property values are automatically configured:

- The `Type` property is given by `serial`.
- The `Name` property is given by concatenating `Serial` with the port specified in the `serial` function.
- The `Port` property is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use

of property name completion. For example, the following commands are all valid.

```
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Refer to Configuring Property Values for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

Example

This example creates the serial port object `s1` associated with the serial port COM1.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
get(s1,{'Type','Name','Port'})
ans =
    'serial'    'Serial-COM1'    'COM1'
```

To specify properties during object creation

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

See Also

Functions

`fclose`, `fopen`

Properties

`Name`, `Port`, `Status`, `Type`

serialbreak

Purpose

Send break to device connected to serial port

Syntax

```
serialbreak(obj)
serialbreak(obj,time)
```

Arguments

`obj`	A serial port object.
`time`	The duration of the break, in milliseconds.

Description

`serialbreak(obj)` sends a break of 10 milliseconds to the device connected to `obj`.

`serialbreak(obj,time)` sends a break to the device with a duration, in milliseconds, specified by `time`. Note that the duration of the break might be inaccurate under some operating systems.

Remarks

For some devices, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the device.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

See Also

Functions

`fopen`, `stopasync`

Properties

`Status`

Purpose

Set object properties

Syntax

```
set(H,'PropertyName',PropertyValue,...)
set(H,a)
set(H,pn,pv,...)
set(H,pn,MxN_pv)
a = set(h)
a = set(h,'Default')
a = set(h,'DefaultObjectTypePropertyName')
pv = set(h,'PropertyName')
```

Description

`set(H,'PropertyName',PropertyValue,...)` sets the named properties to the specified values on the object(s) identified by `H`. `H` can be a vector of handles, in which case `set` sets the properties' values for all the objects.

`set(H,a)` sets the named properties to the specified values on the object(s) identified by `H`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties.

`set(H,pn,pv,...)` sets the named properties specified in the cell array `pn` to the corresponding value in the cell array `pv` for all objects identified in `H`.

`set(H,pn,MxN_pv)` sets `n` property values on each of `m` graphics objects, where `m = length(H)` and `n` is equal to the number of property names contained in the cell array `pn`. This allows you to set a given group of properties to different values on each object.

`a = set(h)` returns the user-settable properties and possible values for the object identified by `h`. `a` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

`a = set(h,'Default')` returns the names of properties having default values set on the object identified by `h`. `set` also returns the possible values if they are strings. `h` must be scalar.

`a = set(h,'Default`*`ObjectTypePropertyName`*`')` returns the possible values of the named property for the specified object type, if the values are strings. The argument `Default`*`ObjectTypePropertyName`* is the word `Default` concatenated with the object type (e.g., `axes`) and the property name (e.g., `CameraPosition`). For example, `DefaultAxesCameraPosition`. `h` must be scalar.

`pv = set(h,'`*`PropertyName`*`')` returns the possible values for the named property. If the possible values are strings, `set` returns each in a cell of the cell array `pv`. For other properties, `set` returns an empty cell array. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

Remarks

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`.

Setting Property Units

Note that if you are setting both the `FontSize` and the `FontUnits` properties in one function call, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`. The same applies to figure and axes uints — always set the `Units` property before setting properties whose values you want to be interpreted in those units. For example,

```
f = figure('Units','characters',...
        'Position',[30 30 120 35]);
```

Examples

Set the Color property of the current axes to blue.

```
set(gca,'Color','b')
```

Change all the `lines` in a plot to black.

```
plot(peaks)
set(findobj('Type','line'),'Color','k')
```

You can define a group of properties in a structure to better organize your code. For example, these statements define a structure called `active`, which contains a set of property definitions used for the `uicontrol` objects in a particular figure. When this figure becomes the current figure, MATLAB changes colors and enables the controls.

```
active.BackgroundColor = [.7 .7 .7];
active.Enable = 'on';
active.ForegroundColor = [0 0 0];

if gcf == control_fig_handle
 set(findobj(control_fig_handle,'Type','uicontrol'),active)
end
```

You can use cell arrays to set properties to different values on each object. For example, these statements define a cell array to set three properties,

```
PropName(1) = {'BackgroundColor'};
PropName(2) = {'Enable'};
PropName(3) = {'ForegroundColor'};
```

These statements define a cell array containing three values for each of three objects (i.e., a 3-by-3 cell array).

```
PropVal(1,1) = {[.5 .5 .5]};
PropVal(1,2) = {'off'};
PropVal(1,3) = {[.9 .9 .9]};
PropVal(2,1) = {[1 0 0]};
PropVal(2,2) = {'on'};
PropVal(2,3) = {[1 1 1]};
PropVal(3,1) = {[.7 .7 .7]};
PropVal(3,2) = {'on'};
PropVal(3,3) = {[0 0 0]};
```

Now pass the arguments to `set`,

```
set(H,PropName,PropVal)
```

where `length(H) = 3` and each element is the handle to a uicontrol.

Setting Different Values for the Same Property on Multiple Objects

Suppose you want to set the value of the `Tag` property on five line objects, each to a different value. Note how the value cell array needs to be transposed to have the proper shape.

```
h = plot(rand(5));
set(h,{'Tag'},{'line1','line2','line3','line4','line5'}')
```

See Also

`findobj`, `gca`, `gcf`, `gco`, `gcbo`, `get`

"Finding and Identifying Graphics Objects" on page 1-93 for related functions

Purpose

Set object or interface property to specified value

Syntax

```
h.set('pname', value)
h.set('pname1', value1, 'pname2', value2, ...)
set(h, ...)
```

Description

`h.set('pname', value)` sets the property specified in the string `pname` to the given `value`.

`h.set('pname1', value1, 'pname2', value2, ...)` sets each property specified in the `pname` strings to the given `value`.

`set(h, ...)` is an alternate syntax for the same operation.

See "Handling COM Data in MATLAB" in the External Interfaces documentation for information on how MATLAB converts workspace matrices to COM data types.

Examples

Create an `mwsamp` control and use `set` to change the `Label` and `Radius` properties:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);

h.set('Label', 'Click to fire event', 'Radius', 40);
h.invoke('Redraw');
```

Here is another way to do the same thing, only without `set` and `invoke`:

```
h.Label = 'Click to fire event';
h.Radius = 40;
h.Redraw;
```

See Also

`get`, `inspect`, `isprop`, `addproperty`, `deleteproperty`

Purpose

Configure or display serial port object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *`PropertyName`*.
`PN`	A cell array of property names.
`PV`	A cell array of property values.
`S`	A structure with property names and property values.
`props`	A structure array whose field names are the property names for `obj`, or cell array of possible values.

Description

`set(obj)` displays all configurable properties values for `obj`. If a property has a finite list of possible string values, then these values are also displayed.

`props = set(obj)` returns all configurable properties and their possible values for `obj` to `props`. `props` is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

`set(obj,'PropertyName')` displays the valid values for *PropertyName* if it possesses a finite list of string values.

`props = set(obj,'PropertyName')` returns the valid values for *PropertyName* to `props`. `props` is a cell array of possible string values or an empty cell array if *PropertyName* does not have a finite list of possible values.

`set(obj,'PropertyName',PropertyValue,...)` configures multiple property values with a single command.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be m-by-n where m is equal to the number of serial port objects in `obj` and n is equal to the length of `PN`.

`set(obj,S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are serial port object properties, and whose field values are the values of the corresponding properties.

Remarks

Refer to Configuring Property Values for a list of serial port object properties that you can configure with `set`.

You can use any combination of property name/property value pairs, structures, and cell arrays in one call to `set`. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then the following commands are all valid.

```
set(s,'BaudRate')
set(s,'baudrate')
set(s,'BAUD')
```

If you use the `help` command to display help for `set`, then you need to supply the pathname shown below.

```
help serial/set
```

Examples

This example illustrates some of the ways you can use `set` to configure or return property values for the serial port object `s`.

```
s = serial('COM1');
set(s,'BaudRate',9600,'Parity','even')
set(s,{'StopBits','RecordName'},{2,'sydney.txt'})
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

See Also

Functions

`get`

Purpose

Configure or display timer object properties

Syntax

```
set(obj)
prop_struct = set(obj)
set(obj,'PropertyName')
prop_cell=set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,S)
set(obj,PN,PV)
```

Description

`set(obj)` displays property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object.

`prop_struct = set(obj)` returns the property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object. The return value, `prop_struct`, is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values or empty cell arrays if the property does not have a finite set of possible string values.

`set(obj,'PropertyName')` displays the possible values for the specified property, *`PropertyName`*, of timer object `obj`. `obj` must be a single timer object.

`prop_cell=set(obj,'PropertyName')` returns the possible values for the specified property, *`PropertyName`*, of timer object `obj`. `obj` must be a single timer object. The returned array, `prop_cell`, is a cell array of possible value strings or an empty cell array if the property does not have a finite set of possible string values.

`set(obj,'PropertyName',PropertyValue,...)` configures the property, *`PropertyName`*, to the specified value, `PropertyValue`, for timer object `obj`. You can specify multiple property name/property value pairs in a single statement. `obj` can be a single timer object or a vector of timer objects, in which case `set` configures the property values for all the timer objects specified.

`set(obj,S)` configures the properties of `obj`, with the values specified in `S`, where `S` is a structure whose field names are object property names.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings, `PN`, to the corresponding values in the cell array `PV`, for the timer object `obj`. `PN` must be a vector. If `obj` is an array of timer objects, `PV` can be an M-by-N cell array, where M is equal to the length of timer object array and N is equal to the length of `PN`. In this case, each timer object is updated with a different set of values for the list of property names contained in `PN`.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `set`.

Examples

Create a timer object.

```
t = timer;
```

Display all configurable properties and their possible values.

```
set(t)
    BusyMode: [ {drop} | queue | error ]
    ErrorFcn: string -or- function handle -or- cell array
    ExecutionMode: [ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
    Name
    ObjectVisibility: [ {on} | off ]
    Period
    StartDelay
    StartFcn: string -or- function handle -or- cell array
    StopFcn: string -or- function handle -or- cell array
    Tag
    TasksToExecute
    TimerFcn: string -or- function handle -or- cell array
    UserData
```

View the possible values of the `ExecutionMode` property.

```
set(t, 'ExecutionMode')
[ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
```

Set the value of a specific timer object property.

```
set(t, 'ExecutionMode', 'FixedRate')
```

Set the values of several properties of the timer object.

```
set(t, 'TimerFcn', 'callbk', 'Period', 10)
```

Use a cell array to specify the names of the properties you want to set and another cell array to specify the values of these properties.

```
set(t, {'StartDelay', 'Period'}, {30, 30})
```

See Also

`timer`, `get(timer)`

set (timeseries)

Purpose

Set properties of `timeseries` object

Syntax

```
set(ts,'Property',Value)
set(ts,'Property1',Value1,'Property2',Value2,...)
set(ts,'Property')
set(ts)
```

Description

`set(ts,'Property',Value)` sets the property `'Property'` of the `timeseries` object `ts` to the value `Value`. The following syntax is equivalent:

```
ts.Property = Value
```

`set(ts,'Property1',Value1,'Property2',Value2,...)` sets multiple property values for `ts` with a single statement.

`set(ts,'Property')` displays values for the specified property of the `timeseries` object ts.

`set(ts)` displays all properties and values of the `timeseries` object `ts`.

See Also

`get (timeseries)`

Purpose

Set properties of `tscollection` object

Syntax

```
set(tsc,'Property',Value)
set(tsc,'Property1',Value1,'Property2',Value2,...)
set(tsc,'Property')
```

Description

`set(tsc,'Property',Value)` sets the property `'Property'` of the `tscollection` `tsc` to the value `Value`. The following syntax is equivalent:

```
tsc.Property = Value
```

`set(tsc,'Property1',Value1,'Property2',Value2,...)` sets multiple property values for `tsc` with a single statement.

`set(tsc,'Property')` displays values for the specified property in the time-series collection `tsc`.

`set(tsc)` displays all properties and values of the `tscollection` object `tsc`.

See Also

`get (tscollection)`

setabstime (timeseries)

Purpose

Set times of `timeseries` object as date strings

Syntax

```
ts = setabstime(ts,Times)
ts = setabstime(ts,Times,Format)
```

Description

`ts = setabstime(ts,Times)` sets the times in `ts` to the date strings specified in `Times`. Times must either be a cell array of strings, or a `char` array containing valid date or time values in the same date format.

`ts = setabstime(ts,Times,Format)` explicitly specifies the date-string format used in Times.

Examples

1 Create a time-series object.

```
ts = timeseries(rand(3,1))
```

2 Set the absolute time vector.

```
ts = setabstime(ts,{'12-DEC-2005 12:34:56',...
     '12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})
```

See Also

`datestr`, `getabstime (timeseries)`, `timeseries`

Purpose

Set times of tscollection object as date strings

Syntax

```
tsc = setabstime(tsc,Times)
tsc = setabstime(tsc,Times,format)
```

Description

`tsc = setabstime(tsc,Times)` sets the times in tsc using the date strings Times. Times must be either a cell array of strings, or a `char` array containing valid date or time values in the same date format.

`tsc = setabstime(tsc,Times,format)` specifies the date-string format used in Times explicitly.

Examples

1 Create a tscollection object.

```
tsc = tscollection(timeseries(rand(3,1)))
```

2 Set the absolute time vector.

```
tsc = setabstime(tsc,{'12-DEC-2005 12:34:56',...
      '12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})
```

See Also

datestr, getabstime (tscollection), tscollection

setappdata

Purpose

Specify application-defined data

Syntax

```
setappdata(h,'name',value)
```

Description

`setappdata(h,'name',value)` sets application-defined data for the object with handle `h`. The application-defined data, which is created if it does not already exist, is assigned the specified `name` and `value`. The `value` can be any type of data.

See Also

`getappdata`, `isappdata`, `rmappdata`

Purpose

Find set difference of two vectors

Syntax

```
c = setdiff(A, B)
c = setdiff(A, B, 'rows')
[c,i] = setdiff(...)
```

Description

`c = setdiff(A, B)` returns the values in `A` that are not in `B`. In set theory terms, `c = A - B`. Inputs `A` and `B` can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

`c = setdiff(A, B, 'rows')`, when `A` and `B` are matrices with the same number of columns, returns the rows from `A` that are not in `B`.

`[c,i] = setdiff(...)` also returns an index vector `index` such that `c = a(i)` or `c = a(i,:)`.

Remarks

Because NaN is considered to be not equal to itself, it is always in the result `c` if it is in `A`.

Examples

```
A = magic(5);
B = magic(4);
[c, i] = setdiff(A(:), B(:));
c' =    17    18    19    20    21    22    23    24    25
i' =     1    10    14    18    19    23     2     6    15
```

See Also

`intersect`, `ismember`, `issorted`, `setxor`, `union`, `unique`

setenv

Purpose

Set environment variable

Syntax

```
setenv(name, value)
setenv(name)
```

Description

`setenv(name, value)` sets the value of an environment variable belonging to the underlying operating system. Inputs `name` and `value` are both strings. If `name` already exists as an environment variable, then `setenv` replaces its current value with the string given in `value`. If `name` does not exist, `setenv` creates a new environment variable called `name` and assigns `value` to it.

`setenv(name)` is equivalent to `setenv(name, '')` and assigns a null value to the variable `name`. Under the Windows operating system, this is equivalent to undefining the variable. On most UNIX-like platforms, it is possible to have an environment variable defined as empty.

The maximum number of characters in `name` is 2^{15} - 2 (or 32766). If `name` contains the character `=`, `setenv` throws an error. The behavior of environment variables with `=` in the name is not well-defined.

On all platforms, `setenv` passes the `name` and `value` strings to the operating system unchanged. Special characters such as `;`, `/`, `:`, `$`, `%`, etc. are left unexpanded and intact in the variable value.

Values assigned to variables using `setenv` are picked up by any process that is spawned using the MATLAB `system`, `unix`, `dos` or `!` functions. You can retrieve any value set with `setenv` by using `getenv(name)`.

Examples

% Set and retrieve a new value for the environment variable `TEMP`:

```
setenv('TEMP', 'C:\TEMP');
getenv('TEMP')
```

% Append the `Perl\bin` directory to your system `PATH` variable:

```
setenv('PATH', [getenv('PATH') ';D:\Perl\bin']);
```

See Also

`getenv`, `system`, `unix`, `dos`, `!`

Purpose

Set value of structure array field

Syntax

```
s = setfield(s, 'field', v)
s = setfield(s, {i,j}, 'field', {k}, v)
```

Description

`s = setfield(s, 'field', v)`, where `s` is a 1-by-1 structure, sets the contents of the specified field to the value `v`. If `field` is not an existing field in structure `s`, MATLAB creates that field and assigns the value `v` to it. This is equivalent to the syntax `s.field = v`.

`s = setfield(s, {i,j}, 'field', {k}, v)` sets the contents of the specified field to the value `v`. If `field` is not an existing field in structure `s`, MATLAB creates that field and assigns the value `v` to it. This is equivalent to the syntax `s(i,j).field(k) = v`. All subscripts must be passed as cell arrays — that is, they must be enclosed in curly braces (similar to `{i,j}` and `{k}` above). Pass field references as strings.

See "Naming conventions for Structure Field Names" for guidelines to creating valid field names.

Remarks

In many cases, you can use dynamic field names in place of the `getfield` and `setfield` functions. Dynamic field names express structure fields as variable expressions that MATLAB evaluates at run-time. See Solution 1-19QWG for information about using dynamic field names versus the `getfield` and `setfield` functions.

Examples

Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1;
```

You can change the `name` field of `mystr(2,1)` using

```
mystr = setfield(mystr, {2,1}, 'name', 'ted');
mystr(2,1).name
```

```
ans =

ted
```

The following example sets fields of a structure using `setfield` with variable and quoted field names and additional subscripting arguments.

```
class = 5;   student = 'John_Doe';
grades_Doe = [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
grades = [];

grades = setfield(grades, {class}, student, 'Math', ...
   {10, 21:30}, grades_Doe);
```

You can check the outcome using the standard structure syntax.

```
grades(class).John_Doe.Math(10, 21:30)

ans =

    85   89   76   93   85   91   68   84   95   73
```

See Also

`getfield`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, "Using Dynamic Field Names"

Purpose

Set default interpolation method for `timeseries` object

Syntax

```
ts = setinterpmethod(ts,Method)
ts = setinterpmethod(ts,FHandle)
ts = setinterpmethod(ts,InterpObj),
```

Description

`ts = setinterpmethod(ts,Method)` sets the default interpolation method for `timeseries` object `ts`, where `Method` is a string. `Method` in `ts`. `Method` is either `'linear'` or `'zoh'` (zero-order hold). For example:

```
ts = timeseries(rand(100,1),1:100);
ts = setinterpmethod(ts,'zoh');
```

`ts = setinterpmethod(ts,FHandle)` sets the default interpolation method for `timeseries` object `ts`, where FHandle is a function handle to the interpolation method defined by the function handle FHandle. For example:

```
ts = timeseries(rand(100,1),1:100);
myFuncHandle = @(new_Time,Time,Data)...
               interp1(Time,Data,new_Time,...
                       'linear','extrap');
ts = setinterpmethod(ts,myFuncHandle);
ts = resample(ts,[-5:0.1:10]);
plot(ts);
```

Note For FHandle, you must use three input arguments. The order of input arguments must be `new_Time`, `Time`, and `Data`. The single output argument must be the interpolated data only.

`ts = setinterpmethod(ts,InterpObj),` where InterpObj is a `tsdata.interpolation` object that directly replaces the interpolation object stored in `ts`. For example:

```
ts = timeseries(rand(100,1),1:100);
```

```
myFuncHandle = @(new_Time,Time,Data)...
               interp1(Time,Data,new_Time,...
                       'linear','extrap');
myInterpObj = tsdata.interpolation(myFuncHandle);
ts = setinterpmethod(ts,myInterpObj);
```

This method is case sensitive.

See Also

`getinterpmethod`, `timeseries`, `tsprops`

Purpose

Set component position in pixels

Syntax

```
setpixelposition(handle,position)
setpixelposition(handle,position,recursive)
```

Description

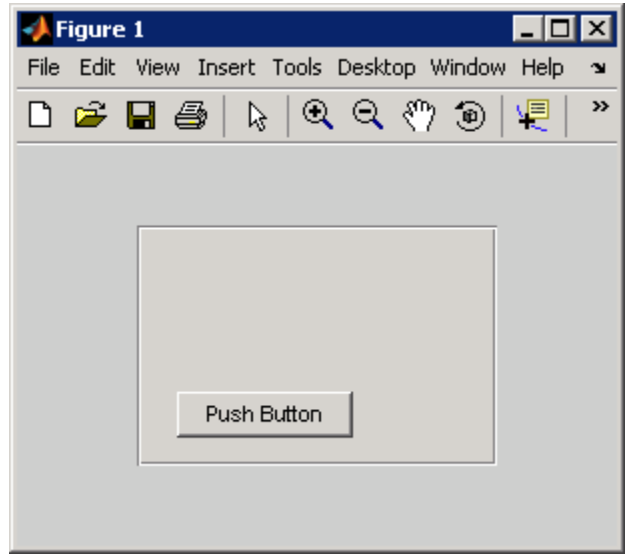
`setpixelposition(handle,position)` sets the position of the component specified by `handle`, to the specified pixel position relative to its parent. `position` is a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

`setpixelposition(handle,position,recursive)` sets the position as above. If `recursive` is true, the position is set relative to the parent figure of `handle`.

Example

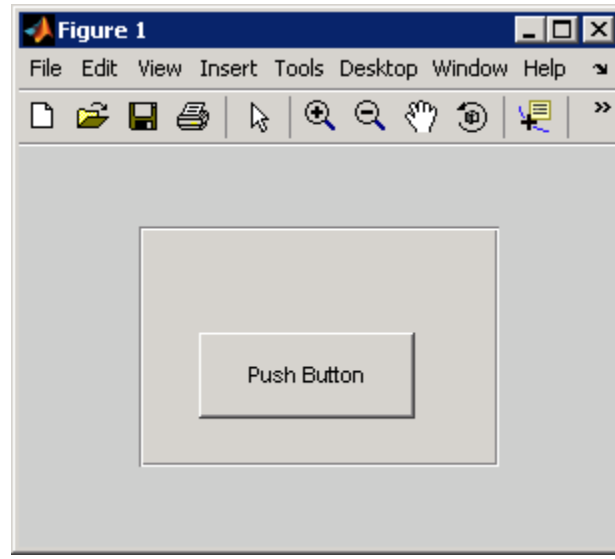
This example first creates a push button within a panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6];
h1 = uicontrol(p,'Style','PushButton','Units','Nomalized',...
                'String','Push Button','Position',[.1 .1 .5 .2]);
```



The example then retrieves the position of the push button and changes its position with respect to the panel.

```
pos1 = getpixelposition(h1);
setpixelposition(h1,pos1 + [10 10 25 25]);
```



See Also

`getpixelposition`, `uicontrol`, `uipanel`

setpref

Purpose

Set preference

Syntax

```
setpref('group','pref',val)
setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,
   valn})
```

Description

`setpref('group','pref',val)` sets the preference specified by `group` and `pref` to the value `val`. Setting a preference that does not yet exist causes it to be created.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g., `'MathWorks_GUIDE_ApplicationPrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,valn})` sets each preference specified in the cell array of names to the corresponding value.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

```
addpref('mytoolbox','version','0.0')
setpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')

ans =
      1.0
```

See Also

`addpref`, `getpref`, `ispref`, `rmpref`, `uigetpref`, `uisetpref`

Purpose

Set string flag

Description

This MATLAB 4 function has been renamed `char` in MATLAB 5.

settimeseriesnames

Purpose

Change name of `timeseries` object in `tscollection`

Syntax

```
tsc = settimeseriesnames(tsc,old,new)
```

Description

`tsc = settimeseriesnames(tsc,old,new)` replaces the `old` name of `timeseries` object with the new name in tsc.

See Also

`tscollection`

Purpose

Find set exclusive OR of two vectors

Syntax

```
c = setxor(A, B)
c = setxor(A, B, 'rows')
[c, ia, ib] = setxor(...)
```

Description

`c = setxor(A, B)` returns the values that are not in the intersection of `A` and `B`. Inputs `A` and `B` can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted.

`c = setxor(A, B, 'rows')`, when `A` and `B` are matrices with the same number of columns, returns the rows that are not in the intersection of `A` and `B`.

`[c, ia, ib] = setxor(...)` also returns index vectors `ia` and `ib` such that `c` is a sorted combination of the elements `c = a(ia)` and `c = b(ib)` or, for row combinations, `c = a(ia,:)` and `c = b(ib,:)`.

Examples

```
a = [-1 0 1 Inf -Inf NaN];
b = [-2 pi 0 Inf];
c = setxor(a, b)

c =
    -Inf  -2.0000  -1.0000   1.0000   3.1416   NaN
```

See Also

`intersect`, `ismember`, `issorted`, `setdiff`, `union`, `unique`

shading

Purpose

Set color shading properties

Syntax

```
shading flat
shading faceted
shading interp
shading(axes_handle,...)
```

Description

The `shading` function controls the color shading of surface and patch graphics objects.

`shading flat` each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.

`shading faceted` flat shading with superimposed black mesh lines. This is the default shading mode.

`shading interp` varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.

`shading(axes_handle,...)` applies the shading type to the objects in the axes specified by `axes_handle`, instead of the current axes.

Examples

Compare a flat, faceted, and interpolated-shaded sphere.

```
subplot(3,1,1)
sphere(16)
axis square
shading flat
title('Flat Shading')

subplot(3,1,2)
sphere(16)
axis square
shading faceted
title('Faceted Shading')

subplot(3,1,3)
```

```
sphere(16)
axis square
shading interp
title('Interpolated Shading')
```

Algorithm

`shading` sets the `EdgeColor` and `FaceColor` properties of all surface and patch graphics objects in the current axes. `shading` sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

`fill`, `fill3`, `hidden`, `mesh`, `patch`, `pcolor`, `surf`

The `EdgeColor` and `FaceColor` properties for patch and surface graphics objects.

"Color Operations" on page 1-98 for related functions

shiftdim

Purpose

Shift dimensions

Syntax

```
B = shiftdim(X,n)
[B,nshifts] = shiftdim(X)
```

Description

`B = shiftdim(X,n)` shifts the dimensions of `X` by `n`. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons.

`[B,nshifts] = shiftdim(X)` returns the array `B` with the same number of elements as `X` but with any leading singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. `nshifts` is the number of dimensions that are removed.

If `X` is a scalar, `shiftdim` has no effect.

Examples

The `shiftdim` command is handy for creating functions that, like `sum` or `diff`, work along the first nonsingleton dimension.

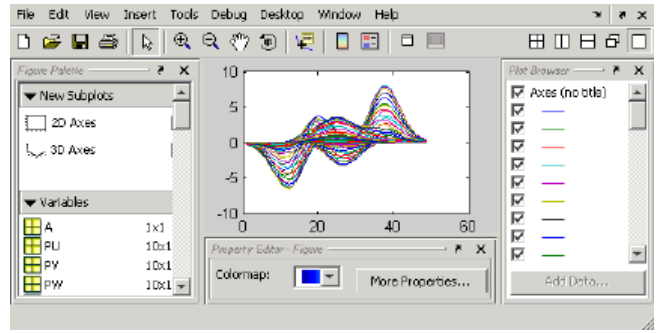
```
a = rand(1,1,3,1,2);
[b,n]  = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
c = shiftdim(b,-n);   % c == a.
d = shiftdim(a,3);    % d is 1-by-2-by-1-by-1-by-3.
```

See Also

`circshift`, `reshape`, `squeeze`

Purpose

Show or hide figure plot tool



GUI Alternatives

Click the larger Plotting Tools icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Individually select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation.

Syntax

```
showplottool('tool')
showplottool('on','tool')
showplottool('off','tool')
showplottool('toggle','tool')
showplottool(figure_handle,...)
```

Description

`showplottool('tool')` shows the specified plot tool on the current figure. *`tool`* can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

`showplottool('on','tool')` shows the specified plot tool on the current figure.

`showplottool('off','tool')` hides the specified plot tool on the current figure.

`showplottool('toggle','tool')` toggles the visibility of the specified plot tool on the current figure.

`showplottool(figure_handle,...)` operates on the specified figure instead of the current figure.

Note When you dock, undock, resize, or reposition a plotting tool and then close it, it will still be configured as you left it the next time you open it. There is no command to reset plotting tools to their original, default locations.

See Also

`figurepalette`, `plotbrowser`, `plottools`, `propertyeditor`

Purpose

Reduce size of patch faces

Syntax

```
shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p)
nfv = shrinkfaces(f,v,sf)
[nf,nv] = shrinkfaces(...)
```

Description

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, MATLAB creates nonshared vertices before performing the face-area reduction.

`nfv = shrinkfaces(p,sf)` returns the face and vertex data in the struct `nfv`, but does not set the `Faces` and `Vertices` properties of patch `p`.

`nfv = shrinkfaces(fv,sf)` uses the face and vertex data from the struct `fv`.

`shrinkfaces(p)` and `shrinkfaces(fv)` (without specifying a shrink factor) assume a shrink factor of 0.3.

`nfv = shrinkfaces(f,v,sf)` uses the face and vertex data from the arrays `f` and `v`.

`[nf,nv] = shrinkfaces(...)` returns the face and vertex data in two separate arrays instead of a struct.

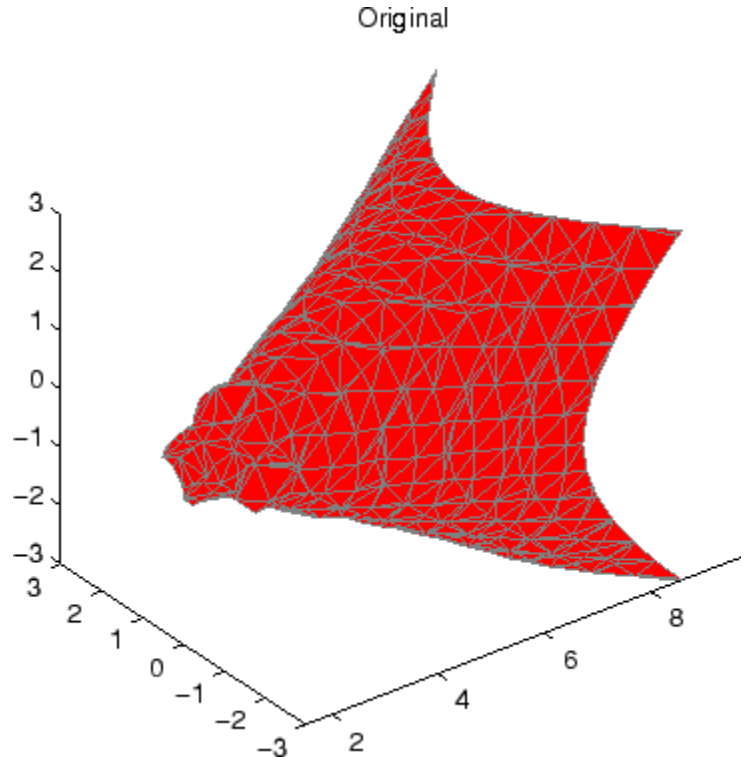
Examples

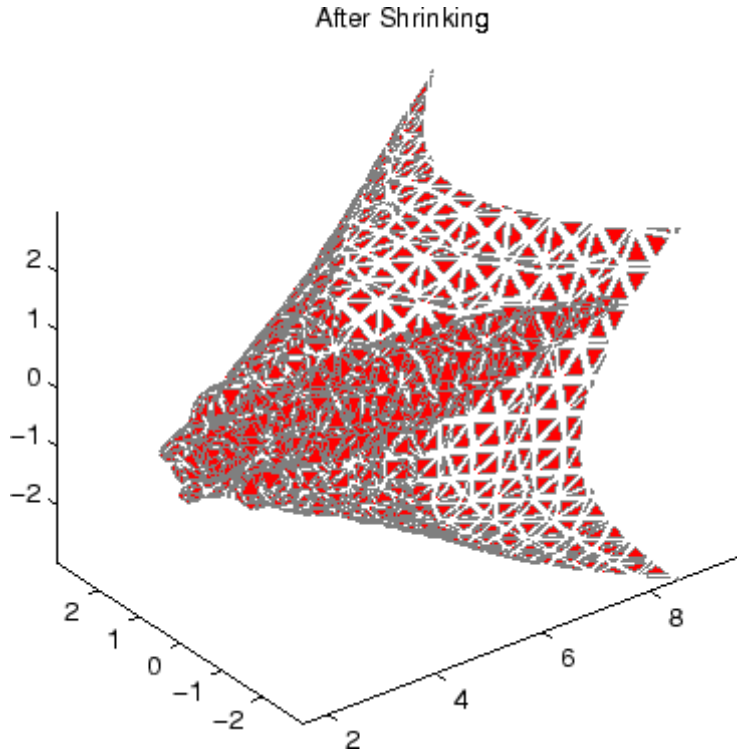
This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). Two isosurfaces provide a before and after view of the effects of shrinking the face size.

- First `reducevolume` samples the flow data at every other point and then `isosurface` generates the faces and vertices data.

- The `patch` command accepts the face/vertex struct and draws the first (`p1`) isosurface.
- Use the `daspect`, `view`, and `axis` commands to set up the view and then add a `title`.
- The `shrinkfaces` command modifies the face/vertex data and passes it directly to `patch`.

```
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
set(p1,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('Original')

figure
p2 = patch(shrinkfaces(fv,.3));
set(p2,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('After Shrinking')
```


Original
3
2
1
0
−1
−2
−3
3
2
1
0
−1
−2
−3
2
4
6
8



See Also

`isosurface`, `patch`, `reducevolume`, `daspect`, `view`, `axis`

"Volume Visualization" on page 1-102 for related functions

Purpose

Signum function

Syntax

```
Y = sign(X)
```

Description

`Y = sign(X)` returns an array `Y` the same size as `X`, where each element of `Y` is:

- `1` if the corresponding element of `X` is greater than zero
- `0` if the corresponding element of `X` equals zero
- `-1` if the corresponding element of `X` is less than zero

For nonzero complex `X`, `sign(X) = X./abs(X)`.

See Also

`abs`, `conj`, `imag`, `real`

Purpose

Sine of argument in radians

Syntax

```
Y = sin(X)
```

Description

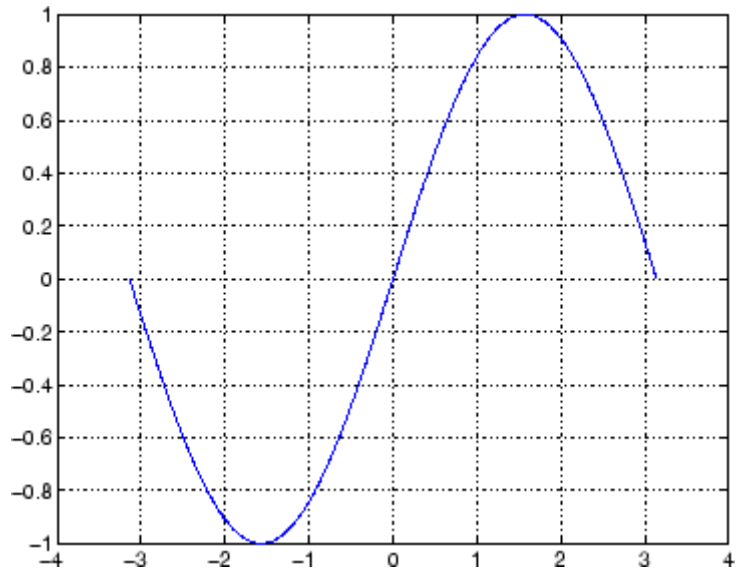
The `sin` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sin(X)` returns the circular sine of the elements of `X`.

Examples

Graph the sine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,sin(x)), grid on
```



The expression `sin(pi)` is not exactly zero, but rather a value the size of the floating-point accuracy `eps`, because `pi` is only a floating-point approximation to the exact value of π.

Definition

The sine can be defined as

$$\sin(x + iy) = \sin(x)\cosh(y) + i\cos(x)\sinh(y)$$

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}$$

Algorithm

`sin` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`sind`, `sinh`, `asin`, `asind`, `asinh`

sind

Purpose

Sine of argument in degrees

Syntax

```
Y = sind(X)
```

Description

`Y = sind(X)` is the sine of the elements of `X`, expressed in degrees. For integers `n`, `sind(n*180)` is exactly zero, whereas `sin(n*pi)` reflects the accuracy of the floating point value of `pi`.

See Also

`sin`, `sinh`, `asin`, `asind`, `asinh`

Purpose

Convert to single precision

Syntax

```
B = single(A)
```

Description

`B = single(A)` converts the matrix `A` to single precision, returning that value in `B`. `A` can be any numeric object (such as a `double`). If `A` is already single precision, `single` has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The `single` class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are `reshape`, `size`, the relational operators, subscripted assignment, and subscripted reference.

You can define your own methods for the `single` class by placing the appropriately named method in an `@single` directory within a directory on your path.

Examples

```
a = magic(4);
b = single(a);

whos
  Name      Size          Bytes  Class

  a         4x4             128  double array
  b         4x4              64  single array
```

See Also

`double`

Purpose

Hyperbolic sine of argument in radians

Syntax

```
Y = sinh(X)
```

Description

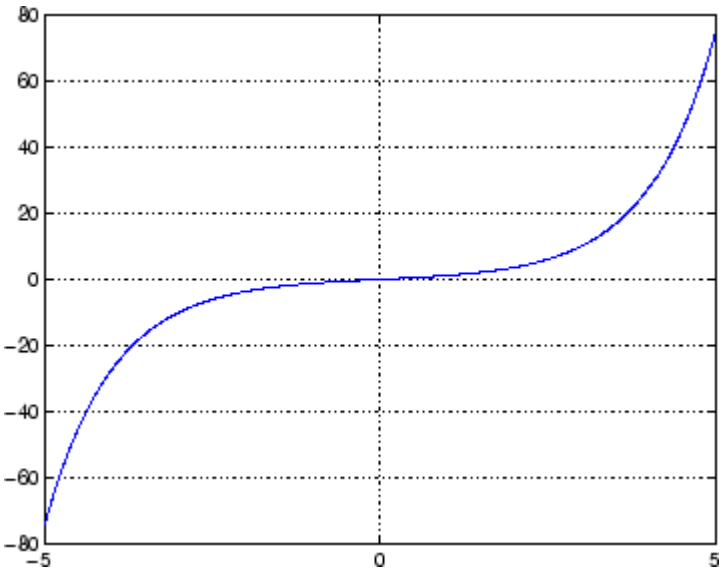
The `sinh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sinh(X)` returns the hyperbolic sine of the elements of `X`.

Examples

Graph the hyperbolic sine function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,sinh(x)), grid on
```



Definition

The hyperbolic sine can be defined as

$$\sinh(z) = \frac{e^z - e^{-z}}{2}$$

Algorithm

`sinh` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`sin`, `sind`, `asin`, `asinh`, `asind`

Purpose

Array dimensions

Syntax

```
d = size(X)
[m,n] = size(X)
m = size(X,dim)
[d1,d2,d3,...,dn] = size(X),
```

Description

`d = size(X)` returns the sizes of each dimension of array `X` in a vector `d` with `ndims(X)` elements. If `X` is a scalar, which MATLAB regards as a 1-by-1 array, `size(X)` returns the vector `[1 1]`.

`[m,n] = size(X)` returns the size of matrix `X` in separate variables `m` and `n`.

`m = size(X,dim)` returns the size of the dimension of `X` specified by scalar `dim`.

`[d1,d2,d3,...,dn] = size(X)`, for `n` > 1, returns the sizes of the dimensions of the array `X` in the variables d1,d2,d3,...,dn, provided the number of output arguments `n` equals `ndims(X)`. If `n` does not equal `ndims(X)`, the following exceptions hold:

`n < ndims(X)`	`di` equals the size of the `i`th dimension of `X` for $1 \le i < n$, but `dn` equals the product of the sizes of the remaining dimensions of `X`, that is, dimensions `n` through `ndims(X)`.
`n > ndims(X)`	`size` returns ones in the "extra" variables, that is, those corresponding to `ndims(X)+1` through `n`.

Note For a Java array, `size` returns the length of the Java array as the number of rows. The number of columns is always 1. For a Java array of arrays, the result describes only the top level array.

Examples

Example 1

The size of the second dimension of `rand(2,3,4)` is 3.

```
m = size(rand(2,3,4),2)

m =
     3
```

Here the size is output as a single vector.

```
d = size(rand(2,3,4))

d =
     2     3     4
```

Here the size of each dimension is assigned to a separate variable.

```
[m,n,p] = size(rand(2,3,4))
m =
     2

n =
     3

p =
     4
```

Example 2

If `X = ones(3,4,5)`, then

```
[d1,d2,d3] = size(X)

d1 =       d2 =       d3 =
     3          4          5
```

But when the number of output variables is less than `ndims(X)`:

```
[d1,d2] = size(X)

d1 =       d2 =
     3          20
```

The “extra” dimensions are collapsed into a single product.

If `n > ndims(X)`, the “extra” variables all represent singleton dimensions:

```
[d1,d2,d3,d4,d5,d6] = size(X)

d1 =        d2 =        d3 =
     3           4           5

d4 =        d5 =        d6 =
     1           1           1
```

See Also

`exist`, `length`, `numel`, `whos`

Purpose

Size of serial port object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`dim`	The dimension of `obj`.
`d`	The number of rows and columns in `obj`.
`m`	The number of rows in `obj`, or the length of the dimension specified by `dim`.
`n`	The number of columns in `obj`.
`m1,m2,...,mn`	The length of the first N dimensions of `obj`.

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in `obj`.

`[m,n] = size(obj)` returns the number of rows and columns in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also

Functions

`length`

size (timeseries)

Purpose

Size of `timeseries` object

Syntax

```
size(ts)
```

Description

`size(ts)` returns `[n 1]`, where `n` is the length of the time vector for `timeseries` object `ts`.

Remarks

If you want the size of the whole data set, use the following syntax:

```
size(ts.data)
```

If you want the size of each data sample, use the following syntax:

```
getdatasamplesize(ts)
```

See Also

`getdatasamplesize`, `isempty (timeseries)`, `length (timeseries)`

Purpose

Size of `tscollection` object

Syntax

```
size(tsc)
```

Description

`size(tsc)` returns `[n m]`, where `n` is the length of the time vector and `m` is the number of `tscollection` members.

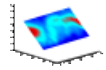
See Also

`length (tscollection)`, `isempty (tscollection)`, `tscollection`

slice

Purpose

Volumetric slice plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
slice(V,sx,sy,sz)
slice(X,Y,Z,V,sx,sy,sz)
slice(V,XI,YI,ZI)
slice(X,Y,Z,V,XI,YI,ZI)
slice(...,'method')
slice(axes_handle,...)
h = slice(...)
```

Description

`slice` displays orthogonal slice planes through volumetric data.

`slice(V,sx,sy,sz)` draws slices along the *x*, *y*, *z* directions in the volume `V` at the points in the vectors `sx`, `sy`, and `sz`. `V` is an *m*-by-*n*-by-*p* volume array containing data values at the default location `X = 1:n`, `Y = 1:m`, `Z = 1:p`. Each element in the vectors `sx`, `sy`, and `sz` defines a slice plane in the *x*-, *y*-, or *z*-axis direction.

`slice(X,Y,Z,V,sx,sy,sz)` draws slices of the volume `V`. `X`, `Y`, and `Z` are three-dimensional arrays specifying the coordinates for `V`. `X`, `Y`, and `Z` must be monotonic and orthogonally spaced (as if produced by the function `meshgrid`). The color at each point is determined by 3-D interpolation into the volume `V`.

`slice(V,XI,YI,ZI)` draws data in the volume `V` for the slices defined by `XI`, `YI`, and `ZI`. `XI`, `YI`, and `ZI` are matrices that define a surface, and the volume is evaluated at the surface points. `XI`, `YI`, and `ZI` must all be the same size.

`slice(X,Y,Z,V,XI,YI,ZI)` draws slices through the volume `V` along the surface defined by the arrays `XI`, `YI`, `ZI`.

`slice(...,'method')` specifies the interpolation method. `'method'` is `'linear'`, `'cubic'`, or `'nearest'`.

- `linear` specifies trilinear interpolation (the default).
- `cubic` specifies tricubic interpolation.
- `nearest` specifies nearest-neighbor interpolation.

`slice(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes object (`gca`). The axes `clim` property is set to span the finite values of `V`.

`h = slice(...)` returns a vector of handles to surface graphics objects.

Remarks

The color drawn at each point is determined by interpolation into the volume `V`.

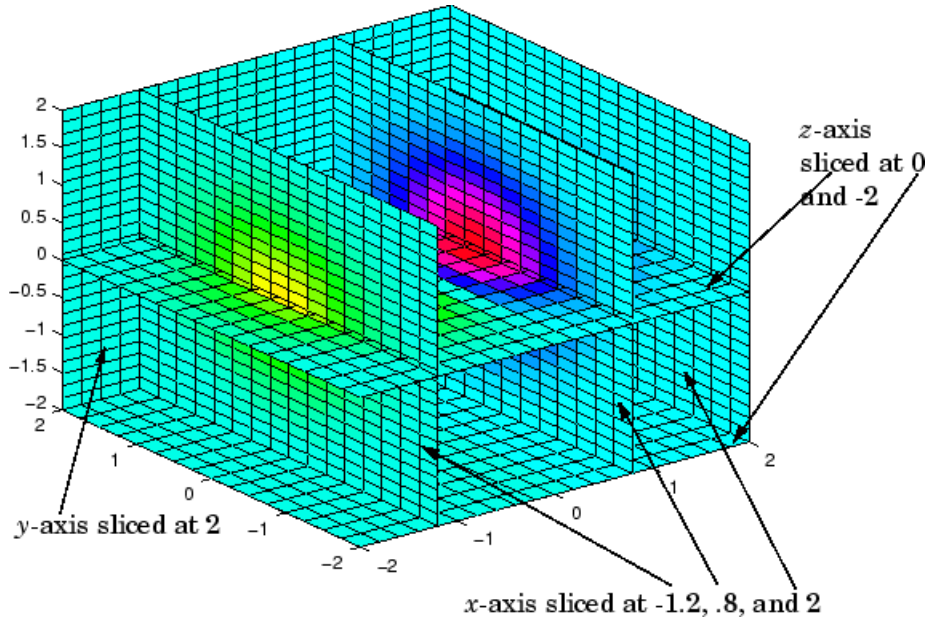
Examples

Visualize the function

$$v = xe^{(-x^2 - y^2 - z^2)}$$

over the range –2 ≤ x ≤ 2, –2 ≤y ≤2, – 2 ≤ z ≤2:

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
xslice = [-1.2,.8,2]; yslice = 2; zslice = [-2,0];
slice(x,y,z,v,xslice,yslice,zslice)
colormap hsv
```



Slicing At Arbitrary Angles

You can also create slices that are oriented in arbitrary planes. To do this,

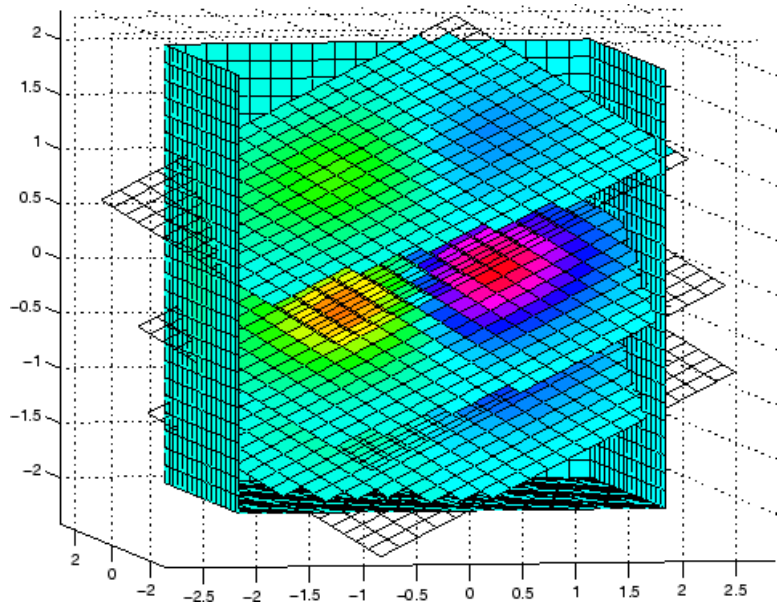
- Create a slice surface in the domain of the volume (`surf`, `linspace`).
- Orient this surface with respect to the axes (`rotate`).
- Get the `XData`, `YData`, and `ZData` of the surface (`get`).
- Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a `for` loop "passes" the plane through the volume along the *z*-axis.

```
for i = -2:.5:2
 hsp = surf(linspace(-2,2,20),linspace(-2,2,20),zeros(20)+i);
```

```
 rotate(hsp,[1,-1,1],30)
 xd = get(hsp,'XData');
 yd = get(hsp,'YData');
 zd = get(hsp,'ZData');
 delete(hsp)
 slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries
 hold on
 slice(x,y,z,v,xd,yd,zd)
 hold off
 axis tight
 view(-5,10)
 drawnow
end
```

The following picture illustrates three positions of the same slice surface as it passes through the volume.

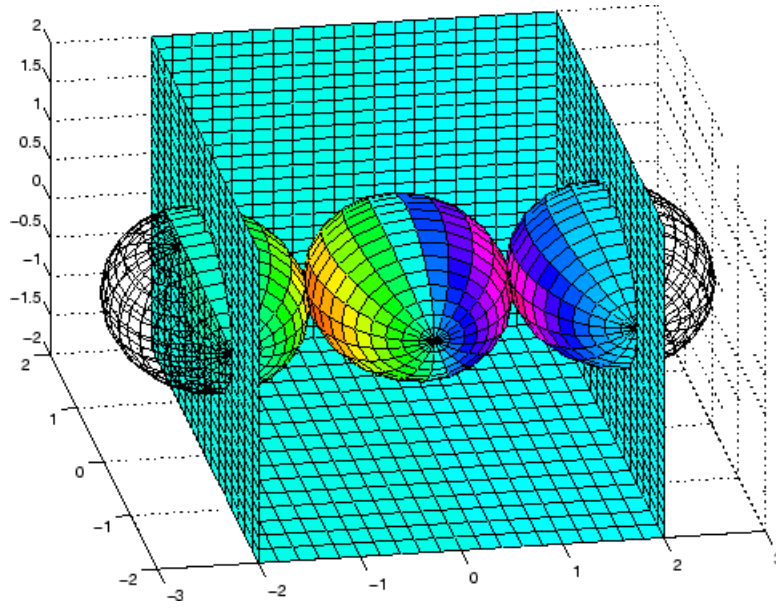


Slicing with a Nonplanar Surface

You can slice the volume with any surface. This example probes the volume created in the previous example by passing a spherical slice surface through the volume.

```
[xsp,ysp,zsp] = sphere;
slice(x,y,z,v,[-2,2],2,-2)  % Draw some volume boundaries

for i = -3:.2:3
 hsp = surface(xsp+i,ysp,zsp);
 rotate(hsp,[1 0 0],90)
 xd = get(hsp,'XData');
 yd = get(hsp,'YData');
 zd = get(hsp,'ZData');
 delete(hsp)
 hold on
 hslicer = slice(x,y,z,v,xd,yd,zd);
 axis tight
 xlim([-3,3])
 view(-10,35)
 drawnow
 delete(hslicer)
 hold off
end
```

The following picture illustrates three positions of the spherical slice surface as it passes through the volume.



See Also

`interp3`, `meshgrid`

"Volume Visualization" on page 1-102 for related functions

Exploring Volumes with Slice Planes for more examples

smooth3

Purpose

Smooth 3-D data

Syntax

Description

`W = smooth3(V)` smooths the input data `V` and returns the smoothed data in `W`.

`W = smooth3(V,'filter')` *`filter`* determines the convolution kernel and can be the strings

- `'gaussian'`
- `'box'` (default)

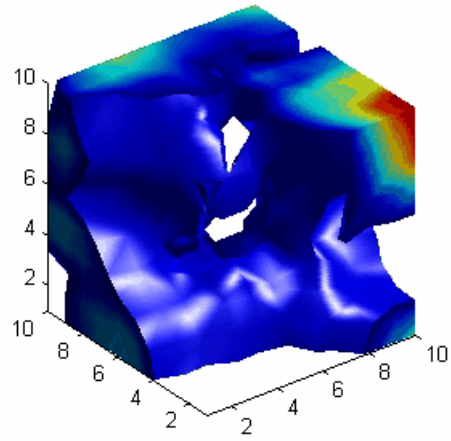
`W = smooth3(V,'filter',size)` sets the size of the convolution kernel (default is `[3 3 3]`). If `size` is scalar, then `size` is interpreted as [`size`, `size`, `size`].

`W = smooth3(V,'filter',size,sd)` sets an attribute of the convolution kernel. When *`filter`* is `gaussian`, `sd` is the standard deviation (default is .65).

Examples

This example smooths some random 3-D data and then creates an isosurface with end caps.

```
rand('seed',0)
data = rand(10,10,10);
data = smooth3(data,'box',5);
p1 = patch(isosurface(data,.5), ...
   'FaceColor','blue','EdgeColor','none');
p2 = patch(isocaps(data,.5), ...
    'FaceColor','interp','EdgeColor','none');
isonormals(data,p1)
view(3); axis vis3d tight
camlight; lighting phong
```



See Also

`isocaps`, `isonormals`, `isosurface`, `patch`

"Volume Visualization" on page 1-102 for related functions

See Displaying an Isosurface for another example.

Purpose

Sort array elements in ascending or descending order

Syntax

```
B = sort(A)
B = sort(A,dim)
B = sort(...,mode)
[B,IX] = sort(A,...)
```

Description

`B = sort(A)` sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

If A is a ...	sort(A) ...
Vector	Sorts the elements of `A`.
Matrix	Sorts each column of `A`.
Multidimensional array	Sorts `A` along the first non-singleton dimension, and returns an array of sorted vectors.
Cell array of strings	Sorts the strings in ASCII dictionary order.

Integer, floating-point, logical, and character arrays are permitted. Floating-point arrays can be complex. For elements of A with identical values, the order of these elements is preserved in the sorted list. When `A` is complex, the elements are sorted by magnitude, i.e., `abs(A)`, and where magnitudes are equal, further sorted by phase angle, i.e., `angle(A)`, on the interval $[-\pi, \pi]$. If `A` includes any `NaN` elements, `sort` places these at the high end.

`B = sort(A,dim)` sorts the elements along the dimension of `A` specified by a scalar `dim`.

`B = sort(...,mode)` sorts the elements in the specified direction, depending on the value of `mode`.

`'ascend'`	Ascending order (default)
`'descend'`	Descending order

`[B,IX] = sort(A,...)` also returns an array of indices `IX`, where `size(IX) == size(A)`. If `A` is a vector, `B = A(IX)`. If `A` is an `m`-by-`n` matrix, then each column of `IX` is a permutation vector of the corresponding column of `A`, such that

```
for j = 1:n
    B(:,j) = A(IX(:,j),j);
end
```

If `A` has repeated elements of equal value, the returned indices preserve the original ordering.

Sorting Complex Entries

If `A` has complex entries `r` and `s`, `sort` orders them according to the following rule: `r` appears before `s` in `sort(A)` if either of the following hold:

- `abs(r) < abs(s)`
- `abs(r) = abs(s)` and `angle(r)<angle(s)`

  where $-\pi < angle(r) \leq \pi$

For example,

```
v = [1 -1 i -i];
angle(v)

ans =

         0    3.1416    1.5708   -1.5708
sort(v)

ans =

        0 - 1.0000i   1.0000
        0 + 1.0000i  -1.0000
```

Note `sort` uses a different rule for ordering complex numbers than do `max` and `min`, or the relational operators < and >. See the Relational Operators reference page for more information.

Examples

Example 1

This example sorts a matrix `A` in each dimension, and then sorts it a third time, returning an array of indices for the sorted result.

```
A = [ 3 7 5
      0 4 2 ];

sort(A,1)

ans =
     0     4     2
     3     7     5

sort(A,2)

ans =
     3     5     7
     0     2     4

[B,IX] = sort(A,2)

B =
     3     5     7
     0     2     4

IX =
     1     3     2
     1     3     2
```

Example 2

This example sorts each column of a matrix in descending order.

```
A = [ 3  7  5
      6  8  3
      0  4  2 ];

sort(A,1,'descend')

ans =
     6   8   5
     3   7   3
     0   4   2
```

This is equivalent to

```
sort(A,'descend')

ans =
     6   8   5
     3   7   3
     0   4   2
```

See Also

issorted, max, mean, median, min, sortrows

Purpose

Sort rows in ascending order

Syntax

```
B = sortrows(A)
B = sortrows(A,column)
[B,index] = sortrows(A,...)
```

Description

`B = sortrows(A)` sorts the rows of `A` in ascending order. Argument `A` must be either a matrix or a column vector.

For strings, this is the familiar dictionary sort. When `A` is complex, the elements are sorted by magnitude, and, where magnitudes are equal, further sorted by phase angle on the interval $[-\pi, \pi]$.

`B = sortrows(A,column)` sorts the matrix based on the columns specified in the vector `column`. If an element of `column` is positive, MATLAB sorts the corresponding column of matrix `A` in ascending order; if an element of `column` is negative, MATLAB sorts the corresponding column in descending order. For example, `sortrows(A,[2 -3])` sorts the rows of `A` first in ascending order for the second column, and then by descending order for the third column.

`[B,index] = sortrows(A,...)` also returns an index vector `index`.

If `A` is a column vector, then `B = A(index)`. If `A` is an `m`-by-`n` matrix, then `B = A(index,:)`.

Examples

Start with a mostly random matrix, `A`:

```
rand('state',0)
A = floor(rand(6,7) * 100);
A(1:4,1)=95;  A(5:6,1)=76;  A(2:4,2)=7;  A(3,3)=73
A =
    95    45    92    41    13     1    84
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

When called with only a single input argument, `sortrows` bases the sort on the first column of the matrix. For any rows that have equal elements in a particular column, (e.g., `A(1:4,1)` for this matrix), sorting is based on the column immediately to the right, (`A(1:4,2)` in this case):

```
sortrows(A)
ans =
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
    95    45    92    41    13     1    84
```

When called with two input arguments, `sortrows` bases the sort entirely on the column specified in the second argument. Rows that have equal elements in this column are sorted; rows with equal elements in other columns are left in their original order:

```
sortrows(A,1)
ans =
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
    95    45    92    41    13     1    84
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
```

This example specifies two columns to sort by: columns 1 and 7. This tells `sortrows` to sort by column 1 first, and then for any rows with equal values in column 1, to sort by column 7:

```
sortrows(A,[1 7])
ans =
    76    79    91     0    19    41     1
    76    61    93    81    27    46    83
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
```

```
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
```

Sort the matrix using the values in column 4 this time and in reverse order:

```
sortrows(A, -4)
ans =
    95     7    73    89    20    74    52
    76    61    93    81    27    46    83
    95    45    92    41    13     1    84
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    76    79    91     0    19    41     1
```

See Also

`issorted`, `sort`

Purpose

Convert vector into sound

Syntax

```
sound(y,Fs)
sound(y)
sound(y,Fs,bits)
```

Description

`sound(y,Fs)` sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC and most UNIX platforms. Values in `y` are assumed to be in the range $-1.0 \le y \le 1.0$. Values outside that range are clipped. Stereo sound is played on platforms that support it when `y` is an `n`-by-2 matrix. The values in column 1 are assigned to the left channel, and those in column 2 to the right.

Note The playback duration that results from setting `Fs` depends on the sound card you have installed. Most sound cards support sample frequencies of approximately 5-10 kHz to 44.1 kHz. Sample frequencies outside this range can produce unexpected results.

`sound(y)` plays the sound at the default sample rate or 8192 Hz.

`sound(y,Fs,bits)` plays the sound using `bits` number of bits/sample, if possible. Most platforms support `bits = 8` or `bits = 16`.

Remarks

MATLAB supports all Windows-compatible sound devices. Additional sound acquisition and generation capability is available in the Data Acquisition Toolbox. The toolbox functionality includes the ability to buffer the acquisition so that you can analyze the data as it is being acquired. See the examples on MATLAB sound acquisition and sound generation.

See Also

`auread`, `auwrite`, `soundsc`, `audioplayer`, `wavread`, `wavwrite`

soundsc

Purpose

Scale data and play as sound

Syntax

```
soundsc(y,Fs)
soundsc(y)
soundsc(y,Fs,bits)
soundsc(y,...,slim)
```

Description

`soundsc(y,Fs)` sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC and most UNIX platforms. The signal `y` is scaled to the range $-1.0 \le y \le 1.0$ before it is played, resulting in a sound that is played as loud as possible without clipping.

Note The playback duration that results from setting `Fs` depends on the sound card you have installed. Most sound cards support sample frequencies of approximately 5-10 kHz to 44.1 kHz. Sample frequencies outside this range can produce unexpected results.

`soundsc(y)` plays the sound at the default sample rate or 8192 Hz.

`soundsc(y,Fs,bits)` plays the sound using `bits` number of bits/sample if possible. Most platforms support `bits = 8` or `bits = 16`.

`soundsc(y,...,slim)`, where `slim = [slow shigh]`, maps the values in `y` between `slow` and `shigh` to the full sound range. The default value is `slim = [min(y) max(y)]`.

Remarks

MATLAB supports all Windows-compatible sound devices.

See Also

`auread`, `auwrite`, `sound`, `wavread`, `wavwrite`

Purpose

Allocate space for sparse matrix

Syntax

```
S = spalloc(m,n,nzmax)
```

Description

`S = spalloc(m,n,nzmax)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nzmax` nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nzmax)` is shorthand for

```
sparse([],[],[],m,n,nzmax)
```

Examples

To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

```
S = spalloc(n,n,3*n);
for j = 1:n
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';end
```

Purpose

Create sparse matrix

Syntax

```
S = sparse(A)
S = sparse(i,j,s,m,n,nzmax)
S = sparse(i,j,s,m,n)
S = sparse(i,j,s)
S = sparse(m,n)
```

Description

The `sparse` function generates matrices in the MATLAB sparse storage organization.

`S = sparse(A)` converts a full matrix to sparse form by squeezing out any zero elements. If `S` is already sparse, `sparse(S)` returns `S`.

`S = sparse(i,j,s,m,n,nzmax)` uses vectors `i`, `j`, and `s` to generate an `m`-by-`n` sparse matrix such that `S(i(k),j(k)) = s(k)`, with space allocated for `nzmax` nonzeros. Vectors `i`, `j`, and `s` are all the same length. Any elements of `s` that are zero are ignored, along with the corresponding values of `i` and `j`. Any elements of `s` that have duplicate values of `i` and `j` are added together.

Note If any value in `i` or `j` is larger than the maximum integer size, `2^31-1`, then the sparse matrix cannot be constructed.

To simplify this six-argument call, you can pass scalars for the argument `s` and one of the arguments `i` or `j`—in which case they are expanded so that `i`, `j`, and `s` all have the same length.

`S = sparse(i,j,s,m,n)` uses `nzmax = length(s)`.

`S = sparse(i,j,s)` uses `m = max(i)` and `n = max(j)`. The maxima are computed before any zeros in `s` are removed, so one of the rows of `[i j s]` might be `[m n 0]`.

`S = sparse(m,n)` abbreviates `sparse([],[],[],m,n,0)`. This generates the ultimate sparse matrix, an `m`-by-`n` all zero matrix.

Remarks

All of the MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse, for example, `A.*S` is at least as sparse as `S`.

Examples

`S = sparse(1:n,1:n,1)` generates a sparse representation of the `n`-by-`n` identity matrix. The same `S` results from `S = sparse(eye(n,n))`, but this would also temporarily generate a full `n`-by-`n` matrix with most of its elements equal to zero.

`B = sparse(10000,10000,pi)` is probably not very useful, but is legal and works; it sets up a `10000`-by-`10000` matrix with only one nonzero element. Don't try `full(B)`; it requires 800 megabytes of storage.

This dissects and then reassembles a sparse matrix:

```
[i,j,s] = find(S);
[m,n] = size(S);
S = sparse(i,j,s,m,n);
```

So does this, if the last row and column have nonzero entries:

```
[i,j,s] = find(S);
S = sparse(i,j,s);
```

See Also

`diag`, `find`, `full`, `issparse`, `nnz`, `nonzeros`, `nzmax`, `spones`, `sprandn`, `sprandsym`, `spy`

The `sparfun` directory

spaugment

Purpose

Form least squares augmented system

Syntax

```
S = spaugment(A,c)
S = spaugment(A)
```

Description

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix `S = [c*I A; A' 0]`. The matrix `S` is related to the least squares problem

```
min norm(b - A*x)
```

by

```
r = b - A*x
S * [r/c; x] = [b; 0]
```

The optimum value of the residual scaling factor `c`, involves `min(svd(A))` and `norm(r)`, which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses `max(max(abs(A)))/1000`.

Note In previous versions of MATLAB, the augmented matrix was used by sparse linear equation solvers, `\` and /, for nonsquare problems. Now, MATLAB performs a least squares solve using the `qr` factorization of `A` instead.

See Also

`spparms`

Purpose

Import matrix from sparse matrix external format

Syntax

```
S = spconvert(D)
```

Description

`spconvert` is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. `spconvert` is the second step in the process:

1. Load an ASCII data file containing `[i,j,v]` or `[i,j,re,im]` as rows into a MATLAB variable.
2. Convert that variable into a MATLAB sparse matrix.

`S = spconvert(D)` converts a matrix `D` with rows containing `[i,j,s]` or `[i,j,r,s]` to the corresponding sparse matrix. `D` must have an `nnz` or `nnz+1` row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form `[m n 0]` or `[m n 0 0]` anywhere in `D` can be used to specify `size(S)`. If `D` is already sparse, no conversion is done, so `spconvert` can be used after `D` is loaded from either a MAT-file or an ASCII file.

Examples

Suppose the ASCII file `uphill.dat` contains

```
1     1     1.000000000000000
1     2     0.500000000000000
2     2     0.333333333333333
1     3     0.333333333333333
2     3     0.250000000000000
3     3     0.200000000000000
1     4     0.250000000000000
2     4     0.200000000000000
3     4     0.166666666666667
4     4     0.142857142857143
4     4     0.000000000000000
```

Then the statements

```
load uphill.dat
H = spconvert(uphill)

H =
   (1,1)       1.0000
   (1,2)       0.5000
   (2,2)       0.3333
   (1,3)       0.3333
   (2,3)       0.2500
   (3,3)       0.2000
   (1,4)       0.2500
   (2,4)       0.2000
   (3,4)       0.1667
   (4,4)       0.1429
```

recreate `sparse(triu(hilb(4)))`, possibly with roundoff errors. In this case, the last line of the input file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.

Purpose

Extract and create sparse band and diagonal matrices

Syntax

```
B = spdiags(A)
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)
```

Description

The `spdiags` function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible.

`B = spdiags(A)` extracts all nonzero diagonals from the `m`-by-`n` matrix `A`. `B` is a `min(m,n)`-by-`p` matrix whose columns are the `p` nonzero diagonals of `A`.

`[B,d] = spdiags(A)` returns a vector `d` of length `p`, whose integer components specify the diagonals in `A`.

`B = spdiags(A,d)` extracts the diagonals specified by `d`.

`A = spdiags(B,d,A)` replaces the diagonals specified by `d` with the columns of `B`. The output is sparse.

`A = spdiags(B,d,m,n)` creates an `m`-by-`n` sparse matrix by taking the columns of `B` and placing them along the diagonals specified by `d`.

Note In this syntax, if a column of `B` is longer than the diagonal it is replacing, and `m >= n`, `spdiags` takes elements of super-diagonals from the lower part of the column of `B`, and elements of sub-diagonals from the upper part of the column of `B`. However, if `m < n` , then super-diagonals are from the upper part of the column of `B`, and sub-diagonals from the lower part. (See “Example 5A” on page 2-2967 and “Example 5B” on page 2-2969, below).

Arguments

The `spdiags` function deals with three matrices, in various combinations, as both input and output.

A	An m-by-n matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on p diagonals.
B	A min(m,n)-by-p matrix, usually (but not necessarily) full, whose columns are the diagonals of A.
d	A vector of length p whose integer components specify the diagonals in A.

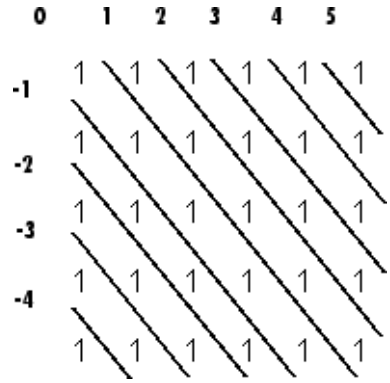
Roughly, A, B, and d are related by

```
for k = 1:p
    B(:,k) = diag(A,d(k))
end
```

Some elements of B, corresponding to positions outside of A, are not defined by these loops. They are not referenced when B is input and are set to zero when B is output.

How the Diagonals of A are Listed in the Vector d

An m-by-n matrix A has m+n-1diagonals. These are specified in the vector d using indices from -m+1 to n-1. For example, if A is 5-by-6, it has 10 diagonals, which are specified in the vector d using the indices -4, -3 , ... 4, 5. The following diagram illustrates this for a vector of all ones.



Examples

Example 1

For the following matrix,

```
A=[0 5 0 10 0 0;...
0 0 6 0 11 0;...
3 0 0 7 0 12;...
1 4 0 0 8 0;...
0 2 5 0 0 9]

A =

     0     5     0    10     0     0
     0     0     6     0    11     0
     3     0     0     7     0    12
     1     4     0     0     8     0
     0     2     5     0     0     9
```

the command

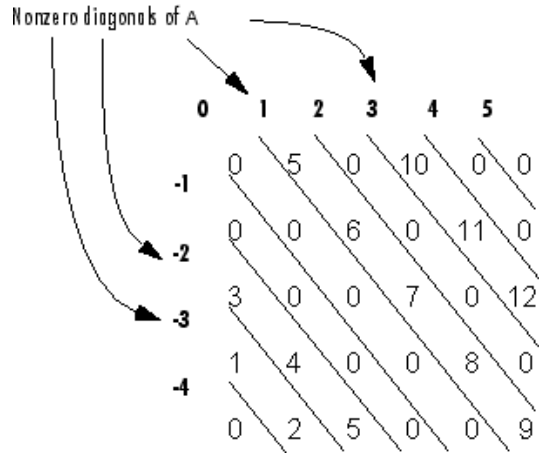
```
[B, d]  =spdiags(A)
```

returns

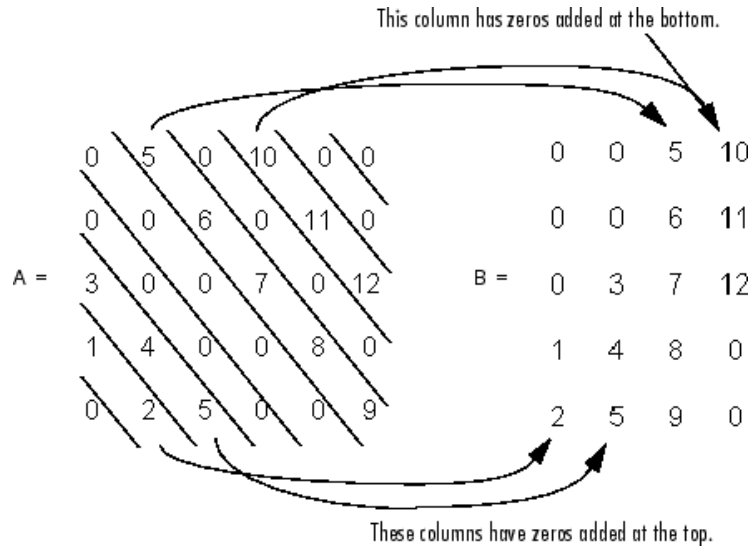
```
B =

     0     0     5    10
     0     0     6    11
     0     3     7    12
     1     4     8     0
     2     5     9     0


d =


    -3
    -2
     1
```

3

The columns of the first output B contain the nonzero diagonals of A. The second output d lists the indices of the nonzero diagonals of A, as shown in the following diagram. See “How the Diagonals of A are Listed in the Vector d” on page 2-2962.



Note that the longest nonzero diagonal in A is contained in column 3 of B. The other nonzero diagonals of A have extra zeros added to their corresponding columns in B, to give all columns of B the same length. For the nonzero diagonals below the main diagonal of A, extra zeros are added at the tops of columns. For the nonzero diagonals above the main diagonal of A, extra zeros are added at the bottoms of columns. This is illustrated by the following diagram.



Example 2

This example generates a sparse tridiagonal representation of the classic second difference operator on `n` points.

```
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson's test matrix (see `gallery`):

```
A = spdiags(abs(-(n-1)/2:(n-1)/2)',0,A)
```

Finally, recover the three diagonals:

```
B = spdiags(A)
```

Example 3

The second example is not square.

```
A = [11    0   13    0
      0   22    0   24
```

```
     0     0    33     0
    41     0     0    44
     0    52     0     0
     0     0    63     0
     0     0     0    74]
```

Here `m =7`, `n = 4`, and `p = 3`.

The statement `[B,d] = spdiags(A)` produces `d = [-3 0 2]'` and

```
B = [41    11     0
     52    22     0
     63    33    13
     74    44    24]
```

Conversely, with the above `B` and `d`, the expression `spdiags(B,d,7,4)` reproduces the original `A`.

Example 4

This example shows how `spdiags` creates the diagonals when the columns of `B` are longer than the diagonals they are replacing.

```
B = repmat((1:6)',[1 7])

B =

    1  1  1  1  1  1  1
    2  2  2  2  2  2  2
    3  3  3  3  3  3  3
    4  4  4  4  4  4  4
    5  5  5  5  5  5  5
    6  6  6  6  6  6  6

d = [-4 -2 -1 0 3 4 5];
A = spdiags(B,d,6,6);
full(A)

ans =
```

```
1   0   0   4   5   6
1   2   0   0   5   6
1   2   3   0   0   6
0   2   3   4   0   0
1   0   3   4   5   0
0   2   0   4   5   6
```

Example 5A

This example illustrates the use of the syntax `A = spdiags(B,d,m,n)`, under three conditions:

- `m` is equal to `n`
- `m` is greater than `n`
- `m` is less than `n`

The command used in this example is

```
A = full(spdiags(B, [-2 0 2], m, n))
```

where `B` is the 5-by-3 matrix shown below. The resulting matrix `A` has dimensions `m`-by-`n`, and has nonzero diagonals at `[-2 0 2]` (a sub-diagonal at `-2`, the main diagonal, and a super-diagonal at `2`).

```
B =
    1     6    11
    2     7    12
    3     8    13
    4     9    14
    5    10    15
```

The first and third columns of matrix `B` are used to create the sub- and super-diagonals of `A` respectively. In all three cases though, these two outer columns of `B` are longer than the resulting diagonals of `A`. Because of this, only a part of the columns is used in `A`.

When `m == n` or `m > n`, `spdiags` takes elements of the super-diagonal in `A` from the lower part of the corresponding column of `B`, and elements of the sub-diagonal in `A` from the upper part of the corresponding column of `B`.

When `m < n`, `spdiags` does the opposite, taking elements of the super-diagonal in `A` from the upper part of the corresponding column of `B`, and elements of the sub-diagonal in `A` from the lower part of the corresponding column of `B`.

Part 1 — m is equal to n.

```
A = full(spdiags(B, [-2 0 2], 5, 5))
 Matrix B                       Matrix A

1    6   11                    6    0   13    0    0
2    7   12                    0    7    0   14    0
3    8   13  == spdiags =>     1    0    8    0   15
4    9   14                    0    2    0    9    0
5   10   15                    0    0    3    0   10
```

`A(3,1)`, `A(4,2)`, and `A(5,3)` are taken from the upper part of `B(:,1)`.

`A(1,3)`, `A(2,4)`, and `A(3,5)` are taken from the lower part of `B(:,3)`.

Part 2 — m is greater than n.

```
A = full(spdiags(B, [-2 0 2], 5, 4))
 Matrix B                       Matrix A

1    6   11                    6    0   13    0
2    7   12                    0    7    0   14
3    8   13  == spdiags =>     1    0    8    0
4    9   14                    0    2    0    9
5   10   15                    0    0    3    0
```

Same as in Part A.

Part 3 — m is less than n.

```
A = full(spdiags(B, [-2 0 2], 4, 5))
 Matrix B                       Matrix A

1    6   11                    6    0   11    0    0
2    7   12                    0    7    0   12    0
3    8   13  == spdiags =>     3    0    8    0   13
4    9   14                    0    4    0    9    0
5   10   15
```

`A(3,1)` and `A(4,2)` are taken from the lower part of `B(:,1)`.

`A(1,3)`, `A(2,4)`, and `A(3,5)` are taken from the upper part of `B(:,3)`.

Example 5B

Extract the diagonals from the first part of this example back into a column format using the command

```
B = spdiags(A)
```

You can see that in each case the original columns are restored (minus those elements that had overflowed the super- and sub-diagonals of matrix `A`).

Part 1.

```
        Matrix A                          Matrix B

6    0   13    0    0                    1   6   0
0    7    0   14    0                    2   7   0
1    0    8    0   15  == spdiags =>     3   8  13
0    2    0    9    0                    0   9  14
0    0    3    0   10                    0  10  15
```

Part 2.

```
      Matrix A                      Matrix B
```

```
 6    0   13    0                 1   6   0
 0    7    0   14                 2   7   0
 1    0    8    0  == spdiags =>  3   8  13
 0    2    0    9                 0   9  14
 0    0    3    0
```

Part 3.

```
       Matrix A                        Matrix B

 6    0   11    0    0                 0   6  11
 0    7    0   12    0                 0   7  12
 3    0    8    0   13  == spdiags =>  3   8  13
 0    4    0    9    0                 4   9   0
```

See Also

diag, speye

Purpose

Calculate specular reflectance

Syntax

```
R = specular(Nx,Ny,Nz,S,V)
```

Description

`R = specular(Nx,Ny,Nz,S,V)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` and `V` specify the direction to the light source and to the viewer, respectively. You can specify these directions as three vectors`[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

The specular highlight is strongest when the normal vector is in the direction of `(S+V)/2` where `S` is the source direction, and `V` is the view direction.

The surface spread exponent can be specified by including a sixth argument as in `specular(Nx,Ny,Nz,S,V,spread)`.

speye

Purpose

Sparse identity matrix

Syntax

```
S = speye(m,n)
S = speye(n)
```

Description

`S = speye(m,n)` forms an `m`-by-`n` sparse matrix with `1`s on the main diagonal.

`S = speye(n)` abbreviates `speye(n,n)`.

Examples

`I =s peye(1000)` forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as `I = sparse(eye(1000,1000))`, but the latter requires eight megabytes for temporary storage for the full representation.

See Also

`spalloc`, `spones`, `spdiags`, `sprand`, `sprandn`

Purpose

Apply function to nonzero sparse matrix elements

Syntax

```
f = spfun(fun,S)
```

Description

The `spfun` function selectively applies a function to only the *nonzero* elements of a sparse matrix `S`, preserving the sparsity pattern of the original matrix (except for underflow or if `fun` returns zero for some nonzero elements of `S`).

`f = spfun(fun,S)` evaluates `fun(S)` on the nonzero elements of `S`. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions" in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

Remarks

Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

Examples

Given the 4-by-4 sparse diagonal matrix

```
S = spdiags([1:4]',0,4,4)

S =
   (1,1)        1
   (2,2)        2
   (3,3)        3
   (4,4)        4
```

Because `fun` returns nonzero values for all nonzero element of `S`, `f = spfun(@exp,S)` has the same sparsity pattern as `S`.

```
f =
   (1,1)        2.7183
   (2,2)        7.3891
   (3,3)       20.0855
   (4,4)       54.5982
```

whereas `exp(S)` has `1`s where `S` has `0`s.

```
full(exp(S))

ans =
    2.7183    1.0000    1.0000    1.0000
    1.0000    7.3891    1.0000    1.0000
    1.0000    1.0000   20.0855    1.0000
    1.0000    1.0000    1.0000   54.5982
```

See Also

`function_handle (@)`

Purpose

Transform spherical coordinates to Cartesian

Syntax

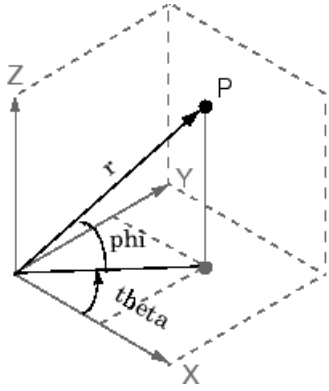
```
[x,y,z] = sph2cart(THETA,PHI,R)
```

Description

`[x,y,z] = sph2cart(THETA,PHI,R)` transforms the corresponding elements of spherical coordinate arrays to Cartesian, or *xyz*, coordinates. `THETA`, `PHI`, and `R` must all be the same size. `THETA` and `PHI` are angular displacements in radians from the positive *x*-axis and from the *x*-*y* plane, respectively.

Algorithm

The mapping from spherical coordinates to three-dimensional Cartesian coordinates is




```
x = r .* cos(phi) .* cos(theta)
y = r .* cos(phi) .* sin(theta)
        z = r .* sin(phi)
```

See Also

`cart2pol`, `cart2sph`, `pol2cart`

sphere

Purpose

Generate sphere

Syntax

```
sphere
sphere(n)
[X,Y,Z] = sphere(n)
```

Description

The `sphere` function generates the *x*-, *y*-, and *z*-coordinates of a unit sphere for use with `surf` and `mesh`.

`sphere` generates a sphere consisting of 20-by-20 faces.

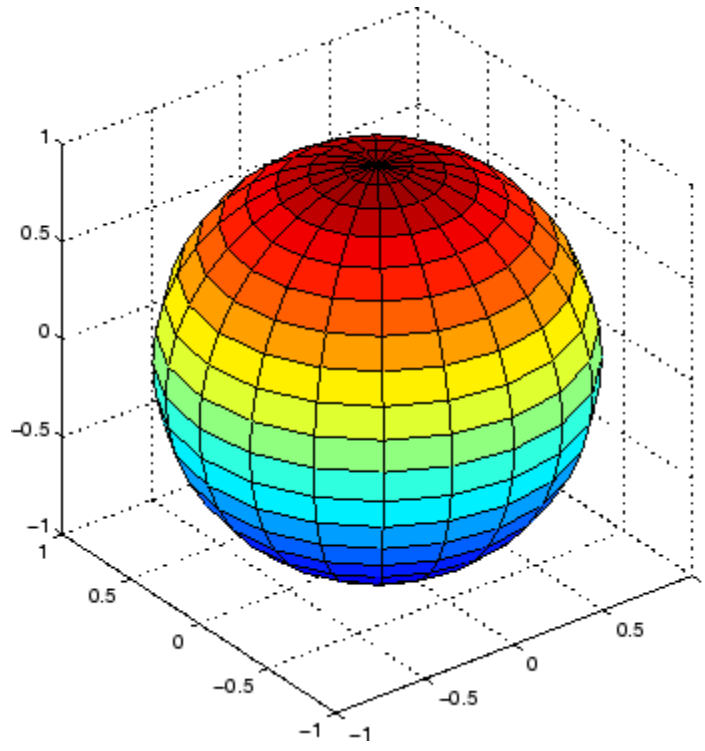
`sphere(n)` draws a `surf` plot of an `n`-by-`n` sphere in the current figure.

`[X,Y,Z] = sphere(n)` returns the coordinates of a sphere in three matrices that are `(n+1)`-by-`(n+1)` in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

Examples

Generate and plot a sphere.

```
sphere
axis equal
```



See Also

`cylinder`, `axis equal`

“Polygons and Surfaces” on page 1-90 for related functions

spinmap

Purpose

Spin colormap

Syntax

```
spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')
```

Description

The `spinmap` function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

`spinmap` cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

`spinmap(t)` rotates the colormap for approximately `10*t` seconds. The amount of time specified by `t` depends on your hardware configuration (e.g., if you are running MATLAB over a network).

`spinmap(t,inc)` rotates the colormap for approximately `10*t` seconds and specifies an increment `inc` by which the colormap shifts. When `inc` is 1, the rotation appears smoother than the default (i.e., 2). Increments greater than 2 are less smooth than the default. A negative increment (e.g., –2) rotates the colormap in a negative direction.

`spinmap('inf')` rotates the colormap for an infinite amount of time. To break the loop, press **Ctrl+C**.

See Also

`colormap`, `colormapeditor`

"Color Operations" on page 1-98 for related functions

Purpose

Cubic spline data interpolation

Syntax

```
pp = spline(x,Y)
yy = spline(x,Y,xx)
```

Description

`pp = spline(x,Y)` returns the piecewise polynomial form of the cubic spline interpolant for later use with `ppval` and the spline utility `unmkpp`. `x` must be a vector. `Y` can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If `Y` is a scalar or vector, it must have the same length as `x`. A scalar value for `x` or `Y` is expanded to have the same length as the other. See Exceptions (1) for an exception to this rule, in which the not-a-knot end conditions are used.
- If `Y` is an array that is not a vector, the size of `Y` must have the form `[d1,d2,...,dk,n]`, where `n` is the length of `x`. The interpolation is performed for each `d1-by-d2-by-...-dk` value in `Y`. See Exceptions (2) for an exception to this rule.

`yy = spline(x,Y,xx)` is the same as `yy = ppval(spline(x,Y),xx)`, thus providing, in `yy`, the values of the interpolant at `xx`. `xx` can be a scalar, a vector, or a multidimensional array. The sizes of `xx` and `yy` are related as follows:

- If `Y` is a scalar or vector, `yy` has the same size as `xx`.
- If `Y` is an array that is not a vector,
  - If `xx` is a scalar or vector, `size(yy)` equals `[d1, d2, ..., dk, length(xx)]`.
  - If `xx` is an array of size `[m1,m2,...,mj]`, `size(yy)` equals `[d1,d2,...,dk,m1,m2,...,mj]`.

Exceptions

1. If `Y` is a vector that contains two more values than `x` has entries, the first and last value in `Y` are used as the endslopes for the cubic spline. If `Y` is a vector, this means
   - `f(x) = Y(2:end-1)`
   - `df(min(x)) = Y(1)`
   - `df(max(x)) = Y(end)`
2. If `Y` is a matrix or an `N`-dimensional array with `size(Y,N)` equal to `length(x)+2`, the following hold:
   - `f(x(j))` matches the value `Y(:,...,:,j+1)` for `j=1:length(x)`
   - `Df(min(x))` matches `Y(:,:,...:,1)`
   - `Df(max(x))` matches `Y(:,:,...:,end)`

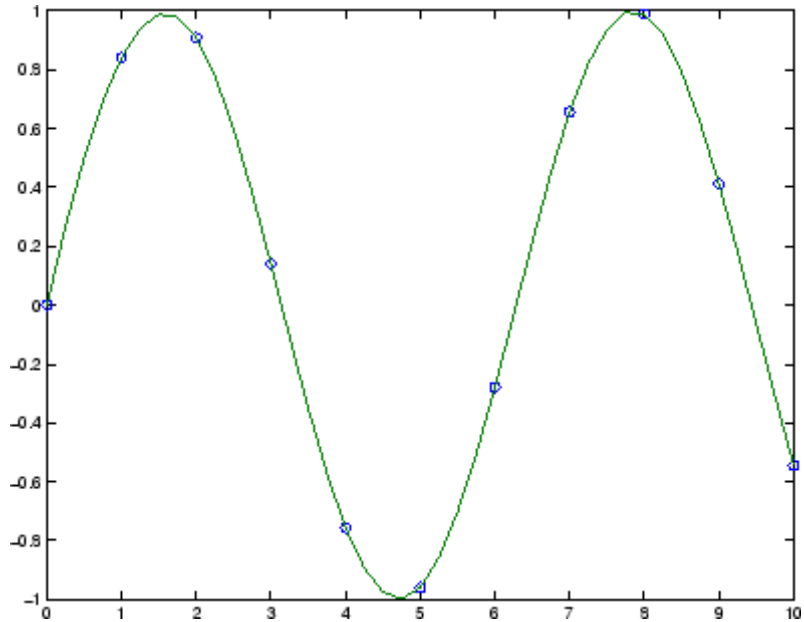
Note You can also perform spline interpolation using the `interp1` function with the command `interp1(x,y,xx,'spline')`. Note that while `spline` performs interpolation on rows of an input matrix, `interp1` performs interpolation on columns of an input matrix.

Examples

Example 1

This generates a sine curve, then samples the spline over a finer mesh.

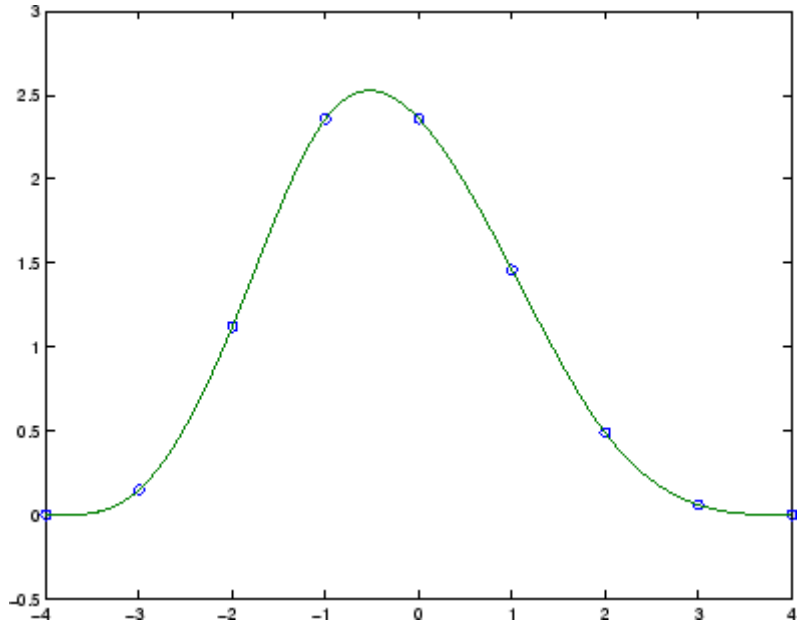
```
x = 0:10;
y = sin(x);
xx = 0:.25:10;
yy = spline(x,y,xx);
plot(x,y,'o',xx,yy)
```



Example 2

This illustrates the use of clamped or complete spline interpolation where end slopes are prescribed. Zero slopes at the ends of an interpolant to the values of a certain distribution are enforced.

```
x = -4:4;
y = [0 .15 1.12 2.36 2.36 1.46 .49 .06 0];
cs = spline(x,[0 y 0]);
xx = linspace(-4,4,101);
plot(x,y,'o',xx,ppval(cs,xx),'-');
```



Example 3

The two vectors

```
t = 1900:10:1990;
p = [ 75.995  91.972  105.711  123.203  131.669 ...
      150.697 179.323  203.212  226.505  249.633 ];
```

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

```
spline(t,p,2000)
```

uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

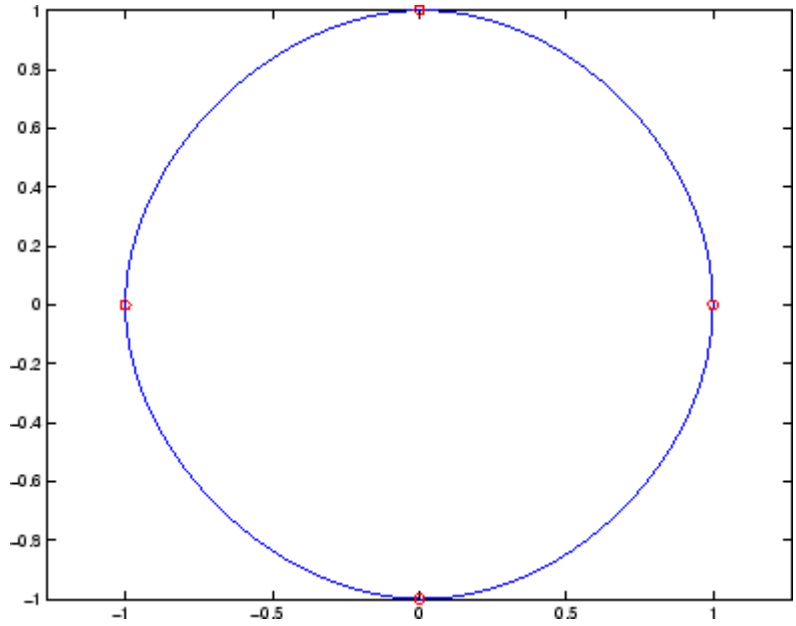
```
ans =
    270.6060
```

Example 4

The statements

```
x = pi*[0:.5:2];
y = [0  1  0 -1  0  1  0;
     1  0  1  0 -1  0  1];
pp = spline(x,y);
yy = ppval(pp, linspace(0,2*pi,101));
plot(yy(1,:),yy(2,:),'-b',y(1,2:5),y(2,2:5),'or'), axis equal
```

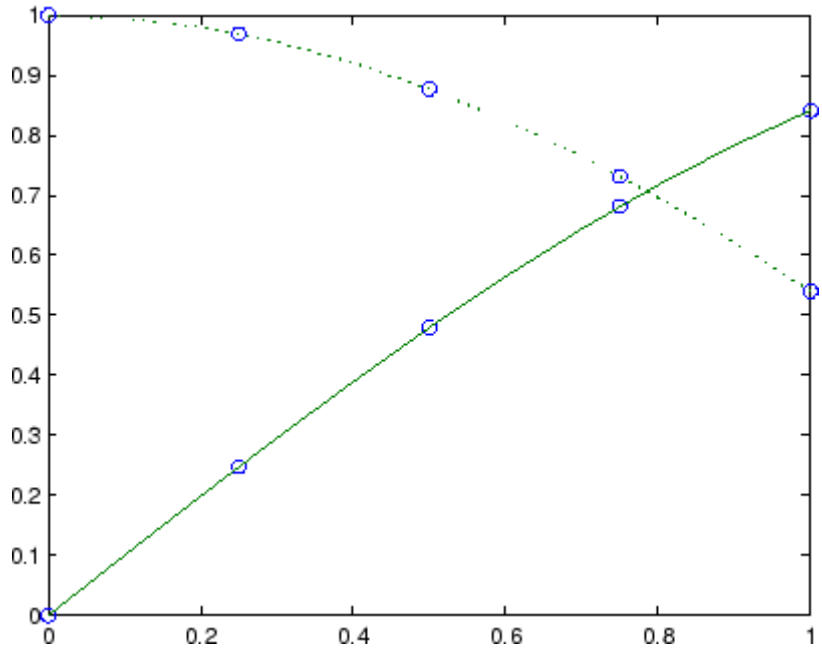
generate the plot of a circle, with the five data points `y(:,2),...,y(:,6)` marked with o's. Note that this `y` contains two more values (i.e., two more columns) than does `x`, hence `y(:,1)` and `y(:,end)` are used as endslopes.



Example 5

The following code generates sine and cosine curves, then samples the splines over a finer mesh.

```
x = 0:.25:1;
Y = [sin(x); cos(x)];
xx = 0:.1:1;
YY = spline(x,Y,xx);
plot(x,Y(1,:),'o',xx,YY(1,:),'-'); hold on;
plot(x,Y(2,:),'o',xx,YY(2,:),':'); hold off;
```



Algorithm

A tridiagonal linear system (with, possibly, several right sides) is being solved for the information needed to describe the coefficients of the various cubic polynomials which make up the interpolating spline. `spline` uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines

form a small suite of functions for working with piecewise polynomials. For access to more advanced features, see the M-file help for these functions and the Spline Toolbox.

See Also

`interp1`, `ppval`, `mkpp`, `pchip`, `unmkpp`

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

spones

Purpose

Replace nonzero sparse matrix elements with ones

Syntax

```
R = spones(S)
```

Description

`R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with `1`'s in the nonzero positions.

Examples

`c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

See Also

`nnz`, `spalloc`, `spfun`

Purpose

Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description

`spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse routines, particularly the minimum degree orderings, `colmmd` and `symmmd`, and also within sparse backslash. In ordinary use, you should never need to deal with this function.

The meanings of the `key` parameters are

`'spumoni'`	Sparse Monitor flag:
`0`	Produces no diagnostic output, the default
`1`	Produces information about choice of algorithm based on matrix structure, and about storage allocation
`2`	Also produces very detailed information about the sparse matrix algorithms
`'thr_rel'`, `'thr_abs'`	Minimum degree threshold is `thr_rel*mindegree + thr_abs`.
`'exact_d'`	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.
`'supernd'`	If positive, minimum degree amalgamates the supernodes every `supernd` stages.

`'rreduce'`	If positive, minimum degree does row reduction every `rreduce` stages.
`'wh_frac'`	Rows with `density > wh_frac` are ignored in `colmmd`.
`'autommd'`	Nonzero to use minimum degree (MMD) orderings with QR-based \ and /.
`'autoamd'`	Nonzero to use `colamd` ordering with the UMFPACK LU-based \ and /, and to use `amd` with CHOLMOD Cholesky-based \ and /.
`'piv_tol'`	Pivot tolerance used by the UMFPACK LU-based \ and /.
`'bandden'`	Band density used by LAPACK-based \ and / for banded matrices. Band density is defined as (# nonzeros in the band)/(# nonzeros in a full band). If `bandden = 1.0`, never use band solver. If `bandden = 0.0`, always use band solver. Default is `0.5`.
`'umfpack'`	Nonzero to use UMFPACK instead of the v4 LU-based solver in \ and /.
`'sym_tol'`	Symmetric pivot tolerance used by UMFPACK. See `lu` for more information about the role of the symmetric pivot tolerance.

Note LU-based \ and / (UMFPACK) on square matrices use a modified `colamd` or `amd`. Cholesky-based \ and / (CHOLMOD) on symmetric positive definite matrices use `amd`. QR-based \ and / on rectangular matrices use `colmmd`.

`spparms`, by itself, prints a description of the current settings.

`values = spparms` returns a vector whose components give the current settings.

`[keys,values] = spparms` returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

`spparms(values)`, with no output argument, sets all the parameters to the values specified by the argument vector.

`value = spparms('key')` returns the current setting of one parameter.

`spparms('default')` sets all the parameters to their default settings.

`spparms('tight')` sets the minimum degree ordering parameters to their *tight* settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for `default` and `tight` settings are

	Keyword	Default	Tight
values(1)	'spumoni'	0.0	
values(2)	'thr_rel'	1.1	1.0
values(3)	'thr_abs'	1.0	0.0
values(4)	'exact_d'	0.0	1.0
values(5)	'supernd'	3.0	1.0
values(6)	'rreduce'	3.0	1.0
values(7)	'wh_frac'	0.5	0.5
values(8)	'autommd'	1.0	
values(9)	'autoamd'	1.0	
values(10)	'piv_tol'	0.1	
values(11)	'bandden'	0.5	
values(12)	'umfpack'	1.0	
values(13)	'sym_tol'	0.001	

Notes

Sparse A\b on Symmetric Positive Definite A

Sparse `A\b` on symmetric positive definite `A` uses CHOLMOD in conjunction with the `amd` reordering routine.

The parameter `'autoamd'` turns the `amd` reordering on or off within the solver.

Sparse A\b on General Square A

Sparse `A\b` on general square `A` usually uses UMFPACK in conjunction with `amd` or a modified `colamd` reordering routine.

The parameter `'umfpack'` turns the use of the UMFPACK software on or off within the solver.

If UMFPACK is used,

- The parameter `'piv_tol'` controls pivoting within the solver.
- The parameter `'autoamd'` turns `amd` and the modified `colamd` on or off within the solver.

If UMFPACK is not used,

- An LU-based solver is used in conjunction with the `colmmd` reordering routine.
- If UMFPACK is not used, then the parameter `'autommd'` turns the `colmmd` reordering routine on or off within the solver.
- If UMFPACK is not used and `colmmd` is used within the solver, then the minimum degree parameters affect the reordering routine within the solver.

Sparse A\b on Rectangular A

Sparse `A\b` on rectangular `A` uses a QR-based solve in conjunction with the `colmmd` reordering routine.

The parameter `'autommd'` turns the `colmmd` reordering on or off within the solver.

If `colmmd` is used within the solver, then the minimum degree parameters affect the reordering routine within the solver.

See Also

`\`, `chol`, `lu`, `qr`, `colamd`, `colmmd`, `symmmd`

References

[1] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications*, Vol. 13, 1992, pp. 333-356.

[2] Davis, T. A., *UMFPACK Version 4.6 User Guide* (`http://www.cise.ufl.edu/research/sparse/umfpack/`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2002.

[3] Davis, T. A., *CHOLMOD Version 1.0 User Guide* (`http://www.cise.ufl.edu/research/sparse/cholmod`), Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, 2005.

sprand

Purpose

Sparse uniformly distributed random matrix

Syntax

```
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)
```

Description

`R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

`R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (`0 <= density <= 1`).

`R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

`sprand` uses the internal state information set with the `rand` function.

See Also

`sprandn`, `sprandsym`

Purpose

Sparse normally distributed random matrix

Syntax

```
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)
```

Description

`R = sprandn(S)` has the same sparsity structure as `S`, but normally distributed random entries with mean `0` and variance `1`.

`R = sprandn(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (`0 <= density <= 1`).

`R = sprandn(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

`sprandn` uses the internal state information set with the `randn` function.

See Also

`sprand`, `sprandsym`

sprandsym

Purpose

Sparse symmetric random matrix

Syntax

```
R = sprandsym(S)
R = sprandsym(n,density)
R = sprandsym(n,density,rc)
R = sprandsym(n,density,rc,kind)
```

Description

`R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as `S`. Its elements are normally distributed, with mean `0` and variance `1`.

`R = sprandsym(n,density)` returns a symmetric random, `n`-by-`n`, sparse matrix with approximately `density*n*n` nonzeros; each entry is the sum of one or more normally distributed random samples, and (`0 <= density <= 1`).

`R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to `rc`. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in **[–1, 1]**.

If `rc` is a vector of length `n`, then `R` has eigenvalues `rc`. Thus, if `rc` is a positive (nonnegative) vector then `R` is a positive definite matrix. In either case, `R` is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

`R = sprandsym(n,density,rc,kind)` returns a positive definite matrix. Argument `kind` can be:

- `1` to generate `R` by random Jacobi rotation of a positive definite diagonal matrix. `R` has the desired condition number exactly.
- `2` to generate an `R` that is a shifted sum of outer products. `R` has the desired condition number only approximately, but has less structure.
- `3` to generate an `R` that has the same structure as the matrix `S` and approximate condition number `1/rc`. `density` is ignored.

See Also

`sprand`, `sprandn`

Purpose

Structural rank

Syntax

```
r = sprank(A)
```

Description

`r = sprank(A)` is the structural rank of the sparse matrix `A`. For all values of `A`,

```
sprank(A) >= rank(full(A))
```

In exact arithmetic, `sprank(A) == rank(full(sprandn(A)))` with a probability of one.

Examples

```
A = [1    0    2    0
     2    0    4    0 ];

A = sparse(A);

sprank(A)

ans =
     2

rank(full(A))

ans =
     1
```

See Also

`dmperm`

Purpose

Write formatted data to string

Syntax

```
[s, errmsg] = sprintf(format, A, ...)
```

Description

`[s, errmsg] = sprintf(format, A, ...)` formats the data in matrix `A` (and in any additional matrix arguments) under control of the specified `format` string and returns it in the MATLAB string variable `s`. The `sprintf` function returns an error message string `errmsg` if an error occurred. `errmsg` is an empty matrix if no error occurred.

`sprintf` is the same as `fprintf` except that it returns the data in a MATLAB string variable rather than writing it to a file.

See "Formatting Strings" in the MATLAB Programming documentation for more detailed information on using string formatting commands.

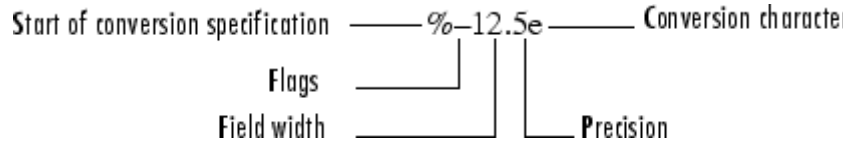
Format String

The `format` argument is a string containing ordinary characters and/or C language conversion specifications. A conversion specification controls the notation, alignment, significant digits, field width, and other aspects of output format. The format string can contain escape characters to represent nonprinting characters such as newline characters and tabs.

Conversion specifications begin with the `%` character and contain these optional and required elements:

- Flags (optional)
- Width and precision fields (optional)
- A subtype specifier (optional)
- Conversion character (required)

You specify these elements in the following order:



Flags

You can control the alignment of the output using any of these optional flags.

Character	Description	Example
A minus sign (-)	Left-justifies the converted argument in its field	`% 5.2d`
A plus sign (+)	Always prints a sign character (+ or –)	`%+5.2d`
Zero (0)	Pad with zeros rather than spaces.	`%05.2f`

Field Width and Precision Specifications

You can control the width and precision of the output by including these options in the format string.

Character	Description	Example
Field width	A digit string specifying the minimum number of digits to be printed.	`%6f`
Precision	A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point	`%6.2f`

Conversion Characters

Conversion characters specify the notation of the output.

Specifier	Description
`%c`	Single character
`%d`	Decimal notation (signed)
`%e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`, as defined in [2]. Insignificant zeros do not print.
`%G`	Same as `%g`, but using an uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a–f`)
`%X`	Hexadecimal notation (using uppercase letters `A–F`)

The following tables describe the nonalphanumeric characters found in format specification strings.

Escape Characters

This table lists the escape character sequences you use to specify non-printing characters in a format specification.

Character	Description
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\\	Backslash
\””r ”” (two single quotes)	Single quotation mark
%%	Percent character

Remarks

The sprintf function behaves like its ANSI C language namesake with these exceptions and extensions.

- If you use sprintf to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the fix, floor, ceil, or round functions to change the value in the double into a value that can be represented as an integer before passing it to sprintf.
- The following nonstandard subtype specifiers are supported for the conversion characters %o, %u, %x, and %X.

b	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '%bx'.
t	The underlying C data type is a float rather than an unsigned integer.

For example, to print a double value in hexadecimal use the format `'%bx'`.

- The `sprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of `A` (columnwise) until all the elements are used up. The function then continues in a similar manner through any additional matrix arguments.
- If `%s` is used to print part of a nonscalar double argument, the following behavior occurs:

  a Successive values are printed as long as they are integers and in the range of a valid character. The first invalid character terminates the printing for this `%s` specifier and is used for a later specifier. For example, `pi` terminates the string below and is printed using `%f` format.

  ```
  Str = [65 66 67 pi];
  sprintf('%s %f', Str)
  ans =
  ABC 3.141593
  ```

  b If the first value to print is not a valid character, then just that value is printed for this `%s` specifier using an `e` conversion as a warning to the user. For example, `pi` is formatted by `%s` below in exponential notation, and `65`, though representing a valid character, is formatted as fixed-point (`%f`).

  ```
  Str = [pi 65 66 67];
  sprintf('%s %f %s', Str)
  ans =
  3.141593e+000 65.000000 BC
  ```

  c One exception is zero, which is a valid character. If zero is found first, `%s` prints nothing and the value is skipped. If zero is found after at least one valid character, it terminates the printing for this `%s` specifier and is used for a later specifier.

- `sprintf` prints negative zero and exponents differently on some platforms, as shown in the following tables.

Negative Zero Printed with %e, %E, %f, %g, or %G

	Display of Negative Zero		
Platform	**%e or %E**	**%f**	**%g or %G**
PC	0.000000e+000	0.000000	0
Others	-0.000000e+00	-0.000000	-0

Exponents Printed with %e, %E, %g, or %G

Platform	Minimum Digits in Exponent	Example
PC	3	`1.23e+004`
UNIX	2	`1.23e+04`

You can resolve this difference in exponents by postprocessing the results of `sprintf`. For example, to make the PC output look like that of UNIX, use

```
a = sprintf('%e', 12345.678);
if ispc, a = strrep(a, 'e+0', 'e+');   end
```

Examples

Command	Result
`sprintf('%0.5g',(1+sqrt(5))/2)`	`1.618`
`sprintf('%0.5g',1/eps)`	`4.5036e+15`
`sprintf('%15.5f',1/eps)`	`4503599627370496.00000`
`sprintf('%d',round(pi))`	`3`

Command	Result
`sprintf('%s','hello')`	`hello`
`sprintf('The array is %dx%d.',2,3)`	`The array is 2x3`
`sprintf('\n')`	Line termination character on all platforms

See Also

`int2str`, `num2str`, `sscanf`

References

[1] Kernighan, B.W., and D.M. Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: “Programming Language C,” ANSI, 1430 Broadway, New York, NY 10018.

Purpose

Visualize sparsity pattern

Syntax

```
spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',markersize)
```

Description

plots the

`spy(S)` sparsity pattern of any matrix `S`.

`spy(S,markersize)`, where `markersize` is an integer, plots the sparsity pattern using markers of the specified point size.

`spy(S,'LineSpec')`, where `LineSpec` is a string, uses the specified plot marker type and color.

`spy(S,'LineSpec',markersize)` uses the specified type, color, and size for the plot markers.

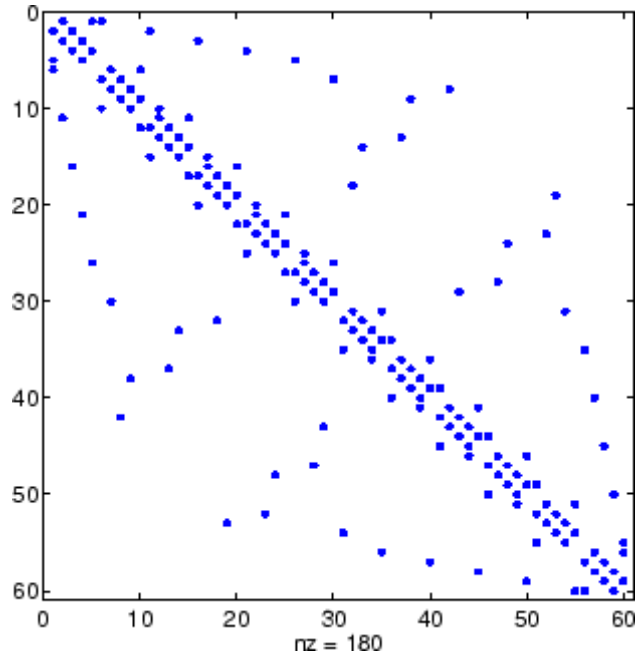
`S` is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

Note `spy` replaces `format +`, which takes much more space to display essentially the same information.

Examples

This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

```
B = bucky;
spy(B)
```



See Also

`find`, `gplot`, `LineSpec`, `symamd`, `symrcm`

Purpose

Square root

Syntax

```
B = sqrt(X)
```

Description

`B = sqrt(X)` returns the square root of each element of the array `X`. For the elements of `X` that are negative or complex, `sqrt(X)` produces complex results.

Remarks

See `sqrtm` for the matrix square root.

Examples

```
sqrt((-2:2)')
ans =
      0 + 1.4142i
      0 + 1.0000i
      0
 1.0000
 1.4142
```

See Also

`sqrtm`, `realsqrt`

Purpose

Matrix square root

Syntax

```
X = sqrtm(A)
[X, resnorm] = sqrtm(A)
[X, alpha, condest] = sqrtm(A)
```

Description

`X = sqrtm(A)` is the principal square root of the matrix `A`, i.e. `X*X = A`.

`X` is the unique square root for which every eigenvalue has nonnegative real part. If `A` has any eigenvalues with negative real parts then a complex result is produced. If `A` is singular then `A` may not have a square root. A warning is printed if exact singularity is detected.

`[X, resnorm] = sqrtm(A)` does not print any warning, and returns the residual, `norm(A-X^2,'fro')/norm(A,'fro')`.

`[X, alpha, condest] = sqrtm(A)` returns a stability factor `alpha` and an estimate `condest` of the matrix square root condition number of `X`. The residual `norm(A-X^2,'fro')/norm(A,'fro')` is bounded approximately by `n*alpha*eps` and the Frobenius norm relative error in `X` is bounded approximately by `n*alpha*condest*eps`, where `n = max(size(A))`.

Remarks

If `X` is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like `X = [0 1; 0 0]`, do not have any square roots, real or complex, and `sqrtm` cannot be expected to produce one.

Examples

Example 1

A matrix representation of the fourth difference operator is

```
X =
     5    -4     1     0     0
    -4     6    -4     1     0
     1    -4     6    -4     1
     0     1    -4     6    -4
     0     0     1    -4     5
```

This matrix is symmetric and positive definite. Its unique positive definite square root, `Y = sqrtm(X)`, is a representation of the second difference operator.

```
 Y =
      2    -1    -0    -0    -0
     -1     2    -1     0    -0
      0    -1     2    -1     0
     -0     0    -1     2    -1
     -0    -0    -0    -1     2
```

Example 2

The matrix

```
X =
     7    10
    15    22
```

has four square roots. Two of them are

```
Y1 =
    1.5667    1.7408
    2.6112    4.1779
```

and

```
Y2 =
     1     2
     3     4
```

The other two are `-Y1` and `-Y2`. All four can be obtained from the eigenvalues and vectors of `X`.

```
[V,D] = eig(X);
D =
    0.1386         0
         0   28.8614
```

The four square roots of the diagonal matrix `D` result from the four choices of sign in

```
S =
    -0.3723          0
         0     -5.3723
```

All four `Y`s are of the form

```
Y = V*S/V
```

The `sqrtm` function chooses the two plus signs and produces `Y1`, even though `Y2` is more natural because its entries are integers.

See Also

`expm`, `funm`, `logm`

Purpose

Remove singleton dimensions

Syntax

```
B = squeeze(A)
```

Description

`B = squeeze(A)` returns an array `B` with the same elements as `A`, but with all singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. Two-dimensional arrays are unaffected by `squeeze`; if `A` is a row or column vector or a scalar (1-by-1) value, then `B = A`.

Examples

Consider the 2-by-1-by-3 array `Y = rand(2,1,3)`. This array has a singleton column dimension — that is, there's only one column per page.

```
Y =

Y(:,:,1) =      Y(:,:,2) =
       0.5194          0.0346
       0.8310          0.0535

Y(:,:,3) =
       0.5297
       0.6711
```

The command `Z = squeeze(Y)` yields a 2-by-3 matrix:

```
Z =
    0.5194    0.0346    0.5297
    0.8310    0.0535    0.6711
```

Consider the 1-by-1-by-5 array `mat=repmat(1,[1,1,5])`. This array has only one scalar value per page.

```
mat =

mat(:,:,1) =  mat(:,:,2) =

     1        1
```

```
mat(:,:,3) =   mat(:,:,4) =

     1        1

mat(:,:,5) =

     1
```

The command `squeeze(mat)` yields a 5-by-1 matrix:

```
squeeze(mat)

ans =

     1
     1
     1
     1
     1

size(squeeze(mat))

ans =

    5     1
```

See Also

`reshape`, `shiftdim`

Purpose

Convert state-space filter parameters to transfer function form

Syntax

```
[b,a] = ss2tf(A,B,C,D,iu)
```

Description

`ss2tf` converts a state-space representation of a given system to an equivalent transfer function representation.

`[b,a] = ss2tf(A,B,C,D,iu)` returns the transfer function

$$H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D$$

of the system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

from the `iu`-th input. Vector `a` contains the coefficients of the denominator in descending powers of s. The numerator coefficients are returned in array `b` with as many rows as there are outputs y. `ss2tf` also works with systems in discrete time, in which case it returns the z-transform representation.

The `ss2tf` function is part of the standard MATLAB language.

Algorithm

The `ss2tf` function uses `poly` to find the characteristic polynomial det(sI-A) and the equality:

$$H(s) = C(sI - A)^{-1}B = \frac{\det(sI - A + BC) - \det(sI - A)}{\det(sI - A)}$$

Purpose

Read formatted data from string

Syntax

```
A = sscanf(s, format)
A = sscanf(s, format, size)
[A, count, errmsg, nextindex] = sscanf(...)
```

Description

`A = sscanf(s, format)` reads data from the MATLAB string `s`, converts it according to the specified `format` string, and returns it in matrix `A`. `format` is a string specifying the format of the data to be read. See "Remarks" for details. `sscanf` is the same as `fscanf` except that it reads the data from a MATLAB string rather than reading it from a file. If `s` is a character array with more than one row, `sscanf` reads the characters in column order.

`A = sscanf(s, format, size)` reads the amount of data specified by `size` and converts it according to the specified `format` string. `size` is an argument that determines how much data is read. Valid options are

`n`	Read at most `n` numbers, characters, or strings.
`inf`	Read to the end of the input string.
`[m,n]`	Read at most `(m*n)` numbers, characters, or strings. Fill a matrix of at most `m` rows in column order. `n` can be `inf`, but `m` cannot.

Characteristics of the output matrix `A` depend on the values read from the input string and on the `size` argument. If `sscanf` reads only numbers, and if `size` is not of the form `[m,n]`, matrix `A` is a column vector of numbers. If `sscanf` reads only characters or strings, and if `size` is not of the form `[m,n]`, matrix `A` is a row vector of characters. See the Remarks section for more information.

`sscanf` differs from its C language namesake `scanf()` in an important respect — it is *vectorized* to return a matrix argument. The `format` string is cycled through the input string until the first of these conditions occurs:

- The `format` string fails to match the data in the input string

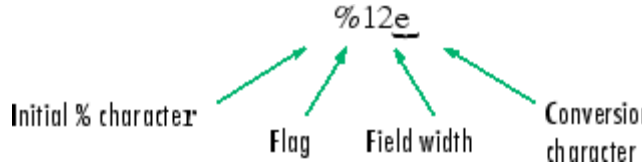
- The amount of data specified by `size` is read
- The end of the string is reached

`[A, count, errmsg, nextindex] = sscanf(...)` reads data from the MATLAB string (character array) `s`, converts it according to the specified `format` string, and returns it in matrix `A`. `count` is an optional output argument that returns the number of values successfully read. `errmsg` is an optional output argument that returns an error message string if an error occurred or an empty string if an error did not occur. `nextindex` is an optional output argument specifying one more than the number of characters scanned in `s`.

Remarks

When MATLAB reads a specified string, it attempts to match the data in the input string to the format string. If a match occurs, the data is written into the output matrix. If a partial match occurs, only the matching data is written to the matrix, and the read operation stops.

The `format` string consists of ordinary characters and/or conversion specifications. Conversion specifications indicate the type of data to be matched and involve the character `%`, optional width fields, and conversion characters, organized as shown below:



Add one or more of these characters between the `%` and the conversion character.

An asterisk (`*`)	Skip over the matched value and do not store it in the output matrix
A digit string	Maximum field width
A letter	The size of the receiving object; for example, `h` for short, as in `%hd` for a short integer, or `l` for long, as in `%ld` for a long integer or `%lg` for a double floating-point number

Valid conversion characters are as shown.

`%c`	Sequence of characters; number specified by field width
`%d`	Base 10 integers
`%e, %f, %g`	Floating-point numbers
`%i`	Defaults to signed base 10 integers. Data starting with `0` is read as base 8. Data starting with `0x` or `0X` is read as base 16.
`%o`	Signed octal integer returned as unsigned
`%s`	A series of non-white-space characters
`%u`	Signed decimal integer
`%x`	Signed hexadecimal integer returned as unsigned
`[...]`	Sequence of characters (`scanlist`)

Format specifiers `%e`, `%f`, and `%g` accept the text `'inf'`, `'-inf'`, `'nan'`, and `'-nan'`. This text is not case sensitive. The `sscanf` function converts these to the numeric representation of `Inf`, `-Inf`, `NaN`, and `-NaN`.

Use `%c` to read space characters, or `%s` to skip all white space.

For more information about format strings, refer to the `scanf()` and `fscanf()` routines in a C language reference manual.

Output Characteristics: Only Numeric Values Read

Format characters that cause `sscanf` to read numbers from the input string are `%d`, `%e`, `%f`, `%g`, `%i`, `%o`, `%u`, and `%x`. When `sscanf` reads only numbers from the input string, the elements of the output matrix `A` are numbers.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a scalar `n`, `sscanf` reads at most `n` numbers from the input string. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a matrix `[m,n]`, `sscanf` reads at most `(m*n)` numbers from the input string. The output matrix contains at most `m` rows and `n` columns. `sscanf` fills the output matrix in column order, using as many columns as it needs to contain all the numbers read from the input. Any unfilled elements in the final column contain zeros.

Output Characteristics: Only Character Values Read

The format characters that cause `sscanf` to read characters and strings from the input string are `%c` and `%s`. When `sscanf` reads only characters and strings from the input string, the elements of the output matrix `A` are characters. When `sscanf` reads a string from the input, the output matrix includes one element for each character in the string.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a row vector with one element for each character read from the input.

When the `size` argument is a scalar `n`, `sscanf` reads at most `n` character or string values from the input string. The output matrix is a row vector with one element for each character read from the input. When string values are read from the input, the output matrix can contain more than `n` columns.

When the `size` argument is a matrix `[m,n]`, `sscanf` reads at most `(m*n)` character or string values from the input string. The output

matrix contains at most `m` rows. `sscanf` fills the output matrix in column order, using as many columns as it needs to contain all the characters read from the input. When string values are read from the input, the output matrix can contain more than `n` columns. Any unfilled elements in the final column contain `char(0)`.

Output Characteristics: Both Numeric and Character Values Read

When `sscanf` reads a combination of numbers and either characters or strings from the input string, the elements of the output matrix `A` are numbers. This is true even when a format specifier such as `'%*d %s'` tells MATLAB to ignore numbers in the input string and output only characters or strings. When `sscanf` reads a string from the input, the output matrix includes one element for each character in the string. All characters are converted to their numeric equivalents in the output matrix.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a column vector with one element for each character read from the input.

When the `size` argument is a scalar `n`, `sscanf` reads at most `n` number, character, or string values from the input string. The output matrix contains at most `n` rows. `sscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than one column. Any unfilled elements in the final column contain zeros.

When the `size` argument is a matrix `[m,n]`, `sscanf` reads at most `(m*n)` number, character, or string values from the input string. The output matrix contains at most `m` rows. `sscanf` fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than `n` columns. Any unfilled elements in the final column contain zeros.

Note This section applies only when `sscanf` actually reads a combination of numbers and either characters or strings from the input string. Even if the format string has both format characters that would result in numbers (such as `%d`) and format characters that would result in characters or strings (such as `%s`), `sscanf` might actually read only numbers or only characters or strings. If `sscanf` reads only numbers, see "Output Characteristics: Only Numeric Values Read" on page 2-3015. If `sscanf` reads only characters or strings, see "Output Characteristics: Only Character Values Read" on page 2-3015.

Examples

Example 1

The statements

```
s = '2.7183  3.1416';
A = sscanf(s,'%f')
```

create a two-element vector containing poor approximations to `e` and `pi`.

Example 2

When using the `%i` conversion specifier, `sscanf` reads data starting with `0` as base 8 and returns the converted value as signed:

```
sscanf('-010', '%i')
ans =
    -8
```

When using `%o`, on the other hand, `sscanf` returns the converted value as unsigned:

```
sscanf('-010', '%o')
ans =
  4.2950e+009
```

Example 3

Create character array `A` representing both character and numeric data:

```
A = ['abc 46 6 ghi'; 'def 7 89 jkl']
A =
   abc 46 6 ghi
   def 7 89 jkl
```

Read `A` into 2-by-N matrix `B`, ignoring the character data. As stated in the Description section, `sscanf` reads the characters in `A` in column order, filling matrix `B` in column order:

```
B = sscanf(A, '%*s %d %d %*s', [2, inf])
B =
   476
   869
```

If you want `sscanf` to return the numeric data in `B` in the same order as in A, you can use this technique:

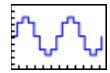
```
for k = 1:2
   C(k,:) = sscanf(A(k, :)', '%*s %d %d %*s', [1, inf]);
end

C
C =
    46     6
     7    89
```

See Also

`eval`, `sprintf`, `textread`

Purpose

Stairstep graph



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
stairs(Y)
stairs(X,Y)
stairs(...,LineSpec)
stairs(...,'PropertyName',propertyvalue)
stairs(axes_handle,...)
h = stairs(...)
[xb,yb] = stairs(Y,...)
hlines = stairs('v6',...)
```

Description

Stairstep graphs are useful for drawing time-history graphs of digitally sampled data.

`stairs(Y)` draws a stairstep graph of the elements of `Y`, drawing one line per column for matrices. The axes `ColorOrder` property determines the color of the lines.

When `Y` is a vector, the *x*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *x*-axis scale ranges from `1` to the number of rows in `Y`.

`stairs(X,Y)` plots the elements in `Y` at the locations specified in `X`.

`X` must be the same size as `Y` or, if `Y` is a matrix, `X` can be a row or a column vector such that

```
length(X) = size(Y,1)
```

`stairs(...,LineSpec)` specifies a line style, marker symbol, and color for the graph. (See `LineSpec` for more information.)

`stairs(...,'PropertyName',propertyvalue)` creates the stairstep graph, applying the specified property settings. See `Stairseries` properties for a description of properties.

`stairs(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes object (`gca`).

`h = stairs(...)` returns the handles of the stairseries objects created (one per matrix column).

`[xb,yb] = stairs(Y,...)` does not draw graphs, but returns vectors `xb` and `yb` such that `plot(xb,yb)` plots the stairstep graph.

Backward-Compatible Version

`hlines = stairs('v6',...)` returns the handles of line objects instead of stairseries objects for compatibility with MATLAB 6.5 and earlier.

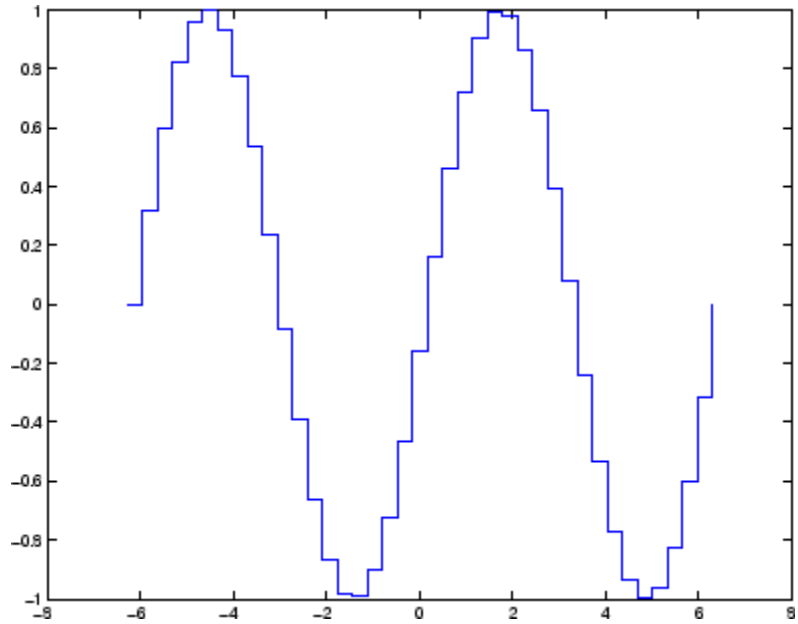
Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Examples

Create a stairstep plot of a sine wave.

```
x = linspace(-2*pi,2*pi,40);
stairs(x,sin(x))
```



See Also

`bar`, `hist`, `stem`

"Discrete Data Plots" on page 1-89 for related functions

`Stairseries Properties` for property descriptions

Stairseries Properties

Purpose

Define stairseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default property values for stairseries objects.

See Plot Objects for information on stairseries objects.

Stairseries Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of stairseries objects in legends. The `Annotation` property enables you to specify whether this stairseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the stairseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the stairseries object in a legend as one entry, but not its children objects

IconDisplayStyle Value	Purpose
`off`	Do not include the stairseries or its children in a legend (default)
`children`	Include only the children of the stairseries as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file

- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

The expression executes in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.

See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying

the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string (default is empty string)

String used by legend for this stairseries object. The `legend` function uses the string defined by the `DisplayName` property to label this stairseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this stairseries object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

```
EraseMode
     {normal} | none | xor | background
```

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`

`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`

`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`

character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the

`Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`

string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For stairseries objects, `Type` is 'hggroup'. The following statement finds all the hggroup objects in the current axes object.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
array

X-axis location of stairs. The `stairs` function uses `XData` to label the *x*-axis. `XData` can be either a matrix equal in size to `YData` or a vector equal in length to the number of rows in `YData`. That is, `length(XData) == size(YData,1)`.

If you do not specify `XData` (i.e., the input argument `x`), the `stairs` function uses the indices of `YData` to create the stairstep graph. See the `XDataMode` property for related information.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `x` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
scalar, vector, or matrix

Stairs plot data. `YData` contains the data plotted in the stairstep graph. Each value in `YData` is represented by a marker in the stairstep graph. If `YData` is a matrix, the `stairs` function creates a line for each column in the matrix.

The input argument `y` in the `stairs` function calling syntax assigns values to `YData`.

`YDataSource`
string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Start timer(s) running

Syntax

```
start(obj)
```

Description

`start(obj)` starts the timer running, represented by the timer object, `obj`. If `obj` is an array of timer objects, `start` starts all the timers. Use the `timer` function to create a timer object.

`start` sets the `Running` property of the timer object, `obj`, to `'on'`, initiates `TimerFcn` callbacks, and executes the `StartFcn` callback.

The timer stops running if one of the following conditions apply:

- The first `TimerFcn` callback completes, if `ExecutionMode` is `'singleShot'`.
- The number of `TimerFcn` callbacks specified in `TasksToExecute` have been executed.
- The `stop(obj)` command is issued.
- An error occurred while executing a `TimerFcn` callback.

See Also

`timer`, `stop`

Purpose

Start timer(s) running at specified time

Syntax

```
startat(obj,time)
startat(obj,S)
startat(obj,S,pivotyear)
startat(obj,Y,M,D)
startat(obj,[Y,M,D])
startat(obj,Y,M,D,H,MI,S)
startat(obj,[Y,M,D,H,MI,S])
```

Description

`startat(obj,time)` starts the timer running, represented by the timer object `obj`, at the time specified by the serial date number `time`. If `obj` is an array of timer objects, `startat` starts all the timers running at the specified time. Use the `timer` function to create the timer object.

`startat` sets the `Running` property of the timer object, `obj`, to `'on'`, initiates `TimerFcn` callbacks, and executes the `StartFcn` callback.

The serial date number, `time`, indicates the number of days that have elapsed since 1-Jan-0000 (starting at 1). See `datenum` for additional information about serial date numbers.

`startat(obj,S)` starts the timer running at the time specified by the date string `S`. The date string must use date format 0, 1, 2, 6, 13, 14, 15, 16, or 23, as defined by the `datestr` function. Date strings with two-character years are interpreted to be within the 100 years centered on the current year.

`startat(obj,S,pivotyear)` uses the specified pivot year as the starting year of the 100-year range in which a two-character year resides. The default pivot year is the current year minus 50 years.

`startat(obj,Y,M,D)` `startat(obj,[Y,M,D])` start the timer at the year (`Y`), month (`M`), and day (`D`) specified. `Y`, `M`, and `D` must be arrays of the same size (or they can be a scalar).

`startat(obj,Y,M,D,H,MI,S)` `startat(obj,[Y,M,D,H,MI,S])` start the timer at the year (`Y`), month (`M`), day (`D`), hour (`H`), minute (`MI`), and second (`S`) specified. `Y`, `M`, `D`, `H`, `MI`, and `S` must be arrays of the same size (or they can be a scalar). Values outside the normal range of each array

are automatically carried to the next unit (for example, month values greater than 12 are carried to years). Month values less than 1 are set to be 1; all other units can wrap and have valid negative values.

The timer stops running if one of the following conditions apply:

- The number of `TimerFcn` callbacks specified in `TasksToExecute` have been executed.
- The `stop(obj)` command is issued.
- An error occurred while executing a `TimerFcn` callback.

Examples

This example uses a timer object to execute a function at a specified time.

```
t1=timer('TimerFcn','disp(''it is 10 o''''clock'')');
startat(t1,'10:00:00');
```

This example uses a timer to display a message when an hour has elapsed.

```
t2=timer('TimerFcn','disp(''It has been an hour now.'')');
startat(t2,now+1/24);
```

See Also

`datenum`, `datestr`, `now`, `timer`, `start`, `stop`

startup

Purpose

MATLAB startup M-file for user-defined options

Syntax

```
startup
```

Description

`startup` automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`, when MATLAB starts. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on the MATLAB search path.

You can create a `startup.m` file in your own MATLAB startup directory. The file can include physical constants, Handle Graphics defaults, engineering conversion factors, or anything else you want predefined in your workspace.

There are other ways to predefine aspects of MATLAB. See Startup Options and About Preferences in the MATLAB Desktop Tools and Development Environment documentation.

Algorithm

Only `matlabrc.m` is actually invoked by MATLAB at startup. However, `matlabrc.m` contains the statements

```
if exist('startup')==2
    startup
end
```

that invoke `startup.m`. You can extend this process to create additional startup M-files, if required.

See Also

`matlabrc`, `matlabroot`, `path`, `quit`

Purpose

Standard deviation

Syntax

```
s = std(X)
s = std(X,flag)
s = std(X,flag,dim)
```

Definition

There are two common textbook definitions for the standard deviation `s` of a data vector `X`.

$$(1)\quad s = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

$$(2)\quad s = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

where

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

and n is the number of elements in the sample. The two forms of the equation differ only in $n - 1$ versus n in the divisor.

Description

`s = std(X)`, where `X` is a vector, returns the standard deviation using (1) above. The result `s` is the square root of an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples.

If `X` is a matrix, `std(X)` returns a row vector containing the standard deviation of the elements of each column of `X`. If `X` is a multidimensional array, `std(X)` is the standard deviation of the elements along the first nonsingleton dimension of `X`.

`s = std(X,flag)` for `flag = 0`, is the same as `std(X)`. For `flag = 1`, `std(X,1)` returns the standard deviation using (2) above, producing the second moment of the set of values about their mean.

`s = std(X,flag,dim)` computes the standard deviations along the dimension of `X` specified by scalar `dim`. Set `flag` to `0` to normalize `Y` by n-1; set `flag` to `1` to normalize by n.

Examples

For matrix `X`

```
X =
      1     5     9
      7    15    22
s = std(X,0,1)
s =
    4.2426    7.0711    9.1924
s = std(X,0,2)
s =
    4.000
    7.5056
```

See Also

`corrcoef`, `cov`, `mean`, `median`, `var`

Purpose

Standard deviation of `timeseries` data

Syntax

```
ts_std = std(ts)
ts_std = std(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_std = std(ts)` returns the standard deviation of the time-series data. When `ts.Data` is a vector, ts_std is the standard deviation of `ts.Data` values. When `ts.Data` is a matrix, ts_std is the standard deviation of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `std` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_std = std(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,1:24,'Name','CountPerSecond')
```

3 Calculate the standard deviation of each data column for this `timeseries` object.

```
std(count_ts)

ans =

   25.3703   41.4057   68.0281
```

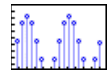
The standard deviation is calculated independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `mean (timeseries)`, `median (timeseries)`, `var (timeseries)`, `timeseries`

Purpose

Plot discrete sequence data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
stem(Y)
stem(X,Y)
stem(...,'fill')
stem(...,LineSpec)
stem(axes_handle,...)
h = stem(...)
hlines = stem('v6',...)
```

Description

A two-dimensional stem plot displays data as lines extending from a baseline along the *x*-axis. A circle (the default) or other marker whose *y*-position represents the data value terminates each stem.

`stem(Y)` plots the data sequence `Y` as stems that extend from equally spaced and automatically generated values along the *x*-axis. When `Y` is a matrix, `stem` plots all elements in a row against the same *x* value.

`stem(X,Y)` plots `X` versus the columns of `Y`. `X` and `Y` must be vectors or matrices of the same size. Additionally, `X` can be a row or a column vector and `Y` a matrix with `length(X)` rows.

`stem(...,'fill')` specifies whether to color the circle at the end of the stem.

`stem(...,LineSpec)` specifies the line style, marker symbol, and color for the stem and top marker (the baseline is not affected). See `LineSpec` for more information.

`stem(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

`h = stem(...)` returns a vector of stemseries object handles in `h`, one handle per column of data in `Y`.

Backward-Compatible Version

`hlines = stem('v6',...)` returns the handles of line objects instead of stemseries objects for compatibility with MATLAB 6.5 and earlier.

`hlines` contains the handles to three `line` graphics objects:

- `hlines(1)` — The marker symbol at the top of each stem
- `hlines(2)` — The stem line
- `hlines(3)` — The baseline handle

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

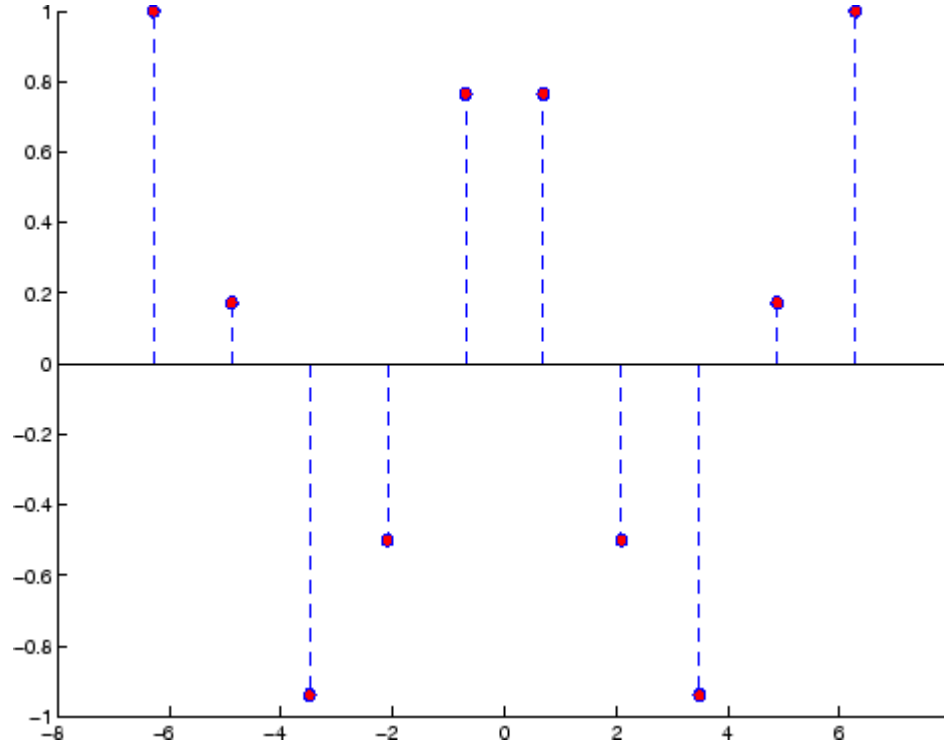
See Plot Objects and Backward Compatibility for more information.

Examples

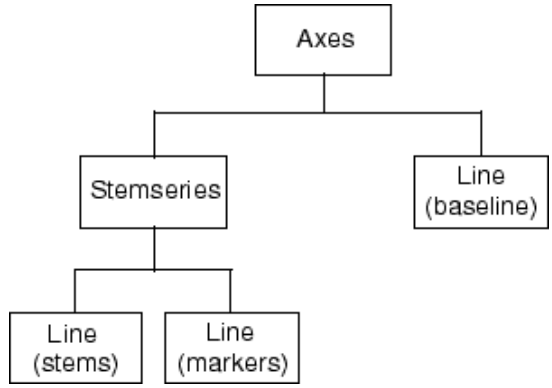
Single Series of Data

This example creates a stem plot representing the cosine of 10 values linearly spaced between 0 and 2π. Note that the line style of the baseline is set by first getting its handle from the stemseries object's `BaseLine` property.

```
t = linspace(-2*pi,2*pi,10);
h = stem(t,cos(t),'fill','--');
set(get(h,'BaseLine'),'LineStyle',':')
set(h,'MarkerFaceColor','red')
```



The following diagram illustrates the parent-child relationship in the previous stem plot. Note that the stemseries object contains two line objects used to draw the stem lines and the end markers. The baseline is a separate line object.

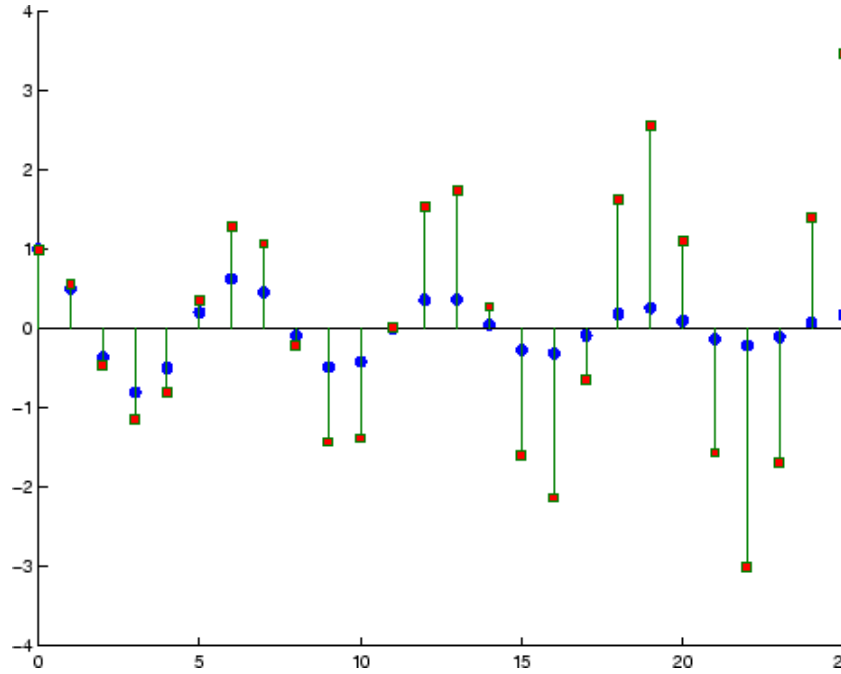


Two Series of Data on One Graph

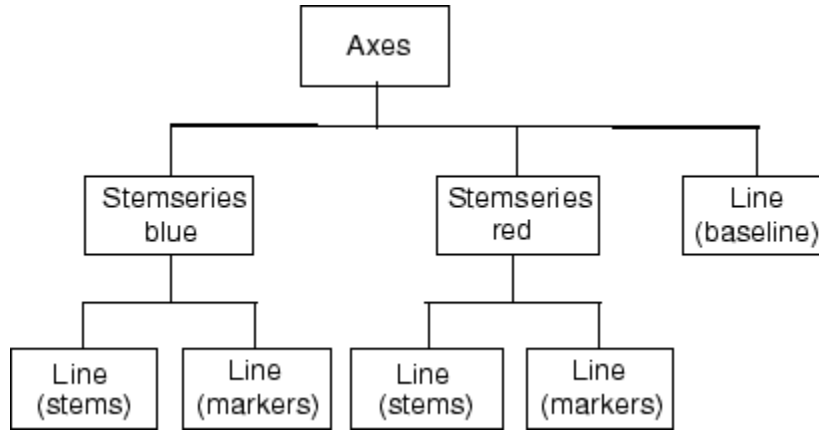
The following example creates a stem plot from a two-column matrix. In this case, the `stem` function creates two stemseries objects, one of each column of data. Both objects' handles are returned in the output argument `h`.

- `h(1)` is the handle to the stemseries object plotting the expression `exp(-.07*x).*cos(x)`.
- `h(2)` is the handle to the stemseries object plotting the expression `exp(.05*x).*cos(x)`.

```
x = 0:25;
y = [exp(-.07*x).*cos(x);exp(.05*x).*cos(x)]';
h = stem(x,y);
set(h(1),'MarkerFaceColor','blue')
set(h(2),'MarkerFaceColor','red','Marker','square')
```



The following diagram illustrates the parent-child relationship in the previous stem plot. Note that each column in the input matrix y results in the creation of a stemseries object, which contains two line objects (one for the stems and one for the markers). The baseline is shared by both stemseries objects.



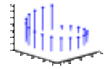
See Also

`bar`, `plot`, `stairs`

`Stemseries` properties for property descriptions

Purpose

Plot 3-D discrete sequence data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3(...,'fill')
stem3(...,LineSpec)
h = stem3(...)
hlines = stem3('v6',...)
```

Description

Three-dimensional stem plots display lines extending from the *x*-*y* plane. A circle (the default) or other marker symbol whose *z*-position represents the data value terminates each stem.

`stem3(Z)` plots the data sequence `Z` as stems that extend from the *x*-*y* plane. *x* and *y* are generated automatically. When `Z` is a row vector, `stem3` plots all elements at equally spaced *x* values against the same *y* value. When `Z` is a column vector, `stem3` plots all elements at equally spaced *y* values against the same *x* value.

`stem3(X,Y,Z)` plots the data sequence `Z` at values specified by `X` and `Y`. `X`, `Y`, and `Z` must all be vectors or matrices of the same size.

`stem3(...,'fill')` specifies whether to color the interior of the circle at the end of the stem.

`stem3(...,LineSpec)` specifies the line style, marker symbol, and color for the stems. See `LineSpec` for more information.

`h = stem3(...)` returns handles to stemseries graphics objects.

Backward-Compatible Version

`hlines = stem3('v6',...)` returns the handles of line objects instead of stemseries objects for compatibility with MATLAB 6.5 and earlier.

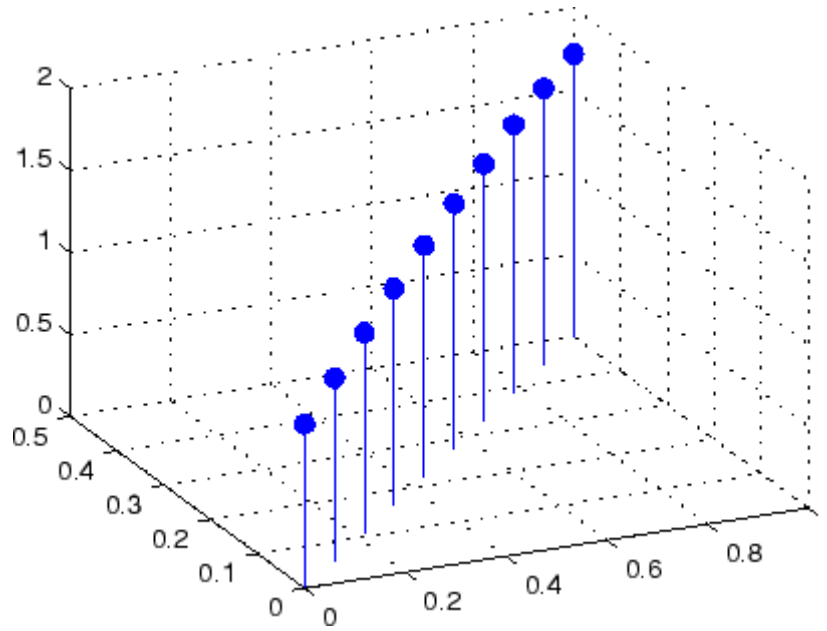
Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Examples

Create a three-dimensional stem plot to visualize a function of two variables.

```
X = linspace(0,1,10);
Y = X./2;
Z = sin(X) + cos(Y);
stem3(X,Y,Z,'fill')
view(-25,30)
```



See Also

`bar`, `plot`, `stairs`, `stem`

"Discrete Data Plots" on page 1-89 for related functions

`Stemseries Properties` for descriptions of properties

Three-Dimensional Stem Plots for more examples

Stemseries Properties

Purpose

Define stemseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

Note that you cannot define default properties for stemseries objects.

See Plot Objects for information on stemseries objects.

Stemseries Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of stemseries objects in legends. The `Annotation` property enables you to specify whether this stemseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the stemseries object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Include the stemseries object in a legend as one entry, but not its children objects
`off`	Do not include the stemseries or its children in a legend (default)
`children`	Include only the children of the stemseries as separate entries in the legend

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BaseLine`
handle of baseline

Handle of the baseline object. This property contains the handle of the line object used as the baseline. You can set the properties of this line using its handle. For example, the following statements create a stem plot, obtain the handle of the baseline from the stemseries object, and then set line properties that make the baseline a dashed, red line.

```
stem_handle = stem(randn(10,1));
baseline_handle = get(stem_handle,'BaseLine');
set(baseline_handle,'LineStyle','--','Color','red')
```

`BaseValue`
y-axis value

Y-axis value where baseline is drawn. You can specify the value along the *y*-axis at which MATLAB draws the baseline.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See "Function Handle Callbacks" for information on how to use function handles to define the callbacks.

`Children`
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to `callback` or `off`, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
`ColorSpec`

Color of stem lines. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

For example, the following statement would produce a stem plot with red lines.

```
h = stem(randn(10,1),'Color','r');
```

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string (default is empty string)

String used by legend for this stemseries object. The `legend` function uses the string defined by the `DisplayName` property to label this stemseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this stemseries object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.

- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

`EraseMode`

```
{normal} | none | xor | background
```

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color

if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.

- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

```
{on} | callback | off
```

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.

- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click th eobject's lines or markers (excluding the baseline, if any) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks.

For example, you might create a stemseries object and set the `Tag` property:

```
t = stem(Y,'Tag','stem1')
```

When you want to access the stemseries object, you can use `findobj` to find the stemseries object's handle. The following statement changes the `MarkerFaceColor` property of the object whose `Tag` is `stem1`.

```
set(findobj('Tag','stem1'),'MarkerFaceColor','red')
```

`Type`
string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For stemseries objects, `Type` is `'hggroup'`. The following statement finds all the hggroup objects in the current axes object.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
array

X-axis location of stems. The stem function draws an individual stem at each *x*-axis location in the `XData` array. `XData` can be either a matrix equal in size to `YData` or a vector equal in length to the number of rows in `YData`. That is, `length(XData) == size(YData,1)`. `XData` does not need to be monotonically increasing.

If you do not specify `XData` (i.e., the input argument `x`), the stem function uses the indices of `YData` to create the stem plot. See the `XDataMode` property for related information.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `x` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

scalar, vector, or matrix

Stem plot data. `YData` contains the data plotted as stems. Each value in `YData` is represented by a marker in the stem plot. If `YData` is a matrix, MATLAB creates a series of stems for each column in the matrix.

The input argument `y` in the `stem` function calling syntax assigns values to `YData`.

`YDataSource`

string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`

vector of coordinates

Z-coordinates. A data defining the stems for 3-D stem graphs. `XData` and `YData` (if specified) must be the same size.

`ZDataSource`

string (MATLAB variable)

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `ZData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Stop timer(s)

Syntax

```
stop(obj)
```

Description

`stop(obj)` stops the timer, represented by the timer object, `obj`. If obj is an array of timer objects, the `stop` function stops them all. Use the `timer` function to create a timer object.

The `stop` function sets the `Running` property of the timer object, `obj`, to `'off'`, halts further `TimerFcn` callbacks, and executes the `StopFcn` callback.

See Also

`timer`, `start`

stopasync

Purpose

Stop asynchronous read and write operations

Syntax

```
stopasync(obj)
```

Arguments

`obj`	A serial port object or an array of serial port objects.

Description

`stopasync(obj)` stops any asynchronous read or write operation that is in progress for `obj`.

Remarks

You can write data asynchronously using the `fprintf` or `fwrite` function. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual`.
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

See Also

Functions

`fprintf`, `fwrite`, `readasync`

Properties

`ReadAsyncMode`, `TransferStatus`

str2double

Purpose

Convert string to double-precision value

Syntax

```
X = str2double('str')
X = str2double(C)
```

Description

`X = str2double('str')` converts the string `str`, which should be an ASCII character representation of a real or complex scalar value, to the MATLAB double-precision representation. The string can contain digits, a comma (thousands separator), a decimal point, a leading `+` or `-` sign, an `e` preceding a power of 10 scale factor, and an `i` for a complex unit.

If `str` does not represent a valid scalar value, `str2double` returns `NaN`.

`X = str2double(C)` converts the strings in the cell array of strings `C` to double precision. The matrix `X` returned will be the same size as `C`.

Examples

Here are some valid `str2double` conversions.

```
str2double('123.45e7')
str2double('123 + 45i')
str2double('3.14159')
str2double('2.7i - 3.14')
str2double({'2.71' '3.1415'})
str2double('1,200.34')
```

See Also

`char`, `hex2num`, `num2str`, `str2num`

Purpose

Construct function handle from function name string

Syntax

```
str2func('str')
```

Description

`str2func('str')` constructs a function handle `fhandle` for the function named in the string `'str'`.

You can create a function handle using either the `@function` syntax or the `str2func` command. You can create an array of function handles from strings by creating the handles individually with `str2func`, and then storing these handles in a cellarray.

Examples

Example 1

To convert the string, `'sin'`, into a handle for that function, type

```
fh = str2func('sin')
fh =
    @sin
```

Example 2

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using `str2func`. The example below passes the variable, `funcname`, to function `makeHandle`, which then creates a function handle. Here is the function M-file:

```
function fh = makeHandle(funcname)
fh = str2func(funcname);
```

This is the code that calls `makdHandle` to construct the function handle:

```
makeHandle('sin')
ans =
    @sin
```

Example 3

To call `str2func` on a cell array of strings, use the `cellfun` function. This returns a cell array of function handles:

```
fh_array = cellfun(@str2func, {'sin' 'cos' 'tan'}, ...
                   'UniformOutput', false);

fh_array{2}(5)
ans =
   0.2837
```

Example 4

In the following example, the `myminbnd` function expects to receive either a function handle or string in the first argument. If you pass a string, `myminbnd` constructs a function handle from it using `str2func`, and then uses that handle in a call to `fminbnd`:

```
function myminbnd(fhandle, lower, upper)
if ischar(fhandle)
   disp 'converting function string to function handle ...'
   fhandle = str2func(fhandle);
end
fminbnd(fhandle, lower, upper)
```

Whether you call `myminbnd` with a function handle or function name string, the function can handle the argument appropriately:

```
myminbnd('humps', 0.3, 1)
converting function string to function handle ...
ans =
    0.6370
```

See Also

`function_handle`, `func2str`, `functions`

Purpose

Form blank-padded character matrix from strings

Syntax

```
S = str2mat(T1, T2, T3, ...)
```

Description

`S = str2mat(T1, T2, T3, ...)` forms the matrix `S` containing the text strings `T1, T2, T3, ...` as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, `Ti`, can itself be a string matrix. This allows the creation of arbitrarily large string matrices. Empty strings are significant.

Note This routine will become obsolete in a future version. Use `char` instead.

Remarks

`str2mat` differs from `strvcat` in that empty strings produce blank rows in the output. In `strvcat`, empty strings are ignored.

Examples

```
x = str2mat('36842', '39751', '38453', '90307');

whos x
  Name      Size         Bytes  Class

  x         4x5             40  char array

x(2,3)

ans =

     7
```

See Also

`char`, `strvcat`

Purpose

Convert string to number

Syntax

```
x = str2num('str')
[x status] = str2num('str')
```

Description

`x = str2num('str')` converts the string `str`, which is an ASCII character representation of a numeric value, to numeric representation. `str2num` also converts string matrices to numeric matrices. If the input string does not represent a valid number or matrix, `str2num(str)` returns the empty matrix in `x`.

The input string can contain

- Digits
- A decimal point
- A leading + or - sign
- A letter `e` or `d` preceding a power of 10 scale factor
- A letter `i` or `j` indicating a complex or imaginary number.

`[x status] = str2num('str')` returns the status of the conversion in logical `status`, where `status` equals logical `1` (`true`) if the conversion succeeds, and logical `0` (`false`) otherwise. If the input string `str` does not represent a valid number or matrix, MATLAB sets `x` to the empty matrix. If the conversions fails, `status` is set to 0.

Space characters can be significant. For instance, `str2num('1+2i')` and `str2num('1 + 2i')` produce `x = 1+2i`, while `str2num('1 +2i')` produces `x = [1 2i]`. You can avoid these problems by using the `str2double` function.

Note `str2num` uses the `eval` function to convert the input argument, so side effects can occur if the string contains calls to functions. Use `str2double` to avoid such side effects, or when the input to `str2num` contains a string that represents a single number.

Examples

`str2num('3.14159e0')` is approximately π.

To convert a string matrix,

```
str2num(['1 2';'3 4'])

ans =

     1     2
     3     4
```

See Also

`num2str`, `hex2num`, `sscanf`, `sparse`, special characters

Purpose

Concatenate strings horizontally

Syntax

```
t = strcat(s1, s2, s3, ...)
```

Description

`t = strcat(s1, s2, s3, ...)` horizontally concatenates corresponding rows of the character arrays `s1`, `s2`, `s3`, etc. All input arrays must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array.

When any of the inputs is a cell array of strings, `strcat` returns a cell array of strings formed by concatenating corresponding elements of `s1`, `s2`, etc. The inputs must all have the same size (or any can be a scalar). Any of the inputs can also be character arrays.

Trailing spaces in character array inputs are ignored and do not appear in the output. This is not true for inputs that are cell arrays of strings. Use the concatenation syntax `[s1 s2 s3 ...]` to preserve trailing spaces.

Remarks

`strcat` and matrix operation are different for strings that contain trailing spaces:

```
a = 'hello '
b = 'goodbye'
strcat(a, b)
ans =
hellogoodbye
[a b]
ans =
hello goodbye
```

Examples

Given two 1-by-2 cell arrays `a` and `b`,

```
a =         b =
    'abcde'     'fghi'          'jkl'     'mn'
```

the command `t = strcat(a,b)` yields

```
t =
    'abcdejkl'    'fghimn'
```

Given the 1-by-1 cell array `c = {`Q'}`, the command `t = strcat(a,b,c)` yields

```
t =
    'abcdejklQ'    'fghimnQ'
```

See Also

`strvcat`, `cat`, `cellstr`

Purpose

Compare strings

Syntax

```
TF = strcmp('str1', 'str2')
TF = strcmp('str', C)
TF = strcmp(C1, C2)
```

Each of these syntaxes apply to both `strcmp` and `strcmpi`. The `strcmp` function is case sensitive in matching strings, while `strcmpi` is not.

Description

Although the following descriptions show only `strcmp`, they apply to `strcmpi` as well. The two functions are the same except that `strcmpi` compares strings without sensitivity to letter case:

`TF = strcmp('str1', 'str2')` compares the strings `str1` and `str2` and returns logical 1 (`true`) if they are identical, and returns logical 0 (`false`) otherwise. `str1` and `str2` can be character arrays of any dimension, but `strcmp` does not return `true` unless the sizes of both arrays are equal, and the contents of the two arrays are the same.

`TF = strcmp('str', C)` compares string `str` to the each element of cell array `C`, where `str` is a character vector (or a 1-by-1 cell array) and `C` is a cell array of strings. The function returns `TF`, a logical array that is the same size as `C` and contains logical 1 (`true`) for those elements of `C` that are a match, and logical 0 (`false`) for those elements that are not. The order of the first two input arguments is not important.

`TF = strcmp(C1, C2)` compares each element of `C1` to the same element in `C2`, where `C1` and `C2` are equal-size cell arrays of strings. Input `C1` or `C2` can also be a character array with the right number of rows. The function returns `TF`, a logical array that is the same size as `C1` and `C2`, and contains logical 1 (`true`) for those elements of `C1` and `C2` that are a match, and logical 0 (`false`) for those elements that are not.

Remarks

These functions are intended for comparison of character data. When used to compare numeric data, they return logical `0`.

Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

The value returned by `strcmp` and `strcmpi` is not the same as the C language convention.

`strcmp` and `strcmpi` support international character sets.

Examples

Example 1

Perform a simple comparison of two strings:

```
strcmp('Yes', 'No')
ans =
     0
strcmp('Yes', 'Yes')
ans =
     1
```

Example 2

Create 3 cell arrays of strings:

```
A = {'MATLAB','SIMULINK';                          ...
     'Toolboxes', 'The MathWorks'};

B = {'Handle Graphics', 'Real Time Workshop';   ...
     'Toolboxes', 'The MathWorks'};

C = {'handle graphics', 'Signal Processing';   ...
     '  Toolboxes', 'The MATHWORKS'};
```

Compare cell arrays `A` and `B` with sensitivity to case:

```
strcmp(A, B)
ans =
     0     0
     1     1
```

Compare cell arrays `B` and `C` without sensitivity to case. Note that `'Toolboxes'` doesn't match because of the leading space characters in `C{2,1}` that do not appear in `B{2,1}`:

```
strcmpi(B, C)
ans =
     1     0
     0     1
```

Example 3

Compare a string vector to a cell array of strings, a string vector to a string array, and a string array to a cell array of strings.

Start by creating a cell array of strings, a string array containing the same strings (plus padding space characters), and a string vector containing one of the strings (plus padding).

```
cellArr = {'It was the best of times'; ...
           'it was the worst of times'; ...
           'it was the age of wisdom'; ...
           'it was the age of foolishness'};

strArr = char(cellArr);

strVec = strArr(3,:)
strVec =
   it was the age of wisdom
```

Remove the space padding from the string vector and compare it to the cell array. MATLAB compares the string with each row of the cell array, finding a match on the third row:

```
strcmp(deblank(strVec), cellArr)
ans =
     0
     0
     1
     0
```

Compare the string vector with the string array. Unlike the case above, MATLAB does not compare the string vector with each row of the string

array. It compares the entire contents of one against the entire contents of the other:

```
strcmp(strVec, strArr)
ans =
     0
```

Lastly, compare each row of the four-row string array against the same rows of the cell array. MATLAB finds them all to be equivalent. Note that in this case you do not have to remove the space padding from the string array:

```
strcmp(strArr, cellArr)
ans =
     1
     1
     1
     1
```

See Also

`strncmp`, `strncmpi`, `strmatch`, `strfind`, `findstr`, `regexp`, `regexpi`, `regexprep`, `regexptranslate`

stream2

Purpose

Compute 2-D streamline data

Syntax

```
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)
```

Description

`XY = stream2(x,y,u,v,startx,starty)` computes streamlines from vector data `u` and `v`. The arrays `x` and `y` define the coordinates for `u` and `v` and must be monotonic and 2-D plaid (such as the data produced by `meshgrid`). `startx` and `starty` define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value `XY` contains a cell array of vertex arrays.

`XY = stream2(u,v,startx,starty)` assumes the arrays `x` and `y` are defined as `[x,y] = meshgrid(1:n,1:m)` where `[m,n] = size(u)`.

`XY = stream2(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify a value, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream2`.

Examples

This example draws 2-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:,:,5),y(:,:,5),u(:,:,5),v(:,:,5),sx,sy));
```

See Also

coneplot, stream3, streamline

"Volume Visualization" on page 1-102 for related functions

Specifying Starting Points for Stream Plots for related information

Purpose

Compute 3-D streamline data

Syntax

```
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)
XYZ = stream3(U,V,W,startx,starty,startz)
XYZ = stream3(...,options)
```

Description

`XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)` computes streamlines from vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). `startx`, `starty`, and `startz` define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value `XYZ` contains a cell array of vertex arrays.

`XYZ = stream3(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)` where `[M,N,P] = size(U)`.

`XYZ = stream3(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream3`.

Examples

This example draws 3-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamline(stream3(x,y,z,u,v,w,sx,sy,sz))
view(3)
```

See Also

`coneplot`, `stream2`, `streamline`

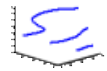
"Volume Visualization" on page 1-102 for related functions

Specifying Starting Points for Stream Plots for related information

streamline

Purpose

Plot streamlines from 2-D or 3-D vector data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamline(X,Y,Z,U,V,W,startx,starty,startz)
streamline(U,V,W,startx,starty,startz)
streamline(XYZ)
streamline(X,Y,U,V,startx,starty)
streamline(U,V,startx,starty)
streamline(XY)
streamline(...,options)
streamline(axes_handle,...)
h = streamline(...)
```

Description

`streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws streamlines from 3-D vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). `startx`, `starty`, `startz` define the starting positions of the streamlines. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

`streamline(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)`, where `[M,N,P] = size(U)`.

`streamline(XYZ)` assumes `XYZ` is a precomputed cell array of vertex arrays (as produced by `stream3`).

`streamline(X,Y,U,V,startx,starty)` draws streamlines from 2-D vector data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (such as the data produced by `meshgrid`). `startx` and `starty` define the starting positions of the streamlines. The output argument `h` contains a vector of line handles, one handle for each streamline.

`streamline(U,V,startx,starty)` assumes the arrays `X` and `Y` are defined as `[X,Y] = meshgrid(1:N,1:M)`, where `[M,N] = size(U)`.

`streamline(XY)` assumes `XY` is a precomputed cell array of vertex arrays (as produced by `stream2`).

`streamline(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 1000

`streamline(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of the into current axes object (`gca`).

`h = streamline(...)` returns a vector of line handles, one handle for each streamline.

Examples

This example draws streamlines from data representing air currents over a region of North America. Loading the `wind` data set creates the variables `x`, `y`, `z`, `u`, `v`, and `w` in the MATLAB workspace.

The plane of streamlines indicates the flow of air from the west to the east (the *x*-direction) beginning at `x = 80` (which is close to the minimum value of the `x` coordinates). The *y*- and *z*-coordinate starting points are multivalued and approximately span the range of these coordinates. `meshgrid` generates the starting positions of the streamlines.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
h = streamline(x,y,z,u,v,w,sx,sy,sz);
set(h,'Color','red')
view(3)
```

See Also

`coneplot`, `stream2`, `stream3`, `streamparticles`

"Volume Visualization" on page 1-102 for related functions

Specifying Starting Points for Stream Plots for related information

Stream Line Plots of Vector Data for another example

Purpose

Plot stream particles

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)
streamparticles(line_handle,...)
h = streamparticles(...)
```

Description

`streamparticles(vertices)` draws stream particles of a vector field. Stream particles are usually represented by markers and can show the position and velocity of a streamline. `vertices` is a cell array of 2-D or 3-D vertices (as if produced by `stream2` or `stream3`).

`streamparticles(vertices,n)` uses `n` to determine how many stream particles to draw. The `ParticleAlignment` property controls how `n` is interpreted.

- If `ParticleAlignment` is set to `off` (the default) and `n` is greater than `1`, approximately `n` particles are drawn evenly spaced over the streamline vertices.

  If `n` is less than or equal to `1`, `n` is interpreted as a fraction of the original stream vertices; for example, if `n` is `0.2`, approximately 20% of the vertices are used.

  `n` determines the upper bound for the number of particles drawn. The actual number of particles can deviate from `n` by as much as a factor of 2.

- If `ParticleAlignment` is `on`, `n` determines the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value. The default value is `n = 1`.

`streamparticles(...,'`*`PropertyName`*`',PropertyValue,...)` controls the stream particles using named properties and specified values. Any unspecified properties have default values. MATLAB ignores the case of property names.

Stream Particle Properties

`Animate` — Stream particle motion [nonnegative integer]

The number of times to animate the stream particles. The default is `0`, which does not animate. `Inf` animates until you enter **Ctrl+C**.

`FrameRate` — Animation frames per second [nonnegative integer]

This property specifies the number of frames per second for the animation. `Inf`, the default, draws the animation as fast as possible. Note that the speed of the animation might be limited by the speed of the computer. In such cases, the value of `FrameRate` cannot necessarily be achieved.

`ParticleAlignment` — Align particles with streamlines [`on` | `{off}`]

Set this property to `on` to draw particles at the beginning of each streamline. This property controls how `streamparticles` interprets the argument `n` (number of stream particles).

Stream particles are line objects. In addition to stream particle properties, you can specify any `line` object property, such as `Marker` and `EraseMode`. `streamparticles` sets the following line properties when called.

Line Property	Value Set by streamparticles
`EraseMode`	`xor`
`LineStyle`	`none`
`Marker`	`o`

Line Property	Value Set by streamparticles
MarkerEdgeColor	none
MarkerFaceColor	red

You can override any of these properties by specifying a property name and value as arguments to `streamparticles`. For example, this statement uses RGB values to set the `MarkerFaceColor` to medium gray:

```
streamparticles(vertices,'MarkerFaceColor',[.5 .5 .5])
```

`streamparticles(line_handle,...)` uses the line object identified by `line_handle` to draw the stream particles.

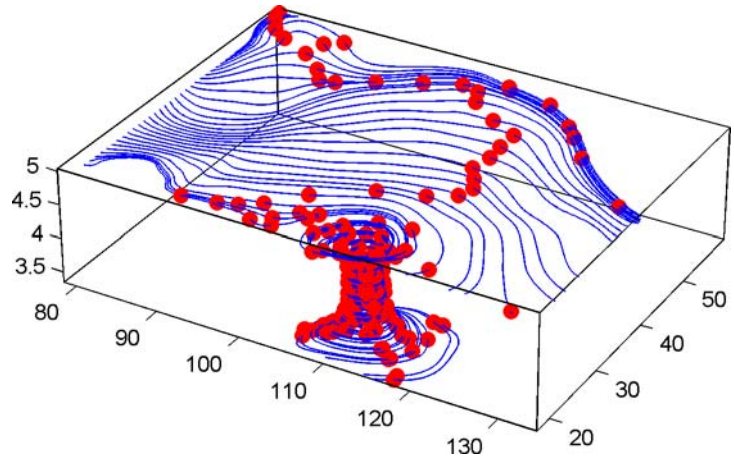
`h = streamparticles(...)` returns a vector of handles to the line objects it creates.

Examples

This example combines streamlines with stream particle animation. The `interpstreamspeed` function determines the vertices along the streamlines where stream particles will be drawn during the animation, thereby controlling the speed of the animation. Setting the axes `DrawMode` property to `fast` provides faster rendering.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = streamline(verts);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.025);
axis tight; view(30,30); daspect([1 1 .125])
camproj perspective; camva(8)
set(gca,'DrawMode','fast')
box on
streamparticles(iverts,35,'animate',10,'ParticleAlignment','on')
```

The following picture is a static view of the animation.



This example uses the streamlines in the `z = 5` plane to animate the flow along these lines with `streamparticles`.

```
load wind
daspect([1 1 1]); view(2)
[verts averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
sl = streamline([verts averts]);
axis tight off;
set(sl,'Visible','off')
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.05);
set(gca,'DrawMode','fast','Position',[0 0 1 1],'ZLim',[4.9 5.1])
set(gcf,'Color','black')
streamparticles(iverts, 200, ...
    'Animate',100,'FrameRate',40, ...
    'MarkerSize',10,'MarkerFaceColor','yellow')
```

See Also

`interpstreamspeed`, `stream3`, `streamline`

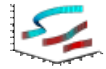
"Volume Visualization" on page 1-102 for related functions

Creating Stream Particle Animations for more details

Specifying Starting Points for Stream Plots for related information

Purpose

3-D stream ribbon plot from vector volume data



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamribbon(X,Y,Z,U,V,W,startx,starty,startz)
streamribbon(U,V,W,startx,starty,startz)
streamribbon(vertices,X,Y,Z,cav,speed)
streamribbon(vertices,cav,speed)
streamribbon(vertices,twistangle)
streamribbon(...,width)
streamribbon(axes_handle,...)
h = streamribbon(...)
```

Description

`streamribbon(X,Y,Z,U,V,W,startx,starty,startz)` draws stream ribbons from vector volume data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `startx`, `starty`, and `startz` define the starting positions of the stream ribbons at the center of the ribbons. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

Generally, you should set the `DataAspectRatio` (`daspect`) before calling `streamribbon`.

`streamribbon(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamribbon(vertices,X,Y,Z,cav,speed)` assumes precomputed streamline vertices, curl angular velocity, and flow speed. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, `cav`, and `speed` are 3-D arrays.

`streamribbon(vertices,cav,speed)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(cav)`.

`streamribbon(vertices,twistangle)` uses the cell array of vectors `twistangle` for the twist of the ribbons (in radians). The size of each corresponding element of `vertices` and `twistangle` must be equal.

`streamribbon(...,width)` sets the width of the ribbons to `width`.

`streamribbon(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

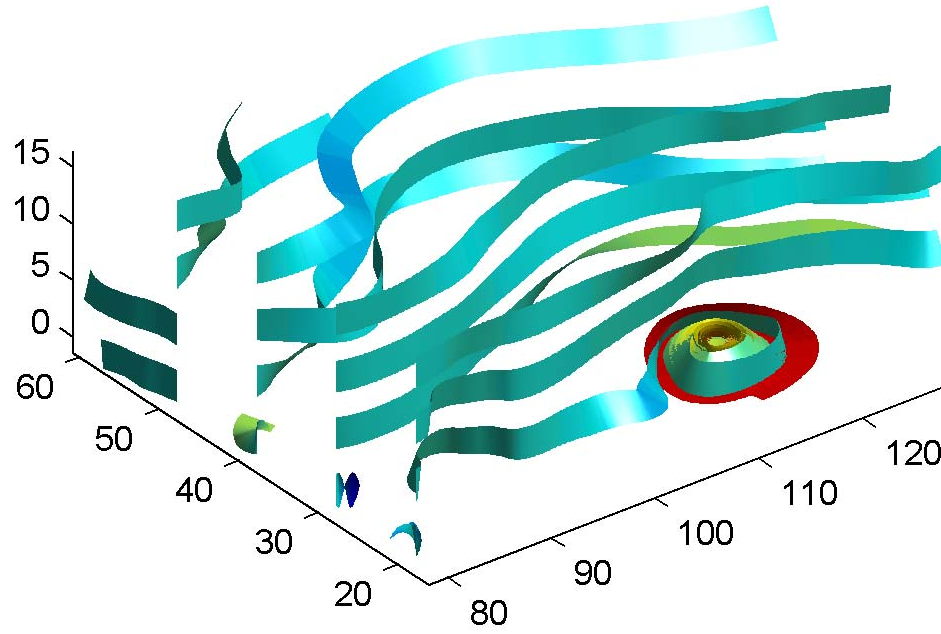
`h = streamribbon(...)` returns a vector of handles (one per start point) to `surface` objects.

Examples

This example uses stream ribbons to indicate the flow in the `wind` data set. Inputs include the coordinates, vector field components, and starting location for the stream ribbons.

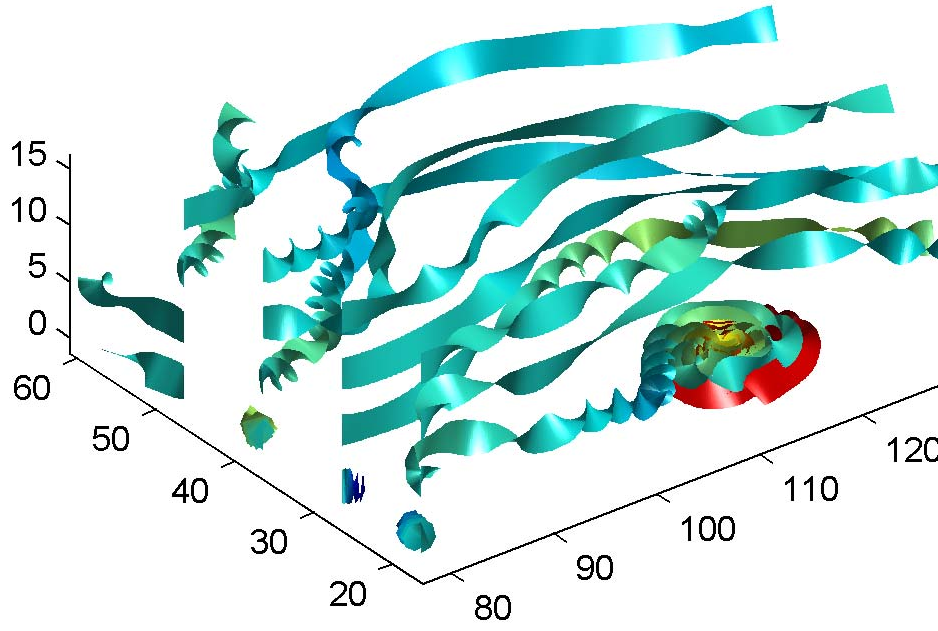
```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
streamribbon(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
axis tight
shading interp;
view(3);
```

```
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`), curl average velocity (`curl`), and speed $\sqrt{u^2 + v^2 + w^2}$. Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to the previous example.

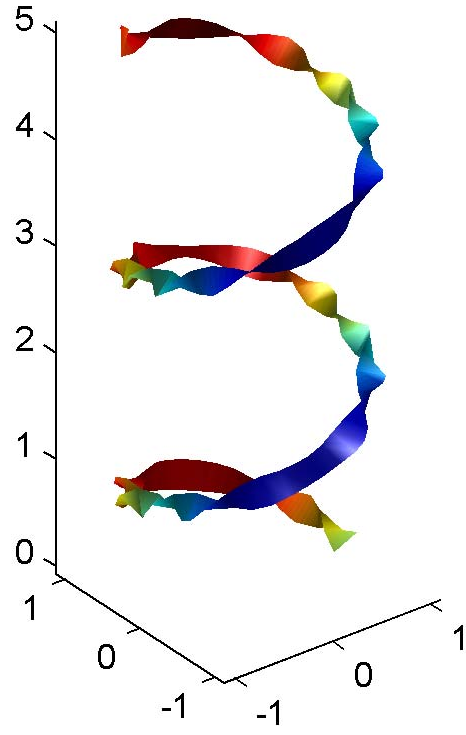
```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*.1;
streamribbon(verts,x,y,z,cav,spd);
%-----Define viewing and lighting
axis tight
shading interp
view(3)
camlight; lighting gouraud
```



This example specifies a twist angle for the stream ribbon.

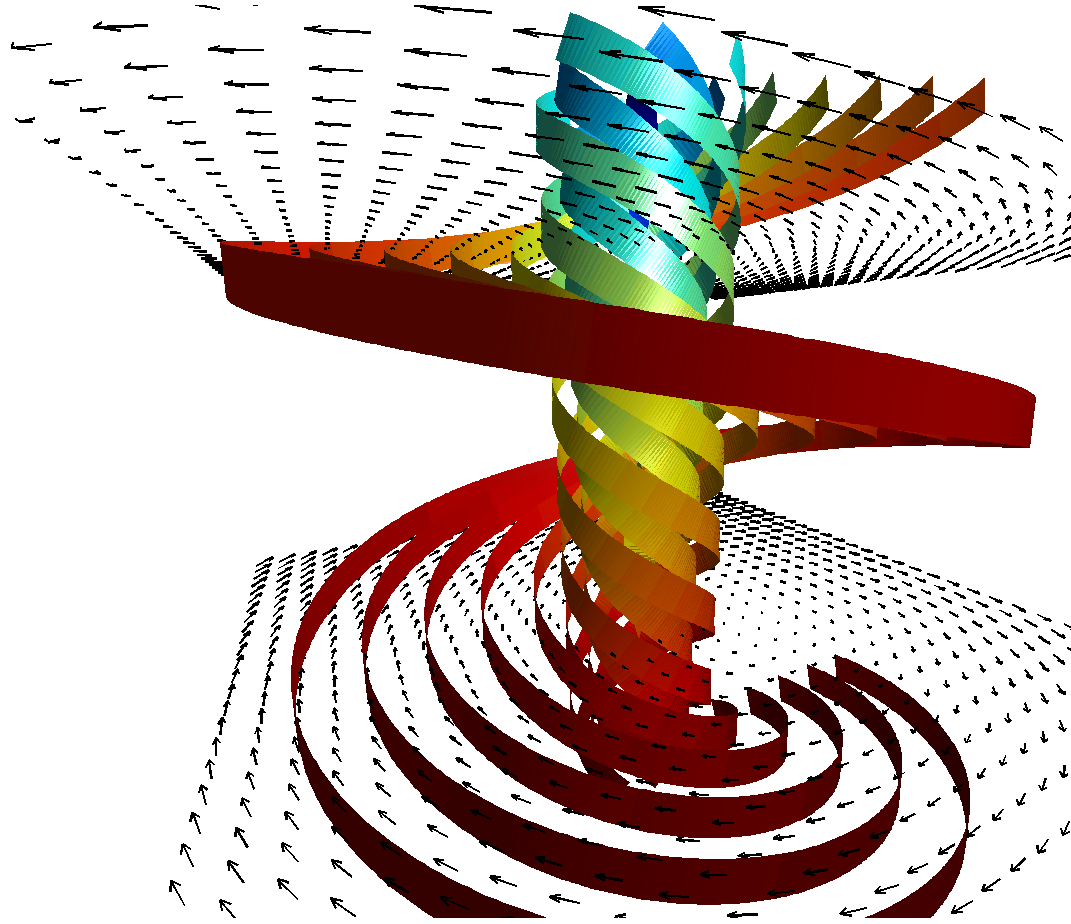
```
t = 0:.15:15;
verts = {[cos(t)' sin(t)' (t/3)']};
twistangle = {cos(t)'};
daspect([1 1 1])
streamribbon(verts,twistangle);
%-----Define viewing and lighting
```

```
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```



This example combines cone plots (`coneplot`) and stream ribbon plots in one graph.

```
%-----Define 3-D arrays x, y, z, u, v, w
xmin = -7; xmax = 7;
ymin = -7; ymax = 7;
zmin = -7; zmax = 7;
x = linspace(xmin,xmax,30);
y = linspace(ymin,ymax,20);
z = linspace(zmin,zmax,20);
[x y z] = meshgrid(x,y,z);
u = y; v = -x; w = 0*x+1;
daspect([1 1 1]);
[cx cy cz] = meshgrid(linspace(xmin,xmax,30),...
 linspace(ymin,ymax,30),[-3 4]);
h = coneplot(x,y,z,u,v,w,cx,cy,cz,'quiver');
set(h,'color','k');
%-----Plot two sets of streamribbons
[sx sy sz] = meshgrid([-1 0 1],[-1 0 1],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
[sx sy sz] = meshgrid([1:6],[0],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
shading interp
view(-30,10) ; axis off tight
camproj perspective; camva(66); camlookat;
camdolly(0,0,.5,'fixtarget')
camlight
```



See Also

`curl`, `streamtube`, `streamline`, `stream3`

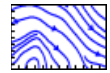
"Volume Visualization" on page 1-102 for related functions

Displaying Curl with Stream Ribbons for another example

Specifying Starting Points for Stream Plots for related information

Purpose

Plot streamlines in slice planes



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamslice(X,Y,Z,U,V,W,startx,starty,startz)
streamslice(U,V,W,startx,starty,startz)
streamslice(X,Y,U,V)
streamslice(U,V)
streamslice(...,density)
streamslice(...,'arrowsmode')
streamslice(...,'method')
streamslice(axes_handle,...)
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)
```

Description

`streamslice(X,Y,Z,U,V,W,startx,starty,startz)` draws well-spaced streamlines (with direction arrows) from vector data `U`, `V`, `W` in axis aligned *x*-, *y*-, *z*-planes starting at the points in the vectors `startx`, `starty`, `startz`. (The section Specifying Starting Points for Stream Plots provides more information on defining starting points.) The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `U`, `V`, `W` must be m-by-n-by-p volume arrays.

Do not assume that the flow is parallel to the slice plane. For example, in a stream slice at a constant `z`, the `z` component of the vector field `W` is ignored when you are calculating the streamlines for that plane.

Stream slices are useful for determining where to start streamlines, stream tubes, and stream ribbons. It is good practice is to set the axes `DataAspectRatio` to `[1 1 1]` when using `streamslice`.

`streamslice(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(X,Y,U,V)` draws well-spaced streamlines (with direction arrows) from vector volume data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`streamslice(U,V)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(...,density)` modifies the automatic spacing of the streamlines. `density` must be greater than `0`. The default value is `1`; higher values produce more streamlines on each plane. For example, `2` produces approximately twice as many streamlines, while `0.5` produces approximately half as many.

`streamslice(...,'arrowsmode')` determines if direction arrows are present or not. `arrowmode` can be

- `arrows` — Draw direction arrows on the streamlines (default).
- `noarrows` — Do not draw direction arrows.

`streamslice(...,'method')` specifies the interpolation method to use. `method` can be

- `linear` — Linear interpolation (default)

- `cubic` — Cubic interpolation
- `nearest` — Nearest-neighbor interpolation

See `interp3` for more information on interpolation methods.

`streamslice(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

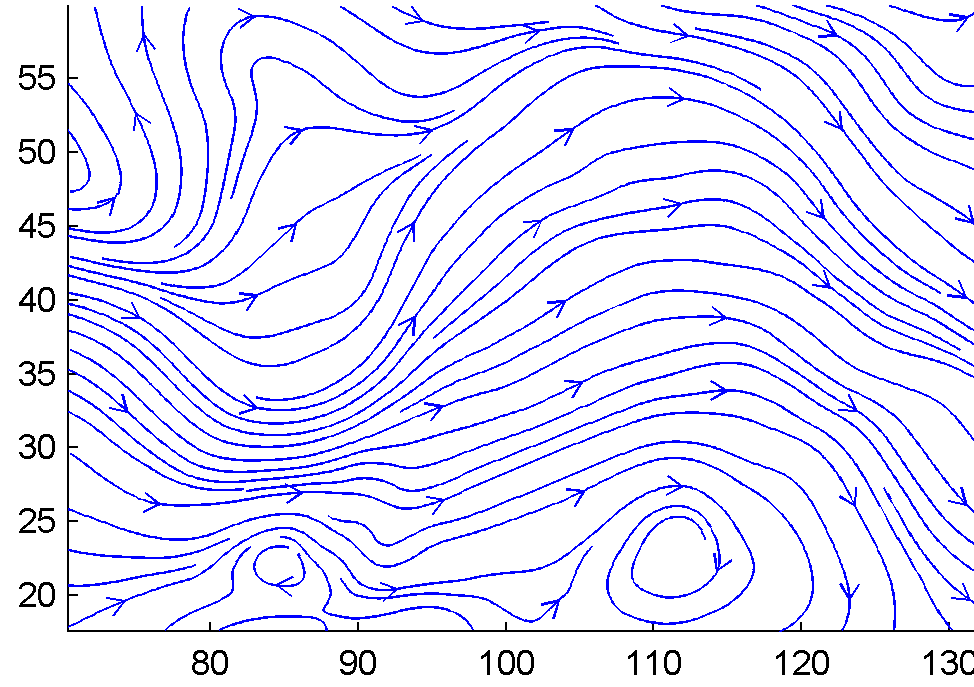
`h = streamslice(...)` returns a vector of handles to the line objects created.

`[vertices arrowvertices] = streamslice(...)` returns two cell arrays of vertices for drawing the streamlines and the arrows. You can pass these values to any of the streamline drawing functions (`streamline`, `streamribbon`, `streamtube`).

Examples

This example creates a stream slice in the `wind` data set at `z = 5`.

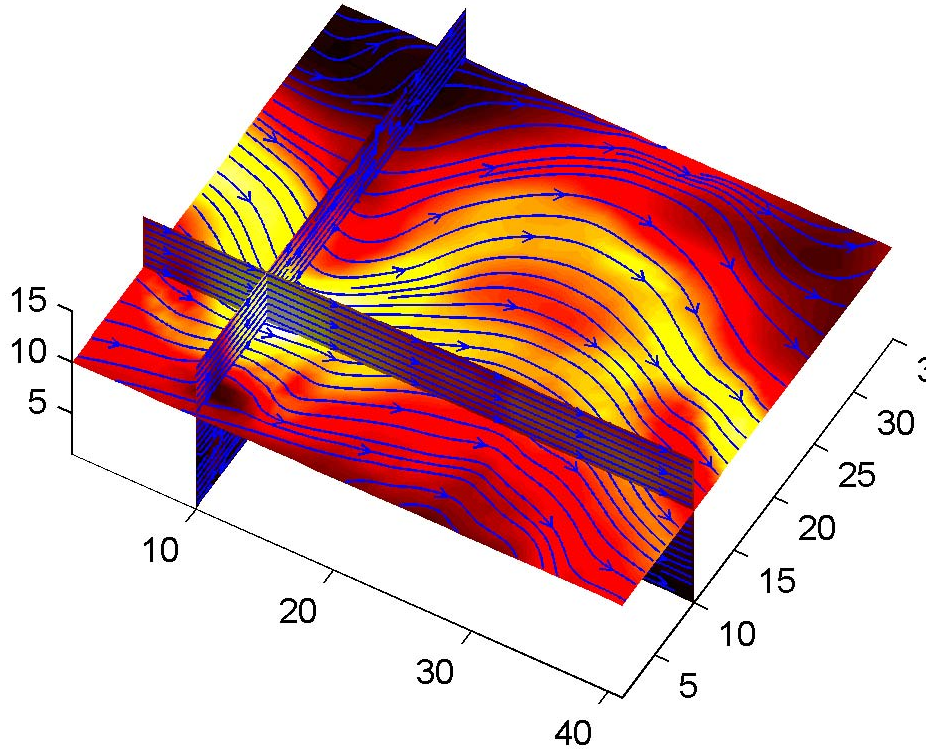
```
load wind
daspect([1 1 1])
streamslice(x,y,z,u,v,w,[],[],[5])
axis tight
```



This example uses `streamslice` to calculate vertex data for the streamlines and the direction arrows. This data is then used by `streamline` to plot the lines and arrows. Slice planes illustrating with color the wind speed $\sqrt{u^2 + v^2 + w^2}$ are drawn by `slice` in the same planes.

```
load wind
```

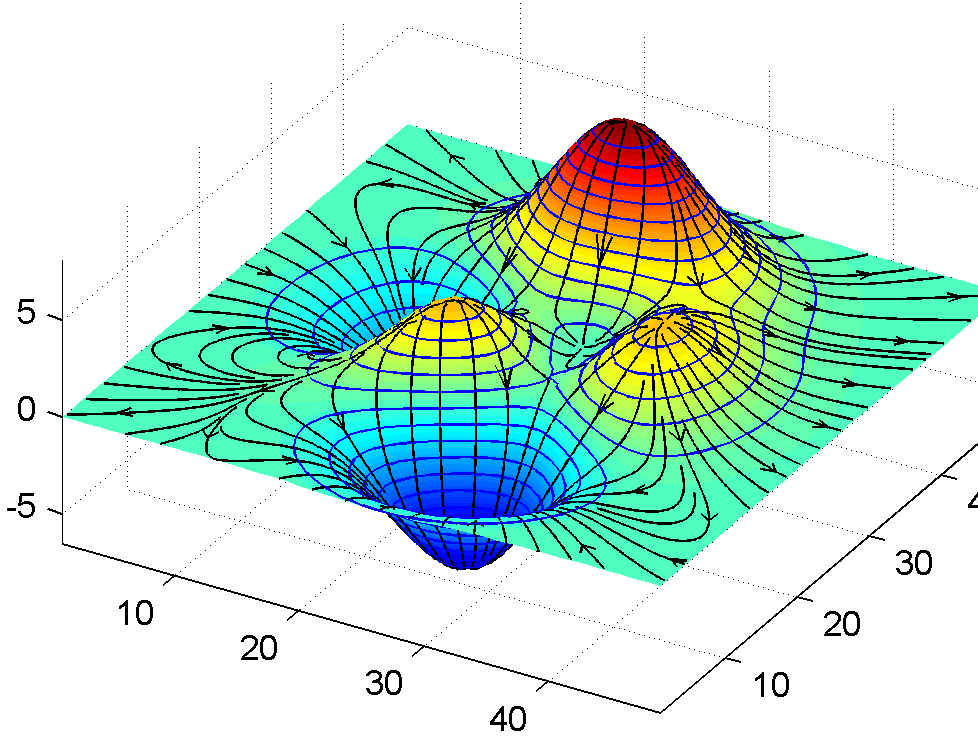
```
daspect([1 1 1])
[verts averts] = streamslice(u,v,w,10,10,10);
streamline([verts averts])
spd = sqrt(u.^2 + v.^2 + w.^2);
hold on;
slice(spd,10,10,10);
colormap(hot)
shading interp
view(30,50); axis(volumebounds(spd));
camlight; material([.5 1 0])
```



This example superimposes contour lines on a surface and then uses `streamslice` to draw lines that indicate the gradient of the surface. `interp2` is used to find the points for the lines that lie on the surface.

```
z = peaks;
surf(z)
shading interp
hold on
```

```
[c ch] = contour3(z,20); set(ch,'edgecolor','b')
[u v] = gradient(z);
h = streamslice(-u,-v);
set(h,'color','k')
for i=1:length(h);
 zi = interp2(z,get(h(i),'xdata'),get(h(i),'ydata'));
 set(h(i),'zdata',zi);
end
view(30,50); axis tight
```



See Also

`contourslice`, `slice`, `streamline`, `volumebounds`

"Volume Visualization" on page 1-102 for related functions

Specifying Starting Points for Stream Plots for related information

Purpose

Create 3-D stream tube plot

GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
streamtube(axes_handle,...)
h = streamtube(...z)
```

Description

`streamtube(X,Y,Z,U,V,W,startx,starty,startz)` draws stream tubes from vector volume data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `startx`, `starty`, and `startz` define the starting positions of the streamlines at the center of the tubes. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The width of the tubes is proportional to the normalized divergence of the vector field.

Generally, you should set the `DataAspectRatio` (`daspect`) before calling `streamtube`.

`streamtube(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamtube(vertices,X,Y,Z,divergence)` assumes precomputed streamline vertices and divergence. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, and `divergence` are 3-D arrays.

`streamtube(vertices,divergence)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(divergence)`.

`streamtube(vertices,width)` specifies the width of the tubes in the cell array of vectors, `width`. The size of each corresponding element of `vertices` and `width` must be equal. `width` can also be a scalar, specifying a single value for the width of all stream tubes.

`streamtube(vertices)` selects the width automatically.

`streamtube(...,[scale n])` scales the width of the tubes by `scale`. The default is `scale = 1`. When the stream tubes are created, using start points or divergence, specifying `scale = 0` suppresses automatic scaling. `n` is the number of points along the circumference of the tube. The default is `n = 20`.

`streamtube(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

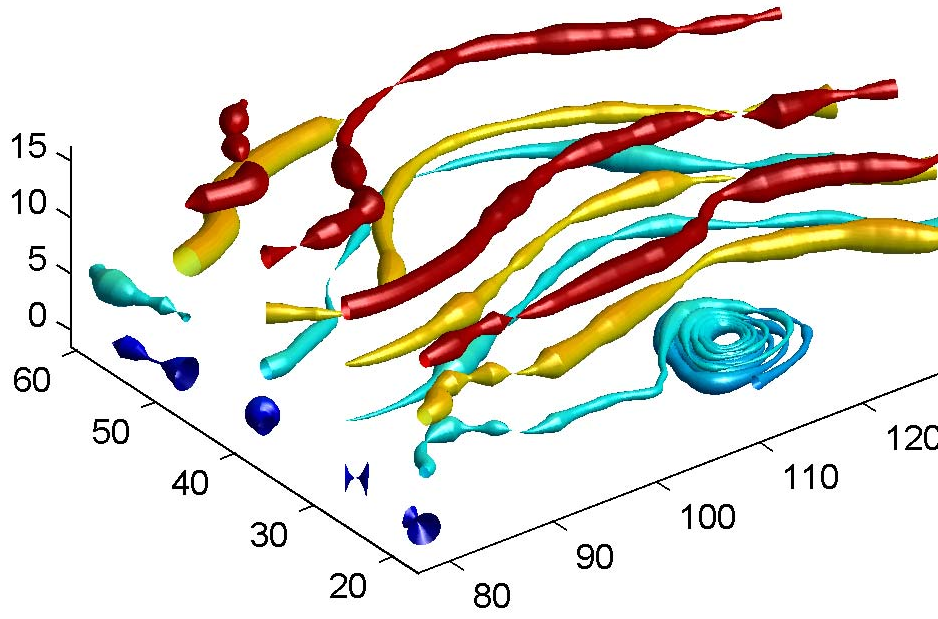
`h = streamtube(...z)` returns a vector of handles (one per start point) to `surface` objects used to draw the stream tubes.

Examples

This example uses stream tubes to indicate the flow in the `wind` data set. Inputs include the coordinates, vector field components, and starting location for the stream tubes.

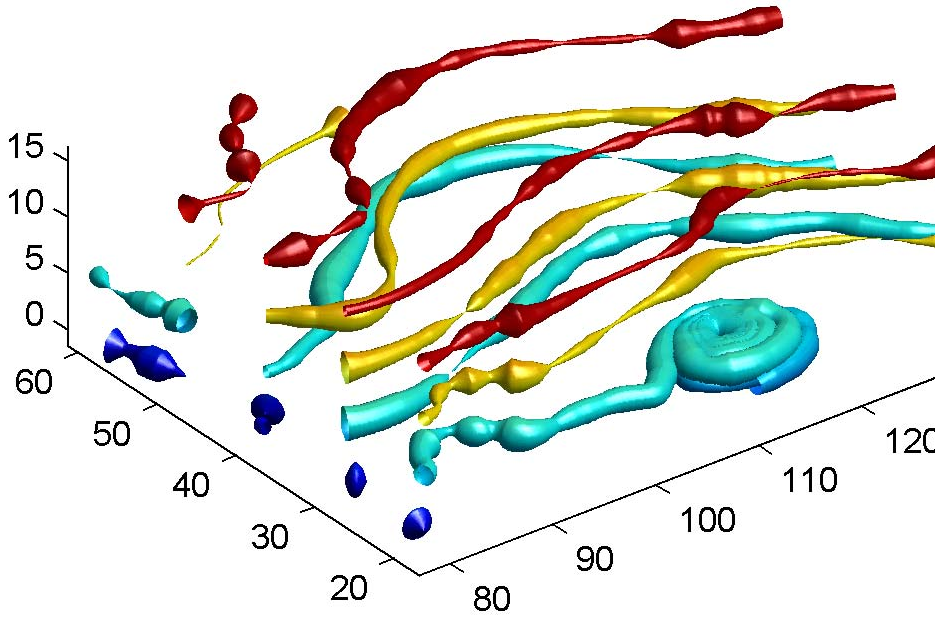
```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
```

```
daspect([1 1 1])
streamtube(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
view(3)
axis tight
shading interp;
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`) and divergence (`divergence`).

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
streamtube(verts,x,y,z,-div);
%-----Define viewing and lighting
view(3)
axis tight
shading interp
camlight; lighting gouraud
```



See Also

`divergence`, `streamribbon`, `streamline`, `stream3`

"Volume Visualization" on page 1-102 for related functions

Displaying Divergence with Stream Tubes for another example

Specifying Starting Points for Stream Plots for related information

strfind

Purpose

Find one string within another

Syntax

```
k = strfind(str, pattern)
k = strfind(cellstr, pattern)
```

Description

`k = strfind(str, pattern)` searches the string `str` for occurrences of a shorter string, `pattern`, and returns the starting index of each such occurrence in the double array `k`. If `pattern` is not found in `str`, or if `pattern` is longer than `str`, then `strfind` returns the empty array `[]`.

`k = strfind(cellstr, pattern)` searches each string in cell array of strings `cellstr` for occurrences of a shorter string, `pattern`, and returns the starting index of each such occurrence in cell array `k`. If `pattern` is not found in a string or if `pattern` is longer then all strings in the cell array, then `strfind` returns the empty array `[]`, for that string in the cell array.

The search performed by `strfind` is case sensitive. Any leading and trailing blanks in `pattern` or in the strings being searched are explicitly included in the comparison.

Examples

Use `strfind` to find a two-letter pattern in string `S`:

```
S = 'Find the starting indices of the pattern string';
strfind(S, 'in')
ans =
     2    15    19    45

strfind(S, 'In')
ans =
     []

strfind(S, ' ')
ans =
     5     9    18    26    29    33    41
```

Use `strfind` on a cell array of strings:

```
cstr = {'How much wood would a woodchuck chuck';
        'if a woodchuck could chuck wood?'};

idx = strfind(cstr, 'wood');

idx{:,:}
ans =
    10    23
ans =
     6    28
```

This means that `'wood'` occurs at indices 10 and 23 in the first string and at indices 6 and 28 in the second.

See Also

`findstr`, `strmatch`, `strtok`, `strcmp`, `strncmp`, `strcmpi`, `strncmpi`, `regexp`, `regexpi`, `regexprep`

Purpose

MATLAB string handling

Syntax

```
S = 'Any Characters'
S = [S1 S2 ...]
S = strcat(S1, S2, ...)
```

Description

`S = 'Any Characters'` creates a character array, or string. The string is actually a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The actual characters displayed depend on the character encoding scheme for a given font. The length of `S` is the number of characters. A quotation within the string is indicated by two quotes.

`S = [S1 S2 ...]` concatenates character arrays `S1`, `S2`, etc. into a new character array, `S`.

`S = strcat(S1, S2, ...)` concatenates `S1`, `S2`, etc., which can be character arrays or "Cell Arrays of Strings". When the inputs are all character arrays, the output is also a character array. When any of the inputs is a cell array of strings, `strcat` returns a cell array of strings.

Trailing spaces in `strcat` character array inputs are ignored and do not appear in the output. This is not true for `strcat` inputs that are cell arrays of strings. Use the `S = [S1 S2 ...]` concatenation syntax, shown above, to preserve trailing spaces.

`S = char(X)` can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.

`X = double(S)` converts the string to its equivalent double-precision numeric codes.

A collection of strings can be created in either of the following two ways:

- As the rows of a character array via `strvcat`
- As a cell array of strings via the curly braces

You can convert between character array and cell array of strings using `char` and `cellstr`. Most string functions support both types.

`ischar(S)` tells if `S` is a string variable. `iscellstr(S)` tells if `S` is a cell array of strings.

Examples

Create a simple string that includes a single quote.

```
msg = 'You''re right!'

msg =
You're right!
```

Create the string `name` using two methods of concatenation.

```
name = ['Thomas' ' R. ' 'Lee']
name = strcat('Thomas',' R.',' Lee')
```

Create a vertical array of strings.

```
C = strvcat('Hello','Yes','No','Goodbye')

C =
Hello
Yes
No
Goodbye
```

Create a cell array of strings.

```
S = {'Hello' 'Yes' 'No' 'Goodbye'}

S =
    'Hello'    'Yes'    'No'    'Goodbye'
```

See Also

`char`, `isstrprop`, `cellstr`, `ischar`, `isletter`, `isspace`, `iscellstr`, `strvcat`, `sprintf`, `sscanf`, `text`, `input`

strjust

Purpose

Justify character array

Syntax

```
T = strjust(S)
T = strjust(S, 'right')
T = strjust(S, 'left')
T = strjust(S, 'center')
```

Description

`T = strjust(S)` or `T = strjust(S, 'right')` returns a right-justified version of the character array `S`.

`T = strjust(S, 'left')` returns a left-justified version of `S`.

`T = strjust(S, 'center')` returns a center-justified version of `S`.

See Also

`deblank`, `strtrim`

Purpose

Find possible matches for string

Syntax

```
x = strmatch(str, strarray)
x = strmatch(str, strarray, 'exact')
```

Description

`x = strmatch(str, strarray)` looks through the rows of the character array or cell array of strings `strarray` to find strings that begin with the text contained in `str`, and returns the matching row indices. Any trailing space characters in `str` or `strarray` are ignored when matching. `strmatch` is fastest when `strarray` is a character array.

`x = strmatch(str, strarray, 'exact')` compares `str` with each row of `strarray`, looking for an exact match of the entire strings. Any trailing space characters in `str` or `strarray` are ignored when matching.

Examples

The statement

```
x = strmatch('max', strvcat('max', 'minimax', 'maximum'))
```

returns `x = [1; 3]` since rows 1 and 3 begin with `'max'`. The statement

```
x = strmatch('max', strvcat('max', 'minimax', 'maximum'),'exact')
```

returns `x = 1`, since only row 1 matches `'max'` exactly.

See Also

`strcmp`, `strcmpi`, `strncmp`, `strncmpi`, `strfind`, `findstr`, `strvcat`, `regexp`, `regexpi`, `regexprep`

Purpose

Compare first n characters of strings

Syntax

```
TF = strncmp('str1', 'str2', n)
TF = strncmp('str', C, n)
TF = strncmp(C1, C2, n)
```

Each of these syntaxes apply to both `strncmp` and `strncmpi`. The `strncmp` function is case sensitive in matching strings, while `strncmpi` is not.

Description

Although the following descriptions show only `strncmp`, they apply to `strncmpi` as well. The two functions are the same except that `strncmpi` compares strings without sensitivity to letter case:

`TF = strncmp('str1', 'str2', n)` compares the first `n` characters of strings `str1` and `str2` and returns logical 1 (`true`) if they are identical, and returns logical 0 (`false`) otherwise. `str1` and `str2` can be character arrays of any dimension, but `strncmp` does not return `true` unless the sizes of both arrays are equal, and the contents of the two arrays are the same.

`TF = strncmp('str', C, n)` compares the first `n` characters of `str` to the first `n` characters of each element of cell array `C`, where `str` is a character vector (or a 1-by-1 cell array), and `C` is a cell array of strings. The function returns `TF`, a logical array that is the same size as `C` and contains logical 1 (`true`) for those elements of `C` that are a match, and logical 0 (`false`) for those elements that are not. The order of the first two input arguments is not important.

`TF = strncmp(C1, C2, n)` compares each element of `C1` to the same element in `C2`, where `C1` and `C2` are equal-size cell arrays of strings. Input `C1` or `C2` can also be a character array with the right number of rows. The function attempts to match only the first `n` characters of each string. The function returns `TF`, a logical array that is the same size as `C1` and `C2`, and contains logical 1 (`true`) for those elements of `C1` and `C2` that are a match, and logical 0 (`false`) for those elements that are not.

Remarks

These functions are intended for comparison of character data. When used to compare numeric data, they return logical `0`.

Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

The value returned by `strncmp` and `strncmpi` is not the same as the C language convention.

`strncmp` and `strncmpi` support international character sets.

Examples

Example 1

From a list of 10 MATLAB functions, find those that apply to using a camera:

```
function_list = {'calendar' 'case' 'camdolly' 'circshift' ...
                 'caxis' 'camtarget' 'cast' 'camorbit' ...
                 'callib' 'cart2sph'};

strncmp(function_list, 'cam', 3)
ans =
    0    0    1    0    0    1    0    1    0    0

function_list{strncmp(function_list, 'cam', 3)}
ans =
   camdolly
ans =
   camtarget
ans =
   camorbit
```

Example 2

Create two 5-by-10 string arrays `str1` and `str2` that are equal except for the element at row 4, column 3. Using linear indexing, this is element 14:

```
str1 = ['AAAAAAAAAA'; 'BBBBBBBBBB'; 'CCCCCCCCCC'; ...
        'DDDDDDDDDD'; 'EEEEEEEEEE']
```

```
str1 =
    AAAAAAAAAA
    BBBBBBBBBB
    CCCCCCCCCC
    DDDDDDDDDD
    EEEEEEEEEE

str2 = str1;
str2(4,3) = '-'
str2 =
    AAAAAAAAAA
    BBBBBBBBBB
    CCCCCCCCCC
    DD-DDDDDDD
    EEEEEEEEEE
```

Because MATLAB compares the arrays in linear order (that is, column by column rather than row by row), `strncmp` finds only the first 13 elements to be the same:

```
str1   A B C D E A B C D E A B C D E
str2   A B C D E A B C D E A B C - E
                                 |
                              element 14

strncmp(str1, str2, 13)
ans =
     1

strncmp(str1, str2, 14)
ans =
     0
```

See Also

`strcmp`, `strcmpi`, `strmatch`, `strfind`, `findstr`, `regexp`, `regexpi`, `regexprep`, `regexptranslate`

Purpose

Read formatted data from string

Note The `textscan` function is intended as a replacement for both `strread` and `textread`.

Syntax

```
A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)
```

Description

`A = strread('str')` reads numeric data from input string `str` into a 1-by-`N` vector `A`, where `N` equals the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See "Example 1" on page 2-3135 below.

`[A, B, ...] = strread('str')` reads numeric data from the string input `str` into scalar output variables `A`, `B`, and so on. The number of output variables must equal the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See "Example 2" on page 2-3135 below.

`[A, B, ...] = strread('str', 'format')` reads data from `str` into variables `A`, `B`, and so on using the specified `format`. The number of output variables `A`, `B`, etc. must be equal to the number of format specifiers (e.g., `%s` or `%d`) in the `format` argument. You can read all of the data in `str` to a single output variable as long as you use only one format specifier in the command. See "Example 4" on page 2-3136 and "Example 5" on page 2-3136 below.

The table Formats for strread on page 2-3132 lists the valid format specifiers. More information on using formats is available under "Formats" on page 2-3134 in the Remarks section below.

`[A, B, ...] = strread('str', 'format', N)` reads data from `str` reusing the `format` string `N` times, where `N` is an integer greater than zero. If `N` is -1, `strread` reads the entire string. When `str` contains

only numeric data, you can set `format` to the empty string (`''`). See “Example 3” on page 2-3136 below.

`[A, B, ...] = strread('str', 'format', N, param, value, ...)` customizes `strread` using `param`/`value` pairs, as listed in the table Parameters and Values for strread on page 2-3133 below. When `str` contains only numeric data, you can set `format` to the empty string (`''`). The `N` argument is optional and may be omitted entirely. See “Example 7” on page 2-3137 below.

Formats for strread

Format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a string that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space separated string.	Cell array of strings
`%q`	Read a double quoted string, ignoring the quotes.	Cell array of strings
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings

Formats for strread (Continued)

Format	Action	Output
`%[^...]`	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
`%*...`	Ignore the characters following `*`. See “Example 8” on page 2-3137 below.	No output
`%w...`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

Parameters and Values for strread

param	value	Action
`whitespace`	`\*` where `*` can be	Treats vector of characters, `*`, as white space. Default is `\b\r\n\t`.
	`b`	Backspace
	`f`	Form feed
	`n`	New line
	`r`	Carriage return
	`t`	Horizontal tab
	`\\`	Backslash
	`\'' or ''`	Single quotation mark
	`%%`	Percent sign

Parameters and Values for strread (Continued)

param	value	Action
`delimiter`	Delimiter character	Specifies delimiter character. Default is one or more whitespace characters.
`expchars`	Exponent characters	Default is `eEdD`.
`bufsize`	Positive integer	Specifies the maximum string length, in bytes. Default is `4095`.
`commentstyle`	`matlab`	Ignores characters after `%`.
`commentstyle`	`shell`	Ignores characters after `#`.
`commentstyle`	`c`	Ignores characters between `/*` and `*/`.
`commentstyle`	`c++`	Ignores characters after `//`.

Remarks

Delimiters

If your data uses a character other than a space as a delimiter, you must use the `strread` parameter `'delimiter'` to specify the delimiter. For example, if the string `str` used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer] = strread(str,'%s %s %f ...
    %d %s','delimiter',';')
```

Formats

The `format` string determines the number and types of return arguments. The number of return arguments must match the number of conversion specifiers in the `format` string.

The `strread` function continues reading `str` until the entire string is read. If there are fewer format specifiers than there are entities in `str`,

strread reapplies the format specifiers, starting over at the beginning. See “Example 5” on page 2-3136 below.

The format string supports a subset of the conversion specifiers and conventions of the C language fscanf routine. White-space characters in the format string are ignored.

Preserving White-Space

If you want to preserve leading and trailing spaces in a string, use the whitespace parameter as shown here:

```
str = '   An  example      of preserving     spaces      ';

strread(str, '%s', 'whitespace', '')
ans =
    '   An  example      of preserving     spaces      '
```

Examples

Example 1

Read numeric data into a 1-by-5 vector:

```
a = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100    8.2400    3.5700    6.2400    9.2700
```

Example 2

Read numeric data into separate scalar variables:

```
[a b c d e] = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100
b =
    8.2400
c =
    3.5700
d =
    6.2400
e =
    9.2700
```

Example 3

Read the only first three numbers in the string, also formatting as floating point:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%4.2f', 3)

a =
    0.4100
    8.2400
    3.5700
```

Example 4

Truncate the data to one decimal digit by specifying format `%3.1f`. The second specifier, `%*1d`, tells `strread` not to read in the remaining decimal digit:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%3.1f %*1d')

a =
    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
```

Example 5

Read six numbers into two variables, reusing the format specifiers:

```
[a b] = strread('0.41 8.24 3.57 6.24 9.27 3.29', '%f %f')

a =
    0.4100
    3.5700
    9.2700
b =
    8.2400
    6.2400
```

```
    3.2900
```

Example 6

Read string and numeric data to two output variables. Ignore commas in the input string:

```
str = 'Section 4, Page 7, Line 26';

[name value] = strread(str, '%s %d,')
name =
    'Section'
    'Page'
    'Line'
value =
     4
     7
    26
```

Example 7

Read the string used in the last example, but this time delimiting with commas instead of spaces:

```
str = 'Section 4, Page 7, Line 26';

[a b c] = strread(str, '%s %s %s', 'delimiter', ',')
a =
    'Section 4'
b =
    'Page 7'
c =
    'Line 26'
```

Example 8

Read selected portions of the input string:

```
str = '<table border=5 width="100%" cellspacing=0>';

[border width space] = strread(str, ...
```

```
   '%*s%*s %c %*s "%4s" %*s %c', 'delimiter', '= ')
border =
     5
width =
    '100%'
space =
     0
```

Example 9

Read the string into two vectors, restricting the `Answer` values to `T` and `F`. Also note that two delimiters (comma and space) are used here:

```
str = 'Answer_1: T, Answer_2: F, Answer_3: F';

[a b] = strread(str, '%s %[TF]', 'delimiter', ', ')
a =
    'Answer_1:'
    'Answer_2:'
    'Answer_3:'
b =
    'T'
    'F'
    'F'
```

See Also

`textscan`, `textread`, `sscanf`

Purpose

Find and replace substring

Syntax

```
str = strrep(str1, str2, str3)
```

Description

`str = strrep(str1, str2, str3)` replaces all occurrences of the string `str2` within string `str1` with the string `str3`.

`strrep(str1, str2, str3)`, when any of `str1`, `str2`, or `str3` is a cell array of strings, returns a cell array the same size as `str1`, `str2`, and `str3` obtained by performing a `strrep` using corresponding elements of the inputs. The inputs must all be the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the right number of rows.

Examples

```
s1 = 'This is a good example.';
str = strrep(s1, 'good', 'great')
str =
This is a great example.
A =
    'MATLAB'       'SIMULINK'
    'Toolboxes'    'The MathWorks'
B =
    'Handle Graphics'    'Real Time Workshop'
    'Toolboxes'          'The MathWorks'
C =
    'Signal Processing'    'Image Processing'
    'MATLAB'               'SIMULINK'
strrep(A, B, C)
ans =
    'MATLAB' 'SIMULINK'
    'MATLAB' 'SIMULINK'
```

See Also

`strfind`

Purpose

Selected parts of string

Syntax

```
token = strtok('str')
token = strtok('str', delimiter)
[token, remain] = strtok('str', ...)
```

Description

`token = strtok('str')` returns in `token` that part of the input string `str` that precedes the first white-space character (the default delimiter). Parsing of the string begins at the first nondelimiting (i.e., nonwhite-space) character and continues to the right until MATLAB either locates a delimiter or reaches the end of the string. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned.

White-space characters include space (ASCII 32), tab (ASCII 9), and carriage return (ASCII 13).

If `str` is a cell array of strings, `token` is a cell array of tokens.

`token = strtok('str', delimiter)` [4] is the same as the above syntax except that you can specify one or more nondefault delimiters in the character vector, `delimiter`. Ignoring any leading delimiters, MATLAB returns in `token` that part of the input string that precedes one of the characters from the given `delimiter` vector.

`[token, remain] = strtok('str', ...)` returns in `remain` a substring of the input string that begins immediately after the `token` substring and ends with the last character in `str`. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned in `token`, and `remain` is an empty string (`''`).

If `str` is a cell array of strings, `token` is a cell array of tokens and `remain` is a character array.

Examples

Example 1

This example uses the default white-space delimiter:

```
s = '  This is a simple example.';
```

```
[token, remain] = strtok(s)
token =
   This
remain =
   is a simple example.
```

Example 2

Take a string of HTML code and break it down into segments delimited by the < and > characters. Write a `while` loop to parse the string and print each segment:

```
s = sprintf('%s%s%s%s', ...
'<ul class=continued><li class=continued>', ...
'<pre><a name="13474"></a>token = strtok', ...
'(''str'', delimiter)<a name="13475"></a>', ...
'token = strtok(''str'')');

remain = s;

while true
   [str, remain] = strtok(remain, '<>');
   if isempty(str),  break;  end
   disp(sprintf('%s', str))
   end
```

Here is the output:

```
ul class=continued
li class=continued
pre
a name="13474"
/a
token = strtok('str', delimiter)
a name="13475"
/a
token = strtok('str')
```

Example 3

Using `strtok` on a cell array of strings returns a cell array of strings in `token` and a character array in `remain`:

```
s = {'all in good time'; ...
     'my dog has fleas'; ...
     'leave no stone unturned'};

remain = s;

for k = 1:4
   [token, remain] = strtok(remain);
   token
   end
```

Here is the output:

```
token =
    'all'
    'my'
    'leave'
token =
    'in'
    'dog'
    'no'
token =
    'good'
    'has'
    'stone'
token =
    'time'
    'fleas'
    'unturned'
```

See Also

`findstr`, `strmatch`

Purpose

Remove leading and trailing white space from string

Syntax

```
S = strtrim(str)
C = strtrim(cstr)
```

Description

`S = strtrim(str)` returns a copy of string `str` with all leading and trailing white-space characters removed. A white-space character is one for which the `isspace` function returns logical `1` (`true`).

`C = strtrim(cstr)` returns a copy of the cell array of strings `cstr` with all leading and trailing white-space characters removed from each string in the cell array.

Examples

Remove the leading white-space characters (spaces and tabs) from `str`:

```
str = sprintf('  \t   Remove    leading white-space')
str =
        Remove    leading white-space

str = strtrim(str)
str =
Remove    leading white-space
```

Remove leading and trailing white-space from the cell array of strings:

```
cstr = {'     Trim leading white-space';
        'Trim trailing white-space     '};

cstr = strtrim(cstr)
cstr =
    'Trim leading white-space'
    'Trim trailing white-space'
```

See Also

`isspace`, `cellstr`, `deblank`, `strjust`

Purpose

Create structure array

Syntax

```
s = struct('field1', values1, 'field2', values2, ...)
s = struct('field1', {}, 'field2', {}, ...)
s = struct
s = struct([])
s = struct(obj)
```

Description

`s = struct('field1', values1, 'field2', values2, ...)` creates a structure array with the specified fields and values. Each value input (`values1`, `values2`, etc.), can either be a cell array or a scalar value. Those that are cell arrays must all have the same dimensions.

The size of the resulting structure is the same size as the value cell arrays, or 1-by-1 if none of the values is a cell array. Elements of the `value` array inputs are placed into corresponding structure array elements.

Note If any of the `values` fields is an empty cell array `{}`, MATLAB creates an empty structure array in which all fields are also empty.

Structure field names must begin with a letter, and are case-sensitive. The rest of the name may contain letters, numerals, and underscore characters. Use the `namelengthmax` function to determine the maximum length of a field name.

`s = struct('field1', {}, 'field2', {}, ...)` creates an empty structure with fields `field1`, `field2`, `...`

`s = struct` creates a 1-by-1 structure with no fields.

`s = struct([])` creates an empty structure with no fields.

`s = struct(obj)` creates a structure identical to the underlying structure in the object `obj`. The class information is lost.

Remarks

Two Ways to Access Fields

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time.

Fields That Are Cell Arrays

To create fields that contain cell arrays, place the cell arrays within a `value` cell array. For instance, to create a 1-by-1 structure, type

```
s = struct('strings',{{'hello','yes'}},'lengths',[5 3])
s =
   strings: {'hello'  'yes'}
   lengths: [5 3]
```

Specifying Cell Versus Noncell Values

When using the syntax

```
s = struct('field1', values1, 'field2', values2, ...)
```

the `values` inputs can be cell arrays or scalar values. For those values that are specified as a cell array, MATLAB assigns each element of `values{m,n,...}` to the corresponding field in each element of structure `s`:

```
s(m,n,...).fieldN = valuesN{m,n,...}
```

For those values that are scalar, MATLAB assigns that single value to the corresponding field for all elements of structure `s`:

```
s(m,n,...).fieldN = valuesN
```

See Example 3, below.

Examples

Example 1

The command

```
s = struct('type', {'big','little'}, 'color', {'red'}, ...
    'x', {3 4})
```

produces a structure array s:

```
s =
1x2 struct array with fields:
    type
    color
    x
```

The value arrays have been distributed among the fields of s:

```
s(1)
ans =
       type: 'big'
     color: 'red'
         x: 3
s(2)
ans =
       type: 'little'
     color: 'red'
         x: 4
```

Example 2

Similarly, the command

```
a.b = struct('z', {});
```

produces an empty structure `a.b` with field `z`.

```
a.b
ans =
      0x0 struct array with fields:
          z
```

Example 3

This example initializes one field `f1` using a cell array, and the other `f2` using a scalar value:

```
s = struct('f1', {1 3; 2 4}, 'f2', 25)
s =
2x2 struct array with fields:
    f1
    f2
```

Field `f1` in each element of `s` is assigned the corresponding value from the cell array `{1 3; 2 4}`:

```
s.f1
ans =
     1
ans =
     2
ans =
     3
ans =
     4
```

Field `f2` for all elements of `s` is assigned one common value because the `values` input for this field was specified as a scalar:

```
s.f2
ans =
    25
ans =
    25
ans =
    25
ans =
    25
```

See Also

isstruct, fieldnames, isfield, orderfields, getfield, setfield, rmfield, substruct, deal, cell2struct, struct2cell, namelengthmax, dynamic field names

Purpose

Convert structure to cell array

Syntax

```
c = struct2cell(s)
```

Description

`c = struct2cell(s)` converts the `m`-by-`n` structure `s` (with `p` fields) into a `p`-by-`m`-by-`n` cell array `c`.

If structure `s` is multidimensional, cell array `c` has size `[p size(s)]`.

Examples

The commands

```
clear s, s.category = 'tree';
s.height = 37.4; s.name = 'birch';
```

create the structure

```
s =
    category: 'tree'
      height: 37.4000
        name: 'birch'
```

Converting the structure to a cell array,

```
c = struct2cell(s)

c =
    'tree'
    [37.4000]
    'birch'
```

See Also

`cell2struct`, `cell`, `iscell`, `struct`, `isstruct`, `fieldnames`, "Using Dynamic Field Names"

Purpose

Apply function to each field of scalar structure

Syntax

```
A = structfun(fun, S)
[A, B, ...] = structfun(fun, S)
[A, ...] = structfun(fun, S, 'param1', value1, ...)
```

Description

`A = structfun(fun, S)` applies the function specified by `fun` to each field of scalar structure `S`, and returns the results in array `A`. `fun` is a function handle to a function that takes one input argument and returns a scalar value. Return value `A` is a column vector that has one element for each field in input structure `S`. The Nth element of `A` is the result of applying `fun` to the Nth field of `S`, and the order of the fields is the same as that returned by a call to `fieldnames`. (`A` is returned as one or more scalar structures when the `UniformOutput` option is set to `false`. See the table below.))

`fun` must return values of the same class each time it is called. If `fun` is a handle to an overloaded function, then `structfun` follows MATLAB dispatching rules in calling the function.

`[A, B, ...] = structfun(fun, S)` returns arrays `A`, `B`, `...`, each array corresponding to one of the output arguments of `fun`. `structfun` calls `fun` each time with as many outputs as there are in the call to `structfun`. `fun` can return output arguments having different classes, but the class of each output must be the same each time `fun` is called.

`[A, ...] = structfun(fun, S, 'param1', value1, ...)` enables you to specify optional parameter name/parameter value pairs. Parameters are

Parameter	Value
`'UniformOutput'`	Logical value indicating whether or not the outputs of `fun` can be returned without encapsulation in a structure. The default value is `true`. If equal to logical 1 (`true`), `fun` must return scalar values that can be concatenated into an array. The outputs can be any of the following types: numeric, logical, char, struct, or cell. If equal to logical 0 (`false`), `structfun` returns a scalar structure or multiple scalar structures having fields that are the same as the fields of the input structure `S`. The values in the output structure fields are the results of calling `fun` on the corresponding values in the input structure `B`. In this case, the outputs can be of any data type.

`'ErrorHandler'`	Function handle specifying the function MATLAB is to call if the call to `fun` fails. MATLAB calls the error handling function with the following input arguments:
	
	• A structure, with the fields `'identifier'`, `'message'`, and `'index'`, respectively containing the identifier of the error that occurred, the text of the error message, and the number of the field (in the same order as returned by field names) at which the error occurred.
	• The input argument at which the call to the function failed.
	
	The error handling function should either rethrow an error or return the same number of outputs as `fun`. These outputs are then returned as the outputs of `structfun`. If `'UniformOutput'` is `true`, the outputs of the error handler must also be scalars of the same type as the outputs of `fun`.
	
	For example,
	
	`function [A, B] = errorFunc(S, ... varargin)`
	`warning(S.identifier, S.message);`

Examples

To create shortened weekday names from the full names, for example: Create a structure with strings in several fields:

```
s.f1 = 'Sunday';
s.f2 = 'Monday';
s.f3 = 'Tuesday';
s.f4 = 'Wednesday';
s.f5 = 'Thursday';
s.f6 = 'Friday';
s.f7 = 'Saturday';

shortNames = structfun(@(x) ( x(1:3) ), s, ...
                          'UniformOutput', false);
```

See Also

`cellfun`, `arrayfun`, `function_handle`, `cell2mat`, `spfun`

Purpose

Concatenate strings vertically

Syntax

```
S = strvcat(t1, t2, t3, ...)
S = strvcat(c)
```

Description

`S = strvcat(t1, t2, t3, ...)` forms the character array `S` containing the text strings (or string matrices) `t1,t2,t3,...` as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

`S = strvcat(c)` when `c` is a cell array of strings, passes each element of `c` as an input to `strvcat`. Empty strings in the input are ignored.

Remarks

If each text parameter, `ti`, is itself a character array, `strvcat` appends them vertically to create arbitrarily large string matrices.

Examples

The command `strvcat('Hello','Yes')` is the same as `['Hello';'Yes ']`, except that `strvcat` performs the padding automatically.

```
t1 = 'first'; t2 = 'string'; t3 = 'matrix'; t4 = 'second';

S1 = strvcat(t1, t2, t3)        S2 = strvcat(t4, t2, t3)

S1 =                            S2 =

first                           second
string                          string
matrix                          matrix

S3 = strvcat(S1, S2)

S3 =
first
string
matrix
second
string
```

```
matrix
```

See Also

strcat, cat, int2str, mat2str, num2str, strings

Purpose

Single index from subscripts

Syntax

```
IND = sub2ind(siz,I,J)
IND = sub2ind(siz,I1,I2,...,In)
```

Description

The `sub2ind` command determines the equivalent single index corresponding to a set of subscript values.

`IND = sub2ind(siz,I,J)` returns the linear index equivalent to the row and column subscripts `I` and `J` for a matrix of size `siz`. `siz` is a 2-element vector, where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

`IND = sub2ind(siz,I1,I2,...,In)` returns the linear index equivalent to the `n` subscripts `I1,I2,...,In` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

Examples

Create a 3-by-4-by-2 array, `A`.

```
A = [17 24 1 8; 2 22 7 14; 4 6 13 20];
A(:,:,2) = A - 10

A(:,:,1) =

    17    24     1     8
     2    22     7    14
     4     6    13    20


A(:,:,2) =

     7    14    -9    -2
    -8    12    -3     4
    -6    -4     3    10
```

The value at row 2, column 1, page 2 of the array is -8.

```
A(2,1,2)
```

```
ans =

    -8
```

To convert `A(2,1,2)` into its equivalent single subscript, use `sub2ind`.

```
sub2ind(size(A),2,1,2)

ans =

    14
```

You can now access the same location in `A` using the single subscripting method.

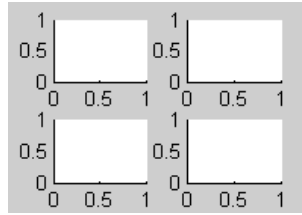
```
A(14)

ans =

    -8
```

See Also

`ind2sub`, `find`, `size`

Purpose

Create axes in tiled positions



GUI Alternatives

To add subplots to a figure, click one of the *New Subplot* icons in the Figure Palette, and slide right to select an arrangement of subplots. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation.

Syntax

```
h = subplot(m,n,p) or subplot(mnp)
subplot(m,n,p,'replace')
subplot(m,n,P)
subplot(h)
subplot('Position',[left bottom width height])
h = subplot(...)
subplot(m,n,p,'v6')
```

Description

`subplot` divides the current figure into rectangular panes that are numbered rowwise. Each pane contains an axes object. Subsequent plots are output to the current pane.

`h = subplot(m,n,p) or subplot(mnp)` breaks the figure window into an `m`-by-`n` matrix of small axes, selects the `p`th axes object for the current plot, and returns the axes handle. The axes are counted along the top row of the figure window, then the second row, etc. For example,

```
subplot(2,1,1), plot(income)
subplot(2,1,2), plot(outgo)
```

plots income on the top half of the window and outgo on the bottom half. If the `CurrentAxes` is nested in a uipanel, the panel is used as

the parent for the subplot instead of the current figure. The new axes object becomes the current axes.

`subplot(m,n,p,'replace')` If the specified axes object already exists, delete it and create a new axes.

`subplot(m,n,P)`, where `P` is a vector, specifies an axes position that covers all the subplot positions listed in `P`, including those spanned by `P`. For example, `subplot(2,3,[2 5])` creates subplots at positions 2 and 5 only (because there are no intervening locations in the grid), while `subplot(2,3,[2 6])` creates axes at positions 2, 3, 5, and 6.

`subplot(h)` makes the axes object with handle `h` current for subsequent plotting commands.

`subplot('Position',[left bottom width height])` creates an axes at the position specified by a four-element vector. `left`, `bottom`, `width`, and `height` are in normalized coordinates in the range from 0.0 to 1.0.

`h = subplot(...)` returns the handle to the new axes object.

Backward-Compatible Version

`subplot(m,n,p,'v6')` places the axes so that the plot boxes are aligned, but does not prevent the labels and ticks from overlapping. Saved subplots created with the `v6` option are compatible with MATLAB 6.5 and earlier versions.

Use the `subplot 'v6'` option and save the figure with the `'v6'` option when you want to be able to load a FIG-file containing subplots into MATLAB Version 6.5 or earlier.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

You can add subplots to GUIs as well as to figures. For information about creating subplots in a GUIDE-generated GUI, see "Creating Subplots" in the MATLAB Creating Graphical User Interfaces documentation.

If a `subplot` specification causes a new axes object to overlap any existing axes, `subplot` deletes the existing axes object and uicontrol objects. However, if the `subplot` specification exactly matches the position of an existing axes object, the matching axes object is not deleted and it becomes the current axes.

`subplot(1,1,1)` or `clf` deletes all axes objects and returns to the default `subplot(1,1,1)` configuration.

You can omit the parentheses and specify `subplot` as

```
subplot mnp
```

where `m` refers to the row, `n` refers to the column, and `p` specifies the pane.

Be aware when creating subplots from scripts that the `Position` property of subplots is not finalized until either

- A `drawnow` command is issued.
- MATLAB returns to await a user command.

That is, the value obtained for subplot `i` by the command

```
get(h(i),'position')
```

will not be correct until the script refreshes the plot or exits.

Special Case: subplot(111)

The command `subplot(111)` is not identical in behavior to `subplot(1,1,1)` and exists only for compatibility with previous releases. This syntax does not immediately create an axes object, but instead sets up the figure so that the next graphics command executes a `clf reset` (deleting all figure children) and creates a new axes object in

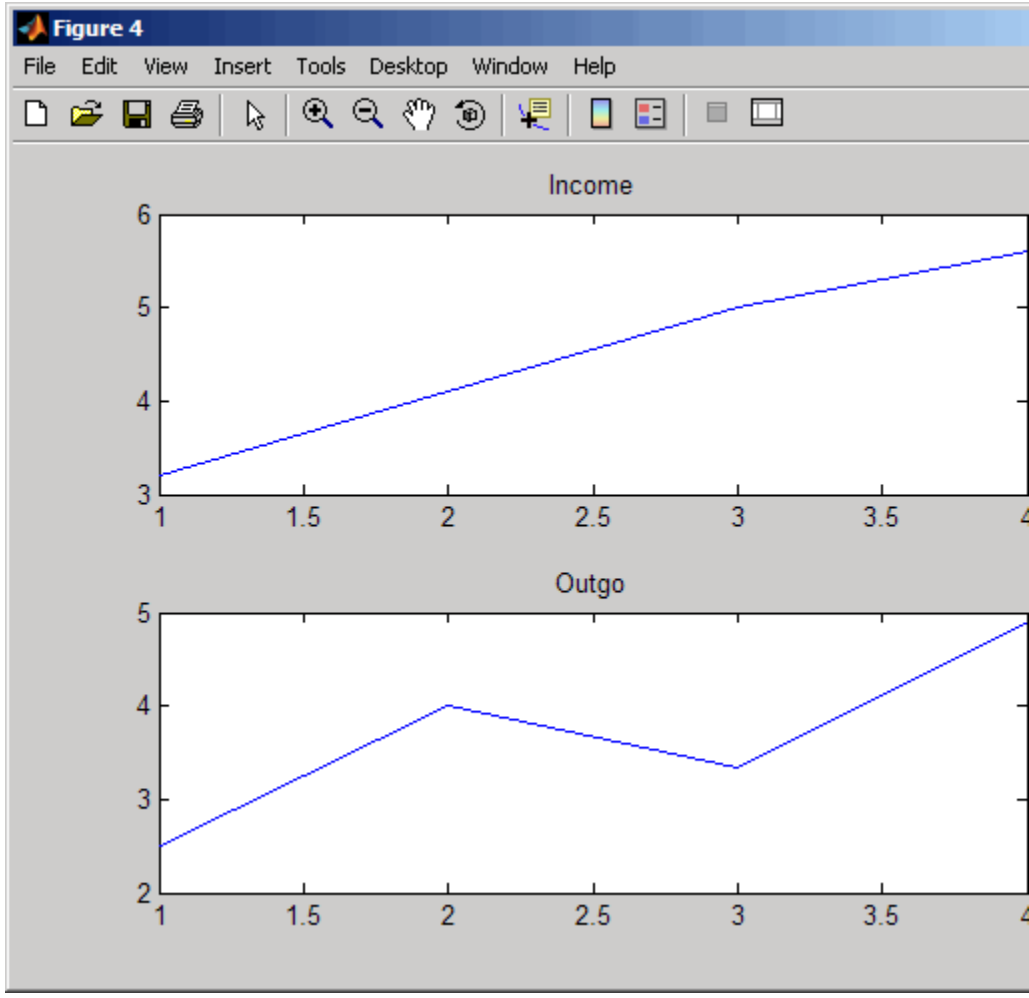
the default position. This syntax does not return a handle, so it is an error to specify a return argument. (MATLAB implements this behavior by setting the figure's `NextPlot` property to `replace`.)

Examples

Upper and Lower Subplots with Titles

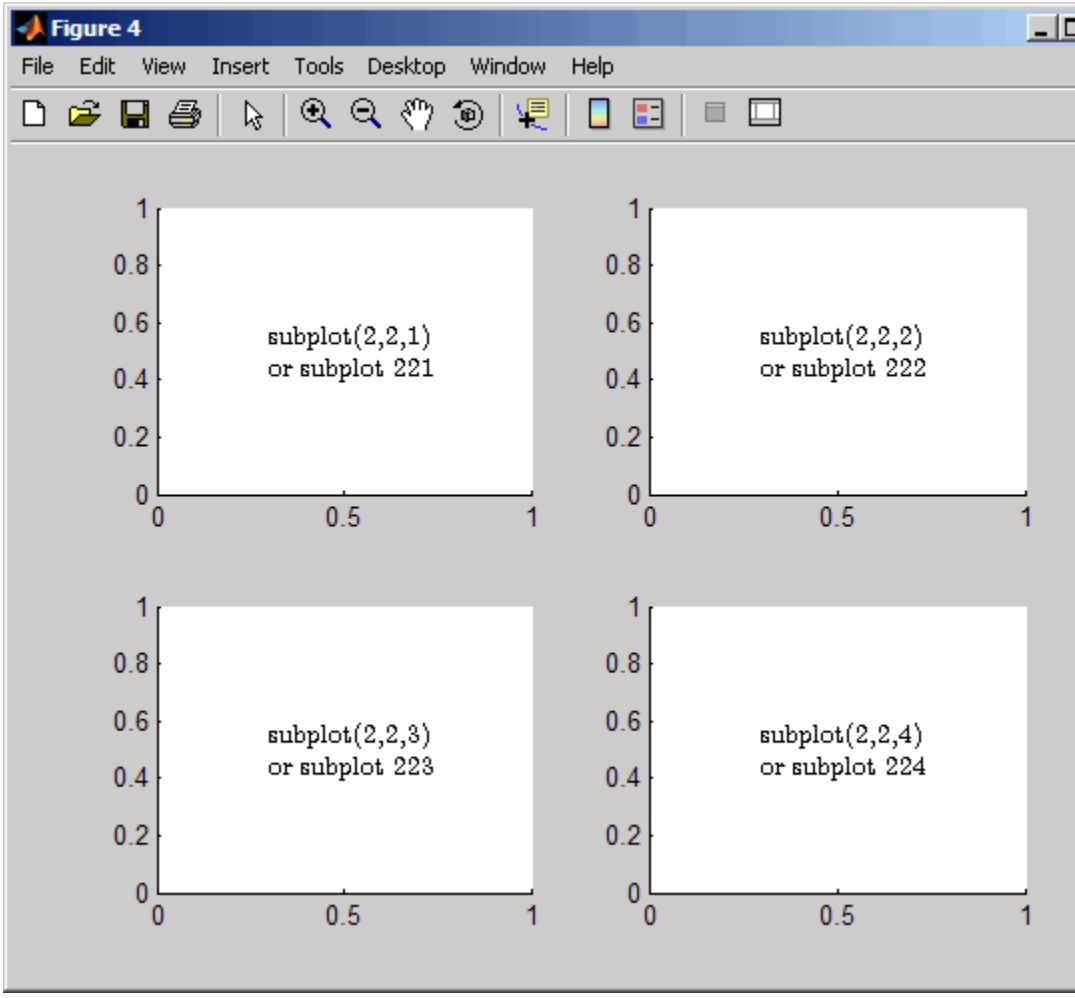
To plot `income` in the top half of a figure and `outgo` in the bottom half,

```
income = [3.2 4.1 5.0 5.6];
outgo = [2.5 4.0 3.35 4.9];
subplot(2,1,1); plot(income)
title('Income')
subplot(2,1,2); plot(outgo)
title('Outgo')
```



Subplots in Quadrants

The following illustration shows four subplot regions and indicates the command used to create each.

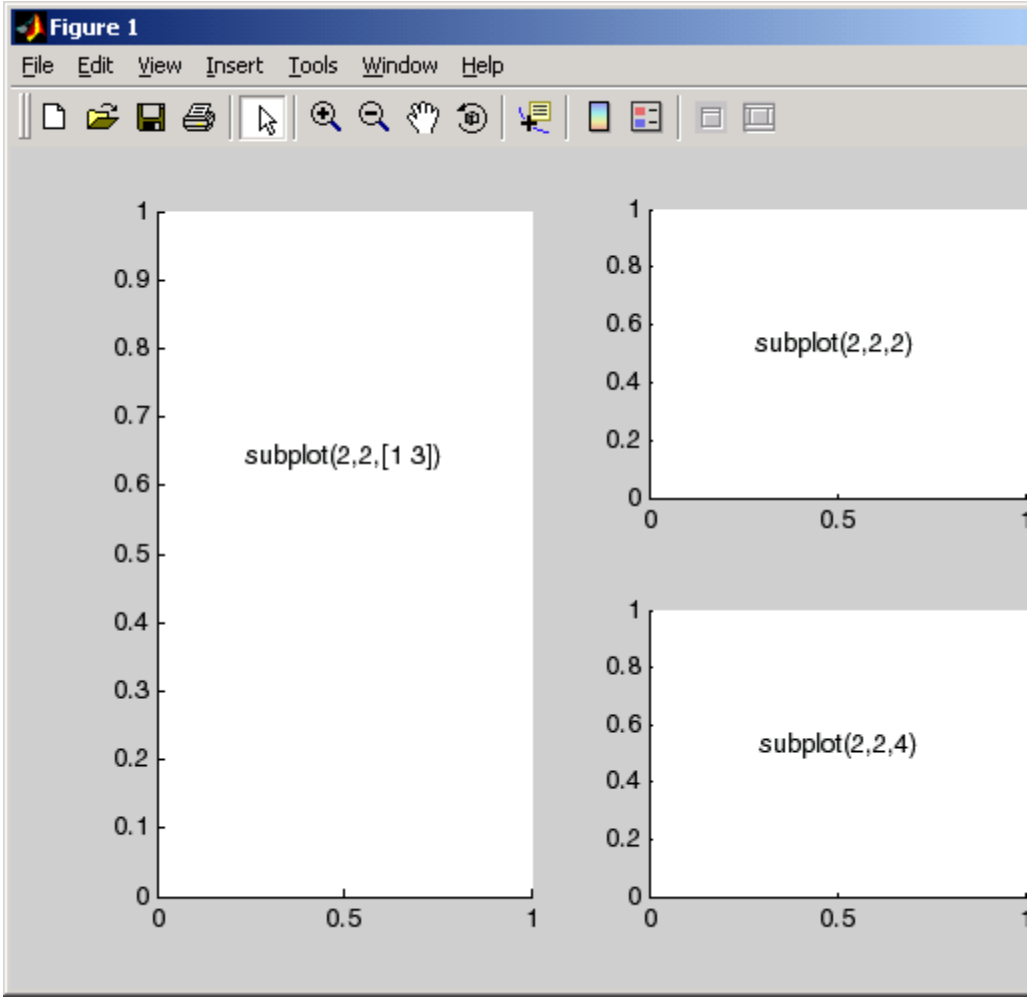


Assymetrical Subplots

The following combinations produce asymmetrical arrangements of subplots.

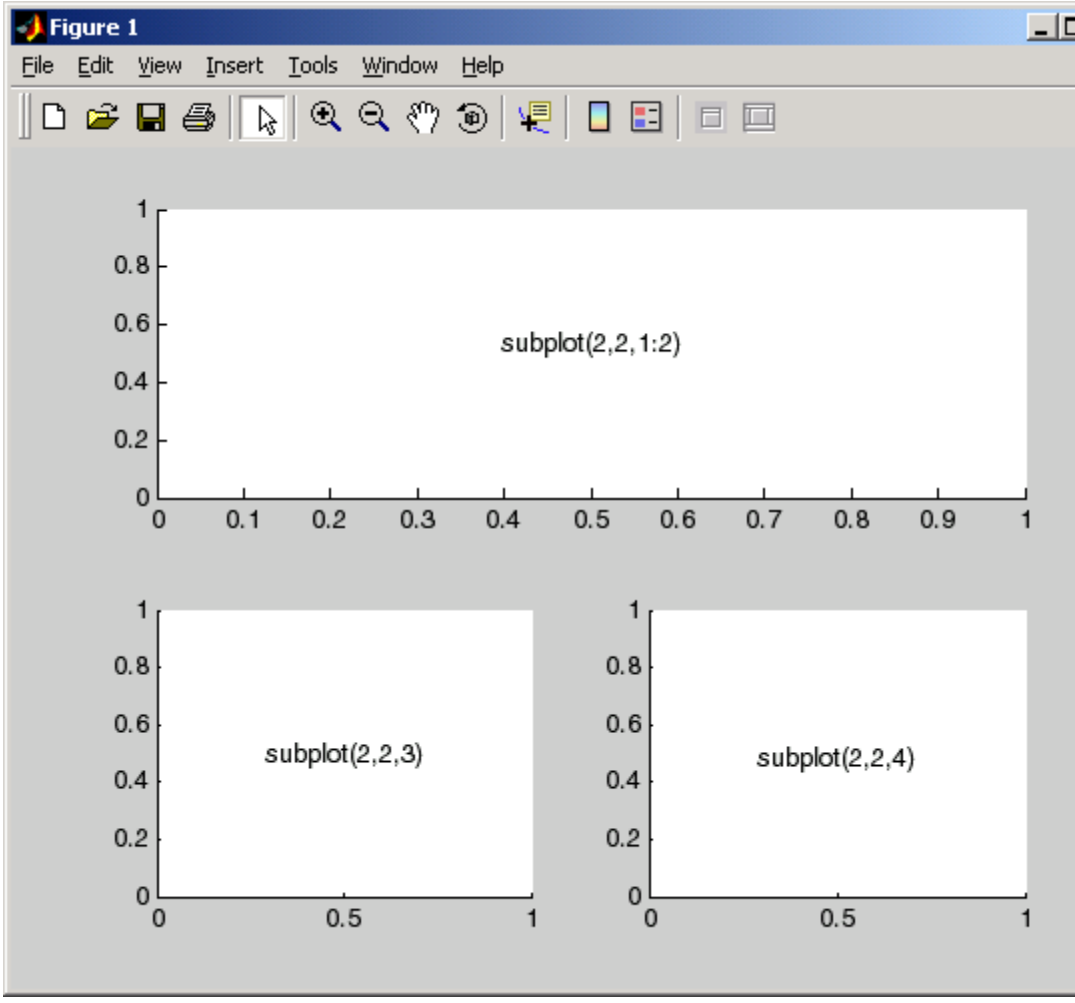
```
subplot(2,2,[1 3])
```

```
subplot(2,2,2)
subplot(2,2,4)
```



You can also use the colon operator to specify multiple locations if they are in sequence.

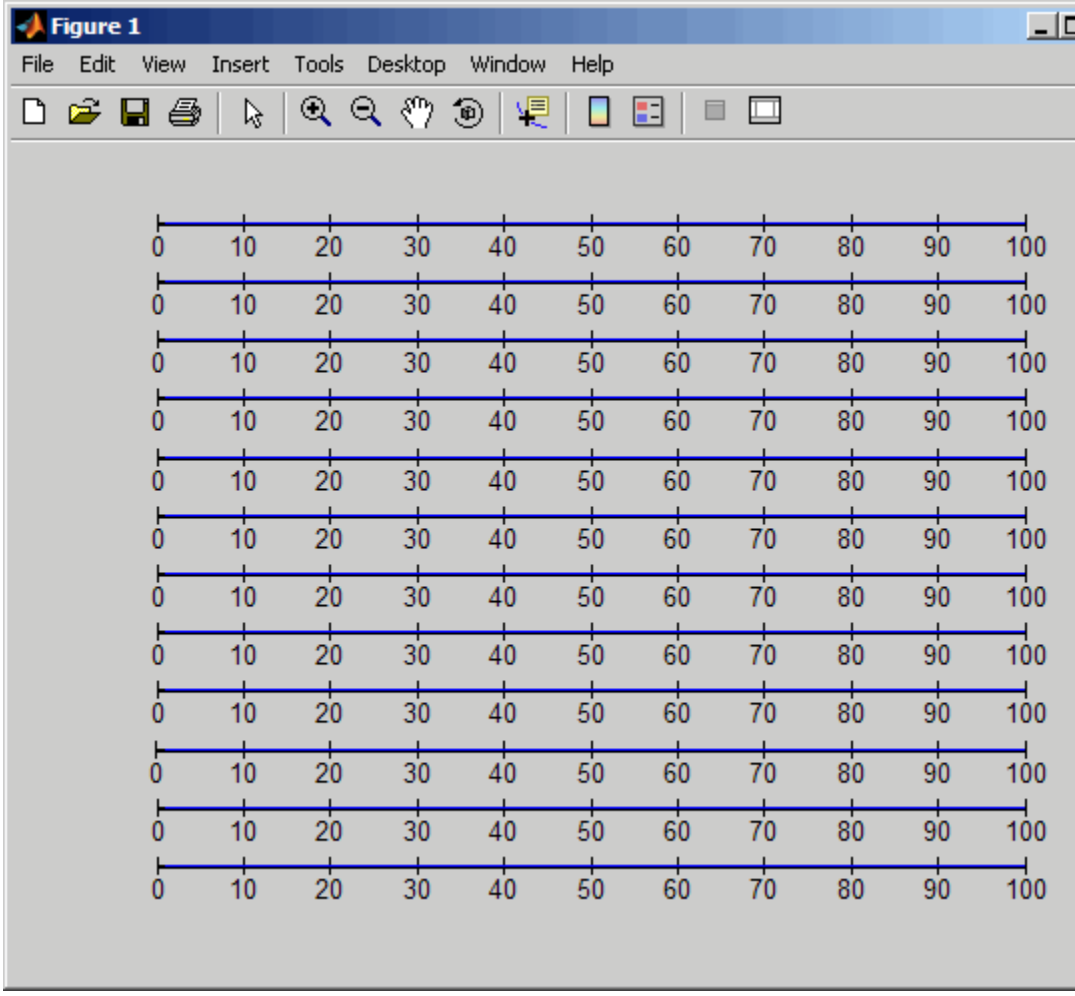
```
subplot(2,2,1:2)
subplot(2,2,3)
subplot(2,2,4)
```



Suppressing Axis Ticks

When you create many subplots in a figure, the axes tickmarks, which are shown by default, can either be obliterated or can cause axes to collapse, as the following code demonstrates:

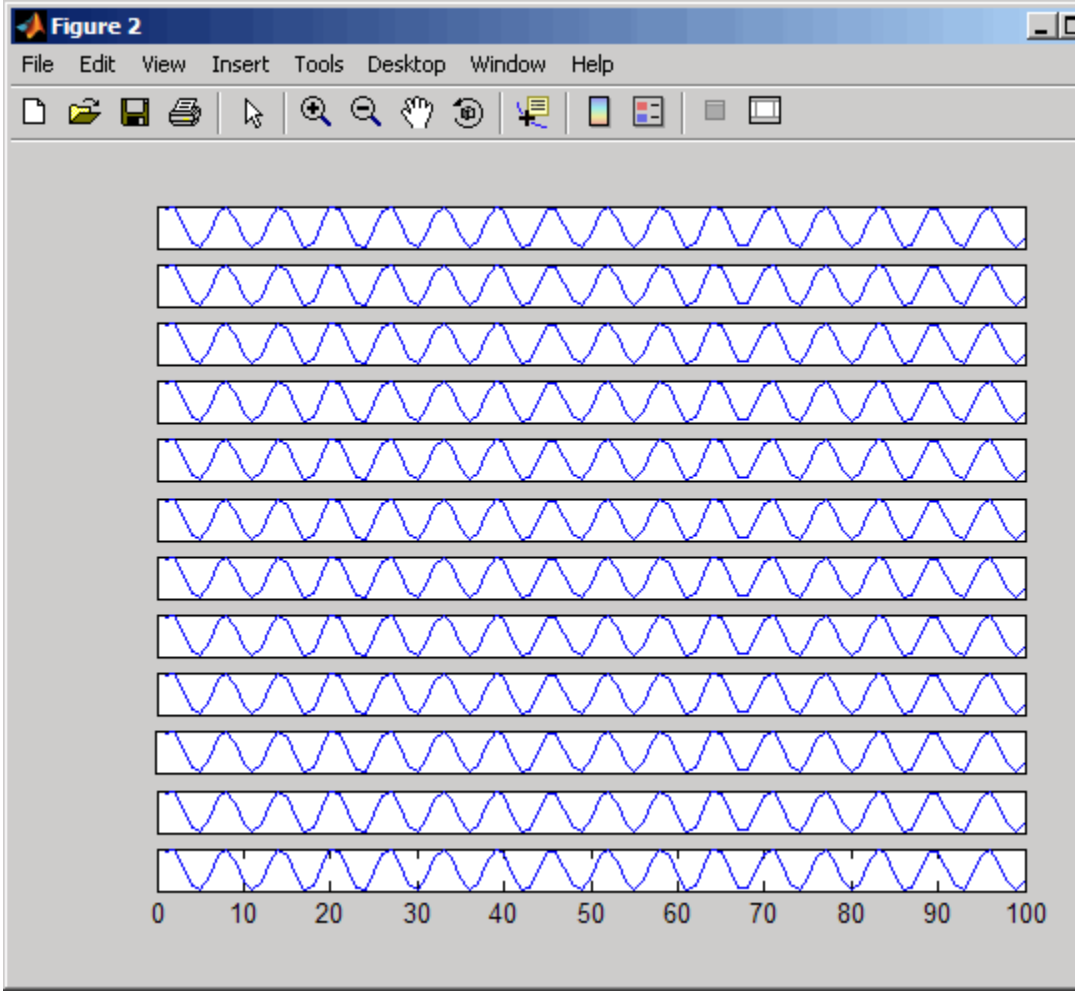
```
figure
for i=1:12
    subplot(12,1,i)
    plot (sin(1:100)*10^(i-1))
end
```



One way to get around this issue is to enlarge the figure to create enough space to properly display the tick labels.

Another approach is to eliminate the clutter by suppressing xticks and yticks for subplots as data are plotted into them. You can then label a single axes if the subplots are stacked, as follows:

```
figure
for i=1:12
    subplot(12,1,i)
    plot (sin(1:100)*10^(i-1))
    set(gca,'xtick',[],'ytick',[])
end
% Reset the bottom subplot to have xticks
set(gca,'xtickMode', 'auto')
```



See Also

`axes`, `cla`, `clf`, `figure`, `gca`

"Basic Plots and Graphs" on page 1-86 for more information

"Creating Subplots" in the MATLAB Creating Graphical User Interfaces documentation describes adding subplots to GUIs.

subsasgn

Purpose

Subscripted assignment for objects

Syntax

```
A = subsasgn(A, S, B)
```

Description

`A = subsasgn(A, S, B)` is called for the syntax `A(i)=B`, `A{i}=B`, or `A.i=B` when `A` is an object. `S` is a structure array with the fields

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Remarks

`subsasgn` is designed to be used by the MATLAB interpreter to handle indexed assignments to objects. Calling `subsasgn` directly as a function is not recommended. If you do use `subsasgn` in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.

In the assignment `A(J,K,...) = B(M,N,...)`, subscripts `J`, `K`, `M`, `N`, etc. may be scalar, vector, or array, provided that all of the following are true:

- The number of subscripts specified for `B`, excluding trailing subscripts equal to 1, does not exceed `ndims(B)`.
- The number of nonscalar subscripts specified for `A` equals the number of nonscalar subscripts specified for `B`. For example, `A(5, 1:4, 1, 2) = B(5:8)` is valid because both sides of the equation use one nonscalar subscript.
- The order and length of all nonscalar subscripts specified for `A` matches the order and length of nonscalar subscripts specified for `B`. For example, `A(1:4, 3, 3:9) = B(5:8, 1:7)` is valid because both sides of the equation (ignoring the one scalar subscript `3`) use a 4-element subscript followed by a 7-element subscript.

See the Remarks section of the `numel` reference page for information concerning the use of `numel` with regards to the overloaded `subsasgn` function.

If A is an array of one of the fundamental MATLAB data types, then assigning a value to A with indexed assignment calls the builtin MATLAB subsasgn method. It does not call any subsasgn method that you may have overloaded for that data type. For example, if A is an array of type double, and there is an @double/subsasgn method on your MATLAB path, the statement A(I) = B does not call this method, but calls the MATLAB builtin subsasgn method instead.

Examples

The syntax A(1:2,:)=B calls A=subsasgn(A,S,B) where S is a 1-by-1 structure with S.type='()' and S.subs = {1:2,':'}. A colon used as a subscript is passed as the string ':'.

The syntax A{1:2}=B calls A=subsasgn(A,S,B) where S.type='{}'.

The syntax A.field=B calls subsasgn(A,S,B) where S.type='.' and S.subs='field'.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases length(S) is the number of subscripting levels. For instance, A(1,2).name(3:5)=B calls A=subsasgn(A,S,B) where S is a 3-by-1 structure array with the following values:

```
S(1).type='()'      S(2).type='.'       S(3).type='()'
S(1).subs={1,2}     S(2).subs='name'    S(3).subs={3:5}
```

See Also

subsref, substruct

See “Handling Subscripted Assignment” for more information about overloaded methods and subsasgn.

subsindex

Purpose

Subscripted indexing for objects

Syntax

```
ind = subsindex(A)
```

Description

`ind = subsindex(A)` is called for the syntax `'X(A)'` when `A` is an object. `subsindex` must return the value of the object as a zero-based integer index. (`ind` must contain integer values in the range `0` to `prod(size(X))-1`.) `subsindex` is called by the default `subsref` and `subsasgn` functions, and you can call it if you overload these functions.

See Also

`subsasgn`, `subsref`

Purpose

Angle between two subspaces

Syntax

```
theta = subspace(A,B)
```

Description

`theta = subspace(A,B)` finds the angle between two subspaces specified by the columns of `A` and `B`. If `A` and `B` are column vectors of unit length, this is the same as `acos(A'*B)`.

Remarks

If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations `A`, and a second realization of the experiment described by `B`, `subspace(A,B)` gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples

Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

```
H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);
```

Note that matrices `A` and `B` are different sizes — `A` has three columns and `B` four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

```
theta = subspace(A,B)
theta =
    1.5708
```

That `A` and `B` are orthogonal is shown by the fact that `theta` is equal to $\pi/2$.

```
theta - pi/2
ans =
     0
```

subsref

Purpose

Subscripted reference for objects

Syntax

```
B = subsref(A, S)
```

Description

`B = subsref(A, S)` is called for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object. `S` is a structure array with the fields

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Remarks

`subsref` is designed to be used by the MATLAB interpreter to handle indexed references to objects. Calling `subsref` directly as a function is not recommended. If you do use `subsref` in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.

See the Remarks section of the `numel` reference page for information concerning the use of `numel` with regards to the overloaded `subsref` function.

If `A` is an array of one of the fundamental MATLAB data types, then referencing a value of `A` using an indexed reference calls the builtin MATLAB `subsref` method. It does not call any `subsref` method that you may have overloaded for that data type. For example, if `A` is an array of type `double`, and there is an `@double/subsref` method on your MATLAB path, the statement `B = A(I)` does not call this method, but calls the MATLAB builtin `subsref` method instead.

Examples

The syntax `A(1:2,:)` calls `subsref(A,S)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs={1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}` calls `subsref(A,S)` where `S.type='{}'` and `S.subs={1:2}`.

The syntax `A.field` calls `subsref(A,S)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is a 3-by-1 structure array with the following values:

```
S(1).type='()'      S(2).type='.'       S(3).type='()'
S(1).subs={1,2}     S(2).subs='name'    S(3).subs={3:5}
```

See Also

`subsasgn`, `substruct`

See "Handling Subscripted Reference" for more information about overloaded methods and `subsref`.

Purpose

Create structure argument for subsasgn or subsref

Syntax

```
S = substruct(type1, subs1, type2, subs2, ...)
```

Description

`S = substruct(type1, subs1, type2, subs2, ...)` creates a structure with the fields required by an overloaded `subsref` or `subsasgn` method. Each `type` string must be one of `'.'`, `'()'`, or `'{}'`. The corresponding `subs` argument must be either a field name (for the `'.'` type) or a cell array containing the index vectors (for the `'()'` or `'{}'` types).

The output `S` is a structure array containing the fields

- `type`: one of `'.'`, `'()'`, or `'{}'`
- `subs`: subscript values (field name or cell array of index vectors)

Examples

To call `subsref` with parameters equivalent to the syntax

```
B = A(3,5).field
```

you can use

```
S = substruct('()', {3,5}, '.', 'field');
B = subsref(A, S);
```

The structure created by `substruct` in this example contains the following:

```
S(1)

ans =

    type: '()'
    subs: {[3]  [5]}

S(2)
```

```
ans =

    type: '.'
    subs: 'field'
```

See Also

subsasgn, subsref

subvolume

Purpose

Extract subset of volume data set

Syntax

```
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)
[Nx,Ny,Nz,Nv] = subvolume(V,limits)
Nv = subvolume(...)
```

Description

`[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)` extracts a subset of the volume data set `V` using the specified axis-aligned `limits`. `limits = [xmin,xmax,ymin, ymax,zmin,zmax]` (Any `NaN`s in the limits indicate that the volume should not be cropped along that axis.)

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The subvolume is returned in `NV` and the coordinates of the subvolume are given in `NX`, `NY`, and `NZ`.

`[Nx,Ny,Nz,Nv] = subvolume(V,limits)` assumes the arrays `X`, `Y`, and `Z` are defined as

```
[X,Y,Z] = meshgrid(1:N,1:M,1:P)
```

where `[M,N,P] = size(V)`.

`Nv = subvolume(...)` returns only the subvolume.

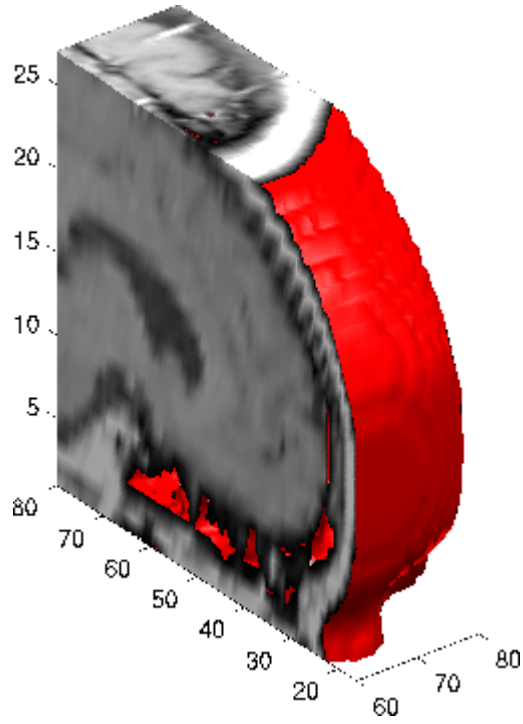
Examples

This example uses a data set that is a collection of MRI slices of a human skull. The data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then a subset of the data is extracted (`subvolume`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).

- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).
- Adding lights to the right and left of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = subvolume(D,[60,80,nan,80,nan,nan]);
p1 = patch(isosurface(x,y,z,D, 5),...
     'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
     'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight right; camlight left; lighting gouraud
```



See Also

`isocaps`, `isonormals`, `isosurface`, `reducepatch`, `reducevolume`, `smooth3`

"Volume Visualization" on page 1-102 for related functions

Purpose

Sum of array elements

Syntax

```
B = sum(A)
B = sum(A,dim)
B = sum(..., 'double')
B = sum(..., dim,'double')
B = sum(..., 'native')
B = sum(..., dim,'native')
```

Description

`B = sum(A)` returns sums along different dimensions of an array.

If `A` is a vector, `sum(A)` returns the sum of the elements.

If `A` is a matrix, `sum(A)` treats the columns of `A` as vectors, returning a row vector of the sums of each column.

If `A` is a multidimensional array, `sum(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = sum(A,dim)` sums along the dimension of `A` specified by scalar `dim`. The `dim` input is an integer value from 1 to `N`, where `N` is the number of dimensions in `A`. Set `dim` to 1 to compute the sum of each column, 2 to sum rows, etc.

`B = sum(..., 'double')` and `B = sum(..., dim,'double')` performs additions in double-precision and return an answer of type `double`, even if `A` has data type `single` or an integer data type. This is the default for integer data types.

`B = sum(..., 'native')` and `B = sum(..., dim,'native')` performs additions in the native data type of `A` and return an answer of the same data type. This is the default for `single` and `double`.

Remarks

`sum(diag(X))` is the `trace` of `X`.

Examples

The magic square of order 3 is

```
M = magic(3)
M =
```

```
     8     1     6
     3     5     7
     4     9     2
```

This is called a magic square because the sums of the elements in each column are the same.

```
sum(M) =
     15    15    15
```

as are the sums of the elements in each row, obtained either by:

- Transposing

```
sum(M') =
     15    15    15
```

- Using the `dim` argument

```
sum(M,1) =
     15
     15
     15
```

transposing:

Nondouble Data Type Support

This section describes the support of sum for data types other than `double`.

Data Type single

You can apply `sum` to an array of type `single` and MATLAB returns an answer of type `single`. For example,

```
sum(single([2 5 8]})

ans =

     15
```

```
class(ans)

ans =

single
```

Integer Data Types

When you apply `sum` to any of the following integer data types, MATLAB returns an answer of type double:

- `int8` and `uint8`
- `int16` and `uint16`
- `int32` and `uint32`

For example,

```
sum(single([2 5 8]});
class(ans)

ans =

single
```

If you want MATLAB to perform additions on an integer data type in the same integer type as the input, use the syntax

```
sum(int8([2 5 8], 'native');
class(ans)

ans =

int8
```

See Also

`accumarray`, `cumsum`, `diff`, `isfloat`, `prod`

sum (timeseries)

Purpose

Sum of `timeseries` data

Syntax

```
ts_sm = sum(ts)
ts_sm = sum(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_sm = sum(ts)` returns the sum of the time-series data. When `ts.Data` is a vector, ts_sm is the sum of `ts.Data` values. When `ts.Data` is a matrix, ts_sm is a row vector containing the sum of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `sum` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_sm = sum(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

1. Load a 24-by-3 data array.

   ```
   load count.dat
   ```

2. Create a `timeseries` object with 24 time values.

   ```
   count_ts = timeseries(count,1:24,'Name','CountPerSecond')
   ```

3. Calculate the sum of each data column for this `timeseries` object.

   ```
   sum(count_ts)
   ```

```
ans =

          768         1117         1574
```

The sum is calculated independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `mean (timeseries)`, `median (timeseries)`, `std (timeseries)`, `var (timeseries)`, `timeseries`

superiorto

Purpose

Establish superior class relationship

Syntax

```
superiorto('class1', 'class2', ...)
```

Description

The `superiorto` function establishes a hierarchy that determines the order in which MATLAB calls object methods.

`superiorto('class1', 'class2', ...)` invoked within a class constructor method (say `myclass.m`) indicates that `myclass`'s method should be invoked if a function is called with an object of class `myclass` and one or more objects of class `class1`, `class2`, and so on.

Remarks

Suppose `A` is of class `'class_a'`, `B` is of class `'class_b'` and `C` is of class `'class_c'`. Also suppose the constructor `class_c.m` contains the statement `superiorto('class_a')`. Then `e = fun(a,c)` or `e = fun(c,a)` invokes `class_c/fun`.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So `fun(b,c)` calls `class_b/fun`, while `fun(c,b)` calls `class_c/fun`.

See Also

`inferiorto`

Purpose

Open MathWorks Technical Support Web page

Syntax

```
support
```

Description

`support` opens the MathWorks Technical Support Web page, `http://www.mathworks.com/support`, in the MATLAB Web browser.

This Web page contains resources including

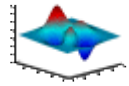
- A search engine, including an option for solutions to common problems
- Information about installation and licensing
- A patch archive for bug fixes you can download
- Other useful resources

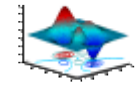
See Also

`doc`, `web`

Purpose

3-D shaded surface plot





GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
surf(Z)
surf(Z,C)
surf(X,Y,Z)
surf(X,Y,Z,C)
surf(...,'PropertyName',PropertyValue)
surf(axes_handles,...)
surfc(...)
h = surf(...)
hsurface = surf('v6',...)
```

Description

Use `surf` and `surfc` to view mathematical functions over a rectangular region. `surf` and `surfc` create colored parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `Z` or `C`.

`surf(Z)` creates a a three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data as well as surface height, so color is proportional to surface height.

`surf(Z,C)` plots the height of `Z`, a single-valued function defined over a geometrically rectangular grid, and uses matrix C, assumed to be the same size as Z, to color the surface.

`surf(X,Y,Z)` creates a shaded surface using `Z` for the color data as well as surface height. `X` and `Y` are vectors or matrices defining the `x` and `y` components of a surface. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, the vertices of the surface faces are *(X(j), Y(i), Z(i,j))* triples.

`surf(X,Y,Z,C)` creates a shaded surface, with color defined by `C`. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surf(...,'PropertyName',PropertyValue)` specifies surface `properties` along with the data.

`surf(axes_handles,...)` and surfc(axes_handles,...) plot into the axes with handle `axes_handle` instead of the current axes (`gca`).

`surfc(...)` draws a contour plot beneath the surface.

`h = surf(...)` and h = surfc(...) return a handle to a surfaceplot graphics object.

Backward-Compatible Version

`hsurface = surf('v6',...)` and hsurface = surfc('v6',...) return the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

`surf` and `surfc` do not accept complex inputs.

Algorithm

Abstractly, a parametric surface is parameterized by two independent variables, `i` and `j`, which vary continuously over a rectangle; for example, $1 \le i \le m$ and $1 \le j \le n$. The three functions `x(i,j)`, `y(i,j)`,

and `z(i,j)` specify the surface. When `i` and `j` are integer values, they define a rectangular grid with integer grid points. The functions `x(i,j)`, `y(i,j)`, and `z(i,j)` become three `m`-by-`n` matrices, `X`, `Y`, and `Z`. Surface color is a fourth function, `c(i,j)`, denoted by matrix `C`.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
          i-1,j
            |
  i,j-1 - i,j - i,j+1
            |
          i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, `[X(:) Y(:) Z(:)]` returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

Surface color can be specified in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of `x` and `y`. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The `shading` function sets the shading. If the shading is `interp`, `C` must be the same size as `X`, `Y`, and `Z`; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is `faceted` (the default) or `flat`, `C(i,j)` specifies the constant color in the surface patch:

```
  (i,j)   -   (i,j+1)
    |   C(i,j)  |
(i+1,j)  -  (i+1,j+1)
```

In this case, C can be the same size as X, Y, and Z and its last row and column are ignored. Alternatively, its row and column dimensions can be one less than those of X, Y, and Z.

The surf and surfc functions specify the viewpoint using view(3).

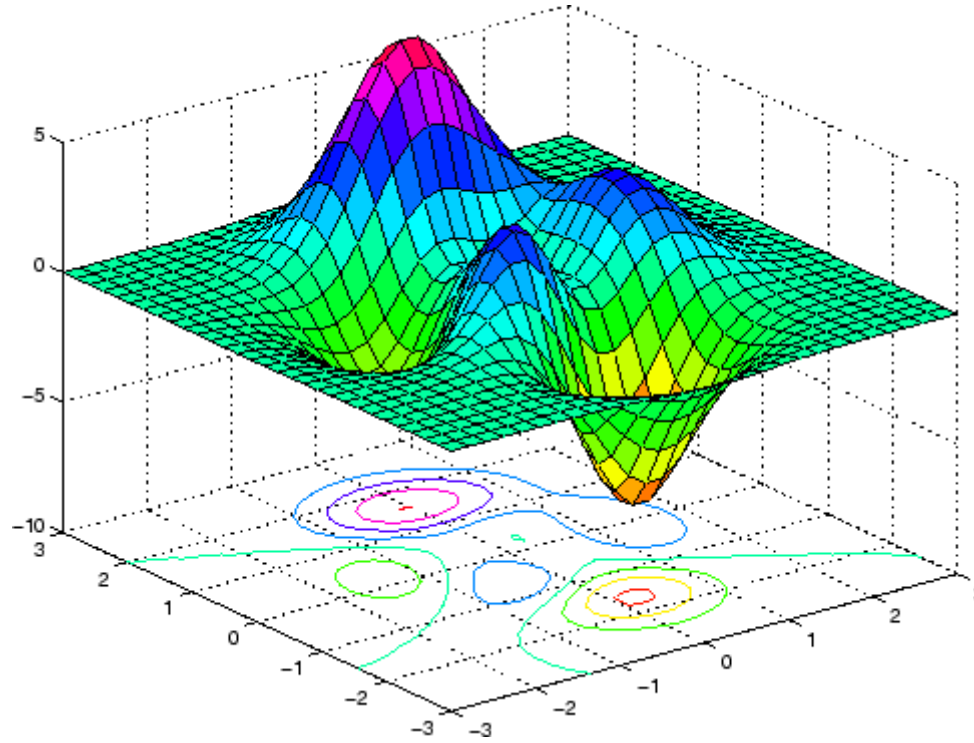
The range of X, Y, and Z or the current setting of the axes XLimMode, YLimMode, and ZLimMode properties (also set by the axis function) determines the axis labels.

The range of C or the current setting of the axes CLim and CLimMode properties (also set by the caxis function) determines the color scaling. The scaled color values are used as indices into the current colormap.

Examples

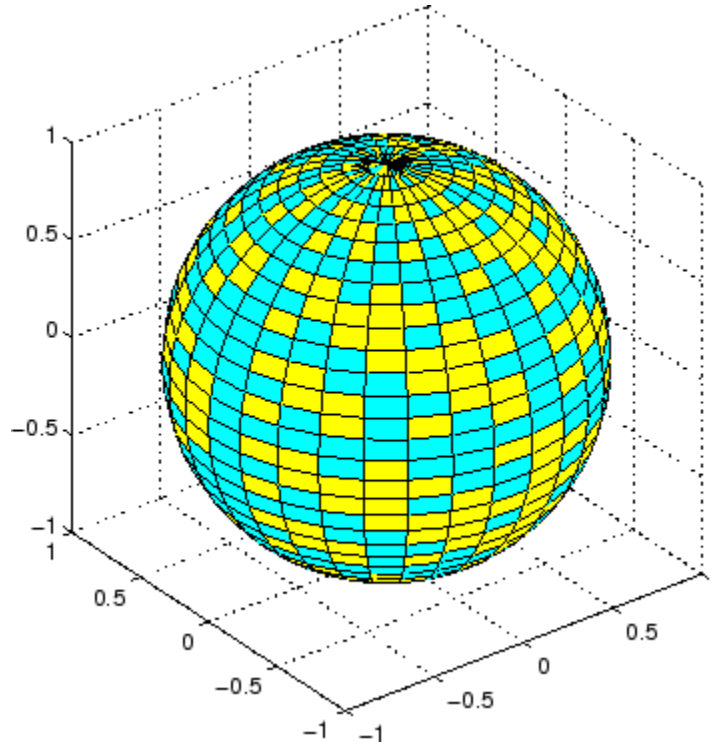
Display a surfaceplot and contour plot of the peaks surface.

```
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
```



Color a sphere with the pattern of +1s and -1s in a Hadamard matrix.

```
k = 5;
n = 2^k-1;
[x,y,z] = sphere(n);
c = hadamard(2^k);
surf(x,y,z,c);
colormap([1  1  0; 0  1  1])
axis equal
```



See Also

`axis`, `caxis`, `colormap`, `contour`, `delaunay`, `imagesc`, `mesh`, `pcolor`, `shading`, `trisurf`, `view`

Properties for surfaceplot graphics objects

“Creating Surfaces and Meshes” on page 1-97 for related functions

Representing a Matrix as a Surface for more examples

Coloring Mesh and Surface Plots for information about how to control the coloring of surfaces

surf2patch

Purpose

Convert surface data to patch data

Syntax

```
fvc = surf2patch(Z)
fvc = surf2patch(Z,C)
fvc = surf2patch(X,Y,Z)
fvc = surf2patch(X,Y,Z,C)
fvc = surf2patch(...,'triangles')
[f,v,c] = surf2patch(...)
```

Description

`fvc = surf2patch(h)`

converts the geometry and color data from the `surface` object identified by the handle `h` into patch format and returns the face, vertex, and color data in the struct `fvc`. You can pass this struct directly to the `patch` command.

`fvc = surf2patch(Z)` calculates the patch data from the surface's `ZData` matrix `Z`.

`fvc = surf2patch(Z,C)` calculates the patch data from the surface's `ZData` and `CData` matrices `Z` and `C`.

`fvc = surf2patch(X,Y,Z)` calculates the patch data from the surface's `XData`, `YDataYData`, and `ZData` matrices `X`, `Y`, and `Z`.

`fvc = surf2patch(X,Y,Z,C)` calculates the patch data from the surface's `XData`, `YData`, `ZData`, and `CData` matrices `X`, `Y`, `Z`, and `C`.

`fvc = surf2patch(...,'triangles')` creates triangular faces instead of the quadrilaterals that compose surfaces.

`[f,v,c] = surf2patch(...)` returns the face, vertex, and color data in the three arrays `f`, `v`, and `c` instead of a struct.

Examples

The first example uses the `sphere` command to generate the `XData`, `YData`, and `ZData` of a surface, which is then converted to a patch. Note that the `ZData` (`z`) is passed to `surf2patch` as both the third and fourth arguments — the third argument is the `ZData` and the fourth argument is taken as the `CData`. This is because the `patch` command does not

automatically use the *z*-coordinate data for the color data, as does the `surface` command.

Also, because `patch` is a low-level command, you must set the `view` to 3-D and `shading` to `faceted` to produce the same results produced by the `surf` command.

```
[x y z] = sphere;
patch(surf2patch(x,y,z,z));
shading faceted; view(3)
```

In the second example `surf2patch` calculates face, vertex, and color data from a surface whose handle has been passed as an argument.

```
s = surf(peaks);
pause
patch(surf2patch(s));
delete(s)
shading faceted; view(3)
```

See Also

`patch`, `reducepatch`, `shrinkfaces`, `surface`, `surf`

"Volume Visualization" on page 1-102 for related functions

surface

Purpose

Create surface object

Syntax

```
surface(Z)
surface(Z,C)
surface(X,Y,Z)
surface(X,Y,Z,C)
surface(x,y,Z)
surface(...'PropertyName',PropertyValue,...)
h = surface(...)
```

Description

`surface` is the low-level function for creating surface graphics objects. Surfaces are plots of matrix data created using the row and column indices of each element as the *x*- and *y*-coordinates and the value of each element as the *z*-coordinate.

`surface(Z)` plots the surface specified by the matrix `Z`. Here, `Z` is a single-valued function, defined over a geometrically rectangular grid.

`surface(Z,C)` plots the surface specified by `Z` and colors it according to the data in `C` (see "Examples").

`surface(X,Y,Z)` uses `C = Z`, so color is proportional to surface height above the *x-y* plane.

`surface(X,Y,Z,C)` plots the parametric surface specified by `X`, `Y`, and `Z`, with color specified by `C`.

`surface(x,y,Z)`, surface(x,y,Z,C) replaces the first two matrix arguments with vectors and must have `length(x) = n` and `length(y) = m` where `[m,n] = size(Z)`. In this case, the vertices of the surface facets are the triples `(x(j),y(i),Z(i,j))`. Note that `x` corresponds to the columns of `Z` and `y` corresponds to the rows of `Z`. For a complete discussion of parametric surfaces, see the `surf` function.

`surface(...'PropertyName',PropertyValue,...)` follows the `X`, `Y`, `Z`, and `C` arguments with property name/property value pairs to specify additional surface properties.

`h = surface(...)` returns a handle to the created surface object.

Remarks

surface does not respect the settings of the figure and axes NextPlot properties. It simply adds the surface object to the current axes.

If you do not specify separate color data (C), MATLAB uses the matrix (Z) to determine the coloring of the surface. In this case, color is proportional to values of Z. You can specify a separate matrix to color the surface independently of the data defining the area of the surface.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see set and get for examples of how to specify these data types).

surface provides convenience forms that allow you to omit the property name for the XData, YData, ZData, and CData properties. For example,

```
surface('XData',X,'YData',Y,'ZData',Z,'CData',C)
```

is equivalent to

```
surface(X,Y,Z,C)
```

When you specify only a single matrix input argument,

```
surface(Z)
```

MATLAB assigns the data properties as if you specified

```
surface('XData',[1:size(Z,2)],...
   'YData',[1:size(Z,1)],...
   'ZData',Z,...
   'CData',Z)
```

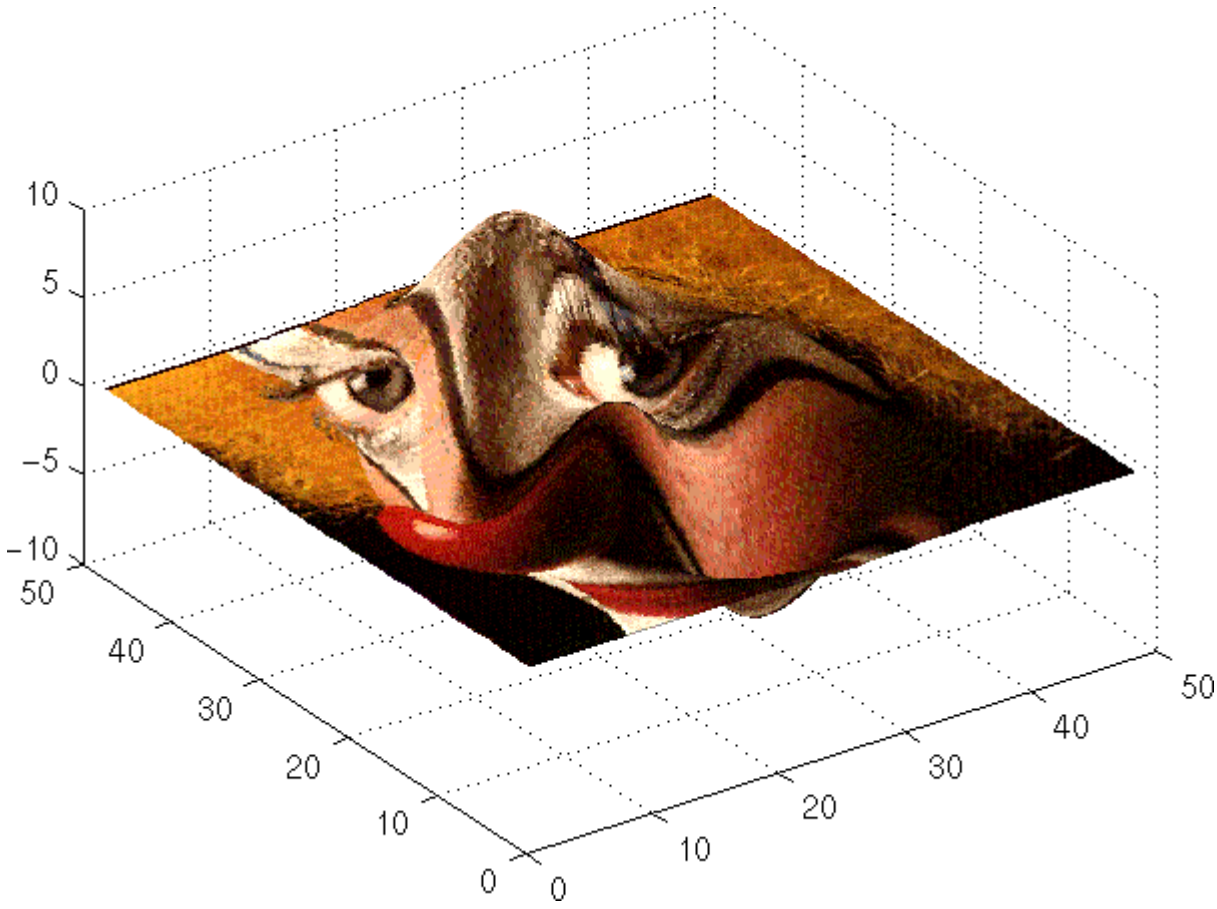
The axis, caxis, colormap, hold, shading, and view commands set graphics properties that affect surfaces. You can also set and query surface property values after creating them using the set and get commands.

Example

This example creates a surface using the peaks M-file to generate the data, and colors it using the clown image. The ZData is a 49-by-49

element matrix, while the `CData` is a 200-by-320 matrix. You must set the surface's `FaceColor` to `texturemap` to use `ZData` and `CData` of different dimensions.

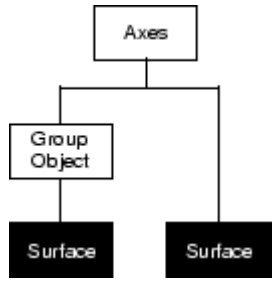
```
load clown
surface(peaks,flipud(X),...
   'FaceColor','texturemap',...
   'EdgeColor','none',...
   'CDataMapping','direct')
colormap(map)
view(-35,45)
```



Note the use of the `surface(Z,C)` convenience form combined with property name/property value pairs.

Since the clown data (`X`) is typically viewed with the `image` command, which MATLAB normally displays with `'ij'` axis numbering and `direct` `CDataMapping`, this example reverses the data in the vertical direction using `flipud` and sets the `CDataMapping` property to `direct`.

Object Hierarchy



Setting Default Properties

You can set default surface properties on the axes, figure, and root levels:

```
set(0,'DefaultSurfaceProperty',PropertyValue...)
set(gcf,'DefaultSurfaceProperty',PropertyValue...)
set(gca,'DefaultSurfaceProperty',PropertyValue...)
```

where *Property* is the name of the surface property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.

See Also

`ColorSpec`, `patch`, `pcolor`, `surf`

Representing a Matrix as a Surface for examples

"Creating Surfaces and Meshes" on page 1-97 and "Object Creation Functions" on page 1-94 for related functions

`Surface Properties` for property descriptions

Purpose

Surface properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see Setting Default Property Values.

See "Core Graphics Objects" for general information about this type of object.

Surface Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces { } enclose default values.

`AlphaData`

m-by-n matrix of `double` or `uint8`

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class double or uint8.

MATLAB determines the transparency in one of three ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)
- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`

```
none | direct | {scaled}
```

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. This property can be any of the following:

- `none` — The transparency values of `AlphaData` are between 0 and 1 or are clamped to this range (the default).
- `scaled` — Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` — use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength`

scalar >= 0 and <= 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientLightColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the surface `DiffuseStrength` and `SpecularStrength` properties.

`Annotation`

`hg.Annotation` object Read Only

Control the display of surface objects in legends. The `Annotation` property enables you to specify whether this surface object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the surface object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this surface object in a legend (default)
`off`	Do not include this surface object in a legend
`children`	Same as `on` because surface objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BackFaceLighting`
`unlit | lit | reverselit`

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera.

- `unlit` — Face is not lit.
- `lit` — Face is lit in normal way.
- `reverselit` — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See "Back Face Lighting" for an example.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then

interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the surface object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property). For example, the following function takes different action depending on what type of selection was made:

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
```

```
            set(src,'Selected','on')
         case 'alt'
            disp('User did a control-click')
            set(src,'Selected','on')
            set(src,'SelectionHighlight','off')
      end
   end
```

Suppose `h` is the handle of a surface object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`CData`

matrix (of type `double`)

Vertex colors. A matrix containing values that specify the color at every point in `ZData`.

Mapping CData to a Colormap

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see `caxis`) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property.

CData as True Color

True color defines an RGB value for each vertex. If the coordinate data (`XData`, for example) are contained in *m*-by-*n* matrices, then `CData` must be an *m*-by-*n*-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

Texturemapping the Surface FaceColor

If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`, but must be of type `double` or `uint8`. In this case, MATLAB maps `CData` to conform to the surface defined by `ZData`.

`CDataMapping`

`{scaled} | direct`

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the surface. (If you use true color specification for `CData`, this property has no effect.)

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`Children`

matrix of handles

Always the empty matrix; surface objects have no children.

`Clipping`

`{on} | off`

Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the surface that is outside the axes rectangle.

`CreateFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during object creation. This property defines a callback function that executes when MATLAB creates a surface object. You must define this property as a default value for surfaces or set the `CreateFcn` property during object creation.

For example, the following statement creates a surface (assuming `x`, `y`, `z`, and `c` are defined), and executes the function referenced by the function handle `@myCreateFcn`.

```
surface(x,y,z,c,'CreateFcn',@myCreateFcn)
```

MATLAB executes this routine after setting all surface properties. Setting this property on an existing surface object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Delete surface callback function. A callback function that executes when you delete the surface object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
```

```
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DiffuseStrength`

scalar >= 0 and <= 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the surface object. See the `AmbientStrength` and `SpecularStrength` properties.

`DisplayName`

string (default is empty string)

String used by legend for this surface object. The `legend` function uses the string defined by the `DisplayName` property to label this surface object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this surface object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EdgeAlpha`

`{scalar = 1} | flat | interp`

Transparency of the surface edges. This property can be any of the following:

- `scalar` — A single non-Nan scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.
- `flat` — The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.
- `interp` — Linear interpolation of the alpha data (`AlphaData`) values at each vertex determines the transparency of the edge.

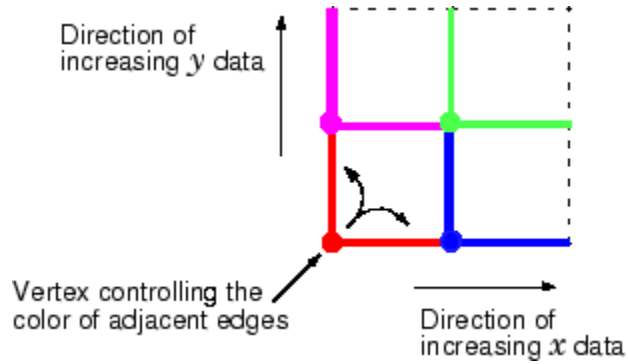
Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

`EdgeColor`

`{ColorSpec} | none | flat | interp`

Color of the surface edge. This property determines how MATLAB colors the edges of the individual faces that make up the surface:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default `EdgeColor` is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge.



- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

`EdgeLighting`

`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of `light` objects on surface edges. Choices are

- `none` — Lights do not affect the edges of this object.

- `flat` — The effect of light objects is uniform across each edge of the surface.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`EraseMode`

```
{normal} | none | xor | background
```

Erase mode. This property controls the technique MATLAB uses to draw and erase surface objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the surface when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode` `none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the surface by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the surface does not damage the color of the objects behind it. However, surface color depends on the color of the screen behind it and is correctly colored only when over the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`.

- `background` — Erase the surface by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased object, but surface objects are always properly colored.

  Printing with Nonnormal Erase Modes

  MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

  You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing nonnormal mode objects.

`FaceAlpha`

```
{scalar = 1} | flat | interp | texturemap
```

Transparency of the surface faces. This property can be any of the following:

- `scalar` — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the faces of the object. `1` (the default) means fully opaque and `0` means completely transparent (invisible).
- `flat` — The values of the alpha data (`AlphaData`) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.
- `interp` — Bilinear interpolation of the alpha data (`AlphaData`) at each vertex determines the transparency of each face.
- `texturemap` — Use transparency for the texture map.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `FaceAlpha`.

`FaceColor`

`ColorSpec | none | {flat} | interp | texturemap`

Color of the surface face. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The values of `CData` determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.
- `interp` — Bilinear interpolation of the values at each vertex (the `CData`) determines the coloring of each face.
- `texturemap` — Texture map the `CData` to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example.)

`FaceLighting`

`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of `light` objects on the surface. Choices are

- `none` — Lights do not affect the faces of this object.
- `flat` — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.

- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. This property is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback routine invokes a function that could potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

`HitTest`

```
{on} | off
```

Selectable by mouse click. `HitTest` determines if the surface can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the surface. If `HitTest` is `off`, clicking on the surface selects the object below it (which may be the axes containing it).

`Interruptible`

```
{on} | off
```

Callback routine interruption mode. The `Interruptible` property controls whether a surface callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

`LineStyle`

`{-} | -- | : | -. | none`

Edge line type. This property determines the line style used to draw surface edges. The available line styles are shown in this table.

Symbol	Line Style
	Solid line (default)
	Dashed line
`:`	Dotted line
`.`	Dash-dot line
`none`	No line

`LineWidth`

scalar

Edge line width. The width of the lines in points used to draw surface edges. The default width is 0.5 points (1 point = 1/72 inch).

`Marker`

marker symbol (see table)

Marker symbol. The `Marker` property specifies symbols that are displayed at vertices. You can set values for the `Marker` property independently from the `LineStyle` property.

You can specify these markers.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle
`*`	Asterisk

Marker Specifier	Description
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Five-pointed star (pentagram)
`h`	Six-pointed star (hexagram)
`none`	No marker (default)

`MarkerEdgeColor`
`none | {auto} | flat | ColorSpec`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `none` specifies no color, which makes nonfilled markers invisible.
- `auto` uses the same color as the `EdgeColor` property.
- `flat` uses the `CData` value of the vertex to determine the color of the maker edge.
- `ColorSpec` defines a single color to use for the edge (see `ColorSpec` for more information).

`MarkerFaceColor`
`{none} | auto | flat | ColorSpec`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `none` makes the interior of the marker transparent, allowing the background to show through.
- `auto` uses the axes `Color` for the marker face color.
- `flat` uses the `CData` value of the vertex to determine the color of the face.
- `ColorSpec` defines a single color to use for all markers on the surface (see `ColorSpec` for more information).

`MarkerSize`
size in points

Marker size. A scalar specifying the marker size, in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 the specified marker size.

`MeshStyle`
`{both} | row | column`

Row and column lines. This property specifies whether to draw all edge lines or just row or column edge lines.

- `both` draws edges for both rows and columns.
- `row` draws row edges only.
- `column` draws column edges only.

`NormalMode`
`{auto} | manual`

MATLAB generated or user-specified normal vectors. When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB

sets this property to `manual` and does not generate its own data. See also the `VertexNormals` property.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of surface object. This property contains the handle of the surface object's parent. The parent of a surface object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When this property is `on`, MATLAB displays a dashed bounding box around the surface if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`SpecularColorReflectance`
scalar in the range 0 to 1

Color of specularly reflected light. When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the light object `Color` property). The proportions vary linearly for values in between.

`SpecularExponent`
scalar >= 1

Harshness of specular reflection. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

`SpecularStrength`
scalar >= 0 and <= 1

Intensity of specular light. This property sets the intensity of the specular component of the light falling on the surface. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surface object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`
string (read only)

Class of the graphics object. The class of the graphics object. For surface objects, `Type` is always the string `'surface'`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the surface. Assign this property the handle of a uicontextmenu object created in the same figure as the surface. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the surface.

`UserData`
matrix

User-specified data. Any matrix you want to associate with the surface object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`VertexNormals`
vector or matrix

Surface normal vectors. This property contains the vertex normals for the surface. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

`Visible`
`{on} | off`

Surface object visibility. By default, all surfaces are visible. When set to `off`, the surface is not visible, but still exists, and you can query and set its properties.

`XData`
vector or matrix

X-coordinates. The *x*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of columns as `ZData`.

`YData`
vector or matrix

Y-coordinates. The *y*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of rows as `ZData`.

`ZData`

matrix

Z-coordinates. The *z*-position of the surfaceplot data points. See the Description section for more information.

Surfaceplot Properties

Purpose

Define surfaceplot properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

Note that you cannot define default properties for surfaceplot objects.

See Plot Objects for information on surfaceplot objects.

Surfaceplot Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces { } enclose default values.

`AlphaData`

m-by-n matrix of `double` or `uint8`

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class double or uint8.

MATLAB determines the transparency in one of three ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)
- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`

`{none} | direct| scaled`

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. It can be any of the following:

- `none` — The transparency values of `AlphaData` are between 0 and 1 or are clamped to this range (the default).
- `scaled` — Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` — Use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength`

scalar >= 0 and <= 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientLightColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of `light` objects. See the surfaceplot `DiffuseStrength` and `SpecularStrength` properties.

`Annotation`

`hg.Annotation` object Read Only

Control the display of surfaceplot objects in legends. The `Annotation` property enables you to specify whether this surfaceplot object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the surfaceplot object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this surfaceplot object in a legend (default)
`off`	Do not include this surfaceplot object in a legend
`children`	Same as `on` because surfaceplot objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

See “Controlling Legends” for more information and examples.

`BackFaceLighting`
`unlit | lit | reverselit`

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera.

- `unlit` — Face is not lit.
- `lit` — Face is lit in normal way.
- `reverselit` — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See Back Face Lighting for an example.

`BeingDeleted`
`on | {off}` Read Only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function

executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`
`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`CData`
matrix

Vertex colors. A matrix containing values that specify the color at every point in `ZData`. If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`. In this case, MATLAB maps `CData` to conform to the surfaceplot defined by `ZData`.

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see `caxis`) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property. Note that any non-texture data passed as an input argument must be of type double.

True color defines an RGB value for each vertex. If the coordinate data (`XData`, for example) are contained in *m*-by-*n* matrices, then `CData` must be an *m*-by-*n*-by-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

`CDataMapping`
`{scaled} | direct`

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the surfaceplot. (If you use true color specification for `CData`, this property has no effect.)

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging

from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`CDataMode`

{auto} | manual

Use automatic or user-specified color data values. If you specify `CData`, MATLAB sets this property to `manual` and uses the `CData` values to color the surfaceplot.

If you set `CDataMode` to `auto` after having specified `CData`, MATLAB resets the color data of the surfaceplot to that defined by `ZData`, overwriting any previous values for `CData`.

`CDataSource`

string (MATLAB variable)

Link CData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `CData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `CData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`Children`
matrix of handles

Always the empty matrix; surfaceplot objects have no children.

`Clipping`
`{on} | off`

Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the surfaceplot that is outside the axes rectangle.

`CreateFcn`
string or function handle

Callback routine executed during object creation. This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`
string or function handle

Callback executed during object deletion. A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DiffuseStrength`
scalar >= 0 and <= 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the object. See the `AmbientStrength` and `SpecularStrength` properties.

`DisplayName`
string (default is empty string)

String used by legend for this surfaceplot object. The `legend` function uses the string defined by the `DisplayName` property to label this surfaceplot object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this surfaceplot object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more examples.

`EdgeAlpha`
`{scalar = 1} | flat | interp`

Transparency of the patch and surface edges. This property can be any of the following:

- `scalar` — A single non-Nan scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.
- `flat` — The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.

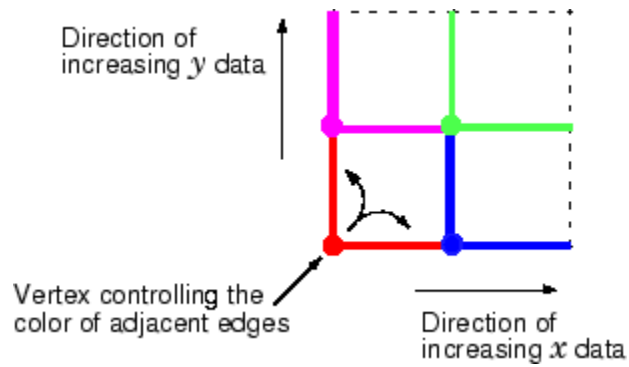
- `interp` — Linear interpolation of the alpha data (`AlphaData`) values at each vertex determines the transparency of the edge.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

`EdgeColor`
`{ColorSpec} | none | flat | interp`

Color of the surfaceplot edge. This property determines how MATLAB colors the edges of the individual faces that make up the surface:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default `EdgeColor` is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge.



- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

`EdgeLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on surfaceplot edges. Choices are

- `none` — Lights do not affect the edges of this object.
- `flat` — The effect of light objects is uniform across each edge of the surface.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing

with `EraseMode` `none`, you cannot print these objects because MATLAB stores no information about their former locations.

- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceAlpha`

```
{scalar = 1} | flat | interp | texturemap
```

Transparency of the surfaceplot faces. This property can be any of the following:

- `scalar` — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the faces of the object. `1` (the default) means fully opaque and `0` means completely transparent (invisible).
- `flat` — The values of the alpha data (`AlphaData`) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.
- `interp` — Bilinear interpolation of the alpha data (`AlphaData`) at each vertex determines the transparency of each face.
- `texturemap` — Use transparency for the texture map.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `FaceAlpha`.

`FaceColor`

`ColorSpec | none | {flat} | interp`

Color of the surfaceplot face. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The values of `CData` determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.
- `interp` — Bilinear interpolation of the values at each vertex (the `CData`) determines the coloring of each face.

- `texturemap` — Texture map the `Cdata` to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example for `surface`.)

`FaceLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on the surface. Choices are

- `none` — Lights do not affect the faces of this object.
- `flat` — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is `on`.
- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to

protect GUIs from command-line users, while allowing callback routines to have access to object handles.

- `off` — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`Interruptible`

`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether an object's callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

`LineStyle`

`{-} | -- | : | -. | none`

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.

Specifier String	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
`none`	No line

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = $^1/_{72}$ inch). The default `LineWidth` is 0.5 points.

`Marker`
character (see table)

Marker symbol. The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

Marker Specifier	Description
`+`	Plus sign
`o`	Circle

Marker Specifier	Description
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Five-pointed star (pentagram)
h	Six-pointed star (hexagram)
none	No marker (default)

`MarkerEdgeColor`
`none | {auto} | flat | ColorSpec`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `none` specifies no color, which makes nonfilled markers invisible.
- `auto` uses the same color as the `EdgeColor` property.
- `flat` uses the `CData` value of the vertex to determine the color of the maker edge.
- `ColorSpec` defines a single color to use for the edge (see `ColorSpec` for more information).

`MarkerFaceColor`
`{none} | auto | flat | ColorSpec`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `none` makes the interior of the marker transparent, allowing the background to show through.
- `auto` uses the axes `Color` for the marker face color.
- `flat` uses the `CData` value of the vertex to determine the color of the face.
- `ColorSpec` defines a single color to use for all markers on the surfaceplot (see `ColorSpec` for more information).

`MarkerSize`
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`MeshStyle`
`{both} | row | column`

Row and column lines. This property specifies whether to draw all edge lines or just row or column edge lines.

- `both` draws edges for both rows and columns.
- `row` draws row edges only.
- `column` draws column edges only.

`NormalMode`
`{auto} | manual`

MATLAB generated or user-specified normal vectors. When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB

sets this property to `manual` and does not generate its own data. See also the `VertexNormals` property.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`SpecularColorReflectance`
scalar in the range 0 to 1

Color of specularly reflected light. When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source.

When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the light object `Color` property). The proportions vary linearly for values in between.

`SpecularExponent`

scalar >= 1

Harshness of specular reflection. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

`SpecularStrength`

scalar >= 0 and <= 1

Intensity of specular light. This property sets the intensity of the specular component of the light falling on the surface. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surfaceplot object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read only)

Class of the graphics object. The class of the graphics object. For surfaceplot objects, `Type` is always the string `'surface'`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with this object. Assign this property the handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

`UserData`
array

User-specified data. This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

`VertexNormals`
vector or matrix

Surfaceplot normal vectors. This property contains the vertex normals for the surfaceplot. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

`Visible`
`{on} | off`

Visibility of this object and its children. By default, a new object's visibility is `on`. This means all children of the object are visible unless the child object's `Visible` property is set to `off`. Setting an object's `Visible` property to `off` prevents the object from being displayed. However, the object still exists and you can set and query its properties.

`XData`
vector or matrix

X-coordinates. The *x*-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of columns as `ZData`.

`XDataMode`
{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `x` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after having specified `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
string (MATLAB variable)

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `XData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

vector or matrix

Y-coordinates. The *y*-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of rows as `ZData`.

`YDataMode`

{auto} | manual

Use automatic or user-specified x-axis values. If you specify `XData`, MATLAB sets this property to `manual`.

If you set `YDataMode` to `auto` after having specified `YData`, MATLAB resets the *y*-axis ticks and *y*-tick labels to the row indices of the `ZData`, overwriting any previous values for `YData`.

`YDataSource`

string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`

matrix

Z-coordinates. The *z*-position of the surfaceplot data points. See the Description section for more information.

`ZDataSource`

string (MATLAB variable)

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `ZData`.

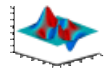
You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Surface plot with colormap-based lighting



GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
surfl(Z)
surfl(...,'light')
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)
```

Description

The `surfl` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surfl(Z)` and surfl(X,Y,Z) create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. `X`, `Y`, and `Z` are vectors or matrices that define the *x*, *y*, and *z* components of a surface.

`surfl(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surfl(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surfl(...,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from a surface to a light source. `s = [sx sy sz]` or `s = [azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`surfl(X,Y,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light,

diffuse reflection, specular reflection, and the specular shine coefficient. `k = [ka kd ks shine]` and defaults to `[.55,.6,.4,10]`.

`h = surfl(...)` returns a handle to a surface graphics object.

Remarks

`surfl` does not accept complex inputs.

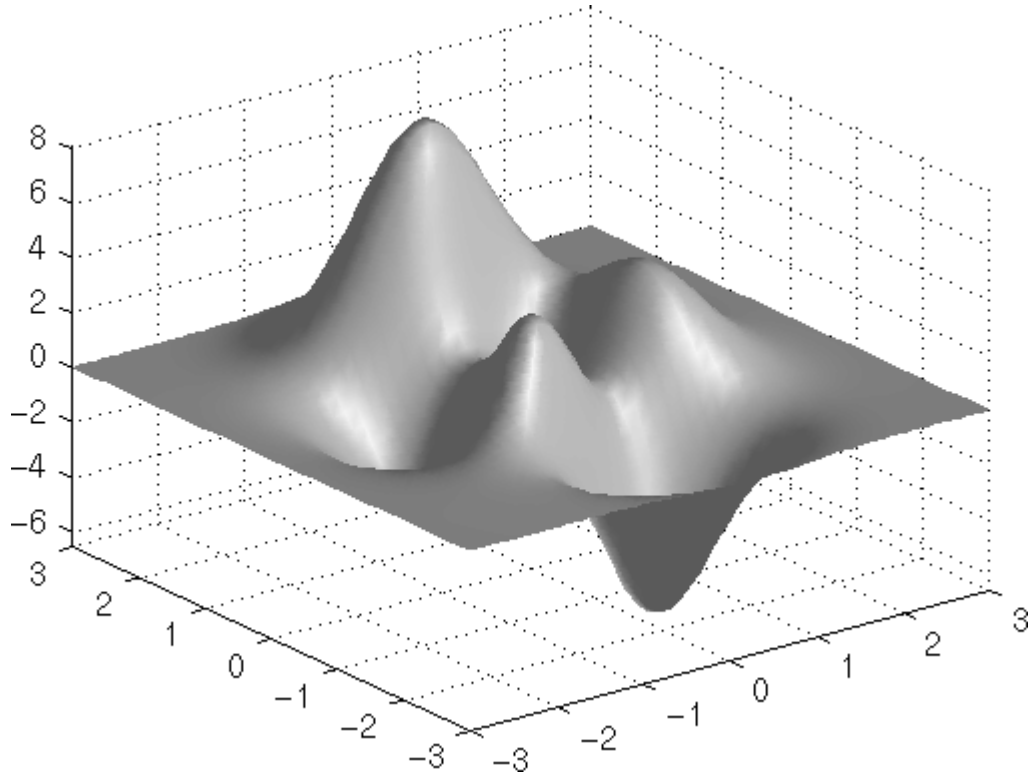
For smoother color transitions, use colormaps that have linear intensity variations (e.g., `gray`, `copper`, `bone`, `pink`).

The ordering of points in the `X`, `Y`, and `Z` matrices defines the inside and outside of parametric surfaces. If you want the opposite side of the surface to reflect the light source, use `surfl(X',Y',Z')`. Because of the way surface normal vectors are computed, `surfl` requires matrices that are at least 3-by-3.

Examples

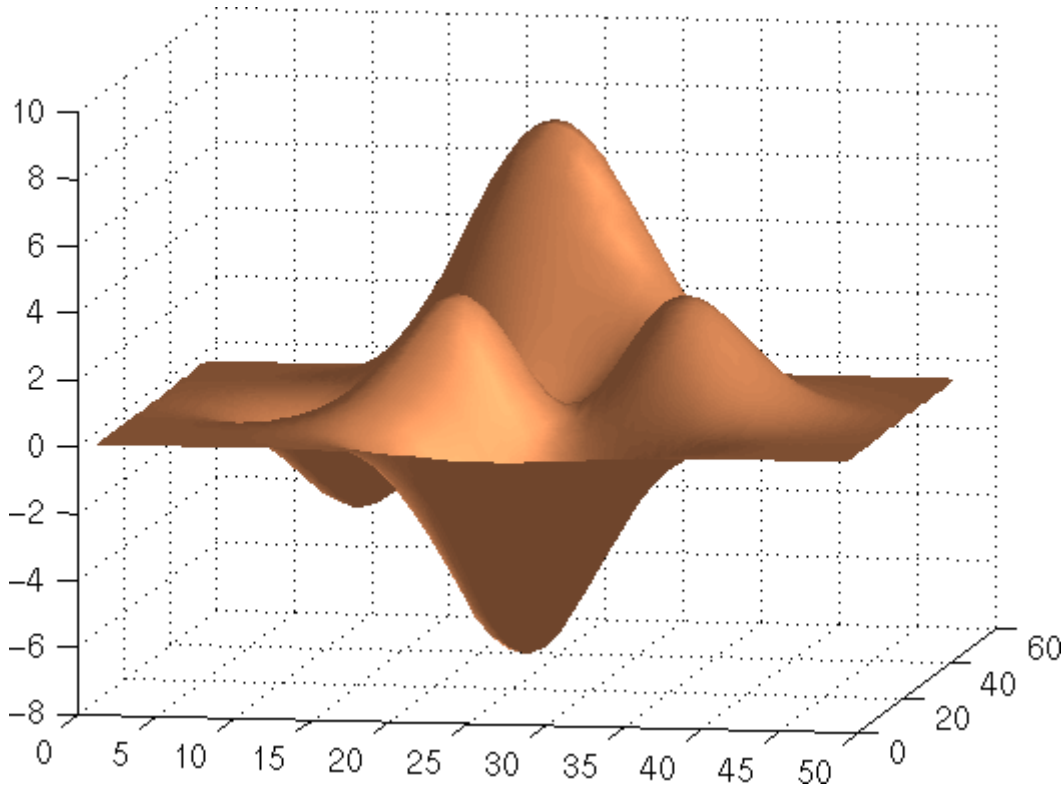
View `peaks` using colormap-based lighting.

```
[x,y] = meshgrid(-3:1/8:3);
z = peaks(x,y);
surfl(x,y,z);
shading interp
colormap(gray);
axis([-3  3  -3  3  -8  8])
```



To plot a lighted surface from a view direction other than the default,

```
view([10 10])
grid on
hold on
surfl(peaks)
shading interp
colormap copper
hold off
```



See Also

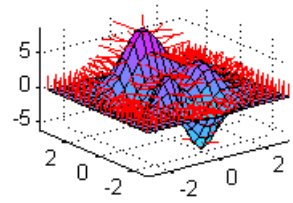
`colormap`, `shading`, `light`

"Creating Surfaces and Meshes" on page 1-97 for functions related to surfaces

"Lighting" on page 1-101 for functions related to lighting

Purpose

Compute and display 3-D surface normals



Syntax

```
surfnorm(Z)
surfnorm(X,Y,Z)
[Nx,Ny,Nz] = surfnorm(...)
```

Description

The `surfnorm` function computes surface normals for the surface defined by `X`, `Y`, and `Z`. The surface normals are unnormalized and valid at each vertex. Normals are not shown for surface elements that face away from the viewer.

`surfnorm(Z)` and `surfnorm(X,Y,Z)` plot a surface and its surface normals. `Z` is a matrix that defines the *z* component of the surface. `X` and `Y` are vectors or matrices that define the *x* and *y* components of the surface.

`[Nx,Ny,Nz] = surfnorm(...)` returns the components of the three-dimensional surface normals for the surface.

Remarks

`surfnorm` does not accept complex inputs.

The direction of the normals is reversed by calling `surfnorm` with transposed arguments:

```
surfnorm(X',Y',Z')
```

`surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.

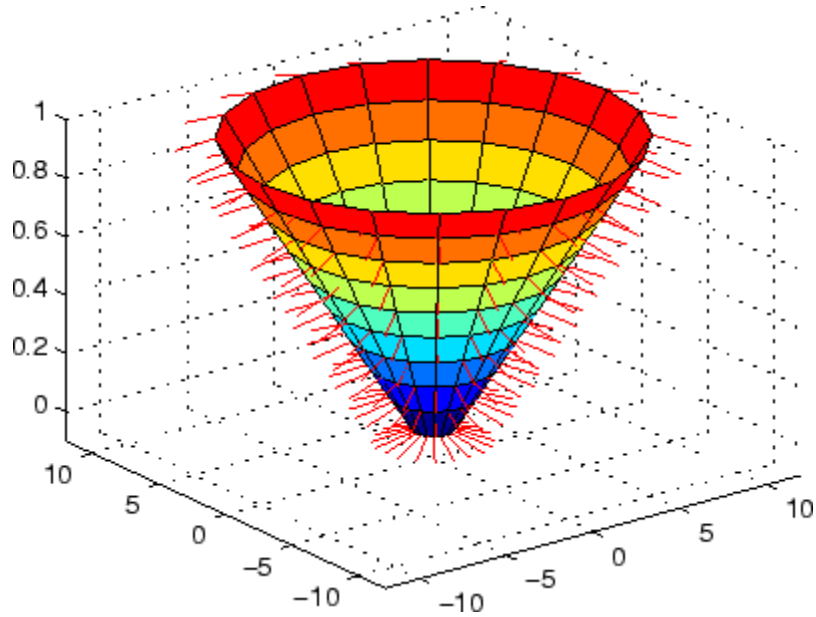
Algorithm

The surface normals are based on a bicubic fit of the data in `X`, `Y`, and `Z`. For each vertex, diagonal vectors are computed and crossed to form the normal.

Examples

Plot the normal vectors for a truncated cone.

```
[x,y,z] = cylinder(1:10);
surfnorm(x,y,z)
axis([-12 12 -12 12 -0.1 1])
```



See Also

`surf`, `quiver3`

"Colormaps" on page 1-99 for related functions

Purpose

Singular value decomposition

Syntax

```
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
[U,S,V] = svd(X,'econ')
```

Description

The `svd` command computes the matrix singular value decomposition.

`s = svd(X)` returns a vector of singular values.

`[U,S,V] = svd(X)` produces a diagonal matrix `S` of the same dimension as `X`, with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` so that `X = U*S*V'`.

`[U,S,V] = svd(X,0)` produces the "economy size" decomposition. If `X` is m-by-n with m > n, then `svd` computes only the first `n` columns of `U` and `S` is n-by-n.

`[U,S,V] = svd(X,'econ')` also produces the "economy size" decomposition. If `X` is m-by-n with m >= n, it is equivalent to `svd(X,0)`. For m < n, only the first m columns of `V` are computed and `S` is m-by-m.

Examples

For the matrix

```
X =
     1     2
     3     4
     5     6
     7     8
```

the statement

```
[U,S,V] = svd(X)
```

produces

```
U =
    -0.1525    -0.8226    -0.3945    -0.3800
```

```
    -0.3499   -0.4214    0.2428    0.8007
    -0.5474   -0.0201    0.6979   -0.4614
    -0.7448    0.3812   -0.5462    0.0407

S =
    14.2691         0
          0    0.6268
          0         0
          0         0

V =
    -0.6414    0.7672
    -0.7672   -0.6414
```

The economy size decomposition generated by

```
[U,S,V] = svd(X,0)
```

produces

```
U =
    -0.1525   -0.8226
    -0.3499   -0.4214
    -0.5474   -0.0201
    -0.7448    0.3812
S =
    14.2691         0
          0    0.6268
V =
    -0.6414    0.7672
    -0.7672   -0.6414
```

Algorithm

`svd` uses the LAPACK routines listed in the following table to compute the singular value decomposition.

	Real	Complex
X double	DGESVD	ZGESVD
X single	SGESVD	CGESVD

Diagnostics

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

```
Solution will not converge.
```

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (http://www.netlib.org/lapack/lug/lapack_lug.html), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Find singular values and vectors

Syntax

```
s = svds(A)
s = svds(A,k)
s = svds(A,k,sigma)
s = svds(A,k,'L')
s = svds(A,k,sigma,options)
[U,S,V] = svds(A,...)
[U,S,V,flag] = svds(A,...)
```

Description

`s = svds(A)` computes the six largest singular values and associated singular vectors of matrix `A`. If `A` is `m`-by-`n`, `svds(A)` manipulates eigenvalues and vectors returned by `eigs(B)`, where `B = [sparse(m,m) A; A' sparse(n,n)]`, to find a few singular values and vectors of `A`. The positive eigenvalues of the symmetric matrix `B` are the same as the singular values of `A`.

`s = svds(A,k)` computes the `k` largest singular values and associated singular vectors of matrix `A`.

`s = svds(A,k,sigma)` computes the `k` singular values closest to the scalar shift sigma. For example, `s = svds(A,k,0)` computes the `k` smallest singular values and associated singular vectors.

`s = svds(A,k,'L')` computes the `k` largest singular values (the default).

`s = svds(A,k,sigma,options)` sets some parameters (see `eigs`):

Option Structure Fields and Descriptions

Field name	Parameter	Default
`options.tol`	Convergence tolerance: `norm(AV-US,1)<=tol*norm(A,1)`	`1e-10`

Option Structure Fields and Descriptions (Continued)

Field name	Parameter	Default
`options.maxit`	Maximum number of iterations	`300`
`options.disp`	Number of values displayed each iteration	`0`

`[U,S,V] = svds(A,...)` returns three output arguments, and if `A` is `m`-by-`n`:

- `U` is `m`-by-`k` with orthonormal columns
- `S` is `k`-by-`k` diagonal
- `V` is `n`-by-`k` with orthonormal columns
- `U*S*V'` is the closest rank `k` approximation to `A`

`[U,S,V,flag] = svds(A,...)` returns a convergence flag. If `eigs` converged then `norn(A*V-U*S,1) <= tol*norm(A,1)` and `flag` is `0`. If `eigs` did not converge, then `flag` is `1`.

Note `svds` is best used to find a few singular values of a large, sparse matrix. To find all the singular values of such a matrix, `svd(full(A))` will usually perform better than `svds(A,min(size(A)))`.

Algorithm

`svds(A,k)` uses `eigs` to find the `k` largest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`.

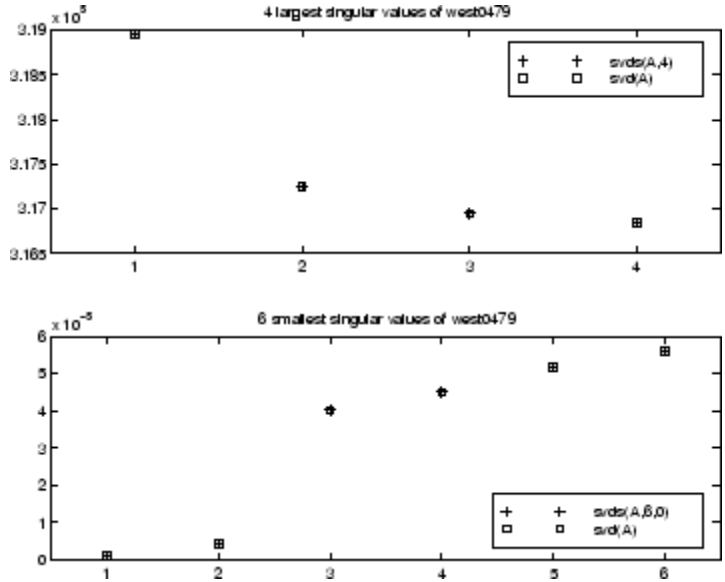
`svds(A,k,0)` uses `eigs` to find the `2k` smallest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`, and then selects the `k` positive eigenvalues and their eigenvectors.

Example

`west0479` is a real 479-by-479 sparse matrix. `svd` calculates all 479 singular values. `svds` picks out the largest and smallest singular values.

```
load west0479
s = svd(full(west0479))
sl = svds(west0479,4)
ss = svds(west0479,6,0)
```

These plots show some of the singular values of `west0479` as computed by `svd` and `svds`.



The largest singular value of `west0479` can be computed a few different ways:

```
svds(west0479,1) =
 3.189517598808622e+05
max(svd(full(west0479))) =
 3.18951759880862e+05
norm(full(west0479)) =
 3.189517598808623e+05
```

and estimated:

```
normest(west0479) =
 3.189385666549991e+05
```

See Also

svd, eigs

Purpose

Swap byte ordering

Syntax

```
Y = swapbytes(X)
```

Description

`Y = swapbytes(X)` reverses the byte ordering of each element in array `X`, converting little-endian values to big-endian (and vice versa). The input array must contain all full, noncomplex, numeric elements.

Examples

Example 1

Reverse the byte order for a scalar 32-bit value, changing hexadecimal `12345678` to `78563412`:

```
A = uint32(hex2dec('12345678'));

B = dec2hex(swapbytes(A))
B =
    78563412
```

Example 2

Reverse the byte order for each element of a 1-by-4 matrix:

```
X = uint16([0 1 128 65535])
X =
      0      1    128  65535

Y = swapbytes(X);
Y =
      0    256  32768  65535
```

Examining the output in hexadecimal notation shows the byte swapping:

```
format hex

X, Y
X =
   0000   0001   0080   ffff
```

```
Y =
   0000   0100   8000   ffff
```

Example 3

Create a three-dimensional array `A` of 16-bit integers and then swap the bytes of each element:

```
format hex

A = uint16(magic(3) * 150);
A(:,:,2) = A * 40;

A
A(:,:,1) =
   04b0   0096   0384
   01c2   02ee   041a
   0258   0546   012c
A(:,:,2) =
   bb80   1770   8ca0
   4650   7530   a410
   5dc0   d2f0   2ee0

swapbytes(A)
ans(:,:,1) =
   b004   9600   8403
   c201   ee02   1a04
   5802   4605   2c01
ans(:,:,2) =
   80bb   7017   a08c
   5046   3075   10a4
   c05d   f0d2   e02e
```

See Also

`typecast`

Purpose

Switch among several cases, based on expression

Syntax

```
switch switch_expr
 case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

Discussion

The `switch` statement syntax is a means of conditionally executing code. In particular, `switch` executes one set of statements selected from an arbitrary number of alternatives. Each alternative is called a *`case`*, and consists of

- The `case` statement
- One or more case expressions
- One or more statements

In its basic syntax, `switch` executes the statements associated with the first case where *`switch_expr`* == *`case_expr`*. When the case expression is a cell array (as in the second case above), the *`case_expr`* matches if any of the elements of the cell array matches the switch expression. If no case expression matches the switch expression, then control passes to the `otherwise` case (if it exists). After the case is executed, program execution resumes with the statement after the `end`.

The *`switch_expr`* can be a scalar or a string. A scalar *`switch_expr`* matches a *`case_expr`* if *`switch_expr`*==*`case_expr`*. A string *`switch_expr`* matches a *`case_expr`* if `strcmp(`*`switch_expr`*`,`*`case_expr`*`)` returns logical `1` (`true`).

Note for C Programmers Unlike the C language `switch` construct, the MATLAB `switch` does not "fall through." That is, `switch` executes only the first matching case; subsequent matching `case`s do not execute. Therefore, `break` statements are not used.

Examples

To execute a certain block of code based on what the string, `method`, is set to,

```
method = 'Bilinear';

switch lower(method)
   case {'linear','bilinear'}
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
   case 'nearest'
      disp('Method is nearest')
   otherwise
      disp('Unknown method.')
end

Method is linear
```

See Also

`case`, `otherwise`, `end`, `if`, `else`, `elseif`, `while`

symamd

Purpose

Symmetric approximate minimum degree permutation

Syntax

```
p = symamd(S)
p = symamd(S,knobs)
[p,stats] = symamd(...)
```

Description

`p = symamd(S)` for a symmetric positive definite matrix `S`, returns the permutation vector `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. To find the ordering for `S`, `symamd` constructs a matrix `M` such that `spones(M'*M) = spones (S)`, and then computes `p = colamd(M)`. The `symamd` function may also work well for symmetric indefinite matrices.

`S` must be square; only the strictly lower triangular part is referenced.

`p = symamd(S,knobs)` where `knobs` is a scalar. If `S` is `n`-by-`n`, rows and columns with more than `knobs*n` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs = spparms('wh_frac')`.

`[p,stats] = symamd(...)` produces the optional vector `stats` that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `symamd`
`stats(2)`	Number of dense or empty columns ignored by `symamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `symamd` (roughly of size `8.4*nnz(tril(S,-1)) + 9n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid
`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists
`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `symamd`. For this reason, `symamd` verifies that `S` is valid:

- If a row index appears two or more times in the same column, `symamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `symamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `symamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`).

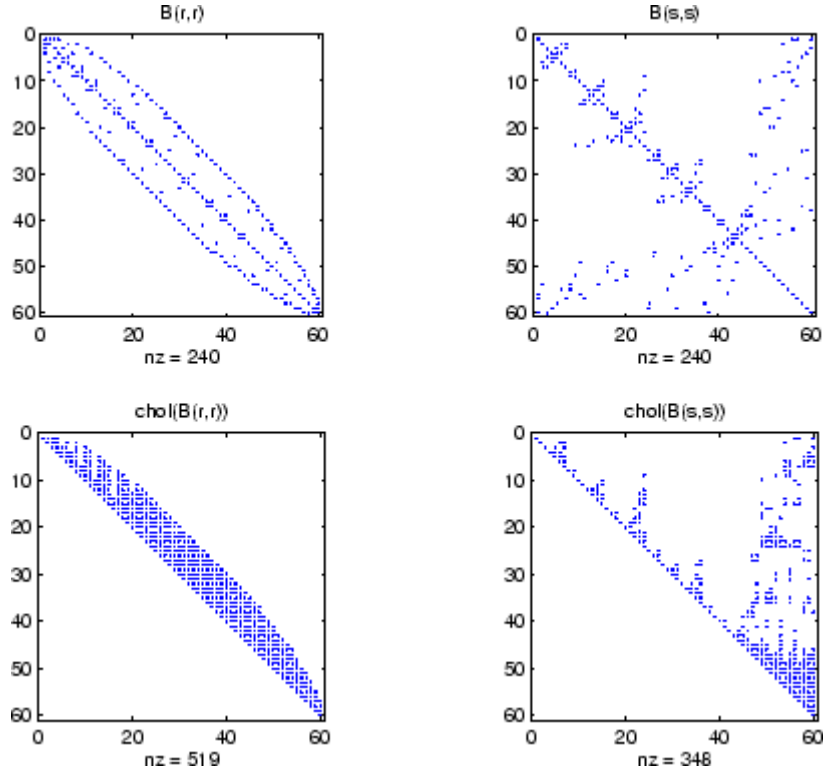
The ordering is followed by a symmetric elimination tree post-ordering.

Note `symamd` tends to be faster than `symmmd` and tends to return a better ordering.

Examples

Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the `symrcm` reference page.

```
B = bucky+4*speye(60);
r = symrcm(B);
p = symamd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R,4), title('B(r,r)')
subplot(2,2,2), spy(S,4), title('B(s,s)')
subplot(2,2,3), spy(chol(R),4), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S),4), title('chol(B(s,s))')
```



Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

See Also

`colamd`, `colperm`, `spparms`, `symrcm`

References

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (`davis@cise.ufl.edu`), University of Florida. The algorithm was developed in collaboration with John Gilbert,

Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory.
Sparse Matrix Algorithms Research at the University of Florida:
http://www.cise.ufl.edu/research/sparse/

Purpose

Symbolic factorization analysis

Syntax

```
count = symbfact(A)
count = symbfact(A,'sym')
count = symbfact(A,'col')
count = symbfact(A,'row')
count = symbfact(A,'lo')
[count,h,parent,post,R] = symbfact(...)
[count,h,parent,post,L] = symbfact(A,type,'lower')
```

Description

`count = symbfact(A)` returns the vector of row counts of `R=chol(A'*A)`. `symbfact` should be much faster than `chol(A)`.

`count = symbfact(A,'sym')` is the same as `count = symbfact(A)`.

`count = symbfact(A,'col')` returns row counts of `R=chol(A'*A)` (without forming it explicitly).

`count = symbfact(A,'row')` returns row counts of `R=chol(A*A')`.

`count = symbfact(A,'lo')` is the same as `count = symbfact(A)` and uses `tril(A)`.

`[count,h,parent,post,R] = symbfact(...)` has several optional return values.

The flop count for a subsequent Cholesky factorization is `sum(count.^2)`

Return Value	Description
`h`	Height of the elimination tree
`parent`	The elimination tree itself
`post`	Postordering of the elimination tree
`R`	0-1 matrix having the structure of `chol(A)` for the symmetric case, `chol(A'*A)` for the `'col'` case, or `chol(A*A')` for the `'row'` case.

`symbfact(A)` and `symbfact(A,'sym')` use the upper triangular part of `A (triu(A))` and assume the lower triangular part is the transpose of the upper triangular part. `symbfact(A,'lo')` uses `tril(A)` instead.

`[count,h,parent,post,L] = symbfact(A,type,'lower')` where `type` is one of `'sym'`,`'col'`, `'row'`, or`'lo'` returns a lower triangular symbolic factor `L=R'`. This form is quicker and requires less memory.

See Also

`chol`, `etree`, `treelayout`

symmlq

Purpose

Symmetric LQ method

Syntax

```
x = symmlq(A,b)
symmlq(A,b,tol)
symmlq(A,b,tol,maxit)
symmlq(A,b,tol,maxit,M)
symmlq(A,b,tol,maxit,M1,M2)
symmlq(A,b,tol,maxit,M1,M2,x0)
[x,flag] = symmlq(A,b,...)
[x,flag,relres] = symmlq(A,b,...)
[x,flag,relres,iter] = symmlq(A,b,...)
[x,flag,relres,iter,resvec] = symmlq(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)
```

Description

`x = symmlq(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should also be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle `afun` such that `afun(x)` returns `A*x`. See "Function Handles" in the MATLAB Programming documentation for more information.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `symmlq` converges, a message to that effect is displayed. If `symmlq` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`symmlq(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `symmlq` uses the default, `1e-6`.

`symmlq(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `symmlq` uses the default, `min(n,20)`.

`symmlq(A,b,tol,maxit,M)` and `symmlq(A,b,tol,maxit,M1,M2)` use the symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = in(sqrt(M))*y`. If `M` is `[]` then `symmlq` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`symmlq(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `symmlq` uses the default, an all-zero vector.

`[x,flag] = symmlq(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`symmlq` converged to the desired tolerance `tol` within `maxit` iterations.
1	`symmlq` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`symmlq` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `symmlq` became too small or too large to continue computing.
5	Preconditioner `M` was not symmetric positive definite.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = symmlq(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = symmlq(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = symmlq(A,b,...)` also returns a vector of estimates of the `symmlq` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)` also returns a vector of estimates of the conjugate gradients residual norms at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50; M1 = spdiags(4*on,0,n,n);

x = symmlq(A,b,tol,maxit,M1);
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in an M-file `run_symmlq` that

- Calls `symmlq` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_symmlq` are available to `afun`.

The following shows the code for `run_symmlq`:

```
function x1 = run_symmlq
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
```

```
x1 = symmlq(@afun,b,tol,maxit,M1);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
       end
end
```

When you enter

```
x1=run_symmlq;
```

MATLAB displays the message

```
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 3

Use a symmetric indefinite matrix that fails with pcg.

```
A = diag([20:-1:1,-1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, symmlq can handle the indefinite matrix A.

```
x = symmlq(A,b,1e-6,40);
symmlq converged at iteration 39 to a solution with relative
residual 1.3e-007
```

See Also

bicg, bicgstab, cgs, lsqr, gmres, minres, pcg, qmr

function_handle (@), mldivide (\)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

Purpose

Sparse symmetric minimum degree ordering

Syntax

```
p = symmmd(S)
```

Note `symmmd` is obsolete and will be removed from a future version of MATLAB. Use `symamd` instead.

Description

`p = symmmd(S)` returns a symmetric minimum degree ordering of `S`. For a symmetric positive definite matrix `S`, this is a permutation `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. Sometimes `symmmd` works well for symmetric indefinite matrices too.

Algorithm

The symmetric minimum degree algorithm is based on the column minimum degree algorithm. In fact, `symmmd(A)` just creates a nonzero structure `K` such that `K'*K` has the same nonzero structure as `A` and then calls the column minimum degree code for `K`.

See Also

`colamd`, `colmmd`, `colperm`, `symamd`, `symrcm`

References

[1] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications 13*, 1992, pp. 333-356.

Purpose

Sparse reverse Cuthill-McKee ordering

Syntax

```
r = symrcm(S)
```

Description

`r = symrcm(S)` returns the symmetric reverse Cuthill-McKee ordering of `S`. This is a permutation `r` such that `S(r,r)` tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric `S`.

For a real, symmetric sparse matrix, `S`, the eigenvalues of `S(r,r)` are the same as those of `S`, but `eig(S(r,r))` probably takes less time to compute than `eig(S)`.

Algorithm

The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

Examples

The statement

```
B = bucky;
```

uses an M-file in the `demos` toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name `bucky`), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together. With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows

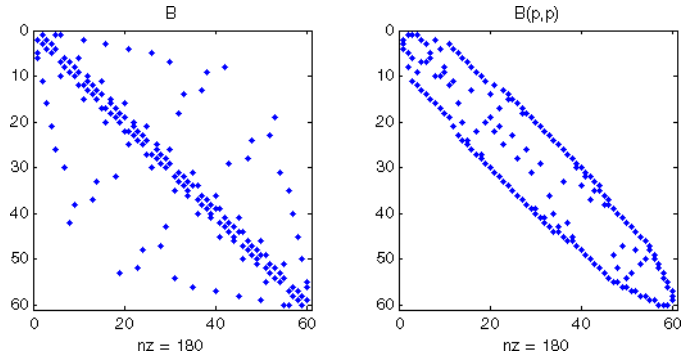
```
subplot(1,2,1), spy(B), title('B')
```

The reverse Cuthill-McKee ordering is obtained with

```
p = symrcm(B);
R = B(p,p);
```

The spy plot shows a much narrower bandwidth.

```
subplot(1,2,2), spy(R), title('B(p,p)')
```



This example is continued in the reference pages for symamd.

The bandwidth can also be computed with

```
[i,j] = find(B);
bw = max(i-j) + 1;
```

The bandwidths of B and R are 35 and 12, respectively.

See Also

colamd, colperm, symamd

References

[1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

Purpose

Determine symbolic variables in expression

Syntax

```
symvar 'expr'
s = symvar('expr')
```

Description

`symvar 'expr'` searches the expression, `expr`, for identifiers other than `i`, `j`, `pi`, `inf`, `nan`, `eps`, and common functions. `symvar` displays those variables that it finds or, if no such variable exists, displays an empty cell array, `{}`.

`s = symvar('expr')` returns the variables in a cell array of strings, `s`. If no such variable exists, `s` is an empty cell array.

Examples

`symvar` finds variables `beta1` and `x`, but skips `pi` and the `cos` function.

```
symvar 'cos(pi*x - beta1)'

ans =

    'beta1'
    'x'
```

See Also

`findstr`

Purpose

Synchronize and resample two `timeseries` objects using common time vector

Syntax

```
[ts1 ts2] = synchronize(ts1,ts2,'SynchronizeMethod')
```

Description

`[ts1 ts2] = synchronize(ts1,ts2,'SynchronizeMethod')` creates two new `timeseries` objects by synchronizing ts1 and ts2 using a common time vector. The string `'SynchronizeMethod'` defines the method for synchronizing the `timeseries` and can be one of the following:

- `'Union'` — Resample `timeseries` objects using a time vector that is a union of the time vectors of ts1 and `ts2` on the time range where the two time vectors overlap.
- `'Intersection'` — Resample `timeseries` objects on a time vector that is the intersection of the time vectors of ts1 and ts2.
- `'Uniform'` — Requires an additional argument as follows:

  ```
  [ts1 ts2] = synchronize(ts1,ts2,'Uniform','Interval',value)
  ```

  This method resamples time series on a uniform time vector, where value specifies the time interval between the two samples. The uniform time vector is the overlap of the time vectors of ts1 and ts2. The interval units are assumed to be the smaller units of ts1 and ts2.

You can specify additional arguments by using property-value pairs:

- `'InterpMethod'`: Forces the specified interpolation method (over the default method) for this synchronize operation. Can be either a string, `'linear'` or `'zoh'`, or a `tsdata.interpolation` object that contains a user-defined interpolation method.
- `'QualityCode'`: Integer (between -128 and 127) used as the quality code for both time series after the synchronization.

- `'KeepOriginalTimes'`: Logical value (`true` or `false`) indicating whether the new time series should keep the original time values. For example,

  ```
  ts1 = timeseries([1 2],[datestr(now); datestr(now+1)]);
  ts2 = timeseries([1 2],[datestr(now-1); datestr(now)]);
  ```

  Note that `ts1.timeinfo.StartDate` is one day after `ts2.timeinfo.StartDate`. If you use

  ```
  [ts1 ts2] = synchronize(ts1,ts2,'union');
  ```

  the `ts1.timeinfo.StartDate` is changed to match `ts2.TimeInfo.StartDate` and `ts1.Time` changes to 1.

  But if you use

  ```
  [ts1 ts2] =
  synchronize(ts1,ts2,'union','KeepOriginalTimes',true);
  ```

  `ts1.timeinfo.StartDate` is unchanged and `ts1.Time` is still `0`.

- `'tolerance'`: Real number used as the tolerance for differentiating two time values when comparing the `ts1` and `ts2` time vectors. The default tolerance is `1e-10`. For example, when the sixth time value in `ts1` is `5+(1e-12)` and the sixth time value in `ts2` is `5-(1e-13)`, both values are treated as 5 by default. To differentiate those two times, you can set `'tolerance'` to a smaller value such as `1e-15`, for example.

See Also

`timeseries`

Purpose

Two ways to call MATLAB functions

Description

You can call MATLAB functions using either *command syntax* or *function syntax*, as described below.

Command Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname arg1 arg2 ... argn
```

Command syntax does not allow you to obtain any values that might be returned by the function. Attempting to assign output from the function to a variable using command syntax generates an error. Use function syntax instead.

Examples of command syntax:

```
save mydata.mat x y z
import java.awt.Button java.lang.String
```

Arguments are treated as string literals. See the examples below, under “Argument Passing” on page 2-3286.

Function Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by commas and enclosed in parentheses:

```
functionname(arg1, arg2, ..., argn)
```

You can assign the output of the function to one or more output values. When assigning to more than one output variable, separate the variables by commas or spaces and enclose them in square brackets (`[]`):

```
[out1,out2,...,outn] = functionname(arg1, arg2, ..., argn)
```

Examples of function syntax:

```
copyfile('srcfile', '..\mytests', 'writable')
[x1,x2,x3,x4] = deal(A{:})
```

Arguments are passed to the function by value. See the examples below, under “Argument Passing” on page 2-3286.

Argument Passing

When calling a function using command syntax, MATLAB passes the arguments as string literals. When using function syntax, arguments are passed by value.

In the following example, assign a value to `A` and then call `disp` on the variable to display the value passed. Calling `disp` with command syntax passes the variable name, `'A'`:

```
A = pi;
disp A
   A
```

while function syntax passes the value assigned to `A`:

```
A = pi;
disp(A)
   3.1416
```

The next example passes two strings to `strcmp` for comparison. Calling the function with command syntax compares the variable names, `'str1'` and `'str2'`:

```
str1 = 'one';    str2 = 'one';
strcmp str1 str2
ans =
     0        (unequal)
```

while function syntax compares the values assigned to the variables, `'one'` and `'one'`:

```
str1 = 'one';    str2 = 'one';
strcmp(str1, str2)
```

```
ans =
     1        (equal)
```

Passing Strings

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotes, (`'string'`). For example, to create a new directory called `myapptests`, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```
dirname = 'myapptests';
mkdir(dirname)
```

See Also

`mlint`

Purpose

Execute operating system command and return result

Syntax

```
system('command')
[status, result] = system('command')
```

Description

`system('command')` calls upon the operating system to run `command`, for example `dir` or `ls` or a UNIX shell script, and directs the output to MATLAB. If `command` runs successfully, `ans` is `0`. If `command` fails or does not exist on your operating system, `ans` is a nonzero value and an explanatory message appears.

`[status, result] = system('command')` calls upon the operating system to run `command`, and directs the output to MATLAB. If `command` runs successfully, `status` is `0` and `result` contains the output from `command`. If `command` fails or does not exist on your operating system, `status` is a nonzero value and `result` contains an explanatory message.

Note Running `system` on Windows with a `command` that relies on the current directory fails when the current directory is specified using a UNC pathname because DOS does not support UNC pathnames. When this happens, MATLAB returns the error:

```
???  Error using ==> system DOS commands may not be
executed when the current directory is a UNC pathname.
```

To work around this limitation, change the directory to a mapped drive prior to running `system` or a function that calls `system`.

Examples

On a Windows system, display the current directory by accessing the operating system.

```
[status currdir] = system('cd')
status =
    0
currdir =
```

```
D:\work\matlab\test
```

See Also

`!` (bang), `computer`, `dos`, `perl`, `unix`, `winopen`

"Running External Programs" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Tangent of argument in radians

Syntax

```
Y = tan(X)
```

Description

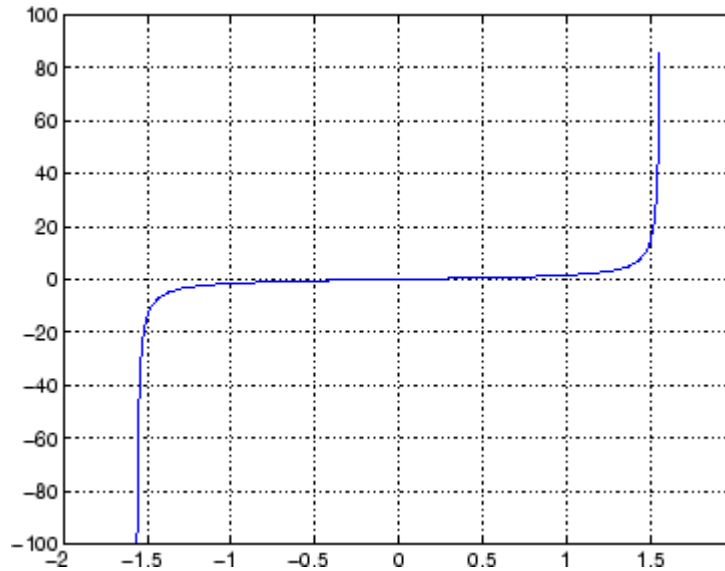
The `tan` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = tan(X)` returns the circular tangent of each element of `X`.

Examples

Graph the tangent function over the domain $-\pi/2 < x < \pi/2$.

```
x = (-pi/2)+0.01:0.01:(pi/2)-0.01;
plot(x,tan(x)), grid on
```



The expression `tan(pi/2)` does not evaluate as infinite but as the reciprocal of the floating point accuracy `eps` since `pi` is only a floating-point approximation to the exact value of π.

Definition

The tangent can be defined as

$$\tan(z) = \frac{\sin(z)}{\cos(z)}$$

Algorithm

`tan` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`tand`, `tanh`, `atan`, `atan2`, `atand`, `atanh`

tand

Purpose

Tangent of argument in degrees

Syntax

```
Y = tand(X)
```

Description

`Y = tand(X)` is the tangent of the elements of `X`, expressed in degrees. For odd integers `n`, `tand(n*90)` is infinite, whereas `tan(n*pi/2)` is large but finite, reflecting the accuracy of the floating point value of `pi`.

See Also

`tan`, `tanh`, `atan`, `atan2`, `atand`, `atanh`

Purpose

Hyperbolic tangent

Syntax

```
Y = tanh(X)
```

Description

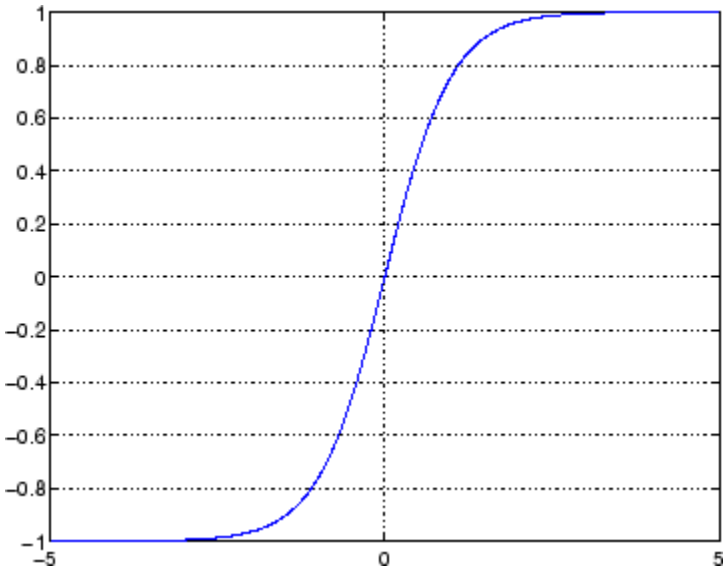
The `tanh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = tanh(X)` returns the hyperbolic tangent of each element of `X`.

Examples

Graph the hyperbolic tangent function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,tanh(x)), grid on
```



Definition

The hyperbolic tangent can be defined as

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$$

Algorithm

`tanh` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see `http://www.netlib.org`.

See Also

`atan`, `atan2`, `tan`

Purpose

Compress files into tar file

Syntax

```
tar(tarfilename,files)
tar(tarfilename,files,rootdir)
entrynames = tar(...)
```

Description

`tar(tarfilename,files)` creates a tar file with the name `tarfilename` from the list of files and directories specified in `files`. Relative paths are stored in the tar file, but absolute paths are not. Directories recursively include all of their content.

`tarfilename` is a string specifying the name of the tar file. The `.tar` extension is appended to `tarfilename` if omitted. The `tarfilename` extension can end in `.tgz` or `.gz`. In this case, `tarfilename` is gzipped.

`files` is a string or cell array of strings containing the list of files or directories included in `tarfilename`. Individual files that are on the MATLAB path can be specified as partial pathnames. Otherwise an individual file can be specified relative to the current directory or with an absolute path. Directories must be specified relative to the current directory or with absolute paths. On UNIX systems, directories can also start with `~/` or *`~username/`*, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a filename or partial pathname.

`tar(tarfilename,files,rootdir)` allows the path for `files` to be specified relative to `rootdir` rather than the current directory.

`entrynames = tar(...)` returns a string cell array of the relative path entry names contained in `tarfilename`.

Example

Tar all files in the current directory to the file `backup.tgz`:

```
tar('backup.tgz','.');
```

See Also

`gzip`, `gunzip`, `untar`, `unzip`, `zip`

tempdir

Purpose

Name of system's temporary directory

Syntax

```
tmp_dir = tempdir
```

Description

`tmp_dir = tempdir` returns the name of the system's temporary directory, if one exists. This function does not create a new directory.

See "Opening Temporary Files and Directories" for more information.

See Also

`tempname`

Purpose

Unique name for temporary file

Syntax

```
tmp_nam = tempname
```

Description

`tmp_nam = tempname` returns a unique string, `tmp_nam`, suitable for use as a temporary filename.

Note The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

See “Opening Temporary Files and Directories” for more information.

See Also

`tempdir`

tetramesh

Purpose

Tetrahedron mesh plot

Syntax

```
tetramesh(T,X,c)
tetramesh(T,X)
h = tetramesh(...)
tetramesh(...,'param','value','param','value'...)
```

Description

`tetramesh(T,X,c)` displays the tetrahedrons defined in the `m`-by-4 matrix `T` as mesh. `T` is usually the output of `delaunayn`. A row of `T` contains indices into `X` of the vertices of a tetrahedron. `X` is an `n`-by-3 matrix, representing `n` points in 3 dimension. The tetrahedron colors are defined by the vector `C`, which is used as indices into the current colormap.

Note If `T` is the output of `delaunay3`, then `X` is the concatenation of the `delaunay3` input arguments `x`, `y`, `z` interpreted as column vectors, i.e., `X = [x(:)  y(:)  z(:)]`.

`tetramesh(T,X)` uses `C = 1:m` as the color for the `m` tetrahedrons. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

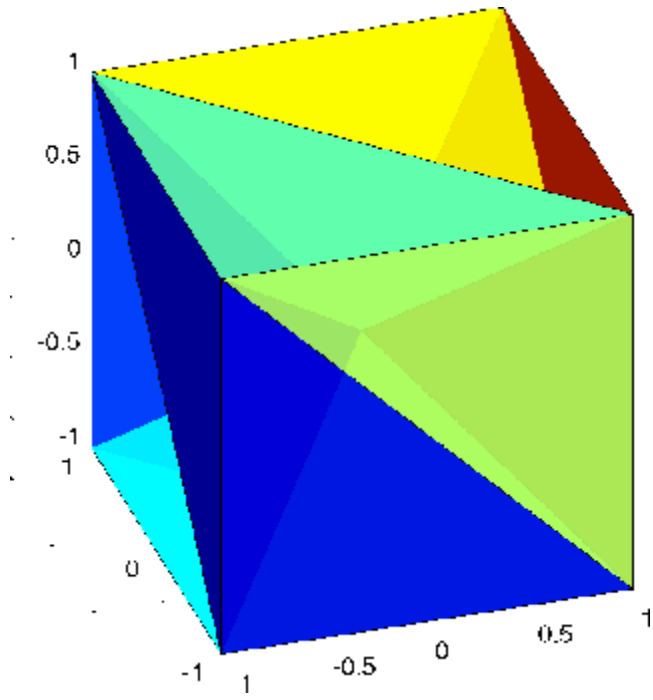
`h = tetramesh(...)` returns a vector of tetrahedron handles. Each element of `h` is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch `'Visible'` property `'on'` or `'off'`.

`tetramesh(...,'param','value','param','value'...)` allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to `0.9`. You can overwrite this value by using the property name/property value pair `('FaceAlpha',value)` where `value` is a number between `0` and `1`. See `Patch Properties` for information about the available properties.

Examples

Generate a 3-dimensional Delaunay tessellation, then use `tetramesh` to visualize the tetrahedrons that form the corresponding simplex.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);  % A cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
X = [x(:) y(:) z(:)];
Tes = delaunayn(X)

Tes =
   9  1  5  6
   3  9  1  5
   2  9  1  6
   2  3  9  4
   2  3  9  1
   7  9  5  6
   7  3  9  5
   8  7  9  6
   8  2  9  6
   8  2  9  4
   8  3  9  4
   8  7  3  9
tetramesh(Tes,X);camorbit(20,0)
```



See Also delaunayn, patch, Patch Properties, trimesh, trisurf

Purpose

Produce TeX format from character string

Syntax

```
texlabel(f)
texlabel(f,'literal')
```

Description

`texlabel(f)` converts the MATLAB expression `f` into the TeX equivalent for use in text strings. It processes Greek variable names (e.g., lambda, delta, etc.) into a string that is displayed as actual Greek letters.

`texlabel(f,'literal')` prints Greek variable names as literals.

If the string is too long to fit into a figure window, then the center of the expression is replaced with a tilde ellipsis (~~~).

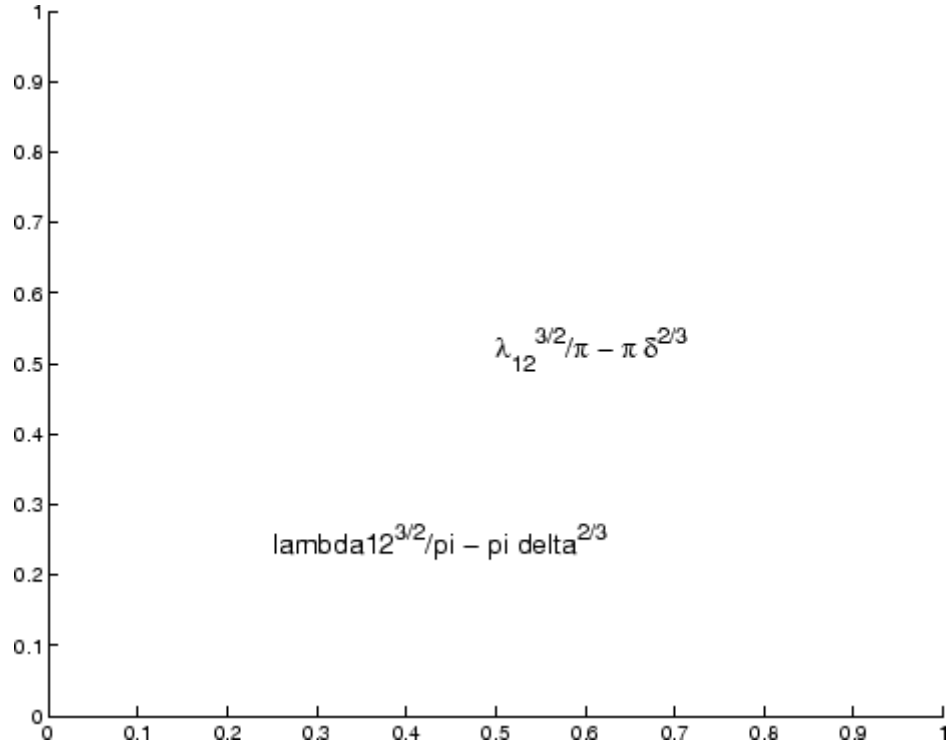
Examples

You can use `texlabel` as an argument to the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` commands. For example,

```
title(texlabel('sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2)'))
```

By default, `texlabel` translates Greek variable names to the equivalent Greek letter. You can select literal interpretation by including the `literal` argument. For example, compare these two commands.

```
text(.5,.5,...
     texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)'))
text(.25,.25,...
     texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)','literal'))
```



See Also

`text`, `title`, `xlabel`, `ylabel`, `zlabel`, the text `String` property

"Annotating Plots" on page 1-87 for related functions

Purpose

Create text object in current axes

Syntax

```
text(x,y,'string')
text(x,y,z,'string')
text(x,y,z,'string','PropertyName',PropertyValue....)
text('PropertyName',PropertyValue....)
h = text(...)
```

Description

`text` is the low-level function for creating text graphics objects. Use `text` to place character strings at specified locations.

`text(x,y,'string')` adds the string in quotes to the location specified by the point `(x,y)`.

`text(x,y,z,'string')` adds the string in 3-D coordinates.

`text(x,y,z,'string','PropertyName',PropertyValue....)` adds the string in quotes to the location defined by the coordinates and uses the values for the specified text properties. See the text property list section at the end of this page for a list of text properties.

`text('PropertyName',PropertyValue....)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

`h = text(...)` returns a column vector of handles to text objects, one handle per object. All forms of the `text` function optionally return this output argument.

See the `String` property for a list of symbols, including Greek letters.

Remarks

Position Text Within the Axes

The default text units are the units used to plot data in the graph. Specify the text location coordinates (the `x`, `y`, and `z` arguments) in the data units of the current graph (see "Example"). You can use other units to position the text by set the text `Units` property to `normalized` or one of the nonrelative units (`pixels`, `inches`, `centimeters`, `points`).

Note that the Axes Units property controls the positioning of the Axes within the figure and is not related to the axes data units used for graphing.

The `Extent`, `VerticalAlignment`, and `HorizontalAlignment` properties control the positioning of the character string with regard to the text location point.

If the coordinates are vectors, `text` writes the string at all locations defined by the list of points. If the character string is an array the same length as `x`, `y`, and `z`, `text` writes the corresponding row of the string array at each point specified.

Multiline Text

When specifying strings for multiple text objects, the string can be

- A cell array of strings
- A padded string matrix

Each element of the specified string array creates a different text object.

When specifying the string for a single text object, cell arrays of strings and padded string matrices result in a text object with a multiline string, while vertical slash characters are not interpreted as separators and result in a single line string containing vertical slashes.

Behavior of the Text Function

`text` is a low-level function that accepts property name/property value pairs as input arguments. However, the convenience form,

```
text(x,y,z,'string')
```

is equivalent to

```
text('Position',[x,y,z],'String','string')
```

You can specify other properties only as property name/property value pairs. See the text property list at the end of this page for a description

of each property. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

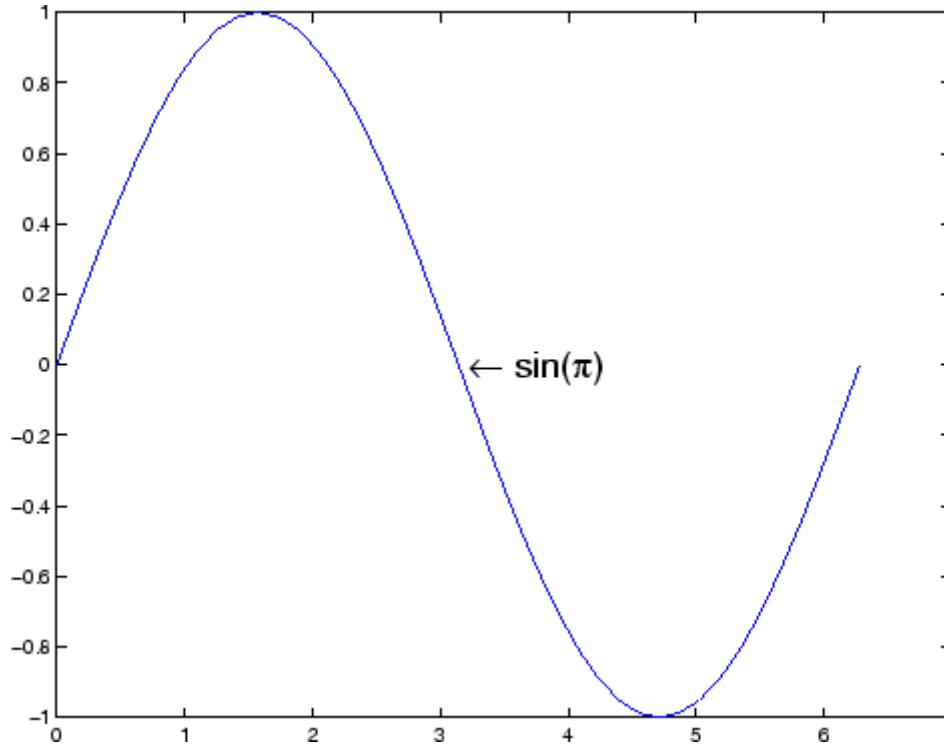
`text` does not respect the setting of the figure or axes `NextPlot` property. This allows you to add text objects to an existing axes without setting `hold` to `on`.

Examples

The statements

```
plot(0:pi/20:2*pi,sin(0:pi/20:2*pi))
text(pi,0,' \leftarrow sin(\pi)','FontSize',18)
```

annotate the point at `(pi,0)` with the string `sin(π)`



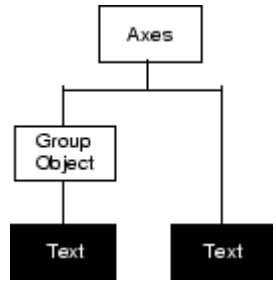
The statement

```
text(x,y,'\ite^{i\omega\tau} = cos(\omega\tau) + i sin(\omega\tau)')
```

uses embedded TeX sequences to produce

$$e^{i\omega\tau} = cos(\omega\tau) + i\ sin(\omega\tau)$$

Object Hierarchy



Setting Default Properties

You can set default text properties on the axes, figure, and root levels:

```
set(0,'DefaulttextProperty',PropertyValue...)
set(gcf,'DefaulttextProperty',PropertyValue...)
set(gca,'DefaulttextProperty',PropertyValue...)
```

Where *Property* is the name of the text property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access text properties.

See Also

`annotation`, `gtext`, `int2str`, `num2str`, `title`, `xlabel`, `ylabel`, `zlabel`, `strings`

"Object Creation Functions" on page 1-94 for related functions

`Text Properties` for property descriptions

Text Properties

Purpose

Text properties

Modifying Properties

You can set and query graphics object properties using the property editor or the `set` and `get` commands.

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see Setting Default Property Values.

See Core Objects for general information about this type of object.

Text Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object Read Only

Control the display of text objects in legends. The `Annotation` property enables you to specify whether this text object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the text object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this text object in a legend (default)
`off`	Do not include this text object in a legend
`children`	Same as `on` because text objects do not have children

Setting the `IconDisplayStyle` property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the `IconDisplayStyle` property

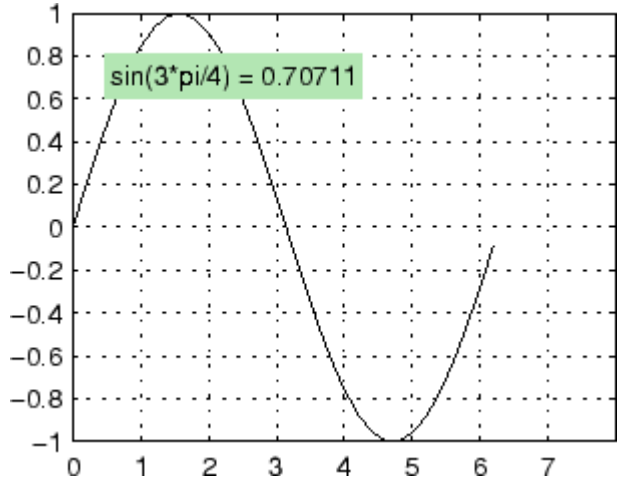
See “Controlling Legends” for more information and examples.

`BackgroundColor`

`ColorSpec | {none}`

Color of text extent rectangle. This property enables you to define a color for the rectangle that encloses the text `Extent` plus the text `Margin`. For example, the following code creates a text object that labels a plot and sets the background color to light green.

```
text(3*pi/4,sin(3*pi/4),...
 ['sin(3*pi/4) = ',num2str(sin(3*pi/4))],...
 'HorizontalAlignment','center',...
 'BackgroundColor',[.7 .9 .7]);
```



For additional features, see the following properties:

- `EdgeColor` — Color of the rectangle's edge (`none` by default).
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increase the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

See also Drawing Text in a Box in the MATLAB Graphics documentation for an example using background color with contour labels.

`BeingDeleted`
`on | {off}` read only

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted, and therefore can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is set to `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback function. A callback function that executes whenever you press a mouse button while the pointer is over the text object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property). For example, the following function takes different action depending on what type of selection was made:

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a text object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```
set(h,'ButtonDownFcn',@button_down)
```

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`Children`
matrix (read only)

The empty matrix; text objects have no children.

`Clipping`
`on | {off}`

Clipping mode. When `Clipping` is `on`, MATLAB does not display any portion of the text that is outside the axes.

`Color`
`ColorSpec`

Text color. A three-element RGB vector or one of the predefined names, specifying the text color. The default value for `Color` is white. See `ColorSpec` for more information on specifying color.

`CreateFcn`
functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during object creation. A callback function that executes when MATLAB creates a text object. You must define this property as a default value for text or in a call to the `text` function that creates a new text object. For example, the statement

```
set(0,'DefaultTextCreateFcn',@text_create)
```

defines a default value on the root level that sets the figure `Pointer` property to crosshairs whenever you create a text object. The callback function must be on your MATLAB path when you execute the above statement.

```
function text_create(src,evnt)
```

```
% src - the object that is the source of the event
% evnt - empty for this property
 set(gcbf,'Pointer','crosshair')
end
```

MATLAB executes this function after setting all text properties. Setting this property on an existing text object has no effect. The function must define at least two input arguments (handle of object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DeleteFcn`

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Delete text callback function. A callback function that executes when you delete the text object (e.g., when you issue a `delete` command or clear the axes `cla` or figure `clf`). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DisplayName`

string (default is empty string)

String used by legend for this text object. The `legend` function uses the string defined by the `DisplayName` property to label this text object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this text object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, legend creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

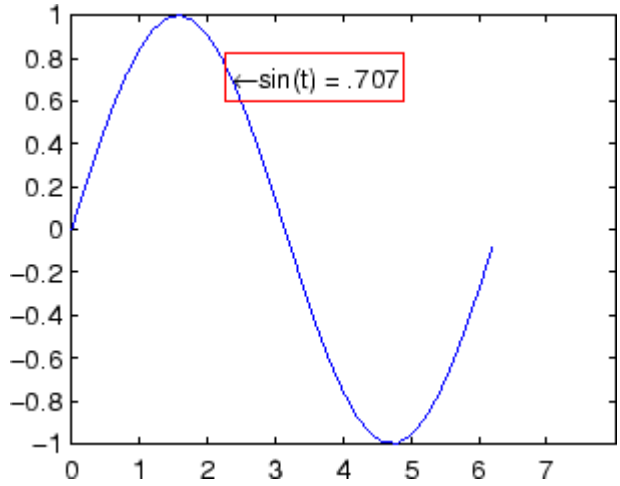
See “Controlling Legends” for more examples.

`EdgeColor`

`ColorSpec | {none}`

Color of edge drawn around text extent rectangle plus margin. This property enables you to specify the color of a box drawn around the text `Extent` plus the text `Margin`. For example, the following code draws a red rectangle around text that labels a plot.

```
text(3*pi/4,sin(3*pi/4),...
'\leftarrowsin(t) = .707',...
'EdgeColor','red');
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle’s interior (`none` by default)
- `LineStyle` — Style of the rectangle’s edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle’s edge line (first set `EdgeColor`)

- `Margin` — Increases the size of the rectangle by adding a margin to the area defined by the text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

`Editing`

```
on | {off}
```

Enable or disable editing mode. When this property is set to the default `off`, you cannot edit the text string interactively (i.e., you must change the `String` property to change the text). When this property is set to `on`, MATLAB places an insert cursor at the end of the text string and enables editing. To apply the new text string,

1. Press the **Esc** key.
2. Click in any figure window (including the current figure).
3. Reset the `Editing` property to `off`.

MATLAB then updates the `String` property to contain the new text and resets the `Editing` property to `off`. You must reset the `Editing` property to `on` to resume editing.

`EraseMode`

```
{normal} | none | xor | background
```

Erase mode. This property controls the technique MATLAB uses to draw and erase text objects. Alternative erase modes are useful for creating animated sequences where controlling the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the text when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode` `none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the text by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. When the text is erased, it does not damage the objects beneath it. However, when text is drawn in `xor` mode, its color depends on the color of the screen beneath it. It is correctly colored only when it is over axes background Color, or the figure background Color if the axes `Color` is set to `none`.
- `background` — Erase the text by drawing it in the axes background Color, or the figure background Color if the axes `Color` is set to `none`. This damages objects that are behind the erased text, but text is always properly colored.

  Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is set to `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look differently on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing nonnormal mode objects.

`Extent`
: position rectangle (read only)

Position and size of text. A four-element read-only vector that defines the size and position of the text string

```
[left,bottom,width,height]
```

If the `Units` property is set to `data` (the default), `left` and `bottom` are the *x*- and *y*-coordinates of the lower left corner of the text `Extent`.

For all other values of `Units`, `left` and `bottom` are the distance from the lower left corner of the axes `position` rectangle to the lower left corner of the text `Extent`. `width` and `height` are the dimensions of the `Extent` rectangle. All measurements are in units specified by the `Units` property.

`FontAngle`

`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

`FontName`

A name, such as `Courier`, or the string `FixedWidth`

Font family. A string specifying the name of the font to use for the text object. To display and print properly, this must be a font that your system supports. The default font is Helvetica.

Specifying a Fixed-Width Font

If you want text to use a fixed-width font that looks good in any locale, you should set `FontName` to the string `FixedWidth`:

```
set(text_handle,'FontName','FixedWidth')
```

This eliminates the need to hard-code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan where multibyte character sets

are used). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` (note that this string is case sensitive) and rely on `FixedWidthFontName` to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`.

Note that setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`
size in `FontUnits`

Font size. A value specifying the font size to use for text in units determined by the `FontUnits` property. The default point size is 10 (1 point = 1/72 inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `bold` or `demi` causes MATLAB to use a bold font.

`FontUnits`
`{points} | normalized | inches | centimeters | pixels`

Font size units. MATLAB uses this property to determine the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

Note that if you are setting both the `FontSize` and the `FontUnits` in one function call, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is set to `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`,

- The object's handle does not appear in its parent's `Children` property.

- Figures do not appear in the root's `CurrentFigure` property.
- Objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property.
- Axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to on to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines if the text can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the text. If `HitTest` is set to `off`, clicking the text selects the object below it (which is usually the axes containing it).

For example, suppose you define the button down function of an image (see the `ButtonDownFcn` property) to display text at the location you click with the mouse.

First define the callback routine.

```
function bd_function
pt = get(gca,'CurrentPoint');
text(pt(1,1),pt(1,2),pt(1,3),...
 '{\fontsize{20}\oplus} The spot to label',...
 'HitTest','off')
```

Now display an image, setting its `ButtonDownFcn` property to the callback routine.

```
load earth
image(X,'ButtonDownFcn','bd_function'); colormap(map)
```

When you click the image, MATLAB displays the text string at that location. With `HitTest` set to `off`, existing text cannot intercept any subsequent button down events that occur over the text. This enables the image's button down function to execute.

`HorizontalAlignment`
`{left} | center | right`

Horizontal alignment of text. This property specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the `Position` property. The following picture illustrates the alignment options.

`HorizontalAlignment` viewed with the `VerticalAlignment` set to middle (the default).

See the `Extent` property for related information.

`Interpreter`
latex | `{tex}` | `none`

Interpret T_EX instructions. This property controls whether MATLAB interprets certain characters in the `String` property as T_EX instructions (default) or displays all characters literally. The options are:

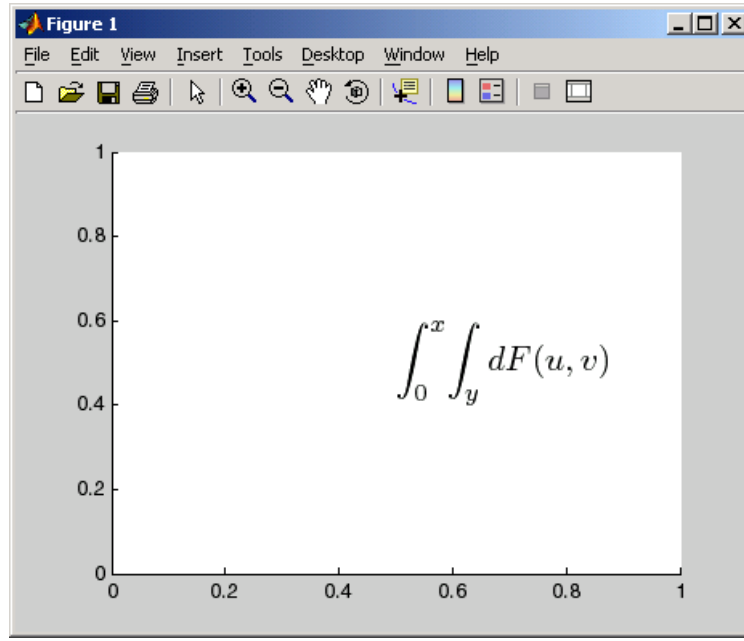
- `latex` — Supports the full L_AT_EX markup language.
- `tex` — Supports a subset of plain T_EX markup language. See the `String` property for a list of supported T_EX instructions.

- `none` — Displays literal characters.

Latex Interpreter

To enable the LaT$_E$X interpreter for text objects, set the `Interpreter` property to `latex`. For example, the following statement displays an equation in a figure at the point [.5 .5], and enlarges the font to 16 points.

```
text('Interpreter','latex',...
 'String','$$\int_0^x\!\int_y dF(u,v)$$',...
 'Position',[.5 .5],...
 'FontSize',16)
```



Information About Using TEX

The following references may be useful to people who are not familiar with T$_E$X.

- Donald E. Knuth, *The TEXbook*, Addison Wesley, 1986.
- The TEX Users Group home page: http://www.tug.org

`Interruptible`
`{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a text callback routine can be interrupted by subsequently invoked callback routines. Text objects have three properties that define callback routines: `ButtonDownFcn`, `CreateFcn`, and `DeleteFcn`. See the `BusyAction` property for information on how MATLAB executes callback routines.

`LineStyle`
`{-} | -- | : | -. | none`

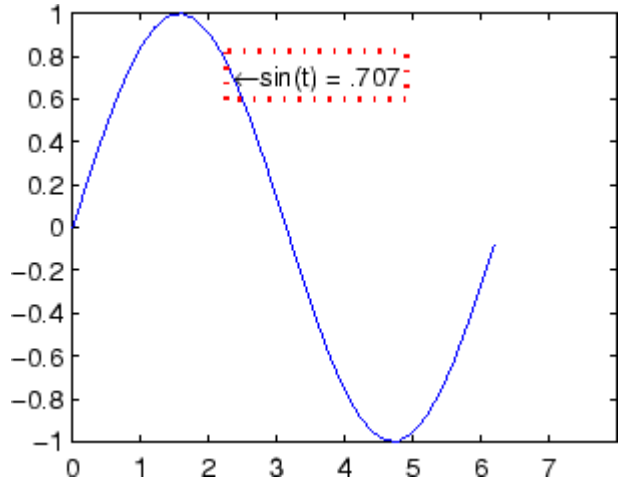
Edge line type. This property determines the line style used to draw the edges of the text `Extent`. The available line styles are shown in the following table.

Symbol	Line Style
-	Solid line (default)
--	Dashed line
:	Dotted line
-.	Dash-dot line
none	No line

For example, the following code draws a red rectangle with a dotted line style around text that labels a plot.

```
text(3*pi/4,sin(3*pi/4),...
     '\leftarrowsin(t) = .707',...
     'EdgeColor','red',...
     'LineWidth',2,...
```

```
'LineStyle',':');
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `EdgeColor` — Color of the rectangle's edge (`none` by default)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

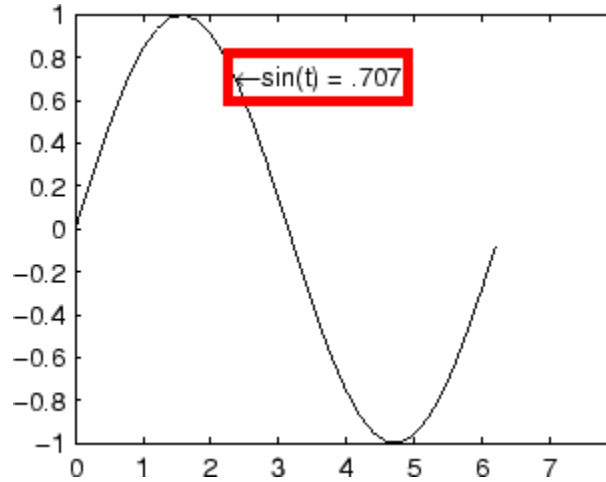
`LineWidth`

scalar (points)

Width of line used to draw text extent rectangle. When you set the text `EdgeColor` property to a color (the default is `none`), MATLAB

displays a rectangle around the text `Extent`. Use the `LineWidth` property to specify the width of the rectangle edge. For example, the following code draws a red rectangle around text that labels a plot and specifies a line width of 3 points:

```
text(3*pi/4,sin(3*pi/4),...
'\leftarrowsin(t) = .707',...
'EdgeColor','red',...
'LineWidth',3);
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `EdgeColor` — Color of the rectangle's edge (`none` by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed

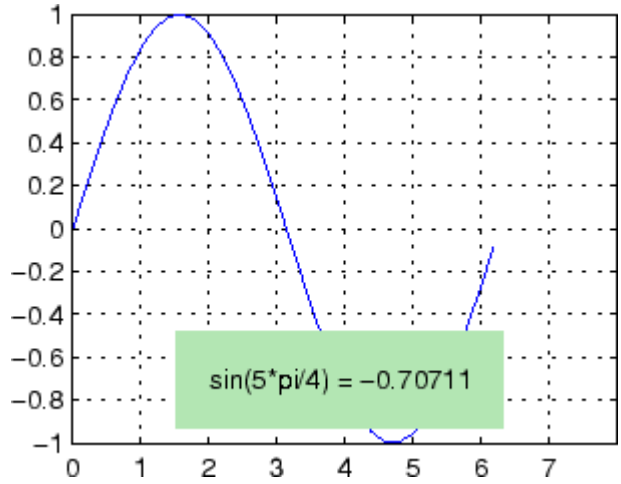
when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

`Margin`

scalar (pixels)

Distance between the text extent and the rectangle edge. When you specify a color for the `BackgroundColor` or `EdgeColor` text properties, MATLAB draws a rectangle around the area defined by the text `Extent` plus the value specified by the `Margin`. For example, the following code displays a light green rectangle with a 10-pixel margin.

```
text(5*pi/4,sin(5*pi/4),...
 ['sin(5*pi/4) = ',num2str(sin(5*pi/4))],...
 'HorizontalAlignment','center',...
 'BackgroundColor',[.7 .9 .7],...
 'Margin',10);
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `EdgeColor` — Color of the rectangle's edge (`none` by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)

See how margin affects text extent properties

This example enables you to change the values of the `Margin` property and observe the effects on the `BackgroundColor` area and the `EdgeColor` rectangle.

Click to view in editor — This link opens the MATLAB editor with the following example.

Click to run example — Use your scroll wheel to vary the `Margin`.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of text object. This property contains the handle of the text object's parent. The parent of a text object is the axes, hggroup, or hgtransform object that contains it.

See Objects That Can Contain Other Objects for more information on parenting graphics objects.

`Position`
`[x,y,[z]]`

Location of text. A two- or three-element vector, `[x y [z]]`, that specifies the location of the text in three dimensions. If you omit the `z` value, it defaults to 0. All measurements are in units specified by the `Units` property. Initial value is `[0 0 0]`.

`Rotation`
scalar (default = 0)

Text orientation. This property determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation).

`Selected`

`on | {off}`

Is object selected? When this property is set to `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also set to `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`

`{on} | off`

Objects are highlighted when selected. When the `Selected` property is set to `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is set to `off`, MATLAB does not draw the handles.

`String`

string

The text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

When the text `Interpreter` property is set to `Tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\beta	β	\phi	Φ	\leq	≤
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	♣
\epsilon	ε	\omega	ω	\diamondsuit	♦
\zeta	ζ	\Gamma	Γ	\heartsuit	♥
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	Θ	\Theta	Θ	\leftrightarrow	↔
\vartheta	ϑ	\Lambda	Λ	\leftarrow	←
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	→
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	μ	\Upsilon	ϒ	\circ	º
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∍	\div	÷
\tau	τ	\cong	≅	\neq	≠
\equiv	≡	\approx	≈	\aleph	
\Im	ℑ	\Re	ℜ	\wp	℘

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\otimes	⊗	\oplus	⊕	\oslash	∅
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in		\o	o
\rfloor	⌋	\lceil	⌈	\nabla	∇
\lfloor	⌊	\cdot	·	\ldots	...
\perp	⊥	\neg	¬	\prime	´
\wedge	∧	\times	x	\0	∅
\rceil	⌉	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	∠	\rangle	∠		

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers:

- `\bf` — Bold font
- `\it` — Italic font
- `\sl` — Oblique font (rarely available)
- `\rm` — Normal font
- `\fontname{`*`fontname`*`}` — Specify the name of the font family to use.
- `\fontsize{`*`fontsize`*`}` — Specify the font size in `FontUnits`.
- `\color(`*`colorSpec`*`)` — Specify color for succeeding characters

Stream modifiers remain in effect until the end of the string or only within the context defined by braces { }.

Specifying Text Color in TeX Strings

Use the `\color` modifier to change the color of characters following it from the previous color (which is `black` by default). Syntax is:

- `\color{colorname}` for the eight basic named colors (`red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`), and plus the four Simulink colors (`gray`, `darkGreen`, `orange`, and `lightBlue`)

  Note that short names (one-letter abbreviations) for colors are not supported by the `\color` modifier.

- `\color[rgb]{r g b}` to specify an RGB triplet with values between 0 and 1 as a cell array

For example,

```
text(.1,.5,['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```

Specifying Subscript and Superscript Characters

The subscript character "_" and the superscript character "^" modify the character or substring defined in braces immediately following.

To print the special characters used to define the TeX strings when `Interpreter` is `Tex`, prefix them with the backslash "\" character: `\\`, `\{`, `\}` `\_`, `\^`.

See the "Examples" on page 2-3305 in the `text` reference page for more information.

When `Interpreter` is set to `none`, no characters in the `String` are interpreted, and all are displayed when the text is drawn.

When `Interpreter` is set to `latex`, MATLAB provides a complete $LaTeX$ interpreter for text objects. See the `Interpreter` property for more information.

`Tag`

string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`

string (read only)

Class of graphics object. For text objects, `Type` is always the string `'text'`.

`Units`

`pixels | normalized | inches | centimeters | points | {data}`

Units of measurement. This property specifies the units MATLAB uses to interpret the `Extent` and `Position` properties. All units are measured from the lower left corner of the axes plot box.

- `Normalized` units map the lower left corner of the rectangle defined by the axes to (0,0) and the upper right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = $^1/_{72}$ inch).
- `data` refers to the data units of the parent axes as determined by the data graphed (not the axes `Units` property, which controls the positioning of the within the figure window).

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the text object. MATLAB does not use this data, but you can access it using `set` and `get`.

`UIContextMenu`
handle of a uicontextmenu object

Associate a context menu with the text. Assign this property the handle of a uicontextmenu object created in the same figure as the text. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the text.

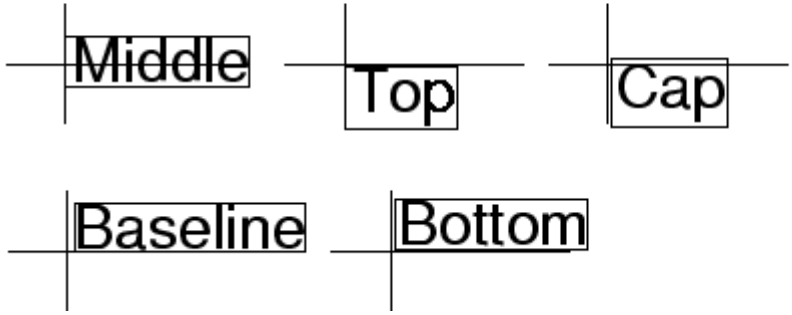
`VerticalAlignment`
`top | cap | {middle} | baseline | bottom`

Vertical alignment of text. This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean

- top — Place the top of the string' s `Extent` rectangle at the specified *y*-position.
- `cap` — Place the string so that the top of a capital letter is at the specified *y*-position.
- `middle` — Place the middle of the string at the specified *y*-position.
- `baseline` — Place font baseline at the specified *y*-position.
- `bottom` — Place the bottom of the string's `Extent` rectangle at the specified *y*-position.

The following picture illustrates the alignment options.

Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



`Visible`
`{on} | off`

Text visibility. By default, all text is visible. When set to `off`, the text is not visible, but still exists, and you can query and set its properties.

Purpose

Read data from text file; write to multiple outputs

Note The `textscan` function is intended as a replacement for both `textread` and `strread`.

Graphical Interface

As an alternative to `textread`, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

Syntax

```
[A,B,C,...] = textread('filename','format')
[A,B,C,...] = textread('filename','format',N)
[...] = textread(...,'param','value',...)
```

Description

`[A,B,C,...] = textread('filename','format')` reads data from the file `'filename'` into the variables `A,B,C,` and so on, using the specified `format`, until the entire file is read. The `filename` and `format` inputs are strings, each enclosed in single quotes. `textread` is useful for reading text files with a known format. `textread` handles both fixed and free format files.

Note When reading large text files, reading from a specific point in a file, or reading file data into a cell array rather than multiple outputs, you might prefer to use the `textscan` function.

`textread` matches and converts groups of characters from the input. Each input field is defined as a string of non-white-space characters that extends to the next white-space or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated white-space characters are treated as one.

The `format` string determines the number and types of return arguments. The number of return arguments is the number of items in the `format` string. The `format` string supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine.

Values for the `format` string are listed in the table below. White-space characters in the `format` string are ignored.

format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a file that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space or delimiter-separated string.	Cell array of strings
`%q`	Read a double quoted string, ignoring the quotes.	Cell array of strings
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings
`%[^...]`	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
`%*...` instead of `%`	Ignore the matching characters specified by `*`.	No output
`%w...` instead of `%`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

`[A,B,C,...] = textread('filename','format',N)` reads the data, reusing the `format` string `N` times, where `N` is an integer greater than zero. If `N` is smaller than zero, `textread` reads the entire file.

`[...] = textread(...,'param','value',...)` customizes `textread` using `param/value` pairs, as listed in the table below.

param	value	Action

	`' '\b`
	`\n`
	`\r`
	`\t`
		Space
		Backspace
		Newline
		Carriage return
		Horizontal tab

`bufsize`	Positive integer	Specifies the maximum string length, in bytes. Default is `4095`.
`commentstyle`	`matlab`	Ignores characters after `%`.
`commentstyle`	`shell`	Ignores characters after `#`.
`commentstyle`	`c`	Ignores characters between `/*` and `*/`.
`commentstyle`	`c++`	Ignores characters after `//`.
`delimiter`	One or more characters	Act as delimiters between elements. Default is none.
`emptyvalue`	Scalar double	Value given to empty cells when reading delimited files. Default is 0.
`endofline`	Single character or `'\r\n'`	Character that denotes the end of a line. Default is determined from file
`expchars`	Exponent characters	Default is `eEdD`.
`headerlines`	Positive integer	Ignores the specified number of lines at the beginning of the file.
`whitespace`	Any from the list below:	Treats vector of characters as white space. Default is `' \b\t'`.

Note When `textread` reads a consecutive series of `whitespace` values, it treats them as one white space. When it reads a consecutive series of `delimiter` values, it treats each as a separate delimiter.

Remarks

If you want to preserve leading and trailing spaces in a string, use the `whitespace` parameter as shown here:

```
textread('myfile.txt', '%s', 'whitespace', '')
ans =
    '   An  example      of preserving    spaces      '
```

Examples

Example 1 — Read All Fields in Free Format File Using %

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a free format file using the `%` format.

```
[names, types, x, y, answer] = textread('mydata.dat', ...
'%s %s %f %d %s', 1)
```

returns

```
names =
    'Sally'
types =
    'Level1'
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

Example 2 — Read as Fixed Format File, Ignoring the Floating Point Value

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a fixed format file, ignoring the floating-point value.

```
[names, types, y, answer] = textread('mydata.dat', ...
'%9c %5s %*f %2d %3s', 1)
```

returns

```
names =
Sally
types =
    'Level1'
y =
    45
answer =
    'Yes'
```

`%*f` in the `format` string causes `textread` to ignore the floating point value, in this case, `12.34`.

Example 3 — Read Using Literal to Ignore Matching Characters

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file, ignoring the characters `Type` in the second field.

```
[names, typenum, x, y, answer] = textread('mydata.dat', ...
'%s Type%d %f %d %s', 1)
```

returns

```
names =
    'Sally'
typenum =
    1
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

`Type%d` in the `format` string causes the characters `Type` in the second field to be ignored, while the rest of the second field is read as a signed integer, in this case, `1`.

Example 4 — Specify Value to Fill Empty Cells

For files with empty cells, use the `emptyvalue` parameter. Suppose the file `data.csv` contains:

```
1,2,3,4,,6
7,8,9,,11,12
```

Read the file using `NaN` to fill any empty cells:

```
data = textread('data.csv', '', 'delimiter', ',', ...
                'emptyvalue', NaN);
```

Example 5 — Read M-File into a Cell Array of Strings

Read the file `fft.m` into cell array of strings.

```
file = textread('fft.m', '%s', 'delimiter', '\n', ...
                'whitespace', '');
```

See Also

`textscan`, `dlmread`, `csvread`, `strread`, `fscanf`

Purpose

Read formatted data from text file or string

Syntax

```
C = textscan(fid, 'format')
C = textscan(fid, 'format', N)
C = textscan(fid, 'format', param, value, ...)
C = textscan(fid, 'format', N, param, value, ...)
C = textscan(str, ...)
[C, position] = textscan(...)
```

Description

Note Before reading a file with `textscan`, you must open the file with the `fopen` function. `fopen` supplies the `fid` input required by `textscan`. When you are finished reading from the file, you should close the file by calling `fclose(fid)`.

`C = textscan(fid, 'format')` reads data from an open text file identified by file identifier `fid` into cell array `C`. MATLAB parses the data into fields and converts it according to the conversion specifiers in `format`. The `format` input is a string enclosed in single quotes. These conversion specifiers determine the type of each cell in the output cell array. The number of specifiers determines the number of cells in the cell array.

`C = textscan(fid, 'format', N)` reads data from the file, reusing the `format` conversion specifier `N` times, where `N` is a positive integer. You can resume reading from the file after `N` cycles by calling `textscan` again using the original `fid`.

`C = textscan(fid, 'format', param, value, ...)` reads data from the file using nondefault parameter settings specified by one or more pairs of `param` and `value` arguments. The section "User Configurable Options" on page 2-3353 lists all valid parameter strings, value descriptions, and defaults.

`C = textscan(fid, 'format', N, param, value, ...)` reads data from the file, reusing the `format` conversion specifier `N` times, and using

nondefault parameter settings specified by pairs of `param` and `value` arguments.

`C = textscan(str, ...)` reads data from string `str` in exactly the same way as it does when reading from a file. You can use the `format`, `N`, and parameter/value arguments described above with this syntax. Unlike when reading from a file, if you call `textscan` more than once on the same string, it does not resume reading where the last call left off but instead reads from the beginning of the string each time.

`[C, position] = textscan(...)` returns the location of the file or string position as the second output argument. For a file, this is exactly equivalent to calling `ftell(fid)` after making the call to `textscan`. For a string, it indicates how many characters were read.

The Difference Between the textscan and textread Functions

The `textscan` function differs from `textread` in the following ways:

- The `textscan` function offers better performance than `textread`, making it a better choice when reading large files.
- With `textscan`, you can start reading at any point in the file. Once the file is open, (`textscan` requires that you open the file first), you can `fseek` to any position in the file and begin the scan at that point. The `textread` function requires that you start reading from the beginning of the file.
- Subsequent `textscan` operations start reading the file at the point where the last scan left off. The `textread` function always begins at the start of the file, regardless of any prior `textread` operations.
- `textscan` returns a single cell array regardless of how many fields you read. With `textscan`, you don't need to match the number of output arguments to the number of fields being read as you would with `textread`.
- `textscan` offers more choices in how the data being read is converted.
- `textscan` offers more user-configurable options.

Field Delimiters

The `textscan` function sees a text file as a collection of blocks. Each block consists of a number of internally consistent fields. Each field consists of a group of characters delimited by a field delimiter character. Fields can span a number of rows. Each row is delimited by an end-of-line (`EOL`) character sequence.

The default field delimiter is the white-space character, (i.e., any character that returns `true` from a call to the `isspace` function). You can set the delimiter to a different character by specifying a `'delimiter'` parameter in the `textscan` command (see "User Configurable Options" on page 2-3353). If a nondefault delimiter is specified, repeated delimiter characters are treated as separate delimiters. When using the default delimiter, repeated white-space characters are treated as a single delimiter.

The default end-of-line character sequence depends on which operating system you are using. You can change the end-of-line setting to a different character sequence by specifying an `'endofline'` parameter in the `textscan` command (see "User Configurable Options" on page 2-3353).

Conversion Specifiers

This table shows the conversion type specifiers supported by `textscan`.

Specifier	Description
`%n`	Read a number and convert to `double`.
`%d`	Read a number and convert to `int32`.
`%d8`	Read a number and convert to `int8`.
`%d16`	Read a number and convert to `int16`.
`%d32`	Read a number and convert to `int32`.
`%d64`	Read a number and convert to `int64`.
`%u`	Read a number and convert to `uint32`.

Specifier	Description
`%u8`	Read a number and convert to `uint8`.
`%u16`	Read a number and convert to `uint16`.
`%u32`	Read a number and convert to `uint32`.
`%u64`	Read a number and convert to `uint64`.
`%f`	Read a number and convert to `double`.
`%f32`	Read a number and convert to `single`.
`%f64`	Read a number and convert to `double`.
`%s`	Read a string.
`%q`	Read a (possibly double-quoted) string.
`%c`	Read one character, including white space.
`%[...]`	Read characters that match characters between the brackets. Stop reading at the first nonmatching character. Use `%[]...]` to include `]` in the set.
`%[^...]`	Read characters that do not match characters between the brackets. Stop reading at the first matching character. Use `%[^]...]` to exclude `]` from the set.
`%*n...`	Ignore `n` characters of the field, where `n` is an integer less than or equal to the number of characters in the field (e.g., `%*4s`).

Specifying Field Length

To read a certain number of characters or digits from a field, specify that number directly following the percent sign. For example, if the file you are reading contains the string

```
'Blackbird singing in the dead of night'
```

then the following command returns only five characters of the first field:

```
C = textscan(fid, '%5s', 1);
C{:}
ans =
   'Black'
```

If you continue reading from the file, `textscan` resumes the operation at the point in the string where you left off. It applies the next format specifier to that portion of the field. For example, execute this command on the same file:

```
C = textscan(fid, '%s %s', 1);
```

Note Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

`textscan` reads starting from where it left off and continues to the next whitespace, returning `'bird'`. The second `%s` reads the word `'singing'`.

The results are

```
C{:}
ans =
    'bird'
ans =
    'singing'
```

Skipping Fields

To skip any field, put an asterisk directly after the percent sign. MATLAB does not create an output cell for any fields that are skipped.

Refer to the example from the last section, where the file you are reading contains the string

```
'Blackbird singing in the dead of night'
```

Seek to the beginning of the file and reread the line, this time skipping the second, fifth, and sixth fields:

```
fseek(fid, 0, -1);
C = textscan(fid, '%s %*s %s %s %*s %*s %s', 1);
```

C is a cell array of cell arrays, each containing a string. Piece together the string and display it:

```
str = '';
for k = 1:length(C)
   str = [str char(C{k}) ' '];
   if k == 4,  disp(str),  end
end

Blackbird in the night
```

Skipping Literal Strings

In addition to skipping entire fields, you can have `textscan` skip leading literal characters in a string. Reading a file containing the following data,

```
Sally    Level1  12.34
Joe      Level2  23.54
Bill     Level3  34.90
```

this command removes the substring `'Level'` from the output and converts the level number to a `uint8`:

```
C = textscan(fid, '%s Level%u8 %f');
```

This returns a cell array `C` with the second cell containing only the unsigned integers:

```
C{1} = {'Sally'; 'Joe'; 'Bill'}       class cell
C{2} = [1; 2; 3]                      class uint8
C{3} = [12.34; 23.54; 34.90]          class double
```

Specifying Numeric Field Length and Decimal Digits

With numeric fields, you can specify the number of digits to read in the same manner described for strings in the section "Specifying Field Length" on page 2-3347. The next example uses a file containing the line

```
'405.36801 551.94387 298.00752 141.90663'
```

This command returns the starting 7 digits of each number in the line. Note that the decimal point counts as a digit.

```
C = textscan(fid, '%7f32 %*n');
C{:} =
   [405.368; 551.943; 298.007; 141.906]
```

You can also control the number of digits that are read to the right of the decimal point for any numeric field of type `%f`, `%f32`, or `%f64`. The format specifier in this command uses a `%9.1` prefix to cause `textscan` to read the first 9 digits of each number, but only include 1 digit of the decimal value in the number it returns:

```
C = textscan(fid, '%9.1f32 %*n');
C{:} =
   [405.3; 551.9; 298.0; 141.9]
```

Conversion of Numeric Fields

This table shows how `textscan` interprets the numeric field specifiers.

Format Specifier	Action Taken
`%n`, `%d`, `%u`, `%f`, and variants thereof	Read to the first delimiter. Example: `%n` reads `'473.238 '` as `473.238`.

Format Specifier	Action Taken
`%Nn`, `%Nd`, `%Nu`, `%Nf`, and variants thereof	Read `N` digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Example: `%5f32` reads `'473.238 '` as `473.2`.
Specifiers that start with `%N.Df`	Read `N` digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Return `D` decimal digits in the output. Example: `%7.2f` reads `'473.238 '` as `473.23`.

Conversion specifiers `%n`, `%d`, `%u`, `%f`, or any variant thereof (e.g., `%d16`) return a `K`-by-1 MATLAB numeric vector of the type indicated by the conversion specifier, where `K` is the number of times that specifier was found in the file. `textscan` converts the numeric fields from the field content to the output type according to the conversion specifier and MATLAB rules regarding overflow and truncation. `NaN`, `Inf`, and `-Inf` are converted according to applicable MATLAB rules.

`textscan` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. Valid forms for a complex number are

Form	Example
`–<real>–<imag>i|j`	`5.7-3.1i`
`–<imag>i|j`	`-7j`

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

Conversion of Strings

This table shows how `textscan` interprets the string field specifiers.

Format Specifier	Action Taken
`%s or %q`	Read to the first delimiter. Example: %s reads 'summer ' as 'summer'.
`%Ns or %Nq`	Read N characters, or to the first delimiter, whichever comes first. Example: %3s reads 'summer ' as 'sum'.
`%[abc]`	Read those characters that match any character specified within the brackets, stopping just before the first character that does not match. Example: `%[mus]` reads `'summer '` as `'summ'`.
`%N[abc]`	Read as many as `N` characters that match any character specified within the brackets, stopping just before the first character that does not match. Example: `%2[mus]` reads `'summer'` as `'su'`.
`%[^abc]`	Read those characters that do not match any character specified within the brackets, stopping just before the first character that does match. Example: `%[^xrg]` reads `'summer '` as `'summe'`.
`%N[^abc]`	Read as many as `N` characters that do not match any character specified within the brackets, stopping just before the first character that does match. Example: `%2[^xrg]` reads `'summer '` as `'su'`.

Conversion specifiers `%s`, `%q`, `%[...]`, and `%[^...]` return a `K`-by-1 MATLAB cell vector of strings, where `K` is the number of times that specifier was found in the file. If you set the `delimiter` parameter to a non-white-space character, or set the `whitespace` parameter to `''`, `textscan` returns all characters in the string field, including white-space. Otherwise each string terminates at the beginning of white-space.

Conversion of Characters

This table shows how textscan interprets the character field specifiers.

Format Specifier	Action Taken
%c	Read one character. Example: %c reads 'Let's go!' as 'L'.
%Nc	Read N characters, including delimiter characters. Example: %9c reads 'Let's go!' as 'Let's go!'.

Conversion specifier %Nc returns a K-by-N MATLAB character array, where K is the number of times that specifier was found in the file. textscan returns all characters, including white-space, but excluding the delimiter.

Conversion of Empty Fields

An empty field in the text file is defined by two adjacent delimiters indicating an empty set of characters, or, in all cases except %c, white-space. The empty field is returned as NaN by default, but is user definable. In addition, you may specify custom strings to be used as empty values, in *numeric fields only*. textscan does not examine nonnumeric fields for custom empty values. See "User Configurable Options" on page 2-3353.

Note MATLAB represents integer NaN as zero. If textscan reads an empty field that is assigned an integer format specifier (one that starts with %d or %u), it returns the empty value as zero rather than as NaN. (See the value returned in C{5} in Example 6 — Using a Nondefault Empty Value.

User Configurable Options

This table shows the valid param-value options and their default values. Parameter names are not case-sensitive.

Parameter	Value	Default
`BufSize`	Maximum string length in bytes	4095
`CollectOutput`	If `true`, MATLAB concatenates consecutive cells of the output that have the same data type into a single array.	0 (`false`)
`CommentStyle`	Symbol(s) designating text to be ignored (see "Values for commentStyle" on page 2-3355, below)	None
`Delimiter`	Delimiter characters	Whitespace
`EmptyValue`	Empty cell value in delimited files	`NaN`
`endOfLine`	End-of-line character	Determined from the file
`expChars`	Exponent characters	`'eEdD'`
`HeaderLines`	Number of lines to skip. (This includes the remainder of the current line, unless you are positioned at the beginning of the file.)	0
`MultipleDelimsAsOne`	If set to 1, `textread` treats consecutive delimiters as a single delimiter. If set to 0, `textread` treats them as separate delimiters. Only valid if the `delimiter` option is specified.	`0`

Parameter	Value	Default
`ReturnOnError`	Behavior on failing to read or convert (`1=true`, or 0)	`1`
`TreatAsEmpty`	String(s) to be treated as an empty value. A single string or cell array of strings can be used.	None
`Whitespace`	White-space characters	`' \b\t'`

White-Space Characters

Leading white-space characters are not included in the processing of any of the data fields. When processing numeric data, trailing whitespace is also assumed to have no significance.

Values for commentStyle

Possible values for the `commentStyle` parameter are

Value	Description	Example
Single string, `S`	Ignore any characters that follow string `S` and are on the same line.	`'%', '//'`
Cell array of two strings, `C`	Ignore any characters that lie between the opening and closing strings in `C`.	`{'/*', '*/'}, {'/%', '%/'}`

Resuming a Text Scan

If `textscan` fails to convert a data field, it stops reading and returns all fields read before the failure. When reading from a file, you can resume reading from the same file by calling `textscan` again using the same file identifier, `fid`. When reading from a string, the two-output argument syntax enables you to resume reading from the string at the

point where the last read terminated. The following command is an example of how you can do this:

```
textscan(str(position+1:end), ...)
```

Remarks

For information on how to use `textscan` to import large data sets, see "Reading Files with Large Data Sets" in the MATLAB Programming documentation.

Examples

Example 1 — Reading Different Types of Data

Text file `scan1.dat` contains data in the following form:

```
Sally  Level1 12.34 45 1.23e10 inf NaN Yes
Joe    Level2 23.54 60 9e19 -inf 0.001 No
Bill   Level3 34.90 12 2e5 10 100 No
```

Read each column into a variable:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%s %s %f32 %d8 %u %f %f %s');
fclose(fid);
```

Note Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

`textscan` returns a 1-by-8 cell array `C` with the following cells:

```
C{1} = {'Sally'; 'Joe'; 'Bill'}          class cell
C{2} = {'Level1'; 'Level2'; 'Level3'}    class cell
C{3} = [12.34; 23.54; 34.9]              class single
C{4} = [45; 60; 12]                      class int8
C{5} = [4294967295; 4294967295; 200000]  class uint32
C{6} = [Inf; -Inf; 10]                   class double
C{7} = [NaN; 0.001; 100]                 class double
C{8} = {'Yes'; 'No'; 'No'}               class cell
```

The first two elements of `C{5}` are the maximum values for a 32-bit unsigned integer, or `intmax('uint32')`.

Example 2 — Reading All But One Field

Read the file as a fixed-format file, skipping the third field:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%7c %6s %*f %d8 %u %f %f %s');
fclose(fid);
```

`textscan` returns a 1-by-8 cell array `C` with the following cells:

```
C{1} = ['Sally  '; 'Joe    '; 'Bill   ']   class char
C{2} = {'Level1'; 'Level2'; 'Level3'}      class cell
C{3} = [45; 60; 12]                        class int8
C{4} = [4294967295; 4294967295; 200000]    class uint32
C{5} = [Inf; -Inf; 10]                     class double
C{6} = [NaN; 0.001; 100]                   class double
C{7} = {'Yes'; 'No'; 'No'}                 class cell
```

Example 3 — Reading Only the First Field

Read the first column into a cell array, skipping the rest of the line:

```
fid = fopen('scan1.dat');
names = textscan(fid, '%s%*[^\n]');
fclose(fid);
```

`textscan` returns a 1-by-1 cell array `names`:

```
size(names)
ans =
     1     1
```

The one cell contains

```
names{1} = {'Sally'; 'Joe'; 'Bill'}       class cell
```

Example 4 — Removing a Literal String in the Output

The second `format` specifier in this example, `%sLevel`, tells `textscan` to read the second field from a line in the file, but to ignore the initial string `'Level'` within that field. All that is left of the field is a numeric digit. `textscan` assigns the next specifier, `%f`, to that digit, converting it to a `double`.

See `C{2}` in the results:

```
fid = fopen('scan1.dat');
C = textscan(fid, '%s Level%u8 %f32 %d8 %u %f %f %s');
fclose(fid);
```

`textscan` returns a 1-by-8 cell array, `C`, with cells

```
C{1} = {'Sally'; 'Joe'; 'Bill'}          class cell
C{2} = [1; 2; 3]                         class uint8
C{3} = [12.34; 23.54; 34.90]             class single
C{4} = [45; 60; 12]                      class int8
C{5} = [4294967295; 4294967295; 200000]  class uint32
C{6} = [Inf; -Inf; 10]                   class double
C{7} = [NaN; 0.001; 100]                 class double
C{8} = {'Yes'; 'No'; 'No'}               class cell
```

Example 5 — Using a Nondefault Delimiter and White-Space

Read the M-file into a cell array of strings:

```
fid = fopen('fft.m');
file = textscan(fid, '%s', 'delimiter', '\n', ...
                'whitespace', '');
fclose(fid);
```

`textscan` returns a 1-by-1 cell array, `file`, that contains a 37-by-1 cell array:

```
file =
    {37x1 cell}
```

Show some of the text from the first three lines of the file:

```
lines = file{1};
lines{1:3, :}
ans =
%FFT Discrete Fourier transform.
ans =
%   FFT(X) is the discrete Fourier transform (DFT) of vector X.  For
ans =
%   matrices, the FFT operation is applied to each column. For N-D
```

Example 6 — Using a Nondefault Empty Value

Read files with empty cells, setting the `emptyvalue` parameter. The file `data.csv` contains

```
1,  2,  3,  4,   ,  6
7,  8,  9,   , 11, 12
```

Read the file as shown here, using `-Inf` in empty cells:

```
fid = fopen('data.csv');
C = textscan(fid, '%f%f%f%f%u32%f', 'delimiter', ',', ...
             'emptyValue', -Inf);
fclose(fid);
```

`textscan` returns a 1-by-6 cell array `C` with the following cells:

```
C{1} = [1; 7]            class double
C{2} = [2; 8]            class double
C{3} = [3; 9]            class double
C{4} = [4; NaN]          class double
C{5} = [-Inf; 11]        class uint32 (-Inf converted to 0)
C{6} = [6; 12]           class double
```

Example 7 — Using Custom Empty Values and Comments

You have a file `data.csv` that contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```

Designate what should be treated as empty values and as comments. Read in all other values from the file:

```
fid = fopen('data5.csv');
C = textscan(fid, '%s%n%n%n%n', 'delimiter', ',', ...
             'treatAsEmpty', {'NA', 'na'}, ...
             'commentStyle', '//');
fclose(fid);
```

This returns the following data in cell array `C`:

```
C{:}
ans =
    'abc'
    'def'
ans =
     2
   NaN
ans =
   NaN
     5
ans =
     3
     6
ans =
     4
     7
```

Example 8 — Reading From a String

Read in a string (quoted from Albert Einstein) using `textscan`:

```
str = ...
  ['Do not worry about your difficulties in Mathematics.' ...
   'I can assure you mine are still greater.'];
```

```
s = textscan(str, '%s', 'delimiter', '.');

s{:}
ans =
    'Do not worry about your difficulties in Mathematics'
    'I can assure you mine are still greater'
```

Example 9 — Handling Multiple Delimiters

This example takes a comma-separated list of names, the test pilots known as the Mercury Seven, and uses `textscan` to return a list of their names in a cell array. When some names are removed from the input list, leaving multiple sequential delimiters, `textscan`, by default, accounts for this. If you override that default by calling `textscan` with the `multipleDelimsAsOne` option, `textscan` ignores the missing names.

Here is the full list of the astronauts:

```
Mercury7 = ...
  'Shepard,Grissom,Glenn,Carpenter,Schirra,Cooper,Slayton';
```

Remove the names Grissom and Cooper from the input string, and `textscan`, by default, does not treat the multiple delimiters as one, and returns an empty string for each missing name:

```
Mercury7 = 'Shepard,,Glenn,Carpenter,Schirra,,Slayton';
names = textscan(Mercury7, '%s', 'delimiter', ',');
names{:}'
ans =
  'Shepard' '' 'Glenn' 'Carpenter' 'Schirra' '' 'Slayton'
```

Using the same input string, but this time setting the `multipleDelimsAsOne` switch, `textscan` ignores the multiple delimiters:

```
names = textscan(Mercury7, '%s', 'delimiter', ',', ...
                 'multipledelimsasone', 1);
names{:}'
```

```
ans =
  'Shepard'  'Glenn'  'Carpenter'  'Schirra'  'Slayton'
```

Example 10 — Using the CollectOutput Switch

Shown below are the contents of a file `wire_gage.txt`. The first line contains four column headers in text. The lines that follow that are numeric data:

```
 AWG |  Area    | Resistance |   Diameter
0000   211600       0.049        0.46
000    167810       0.0618       0.40965
00     133080       0.078        0.3648
0      105530       0.0983       0.32485
1       83694       0.124        0.2893
2       66373       0.1563       0.25763
3       52634       0.197        0.22942
4       41742       0.2485       0.20431
5       33102       0.3133       0.18194
6       26250       0.3951       0.16202
7       20816       0.4982       0.14428
8       16509       0.6282       0.12849
9       13094       0.7921       0.11443
10      10381       0.9989       0.10189
```

When you read the file with `textscan` having the `CollectOutput` switch set to zero, MATLAB returns each column of the numeric data in a separate 44-by-1cell array:

```
format long g
fid = fopen('wire_gage.txt', 'r');

C_text = textscan(fid, '%s', 4, 'delimiter', '|');

C_data0 = textscan(fid, '%d %f %f %f', 'CollectOutput', 0)
C_data0 =
  [44x1 int32]  [44x1 double]  [44x1 double]  [44x1 double]
```

Reading the file with `CollectOutput` set to one collects all data of a common type, `double` in this case, into a single 44-by-3 cell array:

```
frewind(fid)

C_text = textscan(fid, '%s', 4, 'delimiter', '|');

C_data1 = textscan(fid, '%d %f %f %f', 'CollectOutput', 1)
C_data1 =
    [44x1 int32]    [44x3 double]
```

See Also

`dlmread`, `dlmwrite`, `xlswrite`, `fopen`, `fseek`, `importdata`

Purpose

Wrapped string matrix for given uicontrol

Syntax

```
outstring = textwrap(h,instring)
[outstring,position]=textwrap(h,instring)
```

Description

`outstring = textwrap(h,instring)` returns a wrapped string cell array, `outstring`, that fits inside the uicontrol with handle `h`. `instring` is a cell array, with each cell containing a single line of text. `outstring` is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.

`[outstring,position]=textwrap(h,instring)` returns the recommended position of the uicontrol in the units of the uicontrol. `position` considers the extent of the multiline text in the x and y directions.

Example

Place a text-wrapped string in a uicontrol:

```
pos = [10 10 100 10];
h = uicontrol('Style','Text','Position',pos);
string = {'This is a string for the uicontrol.',
          'It should be correctly wrapped inside.'};
[outstring,newpos] = textwrap(h,string);
pos(4) = newpos(4);
set(h,'String',outstring,'Position',[pos(1),pos(2),pos(3)+10,po
s(4)])
```

See Also

`uicontrol`

Purpose

Terminate function and issue exception

Syntax

```
throw(ME)
```

Description

`throw(ME)` terminates the currently running function, issues an exception based on `MException` object `ME`, and returns control to the keyboard or to any enclosing `catch` block. A thrown `MException` displays a message in the Command Window unless it is caught by `try-catch`. `throw` also sets the `MException` stack field to the location from which the `throw` method was called.

Examples

Example 1

This example tests the output of M-file `evaluate_plots` and throws an exception if it is not acceptable:

```
[minval, maxval] = evaluate_plots(p24, p28, p41);
if minval < lower_bound || maxval > upper_bound
    ME = MException('VerifyOutput:OutOfBounds', ...
       'Results are outside the allowable limits');
    throw(ME);
end
```

Example 2

This example attempts to open a file in a directory that is not on the MATLAB path. It uses a nested `try-catch` block to give the user the opportunity to extend the path. If the still cannot be found, the program issues an exception with the first error appended to the second:

```
function data = read_it(filename);
try
   fid = fopen(filename, 'r');
   data = fread(fid);
catch eObj1
   if strcmp(eObj1.identifier, 'MATLAB:FileIO:InvalidFid')
      msg = sprintf('\n%s%s%s', 'Cannot open file ', ...
         filename, '. Try another location?  ');
```

```
        reply = input(msg, 's')
        if reply(1) == 'y'
            newdir = input('Enter directory name:  ', 's');
        else
            throw(eObj1);
        end
        addpath(newdir);
        try
           fid = fopen(filename, 'r');
           data = fread(fid);
        catch eObj2
           eObj3 = addCause(eObj2, eObj1)
           throw(eObj3);
        end
        rmpath(newdir);
     end
  end
  fclose(fid);
```

If you run this function in a `try-catch` block at the command line, you can look at the `MException` object by assigning it to a variable (`e`) with the `catch` command.

```
  try
     d = read_it('anytextfile.txt');
  catch e
  end

  e
  e =
   MException object with properties:

      identifier: 'MATLAB:FileIO:InvalidFid'
         message: 'Invalid file identifier.  Use fopen to
  generate a valid file identifier.'
           stack: [1x1 struct]
           cause: {[1x1 MException]}
```

```
    Cannot open file anytextfile.txt. Try another location?y
  Enter directory name:  xxxxxxx
  Warning: Name is nonexistent or not a directory: xxxxxxx.
  > In path at 110
    In addpath at 89
```

See Also

error, try, catch, assert, MException, rethrow(MException), throwAsCaller(MException), addCause(MException), getReport(MException), disp(MException), isequal(MException), eq(MException), ne(MException), last(MException),

throwAsCaller (MException)

Purpose

Throw exception, as if from calling function

Syntax

```
throwAsCaller(ME)
```

Description

`throwAsCaller(ME)` throws an exception from the currently running M-file based on `MException` object `ME`. MATLAB exits the currently running function and returns control to either the keyboard or an enclosing `catch` block in a calling function. Unlike the `throw` function, MATLAB omits the current stack frame from the `stack` field of the `MException`, thus making the exception look as if it is being thrown by the caller of the function.

In some cases, it is not relevant to show the person running your program the true location that generated an exception, but is better to point to the calling function where the problem really lies. You might also find `throwAsCaller` useful when you want to simplify the error display, or when you have code that you do not want made public.

Examples

The function `klein_bottle`, in this example, generates a Klein Bottle figure by revolving the figure-eight curve defined by XYKLEIN. It defines a few variables and calls the function `draw_klein`, which executes three functions in a `try-catch` block. If there is an error, the `catch` block issues an exception using either `throw` or `throwAsCaller`:

```
function klein_bottle(ab, pq)
rtr = [2 0.5 1];
box = [-3 3 -3 3 -2 2];
vue = [55 60];
draw_klein(ab, rtr, pq, box, vue)


function draw_klein(ab, rtr, pq, box, vue)
clf
try
   tube('xyklein',ab, rtr, pq, box, vue);
   shading interp
   colormap(pink);
```

```
catch ME
   throw(ME)
%   throwAsCaller(ME)
end
```

Call the `klein_bottle` function, passing an incorrect value for the second argument. (The correct value would be a vector, such as `[40 40]`.) Because the `catch` block issues the exception using `throw`, MATLAB displays error messages for line 15 of function `draw_klein`, and for line 5 of function `klein_bottle`:

```
klein_bottle(ab, pi)
??? Attempted to access pq(2); index out of bounds because
   numel(pq)=1.

Error in ==> klein_bottle>draw_klein at 15
   throw(ME);

Error in ==> klein_bottle at 5
draw_figure(ab, rtr, pq, box, vue)
```

Run the function again, this time changing the `klein_bottle.m` file so that the `catch` block uses `throwAsCaller` instead of `throw`. This time, MATLAB only displays the error at line 5 of the main program:

```
klein_bottle(ab, pi)
??? Attempted to access pq(2); index out of bounds because
   numel(pq)=1.

Error in ==> klein_bottle at 5
draw_figure(ab, rtr, pq, box, vue)
```

See Also

`error`, `try`, `catch`, `assert`, `MException`, `throw(MException)`, `rethrow(MException)`, `addCause(MException)`, `getReport(MException)`, `disp(MException)`, `isequal(MException)`, `eq(MException)`, `ne(MException)`, `last(MException)`

Purpose

Measure performance using stopwatch timer

Syntax

```
tic
   any statements
toc
t = toc
```

Description

`tic` starts a stopwatch timer.

`toc` prints the elapsed time since `tic` was used.

`t = toc` returns the elapsed time in `t`.

Remarks

The `tic` and `toc` functions work together to measure elapsed time. `tic` saves the current time that `toc` uses later to measure the elapsed time. The sequence of commands

```
tic
operations
toc
```

measures the amount of time MATLAB takes to complete one or more `operations`, and displays the time in seconds.

Examples

This example measures how the time required to solve a linear system varies with the order of a matrix.

```
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic
    x = A\b;
    t(n) = toc;
end
plot(t)
```

See Also

`clock`, `cputime`, `etime`, `profile`

Purpose

Construct timer object

Syntax

```
T = timer
T = timer('PropertyName1', PropertyValue1, 'PropertyName2',
   PropertyValue2,...)
```

Description

`T = timer` constructs a timer object with default attributes.

`T = timer('PropertyName1', PropertyValue1, 'PropertyName2', PropertyValue2,...)` constructs a timer object in which the given property name/value pairs are set on the object. See "Timer Object Properties" on page 2-3371 for a list of all the properties supported by the timer object.

Note that the property name/property value pairs can be in any format supported by the `set` function, i.e., property/value string pairs, structures, and property/value cell array pairs.

Examples

This example constructs a timer object with a timer callback function handle, `mycallback`, and a 10 second interval.

```
t = timer('TimerFcn',@mycallback, 'Period', 10.0);
```

See Also

`delete(timer)`, `disp(timer)`, `get(timer)`, `isvalid(timer)`, `set(timer)`, `start`, `startat`, `stop`, `timerfind`, `timerfindall`, `wait`

Timer Object Properties

The timer object supports the following properties that control its attributes. The table includes information about the data type of each property and its default value.

To view the value of the properties of a particular timer object, use the `get(timer)` function. To set the value of the properties of a timer object, use the `set(timer)` function.

Property Name	Property Description	Data Types, Values, Defaults, Access	
`AveragePeriod`	Average time between `TimerFcn` executions since the timer started. Note: Value is `NaN` until timer executes two timer callbacks.	Data type	`double`
		Default	`NaN`
		Read only	Always
`BusyMode`	Action taken when a timer has to execute `TimerFcn` before the completion of previous execution of `TimerFcn`. `'drop'` — Do not execute the function `'error'` — Generate an error `'queue'` — Execute function at next opportunity.	Data type	Enumerated string

		Values	`'drop'`
			`'error'`
			`'queue'`

		Default	`'drop'`
		Read only	While `Running = 'on'`
`ErrorFcn`	Function that the timer executes when an error occurs. This function executes before the `StopFcn`. See “Creating Callback Functions” for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`ExecutionMode`	Determines how the timer object schedules timer events. See "Timer Object Execution Modes" for more information.	Data type	Enumerated string

		Values	`'singleShot'`
			`'fixedDelay'`
			`'fixedRate'`
			`'fixedSpacing'`

		Default	`'singleShot'`
		Read only	While `Running = 'on'`
`InstantPeriod`	The time between the last two executions of `TimerFcn`.	Data type	`double`
		Default	`NaN`
		Read only	Always
`Name`	User-supplied name.	Data type	Text string
		Default	`'timer-`*`i`*`'`, where *`i`* is a number indicating the *`i`*th timer object created this session. To reset *`i`* to 1, execute the `clear classes` command.
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`ObjectVisibility`	Provides a way for application developers to prevent end-user access to the timer objects created by their application. The `timerfind` function does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that created it), you can set its properties.	Data type	Enumerated string
		Values	`'off'` `'on'`
		Default	`'on'`
		Read only	Never
`Period`	Specifies the delay, in seconds, between executions of `TimerFcn`.	Data type	`double`
		Value	Any number >= 0.001
		Default	1.0
		Read only	While `Running = 'on'`
`Running`	Indicates whether the timer is currently executing.	Data type	Enumerated string
		Values	`'off'` `'on'`
		Default	`'off'`
		Read only	Always

Property Name	Property Description	Data Types, Values, Defaults, Access	
`StartDelay`	Specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`.	Data type	`double`
		Values	Any number >= 0
		Default	0
		Read only	While `Running` = `'on'`
`StartFcn`	Function the timer calls when it starts. See “Creating Callback Functions” for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`StopFcn`	Function the timer calls when it stops. The timer stops when • You call the timer `stop` function • The timer finishes executing `TimerFcn`, i.e., the value of `TasksExecuted` reaches the limit set by `TasksToExecute`. • An error occurs (The `ErrorFcn` is called first, followed by the `StopFcn`.) See "Creating Callback Functions" for more information.	Date type	Text string, function handle, or cell array
		Default	None
		Read only	Never
`Tag`	User supplied label.	Data type	Text string
		Default	Empty string (`''`)
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`TasksToExecute`	Specifies the number of times the timer should execute the function specified in the `TimerFcn` property.	Data type	`double`
		Values	Any number > 0
		Default	1
		Read only	Never
`TasksExecuted`	The number of times the timer has called `TimerFcn` since the timer was started.	Data type	`double`
		Values	Any number >= 0
		Default	0
		Read only	Always
`TimerFcn`	Timer callback function. See “Creating Callback Functions” for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never
`Type`	Identifies the object type.	Data type	Text string
		Values	`'timer'`
		Read only	Always
`UserData`	User-supplied data.	Data type	User-defined
		Default	`[]`
		Read only	Never

Purpose

Find timer objects

Syntax

```
out = timerfind
out = timerfind('P1', V1, 'P2', V2,...)
out = timerfind(S)
out = timerfind(obj, 'P1', V1, 'P2', V2,...)
```

Description

`out = timerfind` returns an array, `out`, of all the timer objects that exist in memory.

`out = timerfind('P1', V1, 'P2', V2,...)` returns an array, `out`, of timer objects whose property values match those passed as parameter/value pairs, `P1`, `V1`, `P2`, `V2`. Parameter/value pairs may be specified as a cell array.

`out = timerfind(S)` returns an array, `out`, of timer objects whose property values match those defined in the structure, `S`. The field names of `S` are timer object property names and the field values are the corresponding property values.

`out = timerfind(obj, 'P1', V1, 'P2', V2,...)` restricts the search for matching parameter/value pairs to the timer objects listed in `obj`. `obj` can be an array of timer objects.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `timerfind`.

Note that, for most properties, `timerfind` performs case-sensitive searches of property values. For example, if the value of an object's `Name` property is `'MyObject'`, `timerfind` will not find a match if you specify `'myobject'`. Use the `get` function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, `timerfind` will find an object with an `ExecutionMode` property value of `'singleShot'` or `'singleshot'`.

Examples

These examples use `timerfind` to find timer objects with the specified property values.

```
t1 = timer('Tag', 'broadcastProgress', 'Period', 5);
t2 = timer('Tag', 'displayProgress');
out1 = timerfind('Tag', 'displayProgress')
out2 = timerfind({'Period', 'Tag'}, {5, 'broadcastProgress'})
```

See Also

`get(timer)`, `timer`, `timerfindall`

Purpose

Find timer objects, including invisible objects

Syntax

```
out = timerfindall
out = timerfindall('P1', V1, 'P2', V2,...)
out = timerfindall(S)
out = timerfindall(obj, 'P1', V1, 'P2', V2,...)
```

Description

`out = timerfindall` returns an array, `out`, containing all the timer objects that exist in memory, regardless of the value of the object's `ObjectVisibility` property.

`out = timerfindall('P1', V1, 'P2', V2,...)` returns an array, `out`, of timer objects whose property values match those passed as parameter/value pairs, `P1`, `V1`, `P2`, `V2`. Parameter/value pairs may be specified as a cell array.

`out = timerfindall(S)` returns an array, `out`, of timer objects whose property values match those defined in the structure, `S`. The field names of `S` are timer object property names and the field values are the corresponding property values.

`out = timerfindall(obj, 'P1', V1, 'P2', V2,...)` restricts the search for matching parameter/value pairs to the timer objects listed in `obj`. `obj` can be an array of timer objects.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `timerfindall`.

Note that, for most properties, `timerfindall` performs case-sensitive searches of property values. For example, if the value of an object's `Name` property is `'MyObject'`, `timerfindall` will not find a match if you specify `'myobject'`. Use the `get` function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, `timerfindall` will find an object with an `ExecutionMode` property value of `'singleShot'` or `'singleshot'`.

Examples

Create several timer objects.

```
t1 = timer;
t2 = timer;
t3 = timer;
```

Set the `ObjectVisibility` property of one of the objects to `'off'`.

```
t2.ObjectVisibility = 'off';
```

Use `timerfind` to get a listing of all the timer objects in memory. Note that the listing does not include the timer object (`timer-2`) whose `ObjectVisibility` property is set to `'off'`.

```
timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:   Name:
   1       singleShot      1        ''          timer-1
   2       singleShot      1        ''          timer-3
```

Use `timerfindall` to get a listing of all the timer objects in memory. This listing includes the timer object whose `ObjectVisibility` property is set to `'off'`.

```
timerfindall

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:   Name:
   1       singleShot      1        ''          timer-1
   2       singleShot      1        ''          timer-2
   3       singleShot      1        ''          timer-3
```

See Also

`get(timer)`, `timer`, `timerfind`

Purpose

Create `timeseries` object

Syntax

```
ts = timeseries
ts = timeseries(Data)
ts = timeseries(Name)
ts = timeseries(Data,Time)
ts = timeseries(Data,Time,Quality)
ts = timeseries(Data,...,'Parameter',Value,...)
```

Description

`ts = timeseries` creates an empty time-series object.

`ts = timeseries(Data)` creates a time series with the specified `Data`. `ts` has a default time vector that ranges from `0` to `N-1` with a 1-second interval, where `N` is the number of samples. The default name of the `timeseries` object is `'unnamed'`.

`ts = timeseries(Name)` creates an empty time series with the name specified by a string `Name`. This name can differ from the time-series variable name.

`ts = timeseries(Data,Time)` creates a time series with the specified `Data` array and `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

`ts = timeseries(Data,Time,Quality)` creates a `timeseries` object. The `Quality` attribute is an integer vector with values `-128` to `127` that specifies the quality in terms of codes defined by `QualityInfo.Code`.

`ts = timeseries(Data,...,'Parameter',Value,...)` creates a `timeseries` object with optional parameter-value pairs after the `Data`, `Time`, and `Quality` arguments. You can specify the following parameters:

- `Name` — Time-series name entered as a string
- `IsTimeFirst` — Logical value (`true` or `false`) specifying whether the first or last dimension of the data array is aligned with the time vector. You can set this property when the data array is square and, therefore, the dimension that is aligned with time is ambiguous.

- `IsDatenum` — Logical value (`true` or `false`) that when set to `true` specifies that `Time` values are dates in the format of MATLAB serial dates.

Remarks

Definition: timeseries

The time-series object, called `timeseries`, is a MATLAB variable that contains time-indexed data and properties in a single, coherent structure. For example, in addition to data and time values, you can also use the time-series object to store events, descriptive information about data and time, data quality, and the interpolation method.

Definition: Data Sample

A time-series *data sample* consists of one or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

For example, suppose that `ts.data` has the size 5-by-4-by-3 and the time vector has the length 5. Then, the number of samples is `5` and the total number of data values is `5 x 4 x 3 = 60`.

Notes About Quality

When `Quality` is a vector, it must have the same length as the time vector. In this case, each `Quality` value applies to the corresponding data sample. When `Quality` is an array, it must have the same size as the data array. In this case, each `Quality` value applies to the corresponding data value of the `ts.data` array.

Examples

Example 1 — Using Default Time Vector

Create a `timeseries` object called `'LaunchData'` that contains four data sets, each stored as a column of length `5` and using the default time vector:

```
b = timeseries(rand(5, 4),'Name','LaunchData')
```

Example 2 — Using Uniform Time Vector

Create a `timeseries` object containing a single data set of length `5` and a time vector starting at `1` and ending at `5`:

```
b = timeseries(rand(5,1),[1 2 3 4 5])
```

Example 3

Create a `timeseries` object called `'FinancialData'` containing five data points at a single time point:

```
b = timeseries(rand(1,5),1,'Name','FinancialData')
```

See Also

`addsample`, `tscollection`, `tsdata.event`, `tsprops`

Purpose

Add title to current axes

GUI Alternative

To create or modify a plot's title from a GUI, use **Insert Title** from the figure menu. Use the Property Editor, one of the plotting tools  , to modify the position, font, and other properties of a legend. For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
title('string')
title(fname)
title(...,'PropertyName',PropertyValue,...)
title(axes_handle,...)
h = title(...)
```

Description

Each `axes` graphics object can have one title. The title is located at the top and in the center of the axes.

`title('string')` outputs the string at the top and in the center of the current axes.

`title(fname)` evaluates the function that returns a string and displays the string at the top and in the center of the current axes.

`title(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the `text` graphics object that `title` creates. Do not use the `'String'` text property to set the title string; the content of the title should be given by the first argument.

`title(axes_handle,...)` adds the title to the specified axes.

`h = title(...)` returns the handle to the text object used as the title.

Examples

Display today's date in the current axes:

```
title(date)
```

Include a variable's value in a title:

```
f = 70;
c = (f-32)/1.8;
```

```
title(['Temperature is ',num2str(c),'C'])
```

Include a variable's value in a title and set the color of the title to yellow:

```
n = 3;
title(['Case number #',int2str(n)],'Color','y')
```

Include Greek symbols in a title:

```
title('\ite^{\omega\tau} = cos(\omega\tau) + isin(\omega\tau)')
```

Include a superscript character in a title:

```
title('\alpha^2')
```

Include a subscript character in a title:

```
title('X_1')
```

The text object `String` property lists the available symbols.

Create a multiline title using a multiline cell array.

```
title({'First line';'Second line'})
```

Remarks

`title` sets the `Title` property of the current axes graphics object to a new text graphics object. See the text `String` property for more information.

See Also

`gtext`, `int2str`, `num2str`, `text`, `xlabel`, `ylabel`, `zlabel`

"Annotating Plots" on page 1-87 for related functions

Text Properties for information on setting parameter/value pairs in titles

Adding Titles to Graphs for more information on ways to add titles

Purpose

Convert CDF epoch object to MATLAB datenum

Syntax

```
n = todatenum(obj)
```

Description

`n = todatenum(obj)` converts the CDF epoch object `ep_obj` into a MATLAB serial date number. Note that a CDF epoch is the number of milliseconds since `01-Jan-0000` whereas a MATLAB datenum is the number of days since `00-Jan-0000`.

Examples

Construct a CDF epoch object from a date string, and then convert the object back into a MATLAB date string:

```
dstr = datestr(today)
dstr =
   08-Oct-2003

obj = cdfepoch(dstr)
obj =
     cdfepoch object:
     08-Oct-2003 00:00:00

dstr2 = datestr(todatenum(obj))
dstr2 =
   08-Oct-2003
```

See Also

`cdfepoch`, `cdfinfo`, `cdfread`, `cdfwrite`, `datenum`

toeplitz

Purpose

Toeplitz matrix

Syntax

```
T = toeplitz(c,r)
T = toeplitz(r)
```

Description

A *Toeplitz* matrix is defined by one row and one column. A *symmetric Toeplitz* matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

`T = toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix `T` having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` are different, a message is printed and the column element is used.

`T = toeplitz(r)` returns the symmetric or Hermitian Toeplitz matrix formed from vector `r`, where `r` defines the first row of the matrix.

Examples

A Toeplitz matrix with diagonal disagreement is

```
c = [1  2  3  4  5];
r = [1.5  2.5  3.5  4.5  5.5];
toeplitz(c,r)
Column wins diagonal conflict:
ans =
    1.000    2.500    3.500    4.500    5.500
    2.000    1.000    2.500    3.500    4.500
    3.000    2.000    1.000    2.500    3.500
    4.000    3.000    2.000    1.000    2.500
    5.000    4.000    3.000    2.000    1.000
```

See Also

`hankel`, `kron`

Purpose

Root directory for specified toolbox

Syntax

```
toolboxdir('tbxdirname')
s = toolboxdir('tbxdirname')
s = toolboxdir tbxdirname
```

Description

`toolboxdir('tbxdirname')` returns a string that is the absolute path to the specified toolbox, `tbxdirname`, where `tbxdirname` is the directory name for the toolbox.

`s = toolboxdir('tbxdirname')` returns the absolute path to the specified toolbox to the output argument, `s`.

`s = toolboxdir tbxdirname` is the command form of the syntax.

Remarks

`toolboxdir` is particularly useful for MATLAB Compiler. The base directory of all toolboxes installed with MATLAB is

```
matlabroot/toolbox/tbxdirname
```

However, in deployed mode, the base directories of the toolboxes are different. `toolboxdir` returns the correct root directory, whether running from MATLAB or from an application deployed with MATLAB Compiler.

Example

To obtain the pathname for Control System Toolbox, run

```
s = toolboxdir('control')
```

MATLAB returns

```
s = \\myhome\r2007b\matlab\toolbox\control
```

See Also

`matlabroot`

`ctfroot` in MATLAB Compiler

trace

Purpose

Sum of diagonal elements

Syntax

```
b = trace(A)
```

Description

`b = trace(A)` is the sum of the diagonal elements of the matrix `A`.

Algorithm

`trace` is a single-statement M-file.

```
t = sum(diag(A));
```

See Also

`det`, `eig`

Purpose

Transpose `timeseries` object

Syntax

```
ts1 = transpose(ts)
```

Description

`ts1 = transpose(ts)` returns a new `timeseries` object `ts1` with `IsTimeFirst` value set to the opposite of what it is for `ts`. For example, if `ts` has the first data dimension aligned with the time vector, `ts1` has the last data dimension aligned with the time vector.

Remarks

The `transpose` function that is overloaded for the `timeseries` objects does not transpose the data. Instead, this function changes whether the first or the last dimension of the data is aligned with the time vector.

Note To transpose the data, you must transpose the `Data` property of the time series. For example, you can use the syntax `transpose(ts.Data)` or `(ts.Data).'`. `Data` must be a 2-D array.

Consider a time series with 10 samples with the property `IsTimeFirst = True`. When you transpose this time series, the data size is changed from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

The following table summarizes how MATLAB displays the size for time-series data (up to three dimensions) before and after transposing.

Data Size Before and After Transposing

Size of Original Data	Size of Transposed Data
N-by-1	1-by-1-by-N

Data Size Before and After Transposing (Continued)

Size of Original Data	Size of Transposed Data
N-by-M	M-by-1-by-N
N-by-M-by-L	M-by-L-by-N

Examples

Suppose that a `timeseries` object `ts` has `ts.Data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is set to `true`, which means that the first dimension of the data is aligned with the time vector. `transpose(ts)` modifies the `timeseries` object such that the last dimension of the data is now aligned with the time vector. This permutes the data such that the size of `ts.Data` becomes 3-by-2-by-10.

See Also

`ctranspose (timeseries)`, `tsprops`

Purpose

Trapezoidal numerical integration

Syntax

```
Z = trapz(Y)
Z = trapz(X,Y)
Z = trapz(...,dim)
```

Description

`Z = trapz(Y)` computes an approximation of the integral of `Y` via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply `Z` by the spacing increment. Input `Y` can be complex.

If `Y` is a vector, `trapz(Y)` is the integral of `Y`.

If `Y` is a matrix,`trapz(Y)` is a row vector with the integral over each column.

If `Y` is a multidimensional array, `trapz(Y)` works across the first nonsingleton dimension.

`Z = trapz(X,Y)` computes the integral of `Y` with respect to `X` using trapezoidal integration. Inputs `X` and `Y` can be complex.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `trapz(X,Y)` operates across this dimension.

`Z = trapz(...,dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X`, if given, must be the same as `size(Y,dim)`.

Examples

Example 1

The exact value of $\int_0^{\pi} \sin(x)dx$ is 2.

To approximate this numerically on a uniformly spaced grid, use

```
X = 0:pi/100:pi;
Y = sin(X);
```

Then both

```
Z = trapz(X,Y)
```

and

```
Z = pi/100*trapz(Y)
```

produce

```
Z =
    1.9998
```

Example 2

A nonuniformly spaced example is generated by

```
X = sort(rand(1,101)*pi);
Y = sin(X);
Z = trapz(X,Y);
```

The result is not as accurate as the uniformly spaced grid. One random sample produced

```
Z =
    1.9984
```

Example 3

This example uses two complex inputs:

```
z = exp(1i*pi*(0:100)/100);

trapz(z, 1./z)
ans =
   0.0000 + 3.1411i
```

See Also

`cumsum`, `cumtrapz`

Purpose

Lay out tree or forest

Syntax

```
[x,y] = treelayout(parent,post)
[x,y,h,s] = treelayout(parent,post)
```

Description

`[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with `0` for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

See Also

`etree`, `treeplot`, `etreeplot`, `symbfact`

treeplot

Purpose

Plot picture of tree

Syntax

```
treeplot(p)
treeplot(p,nodeSpec,edgeSpec)
```

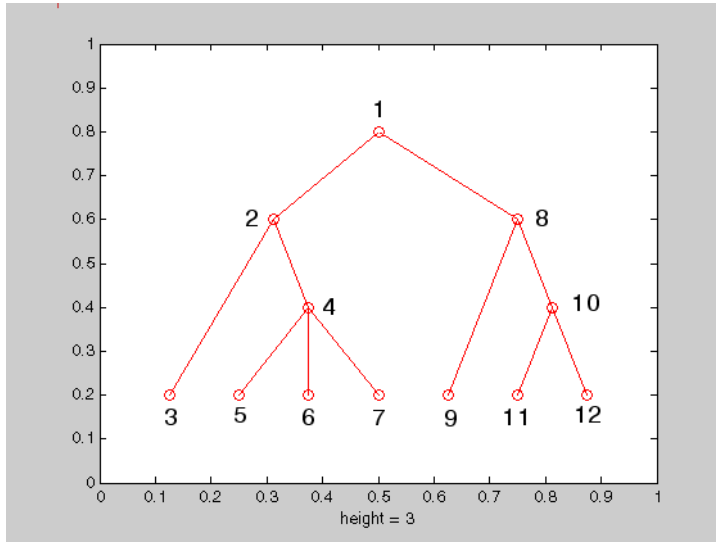
Description

`treeplot(p)` plots a picture of a tree given a vector of parent pointers, with `p(i)` = 0 for a root.

`treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

Examples

To plot a tree with 12 nodes, call `treeplot` with a 12-element input vector. The index of each element in the vector is shown adjacent to each node in the figure below. (These indices are shown only for the point of illustrating the example; they are not part of the `treeplot` output.)



To generate this plot, set the value of each element in the `nodes` vector to the index of its parent, (setting the parent of the root node to zero).

The node marked 1 in the figure is represented by `nodes(1)` in the input vector, and because this is the root node which has a parent of zero, you set its value to zero:

```
nodes(1) = 0;      % Root node
```

`nodes(2)` and `nodes(8)` are children of `nodes(1)`, so set these elements of the input vector to 1:

```
nodes(2) = 1;      nodes(8) = 1;
```

`nodes(5:7)` are children of `nodes(4)`, so set these elements to 4:

```
nodes(5) = 4;      nodes(6) = 4;      nodes(7) = 4;
```

Continue in this manner until each element of the vector identifies its parent. For the plot shown above, the `nodes` vector now looks like this:

```
nodes = [0 1 2 2 4 4 4 1 8 8 10 10];
```

Now call `treeplot` to generate the plot:

```
treeplot(nodes)
```

See Also

`etree`, `etreeplot`, `treelayout`

Purpose

Lower triangular part of matrix

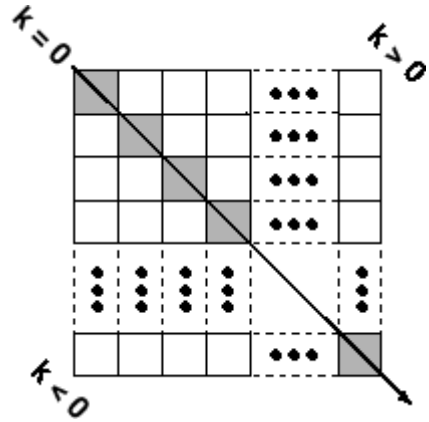
Syntax

```
L = tril(X)
L = tril(X,k)
```

Description

`L = tril(X)` returns the lower triangular part of `X`.

`L = tril(X,k)` returns the elements on and below the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
tril(ones(4,4),-1)

ans =

     0     0     0     0
     1     0     0     0
     1     1     0     0
     1     1     1     0
```

See Also

`diag`, `triu`

Purpose

Triangular mesh plot

Syntax

```
trimesh(Tri,X,Y,Z)
trimesh(Tri,X,Y,Z,C)
trimesh(...'PropertyName',PropertyValue...)
h = trimesh(...)
```

Description

`trimesh(Tri,X,Y,Z)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a mesh. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices.

`trimesh(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`trimesh(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trimesh(...)` returns a handle to a patch graphics object.

Example

Create vertex vectors and a face matrix, then create a triangular mesh plot.

```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x-3,6*x-3);
tri = delaunay(x,y);
trimesh(tri,x,y,z)
```

See Also

`patch`, `tetramesh`, `triplot`, `trisurf`, `delaunay`

"Creating Surfaces and Meshes" on page 1-97 for related functions

triplequad

Purpose

Numerically evaluate triple integral

Syntax

```
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)
```

Description

`triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)` evaluates the triple integral `fun(x,y,z)` over the three dimensional rectangular region `xmin <= x <= xmax`, `ymin <= y <= ymax`, `zmin <= z <= zmax`. `fun` is a function handle. See "Function Handles" in the MATLAB Programming documentation for more information.`fun(x,y,z)` must accept a vector `x` and scalars `y` and `z`, and return a vector of values of the integrand.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples

Pass M-file function handle `@integrnd` to `triplequad`:P

```
Q = triplequad(@integrnd,0,pi,0,1,-1,1);
```

where the M-file `integrnd.m` is

```
function f = integrnd(x,y,z)
f = y*sin(x)+z*cos(x);
```

Pass anonymous function handle `F` to `triplequad`:

```
F = @(x,y,z)y*sin(x)+z*cos(x);
```

```
Q = triplequad(F,0,pi,0,1,-1,1);
```

This example integrates `y*sin(x)+z*cos(x)` over the region `0 <= x <= pi`, `0 <= y <= 1`, `-1 <= z <= 1`. Note that the integrand can be evaluated with a vector `x` and scalars `y` and `z`.

See Also

`dblquad`, `quad`, `quadgk`, `quadl`, `function handle` (`@`), "Anonymous Functions"

triplot

Purpose

2-D triangular plot

Syntax

```
triplot(TRI,x,y)
triplot(TRI,x,y,color)
h = triplot(...)
triplot(...,'param','value','param','value'...)
```

Description

`triplot(TRI,x,y)` displays the triangles defined in the `m`-by-3 matrix `TRI`. A row of `TRI` contains indices into the vectors `x` and `y` that define a single triangle. The default line color is blue.

`triplot(TRI,x,y,color)` uses the string `color` as the line color. `color` can also be a line specification. See `ColorSpec` for a list of valid color strings. See `LineSpec` for information about line specifications.

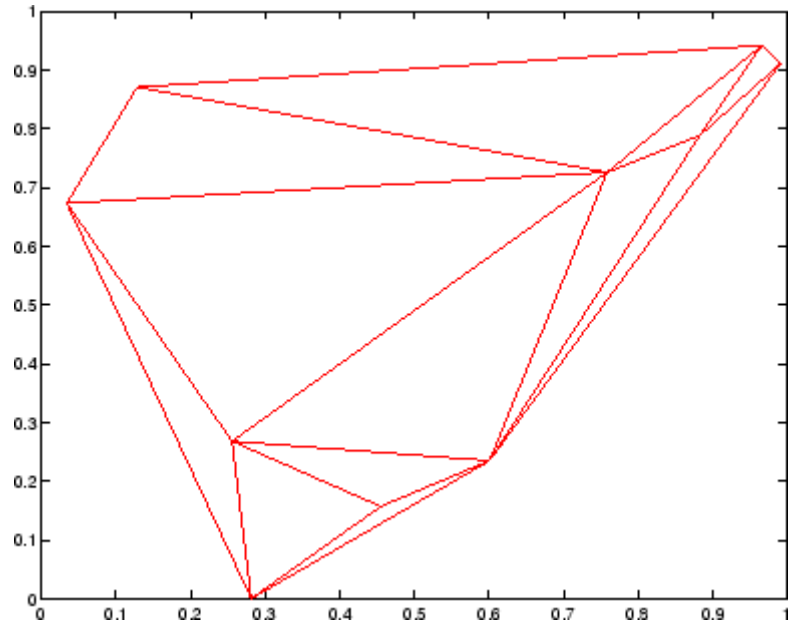
`h = triplot(...)` returns a vector of handles to the displayed triangles.

`triplot(...,'param','value','param','value'...)` allows additional line property name/property value pairs to be used when creating the plot. See `Line Properties` for information about the available properties.

Examples

This code plots the Delaunay triangulation for 10 randomly generated points.

```
rand('state',7);
x = rand(1,10);
y = rand(1,10);
TRI = delaunay(x,y);
triplot(TRI,x,y,'red')
```



See Also

ColorSpec, delaunay, line, Line Properties, LineSpec, plot, trimesh, trisurf

trisurf

Purpose

Triangular surface plot

Syntax

```
trisurf(Tri,X,Y,Z)
trisurf(Tri,X,Y,Z,C)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)
```

Description

`trisurf(Tri,X,Y,Z)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a surface. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices.

`trisurf(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`trisurf(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trisurf(...)` returns a patch handle.

Example

Create vertex vectors and a face matrix, then create a triangular surface plot.

```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x-3,6*x-3);
tri = delaunay(x,y);
trisurf(tri,x,y,z)
```

See Also

`patch`, `surf`, `tetramesh`, `trimesh`, `triplot`, `delaunay`

"Creating Surfaces and Meshes" on page 1-97 for related functions

Purpose

Upper triangular part of matrix

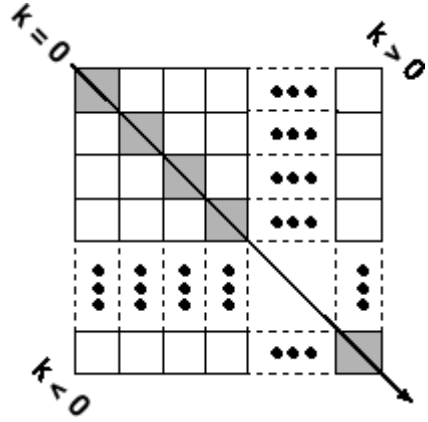
Syntax

```
U = triu(X)
U = triu(X,k)
```

Description

`U = triu(X)` returns the upper triangular part of `X`.

`U = triu(X,k)` returns the element on and above the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
triu(ones(4,4),-1)

ans =

     1     1     1     1
     1     1     1     1
     0     1     1     1
     0     0     1     1
```

See Also

`diag`, `tril`

true

Purpose

Logical 1 (true)

Syntax

```
true
true(n)
true(m, n)
true(m, n, p, ...)
true(size(A))
```

Description

`true` is shorthand for `logical 1`.

`true(n)` is an `n`-by-`n` matrix of logical ones.

`true(m, n)` or `true([m, n])` is an `m`-by-`n` matrix of logical ones.

`true(m, n, p, ...)` or `true([m n p ...])` is an `m`-by-`n`-by-`p`-by-`...` array of logical ones.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`true(size(A))` is an array of logical ones that is the same size as array `A`.

Remarks

`true(n)` is much faster and more memory efficient than `logical(ones(n))`.

See Also

`false`, `logical`

Purpose

Attempt to execute block of code, and catch errors

Syntax

```
try
```

Description

`try` marks the start of a *try block* in a `try-catch` statement. If MATLAB detects an error while executing code in the `try` block, it immediately jumps to the start of the respective *catch block* and executes the error handling code in that block.

A `try-catch` statement is a programming device that enables you to define how certain errors are to be handled in your program. This bypasses the default MATLAB error-handling mechanism when these errors are detected. The `try-catch` statement consists of two blocks of MATLAB code, a *try block* and a *catch block*, delimited by the keywords `try`, `catch`, and `end`:

```
try
   MATLAB commands     % Try block
catch ME
   MATLAB commands     % Catch block
end
```

Each of these blocks consists of one or more MATLAB commands. The `try` block is just another piece of your program code; the commands in this block execute just like any other part of your program. Any errors MATLAB encounters in the `try` block are dealt with by the respective `catch` block. This is where you write your error-handling code. If the `try` block executes without error, MATLAB skips the `catch` block entirely. If an error occurs while executing the `catch` block, the program terminates unless this error is caught by another `try-catch` block.

Specifying the `try`, `catch`, and `end` commands, as well as the commands that make up the `try` and `catch` blocks, on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
ans =
```

```
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

Examples

The `catch` block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension. This is done using a `try-catch` statement that is nested within the original `try-catch`.

```
function d_in = read_image(filename)
file_format = regexp(filename, '(?<=\.)\w+$', 'match');

try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch ME1
   % Get last segment of the error message identifier.
   idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

   % Did the read fail because the file could not be found?
   if strcmp(idSegLast, 'InvalidFid') && ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch file_format
      case 'jpg'    % Change jpg to jpeg
         filename = regexprep(filename, '(?<=\.)\w+$', 'jpeg');
      case 'jpeg'   % Change jpeg to jpg
         filename = regexprep(filename, '(?<=\.)\w+$', 'jpg');
      case 'tif'    % Change tif to tiff
         filename = regexprep(filename, '(?<=\.)\w+$', 'tiff');
      case 'tiff'   % Change tiff to tif
         filename = regexprep(filename, '(?<=\.)\w+$', 'tif');
      otherwise
         disp(sprintf('File %s not found', filename));
         rethrow(ME1);
```

```
            end

            % Try again, with modifed filenames.
            try
               fid = fopen(filename, 'r');
               d_in = fread(fid);
            catch ME2
               disp(sprintf('Unable to access file %s', filename));
               ME2 = addCause(ME2, ME1);
               rethrow(ME2)
            end
         end
      end
```

See Also

catch, rethrow, end, lasterror, eval, evalin

tscollection

Purpose

Create `tscollection` object

Syntax

```
tsc = tscollection(TimeSeries)
tsc = tscollection(Time)
tsc = tscollection(Time,TimeSeries,'Parameter',Value,...)
```

Description

`tsc = tscollection(TimeSeries)` creates a `tscollection` object `tsc` with one or more `timeseries` objects already in the MATLAB workspace. The argument `TimeSeries` can be a

- Single `timeseries` object
- Cell array of `timeseries` objects

`tsc = tscollection(Time)` creates an empty `tscollection` object with the time vector `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

`tsc = tscollection(Time,TimeSeries,'Parameter',Value,...)` creates a `tscollection` object with optional parameter-value pairs you enter after the `Time` and `TimeSeries` arguments. You can specify the following parameters:

- `Name` — String that specifies the name of this `tscollection` object
- `IsDatenum` — Logical value (`true` or `false`) that when set to `true` specifies that the `Time` values are dates in the format of MATLAB serial dates.

Remarks

Definition: Time Series Collection

A time series collection object is a MATLAB variable that groups several time series with a common time vector. The time series that you include in the collection are called members of this collection.

Properties of Time Series Collection Objects

This table lists the properties of the `tscollection` object. You can specify the `Time`, `TimeSeries`, and `Name` properties as input arguments in the constructor.

Property	Description
`Name`	`tscollection` name as a string. This can differ from the `tscollection` name in the MATLAB workspace.
`Time`	When `TimeInfo.StartDate` is empty, values are measured relative to `0` . When `TimeInfo.StartDate` is defined, values represent date strings measured relative to the `StartDate`. The length of `Time` must be the same as the first or the last dimension of `Data` for each collection .
`TimeInfo`	Contains fields for contextual information about `Time`: • `Units` — Time units with any of the following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, `'nanoseconds'` • `Start` — Start time • `End` — End time (read only) • `Increment` — Interval between subsequent time values. `NaN` when times are not uniformly sampled. • `Length` — Length of the time vector (read only) • `Format` — String defining the date string display format. See `datestr`. • `StartDate` — Date string defining the reference date. See `setabstime (tscollection)`. • `UserData` — Any additional user-defined information

Examples

The following example shows how to create a `tscollection` object.

1. Import the sample data.

```
load count.dat
```

2. Create three `timeseries` objects to store each set of data:

```
count1 = timeseries(count(:,1),1:24,'name', 'ts1');
count2 = timeseries(count(:,2),1:24,'name', 'ts2');
```

3. Create a `tscollection` object named `tsc` and add to it two out of three time series already in the MATLAB workspace, by using the following syntax:

```
tsc = tscollection({count1 count2},'name','tsc')
```

See Also

`addts`, `datestr`, `setabstime (tscollection)`, `timeseries`, `tsprops`

Purpose

Construct event object for `timeseries` object

Syntax

```
e = tsdata.event(Name,Time)
e = tsdata.event(Name,Time,'Datenum')
```

Description

`e = tsdata.event(Name,Time)` creates an event object with the specified Name that occurs at the time Time. Time can either be a real value or a date string.

`e = tsdata.event(Name,Time,'Datenum')` uses `'Datenum'` to indicate that the `Time` value is a serial date number generated by the `datenum` function. The `Time` value is converted to a date string after the event is created.

Remarks

You add events by using the `addevent` method.

Fields of the `tsdata.event` object include the following:

- `EventData` — MATLAB array that stores any user-defined information about the event
- `Name` — String that specifies the name of the event
- `Time` — Time value when this event occurs, specified as a real number
- `Units` — Time units
- `StartDate` — A reference date, specified in MATLAB `datestr` format. `StartDate` is empty when you have a numerical (non-date-string) time vector.

tsearch

Purpose

Search for enclosing Delaunay triangle

Syntax

```
T = tsearch(x,y,TRI,xi,yi)
```

Description

`T = tsearch(x,y,TRI,xi,yi)` returns an index into the rows of `TRI` for each point in `xi`, `yi`. The `tsearch` command returns `NaN` for all points outside the convex hull. Requires a triangulation `TRI` of the points `x`,`y` obtained from `delaunay`.

See Also

`delaunay`, `delaunayn`, `dsearch`, `tsearchn`

Purpose

N-D closest simplex search

Syntax

```
t = tsearchn(X,TES,XI)
[t,P] = tsearchn(X,TES,XI)
```

Description

`t = tsearchn(X,TES,XI)` returns the indices `t` of the enclosing simplex of the Delaunay tessellation `TES` for each point in `XI`. `X` is an `m`-by-`n` matrix, representing `m` points in N-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in N-dimensional space. `tsearchn` returns `NaN` for all points outside the convex hull of `X`. `tsearchn` requires a tessellation `TES` of the points `X` obtained from `delaunayn`.

`[t,P] = tsearchn(X,TES,XI)` also returns the barycentric coordinate `P` of `XI` in the simplex `TES`. `P` is a `p`-by-`n+1` matrix. Each row of `P` is the Barycentric coordinate of the corresponding point in `XI`. It is useful for interpolation.

Algorithm

`tsearchn` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`delaunayn`, `griddatan`, `tsearch`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483.

Purpose

Help on `timeseries` object properties

Syntax

```
help timeseries/tsprops
```

Description

`help timeseries/tsprops` lists the properties of the `timeseries` object and briefly describes each property.

Time Series Object Properties

Property	Description
`Data`	Time-series data, where each data sample corresponds to a specific time. The data can be a scalar, a vector, or a multidimensional array. Either the first or last dimension of the data must be aligned with `Time`. By default, `NaN`s are used to represent missing or unspecified data. Set the `TreatNaNasMissing` property to determine how missing data is treated in calculations.
`DataInfo`	Contains fields for storing contextual information about `Data`: • `Unit` — String that specifies data units • `Interpolation` — A `tsdata.interpolation` object that specifies the interpolation method for this time series. Fields of the `tsdata.interpolation` object include: - `Fhandle` — Function handle to a user-defined interpolation function - `Name` — String that specifies the name of the interpolation method. Predefined methods include `'linear'` and `'zoh'` (zero-order hold). `'linear'` is the default. • `UserData` — Any user-defined information entered as a string

Time Series Object Properties (Continued)

Property	Description
`Events`	An array of `tsdata.event` objects that stores event information for this time series. You add events by using the `addevent` method. Fields of the `tsdata.event` object include the following: • `EventData` — Any user-defined information about the event • `Name` — String that specifies the name of the event • `Time` — Time value when this event occurs, specified as a real number or a date string • `Units` — Time units • `StartDate` — A reference date specified in MATLAB date-string format. `StartDate` is empty when you have a numerical (non-date-string) time vector.

Time Series Object Properties (Continued)

Property	Description
`IsTimeFirst`	Logical value (`true` or `false`) specifies whether the first or last dimension of the `Data` array is aligned with the time vector. You can set this property when the `Data` array is square and it is ambiguous which dimension is aligned with time. By default, the first `Data` dimension that matches the length of the time vector is aligned with the time vector. When you set this property to: • `true` — The first dimension of the data array is aligned with the time vector. For example: `ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',true);` • `false` — The last dimension of the data array is aligned with the time vector. For example: `ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',false);` After a time series is created, this property is read only.
`Name`	Time-series name entered as a string. This name can differ from the name of the time-series variable in the MATLAB workspace.
`Quality`	An integer vector or array containing values `-128` to `127` that specifies the quality in terms of codes defined by `QualityInfo.Code`. When `Quality` is a vector, it must have the same length as the time vector. In this case, each `Quality` value applies to a corresponding data sample. When `Quality` is an array, it must have the same size as the data array. In this case, each `Quality` value applies to the corresponding value of the data array.

Time Series Object Properties (Continued)

Property	Description
`QualityInfo`	Provides a lookup table that converts numerical `Quality` codes to readable descriptions. `QualityInfo` fields include the following: • `Code` — Integer vector containing values -128 to 127 that define the "dictionary" of quality codes, which you can assign to each `Data` value by using the `Quality` property • `Description` — Cell vector of strings, where each element provides a readable description of the associated quality `Code` • `UserData` — Stores any additional user-defined information Lengths of `Code` and `Description` must match.
`Time`	Array of time values. When `TimeInfo.StartDate` is empty, the numerical `Time` values are measured relative to `0` in specified units. When `TimeInfo.StartDate` is defined, the time values are date strings measured relative to the `StartDate` in specified units. The length of `Time` must be the same as either the first or the last dimension of `Data`.

Time Series Object Properties (Continued)

Property	Description
`TimeInfo`	Uses the following fields for storing contextual information about `Time`: • `Units` — Time units can have any of following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, or `'nanoseconds'` • `Start` — Start time • `End` — End time (read only) • `Increment` — Interval between two subsequent time values • `Length` — Length of the time vector (read only) • `Format` — String defining the date string display format. See the MATLAB `datestr` function reference page for more information. • `StartDate` — Date string defining the reference date. See the MATLAB `setabstime (timeseries)` function reference page for more information. • `UserData` — Stores any additional user-defined information
`TreatNaNasMissing`	Logical value that specifies how to treat `NaN` values in `Data`: • `true` — (Default) Treat all `NaN` values as missing data except during statistical calculations. • `false` — Include `NaN` values in statistical calculations, in which case `NaN` values are propagated to the result.

See Also

datestr, get (timeseries), set (timeseries), setabstime (timeseries)

tstool

Purpose

Open Time Series Tools GUI

Syntax

```
tstool
tstool(ts)
tstool(tsc)
tstool(sldata)
tstool(ModelDataLogs,'replace')
```

Description

`tstool` starts the Time Series Tools GUI without loading any data.

`tstool(ts)` starts the Time Series Tools GUI and loads the time-series object `ts` from the MATLAB workspace.

`tstool(tsc)` starts the Time Series Tools GUI and loads the time-series collection object `tsc` from the MATLAB workspace.

`tstool(sldata)` starts the Time Series Tools GUI and loads the logged-signal data `sldata` from a Simulink model. If a Simulink logged signal `Name` property contains a `/`, the entire logged signal, including all levels of the signal hierarchy, is not imported into Time Series Tools.

`tstool(ModelDataLogs,'replace')` replaces the logged-signal data object `ModelDataLogs` in the Time Series Tools GUI with an updated logged signal after you rerun the Simulink model. Use this command to update the `ModelDataLogs` object in the Time Series Tools GUI if you change the model or the logged-signal data settings.

See Also

`timeseries`, `tscollection`

Purpose

Display contents of file

Syntax

```
type('filename')
type filename
```

Description

`type('filename')` displays the contents of the specified file in the MATLAB Command Window. Use the full path for `filename`, or use a MATLAB relative partial pathname.

If you do not specify a filename extension and there is no `filename` file without an extension, the `type` function adds the `.m` extension by default. The `type` function checks the directories specified in the MATLAB search path, which makes it convenient for listing the contents of M-files on the screen. Use `type` with `more on` to see the listing one screen at a time.

`type filename` is the command form of the syntax.

Examples

`type('foo.bar')` lists the contents of the file `foo.bar`.

`type foo` lists the contents of the file `foo.` If `foo` does not exist, `type foo` lists the contents of the file `foo.m`.

See Also

`cd`, `dbtype`, `delete`, `dir`, `more`, `partialpath`, `path`, `what`, `who`

Purpose

Convert data types without changing underlying data

Syntax

```
Y = typecast(X, type)
```

Description

`Y = typecast(X, type)` converts a numeric value in `X` to the data type specified by `type`. Input `X` must be a full, noncomplex, numeric scalar or vector. The `type` input is a string set to one of the following: `'uint8'`, `'int8'`, `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, or `'double'`.

`typecast` is different from the MATLAB `cast` function in that it does not alter the input data. `typecast` always returns the same number of bytes in the output `Y` as were in the input `X`. For example, casting the 16-bit integer 1000 to `uint8` with `typecast` returns the full 16 bits in two 8-bit segments (3 and 232) thus keeping its original value (3*256 + 232 = 1000). The `cast` function, on the other hand, truncates the input value to 255.

The output of `typecast` can be formatted differently depending on what system you use it on. Some computer systems store data starting with its most significant byte (an ordering called *big-endian*), while others start with the least significant byte (called *little-endian*).

Note MATLAB issues an error if `X` contains fewer values than are needed to make an output value.

Examples

Example 1

This example converts between data types of the same size:

```
typecast(uint8(255), 'int8')
ans =
   -1

typecast(int16(-1), 'uint16')
ans =
```

```
    65535
```

Example 2

Set X to a 1-by-3 vector of 32-bit integers, then cast it to an 8-bit integer type:

```
X = uint32([1 255 256])
X =
           1         255         256
```

Running this on a little-endian system produces the following results. Each 32-bit value is divided up into four 8-bit segments:

```
Y = typecast(X, 'uint8')
Y =
   1   0   0   0  255   0   0   0   0   1   0   0
```

The third element of `X`, 256, exceeds the 8 bits that it is being converted to in `Y`(9) and thus overflows to `Y`(10):

```
Y(9:12)
ans =
    0    1    0    0
```

Note that `length(Y)` is equal to `4.*length(X)`. Also note the difference between the output of `typecast` versus that of `cast`:

```
Z = cast(X, 'uint8')
Z =
    1  255  255
```

Example 3

This example casts a smaller data type (`uint8`) into a larger one (`uint16`). Displaying the numbers in hexadecimal format makes it easier to see just how the data is being rearranged:

```
format hex
X = uint8([44 55 66 77])
X =
```

```
    2c   37   42   4d
```

The first `typecast` is done on a big-endian system. The four 8-bit segments of the input data are combined to produce two 16-bit segments:

```
Y = typecast(X, 'uint16')
Y =
   2c37   424d
```

The second is done on a little-endian system. Note the difference in byte ordering:

```
Y = typecast(X, 'uint16')
Y =
   372c   4d42
```

You can format the little-endian output into big-endian (and vice versa) using the `swapbytes` function:

```
Y = swapbytes(typecast(X, 'uint16'))
Y =
   2c37   424d
```

Example 4

This example attempts to make a 32-bit value from a vector of three 8-bit values. MATLAB issues an error because there are an insufficient number of bytes in the input:

```
format hex

typecast(uint8([120 86 52]), 'uint32')
??? Too few input values to make output type.

Error in ==> typecast at 29
out = typecastc(in, datatype);
```

Repeat the example, but with a vector of four 8-bit values, and it returns the expected answer:

```
typecast(uint8([120 86 52 18]), 'uint32')
ans =
   12345678
```

See Also

cast, class, swapbytes

uibuttongroup

Purpose

Create container object to exclusively manage radio buttons and toggle buttons

Syntax

```
uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,
   ...)
handle = uibuttongroup(...)
```

Description

A uibuttongroup groups components and manages exclusive selection behavior for radio buttons and toggle buttons that it contains. It can also contain other user interface controls, axes, uipanels, and uibuttongroups. It cannot contain ActiveX controls.

`uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,...)` creates a visible container component in the current figure window. This component manages exclusive selection behavior for uicontrols of style `radiobutton` and `togglebutton`.

Use the `Parent` property to specify the parent as a figure, uipanel, or uibuttongroup. If you do not specify a parent, `uibuttongroup` adds the button group to the current figure. If no figure exists, one is created.

See the Uibuttongroup Properties reference page for more information.

A `uibuttongroup` object can have `axes`, `uicontrol`, `uipanel`, and `uibuttongroup` objects as children. However, only uicontrols of style `radiobutton` and `togglebutton` are managed by the component.

For the children of a `uibuttongroup` object, the `Position` property is interpreted relative to the button group. If you move the button group, the children automatically move with it and maintain their positions in the button group.

If you have a button group that contains a set of radio buttons and toggle buttons and you want:

- An immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions. See the

SelectionChangeFcn property and the example on this reference page for more information.

- Another component such as a push button to base its action on the selection, then that component's Callback callback can get the handle of the selected radio button or toggle button from the button group's SelectedObject property.

handle = uibuttongroup(...) creates a uibuttongroup object and returns a handle to it in handle.

After creating a uibuttongroup, you can set and query its property values using set and get. Run get(handle) to see a list of properties and their current values. Run set(handle) to see a list of object properties you can set and their legal values.

Examples

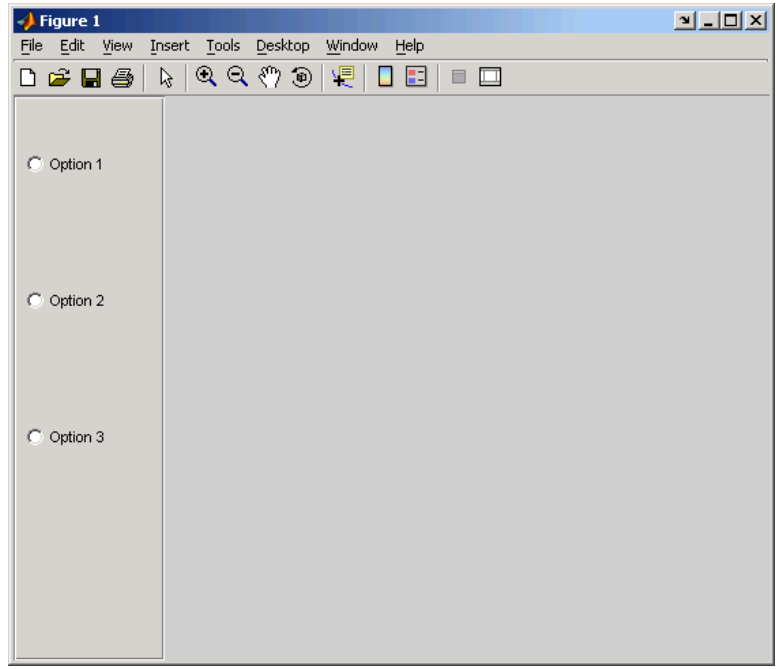
This example creates a uibuttongroup with three radiobuttons. It manages the radiobuttons with the SelectionChangeFcn callback, selcbk.

When you select a new radio button, selcbk displays the uibuttongroup handle on one line, the EventName, OldValue, and NewValue fields of the event data structure on a second line, and the value of the SelectedObject property on a third line.

```
% Create the button group.
h = uibuttongroup('visible','off','Position',[0 0 .2 1]);
% Create three radio buttons in the button group.
u0 = uicontrol('Style','Radio','String','Option 1',...
    'pos',[10 350 100 30],'parent',h,'HandleVisibility','off');
u1 = uicontrol('Style','Radio','String','Option 2',...
    'pos',[10 250 100 30],'parent',h,'HandleVisibility','off');
u2 = uicontrol('Style','Radio','String','Option 3',...
    'pos',[10 150 100 30],'parent',h,'HandleVisibility','off');
% Initialize some button group properties.
set(h,'SelectionChangeFcn',@selcbk);
set(h,'SelectedObject',[]);  % No selection
set(h,'Visible','on');
```

For the `SelectionChangeFcn` callback, `selcbk`, the source and event data structure arguments are available only if `selcbk` is called using a function handle. See `SelectionChangeFcn` for more information.

```
function selcbk(source,eventdata)
disp(source);
disp([eventdata.EventName,'  ',...
     get(eventdata.OldValue,'String'),'  ', ...
     get(eventdata.NewValue,'String')]);
disp(get(get(source,'SelectedObject'),'String'));
```



If you click `Option 2` with no option selected, the `SelectionChangeFcn` callback, `selcbk`, displays:

```
3.0011
```

```
SelectionChanged    Option 2
Option 2
```

If you then click `Option 1`, the `SelectionChangeFcn` callback, `selcbk`, displays:

```
    3.0011

SelectionChanged  Option 2  Option 1
Option 1
```

See Also

`uicontrol`, `uipanel`

Uibuttongroup Properties

Purpose

Describe button group properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

Uibuttongroup takes its default property values from uipanel. To set a uibuttongroup default property value, set the default for the corresponding `uipanel` property. Note that you can set no default values for the uibuttongroup `SelectedObject` and `SelectionChangeFcn` properties.

For more information about changing the default value of a property see "Setting Default Property Values". For an example, see the `CreateFcn` property.

Uibuttongroup Properties

This section describes all properties useful to `uibuttongroup` objects and lists valid values. Curly braces { } enclose default values.

Property Name	Description
`BackgroundColor`	Color of the button group background
`BorderType`	Type of border around the button group
`BorderWidth`	Width of the button group border in pixels
`BusyAction`	Interruption of other callback routines
`ButtonDownFcn`	Button-press callback routine
`Children`	All children of the button group

Property Name	Description
`Clipping`	Clipping of child axes, panels, and button groups to the button group. Does not affect child user interface controls (`uicontrol`)
`CreateFcn`	Callback routine executed during object creation
`DeleteFcn`	Callback routine executed during object deletion
`FontAngle`	Title font angle
`FontName`	Title font name
`FontSize`	Title font size
`FontUnits`	Title font units
`FontWeight`	Title font weight
`ForegroundColor`	Title font color and color of 2-D border line
`HandleVisibility`	Handle accessibility from command line and GUIs
`HighlightColor`	3-D frame highlight color
`Interruptible`	Callback routine interruption mode
`Parent`	`uibuttongroup` object's parent
`Position`	Button group position relative to parent figure, panel, or button group
`ResizeFcn`	User-specified resize routine
`Selected`	Whether object is selected
`SelectedObject`	Currently selected uicontrol of style `radiobutton` or `togglebutton`
`SelectionChangeFcn`	Callback routine executed when the selected radio button or toggle button changes
`SelectionHighlight`	Object highlighted when selected

Property Name	Description
ShadowColor	3-D frame shadow color
Tag	User-specified object identifier
Title	Title string
TitlePosition	Location of title string in relation to the button group
Type	Object class
UIContextMenu	Associate context menu with the button group
Units	Units used to interpret the position vector
UserData	User-specified data
Visible	Button group visibility **Note** Controls the Visible property of child axes, panels, and button groups. Does not affect child user interface controls (uicontrol).

```
BackgroundColor
     ColorSpec
```

Color of the uibuttongroup background. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the ColorSpec reference page for more information on specifying color.

```
BorderType
     none | {etchedin} | etchedout |
     beveledin | beveledout | line
```

Border of the uibuttongroup area. Used to define the button group area graphically. Etched and beveled borders provide a 3-D look. Use the HighlightColor and ShadowColor properties to specify

the border color of etched and beveled borders. A line border is 2-D. Use the `ForegroundColor` property to specify its color.

`BorderWidth`
integer

Width of the button group border. The width of the button group borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`
string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uibuttongroup. This is useful for implementing actions to interactively modify object properties, such as size and position, when they are clicked on (using the `selectmoveresize` function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children`
vector of handles

Children of the uibuttongroup. A vector containing the handles of all children of the uibuttongroup. Although a uibuttongroup manages only uicontrols of style `radiobutton` and `togglebutton`, its children can be axes, uipanels, uibuttongroups, and other uicontrols. You can use this property to reorder the children.

`Clipping`
`{on} | off`

Clipping mode. By default, MATLAB clips a uibuttongroup's child axes, uipanels, and uibuttongroups to the uibuttongroup rectangle. If you set `Clipping` to `off`, the axis, uipanel, or uibuttongroup is displayed outside the button group rectangle. This property does not affect child uicontrols which, by default, can display outside the button group rectangle.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uibuttongroup` object. MATLAB sets all property values for the uibuttongroup before executing the `CreateFcn` callback so these values are available to

the callback. Within the function, use gcbo to get the handle of the uibuttongroup being created.

Setting this property on an existing uibuttongroup object has no effect.

To define a default CreateFcn callback for all new uibuttongroups you must define the same default for all uipanels. This default applies unless you override it by specifying a different CreateFcn callback when you call uibuttongroup. For example, the code

```
set(0,'DefaultUipanelCreateFcn','set(gcbo,...
    ''FontName'',''arial'',''FontSize'',12)')
```

creates a default CreateFcn callback that runs whenever you create a new panel or button group. It sets the default font name and font size of the uipanel or uibuttongroup title.

To override this default and create a button group whose FontName and FontSize properties are set to different values, call uibuttongroup with code similar to

```
hpt = uibuttongroup(...,'CreateFcn','set(gcbo,...
''FontName'',''times'',''FontSize'',14)')
```

Note To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uibuttongroup call. In the example above, if instead of redefining the CreateFcn property for this uibuttongroup, you had explicitly set FontSize to 14, the default CreateFcn callback would have set FontSize back to the system dependent default.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the `uibuttongroup` object (e.g., when you issue a delete command or clear the figure containing the uibuttongroup). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine. The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`FontAngle`
`{normal} | italic | oblique`

Character slant used in the Title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`
string

Font family used in the Title. The name of the font in which to display the `Title`. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set `FontName` to the string `FixedWidth`. This string value is case insensitive.

```
set(uicontrol_handle,'FontName','FixedWidth')
```

This then uses the value of the root `FixedWidthFontName` property, which can be set to the appropriate value for a locale

from `startup.m` in the end user's environment. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`
integer

Title font size. A number specifying the size of the font in which to display the `Title`, in units determined by the `FontUnits` property. The default size is system dependent.

`FontUnits`
`inches | centimeters | normalized | {points} |pixels`

Title font size units. `Normalized` units interpret `FontSize` as a fraction of the height of the uibuttongroup. When you resize the uibuttongroup, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of characters in the title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
`ColorSpec`

Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

Note Uicontrols of style `radiobutton` and `togglebutton` that are managed by a uibuttongroup should not be accessed outside the button group. Set the `HandleVisibility` of such radio buttons and toggle buttons to `off` or `callback` to prevent inadvertent access.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HighlightColor`
`ColorSpec`

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the `ColorSpec` reference page for more information on specifying color.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the waiting callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described above.

`Parent`
handle

Uibuttongroup parent. The handle of the uibuttongroup's parent figure, uipanel, or uibuttongroup. You can move a uibuttongroup object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`
position rectangle

Size and location of uibuttongroup relative to parent. The rectangle defined by this property specifies the size and location of the button group within the parent figure window, uipanel, or uibuttongroup. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the parent object to the lower-left corner of the `uibuttongroup` object. `width` and `height` are the dimensions of the uibuttongroup rectangle, including the title. All measurements are in units specified by the `Units` property.

`ResizeFcn`
string or function handle

Resize callback routine. MATLAB executes this callback routine whenever a user resizes the uibuttongroup and the figure `Resize` property is set to `on`, or in GUIDE, the **Resize behavior** option is set to `Other`. You can query the uibuttongroup `Position` property to determine its new size and position. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

You can use `ResizeFcn` to maintain a GUI layout that is not directly supported by the MATLAB `Position`/`Units` paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following `ResizeFcn` accomplishes this; it keeps the uicontrol whose `Tag` is `'StatusBar'` 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the `Tag` property to retrieve the uicontrol handle, and the `gcbo` function to retrieve the figure handle. Also note the defensive programming regarding figure `Units`, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards.

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure `Position` from within the `ResizeFcn` callback; however, the `ResizeFcn` is not called again as a result.

Note that the `print` command can cause the `ResizeFcn` to be called if the `PaperPositionMode` property is set to `manual` and you have defined a resize function. If you do not want your resize function called by `print`, set the `PaperPositionMode` to `auto`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`Selected`
`on | off` (read only)

Is object selected? This property indicates whether the button group is selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` function to set this property, allowing users to select the object with the mouse.

`SelectedObject`
scalar handle

Currently selected radio button or toggle button uicontrol in the managed group of components. Use this property to determine the currently selected component or to initialize selection of one of the radio buttons or toggle buttons. By default, `SelectedObject` is set to the first uicontrol radio button or toggle button that is added. Set it to `[]` if you want no selection. Note that `SelectionChangeFcn` does not execute when this property is set by the user.

`SelectionChangeFcn`
string or function handle

Callback routine executed when the selected radio button or toggle button changes. If this routine is called as a function handle, `uibuttongroup` passes it two arguments. The first argument, `source`, is the handle of the uibuttongroup. The second argument, `eventdata`, is an event data structure that contains the fields shown in the following table.

Event Data Structure Field	Description
`EventName`	`'SelectionChanged'`
`OldValue`	Handle of the object selected before this event. `[]` if none was selected.
`NewValue`	Handle of the currently selected object.

If you have a button group that contains a set of radio buttons and/or toggle buttons and you want an immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions.

If you want another component such as a push button to base its action on the selection, then that component's `Callback` callback can get the handle of the selected radio button or toggle button from the button group's `SelectedObject` property.

Note For GUIDE GUIs, `hObject` contains the handle of the selected radio button or toggle button. See "Examples: Programming GUIDE GUI Components" for more information.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`ShadowColor`
`ColorSpec`

3-D frame shadow color. `ShadowColor` is a three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the `ColorSpec` reference page for more information on specifying color.

`Tag`

string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the `Tag` value `'FormatTb'`.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

`Title`

string

Title string. The text displayed in the button group title. You can position the title using the `TitlePosition` property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string in the cell array or padded string matrix is displayed; the rest are ignored. Vertical slash (’|’) characters are not interpreted as line breaks and instead show up in the text displayed in the uibuttongroup title.

Setting a property value to `default`, `remove`, or `factory` produces the effect described in “Defining Default Values”. To set `Title` to one of these words, you must precede the word with the backslash character. For example,

```
hp = uibuttongroup(...,'Title','\Default');
```

`TitlePosition`
`{lefttop} | centertop | righttop | leftbottom | centerbottom | rightbottom`

Location ofthe title. This property determines the location of the title string, in relation to the uibuttongroup.

`Type`
`string` (read-only)

Object class. This property identifies the kind of graphics object. For uibuttongroup objects, `Type` is always the string `'uibuttongroup'`.

`UIContextMenu`
handle

Associate a context menu with a uibuttongroup. Assign this property the handle of a `Uicontextmenu` object. MATLAB displays the context menu whenever you right-click the `uibuttongroup`. Use the `uicontextmenu` function to create the context menu.

`Units`
`inches | centimeters | {normalized} | points | pixels | characters`

Units of measurement. MATLAB uses these units to interpret the `Position` property. For the button group itself, units are measured from the lower-left corner of its parent figure window, panel, or button group. For children of the button group, they are measured from the lower-left corner of the button group.

- `Normalized` units map the lower-left corner of the button group or figure window to (0,0) and the upper-right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).
- `Character` units are characters using the default system font; the width of one character is the width of the letter x, the

height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the `uibuttongroup` object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`
`{on} | off`

Uibuttongroup visibility. By default, a `uibuttongroup` object is visible. When set to `off`, the uibuttongroup is not visible, but still exists and you can query and set its properties.

Note The value of a uibuttongroup's `Visible` property also controls the `Visible` property of child axes, uipanels, and uibuttongroups. This property does not affect the `Visible` property of child uicontrols.

Purpose

Create context menu

Syntax

```
handle = uicontextmenu('PropertyName',PropertyValue,...)
```

Description

`handle = uicontextmenu('PropertyName',PropertyValue,...)` creates a context menu, which is a menu that appears when the user right-clicks on a graphics object. See the Uicontextmenu Properties reference page for more information.

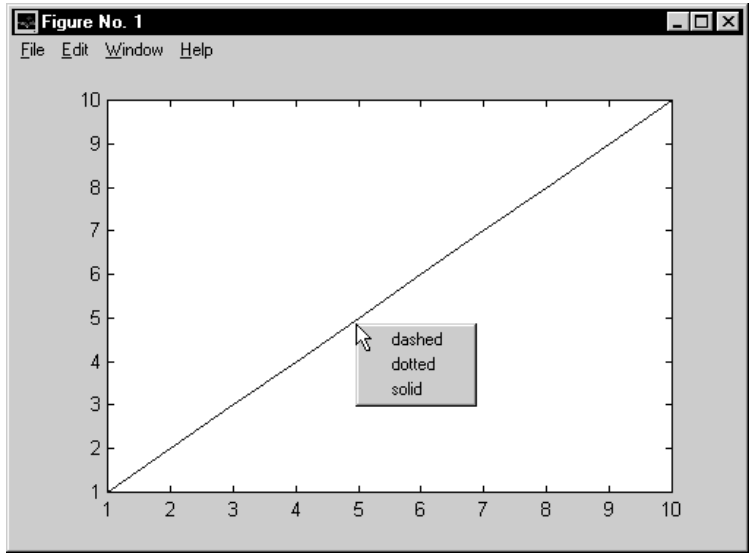
You create context menu items using the `uimenu` function. Menu items appear in the order the `uimenu` statements appear. You associate a context menu with an object using the `UIContextMenu` property for the object and specifying the context menu's handle as the property value.

Example

These statements define a context menu associated with a line. When the user right clicks or presses **Alt+click** anywhere on the line, the menu appears. Menu items enable the user to change the line style.

```
% Define the context menu
cmenu = uicontextmenu;
% Define the line and associate it with the context menu
hline = plot(1:10, 'UIContextMenu', cmenu);
% Define callbacks for context menu items
cb1 = ['set(hline, ''LineStyle'', ''--'')'];
cb2 = ['set(hline, ''LineStyle'', '':'')'];
cb3 = ['set(hline, ''LineStyle'', ''-'')'];
% Define the context menu items
item1 = uimenu(cmenu, 'Label', 'dashed', 'Callback', cb1);
item2 = uimenu(cmenu, 'Label', 'dotted', 'Callback', cb2);
item3 = uimenu(cmenu, 'Label', 'solid', 'Callback', cb3);
```

When the user right clicks or presses **Alt+click** on the line, the context menu appears, as shown in this figure:



See Also

uibuttongroup, uicontrol, uimenu, uipanel

Purpose

Describe context menu properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

For more information about changing the default value of a property see "Setting Default Property Values". For an example, see the `CreateFcn` property.

Uicontext-menu Properties

This section lists all properties useful to `uicontextmenu` objects along with valid values and descriptions of their use. Curly braces {} enclose default values.

Property	Purpose
`BusyAction`	Callback routine interruption
`Callback`	Control action
`Children`	The uimenus defined for the uicontextmenu
`CreateFcn`	Callback routine executed during object creation
`DeleteFcn`	Callback routine executed during object deletion
`HandleVisibility`	Whether handle is accessible from command line and GUIs
`Interruptible`	Callback routine interruption mode
`Parent`	Uicontextmenu object's parent

Property	Purpose
Position	Location of uicontextmenu when Visible is set to on
Tag	User-specified object identifier
Type	Class of graphics object
UserData	User-specified data
Visible	Uicontextmenu visibility

BusyAction

cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. See the Interruptible property for information about controlling a callback's interruptibility.

`Callback`
string

Control action. A routine that executes whenever you right-click an object for which a context menu is defined. The routine executes immediately before the context menu is posted. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children`
matrix

The uimenu items defined for the uicontextmenu.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uicontextmenu` object. MATLAB sets all property values for the uicontextmenu before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uicontextmenu being created.

Setting this property on an existing uicontextmenu object has no effect.

You can define a default `CreateFcn` callback for all new uicontextmenus. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uicontextmenu`. For example, the code

```
set(0,'DefaultUicontextmenuCreateFcn','set(gcbo,...
    ''Visible'',''on'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new context menu. It sets the default `Visible` property of a context menu.

To override this default and create a context menu whose `Visible` property is set to a different value, call `uicontextmenu` with code similar to

```
hpt = uicontextmenu(...,'CreateFcn','set(gcbo,...
''Visible'',''off'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uicontextmenu` call. In the example above, if instead of redefining the `CreateFcn` property for this uicontextmenu, you had explicitly set `Visible` to `off`, the default `CreateFcn` callback would have set `Visible` back to the default, i.e., `on`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Delete uicontextmenu callback routine. A callback routine that executes when you delete the uicontextmenu object (e.g., when you issue a `delete` command or clear the figure containing the uicontextmenu). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility`

settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Parent`
handle

Uicontextmenu's parent. The handle of the uicontextmenu's parent object. You can move a uicontextmenu object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`
vector

Uicontextmenu's position. A two-element vector that defines the location of a context menu posted by setting the `Visible` property value to `on`. Specify `Position` as

```
[x y]
```

where vector elements represent the horizontal and vertical distances in pixels from the bottom left corner of the figure window, panel, or button group to the top left corner of the context menu.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This

is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`

string

Class of graphics object. For uicontextmenu objects, `Type` is always the string `'uicontextmenu'`.

`UserData`

matrix

User-specified data. Any data you want to associate with the uicontextmenu object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`

`on | {off}`

Uicontextmenu visibility. The `Visible` property can be used in two ways:

- Its value indicates whether the context menu is currently posted. While the context menu is posted, the property value is `on`; when the context menu is not posted, its value is `off`.
- Its value can be set to `on` to force the posting of the context menu. Similarly, setting the value to `off` forces the context menu to be removed. When used in this way, the `Position` property determines the location of the posted context menu.

Purpose

Create user interface control object

Syntax

```
handle = uicontrol('PropertyName',PropertyValue,...)
handle = uicontrol(parent,'PropertyName',PropertyValue,...)
handle = uicontrol
uicontrol(uich)
```

Description

`uicontrol` creates a uicontrol graphics objects (user interface controls), which you use to implement graphical user interfaces.

`handle = uicontrol('PropertyName',PropertyValue,...)` creates a uicontrol and assigns the specified properties and values to it. It assigns the default values to any properties you do not specify. The default uicontrol style is a pushbutton. The default parent is the current figure. See the Uicontrol Properties reference page for more information.

`handle = uicontrol(parent,'PropertyName',PropertyValue,...)` creates a uicontrol in the object specified by the handle, `parent`. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence. `parent` can be the handle of a figure, uipanel, or uibuttongroup.

`handle = uicontrol` creates a pushbutton in the current figure. The `uicontrol` function assigns all properties their default values.

`uicontrol(uich)` gives focus to the uicontrol specified by the handle, `uich`.

When selected, most uicontrol objects perform a predefined action. MATLAB supports numerous styles of uicontrols, each suited for a different purpose:

- Check boxes
- Editable text fields
- Frames
- List boxes

- Pop-up menus
- Push buttons
- Radio buttons
- Sliders
- Static text labels
- Toggle buttons

For information on using these uicontrols within GUIDE, the MATLAB GUI development environment, see Examples: Programming GUI Components in the MATLAB Creating GUIs documentation

Specifying the Uicontrol Style

To create a specific type of uicontrol, set the `Style` property as one of the following strings:

- `'checkbox'` – Check boxes generate an action when selected. These devices are useful when providing the user with a number of independent choices. To activate a check box, click the mouse button on the object. The state of the device is indicated on the display.
- `'edit'` – Editable text fields enable users to enter or modify text values. Use editable text when you want text as input. If `Max-Min>1`, then multiple lines are allowed. For multi-line edit boxes, a vertical scrollbar enables scrolling, as do the arrow keys.
- `'frame'` – Frames are rectangles that provide a visual enclosure for regions of a figure window. Frames can make a user interface easier to understand by grouping related controls. Frames have no callback routines associated with them. Only other uicontrols can appear within frames.

  Frames are opaque, not transparent, so the order in which you define uicontrols is important in determining whether uicontrols within a frame are covered by the frame or are visible. *Stacking order* determines the order objects are drawn: objects defined first are drawn first; objects defined later are drawn over existing objects. If

you use a frame to enclose objects, you must define the frame before you define the objects.

Note Most frames in existing GUIs can now be replaced with panels (`uipanel`) or button groups (`uibuttongroup`). GUIDE continues to support frames in those GUIs that contain them, but the frame component does not appear in the GUIDE Layout Editor component palette.

- `'listbox'` – List boxes display a list of items and enable users to select one or more items. The `Min` and `Max` properties control the selection mode:

  If `Max-Min>1`, then multiple selection is allowed.

  If `Max-Min<=1`, then only single selection is allowed.

  The `Value` property indicates selected entries and contains the indices into the list of strings; a vector value indicates multiple selections. MATLAB evaluates the list box's callback routine after any mouse button up event that changes the `Value` property. Therefore, you may need to add a "Done" button to delay action caused by multiple clicks on list items. List boxes differentiate between single and double clicks and set the figure `SelectionType` property to `normal` or `open` accordingly before evaluating the list box's `Callback` property.

- `'popupmenu'` – Pop-up menus (also known as drop-down menus or combo boxes) open to display a list of choices when pressed. When not open, a pop-up menu indicates the current choice. Pop-up menus are useful when you want to provide users with a number of mutually exclusive choices, but do not want to take up the amount of space that a series of radio buttons requires.

- `'pushbutton'` – Push buttons generate an action when pressed. To activate a push button, click the mouse button on the push button.

- `'radiobutton'` – Radio buttons are similar to check boxes, but are intended to be mutually exclusive within a group of related radio

buttons (i.e., only one is in a pressed state at any given time). To activate a radio button, click the mouse button on the object. The state of the device is indicated on the display. Note that your code can implement mutually exclusive behavior for radio buttons.

- `'slider'` – Sliders accept numeric input within a specific range by enabling the user to move a sliding bar. Users move the bar by pressing the mouse button and dragging the pointer over the bar, or by clicking in the trough or on an arrow. The location of the bar indicates a numeric value, which is selected by releasing the mouse button. You can set the minimum, maximum, and current values of the slider.
- `'text'` – Static text boxes display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively and there is no way to invoke the callback routine associated with it.
- `'togglebutton'` – Toggle buttons are controls that execute callbacks when clicked on and indicate their state, either on or off. Toggle buttons are useful for building toolbars.

Remarks

- The `uicontrol` function accepts property name/property value pairs, structures, and cell arrays as input arguments and optionally returns the handle of the created object. You can also set and query property values after creating the object using the `set` and `get` functions.
- A `uicontrol` object is a child of a figure, uipanel, or uibuttongroup and therefore does not require an axes to exist when placed in a figure window, uipanel, or uibuttongroup.
- When MATLAB is paused and a uicontrol has focus, pressing a keyboard key does not cause MATLAB to resume. Click anywhere outside a uicontrol and then press any key. See the `pause` function for more information.

Examples

Example 1

The following statement creates a push button that clears the current axes when pressed.

```
h = uicontrol('Style', 'pushbutton', 'String', 'Clear',...
    'Position', [20 150 100 70], 'Callback', 'cla');
```

This statement gives focus to the pushbutton.

```
uicontrol(h)
```

Example 2

You can create a uicontrol object that changes figure colormaps by specifying a pop-up menu and supplying an M-file name as the object's `Callback`:

```
hpop = uicontrol('Style', 'popup',...
       'String', 'hsv|hot|cool|gray',...
       'Position', [20 320 100 50],...
       'Callback', 'setmap');
```

The above call to `uicontrol` defines four individual choices in the menu: `hsv`, `hot`, `cool`, and `gray`. You specify these choices with the `String` property, separating the choices with the "`|`" character.

The `Callback`, in this case `setmap`, is the name of an M-file that defines a more complicated set of instructions than a single MATLAB command. `setmap` contains these statements:

```
val = get(hpop,'Value');
if val == 1
    colormap(hsv)
elseif val == 2
    colormap(hot)
elseif val == 3
    colormap(cool)
elseif val == 4
    colormap(gray)
```

```
    end
```

The `Value` property contains a number that indicates the selected choice. The choices are numbered sequentially from one to four. The `setmap` M-file can get and then test the contents of the `Value` property to determine what action to take.

See Also

`textwrap`, `uibuttongroup`, `uimenu`, `uipanel`

Purpose

Describe user interface control (`uicontrol`) properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see “Setting Default Property Values”. You can also set default uicontrol properties on the root and figure levels:

```
set(0,'DefaultUicontrolProperty',PropertyValue...)
set(gcf,'DefaultUicontrolProperty',PropertyValue...)
```

where *`Property`* is the name of the uicontrol property whose default value you want to set and `PropertyValue` is the value you are specifying as the default. Use `set` and `get` to access uicontrol properties.

For information on using these uicontrols within GUIDE, the MATLAB GUI development environment, see Programming GUI Components in the MATLAB Creating GUIs documentation.

Uicontrol Properties

This section lists all properties useful to `uicontrol` objects along with valid values and descriptions of their use. Curly braces `{}` enclose default values.

Property	Purpose
`BackgroundColor`	Object background color
`BusyAction`	Callback routine interruption
`ButtonDownFcn`	Button-press callback routine
`Callback`	Control action

Property	Purpose
CData	Truecolor image displayed on the control
Children	Uicontrol objects have no children
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
Enable	Enable or disable the uicontrol
FontAngle	Character slant
FontName	Font family
FontSize	Font size
FontUnits	Font size units
FontWeight	Weight of text characters
ForegroundColor	Color of text
HandleVisibility	Whether handle is accessible from command line and GUIs
HitTest	Whether selectable by mouse click
HorizontalAlignment	Alignment of label string
Interruptible	Callback routine interruption mode
KeyPressFcn	Key press callback routine
ListboxTop	Index of top-most string displayed in list box
Max	Maximum value (depends on uicontrol object)
Min	Minimum value (depends on uicontrol object)
Parent	Uicontrol object's parent
Position	Size and location of uicontrol object

Property	Purpose
Selected	Whether object is selected
SelectionHighlight	Object highlighted when selected
SliderStep	Slider step size
String	Uicontrol object label, also list box and pop-up menu items
Style	Type of uicontrol object
Tag	User-specified object identifier
TooltipString	Content of object's tooltip
Type	Class of graphics object
UIContextMenu	Uicontextmenu object associated with the uicontrol
Units	Units to interpret position vector
UserData	User-specified data
Value	Current value of uicontrol object
Visible	Uicontrol visibility

`BackgroundColor`
`ColorSpec`

Object background color. The color used to fill the uicontrol rectangle. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default color is determined by system settings. See `ColorSpec` for more information on specifying color.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for

which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`ButtonDownFcn`

string or function handle (GUIDE sets this property)

Button-press callback routine. A callback routine that can execute when you press a mouse button while the pointer is on or near a uicontrol. Specifically:

- If the uicontrol's `Enable` property is set to `on`, the `ButtonDownFcn` callback executes when you click the right or left mouse button in a 5-pixel border around the uicontrol or when you click the right mouse button on the control itself.
- If the uicontrol's `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the right or left mouse button in the 5-pixel border or on the control itself.

This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using selectmoveresize, for example).

Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

To add a `ButtonDownFcn` callback in GUIDE, select **View Callbacks** from the Layout Editor **View** menu, then select `ButtonDownFcn`. GUIDE sets this property to the appropriate string and adds the callback to the M-file the next time you save the GUI. Alternatively, you can set this property to the string `%automatic`. The next time you save the GUI, GUIDE sets this property to the appropriate string and adds the callback to the M-file.

Use the `Callback` property to specify the callback routine that executes when you activate the enabled uicontrol (e.g., click a push button).

`Callback`

string or function handle (GUIDE sets this property)

Control action. A routine that executes whenever you activate the uicontrol object (e.g., when you click on a push button or move a slider). Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

For examples of `Callback` callbacks for each style of component:

- For GUIDE GUIs, see “Examples: Programming GUIDE GUI Components”.
- For programmatically created GUIs, see “Examples: Programming GUI Components”.

Callback routines defined for static text do not execute because no action is associated with these objects.

To execute the callback routine for an edit text control, type in the desired text and then do one of the following:

- Click another component, the menu bar, or the background of the GUI.
- For a single line editable text box, press **Enter**.
- For a multiline editable text box, press **Ctl+Enter**.

`CData`
matrix

Truecolor image displayed on control. A three-dimensional matrix of RGB values that defines a truecolor image displayed on a control, which must be a **push button** or **toggle button**. Each value must be between `0.0` and `1.0`. Setting `CData` on a **radio button** or **checkbox** will replace the default `CData` on these controls. The control will continue to work as expected, but its state is not reflected by its appearance when clicked.

For **push buttons** and **toggle buttons**, `CData` overlaps the `String`. In the case of **radio buttons** and **checkboxes**, `CData` takes precedence over `String` and, depending on its size, it can displace the text.

Setting `CData` to `[]` restores the default `CData` for **radio buttons** and **checkboxes**.

`Children`
matrix

The empty matrix; `uicontrol` objects have no children.

`Clipping`
`{on} | off`

This property has no effect on `uicontrol` objects.

`CreateFcn`

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uicontrol` object. MATLAB sets all property values for the uicontrol before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uicontrol being created.

Setting this property on an existing uicontrol object has no effect.

You can define a default `CreateFcn` callback for all new uicontrols. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uicontrol`. For example, the code

```
set(0,'DefaultUicontrolCreateFcn','set(gcbo,...
    ''BackgroundColor'',''white'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new uicontrol. It sets the default background color of all new uicontrols.

To override this default and create a uicontrol whose `BackgroundColor` is set to a different value, call `uicontrol` with code similar to

```
hpt = uicontrol(...,'CreateFcn','set(gcbo,...
''BackgroundColor'',''blue'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uicontrol` call. In the example above, if instead of redefining the `CreateFcn` property for this uicontrol, you had explicitly set `BackgroundColor` to `blue`, the default `CreateFcn` callback would have set `BackgroundColor` back to the default, i.e., `white`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Delete uicontrol callback routine. A callback routine that executes when you delete the uicontrol object (e.g., when you issue a `delete` command or clear the figure containing the uicontrol). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root CallbackObject property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`Enable`

`{on} | inactive | off`

Enable or disable the uicontrol. This property controls how uicontrols respond to mouse button clicks, including which callback routines execute.

- `on` – The uicontrol is operational (the default).
- `inactive` – The uicontrol is not operational, but looks the same as when `Enable` is `on`.
- `off` – The uicontrol is not operational and its image (set by the `Cdata` property) is grayed out.

When you left-click on a uicontrol whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Sets the figure's `SelectionType` property.
2. Executes the uicontrol's `ClickedCallback` routine.
3. Does not set the figure's `CurrentPoint` property and does not execute either the control's `ButtonDownFcn` or the figure's `WindowButtonDownFcn` callback.

When you left-click on a uicontrol whose `Enable` property is `off`, or when you right-click a uicontrol whose `Enable` property has any value, MATLAB performs these actions in this order:

4. Sets the figure's `SelectionType` property.
5. Sets the figure's `CurrentPoint` property.
6. Executes the figure's `WindowButtonDownFcn` callback.

`Extent`

position rectangle (read only)

Size of uicontrol character string. A four-element vector that defines the size and position of the character string used to label the uicontrol. It has the form:

```
[0,0,width,height]
```

The first two elements are always zero. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

Since the `Extent` property is defined in the same units as the uicontrol itself, you can use this property to determine proper sizing for the uicontrol with regard to its label. Do this by

- Defining the `String` property and selecting the font using the relevant properties.
- Getting the value of the `Extent` property.
- Defining the `width` and `height` of the `Position` property to be somewhat larger than the `width` and `height` of the `Extent`.

For multiline strings, the `Extent` rectangle encompasses all the lines of text. For single line strings, the Extent is returned as a single line, even if the string wraps when displayed on the control.

`FontAngle`

`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`

string

Font family. The name of the font in which to display the String. To display and print properly, this must be a font that your system supports. The default font is system dependent.

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set `FontName` to the string `FixedWidth` (this string value is case sensitive):

```
set(uicontrol_handle, 'FontName', 'FixedWidth')
```

This parameter value eliminates the need to hard code the name of a fixed-width font, which may not display text properly on

systems that do not use ASCII character encoding (such as in Japan). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` and rely on the root `FixedWidthFontName` property to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`
size in `FontUnits`

Font size. A number specifying the size of the font in which to display the String, in units determined by the `FontUnits` property. The default point size is system dependent.

`FontUnits`
`{points} | normalized | inches | centimeters | pixels`

Font size units. This property determines the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the uicontrol. When you resize the uicontrol, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = $^{1}/_{72}$ inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
`ColorSpec`

Color of text. This property determines the color of the text defined for the `String` property (the uicontrol label). Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default text color is black. See `ColorSpec` for more information on specifying color.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

Note Radio buttons and toggle buttons that are managed by a uibuttongroup should not be accessed outside the button group. Set the `HandleVisibility` of such radio buttons and toggle buttons to `off` to prevent inadvertent access.

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uicontrol objects.

`HorizontalAlignment`
`left | {center} | right`

Horizontal alignment of label string. This property determines the justification of the text defined for the `String` property (the uicontrol label):

- `left` — Text is left justified with respect to the uicontrol.
- `center` — Text is centered with respect to the uicontrol.
- `right` — Text is right justified with respect to the uicontrol.

On Microsoft Windows systems, this property affects only `edit` and `text` uicontrols.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for

which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`KeyPressFcn`
string or function handle

Key press callback function. A callback routine invoked by a key press when the callback's uicontrol object has focus. Focus is denoted by a border or a dotted border, respectively, in UNIX and Microsoft Windows. If no uicontrol has focus, the figure's key press callback function, if any, is invoked. `KeyPressFcn` can be a function handle, the name of an M-file, or any legal MATLAB expression.

If the specified value is the name of an M-file, the callback routine can query the figure's `CurrentCharacter` property to determine what particular key was pressed and thereby limit the callback execution to specific keys.

If the specified value is a function handle, the callback routine can retrieve information about the key that was pressed from its event data structure argument.

Event Data Structure Field	Description	Examples:			
		a	=	Shift	Shift/a
`Character`	Character interpretation of the key that was pressed.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	Current modifier, such as `'control'`, or an empty cell array if there is no modifier	`{1x0 cell}`	`{1x0 cell}`	`{'shift'}`	`{'shift'}`
`Key`	Name of the key that was pressed.	`'a'`	`'equal'`	`'shift'`	`'a'`

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`ListboxTop`
scalar

Index of top-most string displayed in list box. This property applies only to the `listbox` style of uicontrol. It specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. `ListboxTop` is an index into the array of strings defined by the `String` property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

`Max`

scalar

Maximum value. This property specifies the largest value allowed for the `Value` property. Different styles of uicontrols interpret `Max` differently:

- Check boxes – `Max` is the setting of the `Value` property while the check box is selected.
- Editable text – If `Max - Min` > 1, then editable text boxes accept multiline input. If `Max - Min` <= 1, then editable text boxes accept only single line input.
- List boxes – If `Max - Min` > 1, then list boxes allow multiple item selection. If `Max - Min` <= 1, then list boxes do not allow multiple item selection.
- Radio buttons – `Max` is the setting of the `Value` property when the radio button is selected.
- Sliders – `Max` is the maximum slider value and must be greater than the `Min` property. The default is 1.
- Toggle buttons – `Max` is the value of the `Value` property when the toggle button is selected. The default is 1.
- Pop-up menus, push buttons, and static text do not use the `Max` property.

`Min`

scalar

Minimum value. This property specifies the smallest value allowed for the `Value` property. Different styles of uicontrols interpret `Min` differently:

- Check boxes – `Min` is the setting of the `Value` property while the check box is not selected.
- Editable text – If `Max - Min` > 1, then editable text boxes accept multiline input. If `Max - Min` <= 1, then editable text boxes accept only single line input.
- List boxes – If `Max - Min` > 1, then list boxes allow multiple item selection. If `Max - Min` <= 1, then list boxes allow only single item selection.
- Radio buttons – `Min` is the setting of the `Value` property when the radio button is not selected.
- Sliders – `Min` is the minimum slider value and must be less than `Max`. The default is 0.
- Toggle buttons – `Min` is the value of the `Value` property when the toggle button is not selected. The default is 0.
- Pop-up menus, push buttons, and static text do not use the `Min` property.

`Parent`
handle

Uicontrol parent. The handle of the uicontrol's parent object. You can move a uicontrol object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`
position rectangle

Size and location of uicontrol. The rectangle defined by this property specifies the size and location of the control within

the parent figure window, uipanel, or uibuttongroup. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the parent object to the lower-left corner of the uicontrol object. `width` and `height` are the dimensions of the uicontrol rectangle. All measurements are in units specified by the `Units` property.

On Microsoft Windows systems, the height of pop-up menus is automatically determined by the size of the font. The value you specify for the `height` of the `Position` property has no effect.

The `width` and `height` values determine the orientation of sliders. If `width` is greater than `height`, then the slider is oriented horizontally, If `height` is greater than `width`, then the slider is oriented vertically.

Note The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored. The height element of the position vector is not changed.

On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

`Selected`

`on | {off}` (read only)

Is object selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Object highlight when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`SliderStep`
`[min_step max_step]`

Slider step size. This property controls the amount the slider `Value` changes when you click the mouse on the arrow button (`min_step`) or on the slider trough (`max_step`). Specify `SliderStep` as a two-element vector; each value must be in the range `[0, 1]`. The actual step size is a function of the specified `SliderStep` and the total slider range (`Max - Min`). The default, `[0.01 0.10]`, provides a 1 percent change for clicks on the arrow button and a 10 percent change for clicks in the trough.

For example, if you create the following slider,

```
uicontrol('Style','slider','Min',1,'Max',7,...
          'Value',2,'SliderStep',[0.1 0.6])
```

clicking on the arrow button moves the indicator by,

```
0.1*(7-1)
ans =
     0.6000
```

and clicking in the trough moves the indicator by,

```
0.6*(7-1)
ans =
     3.6000
```

Note that if the specified step size moves the slider to a value outside the range, the indicator moves only to the `Max` or `Min` value.

See also the `Max`, `Min`, and `Value` properties.

`String`
string

Uicontrol label, *list box items*, *pop-up menu choices*.

For check boxes, editable text, push buttons, radio buttons, static text, and toggle buttons, the text displayed on the object. For list boxes and pop-up menus, the set of entries or items displayed in the object.

Note If you specify a numerical value for `String`, MATLAB converts it to `char` but the result may not be what you expect. If you have numerical data, you should first convert it to a string, e.g., using `num2str`, before assigning it to the `String` property.

For uicontrol objects that display only one line of text (check box, push button, radio button, toggle button), if the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('|') characters are not interpreted as line breaks and instead show up in the text displayed in the uicontrol.

For multiple line editable text or static text controls, line breaks occur between each row of the string matrix, and each cell of a cell array of strings. Vertical slash ('|') characters and `\n` characters are not interpreted as line breaks, and instead show up in the text displayed in the uicontrol.

For multiple items on a list box or pop-up menu, you can specify the items in any of the formats shown in the following table.

String Property Format	Example
Cell array of strings	`{'one' 'two' 'three'}`
Padded string matrix	`['one ';'two ';'three']`
String vector separated by vertical slash (|) characters	`['one|two|three']`

If you specify a component width that is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis. Use the `Value` property to set the index of the initial item selected.

For **check boxes**, **push buttons**, **radio buttons**, **toggle buttons**, and the selected item in **popup menus**, when the specified text is clipped because it is too long for the uicontrol, an ellipsis (...) is appended to the text in the active GUI to indicate that it has been clipped.

For **push buttons** and **toggle buttons**, `CData` overlaps the `String`. In the case of **radio buttons** and **checkboxes**, `CData` takes precedence over `String` and, depending on its size, can displace the text.

For **editable text**, the `String` property value is set to the string entered by the user.

Reserved Words There are three reserved words: `default`, `remove`, `factory` (case sensitive). If you want to use one of these reserved words in the `String` property, you must precede it with a backslash (`'\'`) character. For example,

```
h = uicontrol('Style','edit','String','\default');
```

`Style`

`{pushbutton} | togglebutton | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu`

Style of uicontrol object to create. The `Style` property specifies the kind of uicontrol to create. See the `uicontrol` Description section for information on each type.

`Tag`

string (GUIDE sets this property)

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`TooltipString`

string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uicontrol. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

`Type`

string (read only)

Class of graphics object. For uicontrol objects, `Type` is always the string `'uicontrol'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. Assign this property the handle of a `uicontextmenu` object. MATLAB displays the context menu whenever you right-click over the uicontrol. Use the `uicontextmenu` function to create the context menu.

`Units`
`{pixels} | normalized | inches | centimeters | points | characters` (GUIDE default: `normalized`)

Units of measurement. MATLAB uses these units to interpret the `Extent` and `Position` properties. All units are measured from the lower-left corner of the parent object.

- `Normalized` units map the lower-left corner of the parent object to (0,0) and the upper-right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).
- `Character` units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the uicontrol object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Value`

scalar or vector

Current value of uicontrol. The uicontrol style determines the possible values this property can have:

- Check boxes set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- List boxes set `Value` to a vector of indices corresponding to the selected list entries, where 1 corresponds to the first item in the list.
- Pop-up menus set `Value` to the index of the item selected, where 1 corresponds to the first item in the menu. The Examples section shows how to use the `Value` property to determine which item has been selected.
- Radio buttons set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- Sliders set `Value` to the number indicated by the slider bar.
- Toggle buttons set `Value` to `Max` when they are down (selected) and `Min` when up (not selected).
- Editable text, push buttons, and static text do not set this property.

Set the `Value` property either interactively with the mouse or through a call to the `set` function. The display reflects changes made to `Value`.

`Visible`

`{on} | off`

Uicontrol visibility. By default, all uicontrols are visible. When set to `off`, the uicontrol is not visible, but still exists and you can query and set its properties.

Note Setting `Visible` to `off` for uicontrols that are not displayed initially in the GUI, can result in faster startup time for the GUI.

---

uigetdir

Purpose

Open standard dialog box for selecting a directory

Syntax

```
uigetdir
directory_name = uigetdir
directory_name = uigetdir(start_path)
directory_name = uigetdir(start_path,dialog_title)
```

Description

`uigetdir` displays a modal dialog box enabling the user to browse through the directory structure and select a directory or type the name of a directory. If the directory exists, `uigetdir` returns the selected path when the user clicks **OK**. For Windows platforms, `uigetdir` opens a dialog box in the base directory (the Windows desktop) with the current directory selected. See "Remarks" on page 2-3491 for information about UNIX and Mac platforms.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

`directory_name = uigetdir` returns the path to the selected directory when the user clicks **OK**. If the user clicks **Cancel** or closes the dialog window, `directory_name` is set to`0`.

`directory_name = uigetdir(start_path)` opens a dialog box with the directory specified by `start_path` selected. If `start_path` is a valid directory path, the dialog box opens in the specified directory.

If `start_path` is an empty string (`''`), the dialog box opens in the current directory. If `start_path` is not a valid directory path, the dialog box opens in the base directory. For Windows, this is the Windows desktop. See "Remarks" on page 2-3491 for information about UNIX and Mac platforms.

`directory_name = uigetdir(start_path,dialog_title)` opens a dialog box with the specified title. On Windows platforms, the

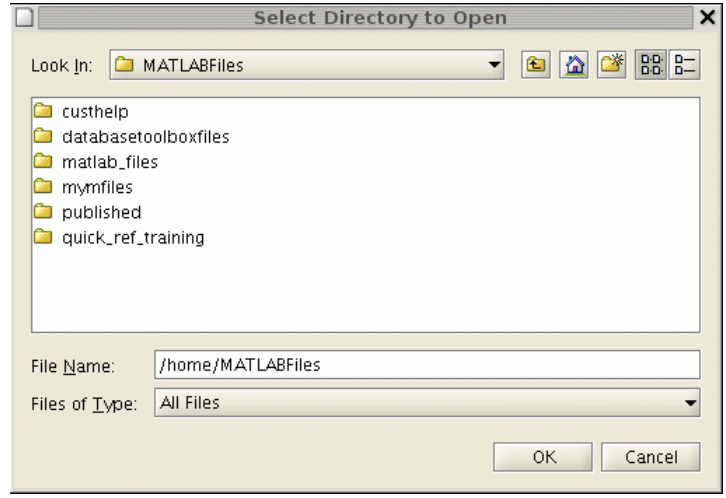
string replaces the default caption inside the dialog box for specifying instructions to the user. The default `dialog_title` is`Select Directory to Open`. See “Remarks” on page 2-3491 for information about UNIX and Mac platforms.

Note On Windows platforms, users can click the **New Folder** button to add a new directory to the directory structure displayed. Users can also drag and drop existing directories.

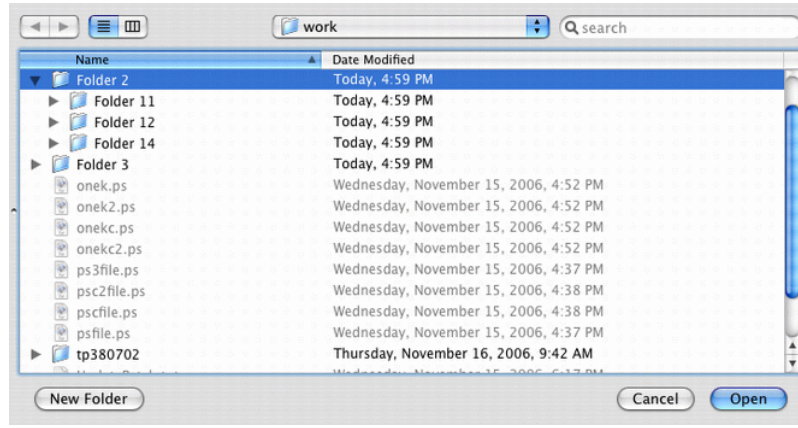
Remarks

For Windows platforms, the dialog box is similar to those shown in the “Examples” on page 2-3492 below.

For UNIX platforms, `uigetdir` opens a dialog box in the base directory (the directory from which MATLAB is started) with the current directory selected. The `dialog_title` string replaces the default title of the dialog box. The dialog box is similar to the one shown in the following figure.



For Mac platforms,`uigetdir` opens a dialog box in the base directory (the current directory) with the current directory open. The `dialog_title` string, if any, is ignored. The dialog box is similar to the one shown in the following figure.



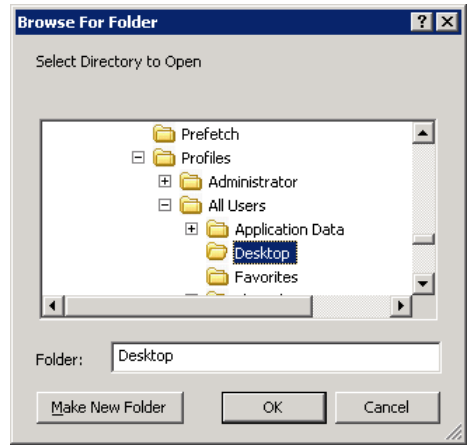
Examples

Example 1

The following statement displays directories on the `C:` drive.

```
dname = uigetdir('C:\');
```

The dialog box is shown in the following figure.



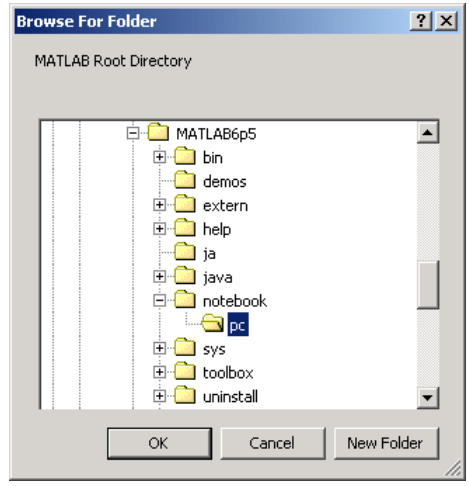
If the user selects the directory `Desktop`, as shown in the figure, and clicks **OK**, `uigetdir` returns

```
dname =
C:\WINNT\Profiles\All Users\Desktop
```

Example 2

The following statement uses the `matlabroot` command to display the MATLAB root directory in the dialog box:

```
uigetdir(matlabroot,'MATLAB Root Directory')
```



If the user selects the directory `MATLAB6.5/notebook/pc`, as shown in the figure, `uigetdir` returns a string like

```
C:\MATLAB6.5\notebook\pc
```

assuming that MATLAB is installed on drive `C:\`.

See Also

`uigetfile`, `uiputfile`

Purpose

Open standard dialog box for retrieving files

Syntax

```
uigetfile
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle,DefaultName)
[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',
   selectmode)
```

Description

`uigetfile` displays a modal dialog box that lists files in the current directory and enables the user to select or type the name of a file to be opened. If the filename is valid and if the file exists, `uigetfile` returns the filename when the user clicks **Open**. Otherwise `uigetfile` displays an appropriate error message from which control returns to the dialog box. The user can then enter another filename or click **Cancel**. If the user clicks **Cancel** or closes the dialog window, `uigetdir` returns `0`.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)` displays only those files with extensions that match `FilterSpec`. `FilterSpec` can be a string or a cell array of strings, and can include the `*` wildcard. For example, `'*.m'` lists all the MATLAB M-files. A `FilterSpec` string can also be a filename. In this case the filename becomes the default filename and the file's extension is used as the default filter. If `FilterSpec` is a string, `uigetfile` appends `'All Files'` to the list of file types.

If `FilterSpec` is a cell array, the first column contains a list of file extensions. The optional second column contains a corresponding list of

descriptions. These descriptions replace standard descriptions in the **Files of type** field. A description cannot be an empty string. “Example 2” on page 2-3499 and “Example 3” on page 2-3500 illustrate use of a cell array as `FilterSpec`.

If `FilterSpec` is not specified, `uigetfile` uses the default list of file types (i.e., all MATLAB files).

After the user clicks **Open** and if the filename exists,`uigetfile` returns the name of the file in `FileName` and its path in `PathName`. If the user clicks **Cancel** or closes the dialog window, `FileName` and `PathName` are set to `0`.

`FilterIndex` is the index of the filter selected in the dialog box. Indexing starts at 1. If the user clicks **Cancel** or closes the dialog window, `FilterIndex` is set to `0`.

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle)` displays a dialog box that has the title `DialogTitle`. To use the default file types and specify a dialog title, enter

```
uigetfile('',DialogTitle)
```

Note For Mac platforms, `DialogTitle` is ignored.

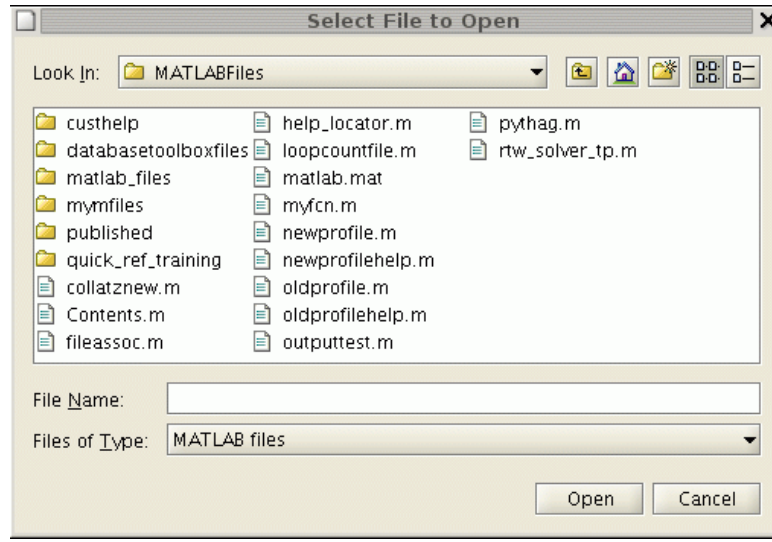
`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle,DefaultName)` displays a dialog box in which the filename specified by `DefaultName` appears in the **File name** field. `DefaultName` can also be a path or a path/filename. In this case, `uigetfile` opens the dialog box in the directory specified by the path. See “Example 6” on page 2-3503 . If the path does not include a filename, it must end with a slash (`/`) or backslash (`\`) separator. For example, `'C:\Work\'`. Note that `uigetfile` recognizes both `'./'` and `'../'` as valid values. If the specified path does not exist, `uigetfile` opens the dialog box in the current directory.

`[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',`*`selectmode`*`)` sets the multiselect mode to specify if multiple file selection is enabled for the `uigetfile` dialog. Valid values for *`selectmode`* are `'on'` and `'off'` (default). If `'MultiSelect'` is `'on'` and the user selects more than one file in the dialog box, then `FileName` is a cell array of strings, each of which represents the name of a selected file. Filenames in the cell array are in the sort order native to your platform. Because multiple selections are always in the same directory, `PathName` is always a string that represents a single directory.

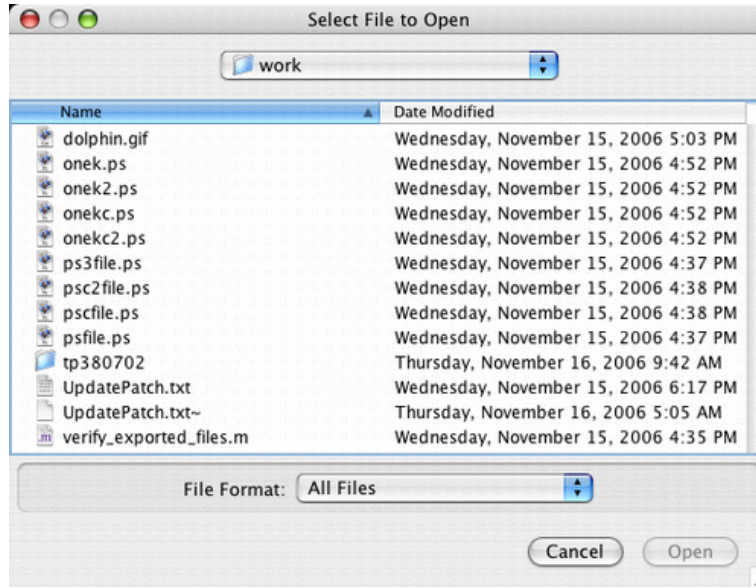
Remarks

For Windows platforms, the dialog box is the Windows dialog box native to your platform. Because of this, it may differ from those shown in “Examples” on page 2-3498 below.

For UNIX platforms, the dialog box is similar to the one shown in the following figure.



For Mac platforms, the dialog box is similar to the one shown in the following figure.



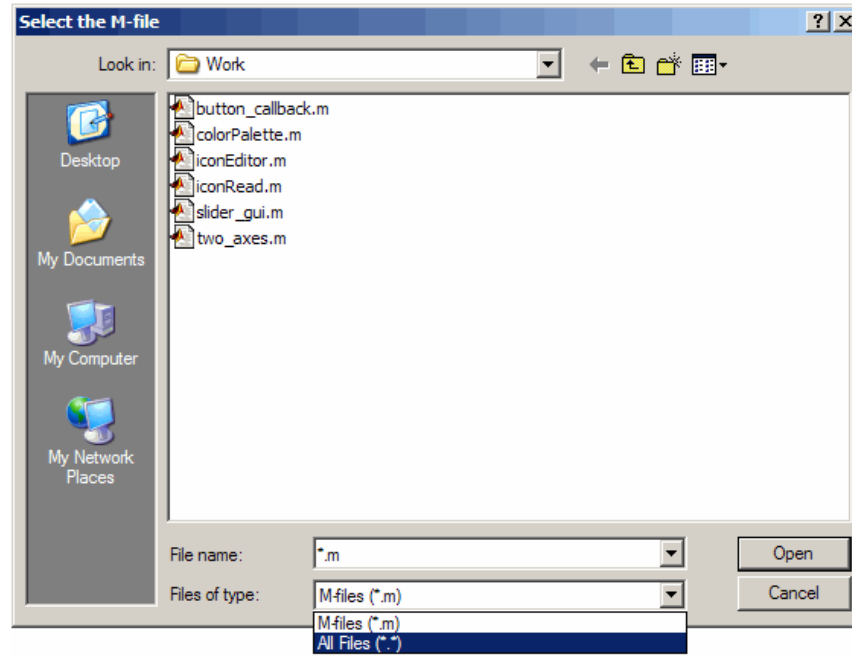
Examples

Example 1

The following statement displays a dialog box that enables the user to retrieve a file. The statement lists all MATLAB M-files within a selected directory. The name and path of the selected file are returned in `FileName` and `PathName`. Note that `uigetfile` appends `All Files(*.*)` to the file types when `FilterSpec` is a string.

```
[FileName,PathName] = uigetfile('*.m','Select the M-file');
```

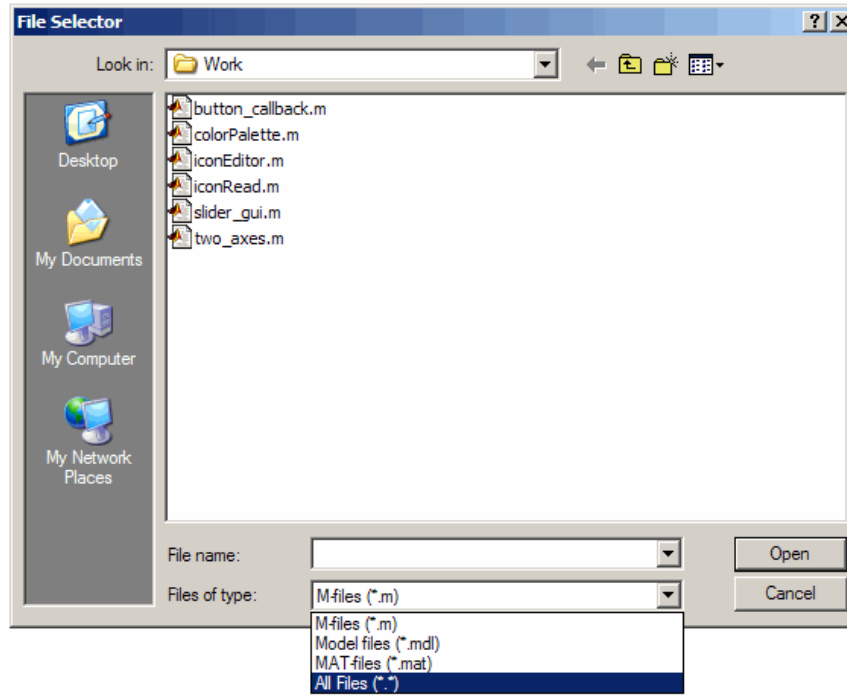
The dialog box is shown in the following figure.



Example 2

To create a list of file types that appears in the **Files of type** list box, separate the file extensions with semicolons, as in the following code. Note that `uigetfile` displays a default description for each known file type, such as "Simulink Models" for `.mdl` files.

```
[filename, pathname] = ...
     uigetfile({'*.m';'*.mdl';'*.mat';'*.*'},'File Selector');
```

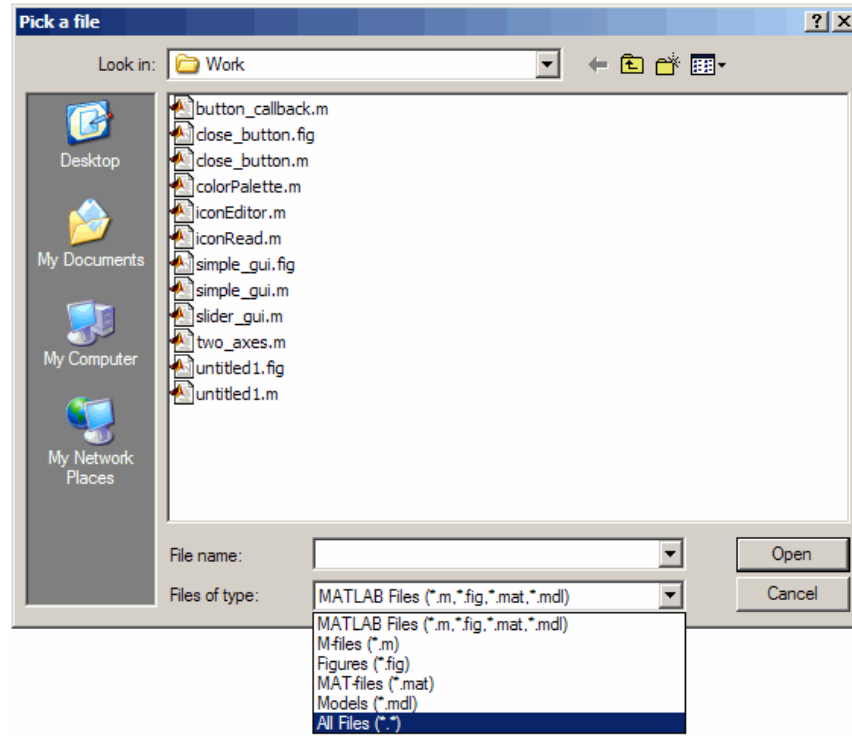


Example 3

If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the `'MATLAB Files'` description.

```
[filename, pathname] = uigetfile( ...
{'*.m;*.fig;*.mat;*.mdl','MATLAB Files (*.m,*.fig,*.mat,*.mdl)';
   '*.m',  'M-files (*.m)'; ...
   '*.fig','Figures (*.fig)'; ...
   '*.mat','MAT-files (*.mat)'; ...
   '*.mdl','Models (*.mdl)'; ...
   '*.*',  'All Files (*.*)'}, ...
   'Pick a file');
```

The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. Note that the first entry of column one contains several extensions, separated by semicolons, all of which are associated with the description `'MATLAB Files (*.m,*.fig,*.mat,*.mdl)'`. The code produces the dialog box shown in the following figure.

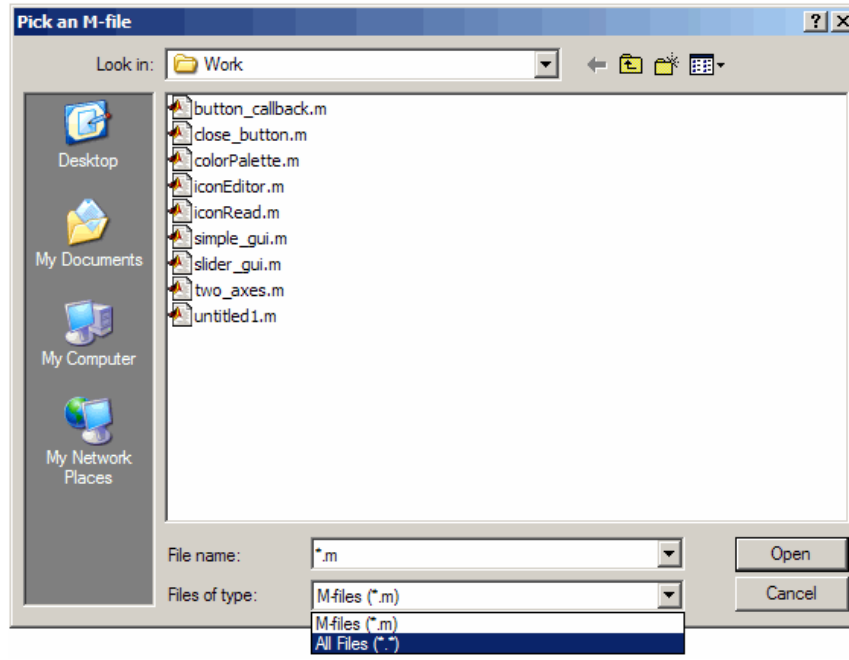


Example 4

The following code checks for the existence of the file and displays a message about the result of the open operation.

```
[filename, pathname] = uigetfile('*.m', 'Pick an M-file');
```

```
if isequal(filename,0)
   disp('User selected Cancel')
else
   disp(['User selected', fullfile(pathname, filename)])
end
```

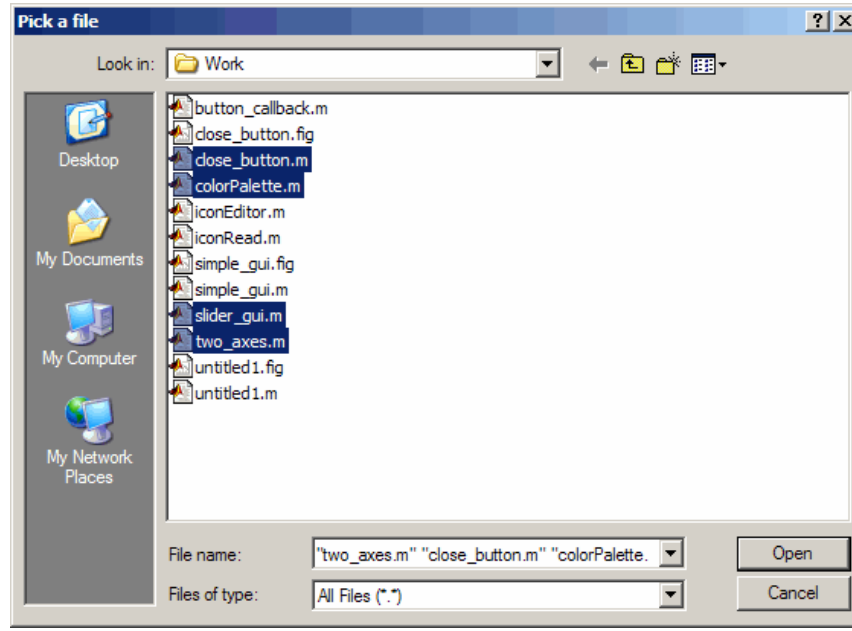


Example 5

This example creates a list of file types and gives them descriptions that are different from the defaults, then enables multiple file selection. The user can select multiple files by holding down the **Shift** or **Ctrl** key and clicking on a file.

```
[filename, pathname, filterindex] = uigetfile( ...
{  '*.mat','MAT-files (*.mat)'; ...
   '*.mdl','Models (*.mdl)'; ...
   '*.*',  'All Files (*.*)'}, ...
```

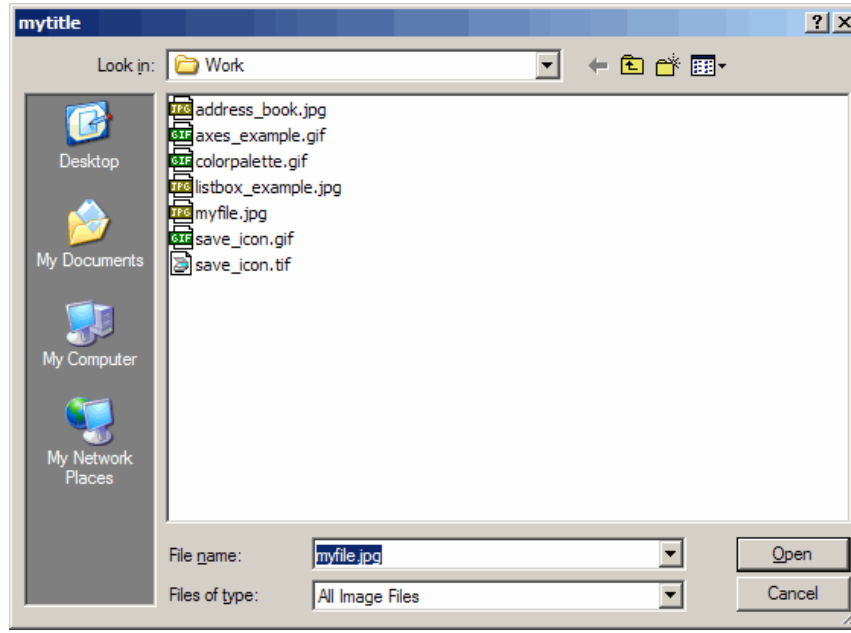
```
'Pick a file', ...
'MultiSelect', 'on');
```



Example 6

This example uses the `DefaultName` argument to specify a start path and a default filename for the dialog box.

```
uigetfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'mytitle',...
          'C:\Work\myfile.jpg')
```



See Also

`uigetdir`, `uiputfile`

Purpose

Open dialog box for retrieving preferences

Syntax

```
value = uigetpref(group,pref,title,question,pref_choices)
[val,dlgshown] = uigetpref(...)
```

Description

`value = uigetpref(group,pref,title,question,pref_choices)` returns one of the strings in `pref_choices`, by doing one of the following:

- Prompting the user with a multiple-choice question dialog box
- Returning a previous answer stored in the preferences database

By default, the dialog box is shown, with each choice on a different pushbutton, and with a checkbox controlling whether the returned value should be stored in preferences and automatically reused in subsequent invocations.

If the user checks the checkbox before choosing one of the push buttons, the push button choice is stored in preferences and returned in `value`. Subsequent calls to `uigetpref` detect that the last choice was stored in preferences, and return that choice immediately without displaying the dialog.

If the user does not check the checkbox before choosing a pushbutton, the selected preference is not stored in preferences. Rather, a special value, `'ask'`, is stored, indicating that subsequent calls to `uigetpref` should display the dialog box.

Note `uigetpref` uses the same preference database as `addpref`, `getpref`, `ispref`, `rmpref`, and `setpref`. However, it registers the preferences it sets in a separate list so that it, and `uisetpref`, can distinguish those preferences that are being managed with `uigetpref`.

For preferences registered with `uigetpref`, you can use `setpref` and `uisetpref` to explicitly change preference values to `'ask'`.

`group` and `pref` define the preference. If the preference does not already exist, `uigetpref` creates it.

`title` defines the string displayed in the dialog box titlebar.

`question` is a descriptive paragraph displayed in the dialog, specified as a string array or cell array of strings. This should contain the question the user is being asked, and should be detailed enough to give the user a clear understanding of their choice and its impact. `uigetpref` inserts line breaks between rows of the string array, between elements of the cell array of strings, or between '|' or newline characters in the string vector.

`pref_choices` is either a string, cell array of strings, or '|'-separated strings specifying the strings to be displayed on the push buttons. Each string element is displayed in a separate push button. The string on the selected pushbutton is returned.

Make `pref_choices` a 2-by-n cell array of strings if the internal preference values are different from the strings displayed on the pushbuttons. The first row contains the preference strings, and the second row contains the related pushbutton strings. Note that the preference values are returned in `value`, not the button labels.

`[val,dlgshown] = uigetpref(...)` returns whether or not the dialog was shown.

Additional arguments can be passed in as parameter-value pairs:

`(...'CheckboxState',state)` sets the initial state of the checkbox, either checked or unchecked. `state` can be either `0` (unchecked) or `1` (checked). By default it is 0.

`(...'CheckboxString',cbstr)` sets the string `cbstr` on the checkbox. By default it is `'Never show this dialog again'`.

`(...'HelpString',hstr)` sets the string `hstr` on the help button. By default the string is empty and there is no help button.

`(...'HelpFcn',hfcn)` sets the callback that is executed when the help button is pressed. By default it is `doc('uigetpref')`. Note that if there is no `'HelpString'` option, a button is not created.

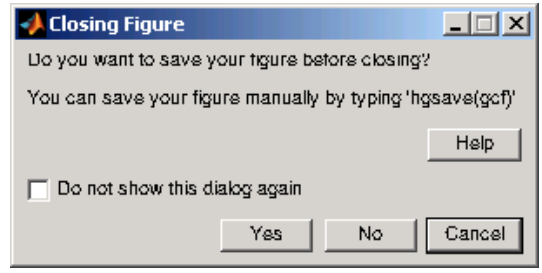
`(...'ExtraOptions',eo)` creates extra buttons which are not mapped to any preference settings. `eo` can be a string or a cell array of strings. By default it is `{}` and no extra buttons are created. If the user chooses one of these buttons, the dialog is closed and the string is returned in `value`.

`(...'DefaultButton',dbstr)` sets the string value `dbstr` that is returned if the dialog is closed. By default, it is the first button. Note that `dbstr` does not have to correspond to a preference or `ExtraOption`.

Note If the preference does not already exist in the preference database, `uigetpref` creates it. Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

This example creates the following preference dialog for the `'savefigurebeforeclosing'` preference in the `'mygraphics'` group.



It uses the cell array `{'always','never';'Yes','No'}` to define the preference values as `'always'` and `'never'`, and their corresponding button labels as `'Yes'` and `'No'`.

```
[selectedButton,dlgShown]=uigetpref('mygraphics',... % Group
       'savefigurebeforeclosing',...           % Preference
       'Closing Figure',...                    % Window title
       {'Do you want to save your figure before closing?'
```

```
 ''
 'You can save your figure manually by typing ''hgsave(gcf)'''},...
{'always','never';'Yes','No'},...       % Values and button strings
'ExtraOptions','Cancel',...             % Additional button
'DefaultButton','Cancel',...            % Default choice
'HelpString','Help',...                 % String for Help button
'HelpFcn','doc(''closereq'');')         % Callback for Help button
```

See Also

addpref, getpref, ispref, rmpref, setpref, uisetpref

Purpose

Open Import Wizard to import data

Syntax

```
uiimport
uiimport(filename)
uiimport('-file')
uiimport('-pastespecial')
S = uiimport(...)
```

Description

`uiimport` starts the Import Wizard in the current directory, presenting options to load data from a file or the clipboard.

`uiimport(filename)` starts the Import Wizard, opening the file specified in `filename`. The Import Wizard displays a preview of the data in the file.

`uiimport('-file')` works as above but presents the file selection dialog first.

`uiimport('-pastespecial')` works as above but presents the clipboard contents first.

`S = uiimport(...)` works as above with resulting variables stored as fields in the struct `S`.

Note For ASCII data, you must verify that the Import Wizard correctly identified the column delimiter.

See Also

`load`, `clipboard`

uimenu

Purpose

Create menus on figure windows

Syntax

```
handle = uimenu('PropertyName',PropertyValue,...)
handle = uimenu(parent,'PropertyName',PropertyValue,...)
```

Description

`uimenu` creates a hierarchy of menus and submenus that are displayed in the figure window's menu bar. You also use `uimenu` to create menu items for context menus.

`handle = uimenu('PropertyName',PropertyValue,...)` creates a menu in the current figure's menu bar using the values of the specified properties and assigns the menu handle to `handle`.

See the Uimenu Properties reference page for more information.

`handle = uimenu(parent,'PropertyName',PropertyValue,...)` creates a submenu of a parent menu or a menu item on a context menu specified by `parent` and assigns the menu handle to `handle`. If `parent` refers to a figure instead of another uimenu object or a uicontextmenu, MATLAB creates a new menu on the referenced figure's menu bar.

Remarks

MATLAB adds the new menu to the existing menu bar. If the figure does not have a menu bar, MATLAB creates one. Each menu choice can itself be a menu that displays its submenu when selected. `uimenu` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

The uimenu `Callback` property defines the action taken when you activate the created menu item.

Uimenus only appear in figures whose `Window Style` is `normal`. If a figure containing uimenu children is changed to `modal`, the uimenu children still exist and are contained in the `Children` list of the figure, but are not displayed until the `WindowStyle` is changed to `normal`.

The value of the figure `MenuBar` property affects the content of the figure menu bar. When `MenuBar` is `figure`, a set of built-in menus precedes any user-created uimenus on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user).

When `MenuBar` is `none`, uimenus are the only items on the menu bar (that is, the built-in menus do not appear).

You can set and query property values after creating the menu using `set` and `get`.

Examples

This example creates a menu labeled **Workspace** whose choices allow users to create a new figure window, save workspace variables, and exit out of MATLAB. In addition, it defines an accelerator key for the Quit option.

```
f = uimenu('Label','Workspace');
    uimenu(f,'Label','New Figure','Callback','figure');
    uimenu(f,'Label','Save','Callback','save');
    uimenu(f,'Label','Quit','Callback','exit',...
           'Separator','on','Accelerator','Q');
```

See Also

`uicontrol`, `uicontextmenu`, `gcbo`, `set`, `get`, `figure`

Uimenu Properties

Purpose

Describe menu properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` commands enable you to set and query the values of properties

You can set default Uimenu properties on the root, figure and menu levels:

```
set(0,'DefaultUimenuPropertyName',PropertyValue...)
set(gcf,'DefaultUimenuPropertyName',PropertyValue...)
set(menu_handle,'DefaultUimenuPropertyName',PropertyValue...)
```

Where *`PropertyName`* is the name of the Uimenu property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of property see "Setting Default Property Values"

Uimenu Properties

This section lists all properties useful to `uimenu` objects along with valid values and instructions for their use. Curly braces { } enclose default values.

Property Name	Property Description
`Accelerator`	Keyboard equivalent
`BusyAction`	Callback routine interruption
`Callback`	Control action
`Checked`	Menu check indicator
`Children`	Handles of submenus

Property Name	Property Description
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
Enable	Enable or disable the uimenu
ForegroundColor	Color of text
HandleVisibility	Whether handle is accessible from command line and GUIs
Interruptible	Callback routine interruption mode
Label	Menu label
Parent	Uimenu object's parent
Position	Relative uimenu position
Separator	Separator line mode
Tag	User-specified object identifier
Type	Class of graphics object
UserData	User-specified data
Visible	Uimenu visibility

`Accelerator`
character

Keyboard equivalent. An alphabetic character specifying the keyboard equivalent for the menu item. This allows users to select a particular menu choice by pressing the specified character in conjunction with another key, instead of selecting the menu item with the mouse. The key sequence is platform specific:

- For Microsoft Windows systems, the sequence is **Ctrl**+`Accelerator`. These keys are reserved for default menu items: c, v, and x.
- For UNIX systems, the sequence is **Ctrl**+`Accelerator`. These keys are reserved for default menu items: o, p, s, and w.

You can define an accelerator only for menu items that do not have children menus. Accelerators work only for menu items that directly execute a callback routine, not items that bring up other menus.

Note that the menu item does not have to be displayed (e.g., a submenu) for the accelerator key to work. However, the window focus must be in the figure when the key sequence is entered.

To remove an accelerator, set `Accelerator` to an empty string, `''`.

`BusyAction`

```
cancel | {queue}
```

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the`Interruptible` property for information about controlling a callback's interruptibility.

`Callback`

string or function handle

Menu action. A callback routine that executes whenever you select the menu. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

A menu with children (submenus) executes its callback routine before displaying the submenus. A menu without children executes its callback routine when you *release* the mouse button (i.e., on the button up event).

`Checked`

`on | {off}`

Menu check indicator. Setting this property to `on` places a check mark next to the corresponding menu item. Setting it to `off` removes the check mark. You can use this feature to create menus that indicate the state of a particular option. For example, suppose you have a menu item called **Show axes** that toggles the visibility of an axes between visible and invisible each time the user selects the menu item. If you want a check to appear next to the menu item when the axes are visible, add the following code to the callback for the **Show axes** menu item:

```
if strcmp(get(gcbo, 'Checked'),'on')
    set(gcbo, 'Checked', 'off');
else
```

```
    set(gcbo, 'Checked', 'on');
end
```

This changes the value of the `Checked` property of the menu item from `on` to `off` or vice versa each time a user selects the menu item.

Note that there is no formal mechanism for indicating that an unchecked menu item will become checked when selected.

Note This property is ignored for top level and parent menus.

`Children`
vector of handles

Handles of submenus. A vector containing the handles of all children of the uimenu object. The children objects of uimenus are other uimenus, which function as submenus. You can use this property to reorder the menus.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uimenu` object. MATLAB sets all property values for the uimenu before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uimenu being created.

Setting this property on an existing uimenu object has no effect.

You can define a default `CreateFcn` callback for all new uimenus. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uimenu`. For example, the code

```
set(0,'DefaultUimenuCreateFcn','set(gcbo,...
    ''Visible'',''on'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new menu. It sets the default `Visible` property of a `uimenu` object.

To override this default and create a menu whose `Visible` property is set to a different value, call `uimenu` with code similar to

```
hpt = uimenu(...,'CreateFcn','set(gcbo,...
''Visible'',''off'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uimenu` call. In the example above, if instead of redefining the `CreateFcn` property for this uimenu, you had explicitly set `Visible` to `off`, the default `CreateFcn` callback would have set `Visible` back to the default, i.e., `on`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Delete uimenu callback routine. A callback routine that executes when you delete the uimenu object (e.g., when you issue a `delete` command or cause the figure containing the uimenu to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root `CallbackObject` property, which is more simply queried using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`Enable`
`{on} | off`

Enable or disable the uimenu. This property controls whether a menu item can be selected. When not enabled (set to `off`), the menu `Label` appears dimmed, indicating the user cannot select it.

`ForegroundColor`
`ColorSpec` X-Windows only

Color of menu label string. This property determines color of the text defined for the `Label` property. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default text color is black. See `ColorSpec` for more information on specifying color.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.

- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`,

getframe, pause, or waitfor functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure's WindowButtonDownFcn callback routine, or an object's ButtonDownFcn or Callback routine are processed according to the rules described above.

`Label`

string

Menu label. A string specifying the text label on the menu item. You can specify a mnemonic for the label using the '&' character. Except as noted below, the character that follows the '&' in the string appears underlined and selects the menu item when you type **Alt+** followed by that character while the menu is visible. The '&' character is not displayed. To display the '&' character in a label, use two '&' characters in the string:

'O&pen selection' yields **Open selection**

'Save && Go' yields **Save & Go**

'Save&&Go' yields **Save & Go**

'Save& Go' yields **Save& Go** (the space is not a mnemonic)

There are three reserved words: `default`, `remove`, `factory` (case sensitive). If you want to use one of these reserved words in the `Label` property, you must precede it with a backslash (`'\'`) character. For example:

`'\remove'` yields **remove**

`'\default'` yields **default**

`'\factory'` yields **factory**

`Parent`
handle

Uimenu's parent. The handle of the uimenu's parent object. The parent of a uimenu object is the figure on whose menu bar it displays, or the uimenu of which it is a submenu. You can move a uimenu object to another figure by setting this property to the handle of the new parent.

`Position`
scalar

Relative menu position. The value of `Position` indicates placement on the menu bar or within a menu. Top-level menus are placed from left to right on the menu bar according to the value of their `Position` property, with 1 representing the left-most position. The individual items within a given menu are placed from top to bottom according to the value of their `Position` property, with 1 representing the top-most position.

`Separator`
`on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line above the menu item.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`
string (read only)

Class of graphics object. For uimenu objects, `Type` is always the string `'uimenu'`.

`UserData`
matrix

User-specified data. Any matrix you want to associate with the uimenu object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Uimenu visibility. By default, all uimenus are visible. When set to `off`, the uimenu is not visible, but still exists and you can query and set its properties.

Purpose

Convert to unsigned integer

Syntax

```
I = uint8(X)
I = uint16(X)
I = uint32(X)
I = uint64(X)
```

Description

`I = uint*(X)` converts the elements of array `X` into unsigned integers. `X` can be any numeric object (such as a `double`). The results of a `uint*` operation are shown in the next table.

Operation	Output Range	Output Type	Bytes per Element	Output Class
`uint8`	0 to 255	Unsigned 8-bit integer	1	`uint8`
`uint16`	0 to 65,535	Unsigned 16-bit integer	2	`uint16`
`uint32`	0 to 4,294,967,295	Unsigned 32-bit integer	4	`uint32`
`uint64`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	8	`uint64`

`double` and `single` values are rounded to the nearest `uint*` value on conversion. A value of `X` that is above or below the range for an integer class is mapped to one of the endpoints of the range. For example,

```
uint16(70000)
ans =
   65535
```

If `X` is already an unsigned integer of the same class, then `uint*` has no effect.

You can define or overload your own methods for `uint*` (as you can for any object) by placing the appropriately named method in an `@uint*` directory within a directory on your path. Type `help datatypes` for the names of the methods you can overload.

Remarks

Most operations that manipulate arrays without changing their elements are defined for integer values. Examples are `reshape`, `size`, the logical and relational operators, subscripted assignment, and subscripted reference.

Some arithmetic operations are defined for integer arrays on interaction with other integer arrays of the same class (e.g., where both operands are `uint16`). Examples of these operations are `+`, `-`, `.*`, `./`, `.\` and `.^`. If at least one operand is scalar, then `*`, `/`, `\`, and `^` are also defined. Integer arrays may also interact with scalar `double` variables, including constants, and the result of the operation is an integer array of the same class. Integer arrays saturate on overflow in arithmetic.

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the `zeros`, `ones`, or `eye` function. For example, to create a 100-by-100 `uint64` array initialized to zero, type

```
I = zeros(100, 100, 'uint64');
```

An easy way to find the range for any MATLAB integer type is to use the `intmin` and `intmax` functions as shown here for `uint32`:

```
intmin('uint32')              intmax('uint32')
ans =                         ans =
   0                             4294967295
```

See Also

`double`, `single`, `int8`, `int16`, `int32`, `int64`, `intmax`, `intmin`

Purpose

Open file selection dialog box with appropriate file filters

Syntax

```
uiopen
uiopen('MATLAB')
uiopen('LOAD')
uiopen('FIGURE')
uiopen('SIMULINK')
uiopen('EDITOR')
```

Description

`uiopen` displays a modal file selection dialog from which a user can select a file to open. The dialog is the same as the one displayed when you select **Open** from the **File** menu in the MATLAB desktop.

Selecting a file in the dialog and clicking **Open** does the following:

- Gets the file using `uigetfile`
- Opens the file in the base workspace using the `open` command

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

Note `uiopen` cannot be compiled. If you want to create a file selection dialog that can be compiled, use `uigetfile`.

`uiopen` or `uiopen('MATLAB')` displays the dialog with the file filter set to all MATLAB files.

`uiopen('LOAD')` displays the dialog with the file filter set to MAT-files (*.mat).

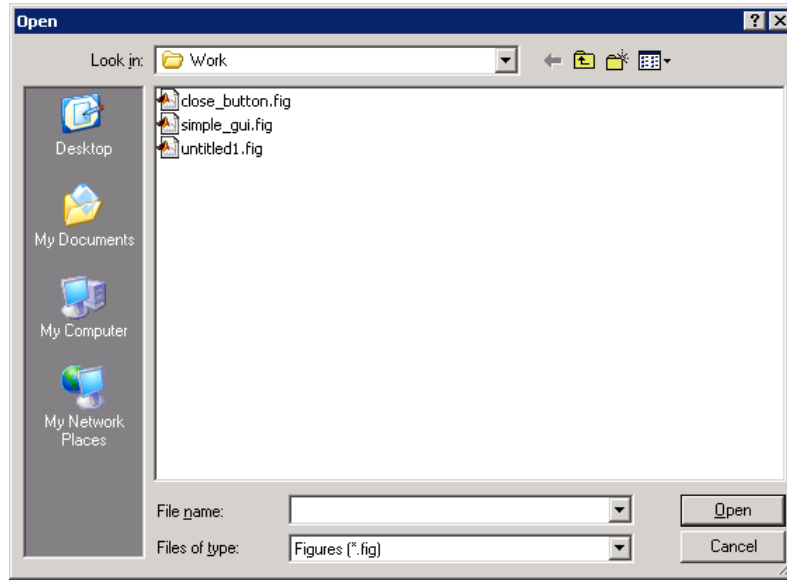
`uiopen('FIGURE')` displays the dialog with the file filter set to figure files (*.fig).

`uiopen('SIMULINK')` displays the dialog with the file filter set to model files (*.mdl).

`uiopen('EDITOR')` displays the dialog with the file filter set to all MATLAB files except for MAT-files and FIG-files. All files are opened in the MATLAB Editor.

Examples

Typing `uiopen('figure')` sets the **Files of type** field to `Figures (*.fig)`:



See Also

`uigetfile`, `uiputfile`, `uisave`

Purpose

Create panel container object

Syntax

```
h = uipanel('PropertyName1',value1,'PropertyName2',value2,
   ...)
h = uipanel(parent,'PropertyName1',value1,'PropertyName2',
   value2,...)
```

Description

A uipanel groups components. It can contain user interface controls with which the user interacts directly. It can also contain axes, other uipanels, and uibuttongroups. It cannot contain ActiveX controls.

`h = uipanel('PropertyName1',value1,'PropertyName2',value2,...)` creates a `uipanel` container object in a figure, uipanel, or uibuttongroup. Use the `Parent` property to specify the parent figure, uipanel, or uibuttongroup. If you do not specify a parent, `uipanel` adds the panel to the current figure. If no figure exists, one is created. See the Uipanel Properties reference page for more information.

`h = uipanel(parent,'PropertyName1',value1,'PropertyName2',value2,...)` creates a uipanel in the object specified by the handle, `parent`. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence. `parent` must be a figure, uipanel, or uibuttongroup.

A `uipanel` object can have `axes`, `uicontrol`, `uipanel`, and `uibuttongroup` objects as children. For the children of a uipanel, the `Position` property is interpreted relative to the uipanel. If you move the panel, the children automatically move with it and maintain their positions relative to the panel.

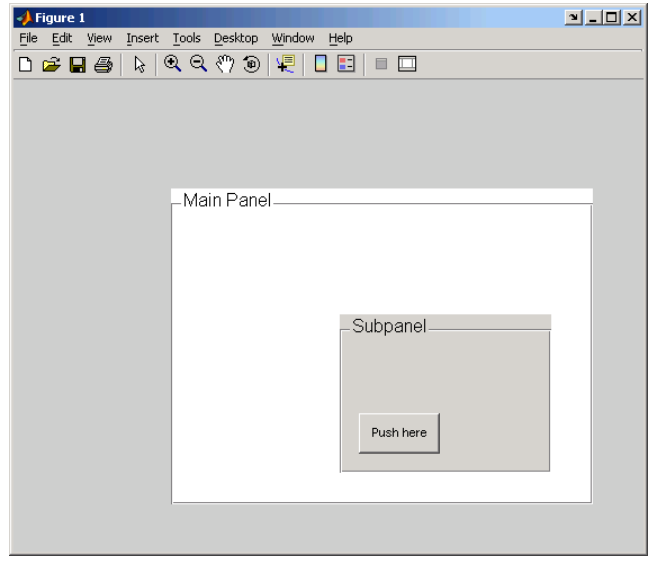
After creating a `uipanel` object, you can set and query its property values using `set` and `get`.

Examples

This example creates a uipanel in a figure, then creates a subpanel in the first panel. Finally, it adds a pushbutton to the subpanel. Both

panels use the default `Units` property value, `normalized`. Note that default `Units` for the `uicontrol` pushbutton is pixels.

```
h = figure;
hp = uipanel('Title','Main Panel','FontSize',12,...
             'BackgroundColor','white',...
             'Position',[.25 .1 .67 .67]);
hsp = uipanel('Parent',hp,'Title','Subpanel','FontSize',12,...
              'Position',[.4 .1 .5 .5]);
hbsp = uicontrol('Parent',hsp,'String','Push here',...
              'Position',[18 18 72 36]);
```



See Also

`hgtransform`, `uibuttongroup`, `uicontrol`

Purpose

Describe panel properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default uipanel properties by typing:

```
set(h,'DefaultUipanelPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, or a uipanel handle. *`PropertyName`* is the name of the uipanel property and `PropertyValue` is the value you specify as the default for that property.

Note Default properties you set for uipanels also apply to uibuttongroups.

For more information about changing the default value of a property see "Setting Default Property Values". For an example, see the `CreateFcn` property.

Uipanel Properties

This section lists all properties useful to `uipanel` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property Name	Description
`BackgroundColor`	Color of the uipanel background
`BorderType`	Type of border around the uipanel area.

Property Name	Description
BorderWidth	Width of the panel border.
BusyAction	Interruption of other callback routines
ButtonDownFcn	Button-press callback routine
Children	All children of the uipanel
Clipping	Clipping of child axes, uipanels, and uibuttongroups to the uipanel. Does not affect child uicontrols.
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
FontAngle	Title font angle
FontName	Title font name
FontSize	Title font size
FontUnits	Title font units
FontWeight	Title font weight
ForegroundColor	Title font color and/or color of 2-D border line
HandleVisibility	Handle accessibility from commandline and GUIs
HighlightColor	3-D frame highlight color
Interruptible	Callback routine interruption mode
Parent	Uipanel object's parent
Position	Panel position relative to parent figure or uipanel
ResizeFcn	User-specified resize routine
Selected	Whether object is selected

Property Name	Description
SelectionHighlight	Object highlighted when selected
ShadowColor	3-D frame shadow color
Tag	User-specified object identifier
Title	Title string
TitlePosition	Location of title string in relation to the panel
Type	Object class
UIContextMenu	Associates uicontextmenu with the uipanel
Units	Units used to interpret the position vector
UserData	User-specified data
Visible	Uipanel visibility. **Note** Controls the Visible property of child axes, uibuttongroups. and uipanels. Does not affect child uicontrols.

`BackgroundColor`
`ColorSpec`

Color of the uipanel background. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the `ColorSpec` reference page for more information on specifying color.

`BorderType`
`none | {etchedin} | etchedout | beveledin | beveledout | line`

Border of the uipanel area. Used to define the panel area graphically. Etched and beveled borders provide a 3-D look. Use

the HighlightColor and ShadowColor properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the ForegroundColor property to specify its color.

BorderWidth
integer

Width of the panel border. The width of the panel borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

BusyAction
cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. See the Interruptible property for information about controlling a callback's interruptibility.

ButtonDownFcn
string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uipanel. This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using the selectmoveresize function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children`
vector of handles

Children of the uipanel. A vector containing the handles of all children of the uipanel. A `uipanel` object's children are axes, uipanels, uibuttongroups, and uicontrols. You can use this property to reorder the children.

`Clipping`
`{on} | off`

Clipping mode. By default, MATLAB clips a uipanel's child axes, uipanels, and uibuttongroups to the uipanel rectangle. If you set `Clipping` to `off`, the axis, uipanel, or uibuttongroup is displayed outside the panel rectangle. This property does not affect child uicontrols which, by default, can display outside the panel rectangle.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uipanel` object. MATLAB sets all property values for the uipanel before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uipanel being created.

Setting this property on an existing uipanel object has no effect.

You can define a default `CreateFcn` callback for all new uipanels. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uipanel`. For example, the code

```
set(0,'DefaultUipanelCreateFcn','set(gcbo,...
    ''FontName'',''arial'',''FontSize'',12)')
```

creates a default `CreateFcn` callback that runs whenever you create a new panel. It sets the default font name and font size of the uipanel title.

Note Uibuttongroup takes its default property values from uipanel. Defining a default property for all uipanels defines the same default property for all uibuttongroups.

To override this default and create a panel whose `FontName and FontSize` properties are set to different values, call `uipanel` with code similar to

```
hpt = uipanel(...,'CreateFcn','set(gcbo,...
''FontName'',''times'',''FontSize'',14)')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uipushtool` call. In the example above, if instead of redefining the `CreateFcn` property for this uipanel, you had explicitly set `Fontsize` to `14`, the default `CreateFcn` callback would have set `FontSize` back to the system dependent default.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uipanel object (e.g., when you issue a delete command or clear the figure containing the uipanel). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine. The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`FontAngle`
`{normal} | italic | oblique`

Character slant used in the Title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`
string

Font family used in the Title. The name of the font in which to display the `Title`. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set `FontName` to the string `FixedWidth` (this string value is case insensitive).

```
set(uicontrol_handle,'FontName','FixedWidth')
```

This then uses the value of the root `FixedWidthFontName` property which can be set to the appropriate value for a locale

from `startup.m` in the end user's environment. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font

`FontSize`
integer

Title font size. A number specifying the size of the font in which to display the Title, in units determined by the `FontUnits` property. The default size is system dependent.

`FontUnits`
`inches | centimeters | normalized | {points} |pixels`

Title font size units. `Normalized` units interpret `FontSize` as a fraction of the height of the uipanel. When you resize the uipanel, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of characters in the title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
`ColorSpec`

Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HighlightColor`
`ColorSpec`

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the `ColorSpec` reference page for more information on specifying color.

```
Interruptible
     {on} | off
```

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Parent`
handle

Uipanel parent. The handle of the uipanel's parent figure, uipanel, or uibuttongroup. You can move a uipanel object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`
position rectangle

Size and location of uipanel relative to parent. The rectangle defined by this property specifies the size and location of the panel within the parent figure window, uipanel, or uibuttongroup. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the parent object to the lower-left corner of the `uipanel` object. `width` and `height` are the dimensions of the uipanel rectangle, including the title. All measurements are in units specified by the `Units` property.

`ResizeFcn`
string or function handle

Resize callback routine. MATLAB executes this callback routine whenever a user resizes the uipanel and the figure `Resize` property is set to `on`, or in GUIDE, the Resize behavior option is set to `Other`. You can query the uipanel `Position` property to determine its new size and position. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

You can use `ResizeFcn` to maintain a GUI layout that is not directly supported by the MATLAB `Position/Units` paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following `ResizeFcn` accomplishes this; it keeps the uicontrol whose `Tag` is `'StatusBar'` 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the `Tag` property to retrieve the uicontrol handle, and the `gcbo` function to retrieve the figure handle. Also note the defensive programming regarding figure `Units`, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards.

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure `Position` from within the `ResizeFcn` callback; however, the `ResizeFcn` is not called again as a result.

Note that the `print` command can cause the `ResizeFcn` to be called if the `PaperPositionMode` property is set to `manual` and you have defined a resize function. If you do not want your resize function called by `print`, set the `PaperPositionMode` to `auto`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See Resize Behavior for information on creating resize functions using GUIDE.

`Selected`
`on | off` (read only)

Is object selected? This property indicates whether the panel is selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`ShadowColor`
`ColorSpec`

3-D frame shadow color. A three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the `ColorSpec` reference page for more information on specifying color.

`Tag`
string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the `Tag` value `'FormatTb'`.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

`Title`

string

Title string. The text displayed in the panel title. You can position the title using the `TitlePosition` property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('|') characters are not interpreted as line breaks and instead show up in the text displayed in the uipanel title.

Setting a property value to `default`, `remove`, or `factory` produces the effect described in "Defining Default Values". To set `Title` to one of these words, you must precede the word with the backslash character. For example,

```
hp = uipanel(...,'Title','\Default');
```

`TitlePosition`

`{lefttop} | centertop | righttop | leftbottom | centerbottom | rightbottom`

Location of the title. This property determines the location of the title string, in relation to the uipanel.

`Type`
`string` (read-only)

Object class. This property identifies the kind of graphics object. For uipanel objects, `Type` is always the string `'uipanel'`.

`UIContextMenu`
handle

Associate a context menu with a uipanel. Assign this property the handle of a `Uicontextmenu` object. MATLAB displays the context menu whenever you right-click the `uipanel`. Use the `uicontextmenu` function to create the context menu.

`Units`
`inches | centimeters | {normalized} | points | pixels | characters`

Units of measurement. MATLAB uses these units to interpret the `Position` property. For the panel itself, units are measured from the lower-left corner of the figure window. For children of the panel, they are measured from the lower-left corner of the panel.

- `Normalized` units map the lower-left corner of the panel or figure window to (0,0) and the upper-right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).
- `Character` units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the `uipanel` object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`
`{on} | off`

Uipanel visibility. By default, a `uipanel` object is visible. When set to `off`, the uipanel is not visible, but still exists and you can query and set its properties.

Note The value of a uipanel's `Visible` property also controls the `Visible` property of child axes, uipanels, and uibuttongroups. This property does not affect the `Visible` property of child uicontrols.

Purpose

Create push button on toolbar

Syntax

```
hpt = uipushtool('PropertyName1',value1,'PropertyName2',
   value2,...)
hpt = uipushtool(ht,...)
```

Description

`hpt = uipushtool('PropertyName1',value1,'PropertyName2',value2,...)` creates a push button on the uitoolbar at the top of the current figure window, and returns a handle to it. `uipushtool` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function.

Type `get(hpt)` to see a list of uipushtool object properties and their current values. Type `set(hpt)` to see a list of uipushtool object properties that you can set and their legal property values. See the Uipushtool Properties reference page for more information.

`hpt = uipushtool(ht,...)` creates a button with `ht` as a parent. `ht` must be a uitoolbar handle.

Remarks

`uipushtool` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

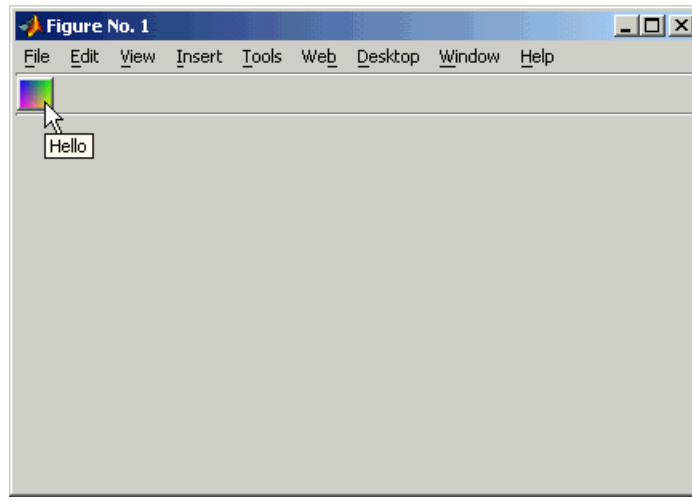
Uipushtools appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures whose `WindowStyle` is `modal`. If the `WindowStyle`of a figure containing a uitoolbar and its uipushtool children is changed to `modal`, the uipushtools still exist and are contained in the `Children` list of the uitoolbar, but are not displayed until the figure `WindowStyle` is changed to `normal` or `docked`.

Examples

This example creates a uitoolbar object and places a uipushtool object on it.

```
h = figure('ToolBar','none')
ht = uitoolbar(h)
a = [.20:.05:0.95];
```

```
b(:,:,1) = repmat(a,16,1)';
b(:,:,2) = repmat(a,16,1);
b(:,:,3) = repmat(flipdim(a,2),16,1);
hpt = uipushtool(ht,'CData',b,'TooltipString','Hello')
```



See Also

get, set, uicontrol, uitoggletool, uitoolbar

Purpose

Describe push tool properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uipushtool properties by typing:

```
set(h,'DefaultUipushtool PropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, a uitoolbar handle, or a uipushtool handle. *`PropertyName`* is the name of the `Uipushtool` property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

Uipushtool Properties

This section lists all properties useful to `uipushtool` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption.
`CData`	Truecolor image displayed on the control.
`ClickedCallback`	Control action.
`CreateFcn`	Callback routine executed during object creation.
`DeleteFcn`	Delete uipushtool callback routine.

Property	Purpose
Enable	Enable or disable the uipushtool.
HandleVisibility	Control access to object's handle.
HitTest	Whether selectable by mouse click
Interruptible	Callback routine interruption mode.
Parent	Handle of uipushtool's parent.
Separator	Separator line mode
Tag	User-specified object label.
TooltipString	Content of object's tooltip.
Type	Object class.
UIContextMenu	Uicontextmenu object associated with the uipushtool
UserData	User specified data.
Visible	Uipushtool visibility.

`BeingDeleted`
`on | {off}` (read only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`CData`

3-dimensional array

Truecolor image displayed on control. An *n*-by-*m*-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your `CData` array is larger than 16 in the first or second dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

`ClickedCallback`

string or function handle

Control action. A routine that executes when the uipushtool's `Enable` property is set to `on`, and you press a mouse button while the pointer is on the push tool itself or in a 5-pixel wide border around it.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uipushtool object. MATLAB sets all property values for the uipushtool before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the push tool being created.

Setting this property on an existing uipushtool object has no effect.

You can define a default `CreateFcn` callback for all new uipushtools. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uipushtool`. For example, the code

```
imga(:,:,1) = rand(20);
imga(:,:,2) = rand(20);
imga(:,:,3) = rand(20);
set(0,'DefaultUipushtoolCreateFcn','set(gcbo,''Cdata'',imga)')
```

creates a default `CreateFcn` callback that runs whenever you create a new push tool. It sets the default image `imga` on the push tool.

To override this default and create a push tool whose `Cdata` property is set to a different image, call `uipushtool` with code similar to

```
a = [.05:.05:0.95];
imgb(:,:,1) = repmat(a,19,1)';
imgb(:,:,2) = repmat(a,19,1);
```

```
imgb(:,:,3) = repmat(flipdim(a,2),19,1);
hpt = uipushtool(...,'CreateFcn','set(gcbo,''CData'',imgb)',..
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uipushtool` call. In the example above, if instead of redefining the `CreateFcn` property for this push tool, you had explicitly set `CData` to `imgb`, the default `CreateFcn` callback would have set `CData` back to `imga`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uipushtool object (e.g., when you call the `delete` function or cause the figure containing the uipushtool to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`Enable`

`{on} | off`

Enable or disable the uipushtool. This property controls how uipushtools respond to mouse button clicks, including which callback routines execute.

- `on` – The uipushtool is operational (the default).
- `off` – The uipushtool is not operational and its image (set by the `Cdata` property) is grayed out.

When you left-click on a uipushtool whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Sets the figure's `SelectionType` property.
2. Executes the push tool's `ClickedCallback` routine.
3. Does not set the figure's `CurrentPoint` property and does not execute the figure's `WindowButtonDownFcn` callback.

When you left-click on a uipushtool whose `Enable` property is `off`, or when you right-click a uipushtool whose `Enable` property has any value, MATLAB performs these actions in this order:

4. Sets the figure's `SelectionType` property.
5. Sets the figure's `CurrentPoint` property.
6. Executes the figure's `WindowButtonDownFcn` callback.
7. Does not execute the push tool's `ClickedCallback` routine.

`HandleVisibility`

```
{on} | callback | off
```

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's

handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HitTest`

```
{on} | off
```

Selectable by mouse click. This property has no effect on uipushtool objects.

`Interruptible`

```
{on} | off
```

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing

- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Parent`
handle

Uipushtool parent. The handle of the uipushtool's parent toolbar. You can move a uipushtool object to another toolbar by setting this property to the handle of the new parent.

`Separator`
`on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line to the left of the uipushtool.

`Tag`
string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the `Tag` value `'Copy'`.

```
h = findobj(uitoolbarhandles,'Tag','Copy')
```

`TooltipString`
string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uipushtool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

`Type`
string (read-only)

Object class. This property identifies the kind of graphics object. For uipushtool objects, `Type` is always the string `'uipushtool'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uipushtool objects.

`UserData`
array

User specified data. You can specify `UserData` as any array you want to associate with the uipushtool object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Uipushtool visibility. By default, all uipushtools are visible. When set to `off`, the uipushtool is not visible, but still exists and you can query and set its properties.

Purpose

Open standard dialog box for saving files

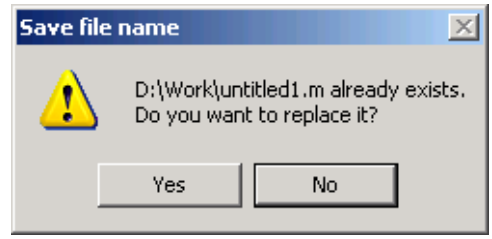
Syntax

```
uiputfile
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec)
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,
   DialogTitle)
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,
   DialogTitle,DefaultName)
```

Description

`uiputfile` displays a modal dialog box used to select or specify a file for saving. The dialog box lists the files and directories in the current directory. If the selected or specified filename is valid, it is returned in `ans`.

If an existing filename is selected or specified, the following warning dialog box is displayed.



The user can select **Yes** to replace the existing file or **No** to return to the dialog to select another filename. If the user selects **Yes**, `uiputfile` returns the name of the file. If the user selects **No**, `uiputfile` returns `0`.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the`uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec)` displays only those files with extensions that match `FilterSpec`. `FilterSpec` can be a string or a cell array of strings, and can include the `*` wildcard. For example, `'*.m'` lists all the MATLAB M-files. A `FilterSpec` string can also be a filename. In this case the filename becomes the default filename and the file's extension is used as the default filter. If `FilterSpec` is a string, `uiputfile` appends `'All Files'` to the list of file types.

If `FilterSpec` is a cell array, the first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace standard descriptions in the **Files of type** field. A description cannot be an empty string. "Example 3" on page 2-3562 and "Example 4" on page 2-3563 illustrate use of a cell array as `FilterSpec`.

If `FilterSpec` is not specified, `uiputfile` uses the default list of file types (i.e., all MATLAB files).

After the user clicks **Save** and if the filename is valid,`uiputfile` returns the name of the selected file in `FileName` and its path in `PathName`. If the user clicks the **Cancel** button, closes the dialog window, or if the filename is not valid, `FileName` and `PathName` are set to `0`.

`FilterIndex` is the index of the filter selected in the dialog box. Indexing starts at 1. If the user clicks the **Cancel** button, closes the dialog window, or if the file does not exist, `FilterIndex` is set to `0`.

If no output arguments are specified, the filename is returned in `ans`.

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle)` displays a dialog box that has the title `DialogTitle`. To use the default file types and specify a dialog title, enter

```
uiputfile('',DialogTitle)
```

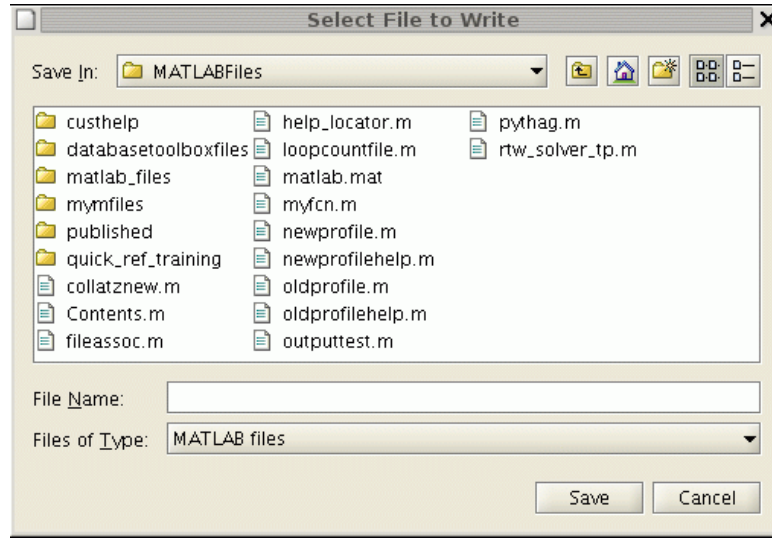
`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle,DefaultName)` displays a dialog box in which the filename specified by `DefaultName` appears in the

File name field. `DefaultName` can also be a path or a path/filename. In this case, `uigetfile` opens the dialog box in the directory specified by the path. See “Example 6” on page 2-3565. If the path does not include a filename, it must end with a slash (`/`) or backslash (`\`) separator. For example, `'C:\Work\'`. Note that `uiputfile` recognizes both `'./'` and `'../'` as valid values. If the specified path does not exist, `uiputfile` opens the dialog box in the current directory.

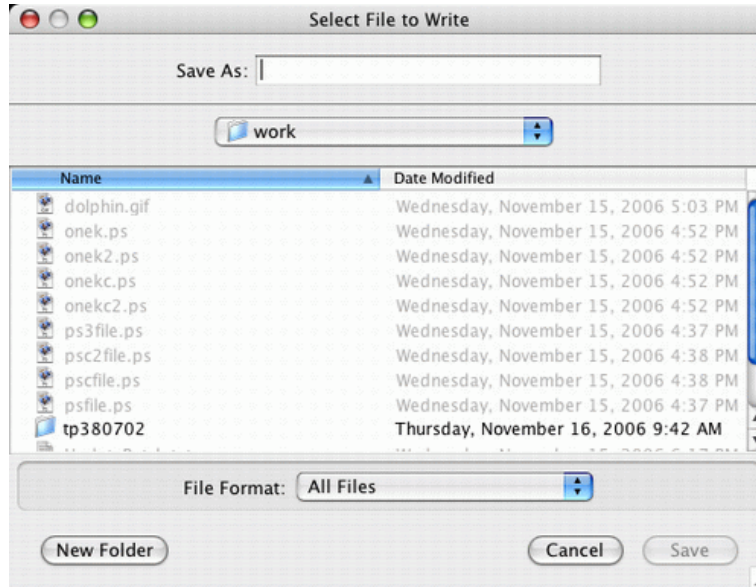
Remarks

For Windows platforms, the dialog box is the Windows dialog box native to your platform. Because of this, it may differ from those shown in the examples below.

For UNIX platforms, the dialog box is similar to the one shown in the following figure.



For Mac platforms, the dialog box is similar to the one shown in the following figure.

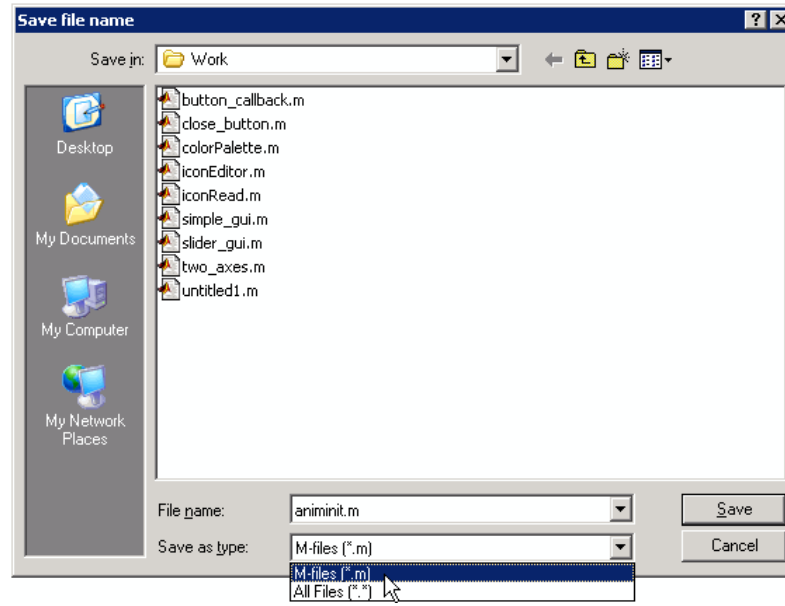


Examples

Example 1

The following statement displays a dialog box titled `'Save file name'` with the **Filename** field set to `animinit.m` and the filter set to `M-files (*.m)`. Because `FilterSpec` is a string, the filter also includes `All Files (*.*)`

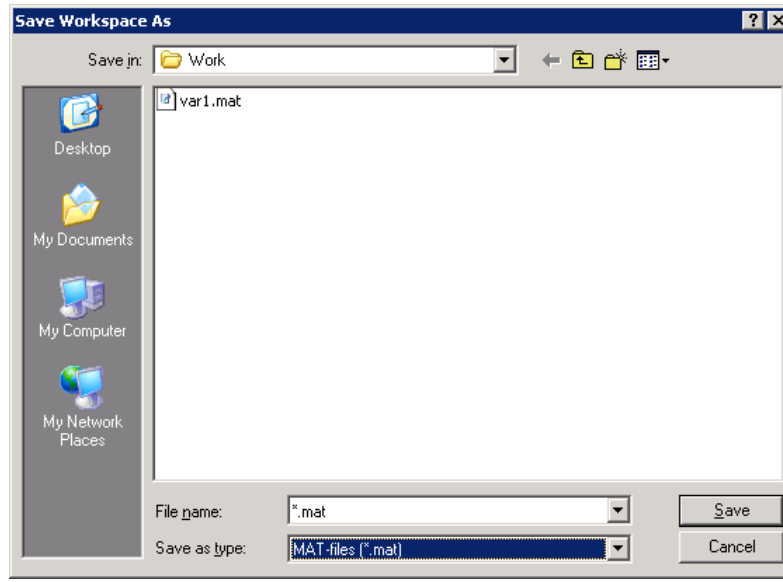
```
[file,path] = uiputfile('animinit.m','Save file name');
```



Example 2

The following statement displays a dialog box titled `'Save Workspace As'` with the filter specifier set to MAT-files.

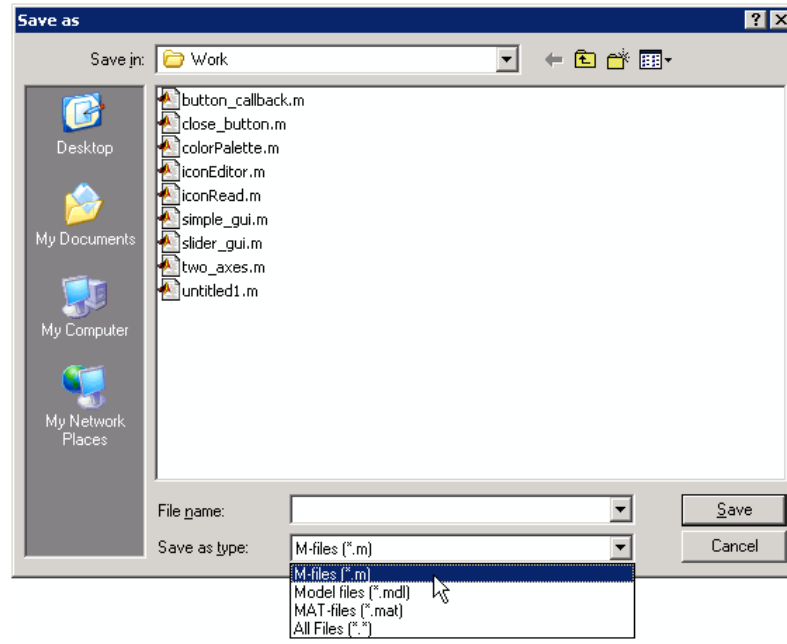
```
[file,path] = uiputfile('*.mat','Save Workspace As');
```



Example 3

To display several file types in the **Save as type** list box, separate each file extension with a semicolon, as in the following code. Note that `uiputfile` displays a default description for each known file type, such as "Simulink Models" for `.mdl` files.

```
[filename, pathname] = uiputfile(...
 {'*.m';'*.mdl';'*.mat';'*.*'},...
 'Save as');
```

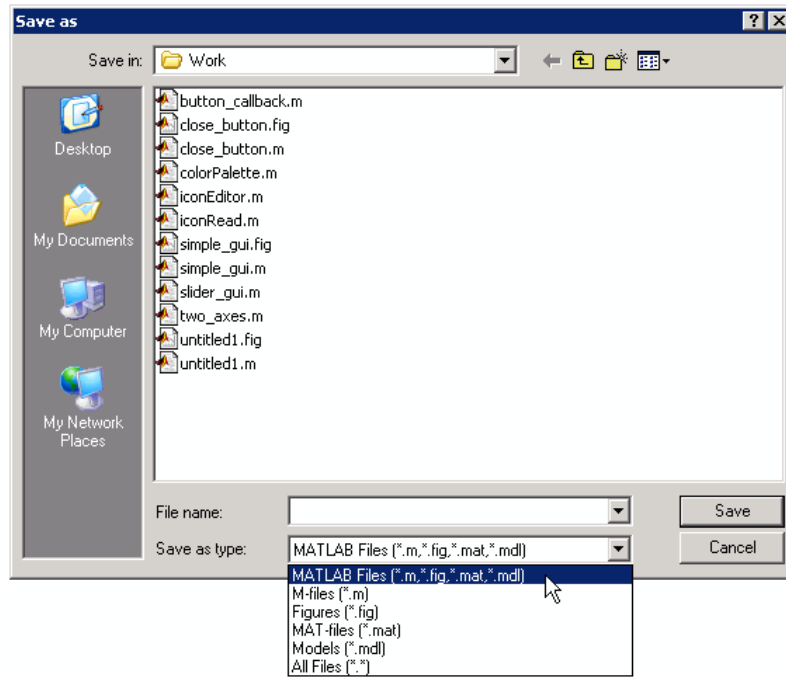


Example 4

If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the `'MATLAB Files'` description.

```
[filename, pathname, filterindex] = uiputfile( ...
{'*.m;*.fig;*.mat;*.mdl','MATLAB Files (*.m,*.fig,*.mat,*.mdl)';
 '*.m', 'M-files (*.m)';...
 '*.fig','Figures (*.fig)';...
 '*.mat','MAT-files (*.mat)';...
 '*.mdl','Models (*.mdl)';...
 '*.*',  'All Files (*.*)'},...
 'Save as');
```

The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. Note that the first entry of column one contains several extensions, separated by semicolons, all of which are associated with the description `'MATLAB Files (*.m,*.fig,*.mat,*.mdl)'`. The code produces the dialog box shown in the following figure.



Example 5

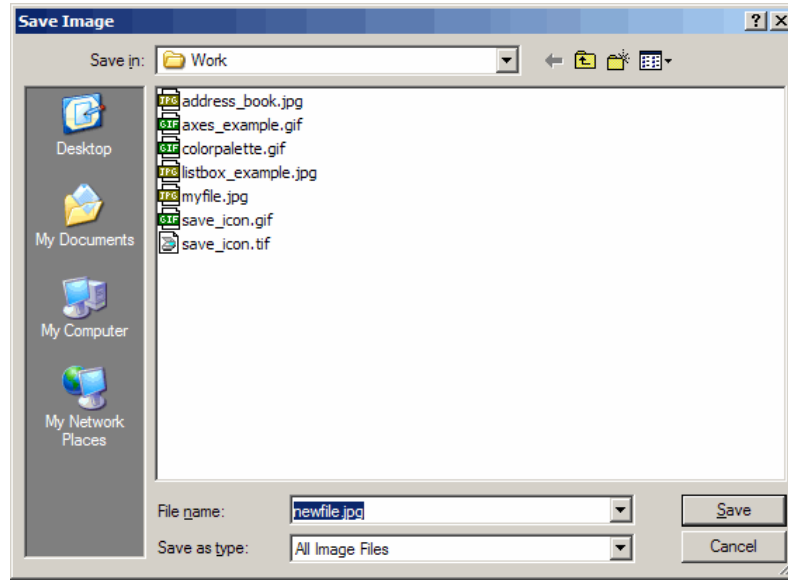
The following code checks for the existence of the file and displays a message about the result of the open operation.

```
[filename, pathname] = uiputfile('*.m','Pick an M-file');
if isequal(filename,0) | isequal(pathname,0)
   disp('User selected Cancel')
else
```

```
    disp(['User selected',fullfile(pathname,filename)])
end
```

Example 6

```
uiputfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'Save Image',...
          'C:\Work\newfile.jpg')
```



See Also

uigetdir, uigetfile

Purpose

Control program execution

Syntax

```
uiwait
uiwait(h)
uiwait(h,timeout)
uiresume(h)
```

Description

The `uiwait` and `uiresume` functions block and resume MATLAB program execution.

`uiwait` blocks execution until `uiresume` is called or the current figure is deleted. This syntax is the same as `uiwait(gcf)`.

`uiwait(h)` blocks execution until `uiresume` is called or the figure `h` is deleted.

`uiwait(h,timeout)` blocks execution until `uiresume` is called, the figure `h` is deleted, or `timeout` seconds elapse.

`uiresume(h)` resumes the M-file execution that `uiwait` suspended.

Remarks

When creating a dialog, you should have a uicontrol component with a callback that calls `uiresume` or a callback that destroys the dialog box. These are the only methods that resume program execution after the `uiwait` function blocks execution.

`uiwait` is a convenient way to use the `waitfor` command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the M-file that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, uiwait/uiresume can block the execution of the M-file *and* restrict user interaction to the dialog only.

Example

This example creates a GUI with a **Continue** push button. The example calls `uiwait` to block MATLAB execution until `uiresume` is called. This happens when the user clicks the **Continue** push button because the push button's `Callback` callback, which responds to the click, calls `uiresume`.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

`gcbf` is the handle of the figure that contains the object whose callback is executing.

"Using a Modal Dialog to Confirm an Operation" is a more complex example for a GUIDE GUI. See "Icon Editor" for an example for a programmatically created GUI.

See Also

`uicontrol`, `uimenu`, `waitfor`, `figure`, `dialog`

Purpose

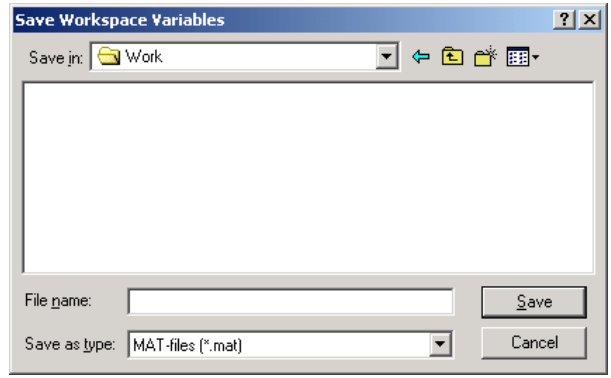
Open standard dialog box for saving workspace variables

Syntax

```
uisave
uisave(variables)
uisave(variables,filename)
```

Description

`uisave` displays the Save Workspace Variables dialog box for saving workspace variables to a MAT-file, as shown in the figure below. By default, the dialog box opens in your current directory.



Note The `uisave` dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

If you type a name in the **File name** field, such as `my_vars`, and click **Save**, the dialog saves all workspace variables in the file `my_vars.mat`. The default filename is `matlab.mat`.

`uisave(variables)` saves only the variables listed in `variables`. For a single variable, `variables` can be a string. For more than one variable, `variables` must be a cell array of strings.

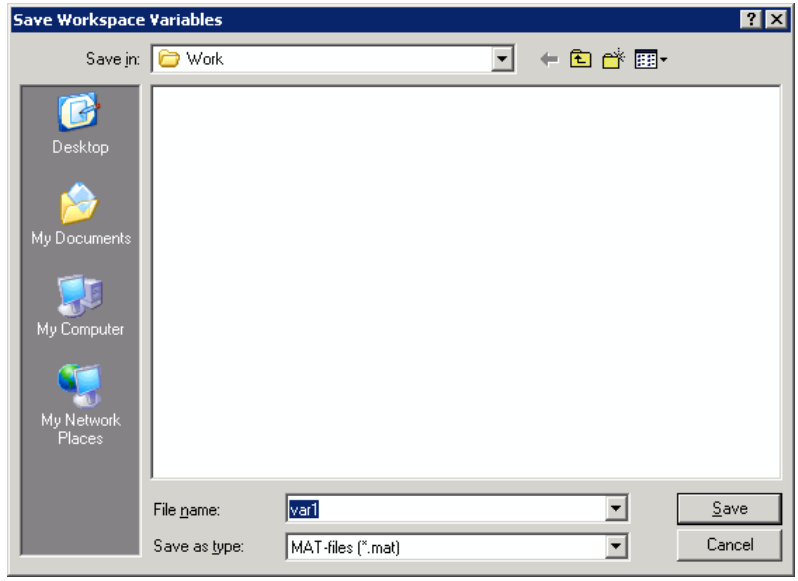
`uisave(variables,filename)` uses the specified `filename` as the default **File name** in the Save Workspace Variables dialog box.

Note `uisave` cannot be compiled. If you want to create a dialog that can be compiled, use `uiputfile`.

Example

This example creates workspace variables `h` and `g`, and then displays the Save Workspace Variables dialog box in the current directory with the default **File name** set to `var1`.

```
h = 365;
g = 52;
uisave({'h','g'},'var1');
```



Clicking **Save** stores the workspace variables `h` and `g` in the file `var1.mat` in the displayed directory.

See Also

uigetfile, uiputfile, uiopen

Purpose

Open standard dialog box for setting object's `ColorSpec`

Syntax

```
c = uisetcolor
c = uisetcolor([r g b])
c = uisetcolor(h)
c = uisetcolor(...,'dialogTitle')
```

Description

`c = uisetcolor` displays a modal color selection dialog appropriate to the platform, and returns the color selected by the user. The dialog box is initialized to `white`.

`c = uisetcolor([r g b])` displays a dialog box initialized to the specified color, and returns the color selected by the user. `r`, `g`, and `b` must be values between 0 and 1.

`c = uisetcolor(h)` displays a dialog box initialized to the color of the object specified by handle `h`, returns the color selected by the user, and applies it to the object. `h` must be the handle to an object containing a color property.

`c = uisetcolor(...,'dialogTitle')` displays a dialog box with the specified title.

If the user presses **Cancel** from the dialog box, or if any error occurs, the output value is set to the input RGB triple, if provided; otherwise, it is set to 0.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

See Also

`ColorSpec`

uisetfont

Purpose

Open standard dialog box for setting object's font characteristics

Syntax

```
uisetfont
uisetfont(h)
uisetfont(S)
uisetfont(...,'DialogTitle')
S = uisetfont(...)
```

Description

`uisetfont` enables you to change font properties (`FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle`) for a `text`, `axes`, or `uicontrol` object. The function returns a structure consisting of font properties and values. You can specify an alternate title for the dialog box.

`uisetfont` displays a modal dialog box and returns the selected font properties.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`uisetfont(h)` displays a modal dialog box, initializing the font property values with the values of those properties for the object whose handle is `h`. Selected font property values are applied to the current object. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(S)` displays a modal dialog box, initializing the font property values with the values defined for the specified structure (`S`). `S` must define legal values for one or more of these properties: `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` and the field names must match the property names exactly. If other properties are defined, they are ignored. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(...,'DialogTitle')` displays a modal dialog box with the title `DialogTitle` and returns the values of the font properties selected in the dialog box.

`S = uisetfont(...)` returns the properties `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` as fields in a structure. If the user presses **Cancel** from the dialog box or if an error occurs, the output value is set to 0.

Example

These statements create a text object, then display a dialog box (labeled Update Font) that enables you to change the font characteristics:

```
h = text(.5,.5,'Figure Annotation');
uisetfont(h,'Update Font')
```

These statements create two push buttons, then set the font properties of one based on the values set for the other:

```
% Create push button with string ABC
c1 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 10 100 20], 'String', 'ABC');
% Create push button with string XYZ
c2 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 50 100 20], 'String', 'XYZ');
% Display set font dialog box for c1, make selections,
& and save to d
d = uisetfont(c1);
% Apply those settings to c2
set(c2, d)
```

See Also

`axes`, `text`, `uicontrol`

uisetpref

Purpose

Manage preferences used in `uigetpref`

Syntax

```
uisetpref('clearall')
```

Description

`uisetpref('clearall')` resets the value of all preferences registered through `uigetpref` to `'ask'`. This causes the dialog box to display when you call `uigetpref`.

Note Use `setpref` to set the value of a particular preference to `'ask'`.

See Also

`setpref`, `uigetpref`

Purpose

Reorder visual stacking order of objects

Syntax

```
uistack(h)
uistack(h,stackopt)
uistack(h,stackopt,step)
```

Description

`uistack(h)` raises the visual stacking order of the objects specified by the handles in `h` by one level (`step` of 1). All handles in `h` must have the same parent.

`uistack(h,stackopt)` moves the objects specified by `h` in the stacking order, where *`stackopt`* is one of the following:

- `'up'` – moves `h` up one position in the stacking order
- `'down'` – moves `h` down one position in the stacking order
- `'top'` – moves `h` to the top of the current stack
- `'bottom'` – moves `h` to the bottom of the current stack

`uistack(h,stackopt,step)` moves the objects specified by `h` up or down the number of levels specified by `step`.

Note In a GUI, `axes` objects are always at a lower level than `uicontrol` objects. You cannot stack an `axes` object on top of a `uicontrol` object.

See "Setting Tab Order" in the MATLAB documentation for information about changing the tab order.

Example

The following code moves the child that is third in the stacking order of the figure handle `hObject` down two positions.

```
v = allchild(hObject)
uistack(v(3),'down',2)
```

uitoggletool

Purpose

Create toggle button on toolbar

Syntax

```
htt = uitoggletool('PropertyName1',value1,'PropertyName2',
   value2,...)
htt = uitoggletool(ht,...)
```

Description

`htt = uitoggletool('PropertyName1',value1,'PropertyName2',value2,...)` creates a toggle button on the uitoolbar at the top of the current figure window, and returns a handle to it. `uitoggletool` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function.

Type `get(htt)` to see a list of uitoggletool object properties and their current values. Type `set(htt)` to see a list of uitoggletool object properties you can set and legal property values. See the Uitoggletool Properties reference page for more information.

`htt = uitoggletool(ht,...)` creates a button with `ht` as a parent. `ht` must be a uitoolbar handle.

Remarks

`uitoggletool` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

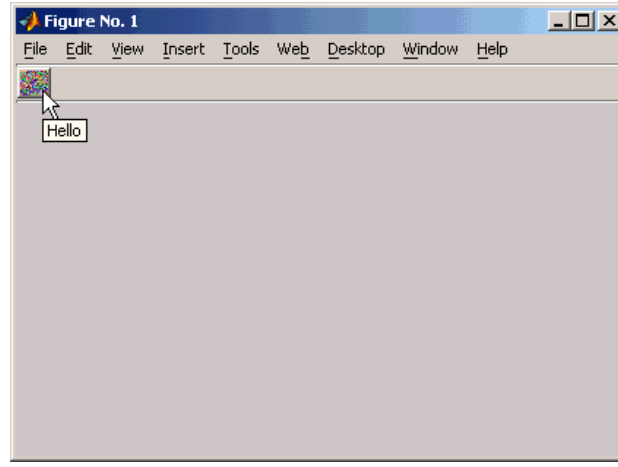
Toggle tools appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures whose `WindowStyle` is `modal`. If the `WindowStyle` property of a figure containing a tool bar and its toggle tool children is changed to `modal`, the toggle tools still exist and are contained in the `Children` list of the tool bar, but are not displayed until the `WindowStyle` is changed to `normal` or `docked`.

Examples

This example creates a `uitoolbar` object and places a `uitoggletool` object on it.

```
h = figure('ToolBar','none');
ht = uitoolbar(h);
a = rand(16,16,3);
```

```
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello');
```



See Also

get, set, uicontrol, uipushtool, uitoolbar

Uitoggletool Properties

Purpose

Describe toggle tool properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uitoggletool properties by typing:

```
set(h,'DefaultUitoggletool<PropertyName>',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, a uitoolbar handle, or a uitoggletool handle. *`PropertyName`* is the name of the Uitoggletool property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see "Setting Default Property Values".

Properties

This section lists all properties useful to `uitoggletool` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption.
`CData`	Truecolor image displayed on the toggle tool.
`ClickedCallback`	Control action independent of the toggle tool position.

Property	Purpose
CreateFcn	Callback routine executed during object creation.
DeleteFcn	Callback routine executed during object deletion.
Enable	Enable or disable the uitoggletool.
HandleVisibility	Control access to object's handle.
HitTest	Whether selectable by mouse click
Interruptible	Callback routine interruption mode.
OffCallback	Control action when toggle tool is set to the off position.
OnCallback	Control action when toggle tool is set to the on position.
Parent	Handle of uitoggletool's parent toolbar.
Separator	Separator line mode.
State	Uitoggletool state.
Tag	User-specified object label.
TooltipString	Content of object's tooltip.
Type	Object class.
UIContextMenu	Uicontextmenu object associated with the uitoggletool
UserData	User specified data.
Visible	Uitoggletool visibility.

`BeingDeleted`
`on | {off}` (read only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in

the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the`Interruptible` property for information about controlling a callback's interruptibility.

`CData`

3-dimensional array

Truecolor image displayed on control. An *n*-by-*m*-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your `CData` array is larger than 16 in the first or second dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

`ClickedCallback`
string or function handle

Control action independent of the toggle tool position. A routine that executes after either the `OnCallback` routine or `OffCallback` routine runs to completion. The uitoggletool's `Enable` property must be set to `on`.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uitoggletool object. MATLAB sets all property values for the uitoggletool before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the toggle tool being created.

Setting this property on an existing uitoggletool object has no effect.

You can define a default `CreateFcn` callback for all new uitoggletools. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uitoggletool`. For example, the statement,

```
set(0,'DefaultUitoggletoolCreateFcn',...
    'set(gcbo,''Enable'',''off'')'
```

creates a default CreateFcn callback that runs whenever you create a new toggle tool. It sets the toggle tool Enable property to off.

To override this default and create a toggle tool whose Enable property is set to on, you could call uitoggletool with code similar to

```
htt = uitoggletool(...,'CreateFcn',...
                   'set(gcbo,''Enable'',''on'')',...)
```

Note To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uitoggletool call. In the example above, if instead of redefining the CreateFcn property for this toggle tool, you had explicitly set Enable to on, the default CreateFcn callback would have set CData back to off.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

DeleteFcn

string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uitoggletool object (e.g., when you call the delete function or cause the figure containing the uitoggletool to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

`Enable`

{on} | off

Enable or disable the uitoggletool. This property controls how uitoggletools respond to mouse button clicks, including which callback routines execute.

- `on` – The uitoggletool is operational (the default).
- `off` – The uitoggletool is not operational and its image (set by the `Cdata` property) is grayed out.

When you left-click on a uitoggletool whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Sets the figure's `SelectionType` property.
2. Executes the toggle tool's `ClickedCallback` routine.
3. Does not set the figure's `CurrentPoint` property and does not execute the figure's `WindowButtonDownFcn` callback.

When you left-click on a uitoggletool whose `Enable` property is `off`, or when you right-click a uitoggletool whose `Enable` property has any value, MATLAB performs these actions in this order:

4. Sets the figure's `SelectionType` property.
5. Sets the figure's `CurrentPoint` property.
6. Executes the figure's `WindowButtonDownFcn` callback.
7. Does not execute the toggle tool's `OnCallback`, `OffCallback`, or `ClickedCallback` routines.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HitTest`

```
{on} | off
```

Selectable by mouse click. This property has no effect on uitoggletool objects.

`Interruptible`

```
{on} | off
```

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below).

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement.

`OffCallback`
string or function handle

Control action. A routine that executes if the uitoggletool's `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `off`.
- The toggle tool is set to the `off` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

`OnCallback`
string or function handle

Control action. A routine that executes if the uitoggletool's `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `on`.
- The toggle tool is set to the `on` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

`Parent`
handle

Uitoggletool parent. The handle of the uitoggletool's parent toolbar. You can move a uitoggletool object to another toolbar by setting this property to the handle of the new parent.

`Separator`
`on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line to left of the uitoggletool.

`State`

`on | {off}`

Uitoggletool state. When the state is `on`, the toggle tool appears in the down, or pressed, position. When the state is `off`, it appears in the up position. Changing the state causes the appropriate `OnCallback` or `OffCallback` routine to run.

`Tag`

string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the `Tag` value `'Bold'`.

```
h = findobj(uitoolbarhandles, 'Tag', 'Bold')
```

`TooltipString`

string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uitoggletool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

`Type`

`string` (read-only)

Object class. This property identifies the kind of graphics object. For uitoggletool objects, `Type` is always the string `'uitoggletool'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uitoggletool objects.

`UserData`
array

User specified data. You can specify `UserData` as any array you want to associate with the uitoggletool object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Uitoggletool visibility. By default, all uitoggletools are visible. When set to `off`, the uitoggletool is not visible, but still exists and you can query and set its properties.

Purpose

Create toolbar on figure

Syntax

```
ht =
uitoolbar('PropertyName1',value1,'PropertyName2',value2,
   ...)
ht = uitoolbar(h,...)
```

Description

`ht = uitoolbar('PropertyName1',value1,'PropertyName2',value2,...)` creates an empty toolbar at the top of the current figure window, and returns a handle to it. `uitoolbar` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function.

Type `get(ht)` to see a list of `uitoolbar` object properties and their current values. Type `set(ht)` to see a list of `uitoolbar` object properties that you can set and legal property values. See the Uitoolbar Properties reference page for more information.

`ht = uitoolbar(h,...)` creates a toolbar with `h` as a parent. `h` must be a figure handle.

Remarks

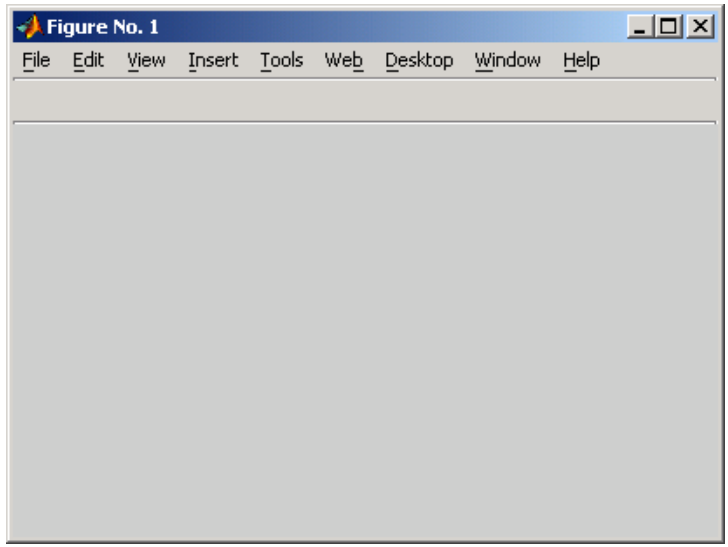
`uitoolbar` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

Uitoolbars appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures whose `WindowStyle` is `modal`. If the `WindowStyle` property of a figure containing a uitoolbar is changed to `modal`, the uitoolbar still exists and is contained in the `Children` list of the figure, but is not displayed until the `WindowStyle` is changed to `normal` or `docked`.

Example

This example creates a figure with no toolbar, then adds a toolbar to it.

```
h = figure('ToolBar','none')
ht = uitoolbar(h)
```



For more information on using the menus and toolbar in a MATLAB figure window, see the online MATLAB Graphics documentation.

See Also

`set`, `get`, `uicontrol`, `uipushtool`, `uitoggletool`

Purpose

Describe toolbar properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uitoolbar properties by typing:

```
set(h,'DefaultUitoolbar PropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, or a uitoolbar handle. *`PropertyName`* is the name of the Uitoolbar property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

Uitoolbar Properties

This section lists all properties useful to `uitoolbar` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption.
`Children`	Handles of uitoolbar's children.
`CreateFcn`	Callback routine executed during object creation.
`DeleteFcn`	Callback routine executed during object deletion.

Property	Purpose
HandleVisibility	Control access to object's handle.
HitTest	Whether selectable by mouse click
Interruptible	Callback routine interruption mode.
Parent	Handle of uitoolbar's parent.
Tag	User-specified object identifier.
Type	Object class.
UIContextMenu	Uicontextmenu object associated with the uitoolbar
UserData	User specified data.
Visible	Uitoolbar visibility.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new

event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

`Children`
vector of handles

Handles of tools on the toolbar. A vector containing the handles of all children of the uitoolbar object, in the order in which they appear on the toolbar. The children objects of uitoolbars are uipushtools and uitoggletools. You can use this property to reorder the children.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uitoolbar object. MATLAB sets all property values for the uitoolbar before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the toolbar being created.

Setting this property on an existing uitoolbar object has no effect.

You can define a default `CreateFcn` callback for all new uitoolbars. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uitoolbar`. For example, the statement,

```
set(0,'DefaultUitoolbarCreateFcn',...
    'set(gcbo,''Visibility'',''off'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new toolbar. It sets the toolbar visibility to `off`.

To override this default and create a toolbar whose `Visibility` property is set to `on`, you could call `uitoolbar` with a call similar to

```
ht = uitoolbar(...,'CreateFcn',...
                'set(gcbo,''Visibility'',''on'')',...)
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uitoolbar` call. In the example above, if instead of redefining the `CreateFcn` property for this toolbar, you had explicitly set `Visibility` to `on`, the default `CreateFcn` callback would have set `Visibility` back to `off`.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

`DeleteFcn`
: string or function handle

Callback routine executed during object deletion. A callback function that executes when the uitoolbar object is deleted (e.g., when you call the `delete` function or cause the figure containing the uitoolbar to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

Within the function, use `gcbo` to get the handle of the toolbar being deleted.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HitTest`

`{on} | off`

Selectable by mouse click. This property has no effect on uitoolbar objects.

`Interruptible`

`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property

of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Parent`

handle

Uitoolbar parent. The handle of the uitoolbar's parent figure. You can move a uitoolbar object to another figure by setting this property to the handle of the new parent.

`Tag`

string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the `Tag` value `'FormatTb'`.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

`Type`

`string` (read-only)

Object class. This property identifies the kind of graphics object. For uitoolbar objects, `Type` is always the string `'uitoolbar'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uitoolbar objects.

`UserData`
array

User specified data. You can specify `UserData` as any array you want to associate with the uitoolbar object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Uitoolbar visibility. By default, all uitoolbars are visible. When set to `off`, the uitoolbar is not visible, but still exists and you can query and set its properties.

Purpose

Undo previous checkout from source control system (UNIX)

GUI Alternatives

As an alternative to the `undocheckout` function, select **Source Control > Undo Checkout** in the **File** menu of the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser. For more information, see "Undoing the Checkout on UNIX".

Syntax

```
undocheckout('filename')
undocheckout({'filename1','filename2', ...,'filenamen'})
```

Description

`undocheckout('filename')` makes the file `filename` available for checkout, where `filename` does not reflect any of the changes you made after you last checked it out. Use the full pathname for `filename` and include the file extension.

`undocheckout({'filename1','filename2', ...,'filenamen'})` makes `filename1` through `filenamen` available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full pathnames for `filenames` and include the file extensions.

Examples

Typing

```
undocheckout({'/myserver/mymfiles/clock.m', ...
'/myserver/mymfiles/calendar.m'})
```

undoes the checkouts of `/myserver/mymfiles/clock.m` and `/myserver/mymfiles/calendar.m` from the source control system.

See Also

`checkin`, `checkout`

For Windows platforms, use `verctrl`.

unicode2native

Purpose

Convert Unicode characters to numeric bytes

Syntax

```
bytes = unicode2native(unicodestr)
bytes = unicode2native(unicodestr, encoding)
```

Description

`bytes = unicode2native(unicodestr)` takes a `char` vector of Unicode characters, `unicodestr`, converts it to MATLAB's default character encoding scheme, and returns the bytes as a `uint8` vector, `bytes`. Output vector `bytes` has the same general array shape as the `unicodestr` input. You can save the output of `unicode2native` to a file using the `fwrite` function.

`bytes = unicode2native(unicodestr, encoding)` converts the Unicode characters to the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. For common names and aliases, see the Web site http://www.iana.org/assignments/character-sets. If `encoding` is unspecified or is the empty string (`''`), MATLAB's default encoding scheme is used.

Examples

This example begins with two strings containing Unicode characters. It assumes that string `str1` contains text in a Western European language and string `str2` contains Japanese text. The example writes both strings into the same file, using the ISO-8859-1 character encoding scheme for the first string and the Shift-JIS encoding scheme for the second string. The example uses `unicode2native` to convert the two strings to the appropriate encoding schemes.

```
fid = fopen('mixed.txt', 'w');
bytes1 = unicode2native(str1, 'ISO-8859-1');
fwrite(fid, bytes1, 'uint8');
bytes2 = unicode2native(str2, 'Shift_JIS');
fwrite(fid, bytes2, 'uint8');
fclose(fid);
```

See Also

`native2unicode`

Purpose

Find set union of two vectors

Syntax

```
c = union(A, B)
c = union(A, B, 'rows')
[c, ia, ib] = union(...)
```

Description

`c = union(A, B)` returns the combined values from `A` and `B` but with no repetitions. In set theoretic terms, `c = A ∪ B`. Inputs `A` and `B` can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

`c = union(A, B, 'rows')` when `A` and `B` are matrices with the same number of columns returns the combined rows from `A` and `B` with no repetitions.

`[c, ia, ib] = union(...)` also returns index vectors `ia` and `ib` such that `c = a(ia) ∪ b(ib)`, or for row combinations, `c = a(ia,:) ∪ b(ib,:)`. If a value appears in both `a` and `b`, `union` indexes its occurrence in `b`. If a value appears more than once in `b` or in `a` (but not in `b`), `union` indexes the last occurrence of the value.

Remarks

Because NaN is considered to be not equal to itself, every occurrence of NaN in `A` or `B` is also included in the result `c`.

Examples

```
a = [-1 0 2 4 6];
b = [-1 0 1 3];
[c, ia, ib] = union(a, b);
c =

    -1     0     1     2     3     4     6

ia =

     3     4     5

ib =
```

```
     1     2     3     4
```

See Also

intersect, setdiff, setxor, unique, ismember, issorted

Purpose

Find unique elements of vector

Syntax

```
b = unique(A)
b = unique(A, 'rows')
[b, m, n] = unique(...)
[b, m, n] = unique(..., occurrence)
```

Description

`b = unique(A)` returns the same values as in `A` but with no repetitions. `A` can be a numeric or character array or a cell array of strings. If `A` is a vector or an array, `b` is a vector of unique values from `A`. If `A` is a cell array of strings, `b` is a cell vector of unique strings from `A`. The resulting vector `b` is sorted in ascending order and its elements are of the same class as `A`.

`b = unique(A, 'rows')` returns the unique rows of `A`.

`[b, m, n] = unique(...)` also returns index vectors `m` and `n` such that `b = A(m)` and `A = b(n)`. Each element of `m` is the greatest subscript such that `b = A(m)`. For row combinations, `b = A(m,:)` and `A = b(n,:)`.

`[b, m, n] = unique(..., occurrence)`, where `occurrence` can be

- `'first'`, which returns the vector `m` to index the first occurrence of each unique value in `A`, or
- `'last'`, which returns the vector `m` to index the last occurrence.

If you do not specify `occurrence`, it defaults to `'last'`.

You can specify `'rows'` in the same command as `'first'` or `'last'`. The order of appearance in the argument list is not important.

Examples

```
A = [1 1 5 6 2 3 3
9 8 6 2 4] A =    1    1    5    6    2    3    3    9    8    6    2    4
```

Get a sorted vector of unique elements of `A`. Also get indices of the first elements in `A` that make up vector `b`, and the first elements in `b` that make up vector `A`:

```
[b1,
m1, n1] = unique(A, 'first') b1 =    1   2   3   4   5   6   8   9 m1 =
 1   5   6  12   3   4   9   8 n1 =    1   1   5   6   2   3   3   8
 6   2   4
```

Verify that `b1 = A(m1)` and `A = b1(n1)`:

```
all(b1 == A(m1)) && all(A
== b1(n1)) ans =      1
```

Get a sorted vector of unique elements of `A`. Also get indices of the last elements in `A` that make up vector `b`, and the last elements in `b` that make up vector `A`:

```
[b2, m2, n2] =
unique(A, 'last') b2 =    1   2   3   4   5   6   8   9 m2 =    2  11
 12   3  10   9   8 n2 =    1   1   5   6   2   3   3   8   7   6   2
```

Verify that `b2 = A(m2)` and `A = b2(n2)`:

```
all(b2
== A(m2)) && all(A == b2(n2)) ans =      1
```

Because `NaN`s are not equal to each other, `unique` treats them as unique elements.

```
unique([1 1 NaN NaN]) ans =       1 NaN NaN
```

See Also

`intersect`, `ismember`, `issorted`, `setdiff`, `setxor`, `union`

Purpose

Execute UNIX command and return result

Syntax

```
unix command
status = unix('command')
[status, result] = unix('command')
[status,result] = unix('command','-echo')
```

Description

`unix command` calls upon the UNIX operating system to execute the given command.

`status = unix('command')` returns completion status to the `status` variable.

`[status, result] = unix('command')` returns the standard output to the `result` variable, in addition to completion status.

`[status,result] = unix('command','-echo')` displays the results in the Command Window as it executes, and assigns the results to `w`.

Note MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.

Examples

List all users that are currently logged in.

```
[s,w] = unix('who');
```

MATLAB returns `0` (success) in `s` and a string containing the list of users in `w`.

In this example

```
[s,w] = unix('why')
s =
     1
```

```
w =
why: Command not found.
```

MATLAB returns a nonzero value in `s` to indicate failure, and returns an error message in `w` because `why` is not a UNIX command.

See Also

`dos`, `!` (exclamation point), `perl`, `system`

"Running External Programs" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Unload external library from memory

Syntax

```
unloadlibrary('libname')
unloadlibrary libname
```

Description

`unloadlibrary('libname')` unloads the functions defined in shared library `shrlib` from memory. If you need to use these functions again, you must first load them back into memory using `loadlibrary`.

`unloadlibrary libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

Examples

Load the MATLAB sample shared library, `shrlibsample`. Call one of its functions, and then unload the library:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsample shrlibsample.h

s.p1 = 476;   s.p2 = -299;   s.p3 = 1000;
calllib('shrlibsample', 'addStructFields', s)
ans =
    1177

unloadlibrary shrlibsample
```

See Also

`loadlibrary`, `libisloaded`, `libfunctions`, `libfunctionsview`, `libpointer`, `libstruct`, `calllib`

unmkpp

Purpose

Piecewise polynomial details

Syntax

```
[breaks,coefs,l,k,d] = unmkpp(pp)
```

Description

`[breaks,coefs,l,k,d] = unmkpp(pp)` extracts, from the piecewise polynomial `pp`, its breaks `breaks`, coefficients `coefs`, number of pieces `l`, order `k`, and dimension `d` of its target. Create `pp` using `spline` or the spline utility `mkpp`.

Examples

This example creates a description of the quadratic polynomial

$$\frac{-x^2}{4} + x$$

as a piecewise polynomial `pp`, then extracts the details of that description.

```
pp = mkpp([-8 -4],[-1/4 1 0]);
[breaks,coefs,l,k,d] = unmkpp(pp)

breaks =
    -8    -4

coefs =
   -0.2500    1.0000         0

l =
     1

k =
     3

d =
     1
```

See Also

`mkpp`, `ppval`, `spline`

Purpose

Unregister all events for control

Syntax

```
h.unregisterallevents
unregisterallevents(h)
```

Description

`h.unregisterallevents` unregisters all events that have previously been registered with control, `h`. After calling `unregisterallevents`, the control will no longer respond to any events until you register them again using the `registerevent` function.

`unregisterallevents(h)` is an alternate syntax for the same operation.

Examples

mwsamp Control Example

Create an `mwsamp` control, registering three events and their respective handler routines. Use the `eventlisteners` function to see the event handler used by each event:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ...
    [0 0 200 200], f, ...
    {'Click' 'myclick'; 'DblClick' 'my2click'; ...
    'MouseDown' 'mymoused'});

h.eventlisteners
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
    'mousedown'    'mymoused'
```

Unregister all of these events at once with `unregisterallevents`. Now, calling `eventlisteners` returns an empty cell array, indicating that there are no longer any events registered with the control:

```
h.unregisterallevents;
h.eventlisteners
ans =
```

```
    {}
```

To unregister specific events, use the `unregisterevent` function. First, create the control and register three events:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f,...
    {'Click' 'myclick'; 'DblClick' 'my2click'; ...
    'MouseDown' 'mymoused'});
```

Next, unregister two of the three events. The `mousedown` event remains registered:

```
h.unregisterevent({'click' 'myclick'; ...
                   'dblclick' 'my2click'});
h.eventlisteners
ans =
    'mousedown'    'mymoused'
```

Excel Example

Create an Excel Workbook object and register some events.

```
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
wb.registerevent({'Activate' 'EvtActivateHndlr'; ...
                  'Deactivate' 'EvtDeactivateHndlr'})
wb.eventlisteners
```

MATLAB shows the events registered to their corresponding event handlers.

```
ans =

    'Activate'      'EvtActivateHndlr'
    'Deactivate'    'EvtDeactivateHndlr'
```

Use `unregisterallevents` to clear the events.

```
wb.unregisterallevents
wb.eventlisteners
```

MATLAB displays an empty cell array, showing that no events are registered.

```
ans =

     {}
```

See Also

`events`, `eventlisteners`, `registerevent`, `unregisterevent`, `isevent`

unregisterevent

Purpose

Unregister event handler with control's event

Syntax

```
h.unregisterevent(event_handler)
unregisterevent(h, event_handler)
```

Description

`h.unregisterevent(event_handler)` unregisters certain event handler routines with their corresponding events. Once you unregister an event, the control no longer responds to any further occurrences of the event.

`unregisterevent(h, event_handler)` is an alternate syntax for the same operation.

You can unregister events at any time after a control has been created. The `event_handler` argument, which is a cell array, specifies both events and event handlers. For example,

```
h.unregisterevent({'event_name',@event_handler});
```

See "Writing Event Handlers" in the External Interfaces documentation.

You must specify events in the `event_handler` argument using the names of the events. Strings used in the `event_handler` argument are not case sensitive. Unlike `actxcontrol` and `registerevent`, `unregisterevent` does not accept numeric event identifiers.

Examples

Control Example

Create an `mwsamp` control and register all events with the same handler routine, `sampev`. Use `eventlisteners` to see the event handler used by each event. In this case, each event, when fired, calls `sampev.m`:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ...
                [0 0 200 200], f, ...
                'sampev');

h.eventlisteners
ans =
```

```
    'click'        'sampev'
    'dblclick'     'sampev'
    'mousedown'    'sampev'
```

Unregister just the `dblclick` event. Now, when you list the registered events using `eventlisteners`, `dblclick` is no longer registered and the control does not respond when you double-click the mouse over it:

```
h.unregisterevent({'dblclick' 'sampev'});
h.eventlisteners
ans =
    'click'        'sampev'
    'mousedown'    'sampev'
```

This time, register the `click` and `dblclick` events with a different event handler for `myclick` and `my2click`, respectively:

```
h.unregisterallevents;
h.registerevent({'click' 'myclick'; ...
                 'dblclick' 'my2click'});
h.eventlisteners
ans =
    'click'       'myclick'
    'dblclick'    'my2click'
```

You can unregister these same events by specifying event names and their handler routines in a cell array. `eventlisteners` now returns an empty cell array, meaning no events are registered for the `mwsamp` control:

```
h.unregisterevent({'click' 'myclick'; ...
                   'dblclick' 'my2click'});
h.eventlisteners
ans =
     {}
```

In this last example, you could have used `unregisterallevents` instead:

```
h.unregisterallevents;
```

Excel Example

Create an Excel Workbook object

```
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
```

Register two events with the your event handler routines, `EvtActivateHndlr` and `EvtDeactivateHndlr`.

```
wb.registerevent({'Activate' 'EvtActivateHndlr'; ...
                  'Deactivate' 'EvtDeactivateHndlr'})
wb.eventlisteners
```

MATLAB shows the events with the corresponding event handlers.

```
ans =

    'Activate'      'EvtActivateHndlr'
    'Deactivate'    'EvtDeactivateHndlr'
```

Next, unregister the `Deactivate` event handler.

```
wb.unregisterevent({'Deactivate' 'EvtDeactivateHndlr'})
wb.eventlisteners
```

MATLAB shows the remaining registered event (`Activate`) with its corresponding event handler.

```
ans =

    'Activate'    'EvtActivateHndlr'
```

See Also

events, eventlisteners, registerevent, unregisterallevents, isevent

untar

Purpose

Extract contents of tar file

Syntax

```
untar(tarfilename)
untar(tarfilename,outputdir)
untar(url, ...)
filenames = untar(...)
```

Description

`untar(tarfilename)` extracts the archived contents of `tarfilename` into the current directory and sets the files' attributes. It overwrites any existing files with the same names as those in the archive if the existing files' attributes and ownerships permit it. For example, files from rerunning `untar` on the same tar filename do not overwrite any of those files that have a read-only attribute; instead, `untar` issues a warning for such files. On Windows platforms, the hidden, system, and archive attributes are not set.

`tarfilename` is a string specifying the name of the tar file. `tarfilename` is gunzipped to a temporary directory and deleted if its extension ends in `.tgz` or `.gz`. If an extension is omitted, `untar` searches for `tarfilename` appended with `.tgz`, `.tar.gz`, or `.tar` until a file exists. `tarfilename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

`untar(tarfilename,outputdir)` uncompresses the archive `tarfilename` into the directory `outputdir`. `outputdir` is created if it does not exist.

`untar(url, ...)` extracts the tar archive from an Internet URL. The URL must include the protocol type (e.g., `'http://'` or `'ftp://'`). The URL is downloaded to a temporary directory and deleted.

`filenames = untar(...)` extracts the tar archive and returns the relative pathnames of the extracted files into the string cell array `filenames`.

Examples

Copy all `.m` files in the current directory to the directory `backup`:

```
tar('mymfiles.tar.gz','*.m');
untar('mymfiles','backup');
```

Run `untar` to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`:

```
url ='http://www.mathworks.com/moler/ncm.tar.gz';
ncmFiles = untar(url,'ncm')
```

See Also

`gzip`, `gunzip`, `tar`, `unzip`, `zip`

Purpose

Correct phase angles to produce smoother phase plots

Syntax

```
Q = unwrap(P)
Q = unwrap(P,tol)
Q = unwrap(P,[],dim)
Q = unwrap(P,tol,dim)
```

Description

`Q = unwrap(P)` corrects the radian phase angles in a vector `P` by adding multiples of $\pm 2\pi$ when absolute jumps between consecutive elements of `P` are greater than or equal to the default jump tolerance of π radians. If `P` is a matrix, `unwrap` operates columnwise. If `P` is a multidimensional array, `unwrap` operates on the first nonsingleton dimension.

`Q = unwrap(P,tol)` uses a jump tolerance `tol` instead of the default value, π.

`Q = unwrap(P,[],dim)` unwraps along `dim` using the default tolerance.

`Q = unwrap(P,tol,dim)` uses a jump tolerance of `tol`.

Note A jump tolerance less than π has the same effect as a tolerance of π. For a tolerance less than π, if a jump is greater than the tolerance but less than π, adding $\pm 2\pi$ would result in a jump larger than the existing one, so `unwrap` chooses the current point. If you want to eliminate jumps that are less than π, try using a finer grid in the domain.

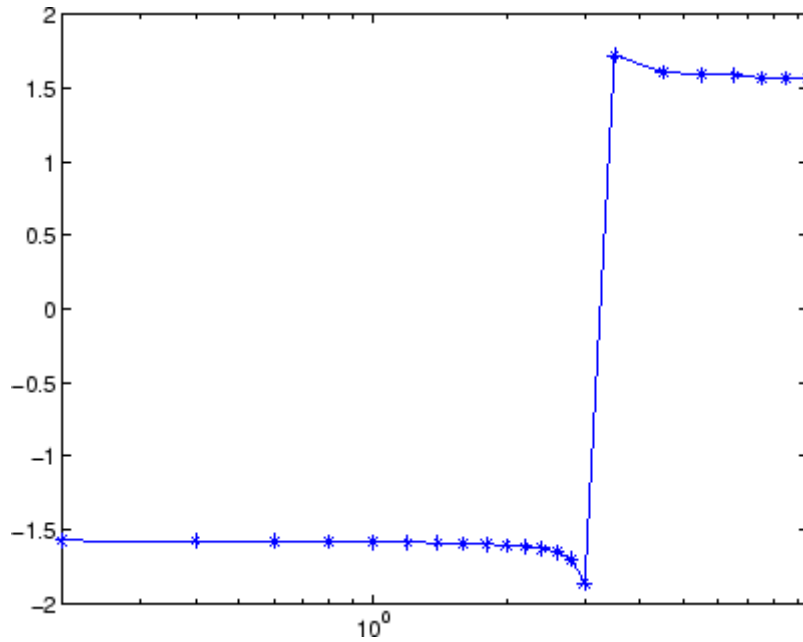
Examples

Example 1

The following phase data comes from the frequency response of a third-order transfer function. The phase curve jumps `3.5873` radians between `w = 3.0` and `w = 3.5`, from `-1.8621` to `1.7252`.

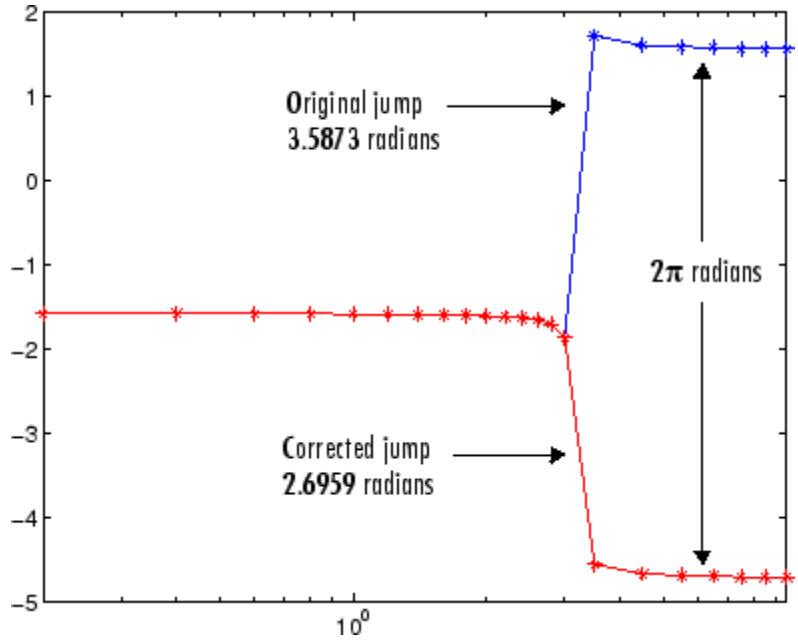
```
w = [0:.2:3,3.5:1:10];
p = [    0
     -1.5728
     -1.5747
     -1.5772
```

```
      -1.5790
      -1.5816
      -1.5852
      -1.5877
      -1.5922
      -1.5976
      -1.6044
      -1.6129
      -1.6269
      -1.6512
      -1.6998
      -1.8621
       1.7252
       1.6124
       1.5930
       1.5916
       1.5708
       1.5708
       1.5708 ];
semilogx(w,p,'b*-'), hold
```



Using `unwrap` to correct the phase angle, the resulting jump is `2.6959`, which is less than the default jump tolerance π. This figure plots the new curve over the original curve.

```
semilogx(w,unwrap(p),'r*-')
```



Note If you have the "Control System Toolbox", you can create the data for this example with the following code.

```
h = freqresp(tf(1,[1 .1 10 0]));
p = angle(h(:));
```

Example 2

Array `P` features smoothly increasing phase angles except for discontinuities at elements `(3,1)` and `(1,2)`.

```
P = [      0    7.0686    1.5708    2.3562
      0.1963    0.9817    1.7671    2.5525
      6.6759    1.1781    1.9635    2.7489
      0.5890    1.3744    2.1598    2.9452 ]
```

The function `Q = unwrap(P)` eliminates these discontinuities.

```
Q =
         0    7.0686    1.5708    2.3562
    0.1963    7.2649    1.7671    2.5525
    0.3927    7.4613    1.9635    2.7489
    0.5890    7.6576    2.1598    2.9452
```

See Also

`abs`, `angle`

Purpose

Extract contents of zip file

Syntax

```
unzip(zipfilename)
unzip(zipfilename,outputdir)
unzip(url, ...)
filenames = unzip(...)
unzip
```

Description

`unzip(zipfilename)` extracts the archived contents of `zipfilename` into the current directory and sets the files' attributes. It overwrites any existing files with the same names as those in the archive if the existing files' attributes and ownerships permit it. For example, files from rerunning `unzip` on the same zip filename do not overwrite any of those files that have a read-only attribute; instead, `unzip` issues a warning for such files.

zipfilename is a string specifying the name of the zip file. The `.zip` extension is appended to zipfilename if omitted. zipfilename can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

`unzip(zipfilename,outputdir)` extracts the contents of zipfilename into the directory `outputdir`.

`unzip(url, ...)` extracts the zipped contents from an Internet URL. The URL must include the protocol type (e.g., `http://`). The URL is downloaded to the temp directory and deleted.

`filenames = unzip(...)` extracts the zip archive and returns the relative pathnames of the extracted files into the string cell array `filenames`.

`unzip` does not support password-protected or encrypted zip archives.

Examples

Example 1

Copy the demos HTML files to the directory `archive`:

```
% Zip the demos html files to demos.zip
zip('demos.zip','*.html',fullfile(matlabroot,'demos'))
```

```
% Unzip demos.zip to the 'directory' archive
unzip('demos','archive')
```

Example 2

Run `unzip` to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`.

```
url ='http://www.mathworks.com/moler/ncm.zip';
ncmFiles = unzip(url,'ncm')
```

See Also

`fileattrib`, `gzip`, `gunzip`, `tar`, `untar`, `zip`

Purpose

Convert string to uppercase

Syntax

```
t  = upper('str')
B = upper(A)
```

Description

`t = upper('str')` converts any lowercase characters in the string `str` to the corresponding uppercase characters and leaves all other characters unchanged.

`B = upper(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `upper` to each string within `A`.

Examples

`upper('attention!')` is `ATTENTION!`.

Remarks

Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`lower`

urlread

Purpose

Read content at URL

Syntax

```
s = urlread('url')
s = urlread('url','method','params')
[s,status] = urlread(...)
```

Description

`s = urlread('url')` reads the content at a URL into the string `s`. If the server returns binary data, `s` will be unreadable.

`s = urlread('url','method','params')` reads the content at a URL into the string `s`, passing information to the server as part of the request where *`method`* can be `get` or `post`, and `params` is a cell array of parameter name/parameter value pairs.

`[s,status] = urlread(...)` catches any errors and returns the error code.

Note If you need to specify a proxy server to connect to the Internet, select **File -> Preferences -> Web** and enter your proxy server address and port. Use this feature if you have a firewall.

Examples

Download Content from Web Page

Use `urlread` to download the contents of the Authors list at the MATLAB Central File Exchange:

```
urlstring = sprintf('%s%s', ...
   'http://www.mathworks.com/matlabcentral/', ...
   'fileexchange/loadAuthorIndex.do');

s = urlread(urlstring);
```

Download Content from File on FTP Server

```
page = 'ftp://ftp.mathworks.com/pub/doc/';
s=urlread(page);
```

```
s
```

MATLAB displays

```
s =

-rw-r--r--   1 ftpuser  ftpusers       448 Nov 15  2004 README
drwxr-xr-x   2 ftpuser  ftpusers       512 Jul 26 13:52 papers
```

Download Content from Local File

```
s = urlread('file:///c:/winnt/matlab.ini')
```

See Also

`urlwrite`

`tcpip` if the Instrument Control Toolbox is installed

Purpose

Save contents of URL to file

Syntax

```
urlwrite('url','filename')
f = urlwrite('url','filename')
f = urlwrite('url','method','params')
[f,status] = urlwrite(...)
```

Description

`urlwrite('url','filename')` reads the contents of the specified URL, saving the contents to `filename`. If you do not specify the path for `filename`, the file is saved in the MATLAB current directory.

`f = urlwrite('url','filename')` reads the contents of the specified URL, saving the contents to `filename` and assigning `filename` to `f`.

`f = urlwrite('url','method','params')` saves the contents of the specified URL to `filename`, passing information to the server as part of the request where *`method`* can be `get` or `post`, and `params` is a cell array of parameter name/parameter value pairs.

`[f,status] = urlwrite(...)` catches any errors and returns the error code.

Note If you need to specify a proxy server to connect to the Internet, select **File -> Preferences -> Web** and enter your proxy server address and port. Use this feature if you have a firewall.

Examples

Download the files submitted to the MATLAB Central File Exchange, saving the results to `samples.html` in the MATLAB current directory.

```
urlstring = sprintf('%s%s', ...
'http://www.mathworks.com/matlabcentral/', ...
'fileexchange/Category.jsp?type=category&id=1');

urlwrite(urlstring, 'samples.html');
```

View the file in the Help browser.

```
open('samples.html')
```

See Also

urlread

Purpose

Determine whether Java feature is supported in MATLAB

Syntax

```
usejava(feature)
```

Description

`usejava(feature)` returns 1 if the specified feature is supported and 0 otherwise. Possible `feature` arguments are shown in the following table.

Feature	Description
`'awt'`	Abstract Window Toolkit components[1] are available
`'desktop'`	The MATLAB interactive desktop is running
`'jvm'`	The Java Virtual Machine is running
`'swing'`	Swing components[2] are available

1. Java's GUI components in the Abstract Window Toolkit
2. Java's lightweight GUI components in the Java Foundation Classes

Examples

The following conditional code ensures that the AWT's GUI components are available before the M-file attempts to display a Java Frame.

```
if usejava('awt')
   myFrame = java.awt.Frame;
else
   disp('Unable to open a Java Frame');
end
```

The next example is part of an M-file that includes Java code. It fails gracefully when run in a MATLAB session that does not have access to a JVM.

```
if ~usejava('jvm')
   error([mfilename ' requires Java to run.']);
end
```

See Also

javachk

validateattributes

Purpose

Check validity of array

Syntax

```
validateattributes(A, classes, attributes)
validateattributes(A, classes, attributes, position)
validateattributes(A, classes, attributes, funname)
validateattributes(A, classes, attributes, funname, varname)
validateattributes(A, classes, attributes, funname, varname,
   position)
```

Description

`validateattributes(A, classes, attributes)` validates that array `A` belongs to at least one of the classes specified by the `classes` input and also has at least one of the attributes specified by the `attributes` input. If the validation succeeds, the command completes without displaying any output and without throwing an error. If the validation does not succeed, MATLAB issues a formatted error message.

The `classes` input is a cell array of one or more strings, each string containing the name of a MATLAB class (i.e., one of the 15 MATLAB data types), the name of a MATLAB class, or the keyword `numeric`. (See the Class Values on page 2-3633 table, below.

The `attributes` input is a cell array of one or more strings, each string describing an array attribute. (See the Attribute Values on page 2-3634 table, below).

`validateattributes(A, classes, attributes, position)` validates array `A` as described above and, if the validation fails, displays an error message that includes the position of the failing variable in the function argument list. The `position` input must be a positive integer.

`validateattributes(A, classes, attributes, funname)` validates array `A` as described above and, if the validation fails, displays an error message that includes the name of the function performing the validation (`funname`). The `funname` input must be a string enclosed in single quotation marks.

`validateattributes(A, classes, attributes, funname, varname)` validates array `A` as described above and, if the validation fails, displays an error message that includes the name of the function

performing the validation (`funname`), and the name of the variable being validated (`varname`). The `funname` and `varname` inputs must be strings enclosed in single quotation marks.

`validateattributes(A, classes, attributes, funname, varname, position)` validates array `A` as described above and, if the validation fails, displays an error message that includes the name of the function performing the validation (`funname`), the name of the variable being validated (`varname`), and the position of this variable in the function argument list (`position`). The `funname` and `varname` inputs must be strings enclosed in single quotation marks. The `position` input must be a positive integer.

Class Values

classes Argument	Contents of Array A
`'numeric'`	Any numeric value
`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `true` or `false`
`'char'`	Character or string
`'struct'`	MATLAB structure

Class Values (Continued)

classes Argument	Contents of Array A
'cell'	Cell array
'function_handle'	Scalar function handle
class name	Object of any MATLAB class

Attribute Values

attributes Argument	Description of array A
'2d'	Array having dimensions M-by-N (includes scalars, vectors, 2-D matrices, and empty arrays)
'column'	Array having dimensions N-by-1
'even'	Numeric or logical array in which all elements are even (includes zero)
'finite'	Numeric array in which all elements are finite
'integer'	Numeric array in which all elements are integer-valued
'nonempty'	Array having no dimension equal to zero
'nonnan'	Numeric array in which there are no elements equal to NaN (Not a Number)
'nonnegative'	Numeric array in which all elements are zero or greater than zero
'nonsparse'	Array that is not sparse
'nonzero'	Numeric or logical array in which all elements are less than or greater than zero
'odd'	Numeric or logical array in which all elements are odd integers

Attribute Values (Continued)

attributes Argument	Description of array A
'positive'	Numeric or logical array in which all elements are greater than zero
'real'	Numeric array in which all elements are real
'row'	Array having dimensions 1-by-N
'scalar'	Array having dimensions 1-by-1
'vector'	Array having dimensions N-by-1 or 1-by-N (includes scalar arrays)

Numeric properties, such as `positive` and `nonnan`, do not apply to strings. If you attempt to validate numeric properties on a string, `validateattributes` generates an error.

Examples

Example 1

This function, which resides in M-file `empl_profile`, compares the values passed in each argument with the specified classes and attributes and throws an error if they are not correct:

```
function empl_profile(empl_id, empl_info, healthplan, ...
   vacation)
validateattributes(empl_id, {'numeric'}, {'integer', ...
   'nonempty'});
validateattributes(empl_info, {'struct'}, {'vector'});
validateattributes(healthplan, {'cell', 'char'}, {'vector'});
validateattributes(vacation, {'numeric'}, {'nonnegative', ...
   'scalar'});
```

Call the function, passing the expected argument types, and the example completes without error:

```
empl_id = 51723;
```

```
empl_info.name = 'John Miller';
empl_info.address = '128 Forsythe St.';
empl_info.town = 'Duluth';  empl_info.state='MN';

empl_profile(empl_id, empl_info, 'HCP Medical Plus', 14.3)
```

If you accidentally pass the argument values out of their correct sequence, MATLAB throws an error in response to the first argument that is not a match:

```
empl_profile(empl_id, empl_info, 14.3, 'HCP Medical Plus')

??? Error using ==> empl_profile1 at 4
Expected input to be one of these types:

  cell, char

Instead its type was double.
```

Example 2

Modify the `empl_profile` M-file shown in the last example, adding arguments to `validateattributes` to display the function name, variable name, and position of the argument:

```
function empl_profile(empl_id, empl_info, healthplan, ...
   vacation)
validateattributes(empl_id, {'numeric'}, {'integer', ...
   'nonempty'}, mfilename, 'Employee Identification', 1);
validateattributes(empl_info, {'struct'}, {'vector'}, ...
   mfilename, 'Employee Info', 2);
validateattributes(healthplan, {'cell', 'char'}, ...
   {'vector'}, mfilename, 'Health Plan', 3);
validateattributes(vacation, {'numeric'}, {'nonnegative', ...
   'scalar'}, mfilename, 'Vacation Accrued', 4);
```

Call `empl_profile` with the argument values out of their correct sequence, MATLAB throws an error that includes the name of the function validating the attributes, the name of the variable that was in error, and it position in the input argument list:

```
??? Error using ==> empl_profile
Expected input number 3, Health Plan, to be one of these types:

  cell, char

Instead its type was double.

Error in ==> empl_profile at 6
validateattributes(healthplan,{'cell', 'char'}, {'vector'}, ...
```

Example 3

Modify the `empl_profile` M-file so that it checks the function inputs using the MATLAB `inputParser`. Use `validateattributes` as the validating function for the `inputParser` methods:

```
function empl_profile(empl_id, varargin)
p = inputParser;

% Validate the input arguments.
addRequired(p, 'empl_id', @(x)validateattributes(x, ...
   {'numeric'}, {'integer'}));
addOptional(p, 'empl_info', '', @(x)validateattributes(...
   x, {'struct'}, {'nonempty'}));
addParamValue(p, 'health', 'HCP Medical Plus', ...
   @(x)validateattributes(x, {'cell', 'char'}, {'vector'}));
addParamValue(p, 'vacation', [], @(x)validateattributes(x, ...
   {'numeric'}, {'nonnegative', 'scalar'}));
parse(p, empl_id, varargin{:});
p.Results
```

Call `empl_profile` using appropriate input arguments:

```
empl_info.name = 'John Miller';
empl_info.address = '128 Forsythe St.';
empl_info.town = 'Duluth';  empl_info.state='MN';

empl_profile(51723, empl_info, 'vacation', 14.3)

ans =
     empl_id: 51723
   empl_info: [1x1 struct]
      health: 'HCP Medical Plus'
    vacation: 14.3000
```

Call `empl_profile` using a character string where a structure is expected:

```
empl_profile(51723, empl_info.name,  'vacation', 14.3)

??? Error using ==> empl_profile at 12
Argument 'empl_info' failed validation with error:
Expected input to be one of these types:

  struct

Instead its type was char.
```

See Also

`validatestring`, `is*`, `isa`, `inputparser`

Purpose

Check validity of text string

Syntax

```
validstr = validatestring(str,strarray)
validstr = validatestring(str, strarray, position)
validstr = validatestring(str, strarray, funname)
validstr = validatestring(str, strarray, funname, varname)
validstr = validatestring(str, strarray, funname, varname,
   position)
```

Description

`validstr = validatestring(str,strarray)` checks the validity of text string `str`. If `str` matches one or more of the text strings in the cell array `strarray`, then MATLAB returns the matching string in `validstr`. If `str` does not match any of the strings in `strarray`, MATLAB issues a formatted error message. MATLAB compares the strings without respect to letter case.

This table shows how `validatestring` determines what value to return, If multiple matches are found, `validatestring` returns the shortest matching string:

Type of Match	Example — Match `'ball'` with . . .	Return Value
Exact match	`ball`, `barn`, `bell`	`ball`
Partial match (leading characters)	`balloon`, `barn`	`balloon`
Multiple partial matches where each string is a subset of another	`ball`, `ballo`, `balloo`, `balloon`	`ball`
Multiple partial matches where strings are unique	`balloon`, `ballet`	Error
No match	`barn`, `bell`	Error

`validstr = validatestring(str, strarray, position)` checks the validity of text string `str` as described above and, if the validation fails, displays an error message that includes the position of the failing

variable in the function argument list. The `position` input must be a positive integer.

`validstr = validatestring(str, strarray, funname)` checks the validity of text string `str` as described above and, if the validation fails, displays an error message that includes the name of the function performing the validation (`funname`). The `funname` input must be a string enclosed in single quotation marks.

`validstr = validatestring(str, strarray, funname, varname)` checks the validity of text string `str` as described above and, if the validation fails, displays an error message that includes the name of the function performing the validation (`funname`), and the name of the variable being validated (`varname`). The `funname` and `varname` inputs must be strings enclosed in single quotation marks.

`validstr = validatestring(str, strarray, funname, varname, position)` checks the validity of text string `str` as described above and, if the validation fails, displays an error message that includes the name of the function performing the validation (`funname`), the name of the variable being validated (`varname`), and the position of this variable in the function argument list (`position`). The `funname` and `varname` inputs must be strings enclosed in single quotation marks. The `position` input must be a positive integer.

Examples

Example 1

Use `validatestring` to find the word `won` in the cell array of strings:

```
validatestring('won', {'wind', 'won', 'when'})
ans =
   won
```

Replace the word `won` with `wonder` in the string array. Because the leading characters of the input string and `wonder` are the same, `validatestring` finds a partial match between the two words and returns the full word `wonder`:

```
validatestring('won', {'wind', 'wonder', 'when'})
```

```
ans =
   wonder
```

If there is more than one partial match, and each string in the array is a subset or superset of the others, `validatestring` returns the shortest matching string:

```
validatestring('wond', {'won', 'wonder', 'wonderful'})
ans =
   wonder
```

However, if each string in the array is not subset or superset of each other, MATLAB throws an error because there is no exact match and it is not clear which of the two partial matches should be returned:

```
validatestring('wond', {'won', 'wonder', 'wondrous'})
??? Error using ==> validatestring at 89
Function VALIDATESTRING expected its input argument to match one of

  won, wonder, wondrous

The input, 'wond', matched more than one valid string.
```

Example 2

This function returns the flight numbers for routes between two cities: a point of origin and point of destination. The function uses `validatestring` to see if the origin and destination are among those covered by the airline. If not, then an error message is displayed:

```
function get_flight_numbers(origin, destination)
% Only part of the airline's flight data is shown here.
   flights.chi2rio = [503, 196, 331, 373, 1475];
   flights.chi2par = [718, 9276, 172, 903, 7724 992, 1158];
   flights.chi2hon = [9193, 880, 471, 391];

   routes = {'Athens', 'Paris', 'Chicago', 'Sydney', ...
      'Cancun', 'London', 'Rio de Janeiro', 'Honolulu', ...
      'Rome', 'New York City'};
```

```
orig = '';  dest = '';

   % See if the cities entered are covered by this airline.
   try
      orig = validatestring(origin, routes);
      dest = validatestring(destination, routes);
   catch
      % If not covered, then display error message.
      if isempty(orig)
         fprintf(...
            'We have no flights with origin: %s.\n', ...
            origin)
      elseif isempty(dest)
         fprintf(...
            'We have no flights with destination: %s.\n', ...
            destination)
      end
   return
   end

% If covered, display the flights from 'orig' to 'dest'.
fprintf(...
   'Flights available from %s to %s are:\n', orig, dest)
reply = eval(...
  ['flights.' lower(orig(1:3)) '2' lower(dest(1:3))])';
fprintf('   Flight %d\n', reply)
```

Enter a point of origin that is not covered by this airline:

```
get_flight_numbers('San Diego', 'Rio de Janeiro')
ans =
We have no flights with origin: San Diego.
```

Enter a destination that is misspelled:

```
get_flight_numbers('Chicago', 'Reo de Janeiro')
ans =
```

```
We have no flights with destination: Reo de Janeiro.
```

Enter a route that is covered:

```
get_flight_numbers('Chicago', 'Rio de Janeiro')
ans =
Flights available from Chicago to Rio de Janeiro are:
   Flight 503
   Flight 196
   Flight 331
   Flight 373
   Flight 1475
```

Example 3

Rewrite the `try-catch` block of Example 2, above by adding `funname`, `varname`, and `position` arguments to the call to `validatestring` and replacing the `return` statement with `rethrow`:

```
% See if the cities entered are covered by this airline.
try
   orig = validatestring(...
      origin, routes, mfilename, 'Flight Origin', 1);
   dest = validatestring(...
      destination, routes, mfilename, ...
        'Flight Destination', 2);
catch e
   % If not covered, then display error message.
   if isempty(orig)
       fprintf(...
          'We have no flights with origin: %s.\n', ...
          origin)
   elseif isempty(dest)
       fprintf(...
          'We have no flights with destination: %s.\n', ...
          destination)
   end
   rethrow(e);
```

```
    end
```

In response to the `rethrow` command, MATLAB displays an error message that includes the function name `get_flight_numbers`, the failing variable name `Flight Destination`', and its position in the argument list, 2:

```
get_flight_numbers('Chicago', 'Reo de Janeiro')
We have no flights with destination: Reo de Janeiro.

??? Error using ==> validatestring at 89
Function GET_FLIGHT_NUMBERS expected its input argument
    number 2, Flight Destination, to match one of these strings:

  Athens, Paris, Chicago, Sydney, Cancun, London, Rio de
Janeiro, Honolulu, Rome

The input, 'Reo de Janeiro', did not match any of the valid strings.

Error in ==> get_flight_numbers at 17
   dest = validatestring(destination, routes, mfilename,
   'destination', 2);
```

See Also

`validateattributes`, `is*`, `isa`, `inputparser`

Purpose

Vandermonde matrix

Syntax

```
A = vander(v)
```

Description

`A = vander(v)` returns the Vandermonde matrix whose columns are powers of the vector `v`, that is, `A(i,j) = v(i)^(n-j)`, where `n = length(v)`.

Examples

```
vander(1:.5:3)

ans =

    1.0000    1.0000    1.0000    1.0000    1.0000
    5.0625    3.3750    2.2500    1.5000    1.0000
   16.0000    8.0000    4.0000    2.0000    1.0000
   39.0625   15.6250    6.2500    2.5000    1.0000
   81.0000   27.0000    9.0000    3.0000    1.0000
```

See Also

`gallery`

Purpose

Variance

Syntax

```
V = var(X)
V = var(X,1)
V = var(X,w)
V = var(X,w,dim)
```

Description

`V = var(X)` returns the variance of `X` for vectors. For matrices, `var(X)`is a row vector containing the variance of each column of `X`. For N-dimensional arrays, var operates along the first nonsingleton dimension of X. The result `V` is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples.

`var` normalizes `V` by `N-1` if `N>1`, where `N` is the sample size. This is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples. For `N=1`, `V` is normalized by `N`.

`V = var(X,1)` normalizes by N and produces the second moment of the sample about its mean.`var(X,0) is equivalent to var(X).`

`V = var(X,w)` computes the variance using the weight vector `w`. The length of `w` must equal the length of the dimension over which `var` operates, and its elements must be nonnegative. The elements of `w` must be positive. `var` normalizes `w` to sum of 1.

`V = var(X,w,dim)` takes the variance along the dimension `dim` of `X`. Pass in `0` for `w` to use the default normalization by N-1, or 1 to use N.

The variance is the square of the standard deviation (STD).

See Also

`corrcoef`, `cov`, `mean`, `median`, `std`

Purpose

Variance of `timeseries` data

Syntax

```
ts_var = var(ts)
ts_var = var(ts,'PropertyName1',PropertyValue1,...)
```

Description

`ts_var = var(ts)` returns the variance of `ts.data`. When `ts.Data` is a vector, ts_var is the variance of `ts.Data` values. When `ts.Data` is a matrix, ts_var is a row vector containing the variance of each column of `ts.Data` (when `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `var` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_var = var(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, `'remove'` (default) or `'interpolate'`, indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by an integer vector, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, `'none'` (default) or `'time'`. When you specify `'time'`, larger time values correspond to larger weights.

Examples

The following example shows how to calculate the variance values of a multi-variate `timeseries` object.

1 Load a 24-by-3 data array.

```
load count.dat
```

2 Create a `timeseries` object with 24 time values.

```
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')
```

3 Calculate the variance of each data column for this `timeseries` object.

```
var(count_ts)
ans =

  1.0e+003 *

    0.6437    1.7144    4.6278
```

The variance is calculated independently for each data column in the `timeseries` object.

See Also

`iqr (timeseries)`, `mean (timeseries)`, `median (timeseries)`, `std (timeseries)`, `timeseries`

Purpose

Variable length input argument list

Syntax

```
function y = bar(varargin)
```

Description

`function y = bar(varargin)` accepts a variable number of arguments into function `bar.m`.

The `varargin` statement is used only inside a function M-file to contain optional input arguments passed to the function. The `varargin` argument must be declared as the last input argument to a function, collecting all the inputs from that point onwards. In the declaration, `varargin` must be lowercase.

Examples

Example 1

Write an M-file function that displays the expected and optional arguments you pass to it

```
function vartest(argA, argB, varargin)

optargin = size(varargin,2);
stdargin = nargin - optargin;

fprintf('Number of inputs = %d\n', nargin)

fprintf('  Inputs from individual arguments(%d):\n', stdargin)
if stdargin >= 1
    fprintf('     %d\n', argA)
end
if stdargin == 2
    fprintf('     %d\n', argB)
end

fprintf('  Inputs packaged in varargin(%d):\n', optargin)
 for k= 1 : size(varargin,2)
     fprintf('     %d\n', varargin{k})
 end
```

Call this function and observe that MATLAB extracts those arguments that are not individually-specified from the `varargin` cell array:

```
vartest(10,20,30,40,50,60,70)
Number of inputs = 7
  Inputs from individual arguments(2):
     10
     20
  Inputs packaged in varargin(5):
     30
     40
     50
     60
     70
```

Example 2

The function

```
function myplot(x,varargin)
plot(x,varargin{:})
```

collects all the inputs starting with the second input into the variable `varargin`. `myplot` uses the comma-separated list syntax `varargin{:}` to pass the optional parameters to `plot`. The call

```
myplot(sin(0:.1:1),'color',[.5 .7 .3],'linestyle',':')
```

results in `varargin` being a 1-by-4 cell array containing the values `'color'`, `[.5 .7 .3]`, `'linestyle'`, and `':'`.

See Also

`varargout`, `nargin`, `nargout`, `nargchk`, `nargoutchk`, `inputname`

Purpose

Variable length output argument list

Syntax

```
function varargout = foo(n)
```

Description

`function varargout = foo(n)` returns a variable number of arguments from function `foo.m`.

The `varargout` statement is used only inside a function M-file to contain the optional output arguments returned by the function. The `varargout` argument must be declared as the last output argument to a function, collecting all the outputs from that point onwards. In the declaration, `varargout` must be lowercase.

Examples

The function

```
function [s,varargout] = mysize(x)
nout = max(nargout,1)-1;
s = size(x);
for k=1:nout, varargout(k) = {s(k)}; end
```

returns the size vector and, optionally, individual sizes. So

```
[s,rows,cols] = mysize(rand(4,5));
```

returns `s = [4 5], rows = 4, cols = 5`.

See Also

`varargin`, `nargin`, `nargout`, `nargchk`, `nargoutchk`, `inputname`

vectorize

Purpose

Vectorize expression

Syntax

```
vectorize(s)
vectorize(fun)
```

Description

`vectorize(s)` where `s` is a string expression, inserts a `.` before any `^`, `*` or `/` in `s`. The result is a character string.

`vectorize(fun)` when `fun` is an inline function object, vectorizes the formula for `fun`. The result is the vectorized version of the inline function.

See Also

`inline`, `cd`, `dbtype`, `delete`, `dir`, `partialpath`, `path`, `what`, `who`

Purpose

Version information for MathWorks products

Graphical Interface

As an alternative to the `ver` function, select **About** from the **Help** menu in any product that has a **Help** menu.

Syntax

```
ver
ver product
v = ver('product')
```

Description

`ver` displays a header containing the current version number, license number, operating system, and Java VM version for MATLAB, followed by the version numbers for Simulink, if installed, and all other MathWorks products installed.

`ver product` displays the MATLAB header information followed by the current version number for `product`. The name `product` corresponds to the directory name that holds the `Contents.m` file for that product. For example, `Contents.m` for the Control System Toolbox resides in the `control` directory. You therefore use `ver control` to obtain the version of this toolbox.

`v = ver('product')` returns the version information to structure array, `v`, having fields `Name`, `Version`, `Release`, and `Date`.

Remarks

To use `ver` with your own product, the first two lines of the `Contents.m` file for the product must be of the form

```
% Toolbox Description
% Version xxx dd-mmm-yyyy
```

Do not include any spaces in the date and use a two-character day; that is, use `02-Sep-2002` instead of `2-Sep-2002`.

Examples

Return version information for the Control System Toolbox by typing

```
ver control
```

MATLAB returns

```
--------------------------------------------------------------------------------
MATLAB Version 7.3.0.22078 (R2006b)
MATLAB License Number: unknown
Operating System: Microsoft Windows XP Version 5.1 (Build 2600: Service Pack 2)
Java VM Version: Java 1.5.0_07 with Sun Microsystems Inc. Java HotSpot(TM) Client VM mixed
--------------------------------------------------------------------------------
Control System Toolbox                                Version 7.1        (R2006b)
```

Return version information for the Control System Toolbox in a structure array, v.

```
v = ver('control')
v =

       Name: 'Control System Toolbox'
    Version: '7.1'
    Release: '(R2006b)'
       Date: '19-Sep-2006'
```

Display version information on MathWorks 'Real-Time' products:

```
v = ver;
for k=1:length(v)
   if strfind(v(k).Name, 'Real-Time')
      disp(sprintf('%s, Version %s', ...
                   v(k).Name, v(k).Version))
   end
end

Real-Time Windows Target, Version 2.6.2
Real-Time Workshop, Version 6.5
Real-Time Workshop Embedded Coder, Version 4.5
```

See Also

`help`, `hostid`, `license`, `version`, `whatsnew`

Help > Check for Updates in the MATLAB desktop.

Purpose

Source control actions (Windows)

GUI Alternatives

As an alternative to the `verctrl` function, use **Source Control** in the **File** menu of the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser.

Syntax

```
verctrl('action',{'filename1','filename2',....},0)
result=verctrl('action',{'filename1','filename2',....},0)
verctrl('action','filename',0)
result=verctrl('isdiff','filename',0)
list = verctrl('all_systems')
```

Description

`verctrl('action',{'filename1','filename2',....},0)` performs the source control operation specified by `'action'` for a single file or multiple files. Enter one file as a string; specify multiple files using a cell array of strings. Use the full paths for each filename and include the extensions. Specify `0` as the last argument. Complete the resulting dialog box to execute the operation; for details about the dialog boxes, see the topic “Source Control Interface on Windows” in the MATLAB Desktop Tools and Development Environment documentation. Available values for `'action'` are as follows:

action Argument	Purpose
`'add'`	Adds files to the source control system. Files can be open in the Editor/Debugger or closed when added.
`'checkin'`	Checks files into the source control system, storing the changes and creating a new version.
`'checkout'`	Retrieves files for editing.
`'get'`	Retrieves files for viewing and compiling, but not editing. When you open the files, they are labeled as read-only.
`'history'`	Displays the history of files.

action Argument	Purpose
`'remove'`	Removes files from the source control system. It does not delete the files from disk, but only from the source control system.
`'runscc'`	Starts the source control system. The filename can be an empty string.
`'uncheckout'`	Cancels a previous checkout operation and restores the contents of the selected files to the precheckout version. All changes made to the files since the checkout are lost.

`result=verctrl('action',{'filename1','filename2',....},0)` performs the source control operation specified by `'action'` on a single file or multiple files. The action can be any one of: `'add'`, `'checkin'`, `'checkout'`, `'get'`, `'history'`, or `'undocheckout'`. `result` is a logical `1` (true) when you complete the operation by clicking **OK** in the resulting dialog box, and is a logical `0` (false) when you abort the operation by clicking **Cancel** in the resulting dialog box.

`verctrl('action','filename',0)` performs the source control operation specified by `'action'` for a single file. Use the full pathname for `'filename'`. Specify `0` as the last argument. Complete any resulting dialog boxes to execute the operation. Available values for `'action'` are as follows:

action Argument	Purpose
`'showdiff'`	Displays the differences between a file and the latest checked in version of the file in the source control system.
`'properties'`	Displays the properties of a file.

`result=verctrl('isdiff','filename',0)` compares `filename` with the latest checked in version of the file in the source control system. `result` is a logical `1` (true) when the files are different, and is a logical `0` (false) when the files are identical. Use the full path for `'filename'`. Specify `0` as the last argument.

`list = verctrl('all_systems')` displays in the Command Window a list of all source control systems installed on your computer.

Examples

Check In a File

Check in `D:\file1.ext` to the source control system.

```
result = verctrl('checkin','D:\file1.ext', 0)
```

This opens the **Check in file(s)** dialog box. Click **OK** to complete the check in. MATLAB displays `result = 1`, indicating the checkin was successful.

Add Files to the Source Control System

Add `D:\file1.ext` and `D:\file2.ext` to the source control system.

```
verctrl('add',{'D:\file1.ext','D:\file2.ext'}, 0)
```

This opens the **Add to source control** dialog box. Click **OK** to complete the operation.

Display the Properties of a File

Display the properties of `D:\file1.ext`.

```
verctrl('properties','D:\file1.ext', 0)
```

This opens the source control properties dialog box for your source control system. The function is complete when you close the properties dialog box.

Show Differences for a File

To show the differences between the version of `file1.ext` that you just edited and saved, with the last version in source control, run

```
verctrl('showdiff','D:\file1.ext',0)
```

MATLAB displays differences dialog boxes and results specific to your source control system. After checking in the file, if you run this statement again, MATLAB displays

```
??? The file is identical to latest version under source control.
```

List All Installed Source Control Systems

To view all of the source control systems installed on your computer, type

```
list = verctrl ('all_systems')
```

MATLAB displays all the source control systems currently installed on your computer. For example:

```
list =
'Microsoft Visual SourceSafe'
'ComponentSoftware RCS'
```

See Also

`checkin`, `checkout`, `undocheckout`, `cmopts`

"Source Control Interface on Windows" in MATLAB Desktop Tools and Development Environment documentation

Purpose

Compare toolbox version to specified version string

Syntax

```
verLessThan(toolbox, version)
```

Description

`verLessThan(toolbox, version)` returns logical 1 (`true`) if the version of the toolbox specified by the string `toolbox` is older than the version specified by the string `version`, and logical 0 (`false`) otherwise. Use this function when you want to write code that can run across multiple versions of MATLAB.

The `toolbox` argument is a string enclosed within single quotation marks that contains the name of a MATLAB toolbox directory. The `version` argument is a string enclosed within single quotation marks that contains the version to compare against. This argument must be in the form `major[.minor[.revision]]`, such as `7`, `7.1`, or `7.0.1`. If `toolbox` does not exist, MATLAB generates an error.

To specify `toolbox`, find the directory that holds the `Contents.m` file for the desired toolbox and use that directory name. To see a list of all toolbox directory names, enter the following command at the MATLAB prompt:

```
dir([matlabroot '/toolbox'])
```

Remarks

The `verLessThan` function is available with MATLAB Version 7.4. If you are running a version of MATLAB earlier than 7.4, you can download the `verLessThan` M-file from the following MathWorks Technical Support solution. You must be running MATLAB Version 6.0 or higher to use this M-file:

`http://www.mathworks.com/support/solutions/data/1-38LI61.html?solution`

Examples

These examples illustrate the proper usage of the `verLessThan` function.

Example 1 – Checking For the Minimum Required Version

```
if verLessThan('simulink', '4.0')
    error('Simulink 4.0 or higher is required.');
end
```

Example 2 – Choosing Which Code to Run

```
if verLessThan('matlab', '7.0.1')
% -- Put code to run under MATLAB 7.0.0 and earlier here --
else
% -- Put code to run under MATLAB 7.0.1 and later here --
end
```

Example 3 – Looking Up the Directory Name

Find the name of the Data Acquisition Toolbox directory:

```
dir([matlabroot '/toolbox/d*'])

   daq        database   des      distcomp   dotnetbuilder
   dastudio   datafeed   dials    dml        dspblks
```

Use the toolbox directory name, `daq`, to compare the Data Acquisition version that MATLAB is currently running against version number `3`:

```
verLessThan('daq', '3')
ans =
     1
```

See Also

`ver`, `version`, `license`, `ispc`, `isunix`, `ismac`, `dir`

Purpose

Version number for MATLAB

Graphical Interface

As an alternative to the `version` function, select **About** from the **Help** menu in the MATLAB desktop.

Syntax

```
version
v = version
[v d] = version
version option
v = version('option')
```

Description

`version` displays the MATLAB version number.

`v = version` returns the MATLAB version number in `v`.

`[v d] = version` also returns a string `d` containing the date of the version.

`version option` displays the following additional information about the version.

Option	Description
`-date`	Release date
`-description`	Release description. Mostly used for Service Pack releases.
`-java`	Java VM (JVM) version used by MATLAB
`-release`	Release number

`v = version('option')` returns additional information about the version. Valid string values for `option` are listed in the table above. You can only specify one output when using this syntax.

Remarks

On Windows and UNIX platforms, MATLAB includes a JVM and uses that version. If you use the MATLAB Java interface and the Java classes you want to use require a different JVM than the version provided with MATLAB, it is possible to run MATLAB with a different

JVM. For details, see Solution 1-1812J on the MathWorks Support Web site.

On the Macintosh platform, MATLAB does not include a JVM, but uses whatever JVM is currently running on the machine.

Examples

```
[v,d] = version
v =
   7.3.0.22078 (R2006b)

d =
   September 19, 2006
```

Run the following command in MATLAB R14 Service Pack 3:

```
['Release R' version('-release') ', '     ...
             version('-description')]

ans =
   Release R14, Service Pack 3
```

See Also

`ver`, `whatsnew`

Help > Check for Updates in the MATLAB desktop.

Purpose

Concatenate arrays vertically

Syntax

```
C = vertcat(A1, A2, ...)
```

Description

`C = vertcat(A1, A2, ...)` vertically concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of columns.

`vertcat` concatenates N-dimensional arrays along the first dimension. The remaining dimensions must match.

MATLAB calls `C = vertcat(A1, A2, ...)` for the syntax `C = [A1; A2; ...]` when any of `A1`, `A2`, etc. is an object.

Examples

Create a 5-by-3 matrix, `A`, and a 3-by-3 matrix, `B`. Then vertically concatenate `A` and `B`.

```
A = magic(5);            % Create 5-by-3 matrix, A
A(:, 4:5) = []

A =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25


B = magic(3)*100         % Create 3-by-3 matrix, B

B =

   800   100   600
   300   500   700
   400   900   200
```

```
C = vertcat(A,B)         % Vertically concatenate A and B

C =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25
   800   100   600
   300   500   700
   400   900   200
```

See Also

horzcat, cat

Purpose

Vertical concatenation of `timeseries` objects

Syntax

```
ts = vertcat(ts1,ts2,...)
```

Description

`ts = vertcat(ts1,ts2,...)` performs

```
ts = [ts1;ts2;...]
```

This operation appends `timeseries` objects. The time vectors must not overlap. The last time in ts1 must be earlier than the first time in ts2. The data sample size of the `timeseries` objects must agree.

See Also

`timeseries`

vertcat (tscollection)

Purpose

Vertical concatenation for `tscollection` objects

Syntax

```
tsc = vertcat(tsc1,tsc2,...)
```

Description

`tsc = vertcat(tsc1,tsc2,...)` performs

```
tsc = [tsc1;tsc2;...]
```

This operation appends `tscollection` objects. The time vectors must not overlap. The last time in tsc1 must be earlier than the first time in tsc2. All `tscollection` objects to be concatenated must have the same `timeseries` members.

See Also

`horzcat (tscollection)`, `tscollection`

Purpose

Viewpoint specification

Syntax

```
view(az,el)
view([x,y,z])
view(2)
view(3)
view(ax,...)
view(T)
[az,el] = view
T = view
```

Description

The position of the viewer (the viewpoint) determines the orientation of the axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,el)` and view([az,el]) set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the *z*-axis as measured in degrees from the negative *y*-axis. Positive values indicate counterclockwise rotation of the viewpoint. `el` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the viewpoint to the Cartesian coordinates `x`, `y`, and `z`. The magnitude of `(x,y,z)` is ignored.

`view(2)` sets the default two-dimensional view, `az = 0, el = 90`.

`view(3)` sets the default three-dimensional view, `az = 37.5, el = 30`.

`view(ax,...)` uses axes `ax` instead of the current axes.

`view(T)` sets the view according to the transformation matrix `T`, which is a 4-by-4 matrix such as a perspective transformation generated by `viewmtx`.

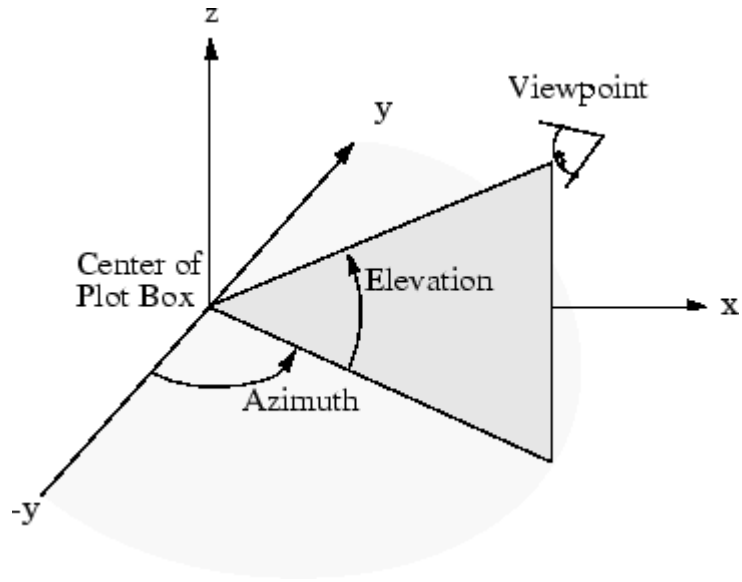
`[az,el] = view` returns the current azimuth and elevation.

`T = view` returns the current 4-by-4 transformation matrix.

Remarks

Azimuth is a polar angle in the *x*-*y* plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the *x*-*y* plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.



Examples

View the object from directly overhead.

```
az = 0;
el = 90;
view(az, el);
```

Set the view along the *y*-axis, with the *x*-axis extending horizontally and the *z*-axis extending vertically in the figure.

```
view([0 0]);
```

Rotate the view about the *z*-axis by 180º.

```
az = 180;
el = 90;
view(az, el);
```

See Also

`viewmtx`, `hgtransform`, `rotate3d`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

Axes graphics object properties `CameraPosition`, `CameraTarget`, `CameraViewAngle`, `Projection`

Defining the View for more information on viewing concepts and techniques

Transforming Objects for information on moving and scaling objects in groups

Purpose

View transformation matrices

Syntax

```
viewmtx
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,phi,xc)
```

Description

`viewmtx` computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

`T = viewmtx(az,el)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `el`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `el` is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,el)
T = view
```

but does not change the current view.

`T = viewmtx(az,el,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

Phi	Description
0 degrees	Orthographic projection
10 degrees	Similar to telephoto lens
25 degrees	Similar to normal lens
60 degrees	Similar to wide-angle lens

You can use the matrix returned to set the view transformation with `view(T)`. The 4-by-4 perspective transformation matrix transforms four-dimensional homogeneous vectors into unnormalized vectors of the

form (*x*,*y*,*z*,*w*), where *w* is not equal to 1. The *x*- and *y*-components of the normalized vector (*x*/*w*, *y*/*w*, *z*/*w*, 1) are the desired two-dimensional components (see example below).

`T = viewmtx(az,el,phi,xc)` returns the perspective transformation matrix using `xc` as the target point within the normalized plot cube (i.e., the camera is looking at the point `xc`). `xc` is the target point that is the center of the view. You specify the point as a three-element vector, `xc = [xc,yc,zc]`, in the interval [0,1]. The default value is `xc = [0,0,0]`.

Remarks

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, `[x,y,z,1]` is the four-dimensional vector corresponding to the three-dimensional point `[x,y,z]`.

Examples

Determine the projected two-dimensional vector corresponding to the three-dimensional point (0.5,0.0,-3.0) using the default view direction. Note that the point is a column vector.

```
A = viewmtx(-37.5,30);
x4d = [.5  0  -3  1]';
x2d = A*x4d;
x2d = x2d(1:2)
x2d =
     0.3967
    -2.4459
```

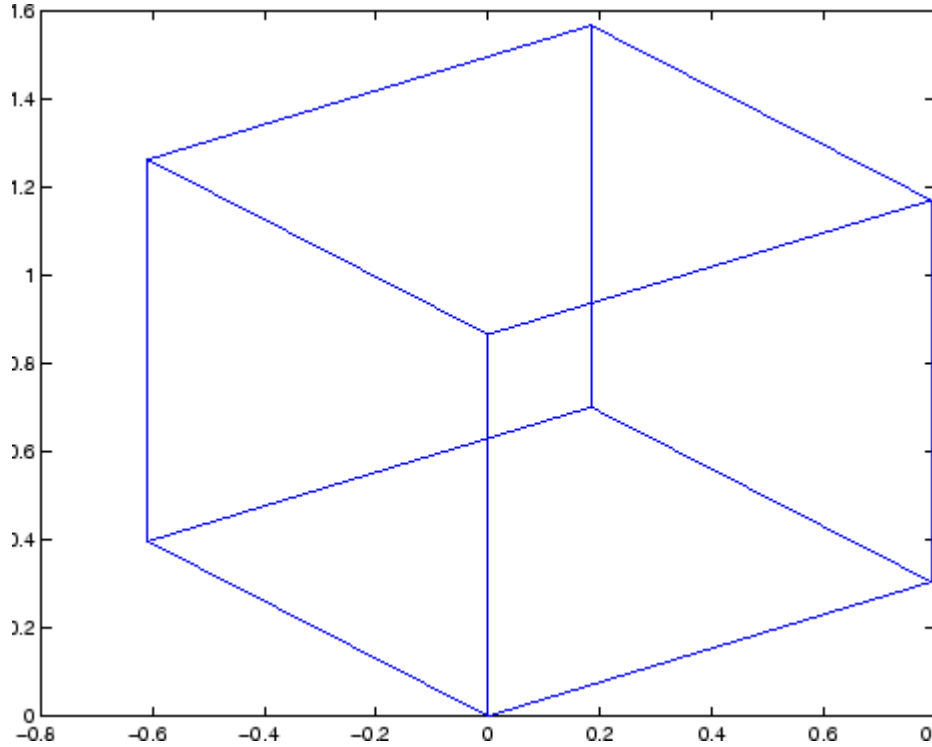
Vectors that trace the edges of a unit cube are

```
x = [0  1  1  0  0  0  1  1  0  0  1  1  1  1  0  0];
y = [0  0  1  1  0  0  0  1  1  0  0  0  1  1  1  1];
z = [0  0  0  0  0  1  1  1  1  1  1  0  0  1  1  0];
```

Transform the points in these vectors to the screen, then plot the object.

```
A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
```

```
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
plot(x2,y2)
```



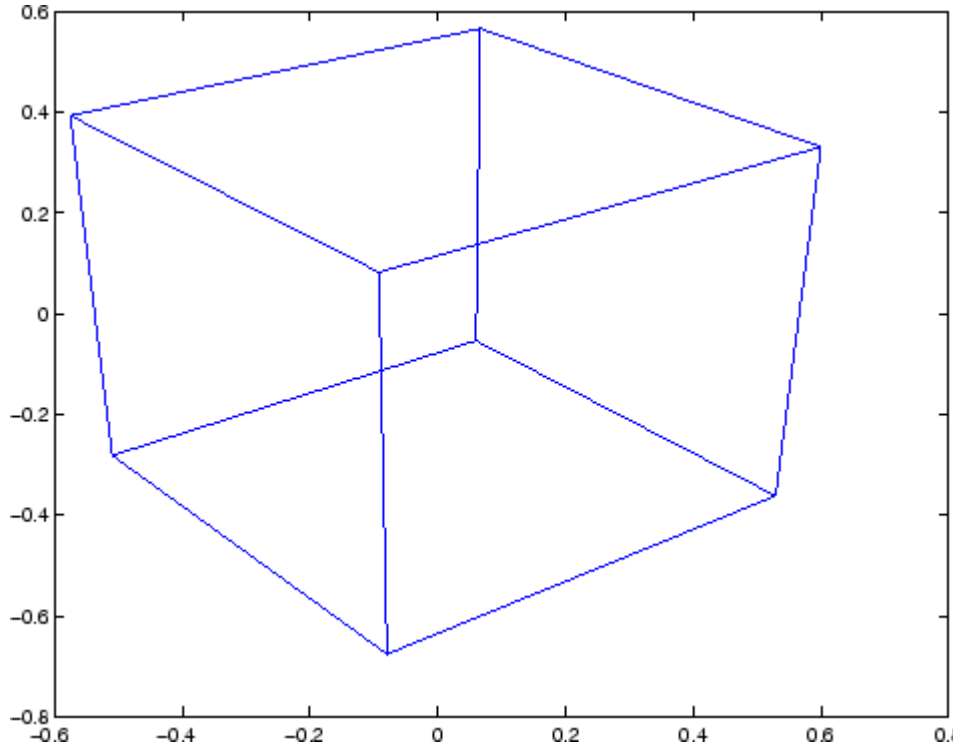
Use a perspective transformation with a 25 degree viewing angle:

```
A = viewmtx(-37.5,30,25);
x4d = [.5  0  -3  1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4)   % Normalize
x2d =
```

```
    0.1777
   -1.8858
```

Transform the cube vectors to the screen and plot the object:

```
A = viewmtx(-37.5,30,25);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)./x2d(4,:);
y2(:) = x2d(2,:)./x2d(4,:);
plot(x2,y2)
```



See Also

`view`, `hgtransform`

"Controlling the Camera Viewpoint" on page 1-99 for related functions

Defining the View for more information on viewing concepts and techniques

Purpose

Coordinate and color limits for volume data

Syntax

```
lims = volumebounds(X,Y,Z,V)
lims = volumebounds(X,Y,Z,U,V,W)
lims = volumebounds(V), lims = volumebounds(U,V,W)
```

Description

`lims = volumebounds(X,Y,Z,V)` returns the x, y, z, and color limits of the current axes for scalar data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax cmin cmax]
```

You can pass this vector to the `axis` command.

`lims = volumebounds(X,Y,Z,U,V,W)` returns the x, y, and z limits of the current axes for vector data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax]
```

`lims = volumebounds(V), lims = volumebounds(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

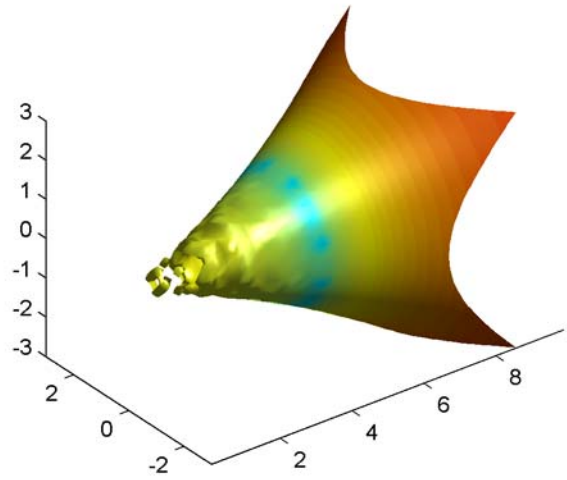
```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(V)`.

Examples

This example uses `volumebounds` to set the axis and color limits for an `isosurface` generated by the `flow` function.

```
[x y z v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
daspect([1 1 1])
isocolors(x,y,z,flipdim(v,2),p)
shading interp
axis(volumebounds(x,y,z,v))
view(3)
camlight
lighting phong
```



See Also

`isosurface`, `streamslice`

"Volume Visualization" on page 1-102 for related functions

Purpose

Voronoi diagram

Syntax

```
voronoi(x,y)
voronoi(x,y,TRI)
voronoi(X,Y,options)
voronoi(AX,...)
voronoi(...,'LineSpec')
h = voronoi(...)
[vx,vy] = voronoi(...)
```

Definition

Consider a set of coplanar points P. For each point P_x in the set P, you can draw a boundary enclosing all the intermediate points lying closer to P_x than to other points in the set P. Such a boundary is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

Description

`voronoi(x,y)` plots the bounded cells of the Voronoi diagram for the points `x`,`y`. Lines-to-infinity are approximated with an arbitrarily distant endpoint.

`voronoi(x,y,TRI)` uses the triangulation `TRI` instead of computing it via `delaunay`.

`voronoi(X,Y,options)` specifies a cell array of strings to be used as options in Qhull via `delaunay`.

If `options` is `[]`, the default `delaunay` options are used. If `options` is `{''}`, no options are used, not even the default.

`voronoi(AX,...)` plots into `AX` instead of `gca`.

`voronoi(...,'LineSpec')` plots the diagram with color and line style specified.

`h = voronoi(...)` returns, in `h`, handles to the line objects created.

`[vx,vy] = voronoi(...)` returns the finite vertices of the Voronoi edges in `vx` and `vy` so that `plot(vx,vy,'-',x,y,'.')` creates the Voronoi diagram. The lines-to-infinity are the last columns of `vx` and

vy. To ensure the lines-to-infinity do not affect the settings of the axis limits, use the commands:

```
h = plot(VX,VY,'-',X,Y,'.');
set(h(1:end-1),'xliminclude','off','yliminclude','off')
```

Note For the topology of the Voronoi diagram, i.e., the vertices for each Voronoi cell, use voronoin.

```
[v,c] = voronoin([x(:) y(:)])
```

Visualization

Use one of these methods to plot a Voronoi diagram:

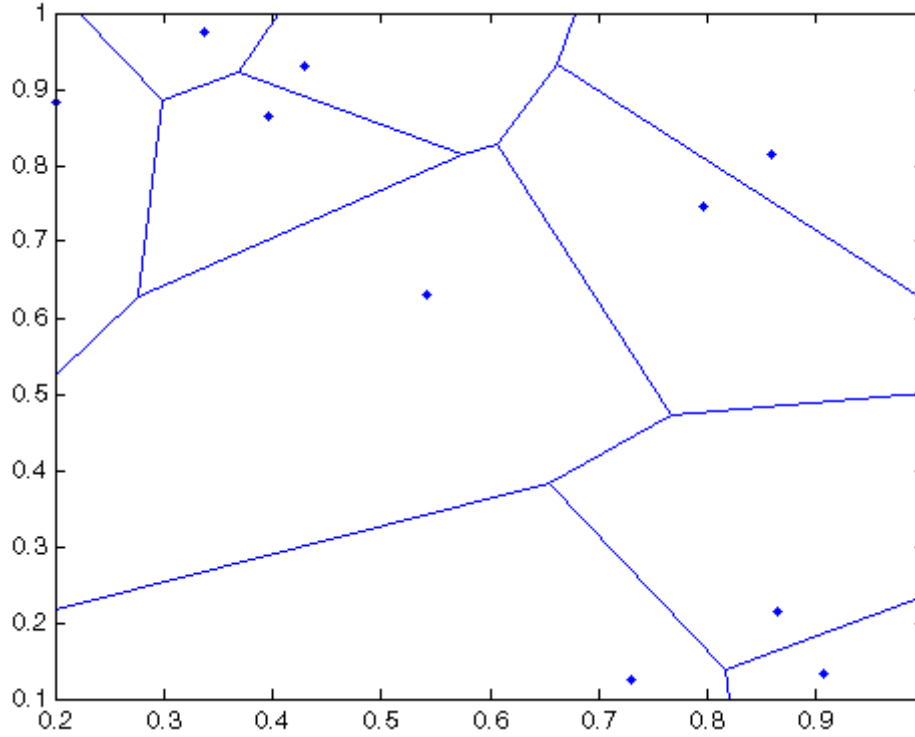
- If you provide no output argument, voronoi plots the diagram. See Example 1.
- To gain more control over color, line style, and other figure properties, use the syntax [vx,vy] = voronoi(...). This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the plot function. See Example 2.
- To fill the cells with color, use voronoin with n = 2 to get the indices of each cell, and then use patch and other plot functions to generate the figure. Note that patch does not fill unbounded cells with color. See Example 3.

Examples

Example 1

This code uses the voronoi function to plot the Voronoi diagram for 10 randomly generated points.

```
rand('state',5);
x = rand(1,10); y = rand(1,10);
voronoi(x,y)
```



Example 2

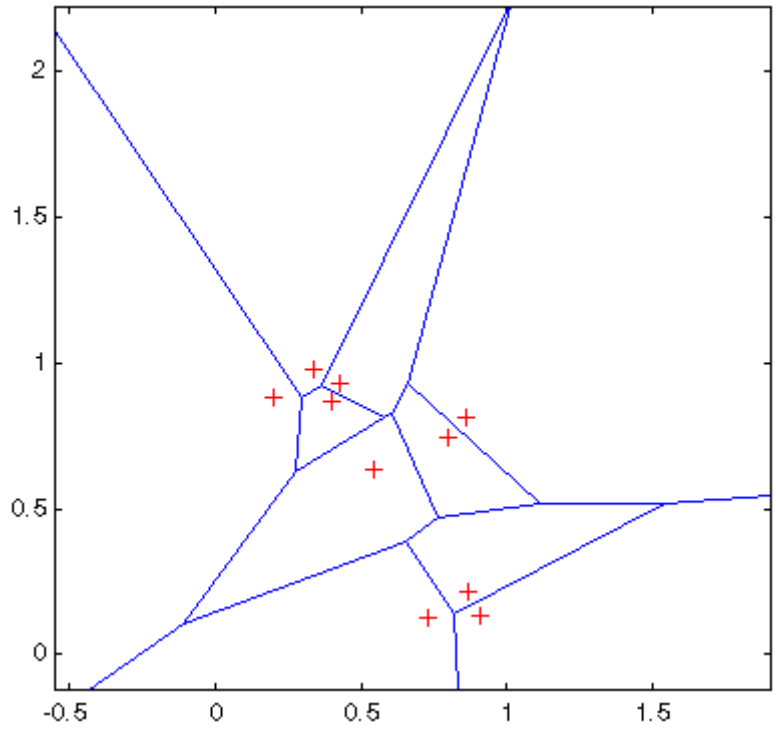
This code uses the vertices of the finite Voronoi edges to plot the Voronoi diagram for the same 10 points.

```
rand('state',5);
x = rand(1,10); y = rand(1,10);
[vx, vy] = voronoi(x,y);
plot(x,y,'r+',vx,vy,'b-'); axis equal
```

Note that you can add this code to get the figure shown in Example 1.

```
xlim([min(x) max(x)])
```

```
ylim([min(y) max(y)])
```

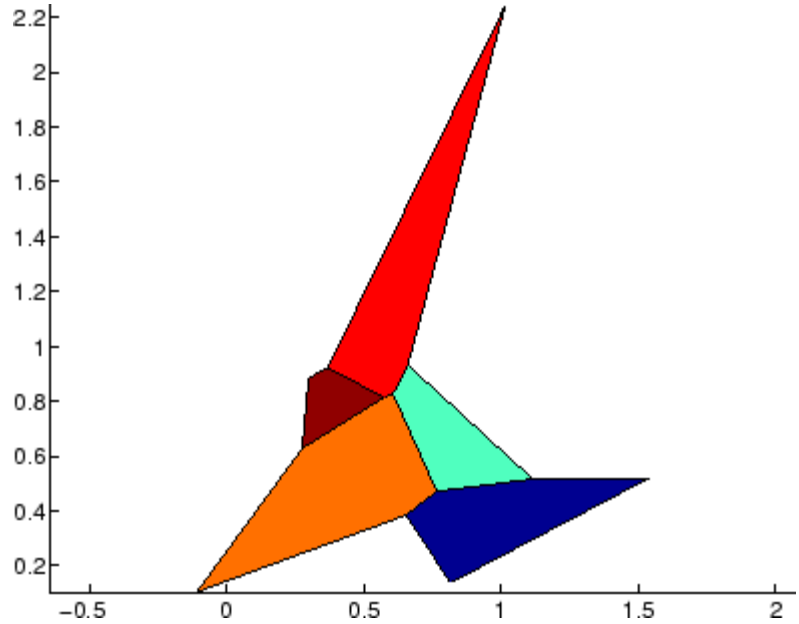


Example 3

This code uses `voronoin` and `patch` to fill the bounded cells of the same Voronoi diagram with color.

```
rand('state',5);
x=rand(10,2);
[v,c]=voronoin(x);
for i = 1:length(c)
if all(c{i}~=1)   % If at least one of the indices is 1,
                  % then it is an open region and we can't
                  % patch that.
```

```
patch(v(c{i},1),v(c{i},2),i); % use color i.
end
end
axis equal
```



Algorithm

If you supply no triangulation `TRI`, the `voronoi` function performs a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`convhull`, `delaunay`, `LineSpec`, `plot`, `voronoin`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF

format at http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/.

Purpose

N-D Voronoi diagram

Syntax

```
[V,C] = voronoin(X)
[V,C] = voronoin(X,options)
```

Description

`[V,C] = voronoin(X)` returns Voronoi vertices `V` and the Voronoi cells `C` of the Voronoi diagram of `X`. `V` is a `numv`-by-`n` array of the `numv` Voronoi vertices in n-dimensional space, each row corresponds to a Voronoi vertex. `C` is a vector cell array where each element contains the indices into `V` of the vertices of the corresponding Voronoi cell. `X` is an `m`-by-`n` array, representing `m` n-dimensional points, where `n > 1` and `m >= n+1`.

The first row of `V` is a point at infinity. If any index in a cell of the cell array is `1`, then the corresponding Voronoi cell contains the first point in `V`, a point at infinity. This means the Voronoi cell is unbounded.

`voronoin` uses Qhull.

`[V,C] = voronoin(X,options)` specifies a cell array of strings `options` to be used in Qhull. The default options are

- `{'Qbb'}` for 2- and 3-dimensional input
- `{'Qbb','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options are used. If `code` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org`.

Visualization

You can plot individual bounded cells of an n-dimensional Voronoi diagram. To do this, use `convhulln` to compute the vertices of the facets that make up the Voronoi cell. Then use `patch` and other plot functions to generate the figure. For an example, see “Tessellation and Interpolation of Scattered Data in Higher Dimensions” in the MATLAB Mathematics documentation.

Examples

Example 1

Let

```
x = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ]
```

then

```
[V,C] = voronoin(x)

V =
       Inf       Inf
       0.3833    0.3833
       0.7000   -1.6500
       0.2875    0.0000
      -0.0000    0.2875
      -0.0000   -0.0000
      -0.0500   -0.5250
      -0.0500   -0.0500
      -1.7500    0.7500
      -1.4500    0.6500
C =

    [1x4 double]
    [1x5 double]
    [1x4 double]
    [1x4 double]
    [1x4 double]
    [1x5 double]
    [1x4 double]
```

Use a `for` loop to see the contents of the cell array `C`.

```
for i=1:length(C), disp(C{i}), end

     4    2    1    3
```

```
    10     5     2     1     9
     9     1     3     7
    10     8     7     9
    10     5     6     8
     8     6     4     3     7
     6     4     2     5
```

In particular, the fifth Voronoi cell consists of 4 points: `V(10,:)`, `V(5,:)`, `V(6,:)`, `V(8,:)`.

Example 2

The following example illustrates the `options` input to `voronoin`. The commands

```
X = [-1 -1; 1 -1; 1 1; -1 1];
[V,C] = voronoin(X)
```

return an error message.

```
? qhull input error: can not scale last coordinate. Input is
cocircular
   or cospherical. Use option 'Qz' to add a point at infinity.
```

The error message indicates that you should add the option `'Qz'`. The following command passes the option `'Qz'`, along with the default `'Qbb'`, to `voronoin`.

```
[V,C] = voronoin(X,{'Qbb','Qz'})
V =

   Inf   Inf
     0     0


C =

    [1x2 double]
    [1x2 double]
```

```
[1x2 double]
[1x2 double]
```

Algorithm

`voronoin` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

See Also

`convhull`, `convhulln`, `delaunay`, `delaunayn`, `voronoi`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in PDF format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/`.

Purpose

Wait until timer stops running

Syntax

```
wait(obj)
```

Description

`wait(obj)` blocks the MATLAB command line and waits until the timer, represented by the timer object `obj`, stops running. When a timer stops running, the value of the timer object's `Running` property changes from `'on'` to `'off'`.

If `obj` is an array of timer objects, `wait` blocks the MATLAB command line until all the timers have stopped running.

If the timer is not running, `wait` returns immediately.

See Also

`timer`, `start`, `stop`

Purpose

Open waitbar

Syntax

```
h = waitbar(x,'message')
waitbar(x,'message','CreateCancelBtn','button_callback')
waitbar(...,property_name,property_value,...)
waitbar(x)
waitbar(x,h)
waitbar(x,h,'updated message')
```

Description

A waitbar shows what percentage of a calculation is complete, as the calculation proceeds.

`h = waitbar(x,'message')` displays a waitbar of fractional length `x`. The waitbar figure is modal. Its handle is returned in `h`. The argument`x` must be between `0` and `1`.

Note A modal figure prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`waitbar(x,'message','CreateCancelBtn','button_callback')` specifying **`CreateCancelBtn`** adds a cancel button to the figure that executes the MATLAB commands specified in `button_callback` when the user clicks the cancel button or the close figure button. `waitbar` sets both the cancel button callback and the figure `CloseRequestFcn` to the string specified in `button_callback`.

`waitbar(...,property_name,property_value,...)` optional arguments `property_name` and `property_value` enable you to set figure properties for the `waitbar`.

`waitbar(x)` subsequent calls to `waitbar(x)` extend the length of the bar to the new position `x`.

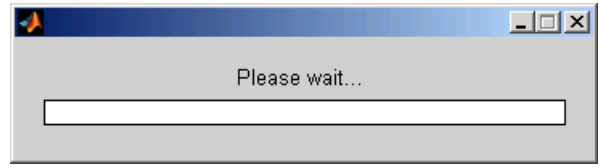
`waitbar(x,h)` extends the length of the bar in the waitbar `h` to the new position `x`.

`waitbar(x,h,'updated message')` updates the message text in the waitbar figure, in addition to setting the fractional length to `x`.

Example

`waitbar` is typically used inside a `for` loop that performs a lengthy computation. For example,

```
h = waitbar(0,'Please wait...');
for i=1:100, % computation here %
waitbar(i/100)
end
close(h)
```



See Also

"Predefined Dialog Boxes" on page 1-104 for related functions

waitfor

Purpose

Wait for condition before resuming execution

Syntax

```
waitfor(h)
waitfor(h,'PropertyName')
waitfor(h,'PropertyName',PropertyValue)
```

Description

The `waitfor` function blocks the caller's execution stream so that command-line expressions, callbacks, and statements in the blocked M-file do not execute until a specified condition is satisfied.

`waitfor(h)` returns when the graphics object identified by `h` is deleted or when a **Ctrl+C** is typed in the Command Window. If `h` does not exist, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName')`, in addition to the conditions in the previous syntax, returns when the value of `'PropertyName'` for the graphics object `h` changes. If `'PropertyName'` is not a valid property for the object, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName',PropertyValue)`, in addition to the conditions in the previous syntax, `waitfor` returns when the value of `'PropertyName'` for the graphics object `h` changes to `PropertyValue`. `waitfor` returns immediately without processing any events if `'PropertyName'` is set to `PropertyValue`.

Remarks

While `waitfor` blocks an execution stream, other execution streams in the form of callbacks may execute as a result of various events (e.g., pressing a mouse button).

`waitfor` can block nested execution streams. For example, a callback invoked during a `waitfor` statement can itself invoke `waitfor`.

See Also

`uiresume`, `uiwait`

"Developing User Interfaces" on page 1-105 for related functions

Purpose

Wait for key press or mouse-button click

Syntax

```
k = waitforbuttonpress
```

Description

`k = waitforbuttonpress` blocks the caller's execution stream until the function detects that the user has clicked a mouse button or pressed a key while the figure window is active. The function returns

- `0` if it detects a mouse button click
- `1` if it detects a key press

Additional information about the event that causes execution to resume is available through the figure's `CurrentCharacter`, `SelectionType`, and `CurrentPoint` properties.

If a `WindowButtonDownFcn` is defined for the figure, its callback is executed before `waitforbuttonpress` returns a value.

Example

These statements display text in the Command Window when the user either clicks a mouse button or types a key in the figure window:

```
w = waitforbuttonpress;
if w == 0
    disp('Button click')
else
    disp('Key press')
end
```

See Also

`dragrect`, `ginput`, `rbbox`, `waitfor`

"Developing User Interfaces" on page 1-105 for related functions

Purpose

Open warning dialog box

Syntax

```
h = warndlg
h = warndlg(warningstring)
h = warndlg(warningstring,dlgname)
h = warndlg(warningstring,dlgname,createmode)
```

Description

`h = warndlg` displays a dialog box named `Warning Dialog` containing the string `This is the default warning string.` The `warndlg` function returns the handle of the dialog box in `h`. The warning dialog box disappears after the user clicks **OK**.

`h = warndlg(warningstring)` displays a dialog box with the title `Warning Dialog` containing the string specified by `warningstring`. The `warningstring` argument can be any valid string format – cell arrays are preferred.

To use multiple lines in your warning, define `warningstring` using either of the following:

- `sprintf` with `newline` characters separating the lines

```
warndlg(sprintf('Message line 1 \n Message line 2'))
```

- Cell arrays of strings

```
warndlg({'Message line 1';'Message line 2'})
```

`h = warndlg(warningstring,dlgname)` displays a dialog box with title `dlgname`.

`h = warndlg(warningstring,dlgname,createmode)` specifies whether the warning dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `warningstring` and `dlgname`. The *`createmode`* argument can be a string or a structure.

If *`createmode`* is a string, it must be one of the values shown in the following table.

createmode Value	Description
`modal`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a modal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.
`non-modal` (default)	Creates a new nonmodal warning dialog box with the specified parameters. Existing warning dialog boxes with the same title are not deleted.
`replace`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a nonmodal warning dialog boxbox as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the`uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in theFigure Properties.

If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings `'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

Examples

The statement

```
warndlg('Pressing OK will clear memory','!! Warning !!')
```

displays this dialog box:

See Also

`dialog`, `errordlg`, `helpdlg`, `inputdlg`, `listdlg`, `msgbox`, `questdlg`

`figure`, `uiwait`, `uiresume`, `warning`

“Predefined Dialog Boxes” on page 1-104 for related functions

Purpose

Warning message

Syntax

```
warning('message')
warning('message', a1, a2,...)
warning('message_id', 'message')
warning('message_id', 'message', a1, a2, ..., an)
s = warning(state, 'message_id')
s = warning(state, mode)
```

Description

`warning('message')` displays the text `'message'` like the `disp` function, except that with `warning`, message display can be suppressed.

`warning('message', a1, a2,...)` displays a message string that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `message` is converted to one of the values `a1`, `a2`, `...` in the argument list.

Note MATLAB converts special characters (like `\n` and `%d`) in the warning message string only when you specify more than one input argument with `warning`. See Example 4 below.

`warning('message_id', 'message')` attaches a unique identifier, or `message_id`, to the warning message. The identifier enables you to single out certain warnings during the execution of your program, controlling what happens when the warnings are encountered. See and "Warning Control" in the MATLAB Programming documentation for more information on the `message_id` argument and how to use it.

`warning('message_id', 'message', a1, a2, ..., an)` includes formatting conversion characters in `message`, and the character translations in arguments `a1, a2, ..., an`.

`s = warning(state, 'message_id')` is a warning control statement that enables you to indicate how you want MATLAB to act on certain warnings. The `state` argument can be `'on'`, `'off'`, or `'query'`. The `message_id` argument can be a message identifier string, `'all'`,

or 'last'. See “Warning Control Statements” in the MATLAB Programming documentation for more information.

Output s is a structure array that indicates the previous state of the selected warnings. The structure has the fields identifier and state. See “Output from Control Statements” in the MATLAB Programming documentation for more.

s = warning(state, mode) is a warning control statement that enables you to display an M-stack trace or display more information with each warning. The state argument can be 'on', 'off', or 'query'. The mode argument can be 'backtrace' or 'verbose'. See “Backtrace and Verbose Modes” in the MATLAB Programming documentation for more information.

Examples

Example 1

Generate a warning that displays a simple string:

```
if ~ischar(p1)
   warning('Input must be a string')
end
```

Example 2

Generate a warning string that is defined at run-time. The first argument defines a message identifier for this warning:

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, "%s".', param)
```

Example 3

Using a message identifier, enable just the actionNotTaken warning from Simulink by first turning off all warnings and then setting just that warning to on:

```
warning off all
warning on Simulink:actionNotTaken
```

Use query to determine the current state of all warnings. It reports that you have set all warnings to off with the exception of Simulink:actionNotTaken:

```
warning query all
The default warning state is 'off'. Warnings not set to the default are

  State  Warning Identifier

     on  Simulink:actionNotTaken
```

Example 4

MATLAB converts special characters (like \n and %d) in the warning message string only when you specify more than one input argument with warning. In the single argument case shown below, \n is taken to mean backslash-n. It is not converted to a newline character:

```
warning('In this case, the newline \n is not converted.')
Warning: In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This is true regardless of whether the additional argument supplies conversion values or is a message identifier:

```
warning('WarnTests:convertTest', ...
      'In this case, the newline \n is converted.')
Warning: In this case, the newline
 is converted.
```

Example 5

Turn on one particular warning, saving the previous state of this one warning in s. Remember that this nonquery syntax performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

After doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

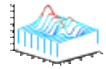
```
warning(s)
```

See Also

lastwarn, warndlg, error, lasterror, errordlg, dbstop, disp, sprintf

Purpose

Waterfall plot



GUI Alternatives

To graph selected variables, use the Plot Selector  in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```
waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)
waterfall(axes_handles,...)
h = waterfall(...)
```

Description

The `waterfall` function draws a mesh similar to the `meshz` function, but it does not generate lines from the columns of the matrices. This produces a “waterfall” effect.

`waterfall(Z)` creates a waterfall plot using `x = 1:size(Z,1)` and `y = 1:size(Z,1)`. `Z` determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in `X`, `Y`, and `Z`. `Z` also determines the color, so color is proportional to the surface height. If `X` and `Y` are vectors, `X` corresponds to the columns of `Z`, and `Y` corresponds to the rows, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(Z)`. `X` and `Y` are vectors or matrices that define the *x*- and *y*-coordinates of the plot. `Z` is a matrix that defines the *z*-coordinates of the plot (i.e., height above a plane). If `C` is omitted, color is proportional to `Z`.

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of `C`, which

must be the same size as `Z`. MATLAB performs a linear transformation on `C` to obtain colors from the current colormap.

`waterfall(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = waterfall(...)` returns the handle of the `patch` graphics object used to draw the plot.

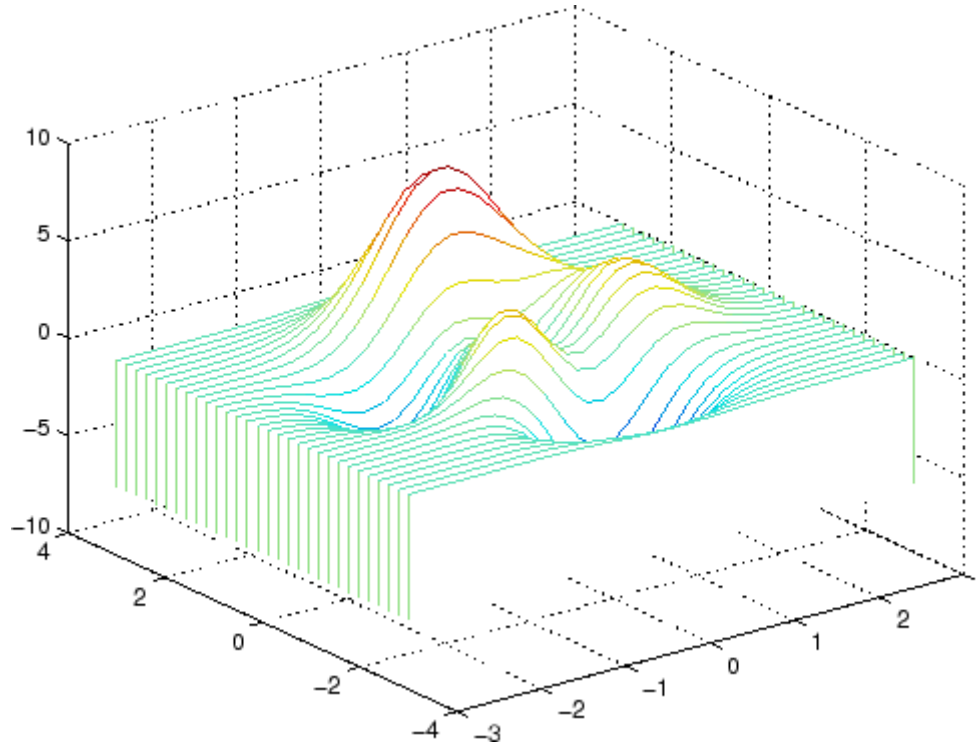
Remarks

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

Examples

Produce a waterfall plot of the `peaks` function.

```
[X,Y,Z] = peaks(30);
waterfall(X,Y,Z)
```



Algorithm

The range of `X`, `Y`, and `Z`, or the current setting of the axes `Llim`, `YLim`, and `ZLim` properties, determines the range of the axes (also set by `axis`). The range of `C`, or the current setting of the axes `CLim` property, determines the color scaling (also set by `caxis`).

The `CData` property for the patch graphics objects specifies the color at every point along the edge of the patch, which determines the color of the lines.

The `waterfall` plot looks like a mesh surface; however, it is a patch graphics object. To create a surface plot similar to `waterfall`, use the `meshz` function and set the `MeshStyle` property of the surface to `'Row'`.

For a discussion of parametric surfaces and related color properties, see `surf`.

See Also

`axes`, `axis`, `caxis`, `meshz`, `ribbon`, `surf`

Properties for patch graphics objects

Purpose

Information about Microsoft WAVE (`.wav`) sound file

Syntax

```
[m d] = wavfinfo(filename)
```

Description

`[m d] = wavfinfo(filename)` returns information about the contents of the WAVE sound file specified by the string `filename`. Enclose the `filename` input in single quotes.

`m` is the string `'Sound (WAV) file'`, if `filename` is a WAVE file. Otherwise, it contains an empty string (`''`).

`d` is a string that reports the number of samples in the file and the number of channels of audio data. If `filename` is not a WAVE file, it contains the string `'Not a WAVE file'`.

See Also

`wavread`

Purpose

Play recorded sound on PC-based audio output device

Syntax

```
wavplay(y,Fs)
wavplay(...,'mode')
```

Description

`wavplay(y,Fs)` plays the audio signal stored in the vector `y` on a PC-based audio output device. You specify the audio signal sampling rate with the integer `Fs` in samples per second. The default value for `Fs` is 11025 Hz (samples per second). `wavplay` supports only 1- or 2-channel (mono or stereo) audio signals.

`wavplay(...,'mode')` specifies how `wavplay` interacts with the command line, according to the string `'mode'`. The string `'mode'` can be

- `'async'`: You have immediate access to the command line as soon as the sound begins to play on the audio output device (a nonblocking device call).
- `'sync'` (default value): You don't have access to the command line until the sound has finished playing (a blocking device call).

The audio signal `y` can be one of four data types. The number of bits used to quantize and play back each sample depends on the data type.

Data Types for wavplay

Data Type	Quantization
Double-precision (default value)	16 bits/sample
Single-precision	16 bits/sample
16-bit signed integer	16 bits/sample
8-bit unsigned integer	8 bits/sample

Remarks

You can play your signal in stereo if `y` is a two-column matrix.

Examples

The MAT-files `gong.mat` and `chirp.mat` both contain an audio signal `y` and a sampling frequency `Fs`. Load and play the gong and the chirp audio signals. Change the names of these signals in between `load` commands and play them sequentially using the `'sync'` option for `wavplay`.

```
load chirp;
y1 = y; Fs1 = Fs;
load gong;
wavplay(y1,Fs1,'sync') % The chirp signal finishes before the
wavplay(y,Fs)          % gong signal begins playing.
```

See Also

`wavrecord`

Purpose

Read Microsoft WAVE (`.wav`) sound file

Graphical Interface

As an alternative to `wavread`, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

Syntax

```
y = wavread(filename)
[y, Fs, nbits] = wavread(filename)
[...] = wavread(filename, N)
[...] = wavread(filename,[N1 N2])
y = wavread(filename, fmt)
siz = wavread(filename,'size')
[y, fs, nbits, opts] = wavread(...)
```

Description

`y = wavread(filename)` loads a WAVE file specified by `filename`, returning the sampled data in `y`. The `filename` input is a string enclosed in single quotes. The `.wav` extension is appended if no extension is given.

`[y, Fs, nbits] = wavread(filename)` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`nbits`) used to encode the data in the file.

`[...] = wavread(filename, N)` returns only the first `N` samples from each channel in the file.

`[...] = wavread(filename,[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

`y = wavread(filename, fmt)` specifies the data type format of `y` used to represent samples read from the file. *`fmt`* can be either of the following values.

Value	Description
`'double'`	`y` contains double-precision normalized samples. This is the default value, if *`fmt`* is omitted.
`'native'`	`y` contains samples in the native data type found in the file. Interpretation of *`fmt`* is case-insensitive, and partial matching is supported.

`siz = wavread(filename,'size')` returns the size of the audio data contained in `filename` in place of the actual audio data, returning the vector `siz = [samples channels]`.

`[y, fs, nbits, opts] = wavread(...)` returns a structure `opts` of additional information contained in the WAV file. The content of this structure differs from file to file. Typical structure fields include `opts.fmt` (audio format information) and `opts.info` (text which may describe title, author, etc.).

Output Scaling

The range of values in `y` depends on the data format *`fmt`* specified. Some examples of output scaling based on typical bit-widths found in a WAV file are given below for both `'double'` and `'native'` formats.

Native Formats

Number of Bits	MATLAB Data Type	Data Range
8	`uint8` (unsigned integer)	0 <= y <= 255
16	`int16` (signed integer)	-32768 <= y <= +32767

Native Formats (Continued)

Number of Bits	MATLAB Data Type	Data Range
24	`int32` (signed integer)	-2^23 <= y <= 2^23-1
32	`single` (floating point)	-1.0 <= y < +1.0

Double Formats

Number of Bits	MATLAB Data Type	Data Range
N<32	`double`	-1.0 <= y < +1.0
N=32	`double`	-1.0 <= y <= +1.0 Note: Values in y might exceed -1.0 or +1.0 for the case of N=32 bit data samples stored in the WAV file.

`wavread` supports multi-channel data, with up to 32 bits per sample.

`wavread` supports Pulse-code Modulation (PCM) data format only.

See Also

`auread`, `auwrite`, `wavwrite`

Purpose

Record sound using PC-based audio input device

Syntax

```
y = wavrecord(n,Fs)
y = wavrecord(...,ch)
y = wavrecord(...,'dtype')
```

Description

`y = wavrecord(n,Fs)` records `n` samples of an audio signal, sampled at a rate of `Fs` Hz (samples per second). The default value for `Fs` is 11025 Hz.

`y = wavrecord(...,ch)` uses `ch` number of input channels from the audio device. `ch` can be either 1 or 2, for mono or stereo, respectively. The default value for `ch` is 1.

`y = wavrecord(...,'dtype')` uses the data type specified by the string `'dtype'` to record the sound. The string `'dtype'` can be one of the following:

- `'double'` (default value), 16 bits/sample
- `'single'`, 16 bits/sample
- `'int16'`, 16 bits/sample
- `'uint8'`, 8 bits/sample

Remarks

Standard sampling rates for PC-based audio hardware are 8000, 11025, 2250, and 44100 samples per second. Stereo signals are returned as two-column matrices. The first column of a stereo audio matrix corresponds to the left input channel, while the second column corresponds to the right input channel.

Examples

Record 5 seconds of 16-bit audio sampled at 11025 Hz. Play back the recorded sound using `wavplay`. Speak into your audio device (or produce your audio signal) while the `wavrecord` command runs.

```
Fs = 11025;
y = wavrecord(5*Fs,Fs,'int16');
wavplay(y,Fs);
```

See Also wavplay

Purpose

Write Microsoft WAVE (`.wav`) sound file

Syntax

```
wavwrite(y,filename)
wavwrite(y,Fs,filename)
wavwrite(y,Fs,N,filename)
```

Description

`wavwrite` writes data to 8-, 16-, 24-, and 32-bit `.wav` files.

`wavwrite(y,filename)` writes the data stored in the variable `y` to a WAVE file called `filename`. The `filename` input is a string enclosed in single quotes. The data has a sample rate of 8000 Hz and is assumed to be 16-bit. Each column of the data represents a separate channel. Therefore, stereo data should be specified as a matrix with two columns. Amplitude values outside the range `[-1,+1]` are clipped prior to writing.

`wavwrite(y,Fs,filename)` writes the data stored in the variable `y` to a WAVE file called `filename`. The data has a sample rate of `Fs` Hz and is assumed to be 16-bit. Amplitude values outside the range `[-1,+1]` are clipped prior to writing.

`wavwrite(y,Fs,N,filename)` writes the data stored in the variable `y` to a WAVE file called `filename`. The data has a sample rate of `Fs` Hz and is `N`-bit, where `N` is 8, 16, 24, or 32. For `N` < 32, amplitude values outside the range `[-1,+1]` are clipped.

Note 8-, 16-, and 24-bit files are type 1 integer pulse code modulation (PCM). 32-bit files are written as type 3 normalized floating point.

See Also

`auwrite`, `wavread`

Purpose

Open Web site or file in Web browser or Help browser

Syntax

```
web
web url
web url -new
web url -notoolbar
web url -noaddressbox
web url -helpbrowser
web url -browser
web(...)
stat = web('url', '-browser')
[stat, h1] = web
[stat, h1, url] = web
```

Description

`web` opens an empty MATLAB "Web Browser". The MATLAB Web browser includes an address field where you can enter a URL, for example, to a Web site or file, a toolbar with common browser buttons, and a MATLAB desktop menu.

`web url` displays the specified URL, `url`, in the MATLAB Web browser. If any MATLAB Web browsers are already open, it displays the page in the browser that last had focus. Files up to 1.5MB in size display in the MATLAB Web browser, while larger files instead display in the default Web browser for your system. If `url` is located in the directory returned when you run `docroot` (an unsupported utility), the URL displays in the MATLAB Help browser instead of the MATLAB Web browser.

`web url -new` displays the specified URL, `url`, in a new MATLAB Web browser.

`web url -notoolbar` displays the specified URL, `url`, in a MATLAB Web browser that does not include the toolbar and address field. If any MATLAB Web browsers are already open, also use the `-new` option; otherwise `url` displays in the browser that last had focus, regardless of its toolbar status.

`web url -noaddressbox` displays the specified URL, `url`, in a MATLAB Web browser that does not include the address field. If any MATLAB Web browsers are already open, also use the `-new` option; otherwise `url`

displays in the browser that last had focus, regardless of its address field status.

`web url -helpbrowser` displays the specified URL, `url`, in the MATLAB Help browser.

`web url -browser` displays the default Web browser for your system and loads the file or Web site specified by the URL `url` in it. Generally, `url` specifies a local file or a Web site on the Internet. The URL can be in any form that the browser supports. On Windows and Macintosh, the default Web browser is determined by the operating system. On UNIX, the Web browser used is specified via `docopt` in the `doccmd` string.

`web(...)` is the functional form of `web`.

`stat = web('url', '-browser')` runs `web` and returns the status of `web` to the variable `stat`.

Value of `stat`	Description
0	Browser was found and launched.
1	Browser was not found.
2	Browser was found but could not be launched.

`[stat, h1] = web` returns the status of `web` to the variable `stat`, and returns a handle to the Java class, `h1`, for the last active browser.

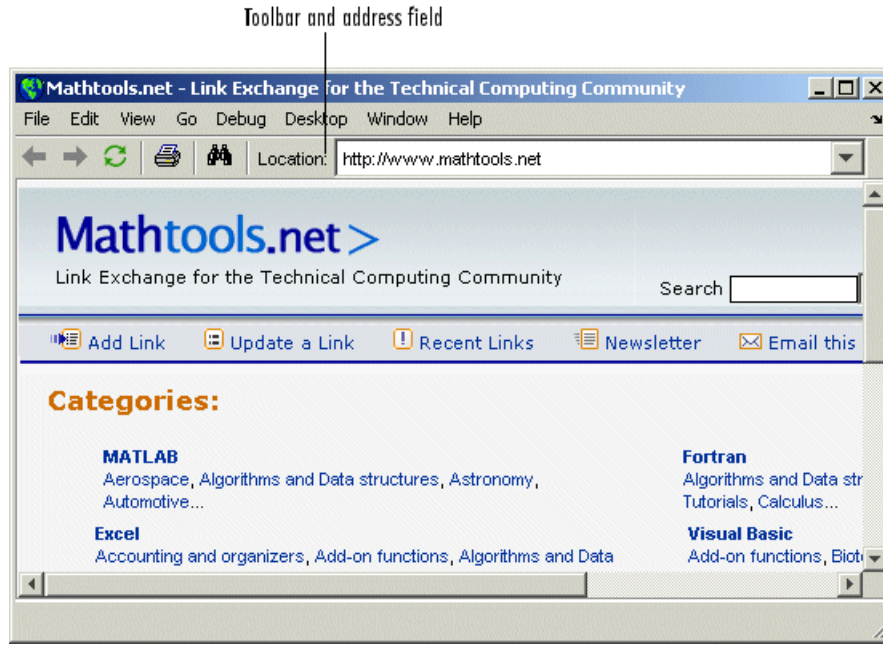
`[stat, h1, url] = web` returns the status of `web` to the variable `stat`, returns a handle to the Java class `h1`, for the last active browser, and returns its current URL to `url`.

Examples

Run

```
web http://www.mathtools.net
```

and MATLAB displays



`web http://www.mathworks.com` loads the MathWorks Web site home page into the MATLAB Web browser.

`web file:///disk/dir1/dir2/foo.html` opens the file `foo.html` in the MATLAB Web browser.

`web(['file:///' which('foo.html')])`opens `foo.html` if the file is on the MATLAB path or in the current directory.

`web('text://<html><h1>Hello World</h1></html>')` displays the HTML-formatted text `Hello World`.

`web ('http://www.mathworks.com', '-new', '-notoolbar')` loads the MathWorks Web site home page into a new MATLAB Web browser that does not include a toolbar or address field.

`web file:///disk/dir1/foo.html -helpbrowser` opens the file `foo.html` in the MATLAB Help browser.

`web file:///disk/dir1/foo.html -browser` opens the file `foo.html` in the system Web browser.

`web mailto:email_address` uses your system browser's default e-mail application to send a message to `email_address`.

`web http://www.mathtools.net -browser` opens a browser to `mathtools.net`. Then `[stat,h1,url]=web` returns

```
stat =
     0

h1 =
com.mathworks.mde.webbrowser.WebBrowser[,0,0,591x140,
layout=java.awt.BorderLayout,alignmentX=null,alignmentY=null,
border=,flags=9,maximumSize=,minimumSize=,preferredSize=]

url =
http://www.mathtools.net/
```

Run `methods(h1)` to view allowable methods for the class. As an example, you can use the method `setCurrentLocation` to change the URL displayed in `h1`, as in

```
setCurrentLocation(h1,'http://www.mathworks.com')
```

See Also

`doc`, `docopt`, `helpbrowser`, `matlabcolon`

"Web Browser" in the MATLAB Desktop Tools and Development Environment documentation

Purpose

Day of week

Syntax

```
[N, S] = weekday(D)
[N, S] = weekday(D, form)
[N, S] = weekday(D, locale)
[N, S] = weekday(D, form, locale)
```

Description

`[N, S] = weekday(D)` returns the day of the week in numeric (`N`) and string (`S`) form for a given serial date number or date string `D`. Input argument `D` can represent more than one date in an array of serial date numbers or a cell array of date strings.

`[N, S] = weekday(D, form)` returns the day of the week in numeric (`N`) and string (`S`) form, where the content of `S` depends on the `form` argument. If `form` is '**`long`**', then `S` contains the full name of the weekday (e.g., `Tuesday`). If `form` is '**`short`**', then `S` contains an abbreviated name (e.g., `Tues`) from this table.

The days of the week are assigned these numbers and abbreviations.

N	S (short)	S (long)
1	Sun	Sunday
2	Mon	Monday
3	Tue	Tuesday
4	Wed	Wednesday
5	Thu	Thursday
6	Fri	Friday
7	Sat	Saturday

`[N, S] = weekday(D, locale)` returns the day of the week in numeric (`N`) and string (`S`) form, where the format of the output depends on the `locale` argument. If `locale` is '**`local`**', then `weekday` uses local format for its output. If `locale` is '**`en_US`**', then `weekday` uses US English.

`[N, S] = weekday(D, form, locale)` returns the day of the week using the formats described above for `form` and `locale`.

Examples

Either

```
[n, s] = weekday(728647)
```

or

```
[n, s] = weekday('19-Dec-1994')
```

returns `n = 2` and `s = Mon`.

See Also

`datenum`, `datevec`, `eomday`

Purpose

List MATLAB files in current directory

Graphical Interface

As an alternative to the `what` function, use the "Current Directory Browser". To open it, select **Current Directory** from the **Desktop** menu in the MATLAB desktop.

Syntax

```
what
what dirname
what class
s = what('dirname')
```

Description

`what` lists the M, MAT, MEX, MDL, and P-files and the class directories that reside in the current working directory.

`what dirname` lists the files in directory `dirname` on the MATLAB search path. It is not necessary to enter the full pathname of the directory. The last component, or last two components, is sufficient.

`what class` lists the files in method directory, `@class`. For example, `what cfit` lists the MATLAB files in `toolbox/curvefit/curvefit/@cfit`.

`s = what('dirname')` returns the results in a structure array with these fields.

Field	Description
`path`	Path to directory
`m`	Cell array of M-file names
`mat`	Cell array of MAT-file names
`mex`	Cell array of MEX-file names
`mdl`	Cell array of MDL-file names
`p`	Cell array of P-file names
`classes`	Cell array of class names

Examples

List the files in `toolbox/matlab/audiovideo`:

```
what audiovideo

M-files in directory matlabroot\toolbox\matlab\audiovideo

Contents             aviinfo                   render_uimgraudiotoolbar
audiodevinfo         aviread                   sound
audioplayerreg       lin2mu                    soundsc
audiorecorderreg     mmcompinfo                wavfinfo
audiouniquename      mmfileinfo                wavplay
aufinfo              movie2avi                 wavread
auread               mu2lin                    wavrecord
auwrite              prefspanel                wavwrite
avifinfo             render_fullaudiotoolbar

MAT-files in directory matlabroot\toolbox\matlab\audiovideo

chirp         handel        splat
gong          laughter      train

MEX-files in directory matlabroot\toolbox\matlab\audiovideo

winaudioplayer         winaudiorecorder

Classes in directory matlabroot\toolbox\matlab\audiovideo

audioplayer      audiorecorder      avifile      mmreader
```

Obtain a structure array containing the MATLAB filenames in `toolbox/matlab/general`:

```
s = what('general')
s =
       path: 'matlabroot:\toolbox\matlab\general'
          m: {87x1 cell}
        mat: {0x1 cell}
```

```
                mex: {2x1 cell}
                mdl: {0x1 cell}
                  p: {'callgraphviz.p'}
            classes: {0x1 cell}
```

See Also

dir, exist, lookfor, mfilename, path, which, who

Purpose

Release Notes for MathWorks products

Syntax

```
whatsnew
```

Description

`whatsnew` displays the Release Notes in the Help browser, presenting information about new features, problems from previous releases that have been fixed in the current release, and compatibility issues, all organized by product.

See Also

`help`, `version`

which

Purpose

Locate functions and files

Graphical Interface

As an alternative to the `which` function, use the “Current Directory Browser”.

Syntax

```
which fun
which classname/fun
which private/fun
which classname/private/fun
which fun1 in fun2
which fun(a,b,c,...)
which file.ext
which fun -all
s = which('fun',...)
```

Description

`which fun` displays the full pathname for the argument `fun`. If `fun` is a

- MATLAB function or Simulink model in an M, P, or MDL file on the MATLAB path, then `which` displays the full pathname for the corresponding file
- Workspace variable, then `which` displays a message identifying `fun` as a variable
- Method in a loaded Java class, then `which` displays the package, class, and method name for that method

If `fun` is an overloaded function or method, then `which fun` returns only the pathname of the first function or method found.

`which classname/fun` displays the full pathname for the M-file defining the `fun` method in MATLAB class, `classname`. For example, `which serial/fopen` displays the path for `fopen.m` in the MATLAB class directory, `@serial`.

`which private/fun` limits the search to private functions. For example, `which private/orthog` displays the path for `orthog.m` in the `/private` subdirectory of `toolbox/matlab/elmat`.

`which classname/private/fun` limits the search to private methods defined by the MATLAB class, `classname`. For example, `which dfilt/private/todtf` displays the path for `todtf.m` in the `private` directory of the `dfilt` class.

`which fun1 in fun2` displays the pathname to function `fun1` in the context of the M-file `fun2`. You can use this form to determine whether a subfunction is being called instead of a function on the path. For example, `which get in editpath` tells you which `get` function is called by `editpath.m`.

During debugging of `fun2`, using `which fun1` gives the same result.

`which fun(a,b,c,...)` displays the path to the specified function with the given input arguments. For example, `which feval(g)`, when `g=inline('sin(x)')`, indicates that `inline/feval.m` would be invoked. `which toLowerCase(s)`, when `s=java.lang.String('my Java string')`, indicates that the `toLowerCase` method in `class java.lang.String` would be invoked.

`which file.ext` displays the full pathname of the specified file if that file is in the current working directory or on the MATLAB path. To display the path for a file that has no file extension, type "`which file.`" (the period following the filename is required). Use `exist` to check for the existence of files anywhere else.

`which fun -all` d isplays the paths to all items on the MATLAB path with the name `fun`. You may use the **`-all`** qualifier with any of the above formats of the `which` function.

`s = which('fun',...)` returns the results of `which` in the string `s`. For workspace variables, `s` is the string '`variable`'. You may specify an output variable in any of the above formats of the `which` function.

If **`-all`** is used with this form, the output `s` is always a cell array of strings, even if only one string is returned.

Examples

The statement below indicates that `pinv` is in the `matfun` directory of MATLAB.

```
which pinv
matlabroot\toolbox\matlab\matfun\pinv.m
```

To find the `fopen` function used on MATLAB `serial` class objects

```
which serial/fopen
matlabroot\toolbox\matlab\iofun\@serial\fopen.m  % serial method
```

To find the `setTitle` method used on objects of the Java `Frame` class, the class must first be loaded into MATLAB. The class is loaded when you create an instance of the class:

```
frameObj = java.awt.Frame;

which setTitle
java.awt.Frame.setTitle  % Frame method
```

When you specify an output variable, `which` returns a cell array of strings to the variable. You must use the *function* form of `which`, enclosing all arguments in parentheses and single quotes:

```
s = which('private/stradd','-all');
whos s
  Name      Size         Bytes  Class
  s         3x1            562  cell array
Grand total is 146 elements using 562 bytes
```

See Also

`dir`, `doc`, `exist`, `lookfor`, `mfilename`, `path`, `type`, `what`, `who`

Purpose

Repeatedly execute statements while condition is true

Syntax

```
while expression, statements, end
```

Description

`while expression, statements, end` repeatedly executes one or more MATLAB *`statements`* in a loop, continuing until *`expression`* no longer holds true or until MATLAB encounters a `break`, or `return` instruction. thus forcing an immediately exit of the loop. If MATLAB encounters a `continue` statement in the loop code, it immediately exits the current pass at the location of the `continue` statement, skipping any remaining code in that pass, and begins another pass at the start of the loop *`statements`* with the value of the loop counter incremented by 1.

`expression` is a MATLAB expression that evaluates to a result of logical 1 (`true`) or logical 0 (`false`). *`expression`* can be scalar or an array. It must contain all real elements, and the statement `all(A(:))` must be equal to logical 1 for the expression to be true.

`expression` usually consists of variables or smaller expressions joined by relational operators (e.g., `count < limit`) or logical functions (e.g., `isreal(A)`). Simple expressions can be combined by logical operators (`&&`, `||`, `~`) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to “Operator Precedence” rules.

```
(count < limit) && ((height - offset) >= 0)
```

`statements` is one or more MATLAB statements to be executed only while the *`expression`* is `true` or nonzero.

The scope of a `while` statement is always terminated with a matching `end`.

See “Program Control Statements”in the MATLAB Programming documentation for more information on controlling the flow of your program code.

Remarks

Nonscalar Expressions

If the evaluated `expression` yields a nonscalar value, then every element of this value must be `true` or nonzero for the entire expression to be considered `true`. For example, the statement `while (A < B)` is `true` only if each element of matrix `A` is less than its corresponding element in matrix `B`. See “Example 2 – Nonscalar Expression” on page 2-3727, below.

Partial Evaluation of the Expression Argument

Within the context of an `if` or `while` expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is `true` or `false` through only partial evaluation.

For example, if `A` equals zero in statement 1 below, then the expression evaluates to `false`, regardless of the value of `B`. In this case, there is no need to evaluate `B` and MATLAB does not do so. In statement 2, if `A` is nonzero, then the expression is `true`, regardless of `B`. Again, MATLAB does not evaluate the latter part of the expression.

```
1)   while (A && B)              2)   while (A || B)
```

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

```
while (b ~= 0) && (a/b > 18.5)
if exist('myfun.m') && (myfun(x) >= y)
if iscell(A) && all(cellfun('isreal', A))
```

Empty Arrays

In most cases, using `while` on an empty array returns `false`. There are some conditions however under which `while` evaluates as `true` on an empty array. Two examples of this are

```
A = [];
while all(A), do_something, end
while 1|A, do_something, end
```

Short-Circuiting Behavior

When used in the context of a `while` or `if` expression, and only in this context, the element-wise `|` and `&` operators use short-circuiting in evaluating their expressions. That is, `A|B` and `A&B` ignore the second operand, `B`, if the first operand, `A`, is sufficient to determine the result.

See "Short-Circuiting in Elementwise Operators" for more information on this.

Examples

Example 1 – Simple while Statement

The variable `eps` is a tolerance used to determine such things as near singularity and rank. Its initial value is the *machine epsilon*, the distance from 1.0 to the next largest floating-point number on your machine. Its calculation demonstrates `while` loops.

```
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
eps = eps*2
```

This example is for the purposes of illustrating `while` loops only and should not be executed in your MATLAB session. Doing so will disable the `eps` function from working in that session.

Example 2 – Nonscalar Expression

Given matrices `A` and `B`,

```
A =                 B =
     1     0             1     1
     2     3             3     4
```

Expression	Evaluates As	Because
`A < B`	`false`	A(1,1) is not less than B(1,1).

Expression	Evaluates As	Because
`A < (B + 1)`	`true`	Every element of A is less than that same element of B with 1 added.
`A & B`	`false`	A(1,2) is false, and B is ignored due to short-circuiting.
`B < 5`	`true`	Every element of B is less than 5.

See Also

`end`, `for`, `break`, `continue`, `return`, `all`, `any`, `if`, `switch`

Purpose

Change axes background color

Syntax

```
whitebg
whitebg(fig)
whitebg(ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig)
```

Description

`whitebg` complements the colors in the current figure.

`whitebg(fig)` complements colors in all figures specified in the vector `fig`.

`whitebg(ColorSpec)` and `whitebg(fig, ColorSpec)` change the color of the axes, which are children of the figure, to the color specified by `ColorSpec`. Without a figure specification, `whitebg` or `whitebg(ColorSpec)` affects the current figure and the root's default properties so subsequent plots and new figures use the new colors.

`whitebg(fig, ColorSpec)` sets the default axes background color of the figures in the vector `fig` to the color specified by `ColorSpec`. Other axes properties and the figure background color can change as well so that graphs maintain adequate contrast. `ColorSpec` can be a 1-by-3 RGB color or a color string such as `'white'` or `'w'`.

`whitebg(fig)` complements the colors of the objects in the specified figures. This syntax is typically used to toggle between black and white axes background colors, and is where `whitebg` gets its name. Include the root window handle `(0)` in `fig` to affect the default properties for new windows or for `clf reset`.

Remarks

`whitebg` works best in cases where all the axes in the figure have the same background color.

`whitebg` changes the colors of the figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. `whitebg` sets the default properties on the root such that all subsequent figures use the new background color.

Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

See Also

`ColorSpec`, `colordef`

The figure graphics object property `InvertHardCopy`

“Color Operations” on page 1-98 for related functions

Purpose

List variables in workspace

Graphical Interface

As an alternative to `whos`, use the Workspace browser. Or use the Current Directory browser to view the contents of MAT-files without loading them.

Syntax

```
who
whos
who(variable_list)
whos(variable_list)
who(variable_list, qualifiers)
whos(variable_list, qualifiers)
s = who(variable_list, qualifiers)
s = whos(variable_list, qualifiers)
who variable_list qualifiers
whos variable_list qualifiers
```

Each of these syntaxes apply to both `who` and `whos`:

Description

`who` lists in alphabetical order all variables in the currently active workspace.

`whos` lists in alphabetical order all variables in the currently active workspace along with their sizes and types. It also reports the totals for sizes.

Note If `who` or `whos` is executed within a nested function, MATLAB lists the variables in the workspace of that function and in the workspaces of all functions containing that function. See the Remarks section, below.

`who(variable_list)` and `whos(variable_list)` list only those variables specified in `variable_list`, where `variable_list` is a comma-delimited list of quoted strings: `'var1', 'var2', ..., 'varN'`. You can use the wildcard character `*` to display variables that

match a pattern. For example, `who('A*')` finds all variables in the current workspace that start with `A`.

`who(variable_list, qualifiers)` and `whos(variable_list, qualifiers)` list those variables in `variable_list` that meet all qualifications specified in `qualifiers`. You can specify any or all of the following qualifiers, and in any order.

Qualifier Syntax	Description	Example
`'global'`	List variables in the global workspace.	`whos('global')`
`'-file', filename`	List variables in the specified MAT-file. Use the full path for `filename`.	`whos('-file', 'mydata')`
`'-regexp', exprlist`	List variables that match any of the regular expressions in `exprlist`.	`whos('-regexp', '[AB].', '\w\d')`

`s = who(variable_list, qualifiers)` returns cell array `s` containing the names of the variables specified in `variable_list` that meet the conditions specified in `qualifiers`.

`s = whos(variable_list, qualifiers)` returns structure `s` containing the following fields for the variables specified in `variable_list` that meet the conditions specified in `qualifiers`:

Field Name	Description
`name`	Name of the variable
`size`	Dimensions of the variable array
`bytes`	Number of bytes allocated for the variable array
`class`	Class of the variable. Set to the string `'(unassigned)'` if the variable has no value.

Field Name	Description
`global`	True if the variable is global; otherwise false
`sparse`	True if the variable is sparse; otherwise false
`complex`	True if the variable is complex; otherwise false
`nesting`	Structure having the following fields: • `function` — Name of the nested or outer function that defines the variable • `level` — Nesting level of that function
`persistent`	True if the variable is persistent; otherwise false

`who variable_list qualifiers` and `whos variable_list qualifiers` are the unquoted forms of the syntax. Both `variable_list` and `qualifiers` are space-delimited lists of unquoted strings.

Remarks

Nested Functions. When you use `who` or `whos` inside of a nested function, MATLAB returns or displays all variables in the workspace of that function, and in the workspaces of all functions in which that function is nested. This applies whether you include calls to `who` or `whos` in your M-file code or if you call `who` or `whos` from the MATLAB debugger.

If your code assigns the output of `whos` to a variable, MATLAB returns the information in a structure array containing the fields described above. If you do not assign the output to a variable, MATLAB displays the information at the Command Window, grouped according to workspace.

If your code assigns the output of `who` to a variable, MATLAB returns the variable names in a cell array of strings. If you do not assign the output, MATLAB displays the variable names at the Command Window, but not grouped according to workspace.

Compressed Data. Information returned by the command `whos -file` is independent of whether the data in that file is compressed or not. The byte counts returned by this command represent the number of bytes data occupies in the MATLAB workspace, and not in the file the data was saved to. See the function reference for `save` for more information on data compression.

MATLAB Objects. `whos -file` *`filename`* does not return the sizes of any MATLAB objects that are stored in file *`filename`*.

Examples

Example 1

Show variable names starting with the letter `a`:

```
who a*
```

Show variables stored in MAT-file `mydata.mat`:

```
who -file mydata
```

Example 2

Return information on variables stored in file `mydata.mat` in structure array `s`:

```
s = whos('-file', 'mydata1')
s =
6x1 struct array with fields:
    name
    size
    bytes
    class
    global
    sparse
    complex
    nesting
    persistent
```

Display the name, size, and class of each of the variables returned by `whos`:

```
for k=1:length(s)
disp(['  ' s(k).name '  ' mat2str(s(k).size) '  ' s(k).class])
end
  A   [1 1]   double
  spArray   [5 5]   double
  strArray   [2 5]   cell
  x   [3 2 2]   double
  y   [4 5]   cell
```

Example 3

Show variables that start with `java` and end with `Array`. Also show their dimensions and class name:

```
whos -file mydata2 -regexp \<java.*Array\>
  Name               Size        Bytes  Class

  javaChrArray       3x1                java.lang.String[][][]
  javaDblArray       4x1                java.lang.Double[][]
  javaIntArray      14x1                java.lang.Integer[][]
```

Example 4

The function shown here uses variables with persistent, global, sparse, and complex attributes:

```
function show_attributes
persistent p;
global g;
o = 1;   g = 2;
s = sparse(eye(5));
c = [4+5i 9-3i 7+6i];
whos
```

When the function is run, `whos` displays these attributes:

```
show_attributes
```

```
  Name      Size            Bytes  Class     Attributes

c           1x3                48  double    complex
g           1x1                 8  double    global
p           1x1                 8  double    persistent
s           5x5                84  double    sparse
```

Example 5

Function `whos_demo` contains two nested functions. One of these functions calls `whos`; the other calls `who`:

```
function whos_demo
date_time = datestr(now);

[str pos] = textscan(date_time, '%s%s%s', ...
                     1, 'delimiter', '- :');
get_date(str);

str = textscan(date_time(pos+1:end), '%s%s%s', ...
               1, 'delimiter', '- :');
get_time(str);

   function get_date(d)
      day = d{1};   mon = d{2};   year = d{3};
      whos
   end
   function get_time(t)
      hour = t{1};   min = t{2};   sec = t{3};
      who
   end
end
```

When nested function `get_date` calls `whos`, MATLAB displays information on the variables in all workspaces that are in scope at the time. This includes nested function `get_date` and also the function in which it is nested, `whos_demo`. The information is grouped by workspace:

```
whos_demo
  Name            Size                Bytes  Class

  ---- get_date ----------------------------------------
  d               1x3                   378  cell
  day             1x1                    64  cell
  mon             1x1                    66  cell
  year            1x1                    68  cell

  ---- whos_demo ---------------------------------------
  ans             0x0                     0  (unassigned)
  date_time       1x20                   40  char
  pos             1x1                     8  double
  str             1x3                   378  cell
```

When nested function `get_time` calls `who`, MATLAB displays names of the variables in the workspaces that are in scope at the time. This includes nested function `get_time` and also the function in which it is nested, `whos_demo`. The information is not grouped by workspace in this case:

```
Your variables are:

hour      min       sec       t       ans       date_time
pos       str
```

See Also

`assignin`, `clear`, `computer`, `dir`, `evalin`, `exist`, `inmem`, `load`, `save`, `what`, `workspace`

wilkinson

Purpose

Wilkinson's eigenvalue test matrix

Syntax

```
W = wilkinson(n)
```

Description

`W = wilkinson(n)` returns one of J. H. Wilkinson's eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Examples

```
wilkinson(7)

ans =

     3     1     0     0     0     0     0
     1     2     1     0     0     0     0
     0     1     1     1     0     0     0
     0     0     1     0     1     0     0
     0     0     0     1     1     1     0
     0     0     0     0     1     2     1
     0     0     0     0     0     1     3
```

The most frequently used case is `wilkinson(21)`. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

See Also

`eig`, `gallery`, `pascal`

Purpose

Open file in appropriate application (Windows)

Syntax

```
winopen(filename)
```

Description

`winopen(filename)` opens `filename` in the appropriate Microsoft Windows application. The `filename` input is a string enclosed in single quotes. The `winopen` function uses the appropriate Windows shell command, and performs the same action as if you double-click the file in the Windows Explorer. If `filename` is not in the current directory, specify the absolute path for `filename`.

Examples

Open the file `thesis.doc`, located in the current directory, in Microsoft Word:

```
winopen('thesis.doc')
```

Open `myresults.html` in your system's default Web browser:

```
winopen('D:/myfiles/myresults.html')
```

See Also

`dos`, `open`, `web`

winqueryreg

Purpose

Item from Microsoft Windows registry

Syntax

```
valnames = winqueryreg('name', 'rootkey', 'subkey')
value = winqueryreg('rootkey', 'subkey', 'valname')
value = winqueryreg('rootkey', 'subkey')
```

Description

`valnames = winqueryreg('name', 'rootkey', 'subkey')` returns all value names in `rootkey\subkey` in a cell array of strings. The first argument is the literal quoted string, `'name'`.

`value = winqueryreg('rootkey', 'subkey', 'valname')` returns the value for value name `valname` in `rootkey\subkey`.

If the value retrieved from the registry is a string, `winqueryreg` returns a string. If the value is a 32-bit integer, `winqueryreg` returns the value as an integer of MATLAB type `int32`.

`value = winqueryreg('rootkey', 'subkey')` returns a value in `rootkey\subkey` that has no value name property.

Note The literal **`name`** argument and the `rootkey` argument are case-sensitive. The `subkey` and `valname` arguments are not.

Remarks

This function works only for the following registry value types:

- strings (REG_SZ)
- expanded strings (REG_EXPAND_SZ)
- 32-bit integer (REG_DWORD)

Examples

Example 1

Get the value of `CLSID` for the MATLAB sample COM control `mwsampctrl.2`:

```
winqueryreg 'HKEY_CLASSES_ROOT' 'mwsamp.mwsampctrl.2\clsid'
```

```
ans =
   {5771A80A-2294-4CAC-A75B-157DCDDD3653}
```

Example 2

Get a list in variable `mousechar` for registry subkey `Mouse`, which is under subkey `Control Panel`, which is under root key HKEY_CURRENT_USER.

```
mousechar = winqueryreg('name', 'HKEY_CURRENT_USER', ...
   'control panel\mouse');
```

For each name in the `mousechar` list, get its value from the registry and then display the name and its value:

```
for k=1:length(mousechar)
   setting = winqueryreg('HKEY_CURRENT_USER', ...
      'control panel\mouse', mousechar{k});
   str = sprintf('%s = %s', mousechar{k}, num2str(setting));
   disp(str)
   end

ActiveWindowTracking = 0
DoubleClickHeight = 4
DoubleClickSpeed = 830
DoubleClickWidth = 4
MouseSpeed = 1
MouseThreshold1 = 6
MouseThreshold2 = 10
SnapToDefaultButton = 0
SwapMouseButtons = 0
```

wk1finfo

Purpose

Determine whether file contains 1-2-3 WK1 worksheet

Syntax

```
[extens, typ] = wk1finfo(filename)
```

Description

`[extens, typ] = wk1finfo(filename)` returns the string 'WK1' in `extens`, and ' 1-2-3 Spreadsheet' in `typ` if the file `filename` contains a readable worksheet. The `filename` input is a string enclosed in single quotes.

Examples

This example returns information on spreadsheet file `matA.wk1`:

```
[extens, typ] = wk1finfo('matA.wk1')

extens =
   WK1
typ =
    123 Spreadsheet
```

See Also

`wk1read`, `wk1write`, `csvread`, `csvwrite`

Purpose

Read Lotus 1-2-3 WK1 spreadsheet file into matrix

Syntax

```
M = wk1read(filename)
M = wk1read(filename,r,c)
M = wk1read(filename,r,c,range)
```

Description

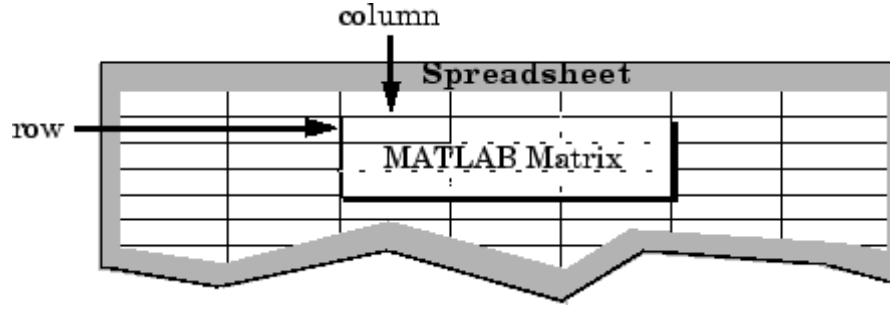
`M = wk1read(filename)` reads a Lotus1-2-3 WK1 spreadsheet file into the matrix `M`. The `filename` input is a string enclosed in single quotes.

`M = wk1read(filename,r,c)` starts reading at the row-column cell offset specified by `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first value in the file.

`M = wk1read(filename,r,c,range)` reads the range of values specified by the parameter `range`, where `range` can be

- A four-element vector specifying the cell range in the format

  ```
  [upper_left_row upper_left_col lower_right_row lower_right_col]
  ```

- A cell range specified as a string, for example, `'A1...C5'`
- A named range specified as a string, for example, `'Sales'`

Examples

Create a 8-by-8 matrix `A` and export it to Lotus spreadsheet `matA.wk1`:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48; 51:58; 61:68; 71:78]
A =
```

```
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48
    51    52    53    54    55    56    57    58
    61    62    63    64    65    66    67    68
    71    72    73    74    75    76    77    78

wk1write('matA.wk1', A);
```

To read in a limited block of the spreadsheet data, specify the upper left row and column of the block using zero-based indexing:

```
M = wk1read('matA.wk1', 3, 2)
M =
    33    34    35    36    37    38
    43    44    45    46    47    48
    53    54    55    56    57    58
    63    64    65    66    67    68
    73    74    75    76    77    78
```

To select a more restricted block of data, you can specify both the upper left and lower right corners of the block you want imported. Read in a range of values from row 4, column 3 (defining the upper left corner) to row 6, column 6 (defining the lower right corner). Note that, unlike the second and third arguments, the range argument `[4 3 6 6]` is one-based:

```
M = wk1read('matA.wk1', 3, 2, [4 3 6 6])
M =
    33    34    35    36
    43    44    45    46
    53    54    55    56
```

See Also

`wk1write`

Purpose

Write matrix to Lotus 1-2-3 WK1 spreadsheet file

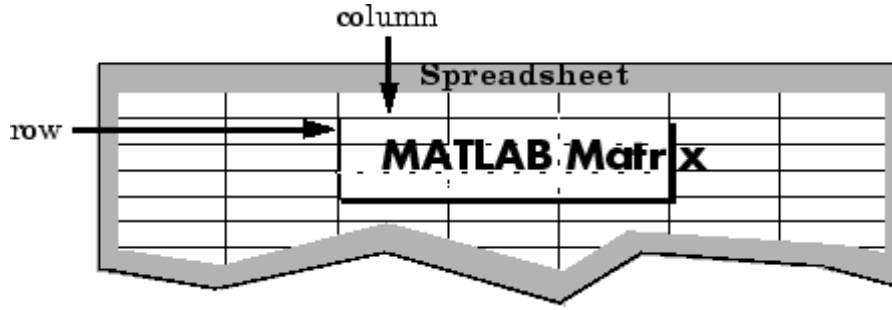
Syntax

```
wk1write(filename,M)
wk1write(filename,M,r,c)
```

Description

`wk1write(filename,M)` writes the matrix `M` into a Lotus1-2-3 WK1 spreadsheet file named `filename`. The `filename` input is a string enclosed in single quotes.

`wk1write(filename,M,r,c)` writes the matrix starting at the spreadsheet location `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first cell in the spreadsheet.



Examples

Write a 4-by-5 matrix A to spreadsheet file matA.wk1. Place the matrix with its upper left corner at row 2, column 3 using zero-based indexing:

```
A = [1:5; 11:15; 21:25; 31:35]
A =
     1     2     3     4     5
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35

wk1write('matA.wk1', A, 2, 3)

M = wk1read('matA.wk1')
M =
```

```
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     1     2     3     4     5
     0     0     0    11    12    13    14    15
     0     0     0    21    22    23    24    25
     0     0     0    31    32    33    34    35
```

See Also

wk1read, dlmwrite, dlmread, csvwrite, csvread

Purpose

Open Workspace browser to manage workspace

GUI Alternatives

As an alternative to the `workspace` function, select **Desktop > Workspace** in the MATLAB desktop.

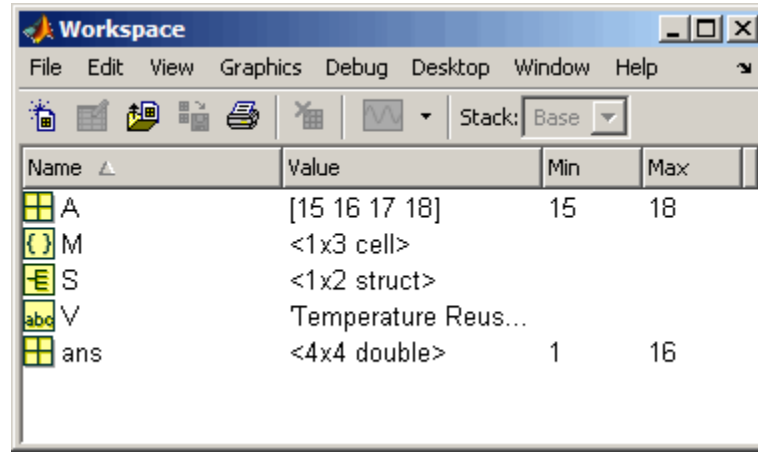
Syntax

```
workspace
```

Description

`workspace` displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the MATLAB workspace. It provides a graphical representation of the `whos` display, and allows you to perform the equivalent of the `clear`, `load`, `open`, and `save` functions.

The Workspace browser also displays and automatically updates statistical calculations for each variable that you can choose to show or hide.



You can edit the value directly in the Workspace browser for small numeric and character arrays. To see and edit a graphical representation of larger variables and for other types, double-click the variable in the Workspace browser. The variable displays in the Array Editor, where you can view the full contents and edit it.

See Also who

Purpose

Label *x*-, *y*-, and *z*-axis

GUI Alternative

To control the presence and appearance of axis labels on a graph, use the Property Editor, one of the plotting tools  . For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
xlabel('string')
xlabel(fname)
xlabel(...,'PropertyName',PropertyValue,...)
xlabel(axes_handle,...)
h = xlabel(...)

ylabel(...)
ylabel(axes_handle,...)
h = ylabel(...)

zlabel(...)
zlabel(axes_handle,...)
h = zlabel(...)
```

Description

Each axes graphics object can have one label for the *x*-, *y*-, and *z*-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

`xlabel('string')` labels the *x*-axis of the current axes.

`xlabel(fname)` evaluates the function `fname`, which must return a string, then displays the string beside the *x*-axis.

`xlabel(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the `text` graphics object created by `xlabel`.

`xlabel(axes_handle,...)`, ylabel(axes_handle,...), and zlabel(axes_handle,...) plot into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = xlabel(...)`, h = ylabel(...), and h = zlabel(...) return the handle to the text object used as the label.

`ylabel(...)` and zlabel(...) label the *y*-axis and *z*-axis, respectively, of the current axes.

Remarks

Reissuing an `xlabel`, `ylabel`, or `zlabel` command causes the new label to replace the old label.

For three-dimensional graphics, MATLAB puts the label in the front or side, so that it is never hidden by the plot.

Examples

Create a multiline label for the *x*-axis using a multiline cell array:

```
xlabel({'first line';'second line'})
```

Create a bold label for the *y*-axis that contains a single quote:

```
ylabel('George''s Popularity','fontsize',12,'fontweight','b')
```

See Also

`strings`, `text`, `title`

“Annotating Plots” on page 1-87 for related functions

“Adding Axis Labels to Graphs” for more information about labeling axes

Purpose

Set or query axis limits

GUI Alternative

To control the upper and lower axis limits on a graph, use the Property Editor, one of the plotting tools  . For details, see The Property Editor in the MATLAB Graphics documentation.

Syntax

```
xlim
xlim([xmin xmax])
xlim('mode')
xlim('auto')
xlim('manual')
xlim(axes_handle,...)
```

Note that the syntax for each of these three functions is the same; only the `xlim` function is used for simplicity. Each operates on the respective *x*-, *y*-, or *z*-axis.

Description

`xlim` with no arguments returns the respective limits of the current axes.

`xlim([xmin xmax])` sets the axis limits in the current axes to the specified values.

`xlim('mode')` returns the current value of the axis limits mode, which can be either `auto` (the default) or `manual`.

`xlim('auto')` sets the axis limit mode to `auto`.

`xlim('manual')` sets the respective axis limit mode to `manual`.

`xlim(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, these functions operate on the current axes.

Remarks

`xlim`, `ylim`, and `zlim` set or query values of the axes object `XLim`, `YLim`, `ZLim`, and `XLimMode`, `YLimMode`, `ZLimMode` properties.

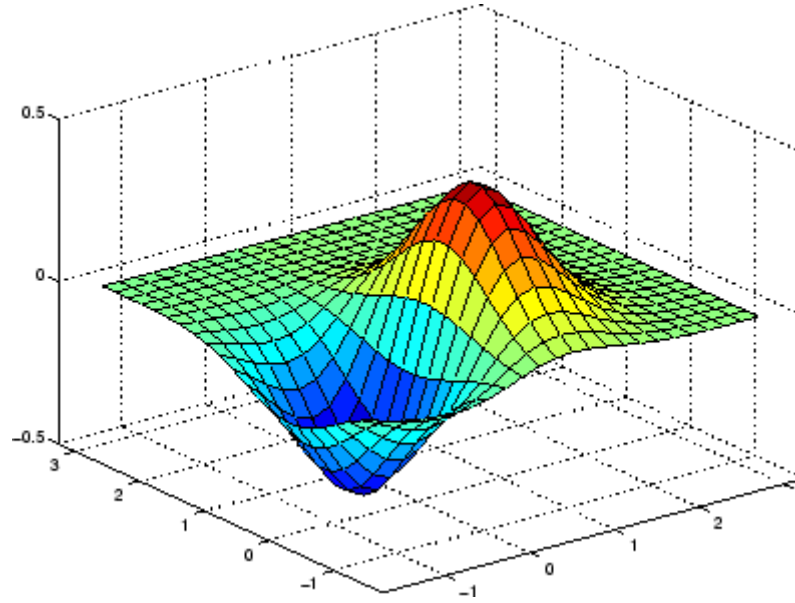
When the axis limit modes are `auto` (the default), MATLAB uses limits that span the range of the data being displayed and are round numbers.

Setting a value for any of the limits also sets the corresponding mode to `manual`. Note that high-level plotting functions like `plot` and `surf` reset both the modes and the limits. If you set the limits on an existing graph and want to maintain these limits while adding more graphs, use the `hold` command.

Examples

This example illustrates how to set the *x*- and *y*-axis limits to match the actual range of the data, rather than the rounded values of `[-2 3]` for the *x*-axis and `[-2 4]` for the *y*-axis originally selected by MATLAB.

```
[x,y] = meshgrid([-1.75:.2:3.25]);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z)
xlim([-1.75 3.25])
ylim([-1.75 3.25])
```



See Also

`axis`

The axes properties `XLim`, `YLim`, `ZLim`

"Setting the Aspect Ratio and Axis Limits" on page 1-100 for related functions

Understanding Axes Aspect Ratio for more information on how axis limits affect the axes

xlsfinfo

Purpose

Determine whether file contains Microsoft Excel (`.xls`) spreadsheet

Syntax

```
typ = xlsfinfo(filename)
[typ, desc] = xlsfinfo(filename)
[typ, desc, fmt] = xlsfinfo(filename)
xlsfinfo filename
```

Description

`typ = xlsfinfo(filename)` returns the string `'Microsoft Excel Spreadsheet'` if the file specified by `filename` is an XLS file that can be read by the MATLAB `xlsread` function. Otherwise, `typ` is the empty string, `''`. The `filename` input is a string enclosed in single quotes.

`[typ, desc] = xlsfinfo(filename)` returns in `desc` a cell array of strings containing the names of each spreadsheet in the file. If a spreadsheet is unreadable, the cell in `desc` that represents that spreadsheet contains an error message.

`[typ, desc, fmt] = xlsfinfo(filename)` returns in the `fmt` output a string containing the actual format of the file as obtained from the Excel COM server. On UNIX systems, or on Windows when the COM server is not available, `fmt` is returned as an empty string, `''`.

Note In the case where an Excel COM server cannot be started, functionality is limited in that some Excel files might not be readable.

`xlsfinfo filename` is the command format for `xlsfinfo`. It returns only the first output, `typ`, assigning it to the MATLAB default variable `ans`.

Examples

Get information about an `.xls` file:

```
[typ, desc, fmt] = xlsfinfo('myaccount.xls')

typ =
    Microsoft Excel Spreadsheet
```

```
desc =
    'Sheet1'    'Income'    'Expenses'

fmt =
    xlWorkbookNormal
```

Export the `.xls` file to comma-separated value (CSV) format. Use `xlsfinfo` to see the format of the exported file:

```
[typ, desc, fmt] = xlsfinfo('myaccount.csv');
fmt

fmt =
    xlCSV
```

Export the `.xls` file to HTML format. `xlsfinfo` returns the following format string:

```
[typ, desc, fmt] = xlsfinfo('myaccount.html');
fmt

fmt =
    xlHtml
```

Export the `.xls` file to XML format. `xlsfinfo` returns the following format string:

```
[typ, desc, fmt] = xlsfinfo('myaccount.xml');
fmt

fmt =
    xlXMLSpreadsheet
```

See Also

`xlsread`, `xlswrite`

xlsread

Purpose

Read Microsoft Excel spreadsheet file (`.xls`)

Syntax

```
num = xlsread(filename)
num = xlsread(filename, -1)
num = xlsread(filename, sheet)
num = xlsread(filename, 'range')
num = xlsread(filename, sheet, 'range')
num = xlsread(filename, sheet, 'range', 'basic')
num = xlsread(filename, ..., functionhandle)
[num, txt]= xlsread(filename, ...)
[num, txt, raw] = xlsread(filename, ...)
[num, txt, raw, X] = xlsread(filename, ..., functionhandle)
xlsread filename sheet range basic
```

Description

`num = xlsread(filename)` returns numeric data in `double` array `num` from the first sheet in the Microsoft Excel spreadsheet file named `filename`. The `filename` argument is a string enclosed in single quotes.

`xlsread` ignores any *outer* rows or columns of the spreadsheet that contain no numeric data. If there are single or multiple nonnumeric rows at the top or bottom, or single or multiple nonnumeric columns to the left or right, `xlsread` does not include these rows or columns in the output. For example, one or more header lines appearing at the top of a spreadsheet are ignored by `xlsread`. Any *inner* rows or columns in which some or all cells contain nonnumeric data are *not* ignored. The nonnumeric cells are instead assigned a value of `NaN`.

The full functionality of `xlsread` depends on the ability to start Excel as a COM server from MATLAB. If your system does not have this capability, the `xlsread` syntax that passes the `'basic'` keyword is recommended. As long as the COM server is available, you can use `xlsread` on Excel files having formats other than XLS (for example, HTML).

Note `xlsread` on UNIX is being grandfathered. If the Excel COM server is not available, `xlsread` reads only strictly XLS files. It cannot read Excel files saved in HTML or other formats.

`num = xlsread(filename, -1)` opens the file `filename` in an Excel window, enabling you to interactively select the worksheet to be read and the range of data on that worksheet to import. To import an entire worksheet, first select the sheet in the Excel window and then click the **OK** button in the Data Selection Dialog box. To import a certain range of data from the sheet, select the worksheet in the Excel window, drag and drop the mouse over the desired range, and then click **OK**. (See "COM Server Requirements" on page 2-3760 below.)

`num = xlsread(filename, sheet)` reads the specified worksheet, where `sheet` is either a positive, double scalar value or a quoted string containing the sheet name. To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`.

`num = xlsread(filename, 'range')` reads data from a specific rectangular region of the default worksheet (`Sheet1`). Specify `range` using the syntax `'C1:C2'`, where `C1` and `C2` are two opposing corners that define the region to be read. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `range` input is not case sensitive and uses Excel A1 notation. (See help in Excel for more information on this notation.) (Also, see "COM Server Requirements" on page 2-3760 below.)

`num = xlsread(filename, sheet, 'range')` reads data from a specific rectangular region (`range`) of the worksheet specified by `sheet`. See the previous two syntax formats for further explanation of the `sheet` and `range` inputs. (See "COM Server Requirements" on page 2-3760 below.)

`num = xlsread(filename, sheet, 'range', 'basic')` imports data from the spreadsheet in basic import mode. This is the mode used on UNIX platforms as well as on Windows when Excel is not available as a COM server. In this mode, `xlsread` does not use Excel as a COM server,

and this limits import ability. Without Excel as a COM server, `range` is ignored and, consequently, the whole active range of a sheet is imported. (You can set `range` to the empty string (`''`)). Also, in basic mode, `sheet` is case-sensitive and must be a quoted string.

`num = xlsread(filename, ..., functionhandle)` calls the function associated with `functionhandle` just prior to obtaining spreadsheet values. This enables you to operate on the spreadsheet data (for example, convert it to a numeric type) before reading it in. (See "COM Server Requirements" on page 2-3760 below.)

You can write your own custom function and pass a handle to this function to `xlsread`. When `xlsread` executes, it reads from the spreadsheet, executes your function on the data read from the spreadsheet, and returns the final results to you. When `xlsread` calls your function, it passes a range interface from Excel to provide access to the data read from the spreadsheet. Your function must include this interface both as an input and output argument. Example 5 below shows how you might use this syntax.

`[num, txt]= xlsread(filename, ...)` returns numeric data in array `num` and text data in cell array `txt`. All cells in `txt` that correspond to numeric data contain the empty string.

If `txt` includes data that was previously written to the file using `xlswrite`, and the `range` specified for that `xlswrite` operation caused undefined data (`'#N/A'`) to be written to the worksheet, then cells containing that undefined data are represented in the `txt` output as `'ActiveX VT_ERROR: '`.

`[num, txt, raw] = xlsread(filename, ...)` returns numeric and text data in `num` and `txt`, and unprocessed cell content in cell array `raw`, which contains both numeric and text data. (See "COM Server Requirements" on page 2-3760 below.)

`[num, txt, raw, X] = xlsread(filename, ..., functionhandle)` calls the function associated with `functionhandle` just prior to reading from the spreadsheet file. This syntax returns one additional output `X` from the function mapped to by `functionhandle`. Example 6 below

shows how you might use this syntax. (See "COM Server Requirements" on page 2-3760 below.)

`xlsread filename sheet range` **`basic`** is the command format for `xlsread`, showing its usage with all input arguments specified. When using this format, you must specify `sheet` as a string, (for example, `Income` or `Sheet4`) and not a numeric index. If the `sheet` name contains space characters, then quotation marks are required around the string, (for example, `'Income 2002'`).

Remarks

Handling Excel Date Values

MATLAB imports date fields from Excel files in the format in which they were stored in the Excel file. If stored in string or date format, `xlsread` returns the date as a string. If stored in a numeric format, `xlsread` returns a numeric date.

Both Excel and MATLAB represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel uses January 1, 1900 as the reference date while MATLAB uses January 0, 0000. Due to this difference in the way Excel and MATLAB compute numeric date values, any numeric date imported from Excel into MATLAB must first be converted before being used in the MATLAB application.

You can do this conversion after the `xlsread` completes, as shown below:

```
excelDates = xlsread(filename)
matlabDates = datenum('30-Dec-1899') + excelDates
datestr(matlabDates,2)
```

You can also do this as part of the `xlsread` operation by writing a conversion routine that acts directly on the Excel COM `Range` object, and then passing a function handle for your routine as an input to `xlsread`. The description above for the following syntax, along with Examples 5 and 6, explain how to do this:

```
[num, txt, raw, X] = xlsread(filename, ..., functionhandle)
```

COM Server Requirements

The following six syntax formats are supported only on computer systems capable of starting Excel as a COM server from MATLAB. They are not supported in `basic` mode.

```
num = xlsread(filename, -1)
num = xlsread(filename, 'range')
num = xlsread(filename, sheet, 'range')
[num, txt, raw] = xlsread(filename, ...)
num = xlsread(filename, ..., functionhandle)
[num, txt, raw, opt] = xlsread(filename, ..., functionhandle)
```

Examples

Example 1— Reading Numeric Data

The Microsoft Excel spreadsheet file `testdata1.xls` contains this data:

1	6
2	7
3	8
4	9
5	10

To read this data into MATLAB, use this command:

```
A = xlsread('testdata1.xls')
A =
     1     6
     2     7
     3     8
     4     9
     5    10
```

Example 2 — Handling Text Data

The Microsoft Excel spreadsheet file `testdata2.xls` contains a mix of numeric and text data:

1	6
2	7

```
    3    8
    4    9
    5   text
```

`xlsread` puts a `NaN` in place of the text data in the result:

```
A = xlsread('testdata2.xls')
A =
    1    6
    2    7
    3    8
    4    9
    5   NaN
```

Example 3 — Selecting a Range of Data

To import only rows `4` and `5` from worksheet `1`, specify the range as `'A4:B5'`:

```
A = xlsread('testdata2.xls', 1, 'A4:B5')

A =
    4    9
    5   NaN
```

Example 4 — Handling Files with Row or Column Headers

A Microsoft Excel spreadsheet labeled `Temperatures` in file `tempdata.xls` contains two columns of numeric data with text headers for each column:

```
Time  Temp
12     98
13     99
14     97
```

If you want to import only the numeric data, use `xlsread` with a single return argument. Specify the filename and sheet name as inputs.

`xlsread` ignores any leading row or column of text in the numeric result.

```
ndata = xlsread('tempdata.xls', 'Temperatures')

ndata =
    12    98
    13    99
    14    97
```

To import both the numeric data and the text data, specify two return values for `xlsread`:

```
[ndata, headertext] = xlsread('tempdata.xls', 'Temperatures')

ndata =
    12    98
    13    99
    14    97

headertext =
    'Time'    'Temp'
```

Example 5 — Passing a Function Handle

This example calls `xlsread` twice, the first time as a simple read from a file, and the second time requesting that `xlsread` execute some user-defined modifications on the data prior to returning the results of the read. These modifications are performed by a user-written function, `setMinMax`, that you pass as a function handle in the call to `xlsread`. When `xlsread` executes, it reads from the spreadsheet, executes the function on the data read from the spreadsheet, and returns the final results to you.

Note The function passed to `xlsread` operates on the copy of the data read from the spreadsheet. It does not modify data in the spreadsheet itself.

Read a 10-by-3 numeric array from Excel spreadsheet `testsheet.xls`. with a simple `xlsread` statement that does not pass a function handle. Note that the values returned range from -587 to +4,149:

```
arr = xlsread('testsheet.xls')
arr =
  1.0e+003 *
    1.0020    4.1490    0.2300
    1.0750    0.1220   -0.4550
   -0.0301    3.0560    0.2471
    0.4070    0.1420   -0.2472
    2.1160   -0.0557   -0.5870
    0.4040    2.9280    0.0265
    0.1723    3.4440    0.1112
    4.1180    0.1820    2.8630
    0.9000    0.0573    1.9750
    0.0163    0.2000   -0.0223
```

In preparation for the second part of this example, write a function `setMinMax` that restricts the values returned from the read to be in the range of 0 to 2000. You will need to pass this function in the call to `xlsread` which will then execute the function on the data it has read before returning it to you.

When `xlsread` calls your function, it passes a range interface from Excel to provide access to the data read from the spreadsheet. This is shown as `DataRange` in this example. Your function must include this interface both as an input and output argument. The output argument allows your function to pass modified data back to `xlsread`:

```
function [DataRange] = setMinMax(DataRange)
maxval = 2000;  minval = 0;

for k = 1:DataRange.Count
   v = DataRange.Value{k};
   if v > maxval || v < minval
      if v > maxval
         DataRange.Value{k} = maxval;
```

```
        else
           DataRange.Value{k} = minval;
        end
     end
  end
```

Now call `xlsread`, passing a function handle for the `setMinMax` function as the final argument. Note the changes from the values returned from the last call to `xlsread`:

```
arr = xlsread('testsheet.xls', '', '', '', @setMinMax)
arr =
  1.0e+003 *
    1.0020    2.0000    0.2300
    1.0750    0.1220         0
         0    2.0000    0.2471
    0.4070    0.1420         0
    2.0000         0         0
    0.4040    2.0000    0.0265
    0.1723    2.0000    0.1112
    2.0000    0.1820    2.0000
    0.9000    0.0573    1.9750
    0.0163    0.2000         0
```

Example 6 — Passing a Function Handle with Additional Output

This example adds onto the previous one by returning an additional output from the call to `setMinMax`. Modify the function so that it not only limits the range of values returned, but also reports which elements of the spreadsheet matrix have been altered. Return this information in a new output argument, `indices`:

```
function [DataRange, indices] = setMinMax(DataRange)
maxval = 2000;  minval = 0;
indices = [];

for k = 1:DataRange.Count
   v = DataRange.Value{k};
```

```
    if v > maxval || v < minval
       if v > maxval
          DataRange.Value{k} = maxval;
       else
          DataRange.Value{k} = minval;
       end
    indices = [indices k];
    end
end
```

When you call `xlsread` this time, account for the three initial outputs, and add a fourth called `idx` to accept the indices returned from `setMinMax`. Call `xlsread` again, and you will see just where the returned matrix has been modified:

```
[arr txt raw idx] = xlsread('testsheet.xls', ...
                            '', '', '', @setMinMax);

idx
idx =
    3   5   8  11  13  15  16  17  22  24  25  28  30
arr
arr =
  1.0e+003 *
    1.0020    2.0000    0.2300
    1.0750    0.1220         0
         0    2.0000    0.2471
    0.4070    0.1420         0
    2.0000         0         0
    0.4040    2.0000    0.0265
    0.1723    2.0000    0.1112
    2.0000    0.1820    2.0000
    0.9000    0.0573    1.9750
    0.0163    0.2000         0
```

See Also

`xlswrite`, `xlsfinfo`, `wk1read`, `textread`, `function_handle`

Purpose

Write Microsoft Excel spreadsheet file (`.xls`)

Syntax

```
xlswrite(filename, M)
xlswrite(filename, M, sheet)
xlswrite(filename, M, 'range')
xlswrite(filename, M, sheet, 'range')
status = xlswrite(filename, ...)
[status, message] = xlswrite(filename, ...)
xlswrite filename M sheet range
```

Description

`xlswrite(filename, M)` writes matrix `M` to the Excel file `filename`. The `filename` input is a string enclosed in single quotes. The input matrix `M` is an `m`-by-`n` numeric, character, or cell array, where `m < 65536` and `n < 256`. The matrix data is written to the first worksheet in the file, starting at cell `A1`.

`xlswrite(filename, M, sheet)` writes matrix `M` to the specified worksheet `sheet` in the file `filename`. The `sheet` argument can be either a positive, double scalar value representing the worksheet index, or a quoted string containing the sheet name.

If `sheet` does not exist, a new sheet is added at the end of the worksheet collection. If `sheet` is an index larger than the number of worksheets, empty sheets are appended until the number of worksheets in the workbook equals `sheet`. In either case, MATLAB generates a warning indicating that it has added a new worksheet.

`xlswrite(filename, M, 'range')` writes matrix `M` to a rectangular region specified by `range` in the first worksheet of the file `filename`. Specify `range` using one of the following quoted string formats:

- A cell designation, such as `'D2'`, to indicate the upper left corner of the region to receive the matrix data.
- Two cell designations separated by a colon, such as `'D2:H4'`, to indicate two opposing corners of the region to receive the matrix data. The range `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet.

The range input is not case sensitive and uses Excel A1 notation. (See help in Excel for more information on this notation.)

The size defined by range should fit the size of M or contain only the first cell, (e.g., 'A2'). If range is larger than the size of M, Excel fills the remainder of the region with #N/A. If range is smaller than the size of M, only the submatrix that fits into range is written to the file specified by filename.

xlswrite(filename, M, sheet, 'range') writes matrix M to a rectangular region specified by range in worksheet sheet of the file filename. See the previous two syntax formats for further explanation of the sheet and range inputs.

status = xlswrite(filename, ...) returns the completion status of the write operation in status. If the write completed successfully, status is equal to logical 1 (true). Otherwise, status is logical 0 (false). Unless you specify an output for xlswrite, no status is displayed in the Command Window.

[status, message] = xlswrite(filename, ...) returns any warning or error message generated by the write operation in the MATLAB structure message. The message structure has two fields:

- message — String containing the text of the warning or error message
- identifier — String containing the message identifier for the warning or error

xlswrite filename M sheet range is the command format for xlswrite, showing its usage with all input arguments specified. When using this format, you must specify sheet as a string (for example, Income or Sheet4). If the sheet name contains space characters, then quotation marks are required around the string (for example, 'Income 2002').

Note The above functionality depends upon having Microsoft Excel as a COM server. In absence of Excel, matrix `M` is written as a text file in Comma-Separated Value (CSV) format. In this mode, the `sheet` and `range` arguments are ignored.

Examples

Example 1 — Writing Numeric Data to the Default Worksheet

Write a 7-element vector to Microsoft Excel file `testdata.xls`. By default, the data is written to cells `A1` through `G1` in the first worksheet in the file:

```
xlswrite('testdata', [12.7 5.02 -98 63.9 0 -.2 56])
```

Example 2 — Writing Mixed Data to a Specific Worksheet

This example writes the following mixed text and numeric data to the file `tempdata.xls`:

```
d = {'Time', 'Temp'; 12 98; 13 99; 14 97};
```

Call `xlswrite`, specifying the worksheet labeled `Temperatures`, and the region within the worksheet to write the data to. The 4-by-2 matrix will be written to the rectangular region that starts at cell `E1` in its upper left corner:

```
s = xlswrite('tempdata.xls', d, 'Temperatures', 'E1')
s =
    1
```

The output status `s` shows that the write operation succeeded. The data appears as shown here in the output file:

```
Time   Temp
  12     98
  13     99
  14     97
```

Example 3 — Appending a New Worksheet to the File

Now write the same data to a worksheet that doesn't yet exist in `tempdata.xls`. In this case, MATLAB appends a new sheet to the workbook, calling it by the name you supplied in the `sheets` input argument, `'NewTemp'`. MATLAB displays a warning indicating that it has added a new worksheet to the file:

```
xlswrite('tempdata.xls', d, 'NewTemp', 'E1')
Warning: Added specified worksheet.
```

If you don't want to see these warnings, you can turn them off using the command indicated in the message above:

```
warning off MATLAB:xlswrite:AddSheet
```

Now try the command again, this time creating another new worksheet, `NewTemp2`. Although the message is not displayed this time, you can still retrieve it and its identifier from the second output argument, `m`:

```
[stat msg] = xlswrite('tempdata.xls', d, 'NewTemp2', 'E1');

msg
msg =
       message: 'Added specified worksheet.'
    identifier: 'MATLAB:xlswrite:AddSheet'
```

See Also

`xlsread`, `xlsfinfo`, `wk1read`, `textread`

Purpose

Parse XML document and return Document Object Model node

Syntax

```
DOMnode = xmlread(filename)
```

Description

`DOMnode = xmlread(filename)` reads a URL or `filename` and returns a Document Object Model node representing the parsed document. The `filename` input is a string enclosed in single quotes. The node can be manipulated by using standard DOM functions.

A properly parsed document displays to the screen as

```
xDoc = xmlread(...)
xDoc =
    [#document: null]
```

Remarks

Find out more about the Document Object Model at the World Wide Web Consortium (W3C) Web site, `http://www.w3.org/DOM/`. For specific information on using Java DOM objects, visit the Sun Web site, `http://www.java.sun.com/xml/docs/api`.

Examples

Example 1

All XML files have a single root element. Some XML files declare a preferred schema file as an attribute of this element. Use the `getAttribute` method of the DOM node to get the name of the preferred schema file:

```
xDoc = xmlread(fullfile(matlabroot, ...
               'toolbox/matlab/general/info.xml'));

xRoot = xDoc.getDocumentElement;
schemaURL = ...
   char(xRoot.getAttribute('xsi:noNamespaceSchemaLocation'))

schemaURL =
   http://www.mathworks.com/namespace/info/v1/info.xsd
```

Example 2

Each `info.xml` file on the MATLAB path contains several `listitem` elements with a `label` and `callback` element. This script finds the callback that corresponds to the label `'Plot Tools'`:

```
infoLabel = 'Plot Tools';
infoCbk = '';
itemFound = false;

xDoc = xmlread(fullfile(matlabroot, ...
               'toolbox/matlab/general/info.xml'));

% Find a deep list of all listitem elements.
allListItems = xDoc.getElementsByTagName('listitem');

% Note that the item list index is zero-based.
for k = 0:allListItems.getLength-1
   thisListItem = allListItems.item(k);
   childNode = thisListItem.getFirstChild;

   while ~isempty(childNode)
      %Filter out text, comments, and processing instructions.
      if childNode.getNodeType == childNode.ELEMENT_NODE
         % Assume that each element has a single
         % org.w3c.dom.Text child.
         childText = char(childNode.getFirstChild.getData);

         switch char(childNode.getTagName)
         case 'label';
            itemFound = strcmp(childText, infoLabel);
         case 'callback' ;
            infoCbk = childText;
         end
      end  % End IF
      childNode = childNode.getNextSibling;
   end  % End WHILE
```

```
   if itemFound
      break;
   else
      infoCbk = '';
   end
end  % End FOR

disp(sprintf('Item "%s" has a callback of "%s".', ...
             infoLabel, infoCbk))
```

Example 3

This function parses an XML file using methods of the DOM node returned by `xmlread`, and stores the data it reads in the `Name`, `Attributes`, `Data`, and `Children` fields of a MATLAB structure:

```
function theStruct = parseXML(filename)
% PARSEXML Convert XML file to a MATLAB structure.
try
   tree = xmlread(filename);
catch
   error('Failed to read XML file %s.',filename);
end

% Recurse over child nodes. This could run into problems
% with very deeply nested trees.
try
   theStruct = parseChildNodes(tree);
catch
   error('Unable to parse XML file %s.');
end


% ----- Subfunction PARSECHILDNODES -----
function children = parseChildNodes(theNode)
% Recurse over node children.
children = [];
if theNode.hasChildNodes
```

```
   childNodes = theNode.getChildNodes;
   numChildNodes = childNodes.getLength;
   allocCell = cell(1, numChildNodes);

   children = struct(             ...
      'Name', allocCell, 'Attributes', allocCell,    ...
      'Data', allocCell, 'Children', allocCell);

    for count = 1:numChildNodes
        theChild = childNodes.item(count-1);
        children(count) = makeStructFromNode(theChild);
    end
end

% ----- Subfunction MAKESTRUCTFROMNODE -----
function nodeStruct = makeStructFromNode(theNode)
% Create structure of node info.

nodeStruct = struct(                        ...
   'Name', char(theNode.getNodeName),       ...
   'Attributes', parseAttributes(theNode),  ...
   'Data', '',                              ...
   'Children', parseChildNodes(theNode));

if any(strcmp(methods(theNode), 'getData'))
   nodeStruct.Data = char(theNode.getData);
else
   nodeStruct.Data = '';
end

% ----- Subfunction PARSEATTRIBUTES -----
function attributes = parseAttributes(theNode)
% Create attributes structure.

attributes = [];
if theNode.hasAttributes
   theAttributes = theNode.getAttributes;
```

```
        numAttributes = theAttributes.getLength;
        allocCell = cell(1, numAttributes);
        attributes = struct('Name', allocCell, 'Value', ...
                            allocCell);

        for count = 1:numAttributes
           attrib = theAttributes.item(count-1);
           attributes(count).Name = char(attrib.getName);
           attributes(count).Value = char(attrib.getValue);
        end
     end
```

See Also

xmlwrite, xslt

Purpose

Serialize XML Document Object Model node

Syntax

```
xmlwrite(filename, DOMnode)
str = xmlwrite(DOMnode)
```

Description

`xmlwrite(filename, DOMnode)` serializes the Document Object Model node `DOMnode` to the file specified by `filename`. The `filename` input is a string enclosed in single quotes.

`str = xmlwrite(DOMnode)` serializes the Document Object Model node `DOMnode` and returns the node tree as a string, `s`.

Remarks

Find out more about the Document Object Model at the World Wide Web Consortium (W3C) Web site, `http://www.w3.org/DOM/`. For specific information on using Java DOM objects, visit the Sun Web site, `http://www.java.sun.com/xml/docs/api`.

Example

```
% Create a sample XML document.
docNode = com.mathworks.xml.XMLUtils.createDocument...
    ('root_element')
docRootNode = docNode.getDocumentElement;
for i=1:20
    thisElement = docNode.createElement('child_node');
    thisElement.appendChild...
        (docNode.createTextNode(sprintf('%i',i)));
    docRootNode.appendChild(thisElement);
end
docNode.appendChild(docNode.createComment('this is a comment'));

% Save the sample XML document.
xmlFileName = [tempname,'.xml'];
xmlwrite(xmlFileName,docNode);
edit(xmlFileName);
```

See Also

`xmlread`, `xslt`

Purpose

Logical exclusive-OR

Syntax

```
C = xor(A, B)
```

Description

`C = xor(A, B)` performs an exclusive OR operation on the corresponding elements of arrays `A` and `B`. The resulting element `C(i,j,...)` is logical true (`1`) if `A(i,j,...)` or `B(i,j,...)`, but not both, is nonzero.

A	B	C
Zero	Zero	0
Zero	Nonzero	1
Nonzero	Zero	1
Nonzero	Nonzero	0

Examples

Given `A = [0 0 pi eps]` and `B = [0 -2.4 0 1]`, then

```
C = xor(A,B)
C =
   0   1   1   0
```

To see where either `A` or `B` has a nonzero element and the other matrix does not,

```
spy(xor(A,B))
```

See Also

`all`, `any`, `find`, Elementwise Logical Operators, Short-Circuit Logical Operators

Purpose

Transform XML document using XSLT engine

Syntax

```
result = xslt(source, style, dest)
[result,style] = xslt(...)
xslt(...,'-web')
```

Description

`result = xslt(source, style, dest)` transforms an XML document using a stylesheet and returns the resulting document's URL. The function uses these inputs, the first of which is required:

- `source` is the filename or URL of the source XML file. `source` can also specify a DOM node.
- `style` is the filename or URL of an XSL stylesheet.
- `dest` is the filename or URL of the desired output document. If `dest` is absent or empty, the function uses a temporary filename. If `dest` is `'-tostring'`, the function returns the output document as a MATLAB string.

`[result,style] = xslt(...)` returns a processed stylesheet appropriate for passing to subsequent XSLT calls as `style`. This prevents costly repeated processing of the stylesheet.

`xslt(...,'-web')` displays the resulting document in the Help Browser.

Remarks

Find out more about XSL stylesheets and how to write them at the World Wide Web Consortium (W3C) web site, `http://www.w3.org/Style/XSL/`.

Example

This example converts the file `info.xml` using the stylesheet `info.xsl`, writing the output to the file `info.html`. It launches the resulting HTML file in the Help Browser. MATLAB has several `info.xml` files that are used by the **Start** menu.

```
xslt info.xml info.xsl info.html -web
```

See Also

xmlread, xmlwrite

Purpose

Create array of all zeros

Syntax

```
B = zeros(n)
B = zeros(m,n)
B = zeros([m n])
B = zeros(m,n,p,...)
B = zeros([m n p ...])
B = zeros(size(A))
zeros(m, n,...,classname)
zeros([m,n,...],classname)
```

Description

`B = zeros(n)` returns an `n`-by-`n` matrix of zeros. An error message appears if `n` is not a scalar.

`B = zeros(m,n)` or `B = zeros([m n])` returns an `m`-by-`n` matrix of zeros.

`B = zeros(m,n,p,...)` or `B = zeros([m n p ...])` returns an `m`-by-`n`-by-`p`-by-`...` array of `zeros`.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`B = zeros(size(A))` returns an array the same size as `A` consisting of all zeros.

`zeros(m, n,...,classname)` or `zeros([m,n,...],classname)` is an `m`-by-`n`-by-... array of zeros of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Example

```
x = zeros(2,3,'int8');
```

Remarks

The MATLAB language does not have a dimension statement; MATLAB automatically allocates storage for matrices. Nevertheless, for large

matrices, MATLAB programs may execute faster if the `zeros` function is used to set aside storage for a matrix whose elements are to be generated one at a time, or a row or column at a time. For example

```
x = zeros(1,n);
for i = 1:n, x(i) = i; end
```

See Also

`eye`, `ones`, `rand`, `randn`, `complex`

Purpose

Compress files into zip file

Syntax

```
zip(zipfile,files)
zip(zipfile,files,rootdir)
entrynames = zip(...)
```

Description

`zip(zipfile,files)` creates a zip file with the name `zipfile` from the list of files and directories specified in `files`. Relative paths are stored in the zip file, but absolute paths are not. Directories recursively include all of their content.

`zipfile` is a string specifying the name of the zip file. The `.zip` extension is appended to `zipfile` if omitted.

`files` is a string or cell array of strings containing the list of files or directories included in `zipfile`. Individual files that are on the MATLAB path can be specified as partial pathnames. Otherwise an individual file can be specified relative to the current directory or with an absolute path. Directories must be specified relative to the current directory or with absolute paths. On UNIX systems, directories can also start with `~/` or `~`*`username`*`/`, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character `*` can be used when specifying files or directories, except when relying on the MATLAB path to resolve a filename or partial pathname.

`zip(zipfile,files,rootdir)` allows the path for `files` to be specified relative to `rootdir` rather than the current directory.

`entrynames = zip(...)` returns a string cell array of the relative path entry names contained in `zipfile`.

Examples

Zip a File

Create a zip file of the file `guide.viewlet`, which is in the `demos` directory of MATLAB. It saves the zip file in `d:/mymfiles/viewlet.zip`.

```
file = fullfile(matlabroot,'demos','guide.viewlet');
zip('d:/mymfiles/viewlet.zip',file)
```

Run `zip` for the files `guide.viewlet` and `import.viewlet` and save the zip file in `viewlets.zip`. The source files and zipped file are in the current directory.

```
zip('viewlets.zip',{'guide.viewlet','import.viewlet'})
```

Zip Selected Files

Run `zip` for all `.m` and `.mat` files in the current directory to the file `backup.zip`:

```
zip('backup',{'*.m','*.mat'});
```

Zip a Directory

Run `zip` for the directory `D:/mymfiles` and its contents to the zip file `mymfiles` in the directory one level up from the current directory.

```
zip('../mymfiles','D:/mymfiles')
```

Run `zip` for the files `thesis.doc` and `defense.ppt`, which are located in `d:/PhD`, to the zip file `thesis.zip` in the current directory.

```
 zip('thesis.zip',{'thesis.doc','defense.ppt'},'d:/PhD')
```

See Also

`gzip`, `gunzip`, `tar`, `untar`, `unzip`

Purpose

Turn zooming on or off or magnify by factor

GUI Alternatives

Use the **Zoom** tools  on the figure toolbar to zoom in or zoom out on a plot, or select **Zoom In** or **Zoom Out** from the figure's **Tools** menu. For details, see "Enlarging the View" in the MATLAB Graphics documentation.

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig, option)
h = zoom(figure_handle)
```

Description

`zoom on` turns on interactive zooming. When interactive zooming is enabled in a figure, pressing a mouse button while your cursor is within an axes zooms into the point or out from the point beneath the mouse. Zooming changes the axes limits. When using zoom mode, you

- Zoom in by positioning the mouse cursor where you want the center of the plot to be and either
  - Press the mouse button or
  - Rotate the mouse scroll wheel away from you (upward).
- Zoom out by positioning the mouse cursor where you want the center of the plot to be and either
  - Simultaneously press **Shift** and the mouse button, or
  - Rotate the mouse scroll wheel toward you (downward).

Each mouse click or scroll wheel click zooms in or out by a factor of 2.

Clicking and dragging over an axes when zooming in is enabled draws a rubberband box. When you release the mouse button, the axes zoom in to the region enclosed by the rubberband box.

Double-clicking over an axes returns the axes to its initial zoom setting in both zoom-in and zoom-out modes.

`zoom off` turns interactive zooming off.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive `zoom` mode is enabled, will return to this zoom level.

`zoom` toggles the interactive zoom status between off and on (restoring the most recently used zoom tool).

`zoom xon` and `zoom yon` set `zoom on` for the *x*- and *y*-axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by `1/factor`.

`zoom(fig, option)` Any of the preceding options can be specified on a figure other than the current figure using this syntax.

`h = zoom(figure_handle)` returns a zoom *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Zoom Mode Objects

Access the following properties of zoom mode objects via `get` and modify some of them using `set`:

`Enable` `'on'|'off'`

Specifies whether this figure mode is currently enabled on the figure.

`FigureHandle <handle>`

The associated figure handle. This read-only property cannot be set.

`Motion` `'horizontal'|'vertical'|'both'`

The type of zooming enabled for the figure.

Direction `'in'|'out'`

The direction of the zoom operation.

RightClickAction `'InverseZoom'|'PostContextMenu'`

The behavior of a right-click action. A value of `'InverseZoom'` causes a right-click to zoom out. A value of `'PostContextMenu'` displays a context menu. This setting persists between MATLAB sessions.

`ButtonDownFilter <function_handle>`

The application can inhibit the zoom operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

```
function [res] = myfunction(obj,event_obj)
% OBJ        handle to the object that has been clicked on.
% EVENT_OBJ  handle to event object (empty in this release).
% RES        a logical flag to determine whether the zoom
%            operation should take place or the 'ButtonDownFcn'
%            property of the object should take precedence.
```

`ActionPreCallback <function_handle>`

Set this callback to listen to when a zoom operation starts. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on.
% event_obj   handle to event object.
```

The event object has the following read-only property:

`Axes`	The handle of the axes that is being zoomed

`ActionPostCallback <function_handle>`

Set this callback to listen to when a zoom operation finishes. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on.
% event_obj   handle to event object. The object has the same
%             properties as the event_obj of the
%             'ActionPreCallback' callback.
```

`UIContextMenu <handle>`

Specifies a custom context menu to be displayed during a right-click action. This property is ignored if the `'RightClickZoomOut'` property has been set to `'on'`.

`flags = isAllowAxesZoom(h,axes)`

Calling the function `isAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a zoom operation is permitted on the axes objects.

`setAllowAxesZoom(h,axes,flag)`

Calling the function `setAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a zoom operation on the axes objects.

`info` `= getAxesZoomMotion(h,axes)`

Calling the function `getAxesZoomMotion` on the zoom object, `H`, with a vector of axes handles, `AXES`, as input returns a character cell array of the same dimension as the axes handle vector, which indicates the type of zoom operation for each axes. Possible values for the type of operation are `'horizontal'`, `'vertical'`, or `'both'`.

`setAxesZoomMotion(h,axes,style)`

Calling the function `setAxesZoomMotion` on the zoom object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, ses the style of zooming on each axes.

Examples

Example 1 — Entering Zoom Mode

Plot a graph and turn on Zoom mode:

```
plot(1:10);
zoom on
% zoom in on the plot
```

Example 2 — Constrained Zoom

Create zoom mode object and constrain to *x*-axis zooming:

```
plot(1:10);
 h = zoom;
set(h,'Motion','horizontal','Enable','on');
% zoom in on the plot in the horizontal direction.
```

Example 3 — Constrained Zoom in Subplots

Create four axes as subplots and set zoom style differently for each by setting a different property for each axes handle:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = zoom;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesZoom(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesZoomMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesZoomMotion(h,ax4,'vertical');
```

```
% Zoom in on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for zoom mode objects to trigger. Copy the following code to a new M-file, execute it, and observe zooming behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = zoom;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-buttonDown events for zoom mode objects to trigger. Copy the following code to a new M-file, execute it, and observe zoom behavior:

```
function demo
% Listen to zoom events
plot(1:10);
h = zoom;
set(h,'ActionPreCallback',@myprecallback);
```

```
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A zoom is about to occur.');
%
function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%.2f %.2f].',newLim));
```

Example 6 — Creating a Context Menu for Zoom Mode

Coding a context menu that lets the user to switch to Pan mode by right-clicking:

```
figure;plot(magic(10))
hCMZ = uicontextmenu;
hZMenu = uimenu('Parent',hCMZ,'Label','Switch to pan','Callback','p
hZoom = zoom(gcf);
set(hZoom,'UIContextMenu',hCMZ);
zoom('on')
```

You cannot add items to the built-in zoom context menu, but you can replace it with your own.

Remarks

`zoom` changes the axes limits by a factor of 2 (in or out) each time you press the mouse button while the cursor is within an axes. You can also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

You can create a zoom mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

When you assign different zoom behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over

to the linked axes, regardless of the behavior you previously set for the other axes.

See Also

`linkaxes`, `pan`, `rotate3d`

"Object Manipulation" on page 1-100 for related functions

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z