# Matlab Tutorial

© 2003 Information Technology Services, UT-Austin

Part I: Getting Started .......... 4

Section 1: Introduction .......... 4
About this Document .......... 4
Introduction to Matlab .......... 5

Section 2: An Overview of Matlab .......... 5
Access to Matlab and ITS Matlab consulting services .......... 5
Getting Started .......... 6
The Desktop Layout .......... 7
The Workspace Window .......... 8
The Current Directory Window .......... 8
The Launch Pad Window .......... 8
The Command History Window .......... 8
The Command Window .......... 9
The Figure Window .......... 10

Section 3: Importing and Exporting Information .......... 10
Command Line Import .......... 11
The Import Wizard .......... 11
Import Functions .......... 13
csvread .......... 13
dlmread .......... 14
load .......... 14
fscanf .......... 14
textread .......... 15
Export Functions .......... 16
diary .......... 16
dlmwrite .......... 17
save .......... 17
fprintf .......... 18
M-file Scripts .......... 19
M-Books .......... 19
Advanced Methods for Binary Information .......... 22

Section 4: Notation, Syntax, and Operations .......... 23
Variable names .......... 23
Numerical conventions .......... 23
Geometrical and directional conventions .......... 24
Operator and delimiter symbolics .......... 24

Part II: Computing and Programming .......... 27

Section 5: Computational Procedures .......... 27
Special Built-in Constants .......... 27

Special Built-in Functions ..... 28
Compound Expressions and Operator Precedence ..... 30
Commutivity of Operations and Finite Decimal Expansion Approximations ..... 32
Computing with matrices and vectors ..... 32
Simultaneous linear equations ..... 33
Eigenvectors and Eigenvalues ..... 34
Poles, residues, and partial fraction expansion ..... 37
Convolution and deconvolution ..... 38
Finite Fourier and Inverse Fourier Transforms ..... 39
Numerical Differentiation and Integration ..... 39
Numerical solution of differential equations ..... 42

Section 6: Programming ..... 42
Using the Editor ..... 42
Types and Structures of M-files ..... 43
Internal Documentation ..... 45
Passing variables by name and value ..... 46
Function evaluation and function handles ..... 47
Function recursion ..... 48
Flow control ..... 49
String evaluation and manipulation ..... 51
Keyboard input ..... 53
Multidimensional arrays and indexing ..... 54
Debugging ..... 56
Using Matlab with External Code ..... 59
Exchanging and viewing text information ..... 59
Compiling and calling external files from Matlab ..... 60
Calling Matlab objects from external progams ..... 62
Using Java Classes in Matlab ..... 62

**Part III: Graphics and Data Analysis ..... 63**

Section 7: Graphics and Data Visualization ..... 64
Two dimensional plotting ..... 64
Three dimensional plotting ..... 65
Animation ..... 68
The Handle Graphics system ..... 68
Customizing displays ..... 69

Section 8: Data Analysis ..... 72
Data analysis functions ..... 72
Regression and curve fitting ..... 73
Signal and image processing ..... 75

**Part IV: Modeling and Simulation ..... 81**

Section 9: Modeling and Simulation ..... 81
System Identification ..... 81
Using the Control System Toolbox ..... 81

Optimization Toolbox ........................................ 84
Using Simulink........................................ 87
Simulink Library Browser ........................................ 88
Construction/ Simulation of Dynamical Systems ........................................ 92

# Part I: Getting Started

## Section 1: Introduction

***About this Document***

This document introduces you to the MatlabR13 suite of applications from Mathworks, Inc. The R13 release consists of version 6.5 of the primary Matlab application along with some auxiliary modeling and simulation applications and specialized toolboxes. The suite as a whole will be surveyed but the primary application, Matlab 6.5, will be the focus of this tutorial. Instruction is aimed toward first-time users; however, those who are already familiar with previous versions of Matlab can use this document to learn about some of Matlab's new features and graphical interface. The document is organized into four parts containing a total of nine sections.

Part I includes the first four sections and serves to get the user acquainted with the Matlab application. The first section provides a brief introduction to this tutorial and to Matlab. The second section gives an overview of the Matlab desktop layout and guides the reader through each of the windows and their functions. Also covered in this section is the layout for the built-in Matlab Help. The third section covers the basic methods of getting data into and exporting data from Matlab, including the import wizard, command line I/O functions, and M-books. The fourth section introduces the notation and syntax used by Matlab and contains essential information for routine usage of the application.

Part II includes the fifth and sixth sections which serve to introduce the basic functionalities of the Matlab application such that the user will be able to perform routine tasks. The fifth section covers computational and numerical methods for doing various mathematical operations. The sixth section goes through the basics of programming in the Matlab programming language and the construction of M-files, which are callable scripts, macros, and functions that can be used from the Matlab command line prompt or can be nested within another such file.

Part III includes the seventh and eighth sections and covers aspects of manipulating and visualizing properties of data sets, including experimental data that have no assumed functional relationships *a priori*. The seventh section treats aspects of graphical display and data visualization, particularly plots. An eighth section covers several ways in which Matlab can be used in the analysis of experimental or collected data.

Part IV encompasses the ninth and final section which deals with optimization methods and with modeling and simulation of dynamical systems. This last section includes an introduction to the auxiliary Simulink application that is distributed with Matlab. Also, this section will introduce the usage of Matlab Toolboxes, which are ensembles of related M-files developed for specific types of applications.

*Introduction to Matlab*

In the 1960s and 1970s before the appearance of personal computers, complex and large scale calculations were done on large mainframes using code primarily developed with Fortran. As a number of related large subroutines were developed for specific computational purposes, they were organized into public domain packages and distributed for free. Matlab was originally created as a front end for one of these, the LINPACK package -- a group of routines for working with matrices and linear algebra. The primary developer, Professor Cleve Moler at the University of New Mexico, eventually founded Mathworks, Inc., to further develop and market the product in a commercial setting. From the original Matlab, a high powered suite of applications has evolved. The current generation release, the MatlabR13 suite, features the newest kernel, Matlab 6.5. It is largely backward compatible with recent Matlab versions, but there are some slight syntax changes. The biggest changes that users of MatlabR11 and earlier versions will notice are the implementation of a desktop layout format for interactive use and differences in the handle graphics that now accommodate several point and click features for labeling and enhancing plots and graphical output.

## Section 2: An Overview of Matlab

*Access to Matlab and ITS Matlab consulting services*

Information Technology Services (ITS) at UT-Austin provides free Matlab 6.5 access to students, faculty, and staff on its time sharing mainframes, or via the desktop computers in the Student Microcomputer Facility (SMF) on the second floor of the Flawn Academic Center (FAC 212). Before using these machines it will be necessary to have an individually funded (IF) or departmental sponsored account issued by ITS. There is no cost for using an IF or departmental account in the SMF, although charges may be incurred if optional validations for internet dialup service or excess mainframe disk storage allocation is desired. Information about getting an ITS- issued account is available at

http://www.utexas.edu/its/account/index.html

Matlab 6.5 is installed on the ITS Unix systems: ccwf.cc.utexas.edu and larry.cc.utexas.edu which run on Solaris 8. A Windows version of Matlab 6.5 is installed on the ITS applications Windows terminal server, earthquake.cc.utexas.edu. To access these installations, an ITS-issued account must have HPW validation (ccwf), UTS validation (larry), or WNT validation (earthquake). Some academic departments (e.g., Chemical Engineering) also have Matlab already installed on machines in their local computer labs. Also, UT-Austin departments are eligible to purchase licenses for Matlab for use within their own labs through the Software Distribution Services unit of ITS. Details of various licensing options available for purchase by academic departments are available at

http://www.utexas.edu/its/sds/products/matlab.html

The MatlabR13 suite containing Matlab 6.5 is also marketed directly by Mathworks, Inc., which has special academic pricing; and, for individuals enrolled as students, a very inexpensive student edition with full functionality is available. Information and pricing is given at

http://www.mathworks.com/store/index.html

The Campus Computer Store in VRC also sells the student version of Matlab. A currently valid ***student*** ID is needed for purchase.

ITS provides free consulting for UT-Austin student, faculty, and staff members using Matlab for education and research purposes. However, direct assistance with homework, class assignments, or projects of a commercial nature is not available. Appointments for walk-in or phone consulting can be made at

http://www.utexas.edu/cc/amicus/index.html

and questions can be submitted via the web form

http://www.utexas.edu/its/rc/ehelp-request.html

or by direct e-mail to math@its.utexas.edu.

***Getting Started***

The procedure for launching Matlab 6.5 is no different from that used with earlier versions. However, the launch method varies.

With Unix, the environment PATH variable must include the directory containing the Matlab installation. On the ITS Unix systems this directory is /usr/local/matlab and there is a script that will automatically augment the PATH variable with necessary additions. This script can be run from a shell prompt with the command `eval`/usr/local/etc/appuser``. Please note that backquotes should be used rather than ordinary single quotes.
If the Matlab application is being run on a remote X terminal using a Unix mainframe application, then the DISPLAY variable has to be set to the local terminal and the local terminal has to grant access to the Unix mainframe. Both of these requirements can be circumvented if the mainframe connection is through a secure shell (SSH) protocol. Otherwise the commands

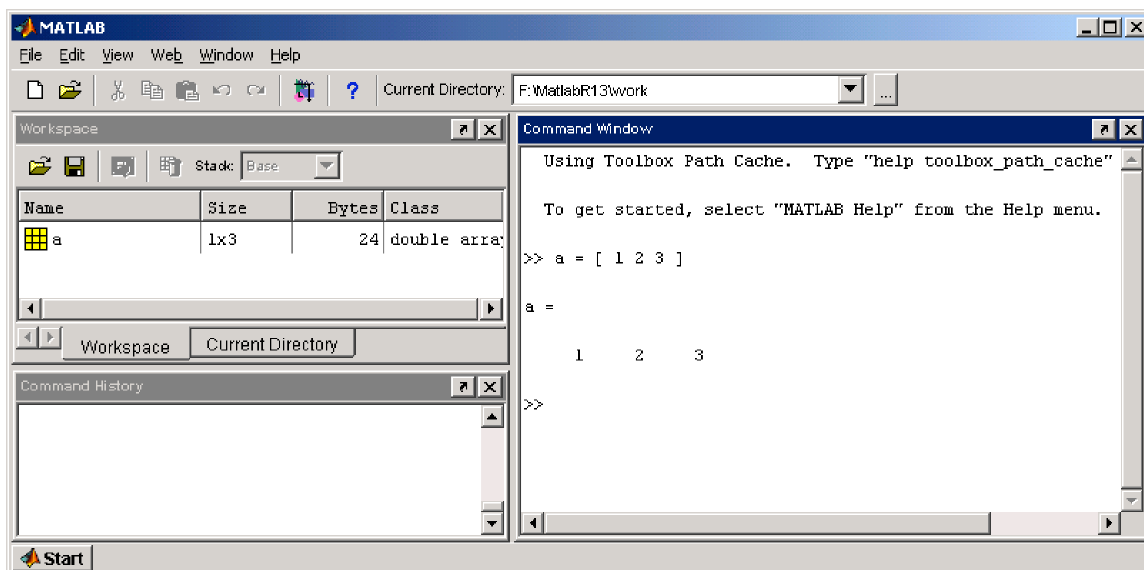
```
setenv DISPLAY myhost.mydomain:0.0
xhost +remotesystem.remotedomain
```

will need to be executed at the shell prompt. Matlab can then be started at a shell prompt with the command `matlab`, or, in background with concurrent access to other processes with the command `matlab &`.

The Windows/NT distribution can be launched by double clicking on a Matlab icon or shortcut. The default working path in this distribution consists of the Work subfolder of the top level installation folder and the hierarchy within the Toolbox subfolder of the top level installation folder. Access to files in other folders can be set by navigating with the browser from the Set Path option of the File pull down menu.

***The Desktop Layout***

Once the path environment is set properly and the desktop appears on the monitor screen, using Matlab 6.5 will be practically the same for all operating system distributions. There are several windows that can be used in arranging the desktop, but a default configuration will appear upon launching. If desired this can be modified by selecting an alternate choice from View -> Desktop Layout . This tutorial will use the layout for MatlabR13 package, which is currently the ITS supported production version. The default desktop for MatlabR13 has a small Current Directory selection window near the top, with the `Work` folder being specified initially. If other folders have been added, then another current directory setting can be selected from among those available. The main area has the Command window on the right and an upper and lower pair of toggled windows on the left side. The upper left panel will show the Workspace window which can be toggled with a Current Directory window. The lower left panel will show a Command History window. A Launch Pad Window toggle can be added to the docked upper left area by selecting it from *View -> Launch Pad* on the top navigation bar. There are also remote `Help` and `Demo` windows that can be launched from the Desktop's `Help` pull down menu, and there is a remote `Figure` window that is launched whenever a command involving graphical display is executed.



***The Workspace Window***

The Workspace window provides an inventory of all the items in the workspace that are currently defined, either by assignment or calculation in the Command window or by importation with a *load* command from the Matlab command line prompt. By default it is displayed in the upper left area of toggled windows when the application is first launched or when *View -> Desktop Layout -> Default* is selected from the navigation bar.

***The Current Directory Window***

. The Current Directory window displays a current directory with a listing of its contents. There is navigation capability for resetting the current directory to any directory among those set in the path. This window is useful for finding the location of particular files and scripts so that they can be edited, moved, renamed, deleted, etc. The default current directory is the `Work` subdirectory of the original Matlab installation directory.

***The Launch Pad Window***

The Launch Pad window, displayed in the upper left of the desktop configuration when toggling after addition with *View -> Launch Pad* from the navigation bar, serves as a convenient assemblage of shortcuts to common operations and windows that may not currently be displayed. For example, the Help window can be launched by clicking on the icon here rather than using the navigation bar pull-down menu. A Demo window for demonstrations can be launched from here, as can auxiliary applications such as Simulink.

***The Command History Window***

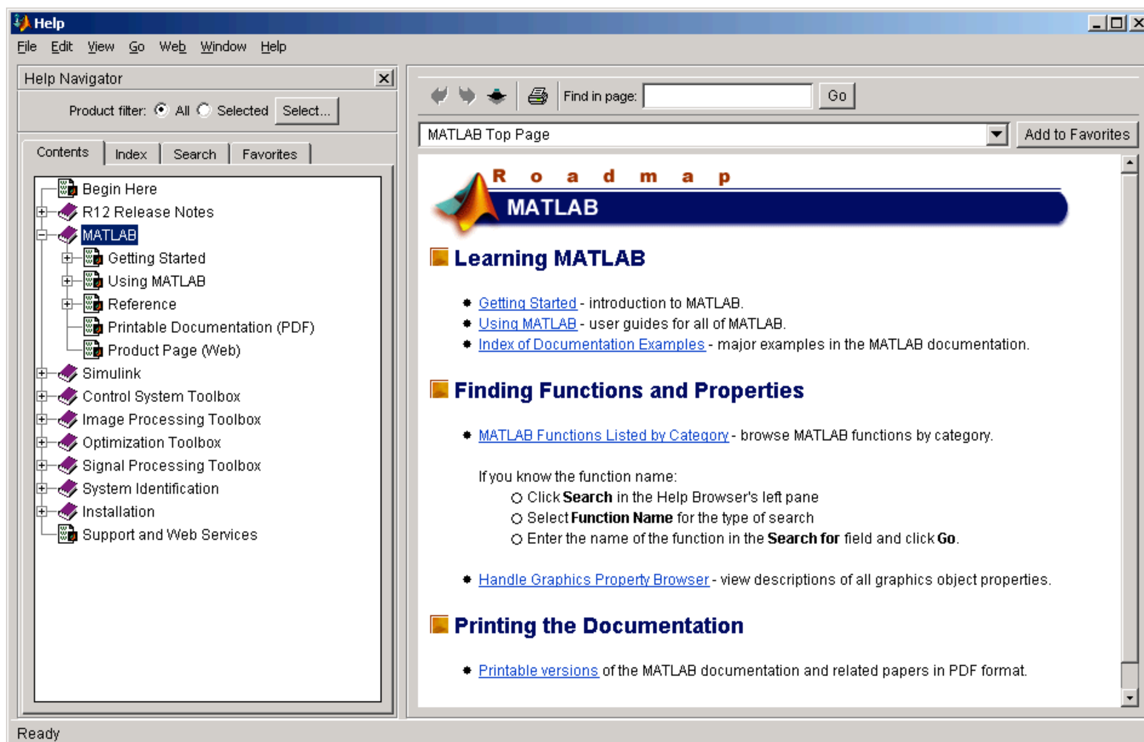
The Command History window, at the lower left in the default desktop, contains a log of commands that have been executed within the `Command` window. This is a convenient feature for tracking when developing or debugging programs or to confirm that commands were executed in a particular sequence during a multistep calculation from the command line.

### *The Command Window*

. The Command window is where the command line prompt for interactive commands is located. This is also the only window that appears if you execute the Unix version of Matlab outside of an X environment, e.g., on a vt100 screen. Commands and scripts can be executed from a vt100 window, but graphics and desktop tools will not be available. The Matlab prompt on the command window consists of two adjacent right angle brackets, i.e., **>>**. Results of command operations will also be displayed in this window unless the command line is terminated by a semi-colon, in which case the display of results is suppressed. If a command or script specified on the command line is questionable or cannot be executed because of invalid syntax, undefined variables, etc., a diagnostic message will be displayed in the `Command` window. The current value of any saved variable is also displayed in this window if its name is entered at a prompt.

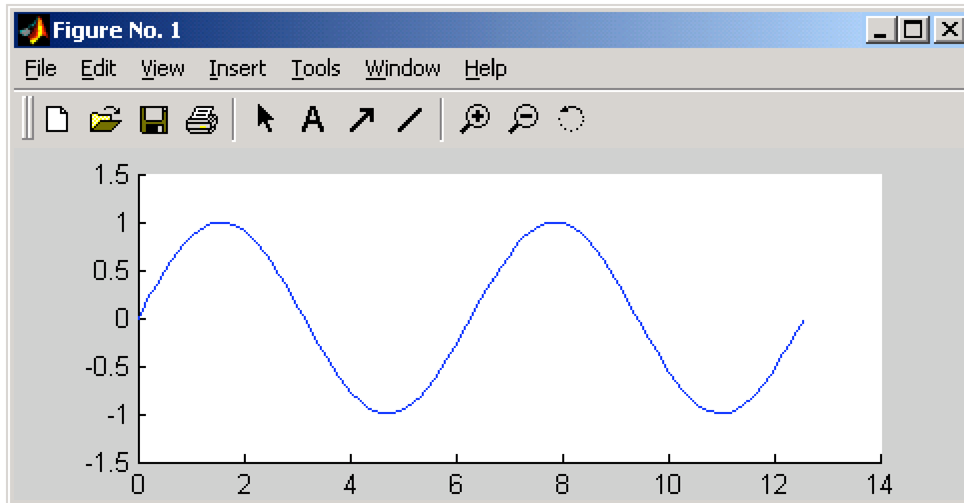
### *The Help Window*

Separate from the main desktop layout is a Help desktop with its own layout. This utility can be launched by selecting *Help ->MATLAB Help* from the Help pull down menu. This Help desktop has a right side which displays the text for help topics, organized as tutorial of sorts. The left side has various tabs that can be brought to the foreground for navigating by table of contents, by indexed keywords, or by a search on a particular string.



***The Figure Window***

There is a Figure window that floats independently from the main desktop. If not already present, it is launched when command execution results in graphical output. From the *Edit* menu on the main toolbar, there are selections for editing figure properties, axis properties, and properties of objects within figures. On the Tools menu of the main toolbar there are selections for further manipulation such as zooming and perspective rotation. There is also a main toolbar menu for *Help*, which includes specific graphics help and demos.




## Section 3: Importing and Exporting Information

Information can be imported into and exported from the Matlab application by several different methods. Some of the more common procedures are discussed below. To follow the specific examples given, copies of the example external data files will need to be copied to a location where they can be accessed by the Matlab program. Thus, before proceeding with this section, find the following documents

http://www.utexas.edu/cc/math/tutorials/matlab6/planetsize.txt
http://www.utexas.edu/cc/math/tutorials/matlab6/planets1.txt
http://www.utexas.edu/cc/math/tutorials/matlab6/planets2.txt
http://www.utexas.edu/cc/math/tutorials/matlab6/planets3.txt
http://www.utexas.edu/cc/math/tutorials/matlab6/planetdata.doc
http://www.utexas.edu/cc/math/tutorials/matlab6/planets6.xls

and copy them into a location within the Matlab search path (for example, in the default work folder or directory). If you cannot access these external documents, you can also find their ascii text contents within this document. Remember that the planetdata.doc and plantets6.xls files are binary and must be transferred in that mode

***Command Line Import***

The most elementary method for importing external information is piece by piece directly from the command line by typing at the keyboard. For example:

```
>> earthradius = 6371
```

will assign the numerical value 6371 to a variable *earthradius*. For extensive amounts of information there are more efficient methods.

***The Import Wizard***

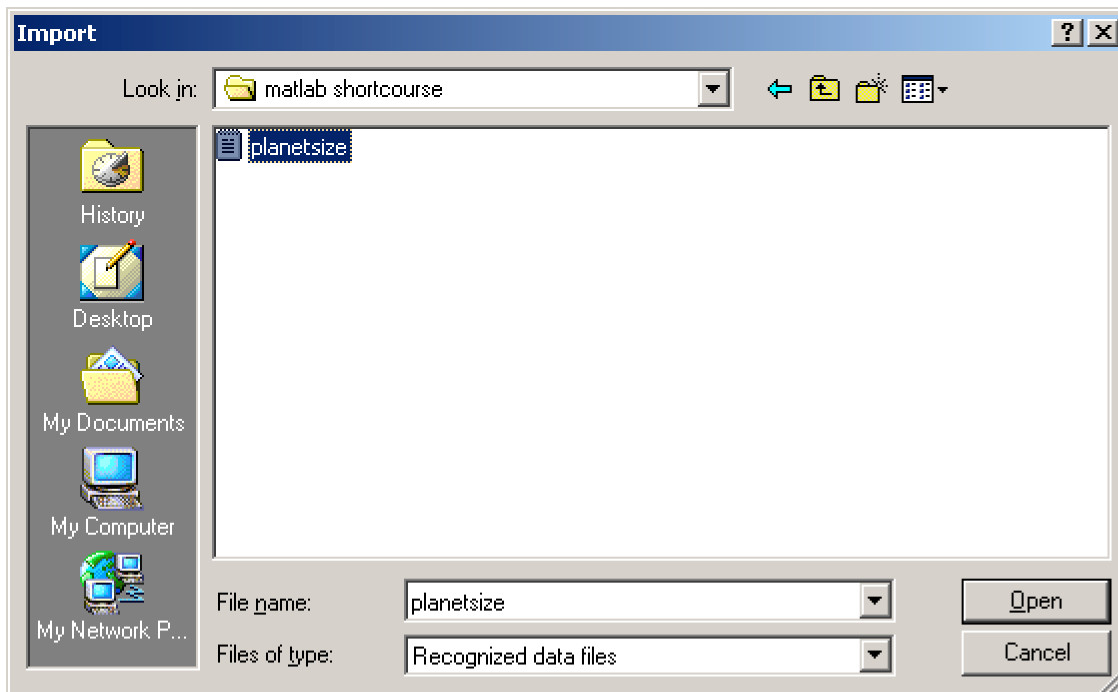
Matlab-6 provides an Import Wizard for convenient importation of data from external files. This tool can be activated by selecting File -> Import Data, or by executing the command

```
>> uiimport
```

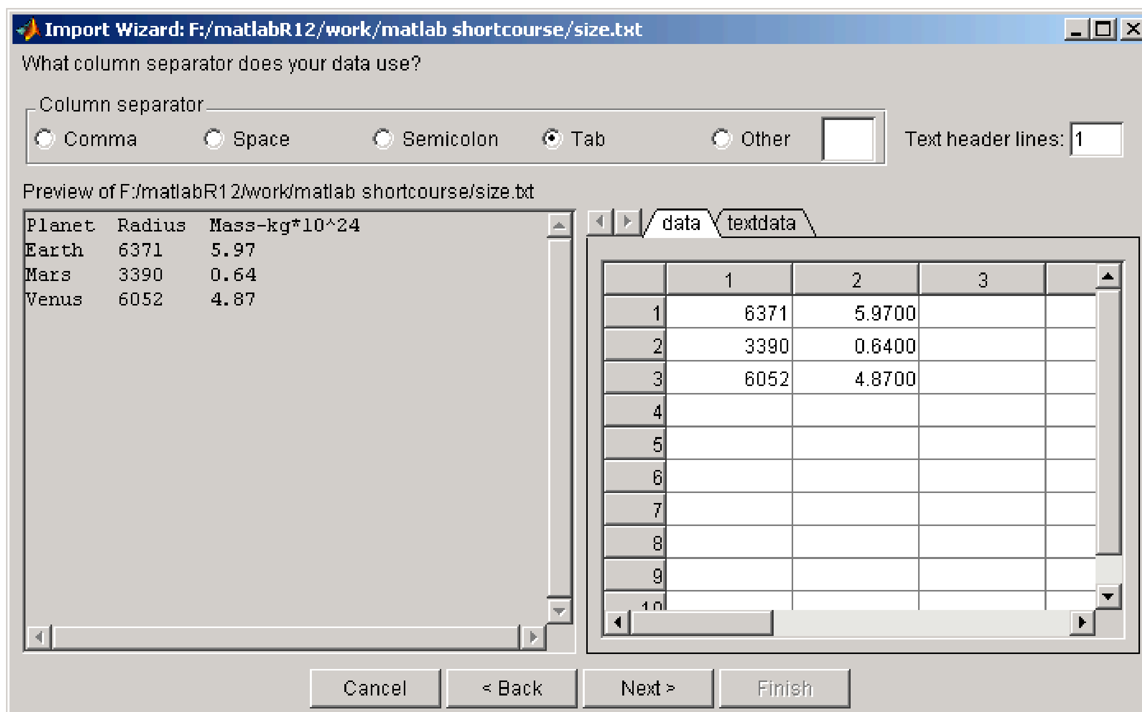
at a Matlab command line prompt. This utility can be used for importing both text and numerical data contained within the same data file, but entries have to be in a matrix format with specified column separators. As an example, consider a small file, *planetsize.txt*, containing a few planets' names, radii, and masses with columns separated by tabs:

| Planet | Radius | Mass-kg*10^24 |
|---|---|---|
| Earth | 6371 | 5.97 |
| Mars | 3390 | 0.64 |
| Venus | 6052 | 4.87 |

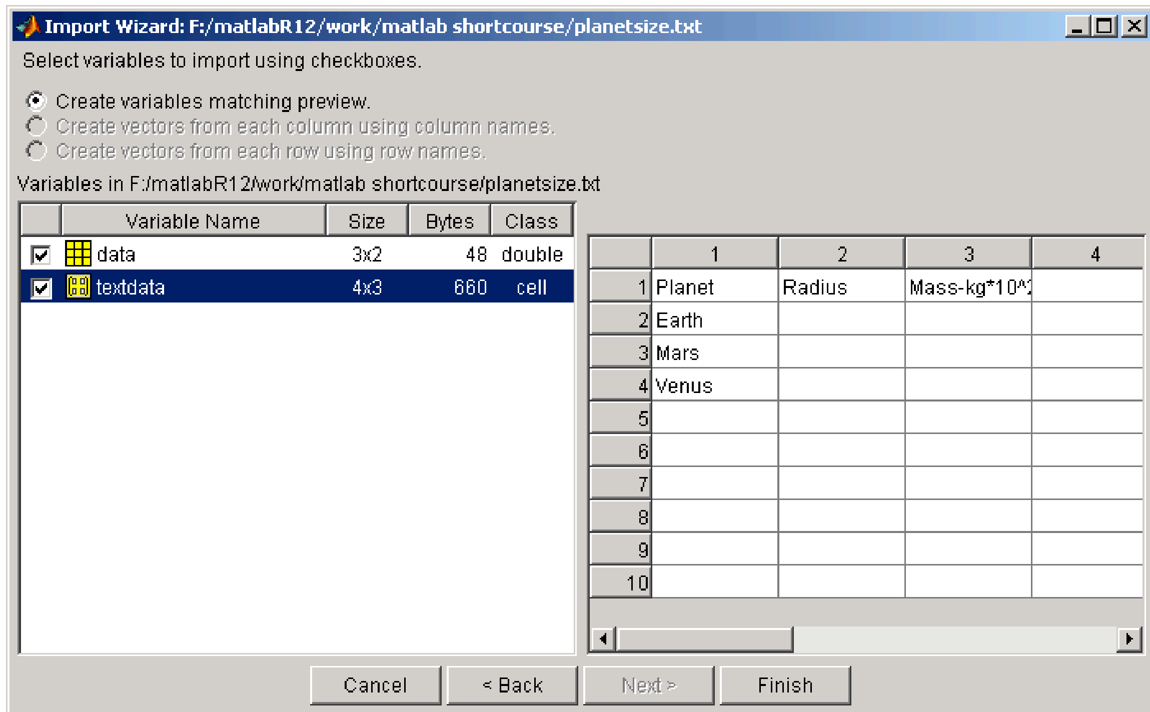
This file can be selected from the import wizard window as shown below




Clicking **Open** will import the file into Matlab. A preview screen appears which lets you confirm the data importation, with separate tabs to examine the numerical data and the text fields:



To continue, click on **Next>** and a new screen will provide the opportunity to selectively choose which information from the file to import into the Matlab workspace. The process is completed by clicking **Finish** .




***Import Functions***

There are a variety of other ways to import complex data using the Command Window. Below several of these functions are described in detail.

**csvread**

This function imports numeric data with comma-separated values (csv). For example, in the planetary data above, suppose that we have an external file *planets1.txt* , containing the kilometer radial distances and 10^24 kg mass values for Earth, Mars and Venus:

```
6371,5.97
3390,0.64
6052,4.87
```

The command

```
>> planets1 = csvread('planets1.txt')
```

would create a 3x2 matrix with the name *planets1* whose content is the same as that shown on the data tab generated previously by using the import wizard for

the *planetsize.txt* file.

**dlmread**

This function is similar to **csvread** but more flexible, allowing the delimiter to be specified by any character rather than restricting it to be a comma. For example, suppose that the file *planets2.txt* had the format

```
6371;5.97
3390;0.64
6052;4.87
```

where values are separated by semi-colons. The command

```
>> planets = dlmread('planets2.txt', ';')
```

would create that same 3x2 matrix with the name *planets2*.

**load**

This is similar to **csvread** and **dlmread** but the separators have to be blank spaces. For example, if the file *planets3.txt* has the structure

```
6371 5.97
3390 0.64
6052 4.87
```

then the command

```
>> load planets3.txt
```

would create the same 3x2 matrix with the name *planets3*.

**fscanf**

This is a lower level import function, equivalent to the C language function of the same name but with the important difference that the result is vectorized. It requires extra manipulations of opening and closing the file, but is more versatile in allowing text and numbers to be read in together. For example, if we want to import the data from our file *planetsize.txt* we could use the following sequence of commands to import the contents of the file in vectorized form

```
fid =fopen('planetsize.txt');
planetsize = fscanf(fid,'%c')
fclose(fid);
```

where the '%c' argument for **fscanf** is identical to the C language, i.e., parsing as character strings. The variable *planetsize* is then actually an 80 element row vector of characters (including blank spaces, tabs, and linefeeds) which have numerical ascii character code values. Characterized data values need to be converted to numbers before mathematical manipulations will make sense. For example, we could convert the characters in thestring representing the earth radius, i.e., elements 39 – 42, to numerical form using the **str2num** Matlab function:

```
>> earthradius  = str2num(planetsize(39:42))
```

The variable *earthradius* will be in numerical form, ready for performing mathematical operations, e.g.,

```
>> earthvolume = (4/3)*pi*((earthradius)^3)
```

**textread**

The **textread** function is similar to the primitive **fscanf** but will allow data variables to be defined as part of the import process. As an example, again suppose that the file *planetsize.txt* contains the information that we want to import into Matlab. We can specify the number of header lines to skip before reaching the actual data (one such line in this example), and the individual data formats for each column of data. The file contains the character string planet names in the first column, the decimal integer planet radii in the second column and the floating point planet masses in the third column. Thus, the command

```
>> [planets, radii, masses] = textread('planetsize.txt',
...
   '%s %d %f','headerlines',1)
```

where "%s" signifies string, "%d" signifies decimal, and "%f" signifies floating point in the format argument, will give the display

```
planets =

    'Earth'
    'Mars'
    'Venus'


radii =

        6371
        3390
        6052
```

```
masses =

    5.9700
    0.6400
    4.8700
```

and add the vector variables *planets*, *radii*, and *masses* to the workspace.

***Export Functions***

**diary**

The simplest way to export data to an external file is make use of the **diary** function. This is a utility for logging a transcript of the Matlab command line input and screen output. The logging process starts subsequent to the command

```
>> diary filename
```

where "filename" is chosen, and terminates with the command

```
>> diary off
```

thus creating an external file that can be edited with a text editor to remove extraneous material. For an example, let us create new variables using the data imported from the *planetsize.txt* file:

```
>> volumes = (4/3).*pi.*((radii).^3);
>> densities = (10^27).*masses./((10^15).*volumes);
>> planetinfo(:,1) = volumes;
>> planetinfo(:,2) = densities;
```

The variable *planetinfo* will then be a 3 x 2 matrix with column 1 containing planet volumes in cubic kilometers and column 2 containing planet densities in grams per cubic centimeter. To export this information to an external file *planets4.txt* we first activate the diary, set the display format to exponential form, type in the name of the variable *planetinfo*, and turn off the **diary**:

```
>> diary planets4.txt
>> format short e
>> planetinfo
>> diary off
```

The external file *planets4.txt* then contains

```
format short e
planetinfo
```

```
planetinfo =

  1.0832e+012  5.5114e+000
  1.6319e+011  3.9219e+000
  9.2851e+011  5.2450e+000

diary off
```

The *planets4.txt* file can then be put into a text editor for deleting unwanted lines, adding column headers, etc.

**dlmwrite**

The **dlmwrite** function allows you to write external data files in which the delimiter can be specified. Let us assume that we have the variables from the **textread** example above loaded into the workspace; that we have defined the vector variables *volumes* and *densities* as shown above; and that we want to write out a new file *planets5.txt* with the new volume and density data. We create a local array *planets5* in the Command Window by typing in the desired data, a column for each of the two vector variables

```
>> planets5(:,1) = volumes;
>> planets5(:,2) = densities;
```

The command

```
dlmwrite('planets5.txt',planets5, '; ')
```

requests that the contents of the array *planets5* be written to an external file *planets5.txt* using a semicolon delimiter. Thus the new external file *planets5.txt* will contain

```
1083206916845.75;5.5114
163187806143.123;3.9219
928507395798.201;5.245
```

**save**

This utility is a primitive function that will save an array in an external file with columns separated by blank space. With no arguments at all, the **save** command will store the current values of all variables in a binary file *matlab.mat* from which they can be retrieved in a subsequent Matlab session. Using the same array *planets5* created above, the command

```
>> save planets5.txt planets5 -ascii
```

will produce an external file `planets5.txt` whose content is

```
  1.0832069e+012  5.5114124e+000
```

```
  1.6318781e+011  3.9218617e+000
  9.2850740e+011  5.2449771e+000
```

Without an extension such as ".txt" and the flag "-ascii", the default external file will be given a default extension ".mat" and will be in binary format.

**fprintf**

This is another low level function that is equivalent to the C language function of the same name. It is useful for exporting information that contains both text and data in a specified format, using syntax similar to that of the C programming language. As an example, suppose we want to export the newly computed planet volume and density values shown above into a file *planetinfo.txt* that has the same layout as the imported file *planetsize.txt.* For this purpose we need to create strings of 49 characters for each line in a new array, both the header line which identifies the data in each column and the three subsequent data lines themselves. We will give the new 4x49 array the arbitrary name *outdata*. The header for the column of planet names, consisting of the first ten characters of each line, can be created by assignment

```
>> outdata(1,1:10) = 'Planet    '
```

and the headers for the the variable columns, spanning 38 subsequent characters, can be generated by

```
>> outdata(1,11:48) = 'Volume (km^3)         Density(g/cc)   '
```

The subsequent data lines will have the planet names and variable values. For these, the numeric values in the *planetinfo* variable need to be converted to text using the Matlab **num2str** utility. For example

```
>> outdata(2,1:10) = 'Earth     '
>> outdata(2,11:48) = num2str(planetinfo(1,:))
>> outdata(3,1:10) = 'Mars      '
>> outdata(3,11:48) = num2str(planetinfo(2,:))
>> outdata(4,1:10) = 'Venus     '
>> outdata(4,11:48) = num2str(planetinfo(3,:))
```

We also need to create an end of line character for each line in the 49th position. This is done using the Matlab **sprintf** command, which functions as a printing command in the same way as its namesake does in the C programming language

```
>> outdata(1:4,49) = sprintf('\n')
```

where "\n" is the notation for the end-of-line character. As an output format we want the text strings with a line feed in the display wherever there is an end-of-line character in the

array. We can create a format variable, for example *outformat*, which specifies the display characteristics by assigning it a string value with parsing instructions

```
>> outformat = '%s \n'
```

where "%s" indicates a string and "\n" indicates a linefeed at the end-of-line character. The new file with the derived variable data can then be created with a sequence of two commands, the first opening a new file with **fopen** and assigning it a numerical file ID and the second printing the contents in the desired format with **fprintf**, both of which are analogous to their namesake commands in the C programming language. In our example we created the output array by row whereas **fprintf** assembles by column. Thus the array that we want exported is actually the transpose of that which we created, i.e., *outdata*' (with the trailing single quote mark) rather than *outdata* itself. In the Command Window we therefore execute the commands

```
>> fid = fopen('planetinfo.txt', 'w')
>> fprintf(fid,outformat,outdata')
```

where the "w" is the permission argument of **fopen** signifying permission to create and write to the file.

### *M-file Scripts*

A sequence of command line instructions can be assembled as a macro for use in importing information. These are called "m files" and have file names with the extension ".m". For example, an external M-file *planetradii.m* containing

```
earthradius = 6371
marsradius = 3390
venusradius = 6052
```

can be used as a source to import these variables with these values using the command

```
>>  planetradii
```

### *M-Books*

The **M-Book** feature is new to Matlab version 6. It is used to transfer data between Matlab and Microsoft Word, a commonly used word processing utility. As an example, suppose that we have a small Microsoft Word document with the name *planetdata.doc* containing the following text

Every planet has a mean radius in kilometers. Here are some examples:

earthradius = 6371

```
marsradius =  3390
venusradius  = 6052
```

This file can be imported into Matlab as an M-book, but if the notebook utility has not been used before, the setting first needs to be configured. With the command

```
>> notebook –setup
```

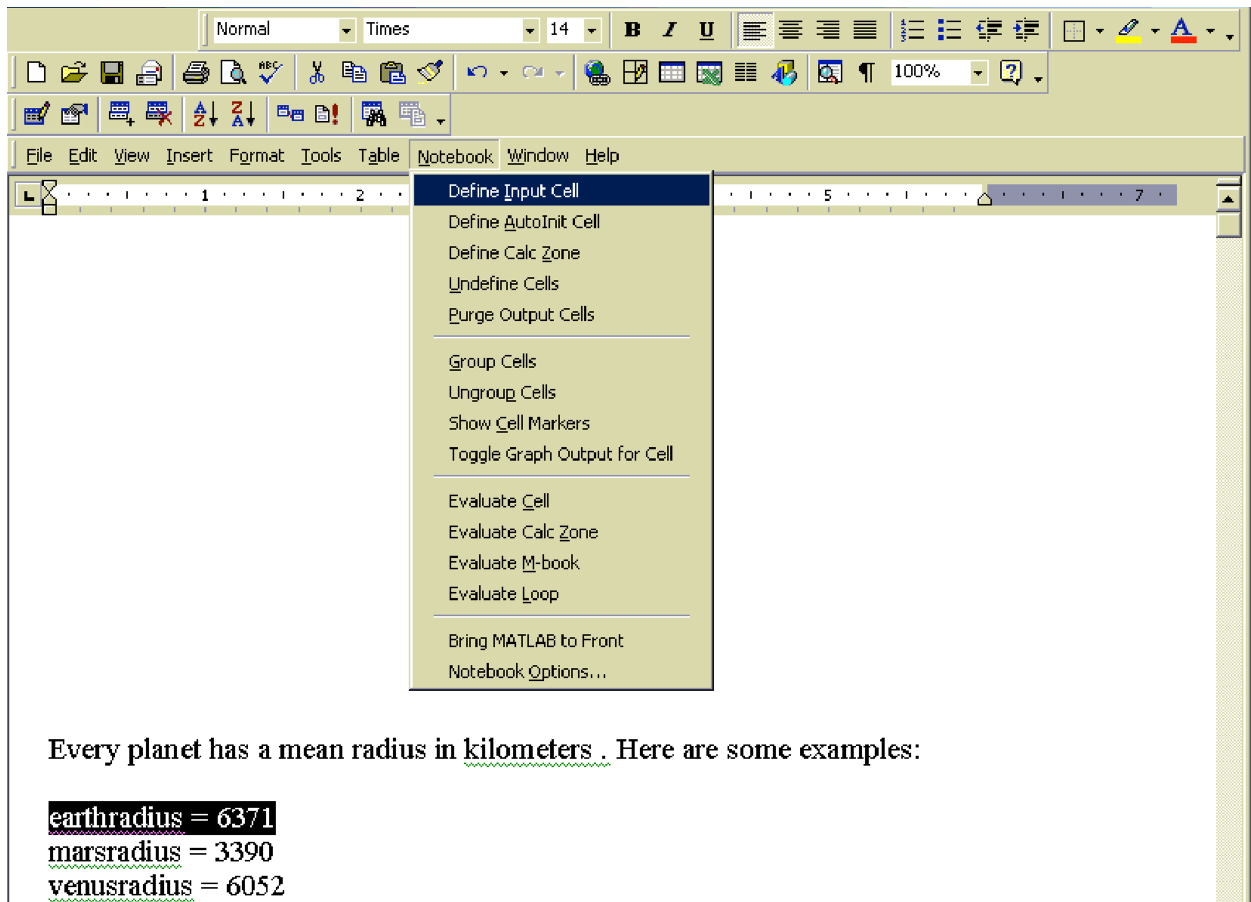
Matlab will prompt for a version of Microsoft Word, its file location and the location of a template to be used for the M-book file. If this configuration is already in place, the command

```
>> notebook planetdata.doc
```

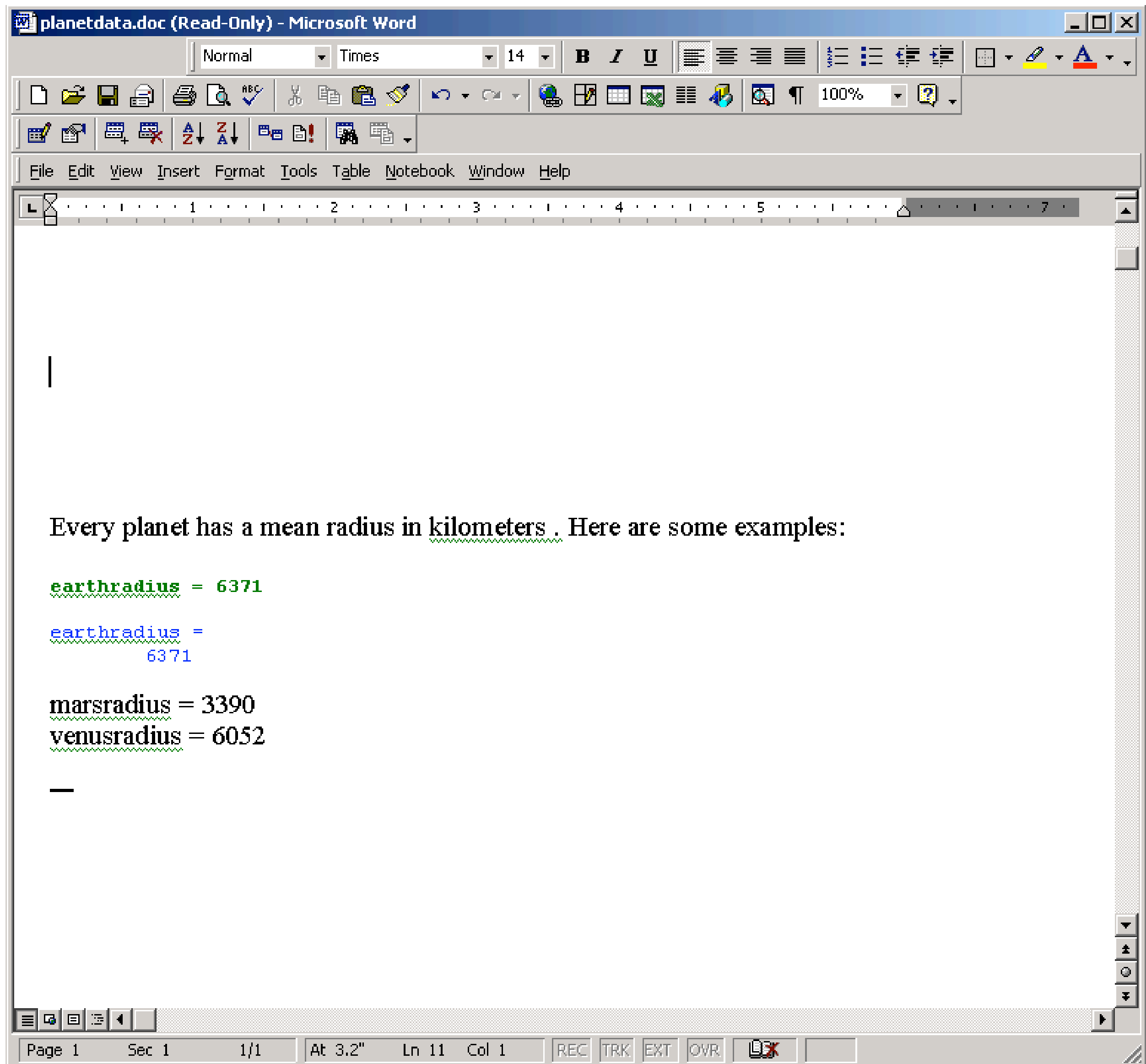
is used to access the external file. If no Microsoft Word file exists yet, one can be created using the generic command

```
>> notebook
```

which opens up a blank Microsoft Word document; followed by Insert -> `File` and navigating to the location of *planetdata.doc*. This procedure also adds a *Notebook* pull down menu to the Word toolbar. Once imported, cells can be defined and evaluated within the document by making the appropriate selections from the *Notebook* pull down menu on the toolbar.


Normal
Times
14
100%
File Edit View Insert Format Tools Table Notebook Window Help
Define Input Cell
Define AutoInit Cell
Define Calc Zone
Undefine Cells
Purge Output Cells
Group Cells
Ungroup Cells
Show Cell Markers
Toggle Graph Output for Cell
Evaluate Cell
Evaluate Calc Zone
Evaluate M-book
Evaluate Loop
Bring MATLAB to Front
Notebook Options...
Every planet has a mean radius in kilometers . Here are some examples:
earthradius = 6371
marsradius = 3390
venusradius = 6052

After subsequently choosing Notebook -> Evaluate Cell , where here we have chosen only a single line in the document to comprise the cell, the window will appear as

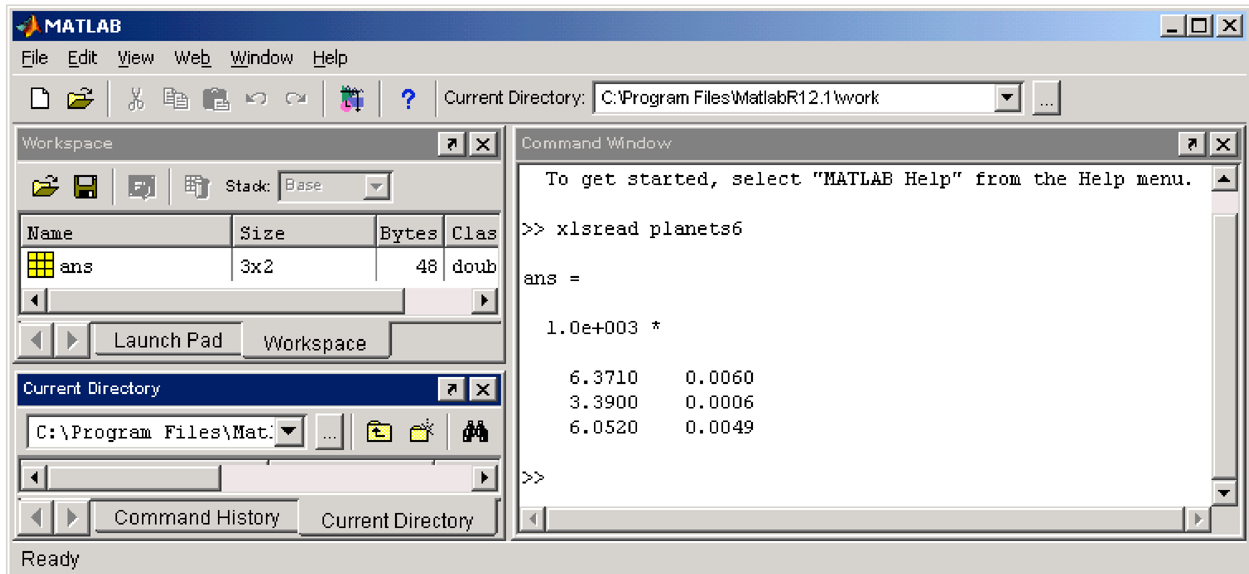



### *Advanced Methods for Binary Information*

In addition to the basic methods presented in this tutorial, there are several utilities available in Matlab for importing specialized types of binary files. Examples are **aviread** for audio-video interleaved (AVI) files, **imread** for image files in common formats such as .jpg or .gif, and **xlsread** for Excel spreadsheets.

For example, the data from the Excel spreadsheet planets6.xls can be imported into the Matlab workspace with the command

>> xlsread planets6




## Section 4: Notation, Syntax, and Operations

Like other command driven applications, Matlab requires information to be presented in a certain manner in order to be properly interpreted. It is a bit more rigid than some, for example case sensitivity for names of constants and variables, and a bit more flexible than others, for example many default values and interpretations for missing or incomplete arguments.

### *Variable names*

The case sensitive names of variables and assigned constants can contain any of the 26 lower case and 26 upper case letters of the standard Latin alphabet along with the ten digits between **0** and **9** and the underscore. A name can be any sequence of 1 to 31 of these characters, but the first must be an upper or lower case letter and there cannot be any embedded blank spaces. Names longer than 31 characters are permitted but only the first 31 are used for unique identification. Constants and variables can be assigned values using the assignment operator, a single equal character ( **=** ), e.g., $a = 3$.

### *Numerical conventions*

Matlab uses standard numerical notation. Numeric variables and constants are stored and displayed as sequences of base 10 digits. The radix point for fraction expansion is the period ( **.** ), and the lower case letter **e** is used for floating point exponent representation. The **e** must be immediately followed by a + or – sign and then an integer. The + is

optional for positive exponents, and arbitrary preceding zeros are permitted. For example, the integer **2000** can be represented as *2.0e+003*, *2e3*, *0002.000e0003*, or even with a negative exponent as *20000e-1*; and the fraction $\frac{1}{2000}$ can be represented as *0.0005*, *5e-4*, or *05.0000e-00004*, or even with a positive exponent as *0.00005e001*.

Matlab accommodates complex-valued numbers of two dimensional divisional algebra using the symbols **i** and **j** as default notations for $\sqrt{-1}$. These two symbols are initially identical, accommodating an historical notational difference between mathematical literature and engineering literature. They are not the separate roots of -1 in the classical **i j k** notation of higher dimensional divisional algebras such as Hamiltonian quaternions, and in fact **k** does not have an initial built-in assigned value. Once **i** or **j** has been assigned some value the symbol must be immediately preceded by a numerical value in order to signify an imaginary part. If not, it will be interpreted as a separate entity. Thus:

(1 + 1i ) is a single complex number with magnitude 1 for both real and imaginary parts
(1 + 1*i) is a sum of the integer 1 and whatever value **i** happens to have at the time

Although **i** has an initial built-in value of $\sqrt{-1}$, it can be overwritten by assignment, and it frequently is because that letter is used routinely to denote an index counter in loops.

***Geometrical and directional conventions***

Most of Matlab's geometrical and directional conventions should be familiar. The principal domain for multivalued functions is symmetrically centered around zero, so that, for example, the inverse sine function *asin* has the principal domain [-π/2, + π/2]. When a branch cut is needed for analytic continuation in the complex plane, it is typically extends from a point on the real axis to negative infinity along that axis. Angles in the complex plane or in polar coordinates are considered positive when measured in a counterclockwise sense. Likewise the chirality of multidimensional coordinate systems is established by the counterclockwise (right hand thumb) rule: if the fingers of the right hand are curled from the positive nth axis toward the positive (n+1)th axis then the right thumb will point in the positive direction of the orthogonal (n+2)th axis..Of course in terms of plotting and visualization, dimensionality is limited to 3. For data analysis using matrix techniques, Matlab interprets columns as variables with rows as observations.

***Operator and delimiter symbolics***

Basic Matlab notation essential for computation includes symbols that distinguish between operations on elements within matrices and between matrices as a whole. The difference between these all stem from the fundamental definition of matrix multiplication specifying that a product matrix element is the inner product of the corresponding row vector of the left hand component with the column vector of the right hand component, e.g.,

$$X = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ and } Y = \begin{pmatrix} e & f \\ g & h \end{pmatrix} \Rightarrow X * Y = \begin{pmatrix} (a*c)+(b*g) & (a*f)+(b*h) \\ (c*e)+(d*g) & (c*f)+(d*h) \end{pmatrix}$$

The Matlab notation for binary operations on matrices, vectors, and their elements are as follows:

+ for matrix or element addition
**-** for matrix or element subtraction
.* for element multiplication
* for matrix multiplication
./ for element division
/ for right matrix division (right multiplication by an inverse)
\ for left matrix division (left multiplication by an inverse)

There is also Matlab notation for a few unary operations

.^ for raising an element to a power
^ for raising a matrix to a power
' for converting to complex conjugate transpose (single forward quote)

Matlab also has specific notation and symbols for delimiters, separators, grouping, type assignment, and so forth: Essential among these are:

**[ ]** for vector and matrix delimiter
**( )** for grouping, vector and matrix element indicies, function argument delimiter
**:** for index range separator
**;** for matrix row separator
*blank space* for matrix column element separator
**,** *comma* for matrix index separator, function argument separator
**.** *period* for radix expansion separator (American convention)
**' '** *single forward quotes* for demarcation of character strings

The Matlab notation for Boolean logicals uses the following symbols:

== for equality
**&** for AND
**|** for inclusive OR,
~ for NOT
<= for less than or equal,
>= for greater than or equal
~= for not equal,
1 for true,
0 for false.

There is not a specific symbol notation for an exclusive OR, but there is a functional equivalent: *xor(p,q)* is the same as the compound logical *(p & ~q)* | *(~p & q)*.

Matlab also has special interpretation for certain symbols related to positioning within a line of command instruction or code. Important and commonly used examples are:

Ignore Further Text: **%** anywhere in a line (uncompiled comment for rest of the line)
Suppress Display: **;** at the end of the line
Continue on Next Line: **…** (three periods) at the end of a line

When working interactively at a keyboard, Matlab has a standard notation to indicate a status of being ready for input

>> *prompt* for keyboard input in the command window

Also for interactive input from a keyboard, Matlab has a couple of standard notations to interrupt functioning. These are:

^c simultaneous control key and lower case **c** to stop execution and return to prompt
^q simultaneous control key and lower case **q** to stop execution and exit Matlab

# Part II: Computing and Programming

## Section 5: Computational Procedures

The fundamental concept to remember when doing computations with Matlab is that everything is actually a matrix, however it may be displayed. Indeed, the name *Matlab* itself is a contraction of the words "MATrix LABoratory". Thus the input

```
>> a = 1
```

does not directly assign the integer **1** to the variable **a**, but rather assigns a *1x1* matrix whose first row, first column element is the integer **1**. Similarly, the input

```
>> b =  'matlab'
```

does not directly assign that character string to the variable, but rather assigns a *1x6* matrix, i.e., a six element row vector of text type. The elements are displayed as characters because of the text data type, but the underlying structure is a vector of integers representing the ascii character code values for the letters, which becomes apparent if it is multiplied by the integer **1**, e.g.,

```
>> c =  1.*'matlab'

c =

   109    97   116   108    97    98
```

Logical expressions can be assigned to a variable, with a binary value specified by their truth or falsity, but the actual variable is again a *1x1* matrix whose element is either **1** or **0** rather than the binary digit itself. For example,

```
>> d = (1+1 = = 2)
```

creates a logical *1x1* matrix **[1]** with the property that *(d + d)* will produce the numeric *1x1* matrix **[2]**, while *(d & d)* will produce the logical *1x1* matrix **[1]**.

***Special Built-in Constants***

In addition to the default assignment for **i** and **j** as the complex-valued $\sqrt{-1}$ , Matlab has several other built-in constants, some of which can be overwritten by assignment,. but

which are then restorable to the original default with a clearing command. Some commonly used ones with universal meaning are

pi for the rational approximation of $\pi$
inf for generic infinity (without regard to cardinality)
NaN for numerically derived expression that is not a number, e.g., $\frac{0}{0}$

One particular built-in constant is a *de facto* variable that is constantly being overwritten:

ans for the evaluation of the most recent input expression with no explicit assignment

Care needs to be taken that no variable is intentionally assigned the name **ans** by overwriting because that variable name can be reassigned a value without explicit instruction during the execution of a program. Thus, though *years* is a commonly used variable name in computational problems, its French equivalent *ans* should be avoided. Similar to this are random number “constants”, also de facto variables because the values are overwritten every time they are called:

rand for a random number from a uniform distribution in [0,1]
randn for a random number from a normal distribution with zero mean and unit variance

These random number constants are similar to the i and j used with complex numbers in terms of retaining their meaning, They represent pseudo random numbers which are revalued sequentially from a starting seed, but once assigned specific values the built-in definitions are lost. However, the built-in functionality can be restored with the commands clear rand and clear randn.

There are also some built-in constants that are characteristic of the particular computer on which calculations are being performed, again susceptible to being overwritten, such as

realmin for the smallest positive real number that can be used
realmax for the largest positive real number that can be used
eps for the smallest real positive number such that *(1 + eps)* differs from *(1)*

Some built-in "constants" are matrices constant by structure but variable in dimension:

zeros(n) for an **n**x**n** matrix whose elements are all 0
ones(n) for an **n**x**n** matrix whose elements are all 1
eye(n) for an **n**x**n** matrix whose diagonal elements are 1 with all other elements 0

***Special Built-in Functions***

Matlab has very many named functions that come with the application. As with variable names, function names are also case sensitive. The pre-defined built-in functions usually have names whose alphabetic characters are all lower case. There are too many to

enumerate in a tutorial, but some that have substantial utility in computation and mathematical manipulation and which use standard nomenclature are the usual trigonometric functions and their inverses (angle values in radians)

sin, cos, tan, csc, sec, cot, asin, acos, atan, acsc, asec, acot

the hyperbolic trigonometric functions and their inverses (angle values in radians)

sinh, cosh, tanh, csch, sech, coth, asinh, acosh, atanh, acsch, asech, acoth

and the exponential and various base logarithms

exp        exponential of the argument
log        natural logarithm of the argument
log2       base 2 logarithm of the argument
log10      base 10 logarithm of the argument

There are also several less commonly used built-in mathematical functions which can be very useful in particular circumstances such as various Bessel functions, gamma functions, beta functions, error functions, elliptic functions, Legendre function, Airy function, etc. All of these built-in mathematical functions are, of course, algorithms to compute rational numerical representation within word memory size, and thus do not produce exact values either for irrational numbers or for numbers whose decimal expansion does not terminate within machine limits.

In addition to these functions that do a numerical mapping, there are also built-in functions for characterizing an argument. Functions for characterizing complex numbers, some of which also are applicable to ordinary real numbers, include

abs    for the absolute value
conj   for the complex conjugate
real   for the real part
imag   for the imaginary part

Similarly, there are built-in functions that characterize ordinary real numbers

factor  for producing a vector of prime number components
gcd     for greatest common divisor
lcm     for least common multiple
round   for closest integer
ceil    for closest integer of greater than or equal value
floor   for closest integer of lesser than or equal value

Matlab also has built-in functions to characterize vectors of elements

length for the number of elements
min for the minimum value among the elements
max for the maximum value of among elements
mean for the mean value of the elements
median for the median value of the elements
std for the standard deviation from the mean of element values

There are some built-in logical functions as well, valued as **1** when true and **0** when false:

isprime for status of a number as prime
isreal for status of a number as real
isfinite for status of a number being finite
isinf for status of a number being infinite

Matlab also has several built-in functions for vectors and matrices in addition to those which are extensions of built-in functions for individual numerical matrix elements

size for number of rows and columns
det for determinant
rank for matrix rank
inv for inverse
trace for sum of diagonal elements
transpose for transposing rows and columns
ctranspose for matrix complex conjugate transpose (adjoint)
eig for column vector of eigenvalues
svd for column vector of singular value decomposition
fft for finite Fourier transform
ifft for inverse finite Fourier transform
sqrtm for matrix square root
expm for matrix exponentiation
logm for matrix logarithm

***Compound Expressions and Operator Precedence***

Users can construct their own functions by defining algorithms or expressions that combine simpler or built-in functions and constants; by producing composites from nesting of simpler functions; by scaling or mapping to a different domain, etc.; and by installing computational code as function m-files in the Matlab search path. Some

simple representative examples, presented here by definition with generic arguments **x** and **y** are:

e = x + log(x)
f = cos(sin(tan(x + sqrt(y))))
g = atan(y/x)
h = sqrt(x^2 + y^2)
Similarly, manipulations can be done to create compound expressions involving matrices and vectors. For simple examples, suppose **M** and **N** are generic square matrices of the same dimension. Some sample user-created composite expressions are:

P = M^2 + 2.*(M*N) + N^2
Q = inv(transpose(sqrtm(M) + logm(N)))
R = fft(N\M)./((length(M))*(rank(N)))

When the construction of composite expressions gets complicated the order and precedence of operations becomes important. In Matlab, expressions are evaluated first with grouping of parentheses, innermost first sequentially to outermost whenever there is nesting. Evaluation then takes place from left to right sequentially at each level of operator precedence. This is of importance when binary operations are not commutative. The precedence order is { .^, ^} > {unary +-} > { .*, *, ./, /, \ } > { + , - } but the sequence of operations can be specified explicitly by parenthesis grouping. As an example consider the following value assignment with no grouping

```
>> k = 1-2^3/4+5*6
```

Matlab would first go from left to right in the right hand side expression looking for a right (i.e., closing) parenthesis. Finding none, it would then search from left to right looking for exponentiation operators. There is one present, linking the digit **2** with the digit **3**, so the first step in constructing a value would be to assign the segment *2^3* with its mathematical value of **8**. That exhausts the top level of precedence, so the procedure is recursively repeated left to right at each level of lower precedence giving a sequence

k = 1-8/4+5*6
k = 1-2+5*6
k = 1-2+30
k = -1+30
k = 29

But what if there were some grouping parameters in the expression? Using the same sequence of elements in **k** suppose we have the value assignment

```
>> m = (1-2)^(3/(4+5)*6)
```

Now when Matlab initially goes from left to right looking for a right parenthesis it will find one immediately following the element **2**. So it then immediately backtracks to find the closest left parenthesis and does a sub-evaluation of the elements within that set of

grouping parentheses. In this case it finds only one operation, i.e., *(1-2)*, and evaluates it to *(-1)*. Then the search continues to the right until the next right parenthesis is found. This continues until all groupings have been collapsed to single elements, after which the recursive left to right processing based on operator precedence continues. In this particular example, the result would be

m = (-1)^(3/(4+5)*6)
m = (-1)^(3/9*6)
m = (-1)^(0.33333…*6)
m = (-1)^(2)
m = 1

Logical operators also have a precedence order: { ~ } > { & } > { | }.

***Commutivity of Operations and Finite Decimal Expansion Approximations***

By definition matrix multiplication is non-commutative, e.g., *M*N* is different from *N*M*, so many expressions involving matrix operations will also be dependent on positioning relative to operators. Addition and multiplication operations on scalars should be commutative, but computational procedures involving round off or irrational number representation by finite decimal expansion can occasionally lead to unexpected results, for example:

`>> t = 0.4 + 0.1 - 0.5` assigns a value 0 to the variable **t**
`>> u = 0.4 - 0.5 + 0.1` assigns a value 2.7756e-017 to the variable **u**

The same problem can also lead to occasional surprises in the value assignment for logical variables, for example

```
>> v = (sin(2*pi) = = sin(4*pi))
```

assigns a value **0** to the variable **v** (logical false), giving the impression that the sine function does not have a period of $2\pi$ .

***Computing with matrices and vectors***

The matrix is the fundamental object in Matlab. Generically a matrix is an **n** row by **m** column array of numbers or objects corresponding to numbers. When **n** is **1** the matrix is a row vector, when **m** is **1** the matrix is a column vector, and when both **n** and **m** are **1** the *1x1* matrix corresponds to a scalar. Development of an end user interface for easy access of subroutines used in numerical computations with matrices, particularly in the context of linear algebra, was the original concept behind the development of Matlab. Thus, calculations involving matrices are really well-suited for the Matlab application. Matrix calculations also can represent an efficiency in doing simultaneous manipulations, i.e., vectorized calculations. This can eliminate the need to cycle through nested loops of

serial commands. As an example, let's compare a matrix computation with an equivalent calculation using scalars. Suppose we have a matrix

$$\text{power1} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The vectorized matrix procedure

```
>> power2 = power1.^2
```

is more efficient than is the scalar procedure:

```
>> for nrow=1:3
      for mcolumn = 1:3
         power2(nrow, mcolumn) = (power1(nrow, mcolumn)).^2
      end
   end
```

***Simultaneous linear equations***

One of the most powerful and commonly used matrix calculation procedure involves the solution of a system of simultaneous linear equations. As an illustrative example we will consider a very elementary case: the grocery bill for customers Abe, Ben, and Cal who purchase various numbers of apples, bananas, and cherries:

| Customer | Apples | Bananas | Cherries | Cost |
|---|---|---|---|---|
| Abe | 1 | 3 | 2 | 0.41 |
| Ben | 3 | 2 | 1 | 0.37 |
| Cal | 2 | 1 | 3 | 0.36 |

From this data we want to know the costs of an individual apple, an individual banana, and an individual cherry. Using Matlab we can create a count matrix **A** that has the quantities purchased arranged with the rows representing the customers and the columns representing the types of fruit, i.e.,

$$A = \begin{pmatrix} 1 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 3 \end{pmatrix}$$

and a column vector **b** representing the total cost for each customer, i.e.,

$$b = \begin{pmatrix} 0.41 \\ 0.37 \\ 0.36 \end{pmatrix}$$

If we then create a column vector **x** representing the individual cost of an apple, a banana, and a cherry in descending order, i.e.,

$$x = \begin{pmatrix} x(1) \\ x(2) \\ x(3) \end{pmatrix}$$

then the system can be stated as the matrix equation

A*x = b

Multiplying each side from the left by the inverse of the matrix **A** will give

((inv(A))*A)*x = (inv(A))*b

But a matrix multiplied by its inverse from the right or left gives an identity matrix **I** with ones on the diagonal and zeros elsewhere, so that

((inv(A))*A)*x = I*x = x

Thus, the simple solution using Matlab is the command

```
>> x = (inv(A))*b
```

which will produce the vector with the unit costs of each individual fruit:

$$x = \begin{pmatrix} 0.0500 \\ 0.0800 \\ 0.0600 \end{pmatrix}$$

Thus the grocer charges 5 cents for an apple, 8 cents for a banana, and 6 cents for a cherry.

***Eigenvectors and Eigenvalues***

Because Matlab was originally developed primarily for linear algebra computations, it is particularly suited for obtaining eigenvectors and eigenvalues of square matrices with

non-zero determinants. The determinant of a square **n** x **n** matrix **A** from which a constant $\lambda$ has been subtracted from the diagonal elements will be a polynomial of degree **n** in $\lambda$ and the roots of that polynomial are the eigenvalues of the original square matrix **A**. The eigenvectors of such a matrix will consist of **n** column vectors, with **n** elements each, which can form an **n** by **n** eigenvector matrix **V**. If the **n** eigenvalues are inserted along the diagonal of an **n** by **n** matrix of zeros to form the diagonal eigenvalue matrix **D**, then original matrix **A** corresponds to

```
>> A = V*D*(inv(V))
```

After left multiplication by inv(**V**) and right multiplication by **V** on each side of the equation it can be seen that the diagonal matrix **D** is equivalent to inv(**V**)***A*** **V**. The Matlab syntax for obtaining these entities is

```
>> [V,D] = eig(A)
```

where eigenvectors are assembled as column vectors in the matrix **V** and eigenvalues are placed on a diagonal of matrix **D** with zeros elsewhere. When there is only a single variable assignment, the *eig* function returns a column vector of the eigenvalues. As an example, let

$$A = \begin{pmatrix} 1 & 2 \\ 5 & 4 \end{pmatrix}$$

Then the command line instruction *d = eig(A)* returns

```
d =
    -1
     6
```

whereas the command line instruction *[V,D] = eig(A)* returns

```
V =

  -0.7071   -0.3714
   0.7071   -0.9285


D =

   -1    0
    0    6
```

Roots of polynomials and zeros of functions

Another computational procedure that is commonly used in Matlab is that of finding the roots of polynomials, or in the case of non-polynomial functions, argument values which produce functional values equal to zero. For example the trivial polynomial $x^2 - 4$ has zeros at $x = -2$ and at $x = +2$, which are also called roots since the function is a polynomial. Similarly the function $sin(2\pi x)$ has zeroes at $x = 0$, $x = \pm 1$, $x = \pm 2$, etc. Although the roots of the trivial polynomial can be evaluated by inspection, it can also be done using Matlab if we form a row vector **p** with coefficients of powers of **x** in descending order: $p = [1 \quad 0 \quad -4]$. The command line instruction will produce a column vector with the roots:

```
>> proots = roots(p)

proots =

    2.0000
   -2.0000
```

The polynomials can be much more elaborate and roots are not always real. For example consider the polynomial $3*x^5 - 10*x^4 + 2*x^3 - 7*x^2 + 4*x - 8$ which can be represented in Matlab as

```
q = [ 3  -10  2  -7  4  -8 ]
```

The roots are easily found and displayed as a column vector with the command line instruction

```
>> qroots = roots(q)

qroots =

   3.3292
  -0.5335 + 0.8285i
  -0.5335 - 0.8285i
   0.5356 + 0.7335i
   0.5356 - 0.7335i
```

The inverse of the *roots* function is the *poly* function, which will produce a row vector of polynomial coefficients in descending power order that describe a polynomial function whose roots are given by the input data vector. This is arbitrary up to a multiplicative factor, so the answer given is with scaling such that the coefficient of the highest power is set equal to **1**. Thus, using the variables defined here,

```
>> qcoeffs  = poly(qroots)
```

will regenerate the values in the vector **q**, scaled by $\frac{1}{3}$ since the lead coefficient was **3**.

Argument values that produce functional values equal to zero in non-polynomial functions, built-in or user-defined, can be obtained in Matlab by using the *fzero* command. This is a search procedure however, and a starting point or an interval must be specified. Whenever a zero functional value is detected the corresponding argument is returned as the answer with no further searching. Thus, for multiple solutions only the first found from the search start value is returned, in the form of a scalar. The target function name is preceded by an @ symbol and a start point or interval has to be specified. Thus, using the trivial example above of *sin(x)*, where we know the zeros by inspection, a particular solution near $\pi$ can be obtained with Matlab by giving the command line instruction

```
>> sinzero = fzero(@sin, [0.9*pi  1.1*pi])

sinzero =

    3.1416
```

which calculates the closest answer as `3.1416`, approximately $\pi$, as expected.

***Poles, residues, and partial fraction expansion***

The *residue* function in Matlab is a bit unusual in that it is its own inverse, with the computational direction determined by the number of input and output parameters. Its utility is to determine the residues, poles, and direct truncated whole polynomial obtained with the ratio of two standard polynomials when that ratio does not reduce to a third standard polynomial, or vice versa. When there are three output variables and two argument vectors, the arguments are taken to be the coefficients of a numerator polynomial and the coefficients of a denominator polynomial. The output vectors are the residues coefficients for the poles, the pole locations in the complex plane, and descending coefficients of the direct whole polynomial part of the expansion. As an example

```
>> pn = [1 -8 26 -50 48];
>> pd = [1 -9 26 -24];
>> [r,p,k] = residue(pn,pd)

r =
    4.0000
    3.0000
    2.0000
p =
    4.0000
    3.0000
    2.0000
k =
     1     1
```

This is equivalent to the expression

$$\frac{x^4 - 8x^3 + 26x^2 - 50x + 48}{x^3 - 9x^2 + 26x - 24} = 1x^1 + 1x^0 + 2\frac{1}{x-2} + 3\frac{1}{x-3} + 4\frac{1}{x-4}$$

Using the *residue* function in reverse, if we know the partial fraction expansion vectors, we can then reconstruct the polynomials whose ratio generated it. In this case the residue coefficient vector, the pole location vector, and the direct whole polynomial vector are given as arguments with the output being the corresponding numerator and denominator polynomials which form an equivalent ratio:

```
>> [qn,qd] = residue(r,p,k)

qn =
    1.0000   -8.0000   26.0000  -50.0000   48.0000

qd =
    1.0000   -9.0000   26.0000  -24.0000
```

The residue function can also be used when the polynomial ratio has poles with multiplicity or order greater than one. Typing help residue at a command line prompt will give details concerning those special circumstances.

***Convolution and deconvolution***

Many engineering applications involving input and output functions to and from a system use numerical convolution and deconvolution to describe each in terms of a system transfer function. These numerical processes correspond to a process of polynomial multiplication and its inverse. If an input function and a transfer function can be represented as vectors of lengths **n** and **m** then the output can be expressed as a convolution vector whose length is *(n + m –1)*. This is similar to polynomial multiplication. Consider for example the polynomials $y = (ax + b)$ and $z = (cx + d)$, both of which have a coefficient vector length of **2**, i.e., *[ a b]* and *[ c d ]*. The multiplication of these two polynomials yields (*acx^2 + (ad + bc)*x + bd*), which has a vector length of (2+2-1) = **3**, i.e., *[ (ac) (ad+bc) (bd) ]*. In Matlab, such convolution can be accomplished by using the *conv* function on the two vectors. The command line instruction in Matlab to find the convolution would require numerical values be assigned to the coefficients *a*, *b*, *c*, and *d.* Using these input vectors

```
>> yzconv = conv(y,z)
```

would generate the convolution vector. The *deconv* function is merely the inverse.
For example, if the output vector **yzconv** and the transfer function **z** were known but the input **y** unknown, then the deconvolution

```
>> ydeconv = deconv(yzconv,z)
```

would reconstruct the values of **y** as a deconvolution vector having the same sequence of elements that produced the known output vector.

***Finite Fourier and Inverse Fourier Transforms***

The process of convolution in one domain corresponds to a process of multiplication in that domain's Fourier transform space. This can simplify calculations of complicated convolution problems and is commonly used in engineering and signal processing. Once the calculations have been performed, the results can be recast in the original domain using an inverse Fourier transform. Matlab provides functions for doing such transforms. The function *fft*, and its 2-D equivalent *fft2*, transform a vector or the columns of a matrix to a Fourier transform domain. Similarly, the inverse functions *ifft* and *ifft2*, transform a vector or the columns of a matrix to the inverse Fourier transform domain. As a trivial example consider the vector **f** = [ 1 2 3 ] and its transformation to a vector **g** in Fourier transform space

```
>> g = fft(f)

g =

   6.0000     -1.5000 + 0.8660i      -1.5000 - 0.8660i
```

then reversion to a vector **h** in the original space which reconstitutes the original **f**:

```
h = ifft(g)

h =

     1     2     3
```

***Numerical Differentiation and Integration***

Matlab has a few functions that can be used in obtaining finite approximations for differentiation and for numerical integration by quadrature. Numerical differentiation methods are difficult because vector or matrix elements cannot always be expanded in resolution to approximate limiting ratio values. However, if a function or data vector can be fit well by an approximating polynomial, then the *polyder* function can be used to get a derivative of that approximate function. This function considers an **n** element argument vector to represent the **n** coefficients of a polynomial of order (**n**-1) in descending order, i.e., the last element is the zero order coefficient. For example, suppose we have an independent variable vector **x** = [ 1 2 3 4 5 ] and suppose we have a dependent functional vector **y** = [ 0 –3 –4 –3 0 ]. We can fit this data exactly to a fourth degree polynomial with **5** coefficients. The Matlab function *polyfit* is used and we choose to store the coefficients in a vector **ypoly**:

```
>> ypoly = polyfit(x,y,4)

ypoly =

   -0.0000    0.0000    1.0000   -6.0000    5.0000
```

The coefficients of the derivative of the approximating (and in this case exact) polynomial representation are then obtained with the *polyder* function, which we will store in a vector designated **ydercoef**. Because the zero order polynomial term is a constant whose derivative is identically zero, that final element is omitted in the vector of derivative coefficients generated by *polyder*. Note that this vector is a vector of coefficients and can be used to obtain numerical values for the "derivative" $\frac{dy}{dx}$ at any particular value of the independent variable **x**.

```
>> ydercoef = polyder(ypoly)

ydercoef =

   -0.0000    0.0000    2.0000   -6.0000
```

The *diff* function in Matlab uses the limiting approximation of the ratio of function change to argument change to obtain a numerical derivative. This can produce very crude results, but can be somewhat useful if the independent variable values are regularly spaced with high resolution and the functional value changes at that resolution are relatively smooth. The syntax for this procedure, using the same example data as were used illustrating the *polyder* method but this time storing in a vector **ynumderiv**, is:

```
>> ynumderiv = diff(y)./diff(x)

ynumderiv =

    -3    -1     1     3
```

With this method, the "derivative" values themselves are the elements of the vector, rather than coefficients. Note that, because of the difference method used, the vector of derivative values is one element shorter than the original data vectors.

Numerical integration can be done with analogous methods. As an illustration, let's consider the integration of the same dependent function variable vector **y** from the numerical differentiation example above over the range of the independent vector **x**, also from that prior example. We can construct a formula-based integration vector using the *polyint* function on the polynomial approximation vector **ypoly**, and we shall store it in a vector yintcoef

```
>> yintcoef =  polyint(ypoly, 0)
```

```
yintcoef =

   -0.0000    0.0000    0.3333   -3.0000    5.0000
0
```

The second argument of the *polyint* function is a constant of integration and will appear as the final element in the quadrature vector if it is specified. With no second argument a value of zero is assumed, so it was unnecessary here but was used anyway to illustrate the two-argument syntax. The elements computed in the vector **yintcoef** are coefficients of a polynomial, in descending order, from which numerical values can be computed for any particular argument value **x**.

Another method for integration is numerical quadrature with Matlab's *quad* function, which uses an adaptive Simpson's Rule algorithm. The arguments for this function are a function name or definition in terms of an independent variable, the lower limit of integration, and the upper limit of integration. Thus, to use the same example, we would need to create a function m-file, say *yvalues.f*, to define the function element by element from data vectors. Alternatively if an explicit relationship can be given, such as the exact polynomial found in this case with *polyfit*, then the functional expression can be given in single quotes as the argument. This latter method will be shown here since creating function m-files are not covered until the following section. The range of **x** in the example is **1** to **5**. The polynomial coefficient vector for the curve fitting had third and fourth order coefficients of zero so that the function value at each value of **x** is given by *x.^2 – 6*x + 5*. Therefore, if we want to store the value of the numerical integration in the variable **ynumint**, we can use the command

```
>> ynumint = quad('x.^2 - 6*x + 5', 1, 5)

ynumint =

  -10.6667
```

Compare that result with the result derived from the **yintcoeff** vector, i.e.,

0.3333*(5^3 – 1^3) -3*(5^2 – 1^2) + 5(5^1 – 1^1) = -10.6708

***Numerical solution of differential equations***

Matlab has several solver functions for handling ordinary differential equations (ODEs), and with the Matlab 6 generation a few functions for handling initial value problems (IVPs) and boundary value problems (BVPs) have been included. There is also a specialized PDE Toolbox with functions for use with partial differential equations, but this Toolbox is not included in the Matlab licenses issued to ITS. However, there is one function included in the standard distribution of Matlab 6 for use in cases of PDEs that are parabolic or elliptic with two dependent variables. In general the syntax for using solvers is

```
>> [x,y] = solver(@func,[xinitial xfinal], y0, {options})
```

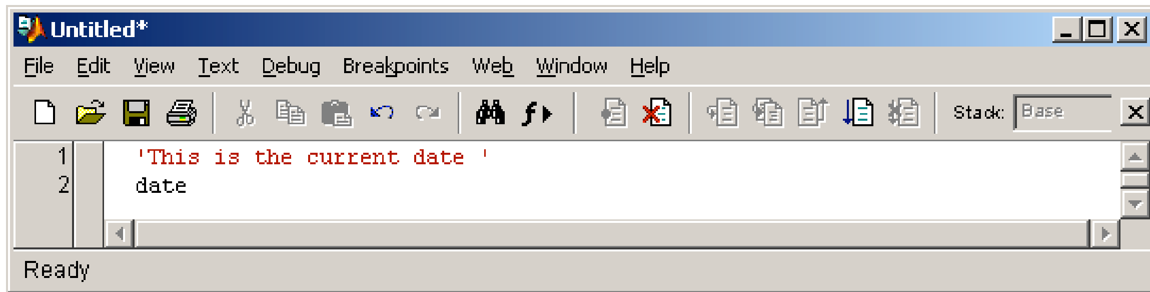
This assumes that a differential equation of order **n** is recast as a system of **n** first order differential equations of **n** variables and that it is described in a function m-file *func* where the independent variable vector **x** spans the range from **xinitial** to **xfinal** and where dependent variable **y** to be solved has a known initial value vector **y0**. The independent variable vector **x** is uniformly spaced and is generated in response to solution iterations. Elements of the vector **y** are the solutions calculated for each value of the independent variable **x**. The most commonly used ODE solvers for non-stiff systems are *ode45* and *ode23*, both of which use a Runge-Kutta type method.. The solvers *ode15s* and *ode23s* are commonly used for stiff equations. For boundary value problems, the general release of Matlab 6 has a solver *bvp4c* and for parabolic and elliptic PDEs the solver *pdepe*.

## Section 6: Programming

Matlab has its own programming language, also known by the name Matlab, which is structured very much like most programming languages. Because its origin involved interfacing with many of the original linear algebra subroutines written in Fortran, Matlab has roots there. But there is also much similarity with C due to later development of graphics functionality and the desktop GUI. The Matlab programming language is thus not particularly difficult or obscure, and the primary task in becoming adept at using it involves learning the various notations and syntax rules.

***Using the Editor***

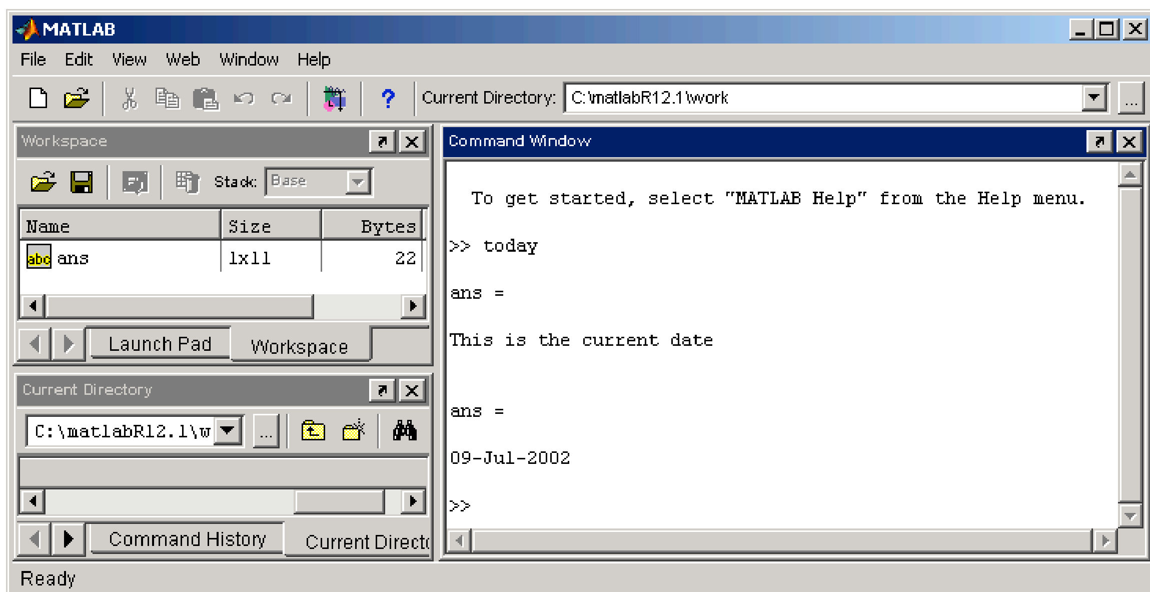
Matlab has the functionality of incorporating user-created files external to the Command Window. These files need to be placed in a folder or directory defined in the pathway and should consist of a series of valid Matlab commands. These are thus equivalent to macros or subroutines in other programming languages, and are called *m-files.* They should have an extension of a period and a lower case **m** so that they can be recognized as Matlab text files. The Matlab 6 GUI has a text editor interface that allows users to create m-files within the Matlab application. It is launched from the navigation toolbar with *File -> New -> M-file* which opens up a new floating window ready for code text input:




This editing window has its own toolbar with conventional Windows type pull down menus and button icons. Lines are numbered outside the text area for convenient referencing when debugging or inserting break, pause or flagging points. This editor tool is very convenient, with color coding and indenting features that aid in construction, but m-files can also be created as plain ascii text in other text editors such as MS Word.

***Types and Structures of M-files***

There are two main types of m-files which are very similar to each other: script files that act as command sequence macros with no parameter passing; and function files that also act as command sequence macros but which accept parameters by value and return a computed number, vector, or matrix assigned by the calling program. M-file scripts have no formal structure other than consisting of a sequence of valid Matlab command line input. A trivial example is the two line m-file shown in the figure illustrating the editor window, which will echo the text string and display the date on the screen. After saving this file, which can be done from the *File* pull down menu, with a name such as *today.m*, any subsequent command line reference to it will result in the execution of its contents. This applies both to input from a Command Window prompt and to any call from within another m-file.



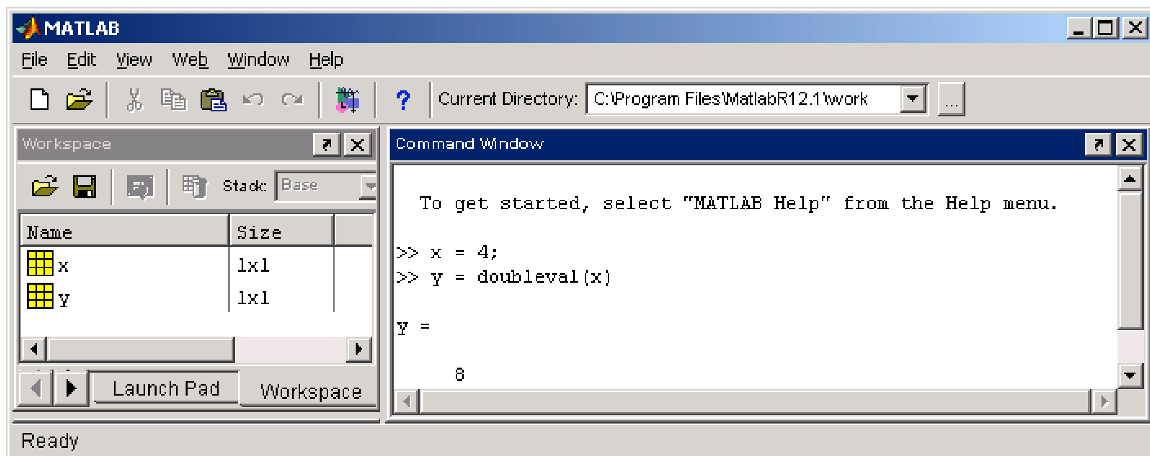
A function m-file is similar except that the first command line should have the form

```
function [ var1 var2 {…} ] = functionname( arg1, arg2, {…}
)
```

and the returned information should assign the computed functional values to **var1**, **var2**, etc., obtained with the function computational algorithm. A trivial example:

```
function f =  doubleval(x)
f = 2*x;
```

which creates a function that doubles the value of the input as the return value. If you save this code with a name such as *doubleval.m*, any subsequent call to *doubleval* from the Command Window prompt or from another script will return a value twice as large.

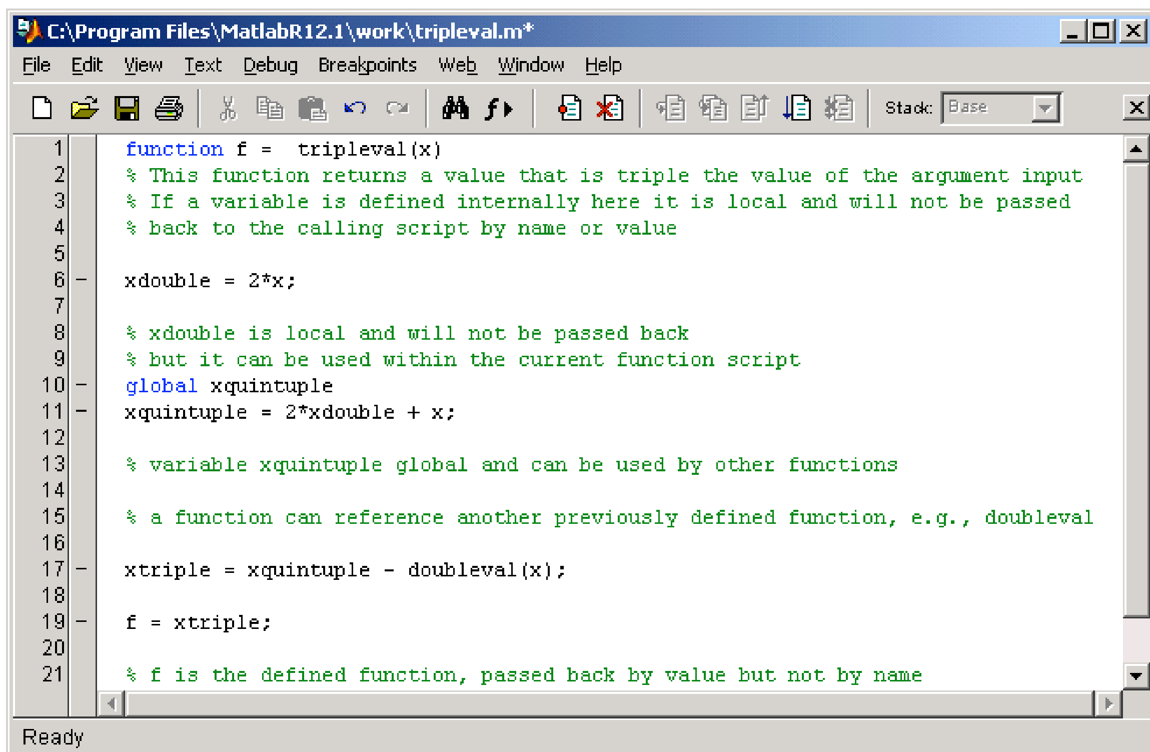


***Internal Documentation***

Matlab m-files can be annotated with internal documentation that is ignored by the command interpreter. Any part of a command line, at a command window prompt or within m-file code, following a percent sign, i.e. %, that is not enclosed in quotes is considered as documentation and does not get processed. The % delimiter can be anywhere on a line. For example, compare the following command line instructions:

```
>> u = 5% + 1%

u =

     5

>> v = '5% + 1%'

v =

5% + 1%
```

For an illustration of internal documentation, here is the display of an annotated function m-file, *tripleval.m*, which returns a value three times that of the input:



***Passing variables by name and value***

If a script m-file is called from a Command Window prompt or by another script, then it is processed as a sequence of commands just as if each line had executed individually or had been incorporated as individual lines of the calling m-file. Thus, all variables created and values assigned to them become a part of the workspace and are recognized by any subsequent reference. On the other hand, function m-files have local variables by default. Only the values specified as function components are returned. However, a variable created within a function m-file can be declared global before assigning a value. If this is done then the next subsequent use of that variable name in another function m-file in which it has also been declared as global will start with the value already established. Every function that is invoked will have its own workspace, but global variables can be shared.

Also, function m-files may themselves contain functions, appended as subfunctions, which are constructed with the same structure as a regular function but with a `return` line indicating the end. Subfunctions within each level can also share global variables and can include their own subfunctions. As an illustration consider the following function m-file

C:\Program Files\MatlabR12.1\work\xseven.m

File Edit View Text Debug Breakpoints Web Window Help

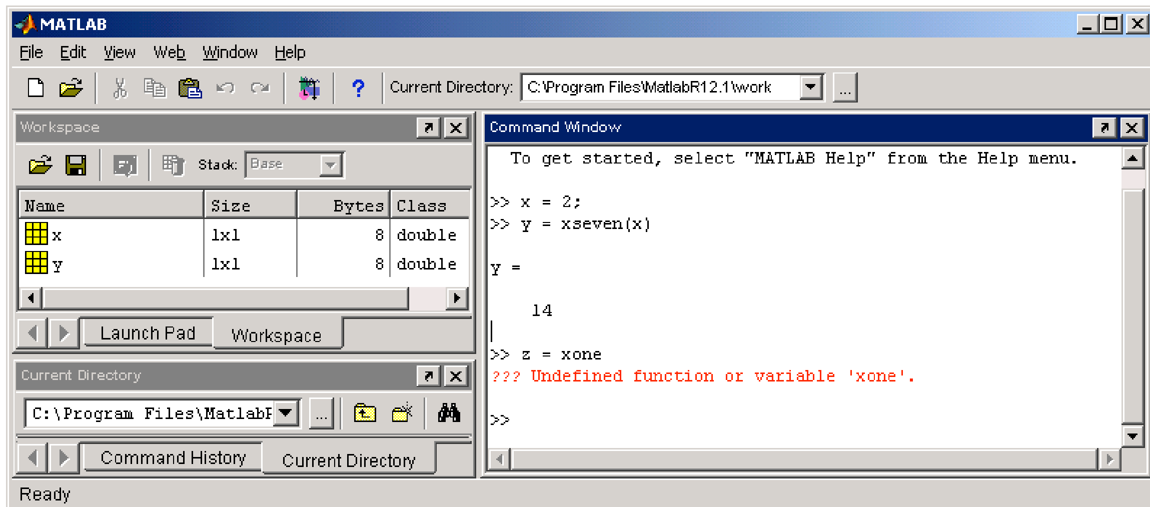
Stack: Base

```
1   function f = xseven(w)
2   % obtains a value seven times the input
3   f = xfour(w) + xthree(w);
4
5   % append subfunction
6     function f = xfour(x)
7     % obtains a value four times the input
8     global xone %defines a global variable
9     xone = x; % assigns a value to global variable
10    f = xone + xthree(x);
11    return
12
13  % append another subfunction
14    function f = xthree(y)
15    % obtains a value three times the input
16    global xone % allows use as defined in subfunction xfour
17    f = xone + xtwo(y);
18    % append a subfunction of the subfunction
19      function f = xtwo(z)
20      % obtains a value two times the input
21      f = 2*z;
22      return
23    return
```

Ready

The function *xseven* has two subfunctions, *xfour* and *xthree*. The subfunctions *xfour* and *xthree* share a global variable *xone*, the subfunction *xfour* calls the same level subfunction *xthree*, and the subfunction *xthree* has its own local subfunction *xtwo*.

Using the function *xseven* is a very contorted way of multiplying a number by seven, but it illustrates function and subfunction architecture. In the Workspace Window notice that a variable generated within a function, e.g., *xone*, does not get carried over.




## *Function evaluation and function handles*

Once a Matlab function m-file has been constructed, named, and placed in the search path, it can be used in several ways. Let's consider the example that we have already constructed and named *xseven.m*. The most direct usage is to assign some variable the value of the function at a given argument, e.g.,

```
y(1) = xseven(5)
```

would assign a value of **35** to the variable **y**, whether interactively from a command line prompt or as a line of code in a script m-file. Alternatively, there is a command *eval* in Matlab which will evaluate a character string as a line of code. Thus, we get the same result if we use the command

```
y(2) = eval('xseven(5)')
```

However, this is not too efficient, as the entire Matlab interpreter is loaded for processing *eval*. For function evaluations the command *feval*, which is able to load only what is needed, can be used for greater efficiency. This command takes as arguments a function name, followed by that function's arguments. The function's arguments can be by value or by variable name if a value has already been assigned. As an example, suppose that we already have a variable **z** assigned the value **5** in the workspace. Then an equivalent

assignment for **y** can be obtained with

```
y(3) = feval('xseven', z)
```

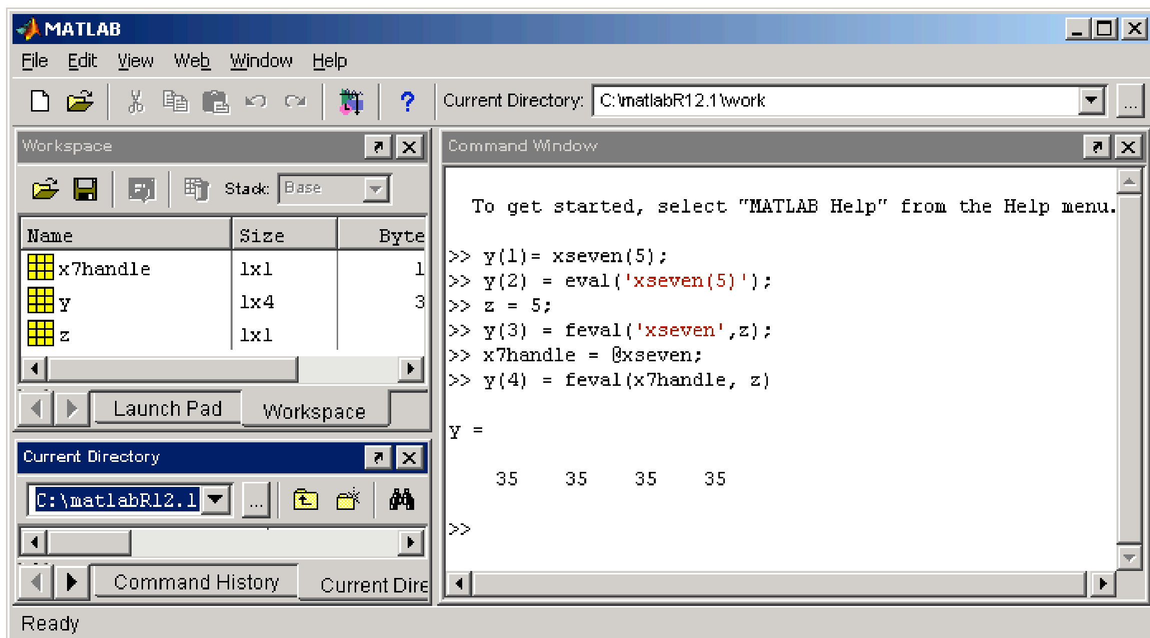
A new feature in version 6 of Matlab is the *function handle* data type. This provides a means of speeding up execution by passing essential information about the function through a *function handle*, referenced with the syntax *@function_name*. The handle can be assigned to a variable name, which in turn can replace the character string in the arguments of *feval*. If we assign *@xseven* to a function handle variable, say

```
x7handle = @xseven
```

then yet another, faster method of assigning the variable **y** above would be

```
y(4)  = feval(x7handle,z)
```

The results of all these methods, which have been stored in the vector **y**,are identical and have the value **35**, i.e., the argument **5** operated on by the function *xseven* which in a roundabout way calculates a multiplication by seven.




### *Function recursion*

A function value can also be passed back to itself through recursion. A simple example is a function that computes a triangular number (sum of all integers less than or equal to the argument). In this function script notice that the function *triangular* calls itself:

```
function f = triangular(n)

% finds the sum of all integers from 1 through n

t = n;
if n > 1
    t = t + triangular(n-1);
end;
f = t;
```

***Flow control***

In Matlab, flow control procedures, as well as syntax, are very similar to procedures in many other programming environments. The *for* and *while* commands are constructs for executing a block of commands multiple times until a specified incrementing index value is reached or a logical termination condition is met. The finish of a repeating block of

commands must be specified with an *end* statement. The general format is thus

```
for  {index} = {firstvalue}:{lastvalue}
{command1; command2; ...}
end

while {logical condition}
{command1; command2; ...}
end
```

Just as in other programming environments, the *for* and *while* loops can have layers of nesting. As an example let's look at a script for finding the number of seconds in a week, with emphasis on illustrating loops and nesting rather than efficiency:

```
% find the number of seconds in a week
total   =  0;       % start with none yet counted
for  m = 1:7        % consider every day in a week
    for  n = 1:24   % consider every hour in a day
        minval = 1;  % start minutes counter
         while (minval <= 60)
             secval = 1;  % start seconds counter
             while (secval <= 60)
                 total = total + 1;  % increment total
                 secval = secval + 1;% increment secs
             end;
             minval = minval + 1;    % increment mins
```

```
            end;
        end;
    end;
```

Matlab also has the capability of logical contingency flow control with the *if* and *switch* commands. The general construction of an *if* command is

```
if {logical condition}
   {command1; command2; ...}
    elseif {logical condition}
        {command1; command2; ...}
      .
      .
      .
    elseif {logical condition}
        {command1; command2; ...}
    else
        {command1; command2; ...}
end
```

The blocks of commands are executed if the specified logical condition is true and an optional block of commands following *else* is executed if none of the preceding logical conditions are true. The *elseif* segment can be expanded to include multiple logical conditions and command blocks, but only the block following the first logically true condition will be executed.

The *switch* command has a similar function but flow is controlled by the current value of a specified expression . The general structure is

```
switch {expression}
     case {value1}
          {command1; command2; ...}
     case {value2}
          {command1; command2; ...}
     otherwise
          {command1; command2; ...}
end
```

The command block executed is that following the *case* value which corresponds to the value of the specified expression. An optional *otherwise* command block can be included to cover expression values which are not enumerated in the cases.

As an example of using *if* and *switch* flow control, consider a script that determines the number of non-primes, primes, and twin primes (incrementing by 2 gives another prime) in a particular range of integers, for example between 1001 and 2000:

```
% find prime status of integers in a range
nontwins   =  0;    % start with none yet counted
twins = 0;          % start with none yet counted
nonprimes = 0;      % start with none yet counted
for  n = 1001:2000   % test all numbers in the range
    if isprime(n)
        switch isprime(n+2)
        case 1
            twins = twins + 1;        % increment
        otherwise
            nontwins = nontwins + 1;  % increment
        end;
    else
        nonprimes = nonprimes + 1;    % increment
    end;
end;
notprime = nonprimes
prime = nontwins + twins
twinprimes = twins
```

***String evaluation and manipulation***

Although character strings are stored as vectors of ascii character code integers within Matlab, they can be evaluated and manipulated in certain circumstances. If a string contains text that represents a valid Matlab command or expression, then the *eval* command can process the string as if it were command line input. Similarly, if a string consists of the name of a defined or built-in function, then the *feval* command specifying the function name and its input arguments can be used to generate a value. As an illustration consider the following script that computes the integer within a given range that has the largest positive difference between its sine and cosine:

```
% maximum difference in sin vs cos in integer range
maxnum = 1; % initialize with first argument
maxdiff(1) = sin(1) - cos(1); % initialize difference
x = 'sin(n) - cos(n)'; % make a string function
for n = 2:1000         % select integer range
    y(n) = eval(x);    % evaluate the string
    maxdiff(n) = feval('max',y(1:n)); % update maximum
    if (y(n)>maxdiff(n-1))
        maxnum = n;    % update the argument if needed
    end;
```

```
end;
maxnum              % integer maximizing (sin - cos)
maxdiff = feval('max',y) % difference maximum
```

In this particular example the result will be the integer that most closely approximates a value of $\frac{3}{4}\pi + 2k\pi$, which, for the selected range, turns out to be 882, or about $\frac{3}{4}\pi + 280\pi$. There are also special built-in functions for evaluating and manipulating strings which are the text equivalent of integers, numbers, or even matrices. All these require that any arguments which are not decimal-based integers or numbers be in a character string format (enclosed within single quotes). This class of functions includes

| | |
|---|---|
| str2int and int2str | [converting from strings to integers and vice versa] |
| str2num and num2str | [converting from strings to numbers and vice versa] |
| int2str | [converting integers to character strings] |
| hex2dec and dec2hex | [converting hexadecimal strings to integers and vice versa] |
| bin2dec and dec2bin | [converting binary strings to integers and vice versa] |
| base2dec and dec2base | [converting nondecimal strings to integers and vice versa] |

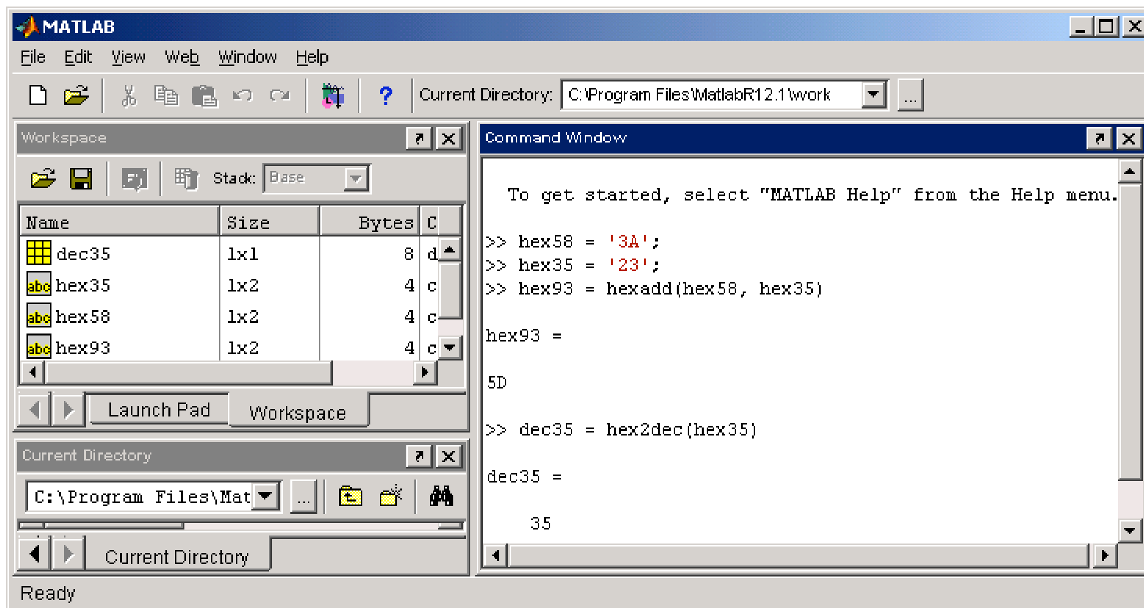
The general syntax used with these functions is

stringvalue = *number2string*(*number*)
numbervalue = *string2number*(*string*)

As an example of the use of these commands we can construct a function for adding hexadecimal numbers, a feature which is not a built-in Matlab utility. Here is one way to do this, where the input is two hexadecimal character strings:

```
function f = hexadd(hex1,hex2)
% Add 2 hexadecimal numbers represented by text
strings
newhex = dec2hex(hex2dec(hex1) + hex2dec(hex2));
f = newhex;
```

Note that the output is also a hexadecimal character string:




### *Keyboard input*

Matlab allows the programmer to include interactive input from the keyboard with the *input* command. The command incorporates a prompt in the Command Window so that the user knows that an input is waiting. A variable is assigned the value of the user's input according to the syntax of the *input* command's argument. The syntax for the *input* command argument is a character string for prompting the user, enclosed in single quotes, followed by a lower case **s** in single quotes if the input is to be considered a string rather than a number or matrix. This is a common way to allow a user to specify a starting guess in an optimization script or to specify a cutoff for an infinite series computation or iterative process. As an example, let's construct a script to evaluate $\pi$ recursively from an iterative algorithm in which the user can set a limit on the number of iterations:

```
% script to illustrate user keyboard input
disp('Iterative Approximation to pi');
reply = input('Limit to iterations? [y/n]','s');
if (reply == 'y')
    itmax = input('Maximum iterations?');
else
    itmax = 100;
end;
iter = 0; % initialize iteration number
res = 0.0001; % resolution for detecting change
prevpi = 2; % initialize a previous value
piapprox = 2 + 2/sqrt(2); % initial guess
while  ((abs(piapprox  -  prevpi)  >=  res)&  (iter  <
itmax))
    prevpi = piapprox; % reset the previous value
```

```
        piapprox = 2 + 2/sqrt(piapprox); % new iteration
        iter = iter + 1; %increment iteration number
    end;
    iterations = iter
    pi = pi
    picalc = piapprox
    difference = pi - piapprox
```

If you test out this script you will find that a consistent value is reached after six iterations even if more than that number is specified as the maximum. However you will also notice that the iterative convergence is to a number very slightly less than the analytic value of $\pi$ at the same resolution. This is probably a consequence of errors propagating from imprecision in the finite decimal representation of $\sqrt{2}$ . It should be a reminder to programmers to be aware of round-off and truncation effects when constructing evaluation schemes involving a substantial number of sequential operations involving floating point approximations.

***Multidimensional arrays and indexing***

The fundamental unit in Matlab is a two dimensional matrix whose elements can be referenced by row position **n** and column position **m**. These element locations are specified in parentheses following the matrix name, with the row position given first. Thus, for example *A(2,3)* refers to the element of matrix **A** which occupies the second row and third column. However, the contents of the matrix are stored within Matlab memory, regardless of how they may be displayed or referenced, as a single column matrix, i.e., a column vector, with sequential values column by column. Therefore an element in an **N**x**M** matrix **A** can also be referenced by a single index, *A(p)* where

```
    p = (m-1)*N + n
```

so that the last element in the first column has index $p = N$, the first element in the second column has index $p = N+1$, etc..
Matrices can be concatenated by defining a matrix of matrices, or by using the *cat* command, but of course the result of every concatenation must be be a rectangular array. As an example, suppose we have four simple rectangular matrices

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 4 & 5 & 6 \\ 3 & 2 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad D = \begin{pmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{pmatrix}$$

These can be concatenated as

E = [A B] giving a matrix of 2 rows and 6 columns

F =  [C; D]     giving a  matirx of  6 rows  and 2 columns

The *cat* command is equivalent, using a direction argument, **1** or **2**, to specify concatenation in a column or in a row respectively.  Thus, we would get the same result as above by constructing the matrices **E** and **F** as

E = cat(2, A, B)
F = cat(1, C, D)

If  an element position outside a matrix is ***referenced***, there will be an error message displayed, for example when trying to display *A(3,2)* from above.  However, if an element outside the current range is ***assigned*** a value, Matlab will accept the assignment and fill in all other element positions necessary to form an expanded rectangular matrix with zeros.  We cannot directly concatenate all four of the example matrices above as

G = [A B;  C D]

because **A** and **C** have differing numbers of columns and rows, as do **B** and **D**.  However, if we assign a value (**0** for example) to *A(3,3)*, *B(3,3)*, *C(3,3)*, and *D(3,3)*, then all four of the component matrices will be restructured as square matrices of dimension 3, padded with zeroes where not explicitly defined., and the assignment for **G** above can be processed.

Arrays of dimension greater than two can be useful, for specifying three dimensional coordinates, assembling functional values determined by three or more independent variables, and so forth.  Matlab was developed in the context of matrix algebra where most functions, matrix multiplication and so forth, are defined on the basis of two dimensional rectangular arrays, so two dimensional arrays are the primary objects. Nevertheless, Matlab can accommodate higher dimensional array notation.  The positional index numbers can be expanded to three, four, or more.  However, display is still limited to two dimensional “cross sections” or slices, and the actual storage is again as a single column vector with an internal index for a multidimensional array element assigned sequentially by row position, then by column position, then by third index, and so forth.  Higher dimensions are not really amenable to visual conceptualization, but think of a four dimensional array being like a set of books along a shelf, each having the same number of pages and with each page having a same size rectangular matrix.  If there are five books each with six pages of matrices with seven columns and eight rows, then a partiuclar element in the array **Z** with all the aggregate matrix elements could be referenced in multidimensional form as *Z(row, column, page, book)*, e.g. *Z(1,2,3,4)* for the first row and second column of the matrix on page **3** of book **4**.  This would correspond to an alternative single column internal storage reference. Before book **4** there are three books with six pages with *7x8* matrices, thus containing *3*6*7*8 = 1008* total elements.  Then before page **3** there are two pages with 7x8 matrices, thus containing *2*7*8 = 112* total elements. Finally, there is one column before the second, in the matrix on page **3** of book **4**, thus containing *1*8 = 8* total elements.  Hence the internal indexing

would assign *1008 + 112 + 8 = 1128* positions before getting to *Z(1,2,3,4)*. Therefore, the programmer could reference *Z(1,2,3,4)* alternatively by *Z(1129)* in this example.

***Debugging***

Matlab has extensive features to assist in debugging program errors, both compile time errors that primarily involve user mistakes in syntax and run time errors which either interrupt execution or produce unexpected or erroneous results. Help for both command line input and m-file source code debugging is available.

The Matlab interpreter will display error messages in the Command Window when it encounters some operational instruction that it cannot understand. Typically it will give some feedback about the problem, including a line number if the error is in an m-file. For example, suppose at the command line we type

```
>>  x(0) = (1^0)/factorial(0)
```

The interpreter will respond with

```
???  Index into matrix is negative or zero
```

because array indices are restricted to positive integers. This could also happen if the problem occurred within an m-file script. Let an initial version of a script *ebase.m*, which is supposed to find the value of the natural logarithm base be constructed as

```
     %script to find the natural logarithm base e
     evalue = 0                   % initialize the value
     prev = -1;                   % initialize a previous value
     n = 0;                       % initialize index
     while (evalue ~= prev)       % iterate if still converging
           nterm(n) = (1^n)/factorial(n);  % next term
           prev = evalue;         % update previous value
           evalue = prev + nterm(n);       % update value
           n = n + 1;                      % increment
     index
     end;
     evalue                       % show final value
```

There will be the same syntax error concerning a non-positive integer index shown, but this time there will also be an additional diagnostic comment of the form

```
Error in ==> C:\matlabR12.1\work\ebase.m
On line 6  ==> nterm(n) = (1^n)/factorial(n); % next term
```

We, as the user, can then edit the script to make an adjustment. If we do not care about keeping the values of the individual terms in the series we can merely convert **nterm** to a transient valued scalar by removing the index , or we could fix the problem by replacing **nterm(n)** with **nterm(n+1)** in the script.

Run time errors are usually more difficult to spot and correct, and there are sometimes several possibilities as to the origin. Operations that would terminate execution in other programming environments do not always do so in Matlab. For example value assignment involving division of a non-zero number by zero will give an assignment of **Inf** (infinity) without stopping program execution. Likewise, assignment involving division of zero by zero produces an assignment of **NaN** (not a number), without halting execution. Errors in logic from user programming can lead to false calculations without giving any diagnostic warning. In the *ebase.m* script for obtaining a value for the natural logarithm base, suppose the series term had been written as

```
nterm(n+1) = (1^n)/gamma(n);
```

where we have made the logic mistake of using *gamma(n)* rather than *factorial(n)*. Because the gamma function for a positive integer is actually the factorial function of that same integer argument less one, i.e.,

```
gamma(n) = factorial(n-1)
```

we have a logic error in user supplied instruction. The function *gamma(0)* will evaluate to *Inf* so that the first series term computes to zero with the result that the base value for natural logarithms remains at the initialization value without any subsequent iteration being performed. This error in logic can be fixed by the user via changing *gamma(n)* to *gamma(n+1)*, or its equivalent, *factorial(n)*.

Other common logic errors that will not cause termination of program execution are things such as infinite loops, mistakes in operator precedence or grouping, unintentionally overwriting a variable value from name duplication, and systematic propagation of round-off or truncation error. There are several methods to use in tracking down such errors. One straightforward way is to delete the semicolons from the end of assignment instruction lines so that variable values are displayed in the Command Window during program execution. Another method is to have some user-specified diagnostic string displayed at certain designated points in the coding or whenever some logical condition (such as the value of a variable exceeding a certain number) is satisfied. This can be done with the *disp* command, which displays the value of its argument in the Command Window. The programmer can also insert instances of the *keyboard* command in the code, which will cause execution to halt and give a **K>>** prompt in the Command Window. At this point instructions such as changing or testing a variable value can be given from the keyboard. Upon a keyboard command of *return*, the program execution continues. An example code to illustrate these techniques is the following for getting the value of a Riemann zeta function $\zeta$ from its infinite series definition, i.e., the sum of all reciprocals of positive integers raised to the power given as the argument We assume it has been saved in a file *zeta.m*. This script should find an asymptotic value for zeta with an argument supplied within the code, in this instance **2**. The theoretical asymptotic value for $\zeta(2)$ is $\frac{\pi^2}{6}$, but we don't know the rate of convergence, or the processor time

needed. To periodically check the progress and confirm that the program is not stalled or executing an infinite loop, a line has been included within the *while* loop to display the series term number after every million iterations. This is followed by a keyboard command to allow us to check the current sum from a command line prompt and terminate the loop if we think a sufficient value has been reached . The script can be run in debug mode with *Debug -> Save and Run* from the Editor toolbar

C:\matlabR12.1\work\zeta.m*

File Edit View Text Debug Breakpoints Web Window Help

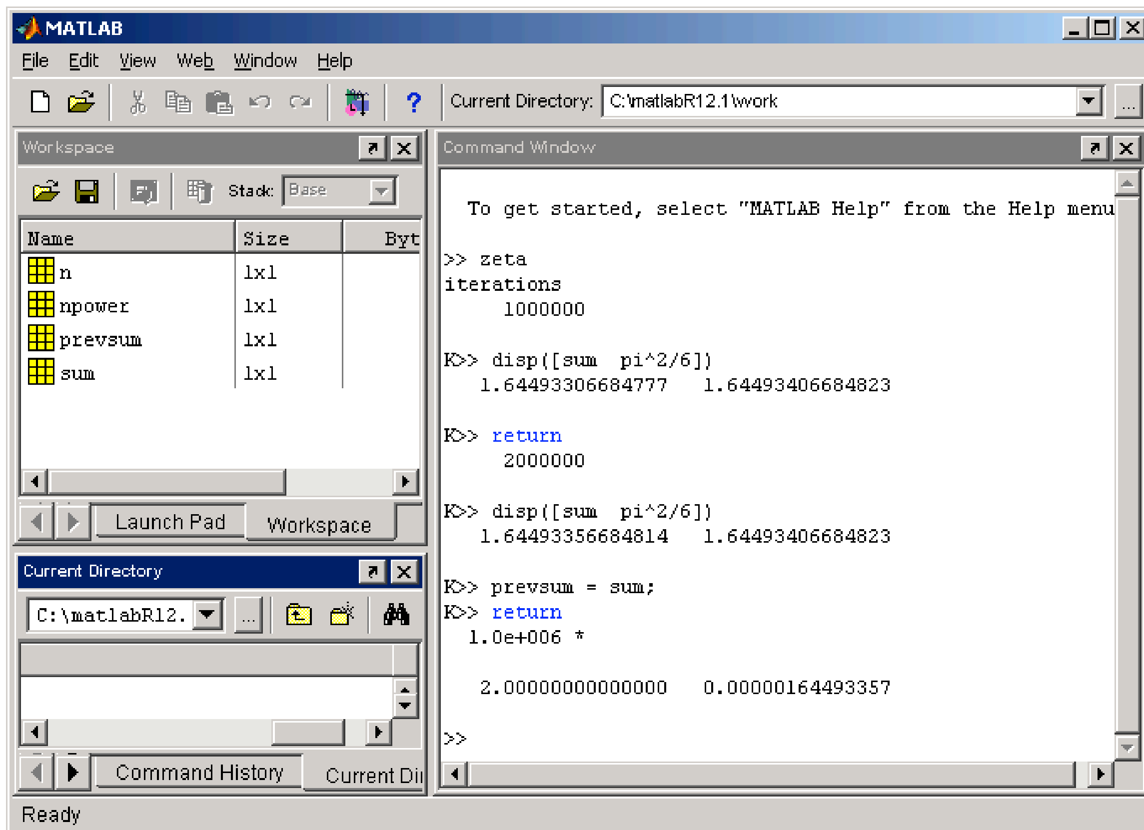
Step F10
Step In F11
Step Out Shift+F11
Save and Run F5
Go Until Cursor
Exit Debug Mode

Stack: Base

```
 1
 2
 3
 4
 5
 6
 7    % asymptotic evaluation of the Riemann zeta function
 8 -  sum = 0;            % initialize the asymptotic sum
 9 -  prevsum = -1;       % initialize a previous sum
10 -  npower = 2;         % specifiy the zeta function parameter value
11 -  n = 1;              % initialize the series term counter
12 -  disp('iterations')
13 -  while (sum ~= prevsum)            % test for convergence
14 -     prevsum = sum;                 % reset the comparison value
15 -     sum = prevsum + n^(-npower); % add series term velue
16 -     n = n + 1;
17       % include a monitoring/debugging instruction
18 -     if (n == 1e6*floor(n/1e6))    % identify each millionth term
19 -        disp(n)                    % display the series term number
20 -        keyboard                   % start keyboard prompt input
21 -     end;
22       % return to regular algorithm
23 -  end;
24 -  disp([n sum])                    % display final term number and sum
```

Ready

From the keyboard let's check the sum a couple of times at millionth term intervals to confirm that we are really in a long process that indeed is converging to an expected value rather than being stuck in some unending computational process, then terminate the iteration when we reach an acceptable level of accuracy.




### *Using Matlab with External Code*

When developing programs to be run in Matlab or in another environment in which Matlab files can be called, there may be times when various interface tools will be needed. A wide variety of such tools exist, but naturally they will be platform-dependent or source code language-dependent. Thus, there are not really any generic methods. A developer will have to customize the tools for the particular circumstances.

### *Exchanging and viewing text information*

Matlab programs can communicate with and use external code in many circumstances. The simplest examples are transferring data to and from external files in the form of ascii numerical matrices with the *save* and *load* commands. Communication can also be on the level of examining and transferring ascii text source code. For example, consider a simple piece of ascii text

```
function f = conevolume(r,h)
% volume of a cone with base radius r and height h
f = pi*(r^2)*h/3;
```

saved in three formats in the Matlab Work folder: *conevolume.doc* (MS Word document), *conevolume.txt* (plain ascii text document), and *conevolumne.m* (Matlab m-file).
All can be opened from the Matlab Editor, but the MS Word document will contain binary code and not be useable. Similarly, all these files can be opened and viewed as text in MS Word. The MS Word text file can be used by Matlab if it is opened in an editor and saved as ascii or m-file with a ".m" extension.

***Compiling and calling external files from Matlab***

Matlab can compile external Fortran and C code into MEX (Matlab EXternal) files using the *mex* command. The executable is stored in a file with an extension that depends on the external platform. Examples are *filename.dll* for a Windows platform and *filename.mexsol* for a Sun Solaris OS platform. When compiled these external code files can be utilized just as if they were built-in Matlab scripts or functions. As an illustration, let's consider some code written in C that accomplishes the same task as the *conevolume.m* code above, compiled on a Windows platform. In order to avoid the confusion of having programs of the same name with various extensions in the same folder or directory, we will name the source code for this C version *cvolume.c* to distinguish it from *conevolume.m*

```
/*=====================================================
=
 * CVOLUME.C volume of cone from base radius and
height
 *   Calling syntax:     v = cvolume(r,h)

*=====================================================*/

#include <math.h>
#include <mex.h>

/* Input Arguments */

#define   R_IN prhs[0]
#define   H_IN prhs[1]

/* Output Arguments */

#define   V_OUT     plhs[0]

#define PI 3.14159265

static void cvolume(
             double v[],
             double r[],
             double h[]
             )
{
```

```
    v[0] = PI*r[0]*r[0]*h[0]/3;
    return;
}


void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )

{
    double *v;
    double *r,*h;

    V_OUT = mxCreateDoubleMatrix(1,1, mxREAL);

    /* Assign pointers to the various parameters */

    v = mxGetPr(V_OUT);
    r = mxGetPr(R_IN);
    h = mxGetPr(H_IN);

    /* Do the actual computation in a subroutine */
    cvolume(v,r,h);
    return;

}
```

You can see that the C code is quite a bit longer and more complex, so normally you would just use the Matlab programming language to do such a simple calculation. However, there may be occasions when execution time can be improved by using compiled code of another language. This situation can occur, for example, when the Matlab code involves extensive looping or iterating. Also, some other languages may have libraries available for specialized purposes or functions.

To compile this code, you will need to setup for MEX files if you have not done so already. At the Command Window Matlab prompt, type

```
>> mex -setup
```

and you will subsequently be prompted for information about your computer's operating system and the compiler that you want to use. Unix systems usually have Fortran and C compilers such as *f77* or *gcc* installed, but Windows platforms may not come configured with any. Thus, Matlab has a bundled compiler, *Lcc*, that is used as the default C complier if no other is specified. Unfortunately the current release of Matlab does not have a bundled Fortran compiler. After finishing the setup, you should be ready to compile programs, at least ones written in C. The sytax for compiling is to give the file name and extension as the target for the *mex* command, e.g.

```
>> mex cvolume.c
```

to compile the C code given in the illustrative example. If you do this on a Windows platform, you will generate an executable *cvolume.dll*. If you compile it on a Solaris platform, you will generate an executable *cvolume.mexsol*, etc.

Once you have done the compilation, you can use the command *cvolume* at a Matlab prompt and get the same answer as you get using the command *conevolume* whose source is a Matlab language m-file. The volume of a cone which has a base radius **1** and a perpendicular height **1** is computed by these two codes as

```
>> volume_Matlab = conevolume(1,1)

volume_Matlab =

     1.0472

>> volume_C = cvolume(1,1)

volume_C =

     1.0472
```

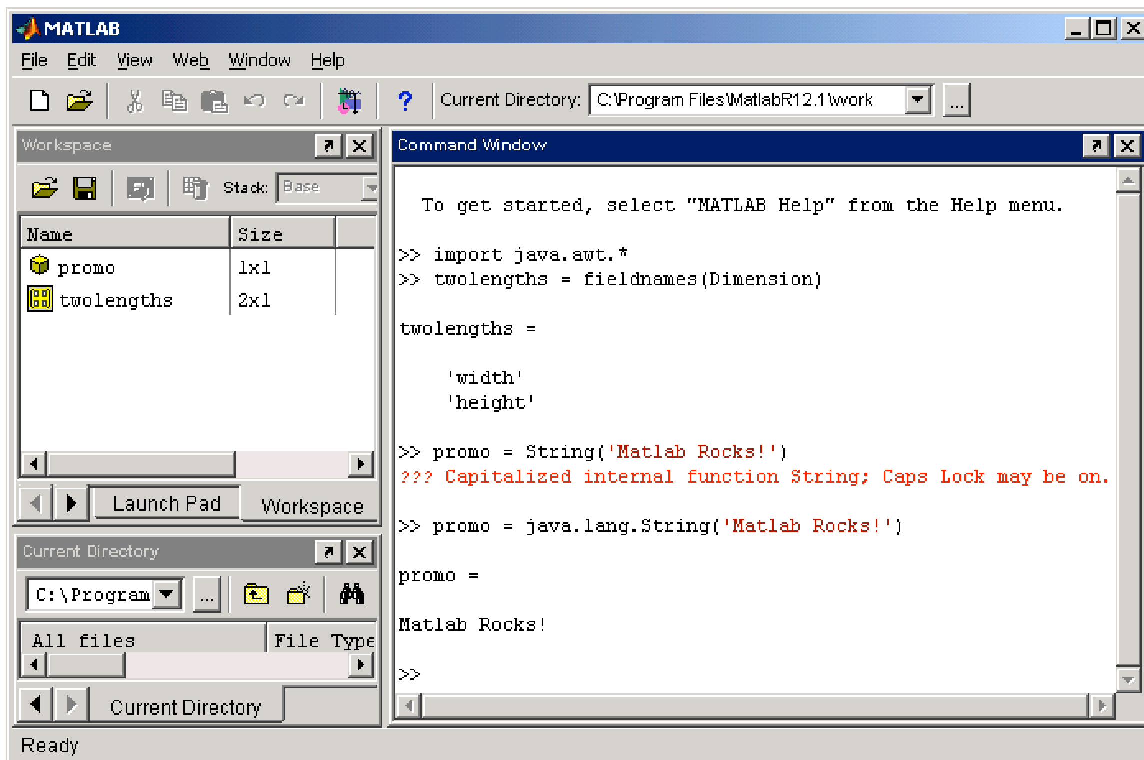
The current distribution of Matlab has a more elaborate example based on a three body orbital dynamics problem, which consists of an m-file *yprime.m* and an equivalent C code version *yprime.c* that can be compiled as a MEX-file. Those example files are located in the directory or folder *Matlab_root -> extern -> examples -> mex*

***Calling Matlab objects from external progams***

Matlab has an engine with functions that allow it to act as a callable external function or subroutine library in Fortran and C programs. The functions open the Matlab engine, close the engine, get and send matrix arrays, and evaluate Matlab strings. There is extensive information about these in the built-in Matlab help utility, with examples of Fortran and C programs that call the Matlab engine. External Fortran and C programs can also call binary Matlab files such as those created by default with the *save* command. The *save* command with no format flag will save the current workspace information in a binary file format (MAT) with a default name *filename.mat*, or the default *matlab.mat* if no file name is supplied. MAT file information can be accessed by external programs with commands similar to those for the Matlab engine. There is extensive information about the commands for exchanging data with external programs using MAT files in the built-in Matlab help utility.

***Using Java Classes in Matlab***

The desktop interface that is used for Matlab starting with kernel version 6 was developed and written in Java. As a result, Matlab now has an integrated Java Virtual Machine (JVM) and Java classes can be used directly in Matlab at the command line or within function m-files. Before usage, there must be a file with the name *classpath.txt* in the Matlab search path. Matlab has a default file located in the folder or directory *Matlab_root -> toolbox -> local*. This can be copied to a directory in the Matlab path and edited if the programmer has created classes other than those in the default list.
A class or a package of classes can be imported into the workspace with the *import* command. If this has not been done for a particular class, that class has to be referenced by its complete name. As an illustration, consider a simple example where the default *classpath.txt* file has already been copied to the current working directory and the Java *awt* class package has been imported but the Java *lang* class package has not. In this case we can use a class name from the imported package directly on the command line but must give the complete full name for a class in the package which has not been imported




The *Dimension* class is recognized because its package has been imported, but the *String* class is not recognized as a Java class because its package has not been imported. However, if the complete name of the *String* class, i.e., *java.lang.String*, is used then the Matlab interpreter recognizes it as a Java class.

# Part III: Graphics and Data Analysis

# Section 7: Graphics and Data Visualization

Matlab has a high level graphics capability that allows users to display data in various forms without having to incorporate extensive information into a command or into scripts. This easy procedure uses default values for graphical objects in Matlab's object oriented graphics system, Handle Graphics. For more customized and advanced use, the values can be specified or changed on the command line or in the text of m-file scripts; and with the release of the Matlab 6 kernel there is also a point and click GUI for users to change object values in order to alter display characteristics. Display possibilities include 2-D plots, 3-D plots, visual aids such as pie charts and histograms, contours, and animation. In addition there are many attributes of objects that can be customized, including scaling, colors, fonts, perspective angles, lighting and shading, and so forth.

***Two dimensional plotting***

The high level graphics for two dimensional plotting accommodate displays of pairs of data sets in rectangular linear Cartesian coordinates, on a semilog axis system, on a log-log axis system, and in polar coordinates. The elementary syntax is

```
>> plot(x,y)         % linear abscissa and ordinate
>> semilogx(x,y)     % logarithmic abscissa
>> semilogy(x,y)     % logarithmic ordinate
>> loglog(x,y)       % logarithmic abscissa and
ordinate
>> polar(theta,rho) % polar graphing
```

If only one vector argument is supplied, it is considered to be the second (dependent) variable, i.e., the ordinate or radial distance, and the sequence position in the vector is used for the corresponding independent variable, i.e., abscissa or counterclockwise angle. When a vector containing complex valued quantities is plotted, Matlab ignores the imaginary part and the display represents only the real part. Trailing character string options in the argument list can specify a line color or type. For example:

```
>> t = (1:0.1:10);
>> u = (1:0.1:10);
>> x = sin(t)./t;
>> y = cos(u)./u;
>> plot(t,x,'r*')
```

would give a 2-D plot of a damped sine wave with points displayed as red asterisks. We could obtain a 2-D plot of a damped cosine wave in a similar manner. There is a toggle for getting a new plot (hold off) and for superimposing on an existing plot (hold on). Assume that we want a second plot superimposed on the first with the same scaling but with the points displayed in blue circles. Then we would use the additional commands

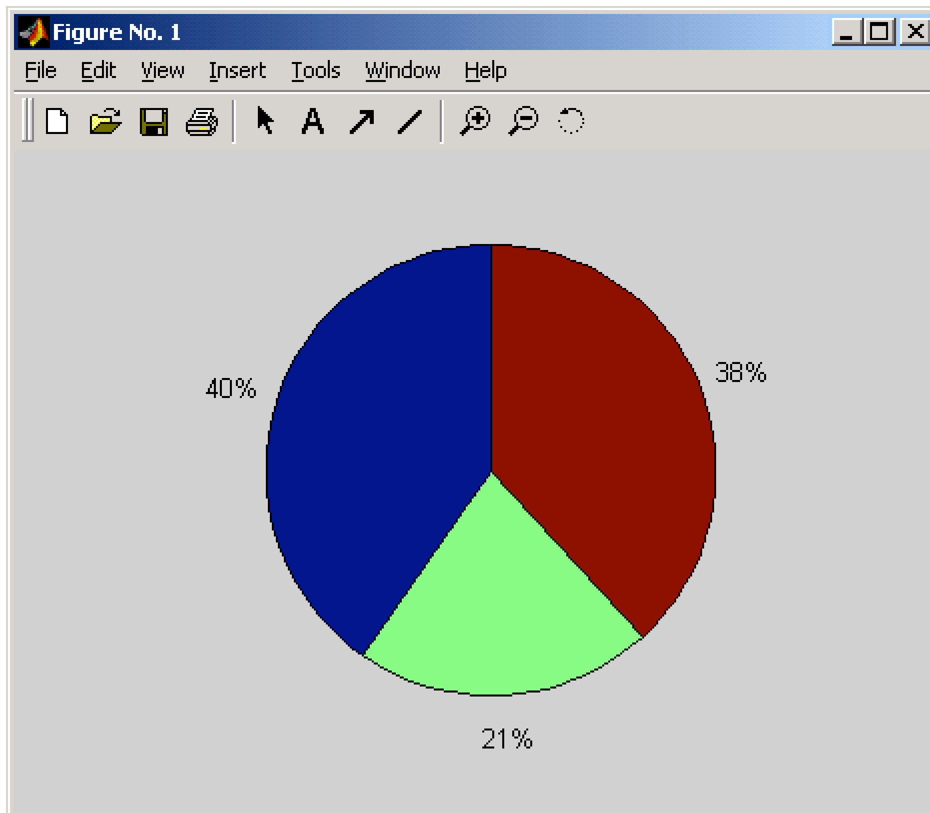
```
>> hold on;
>> plot(u,y,'bo')
```

There are also several display options for discrete data that are commonly used in presentations for general or business audiences

bar    for a vertical bar graph
barh  for a horizontal bar graph
stem  for a stem plot
area   for display of vectors as stacked plots
pie     for a pie chart
hist    for a histogram in cartesian coordinates
rose   for a histogram in polar coordinates

As a simple illustration, load the file planets3.txt from section 3 and label its first column as radii, then plot a pie chart:

```
>> load planets3.txt;
>> radii = planets3(:,1);
>> pie(radii)
```




***Three dimensional plotting***

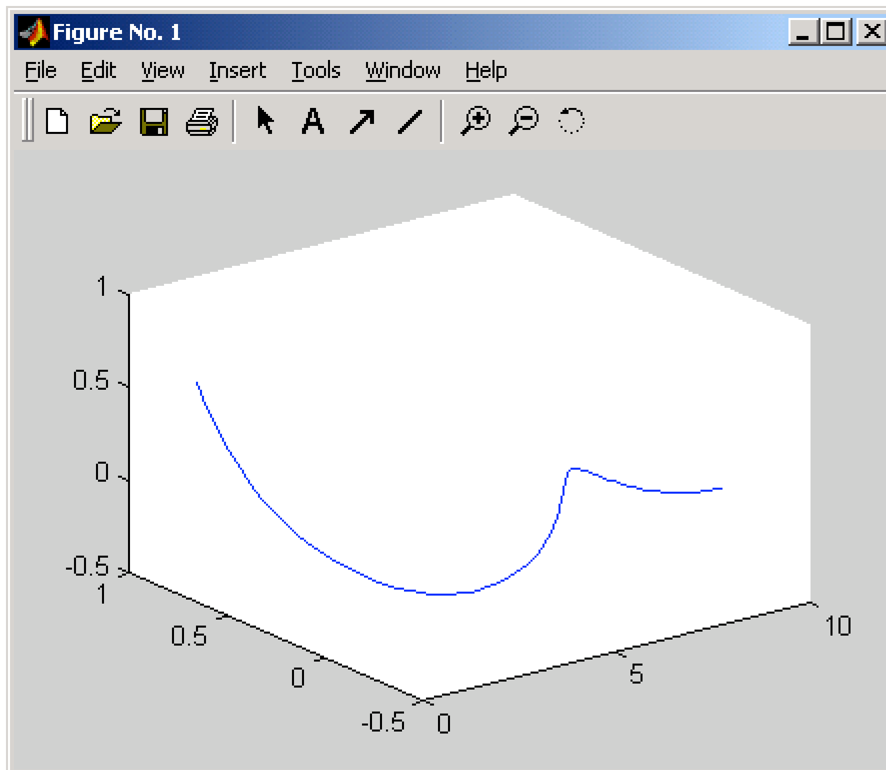
Matlab has several commands used in displaying data in three dimensions. Among these are routines for displaying lines with a 3-D perspective, making surface plots of a function of two independent variables, and making contour plots.

```
>> plot3(x,y,z)  % line through three dimensional
space
>> mesh(q)       % wire frame surface for q(x,y)
>> surf(q)       % quadrilateral surface rendering
```

The *plot3* command requires 3 vector arguments, the first two forming a grid of independent variable values and the third being the functional values obtained from the independent variable values. Thus, all three vectors must be the same length. In contrast, the argument for the surface plots require that the argument be a matrix whose values correspond to pairs of independent variable values in a grid. Using the same functions that we created to illustrate 2-D plotting, we can now plot a line in 3-D:

```
>> hold off;
>> plot3(t,x,y)
```

The screen display will use default values for observation angle




More perspective of the 3-D nature of the curve been be obtained by rotating it in 3-D by selecting the Rotate3D item on the Figure Window's Tool menu bar or by clicking the rotation button at the end of the Figure Window's icon tool bar, then moving the cursor to the figure area and moving the mouse around while depressing its left key.

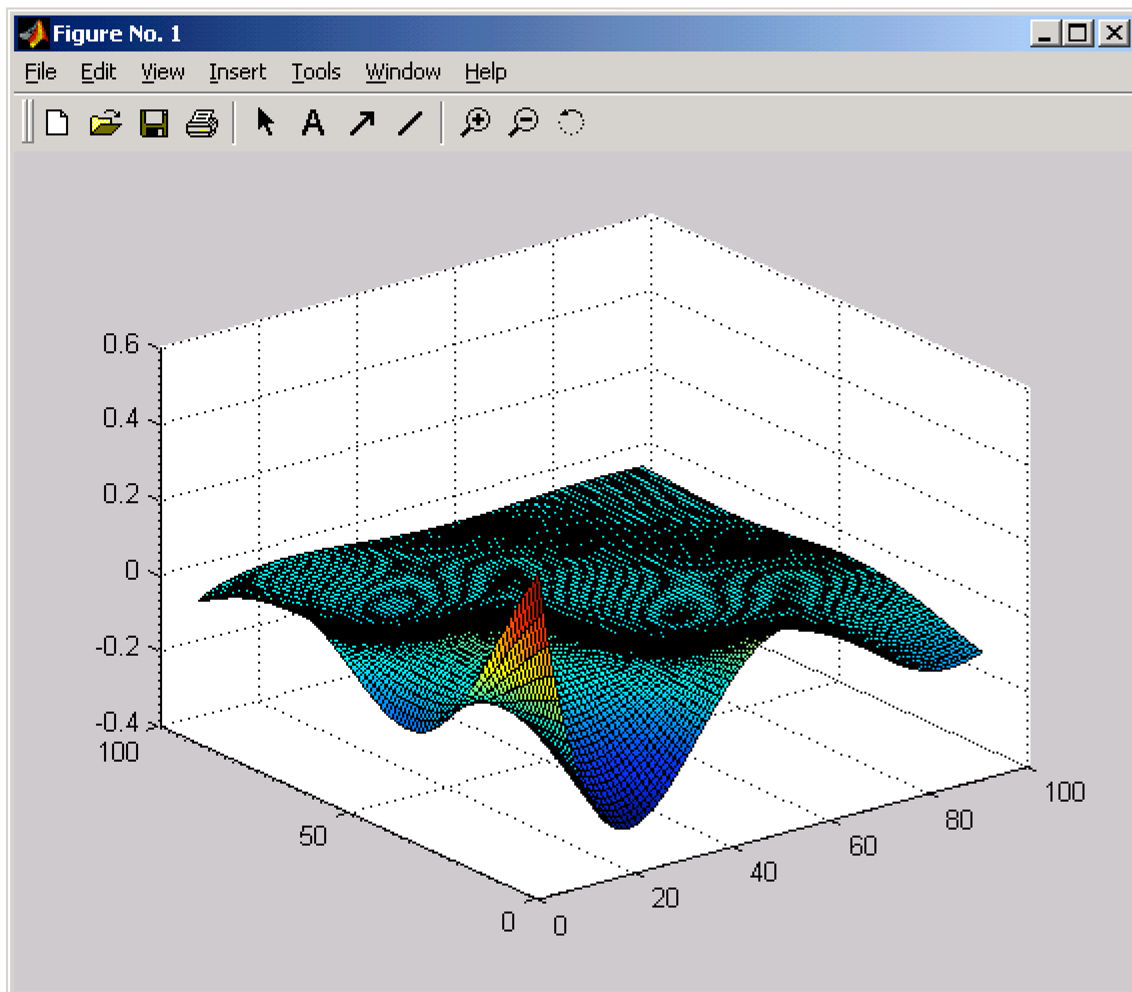
To illustrate surface plotting, let us form an underlying grid with the vectors **t** and **u** and then create a function **z** using both **x** and **y**:

```
>> for i=1:length(t)
       for j=1:length(u)
          z(i,j) = x(i).*y(j);
       end;
   end;
```

Then let us create a surface plot of the new function **z**:

```
>> surf(z)
```

The 3-D surface will subsequently appear in the Figure Window.




Just as with the line example, the surface can be rotated in 3-D using the same procedures. If a contour projection onto the grid plane is desired, then the function *surfc* can be used. Likewise, the commands

```
>> mesh(z)
>> meshc(z)
```

would create similar 3-D plots, but with the surface represented in a wire frame display.

***Animation***

Data animation in Matlab is accomplished by a sequential display of graphic frames. At an elementary level the frames can be simple plots of points within a Figure Window. If the *hold* value is set to *off*, then each point will appear by itself, creating a display similar to a moving cursor on the screen.. If the *hold* value is set to *on*, then then each frame will incorporate the data from previous frames and the display will show a tracking type of pattern. In some cases this method may seem a bit crude because of blinking as each frame replaces the previous one, and delays in generating frames if needed computations are extensive or complex. An alternative procedure is the use of the *movie* feature. For this, a set of frames is assembled ahead of time into a movie type variable using the *getframe* function. Subsequently, the sequence of frames can be played in the Figure Window using the *movie* command. As an illustration, let's use the **x** and **y** vectors that we have created above and create an animation of **x** versus **y** with the simple overlay method but assembling a movie file simultaneously:

```
hold on;
for k = 1:length(x)
    axis([-1.5 1.5 -1.5 1.5]);
    plot(x(k),y(k),'b*');
    M(k) = getframe;
end;
pause;
clf;
movie(M)
```

In this example the previously plotted points are retained, and as each point is plotted the current figure is saved as a frame for a movie with the *getframe* value. The *pause* command will hold the final picture until the next keyboard activity and the *clf* command will clear the Figure Window so that the movie will start with a blank screen.

***The Handle Graphics system***

Handle graphics is Matlab's object oriented system for handling components used in constructing graphical displays. The objects are the basic drawing elements. Each object is identified with a handle and the handle contains information about the object's characteristics and properties. Specific values of object properties can be edited using the handles. There are two specific built-in graphics handles that are used for default plotting. The handle *gcf* refers to the current figure and its information is used for generating a Figure Window when none exists at the time a command producing graphical output is issued. The handle gca refers to the current axes and likewise its information is used for constructing axes when none exist at a time a plotting command is

issued. The handle *gco* is refers to a graphical object that has been selected in the Figure Window. It will be empty if there is no Figure Window present or if an object has not been selected. If a figure object has been selected then information about that object's properties will be present in *gco*.

The current values within these handles can be obtained with the commands

```
>> get(gcf)
>> get(gca)
>> get(gco)
```

Current values may be edited using the syntax *set(handle,'Property',value)*. For example, the property specifying the color background within the axes has the name 'Color' and its default value is white ( 'w' or [1 1 1]). If we want to change this color to yellow we could do so by selecting the graph axes box within the Figure Window and then giving the command

```
>> set(gca,'Color','y')
```

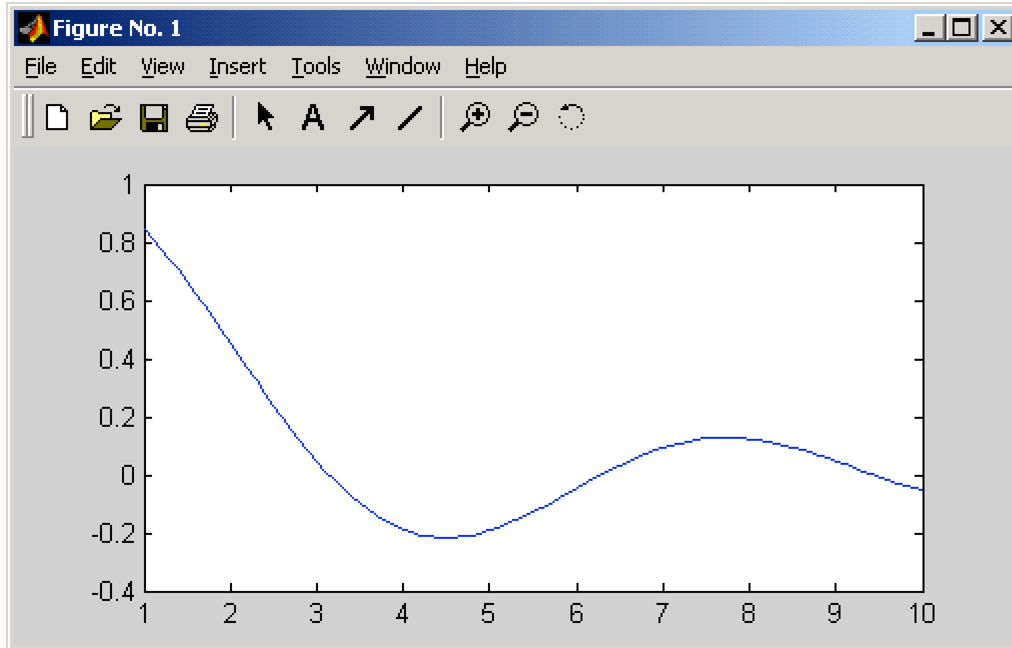
Similarly, the character of other graphics properties can be edited within the appropriate handle.

***Customizing displays***

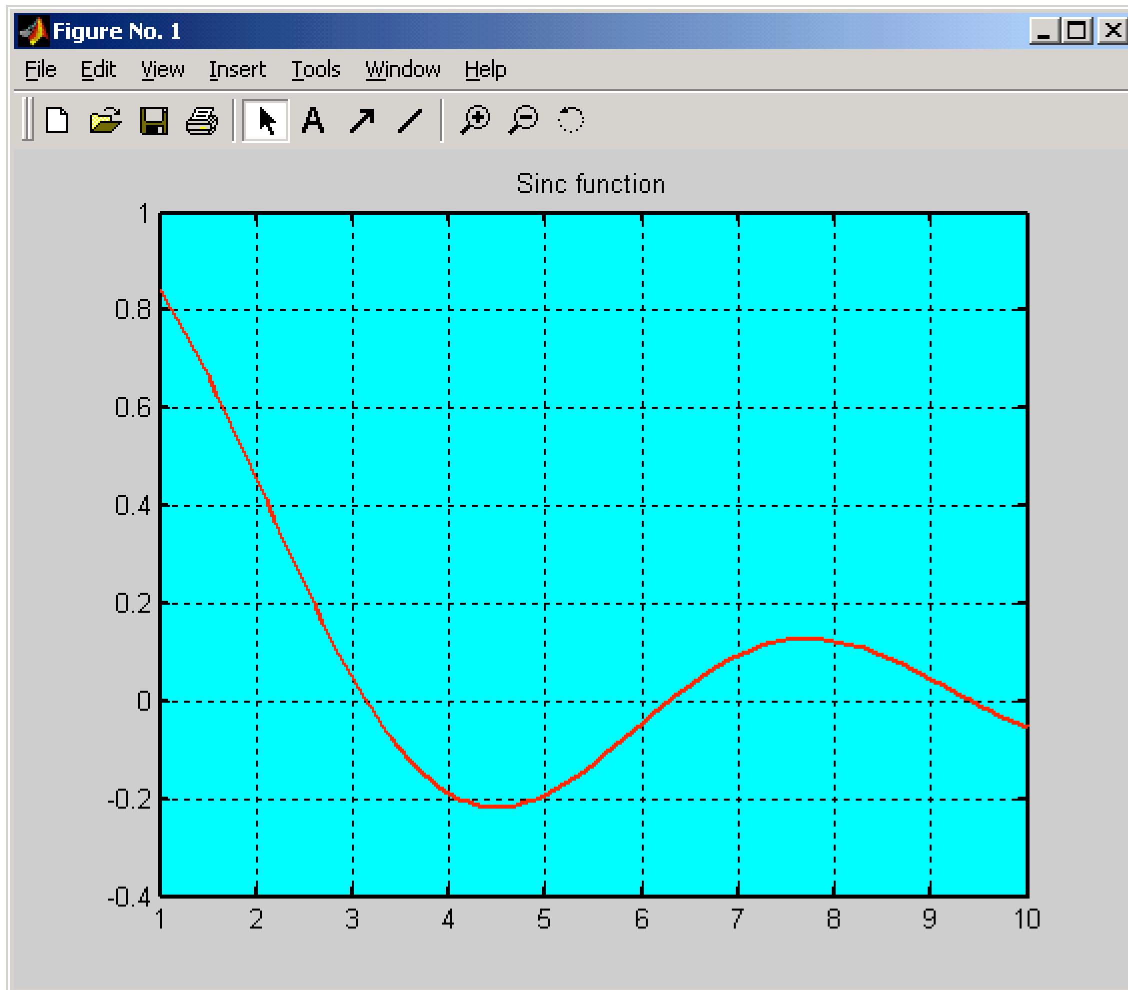
When first generating a plot, the default values in the *gcf* and *gca*, or whatever editing to them has been done, are used in constructing the display in the Figure Window. As an alternative to editing the handles, Matlab has a point and click interface for editing some of the more standard attributes. As an illustration of the GUI method and the handle editing method, let's start out with a simple figure plot with default values to display the vector x that was created in the 2-D plotting example. We first need to clear the display and then give the simple plot command

```
>> clf
>> plot(t,x)
```

In this case the default values in the handles will produce a plot figure where there is no title, the axes scaled to the data range, inward tick marks with values on the left and bottom, and the data represented by a solid blue line.




For point and click editing, click on the backward arrow in the icon bar underneath the toolbar, then make a selection from the Edit menu. In this example, let's choose Axes Properties. Selecting this will bring up a Property Editor Window. The Scale tab should be viewable at first and on this tab, let's check the boxes to show the Grid on the X axis and on the Y axis and then click on the Apply button. Then click on the Style tab and in the Axis line width menu, select 2.0 to replace the default 0.5 to give thicker axis lines and select Cyan from the Color background menu to change from the default white. After clicking the Apply button, the changes should be apparent in the Figure Window: Now click on the Labels tab and type in some text for a Figure title, for example "Sinc function", since this equation being plotted is that of the so-called sinc function , i.e., the function *sinc(t) = sin(t)/t*. Again click on the Apply button to make the change appear. Now go back to the Figure Window and click on the plotted line to select it, then choose Current Object Properties from the Edit menu. Now a Property Editor window for the selected line will appear. The style tab should appear on top. Change the Line width selection menu from the default 0.5 to 2.0 again. Then let's change the Color menu from the default "Blue" to "Red". These changes also take effect by clicking on the Apply button. At this point the figure should look like the following:



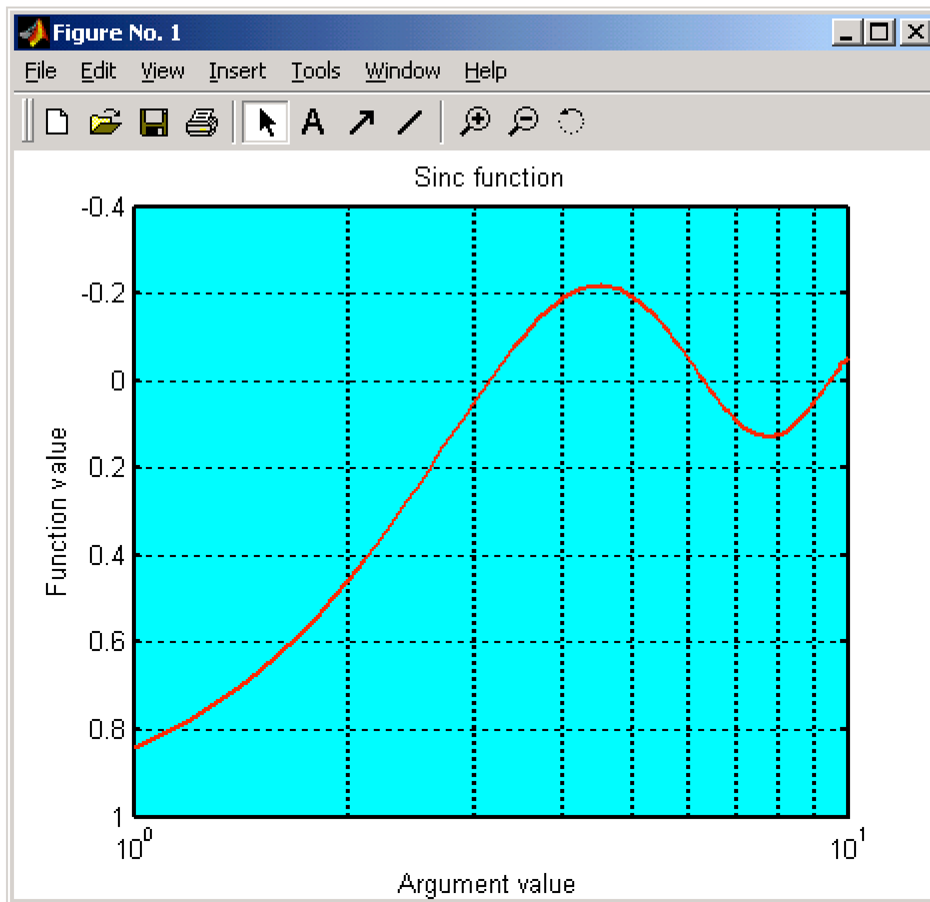

Next, let's make a couple of changes by direct handle editing:

```
>> set(gca,'XScale','Log');
>> set(gca,'YDir','Reverse');
>> set(gcf,'Color','w');
```

This will change the abscissa to a log scale and will reverse the direction of the ordinate and change the figure background from gray to white. Finally, let's make a couple of changes with the graphic commands *xlabel* and *ylabel* at the command prompt

```
>> xlabel('Argument value');
>> ylabel('Function value');
```

to add labels to the axes. We now have a final customized display




## Section 8: Data Analysis

***Data analysis functions***

Matlab was not developed as a statistical package, yet there are some elementary statistical functions built into the kernel and there is a specialized Statistics Toolbox available for purchase from Mathworks but which is not currently included in the license for Matlab on the ITS servers. Among the functions distributed with Matlab itself are

| | |
|---|---|
| mean | for mean or average value of elements |
| median | for median value of elements |
| min | for smallest component |
| max | for largest component |
| std | for standard deviation from the mean of elements |
| sum | for sum of elements |
| prod | for product of elements |
| sort | for sorting elements in ascending order |
| sortrows | for sorting rows in ascending order of first column value |

cov for variance of a vector or covariance of a matrix
corrcoef for correlation coefficient
The sorting functions are by absolute value when complex numbers are involved and alphabetically or by ascii character code value when character strings are involved. To illustrate some of these we will use the simple matrix file with planet radius and mass data described in Section 3. First let's load the data and sort it

```
>> load planets1.txt
>> radiussort = sort(planets1)
```

Then let's find the mean and the standard deviation of the densities of earth, mars, and venus in kg^10-24/km^3 even though the small sample size of 3 planets will lead to quite a substantial variance.

```
>> density =
3*planets1(:,2)./(4*pi*(planets1(:,1)).^3);
>> meandensity = mean(density)
meandensity =

      4.8928e-012

>> densitystd = std(density)

densitystd =

      8.5130e-013
```

***Regression and curve fitting***

Matlab has a function *polyfit* for fitting curves with polynomial regression and the function *polyval* for evaluating the polynomial fit. The *polyfit* function has syntax

```
>> polyfit(var1, var2, order)
```

where **var1** and **var2** are equal length vectors and **order** is the order of the polynomial to be used in the fitting. For illustration, let's create a noisy quadratic function and see how the *polyfit* function works.

```
>> t = [0:1:10];
>> for i=1:11
       y(i) = (1 + 0.1*rand(1) - 0.05).*((t(i)).^2);
   end;
```

Then we try fitting this function **y** with a linear and with a quadratic regression on **t**.

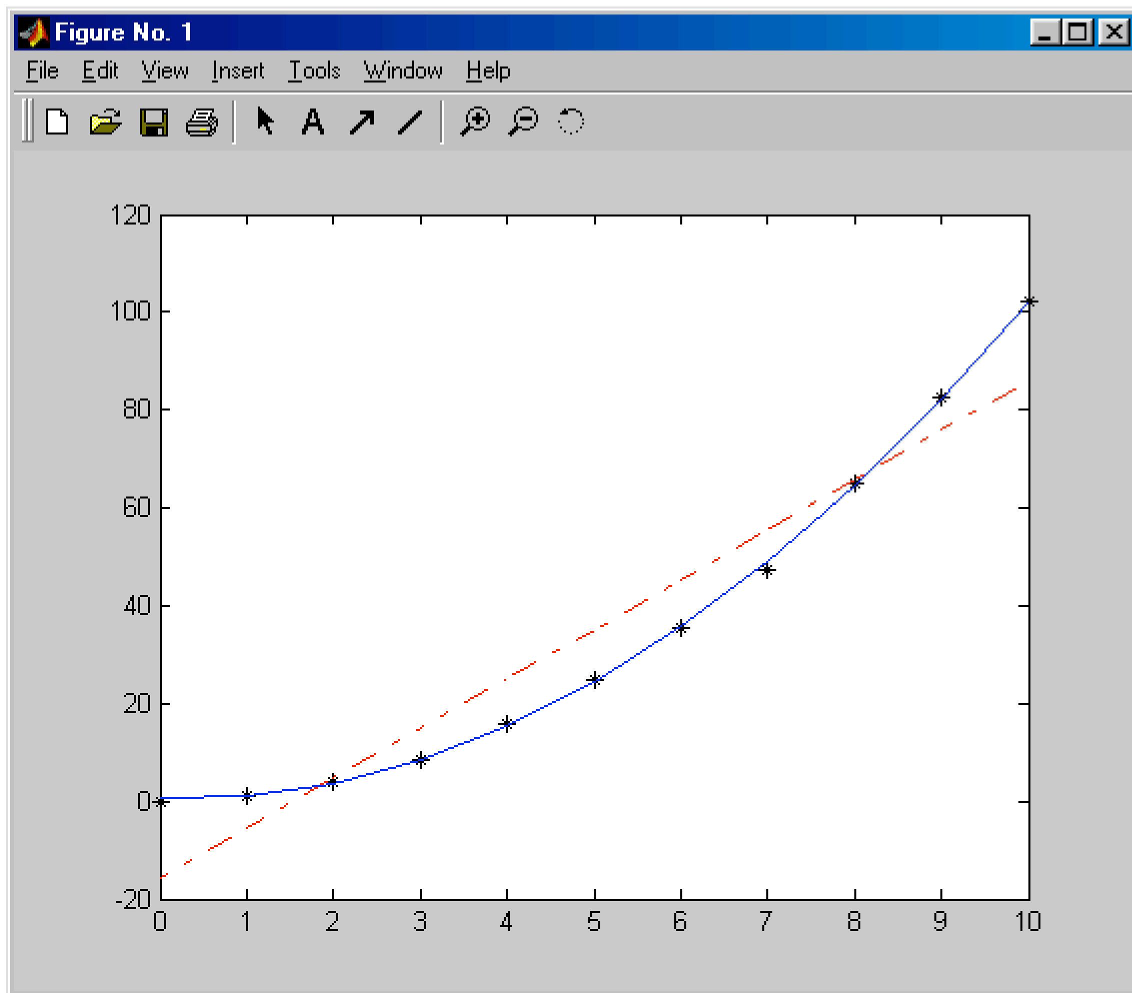
```
>> ylintest = polyfit(t,y,1);
>> yquadtest = polyfit(t,y,2);
```

These new variables are vectors containing coefficients for a power series in **t** which give the best fit to the actual **y** data, in descending order such that the coefficient of the highest power comes first and the constant term, i.e., the **y** intercept, comes last. The theoretical values for y based on the best fit polynomial can be obtained with *polyval*:

```
>> ylinfit = polyval(ylintest,t);
>> yquadfit = polyval(yquadtest,t);
```

A plot of the actual data and the regression fittings will show that the quadratic fit is the most accurate, as expected.

```
>> plot(t,y,'k*');        %black asterisks for data
>> hold on;
>> plot(t,ylinfit,'r-.');%red dash-dot line for linear
>> plot(t,yquadfit,'b'); %blue solid line for
quadratic
```



This can also be seen by looking at the sum of the squares of the residuals:

```
>> res2lin = sum((ylinfit - y).^2)
>> res2quad = sum((yquadfit - y).^2)
```

The exact values will depend on the seed used for generating random numbers in the original construction of the data, but the sum for the quadratic fit should be around 2% of that for the linear fit.

### *Signal and image processing*

Matlab has a Signal Processing Toolbox that contains function M-files that are algorithms for implementing several signal processing tasks. In addition a few of the most elementary functions, for example the finite fourier transform *fft*, are included in the basic Matlab distribution.. The Toolbox has specialized functions related to filtering, waveform generation and spectral analysis, along with a special GUI called SPTool. The most elementary filtering function, *filter*, is included in Matlab and has the syntax

```
>> output = filter(numcoeff, denomcoeff, input)
```

where an output vector is formed from filtering an input vector with a transfer function whose numerator coefficients are contained in the vector *numcoeff* and whose denominator coefficients are contained in the vector *denomcoeff*. As an example, consider a simple input vector x(n) = [1 3 5 7] for n = [1 2 3 4] and a filter with transfer function

$$h(n) = \frac{1 + n^{-1}}{1 + 2n^{-1} + n^{-2}}$$

We could obtain the filtered function y(n) = h(n)x(n) with the commands

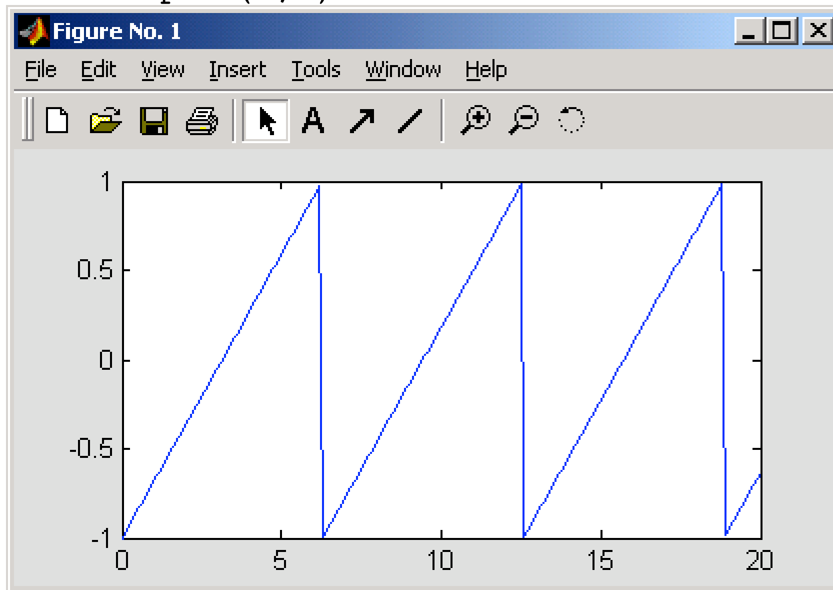
```
>> x = [1 3 5 7];
>> num = [1 1];
>> denom = [1 2 1];
>> y = filter(x,num,denom)

y =

     1     4     8
```

Matlab has several wave generation functions. Some of the most commonly used are

sawtooth for a triangle wave generator
pulstran for a pulse train generator
square for a square wave generator

As an illustration, let's generate a triangular, sawtooth shaped wave:

```
>> t = [0:0.1:20];
>> x = sawtooth(t);
>> plot(t,x)
```

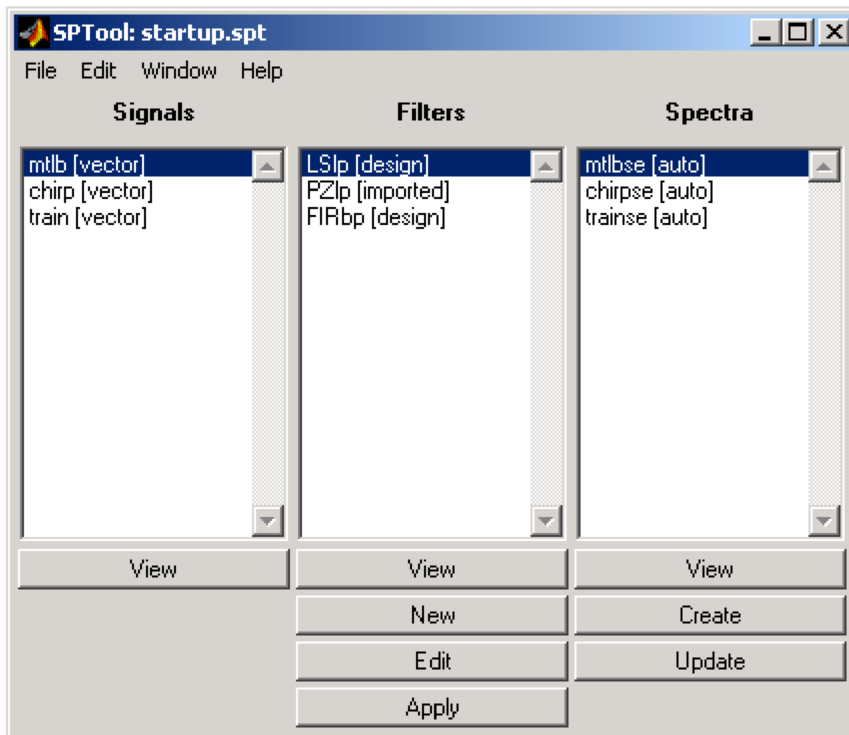



The sawtooth function is defined as -1 at multiples of $2\pi$ and linear with a slope of $1/\pi$ at all other points.

The Signal Processing GUI is launched with the command

>> SPTool

which creates a window with selections of  signals, filters, and spectra




The view buttons on this window can be used to launch a Signal Browser, a Filter Viewer, or a Spectrum Viewer.

There is another Toolbox with M-file functions for manipulating arrays containing image information, the Image Processing Toolbox. Also, many of the Signal Processing Toolbox functions can be used in conjunction with image processing. Within the Image Processing Toolbox are routines for geometric operations, image analysis and enhancement, and region of interest operations. But before working on an image, it has to be imported into the workspace. This can be done with the *load* command if the image data has already been stored in a MAT file. Images with supported formats (e.g., jpg, gif, tiff, png) can be imported with the *imread* command and displayed in the Figure Window with the *imshow* command. Once an image has been processed as desired, it can then be exported in a supported format using the *imwrite* command. As an example, let's import a low contrast image. tower.jpg, enhance its contrast, and export a comparison image. For this exercise, you will need to put the image file in your Matlab path. The process can be done with any image with a supported format, but in this case we choose a low contrast nighttime picture of the University tower, available for download at

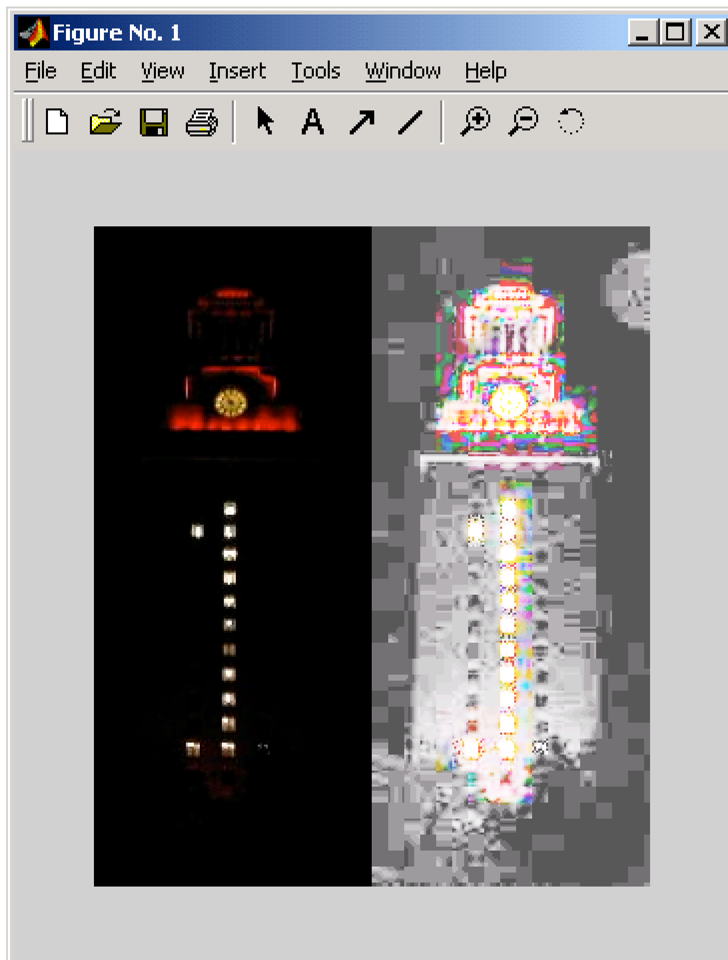
http://www.utexas.edu/cc/math/tutorials/matlab6/tower.jpg

```
>> I = imread('tower.jpg'); % create image data matrix
>> imshow(I)                % display in Figure Window
```

Now let's add contrast with the *histeq* command. This spreads out intensities over the entire possible range, and is particularly useful if actual intensities of all the pixels are clustered over a narrow range.

```
>> J  =  histeq(I); % create new image with contrast
>> K =  [ I J ];    % place old and new side by side
>> imshow(K)        % display the contrast
>> imwrite(K,'towercontrast.jpg') % write new file
```

Now there should be a new display in the Figure Window showing both the original and contrast enhanced picture side by side




and there should also be a new image file, towercontrast.jpg, in the current directory. In this exercise the data matrix is 3-dimensional, as can be seen in the *Size* column of the Workspace Window. This is an RGB (red, blue, green) format, in which the primary color component is the third index. The first two indices specify the vertical and horizontal position of the pixel, top to bottom, left to right. Element values represent intensities. Thus, for example, displaying an element in the original image data matrix

```
>> disp(I(50,50,2))
    13
```

tells us that the green intensity of the pixel in the 50th row and 50th column has an intensity value of 13.

This is not the only format for storing image data information. Image objects in Matlab can also be two dimensional data arrays. The indices correspond to pixels in a rectangular image and the elements can be either magnitudes of intensity or pointers to color or intensity information in a separate color map matrix. Color map matrices are three columns representing components of the primary colors red, blue, and green. Each row is a particular weighting of each of these in a range from **0** to **1**. Indexed matrices, whose elements point to the color map, will have entries corresponding to particular lines in the map. The image whose pointers are in matrix **X** will have its pixel content determined by the RGB values in the associated color map matrix **Y** using the command

```
>> image(X), colormap(Y)
```

As a small example, let's create a game board for chess or checkers. First we will create the *8x8* layout of the board. Pixels whose row and column sum is odd need a different color from those whose row and column sum is even. But the color within each set needs to be uniform, so only two lines are needed in the color map. First create the matrix with pointers to alternating colors

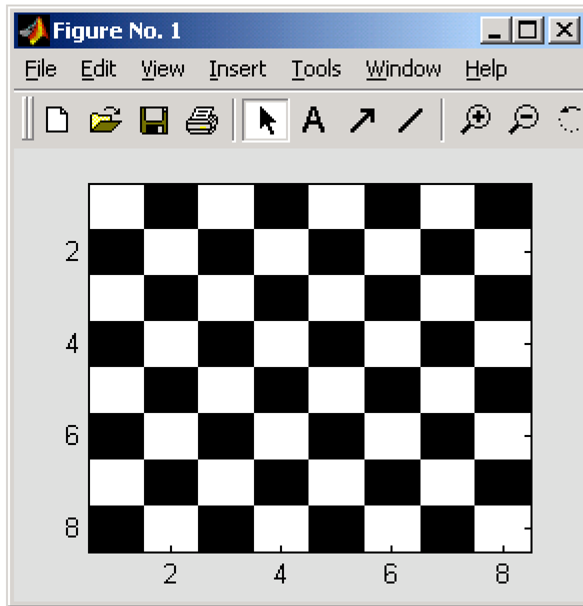
```
>> Z = [eye(2) eye(2) eye(2) eye(2)];
>> C = 2*ones(8) - [Z;Z;Z;Z]
```

Now **C** is the matrix of color map pointers for a checkerboard image. Next we create a color map with three columns for RGB values with two rows, one for each of the two colors whose pointer values are **1** or **2**. Let's define the color map as the variable *colors*.

```
>> colors(1,:) =  [ 1 1 1 ]; % pointer value 1 white
>> colors(2,:) =  [ 0 0 0 ], % pointer value 2 black
```

The *image* command will create a color pixel image, with the color map specified with the *colormap* command.

```
>> image(C)
>> colormap(colors)
```




For a more intricate image with a more extensive color map, you can look at one of the example files distributed with Matlab. One such binary file is *mandrill.mat*, a facial image of a gorilla, whose data includes the matrix **X** with pointer data and the color map matrix *map*. Thus, the commands needed for viewing are

```
>> load mandrill
>> image(X)
>> colormap(map)
```

Another type of image format is one in which the elements of the image matrix are intensities rather than pointers to a mapping. The command to generate an image from this format is *imagesc*. We can see our previous checkerboard image from matrix **C** again with the command

```
>> imagesc(C)
```

Images can be manipulated by changing their data matrices or color maps. For example we can change the black and white checkerboard to a red and cyan one with the command

```
>> colors(:,1) = sort(colors(:,1))
```

or we can remove the blue component of the color map used for the gorilla image with

>> map(:,3) = 0;

Demonstrations of many other ways for manipulating and analyzing images are available in the documentation that comes with Matlab. Expand the Image Processing Toolbox line in the Launch Pad Window and double click on *Demos* to see these.

# Part IV: Modeling and Simulation

## Section 9: Modeling and Simulation

Matlab has several auxiliary Toolboxes distributed by Mathworks, Inc., that are useful in constructing models and simulating dynamical systems. These include the System Identification Toolbox, the Optimization Toolbox, and the Control System Toolbox. These toolboxes are collections of m-files that have been developed for specialized applications. There is also a specialized application, Simulink, which is useful in modular construction and real time simulation of dynamical systems.

### *System Identification*

The System Identification Toolbox contains many features for processing experimental data and is used for testing the appropriateness of various models by optimizing values of model parameters. It is particularly useful in working with dynamical systems data and time series analyses. This toolbox is included in the Matlab installations on all the ITS servers. The identification process is a bit complex, but a guided tour through a simple example can be accessed with the *iddemo* command at a Command Window prompt.

### *Using the Control System Toolbox*

The Control System Toolbox contains routines for the design, manipulation and optimization of LTI (linear time invariant) systems of the form

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

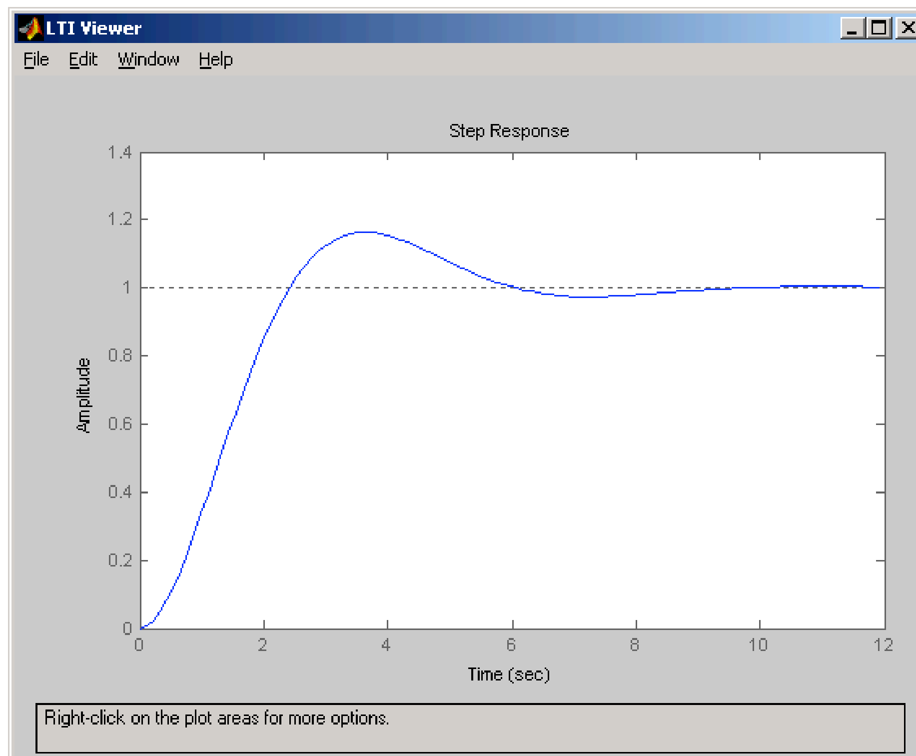
where matrices A, B ,C ,D are constant. It can be used individually or as a post-processing tool for a system created with Simulink. The Control System Toolbox also supports two auxiliary applications, the LTI Viewer and the SISO Design Tool. The LTI Viewer is basically used to plot graphs of the system response due to various inputs and the SISO Design Tool is used to design single input-single output systems, that is systems for which the input and output vectors have dimensions 1 by 1. These applications can be launched by double-clicking on the Control System Toolbox icon in the Launch Pad window of the default Matlab desktop. We don't comment on the SISO design tool since that would require knowledge of control theory, but we give an example of the use of LTI Viewer. Consider the system

$$A = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix},\ B = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ C = \begin{bmatrix} 0 & 1 \end{bmatrix},\ D = [0].$$

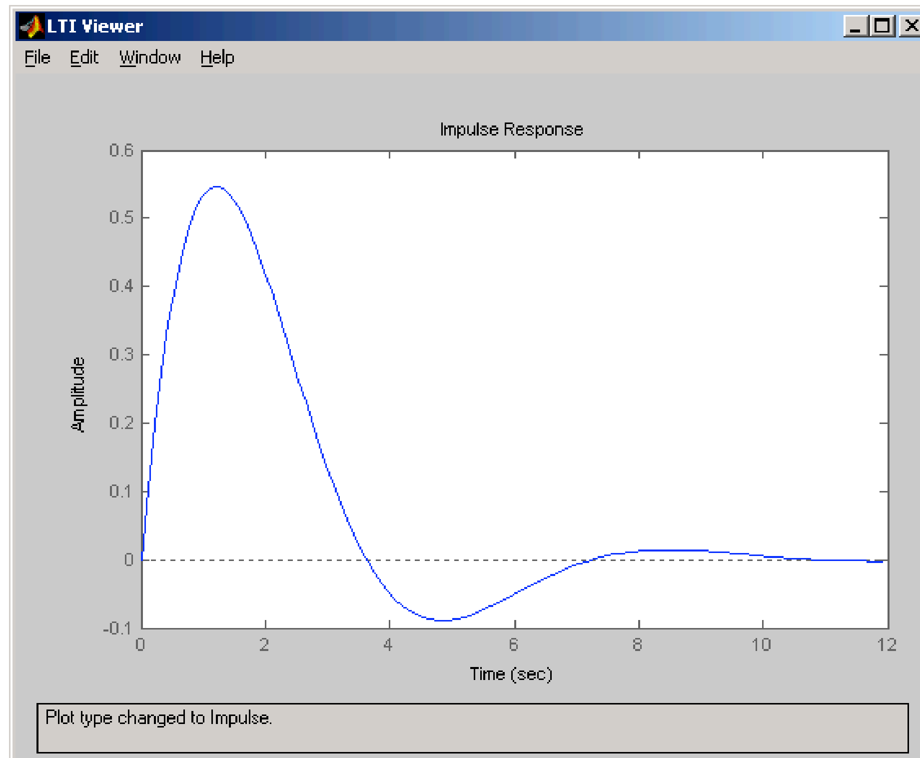
To import the system to the LTI Viewer, we create a system object using the **ss** command.

```
>> A=[0 1;-1 -1];
>> B=[0 1]';
>> C=[1 0];
>> D=0;
>> s1=ss(A,B,C,D)

a =
         x1   x2
   x1    0    1
   x2   -1   -1


b =
         u1
   x1    0
   x2    1


c =
         x1   x2
   y1    1    0


d =
         u1
   y1    0

Continuous-time model.
```

Here s1 is the object corresponding to our system. Next, select the options **New Viewer** and **Import** from the **File** menu, and then choose the object s1. The following figure will appear which shows the response of the system to a unit step input. By default the initial condition here is zero.

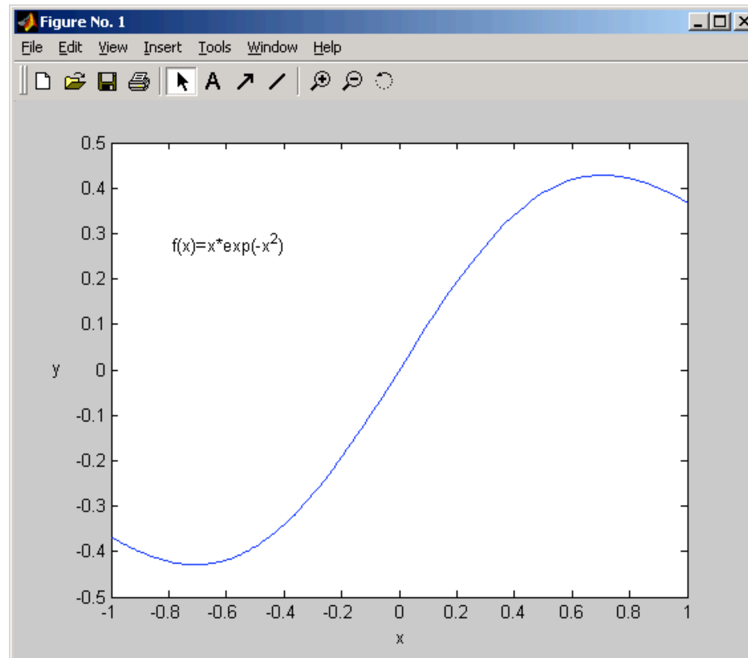



Next, right click on that figure, select the **Plot type** option and then the **Impulse** option, to get the following figure which is a plot of the response of the system to a unit impulse at time zero.




### *Optimization Toolbox*

The Optimization Toolbox offers a rich variety of routines used for the minimization and maximization of functions under constraints. We will describe only two simple and commonly used examples. The first one is **fminbnd** which calculates the location in a given interval at which a function attains its minimum. Note that the maximum of a function $f(x)$ is equal to minus the minimum of $-f(x)$, hence we can use **fminbnd** to compute locations of maxima of functions too. Suppose now that we want to compute the minimum and maximum values of $f(x) = x \cdot e^{-x^2}$ in the interval $[-1,1]$.




Then we type in the command line:

```
>> x=fminbnd('x*exp(-x^2)',-1,1)

x =

   -0.7071

>> x*exp(-x^2)

ans =

   -0.4289

>> x=fminbnd('-x*exp(-x^2)',-1,1)

x =

    0.7071


>> -(-x*exp(-x^2))

ans =

    0.4289
```

The minimum and maximum values of $f(x) = x \cdot e^{-x^2}$ are `-0.4289 and 0.4289` and are attained at `-0.7071` and `0.7071` respectively.

Next, consider the problem of linear optimization, which is frequently encountered in Operations Research and Mathematical Economy. The objective is to find n real numbers that minimize the linear expression

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$$

subject to the constraints:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \le b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \le b_2$$
$$\ldots$$
$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \le b_m$$

or in matrix form,

minimize $c^{\mathrm{T}}x$ such that $Ax \le b$.

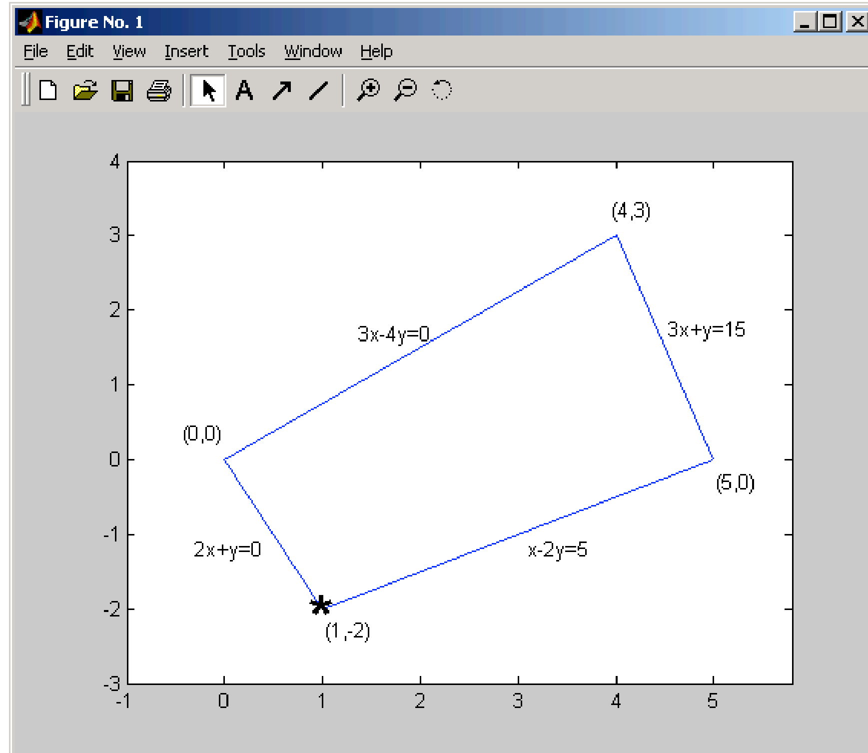
The problem can be solved via the function **lp** . As an example consider the following 2-D linear optimization problem:

Minimize $2x_1 + 3x_2$, so that the constraints
$-3x_1 + 4x_2 \le 0$, $3x_1 + x_2 \le 15$, $x_1 - 2x_2 \le 5$, $-2x_1 - x_2 \le 0$ hold.

To solve it we type:

```
>> c=[2 3];
>> A=[-3 4; 3 1; 1 -2; -2 -1];
>> b=[0 15 5 0]';
>> lp(c,A,b)
>>ans =

     1.0000
    -2.0000
```

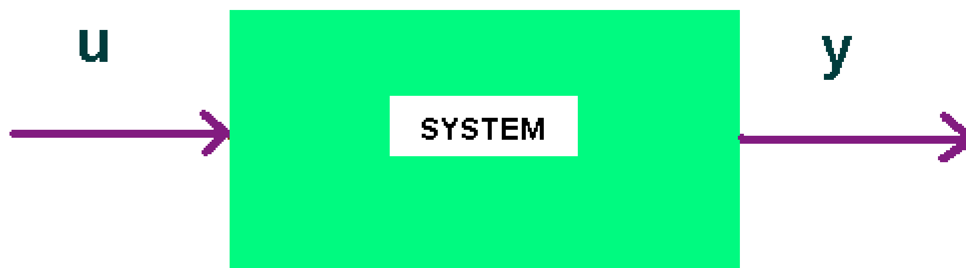
Next we give a geometrical interpretation of this solution. It can be shown that the pair (x1,x2) that solves the problem, is one of the vertices of the quadrilateral in the following figure:




The edges of the trapezoid correspond to the optimization constraints and the points in its interior satisfy all of them, hence the solution must be attained in the trapezoid or on its boundary. Due to a theorem in Linear Optimization, the solution is attained at one of the trapezoids' vertices, in this case at the point (1,-2).

***Using Simulink***

Simulink is a simulation tools library for dynamical systems. Any system in nature can roughly be thought of as a "black box" receiving an input vector **u** and eliciting a unique output vector **y**. In the case that both **u** and **y** vary with time we are talking about dynamic systems.



Associated with a system is the so-called state vector which loosely speaking contains the required information at time $t_0$ that together with knowledge of the input for time greater than $t_0$, uniquely determines the output for $t \geq t_0$. A general continuous dynamical system can be modeled by using the following set of ordinary differential and algebraic equations:

$$\dot{x} = f(t, x, u)$$
$$y = g(t, x, u)$$

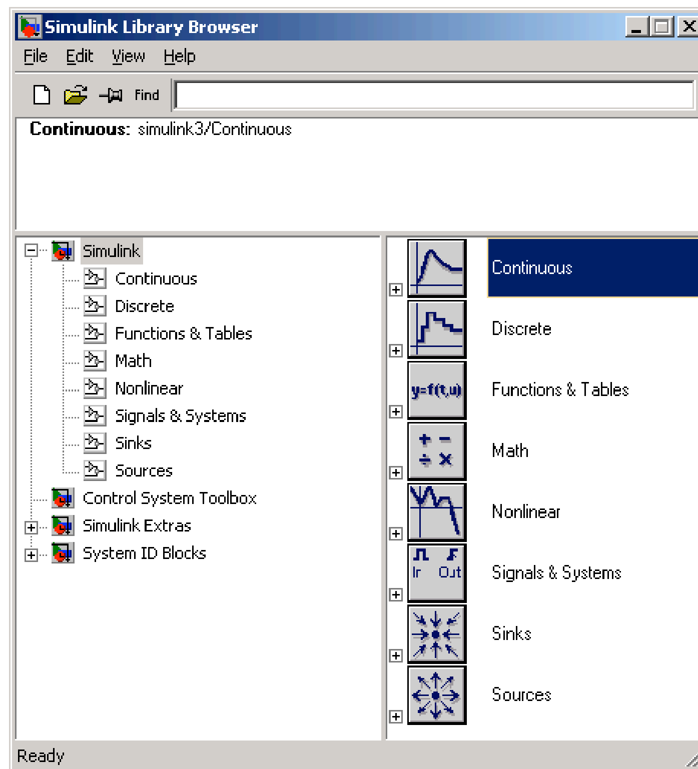
for $t \geq t_0$ and $x(t_0) = x_0$, where $f, g$ are general (possibly non-linear functions). In the following we will consider only linear systems of the form:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \qquad (1)$$

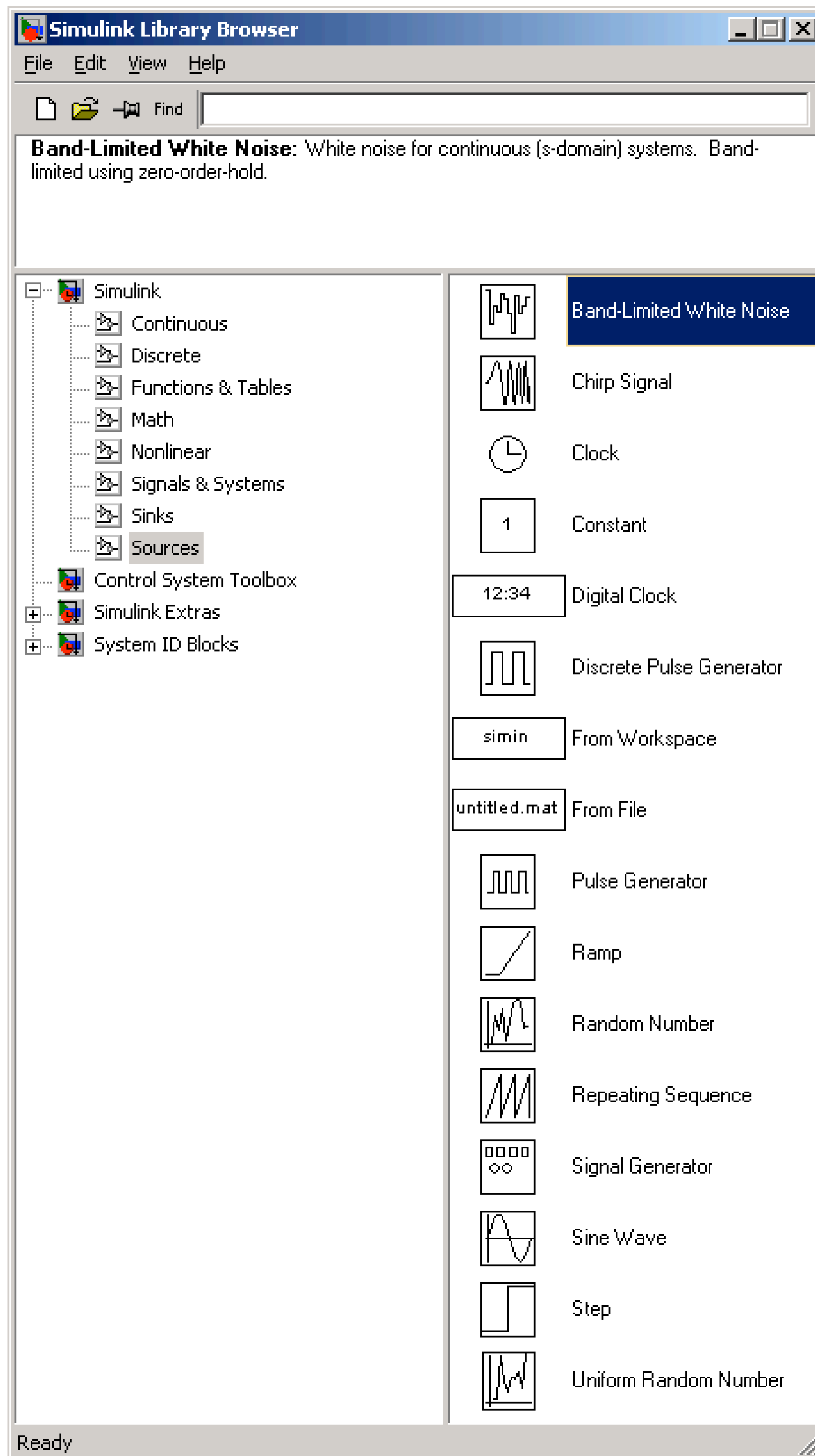
where $A, B, C, D$ are matrices and $x, u, y$ the state, input and output vectors respectively. *Of course the dimensions of $A, B, C, D$ are such that the matrix manipulations on the right hand side of (1) are well defined.*

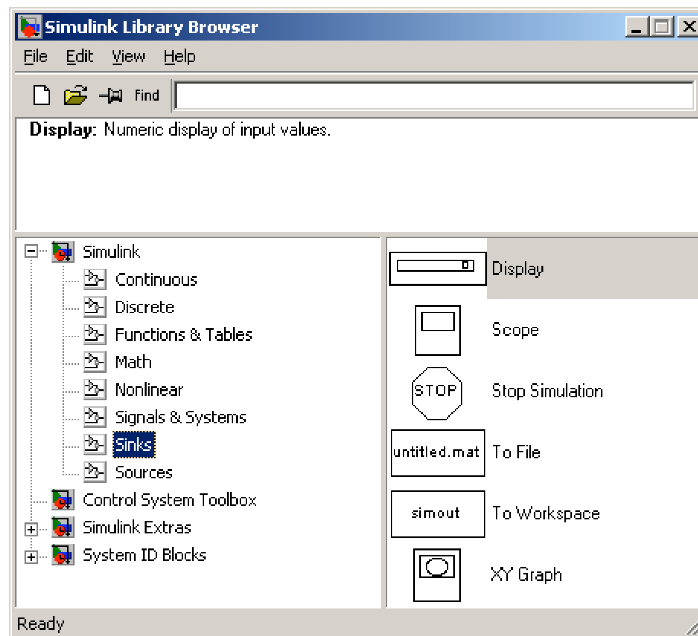
***Simulink Library Browser***

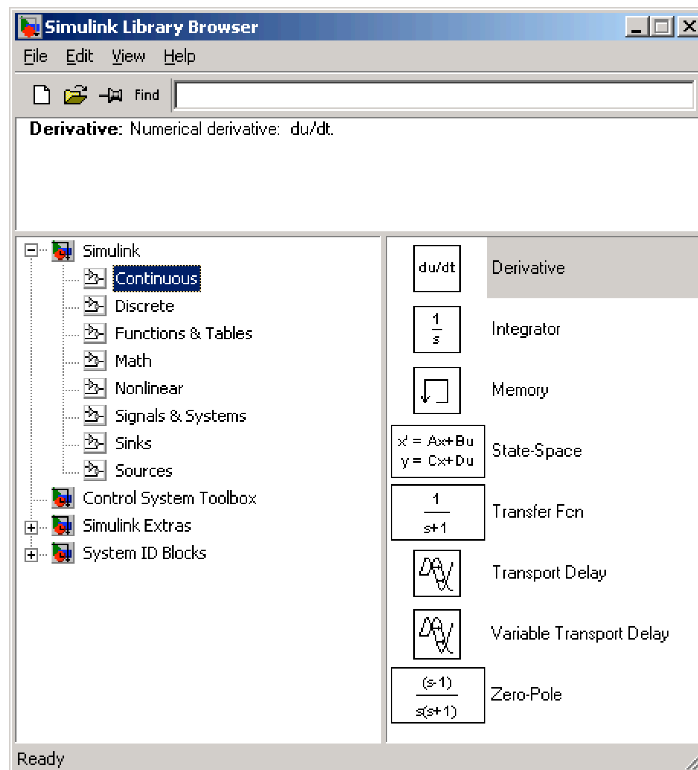
Simulink can be launched by double-clicking on the Simulink icon in the Launch pad window of the default Matlab desktop.




The library's functionalities are divided into eight groups (click on any of the category icons for both the Unix or Windows versions). For example the categories **Sources** and **Sinks** contain various kinds of inputs and ways to handle or display the output respectively. Also, the group **Continuous** that will be used later deals with dynamical systems.

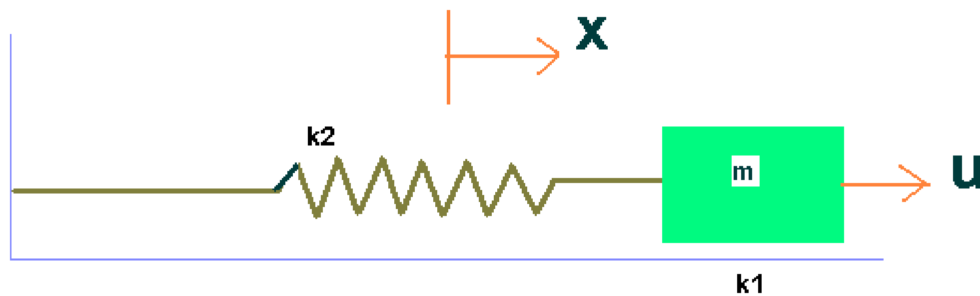

Simulink Library Browser
File Edit View Help
Find
Band-Limited White Noise: White noise for continuous (s-domain) systems. Band-limited using zero-order-hold.
Simulink
Continuous
Discrete
Functions & Tables
Math
Nonlinear
Signals & Systems
Sinks
Sources
Control System Toolbox
Simulink Extras
System ID Blocks
Band-Limited White Noise
Chirp Signal
Clock
1
Constant
12:34
Digital Clock
Discrete Pulse Generator
simin
From Workspace
untitled.mat
From File
Pulse Generator
Ramp
Random Number
Repeating Sequence
Signal Generator
Sine Wave
Step
Uniform Random Number
Ready


Simulink Library Browser
File Edit View Help
Find
Display: Numeric display of input values.
Simulink
Continuous
Discrete
Functions & Tables
Math
Nonlinear
Signals & Systems
Sinks
Sources
Control System Toolbox
Simulink Extras
System ID Blocks
Display
Scope
STOP
Stop Simulation
untitled.mat
To File
simout
To Workspace
XY Graph
Ready


Simulink Library Browser
File Edit View Help
Find
Derivative: Numerical derivative: du/dt.
Simulink
Continuous
Discrete
Functions & Tables
Math
Nonlinear
Signals & Systems
Sinks
Sources
Control System Toolbox
Simulink Extras
System ID Blocks
du/dt
Derivative
Integrator
Memory
x' = Ax+Bu
y = Cx+Du
State-Space
Transfer Fcn
Transport Delay
Variable Transport Delay
Zero-Pole
Ready

***Construction/ Simulation of Dynamical Systems***

In the following, we consider a simple physical example to illustrate the usage of Simulink. One of the simplest systems introduced in mechanics classes is the vibrating spring,




which can be modeled by the ordinary differential equation

$$my'' + k_1 y' + k_2 y = u \qquad (2)$$

Here $m$ is the mass of the body supported by the spring; $k_1$ and $k_2$ are the viscous and spring friction coefficients respectively, and $u$ is the force applied to the body. The unknown function $y$ is the distance of the body from the equilibrium position. Our first observation is that the differential equation describing the motion of the body is of second order (in other words the highest differentiation order of the equation is 2). To reduce it to a system of differential equations of first order (so that we can use Simulink) we make the following substitution:

$$x_1 = y$$
$$x_2 = y'$$

Then (1) becomes:

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = -\frac{k_2}{m}x_1 - \frac{k_1}{m}x_2 + \frac{1}{m}u$$

or in matrix form:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k_2}{m} & -\frac{k_1}{m} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \cdot u \qquad (3)$$

Here $x_1$ and $x_2$ are the state variables and $u$ the input to the system. The output can be selected in various ways depending on what characteristics of the system are desired to be measured; it could be $x_1$ (that is displacement), $x_2$ (velocity) or a linear combination of $x_1, x_2$ and $u$. For our purposes we simply define the output to be $x_1$. That is,

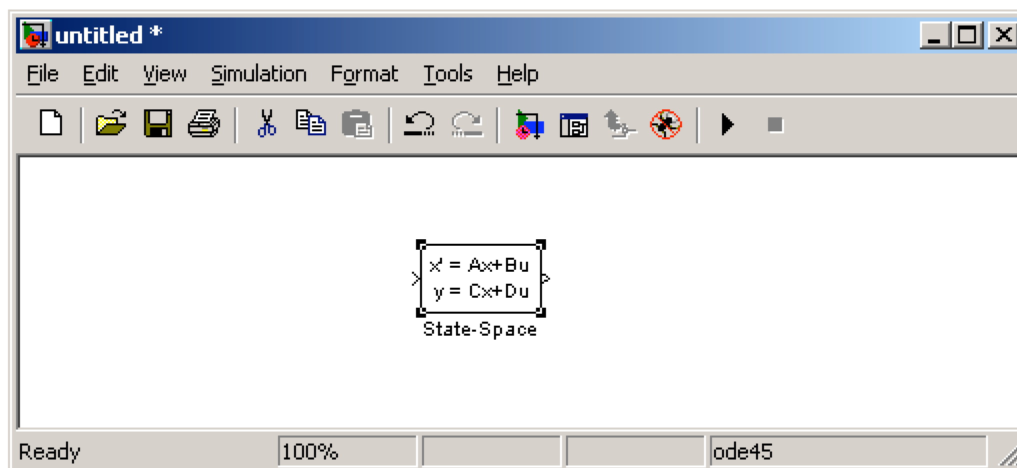
$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \cdot u \qquad (4)$$

in matrix form.

Equations (3) and (4) constitute the representation of the system in the form (1), with

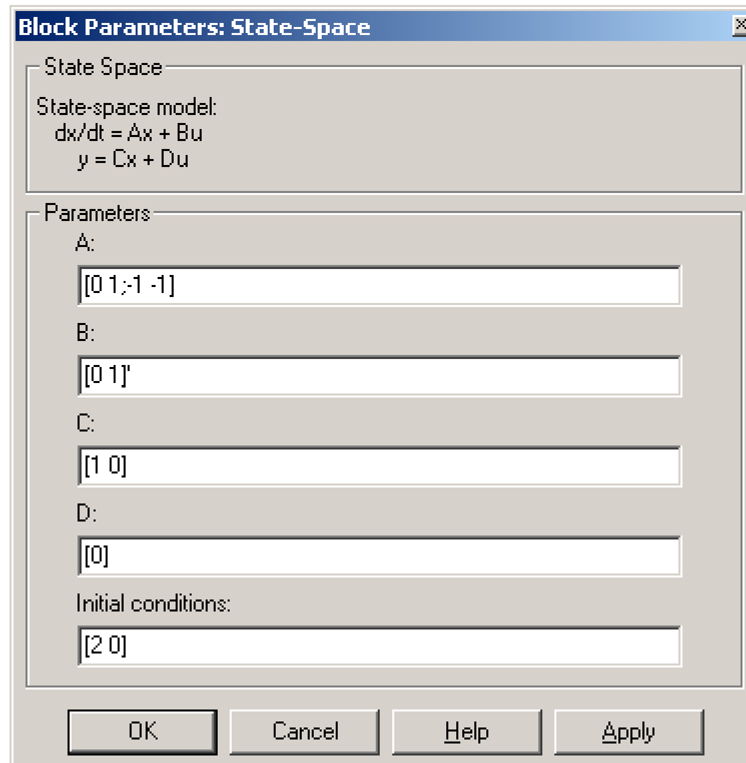
$$A = \begin{bmatrix} 0 & 1 \\ -\frac{k_2}{m} & -\frac{k_1}{m} \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 \end{bmatrix}.$$

Furthermore, we take $m = k_1 = k_2 = 1$, that is $A = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Our initial conditions are $x_1(0) = 2$, $x_2(0) = 0$, that is at time $t = 0$ the body is located at a distance of 2 units from the equilibrium position and its velocity is 0. The next step is to build the system using Simulink. Clicking on the **New model** button on the upper left corner of the Simulink library browser a window pops out. Next double click on the **Continuous** button, select the **State-Space** icon and drag it into the new model window.

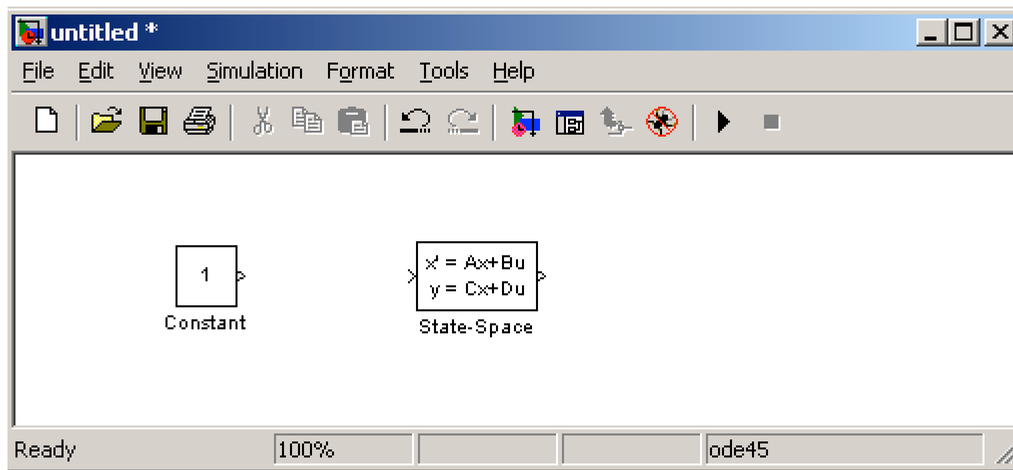



Double clicking on the dragged **State-Space** icon the block parameters window appears in which we specify the matrices $A, B, C$ and $D$ as well as the initial condition vector. Note that the matrices are entered in the one row format described in section 4, but for

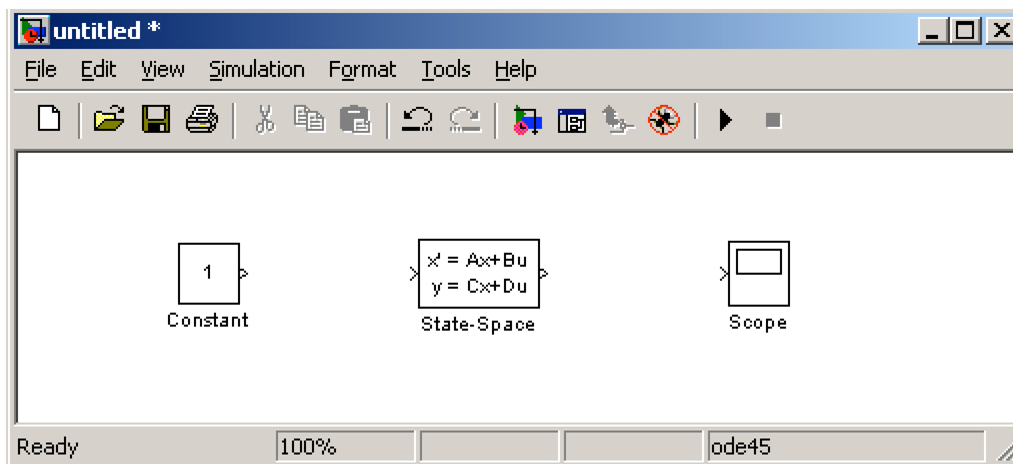
large matrices it is more convenient to define $A, B, C, D$ in the command line (possibly with other names) and simply enter their names in the block parameters window.



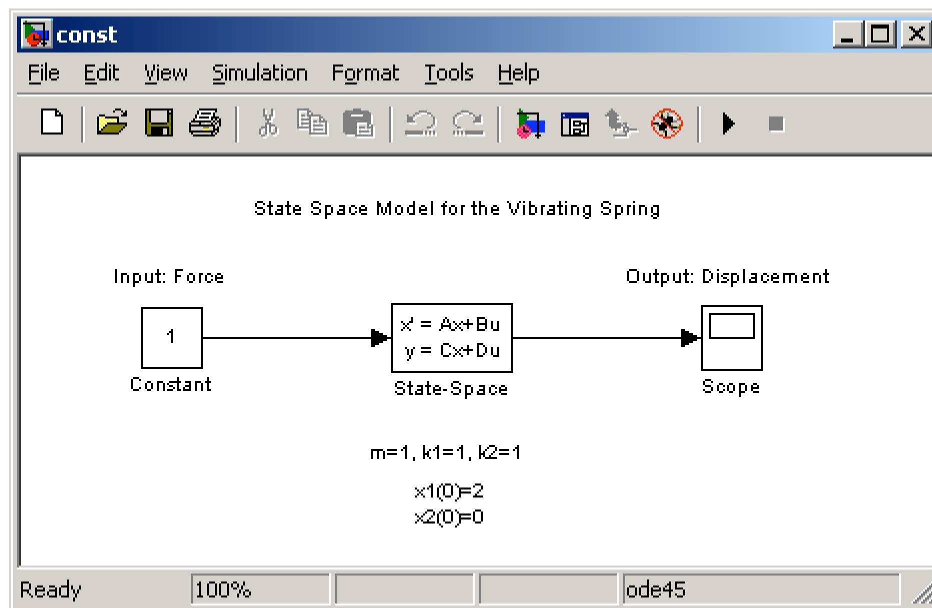

In the following we double click the **Sources** icon to select an input for the system. Lets choose the input **Constant** and drag it into the new model window. Again, by double clicking on the dragged **Constant** icon we specify the value of the constant input. For our example we take $u = 1$.



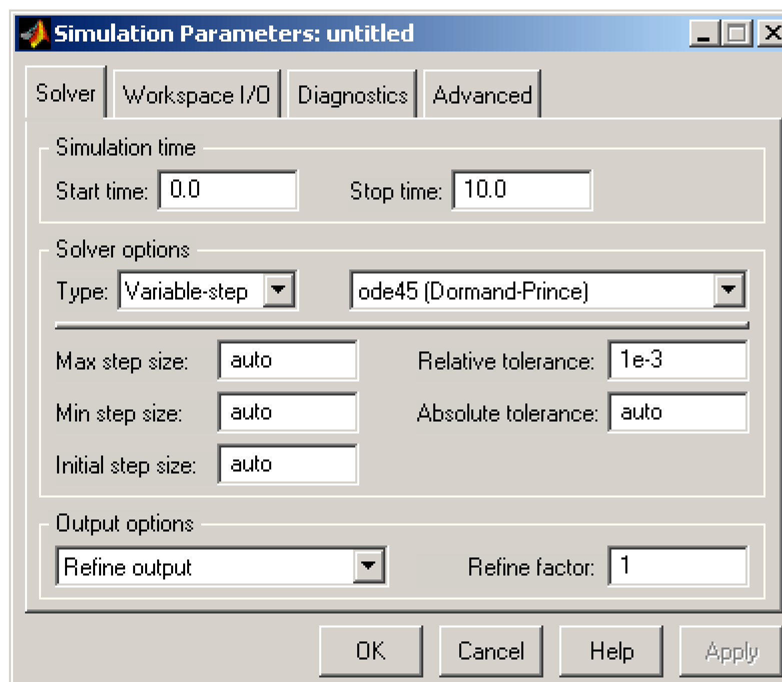

The output is selected by clicking on the **Sinks** button. Choose for example the **Scope** icon and drag it into the model window (Scope provides a graph of the system's output).



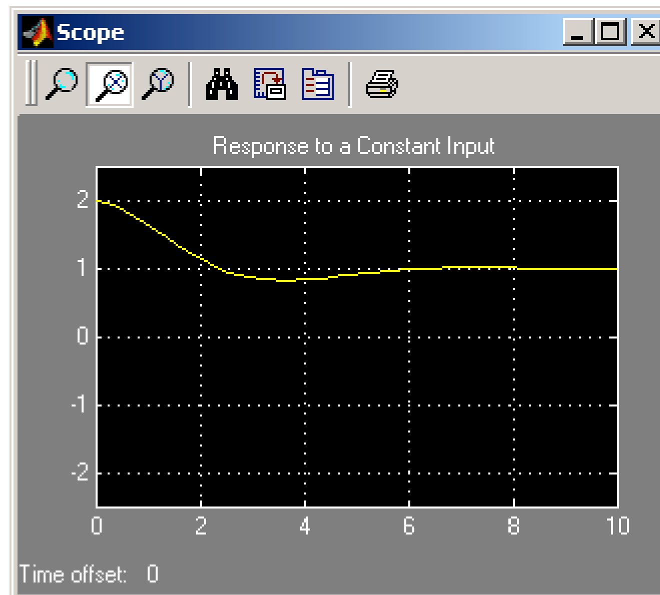

Next, connect the system blocks with arrows. For example, to connect the constant and state-space blocks, click on the right arrow of the constant block and move the cursor to the left arrow of the state-space block while holding the left mouse key down. We can also add text on the model window by double clicking at any point of it and inserting the desired information.



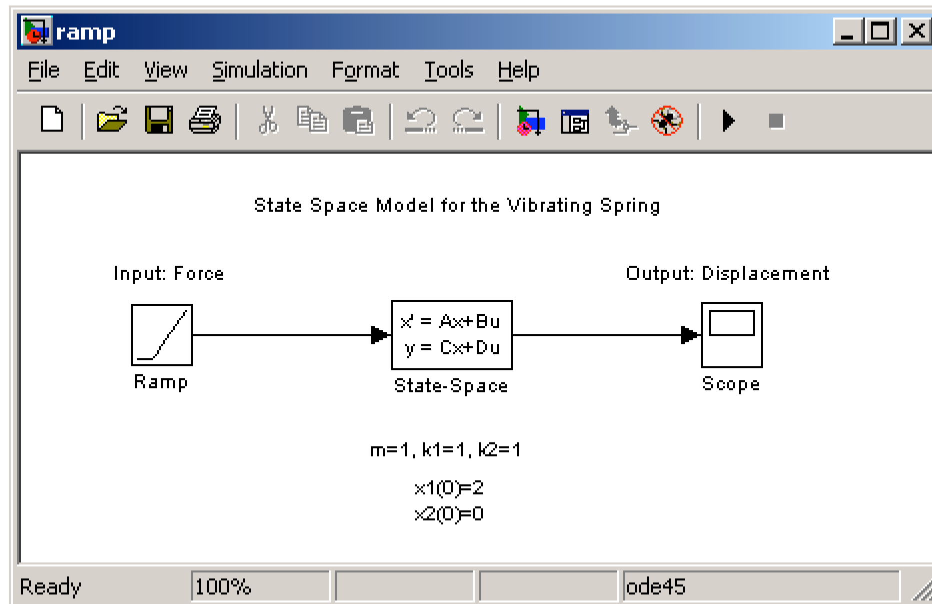

Finally click on the **Simulation** pull down menu on the toolbar and select the **Simulation Parameters** option to specify the simulation parameters (the simulation initial and final time and the ODE solver to be used for example). In this example we choose $t_{final} = 10$.

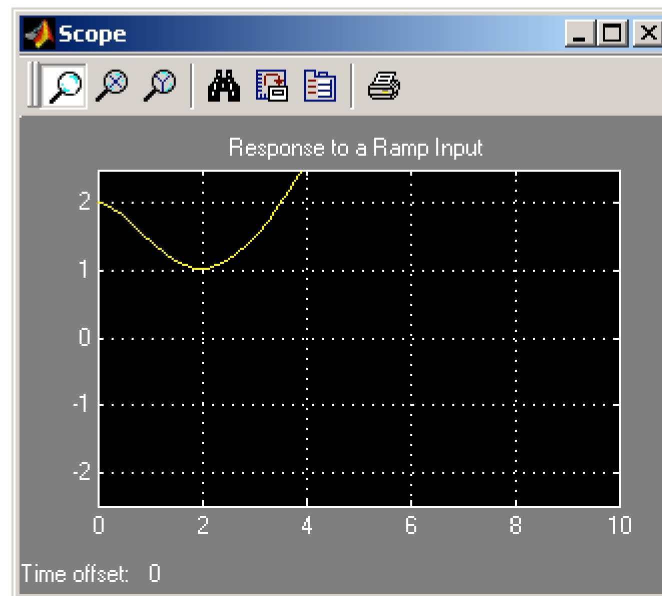


Now that the system has been built up, we are ready to run the simulation, in other words numerically solve the system of ODEs to obtain the output $y$. Simply, press **Start** on the **Simulation** toolbar menu end then double click on the **Scope** icon to obtain a plot of the output.

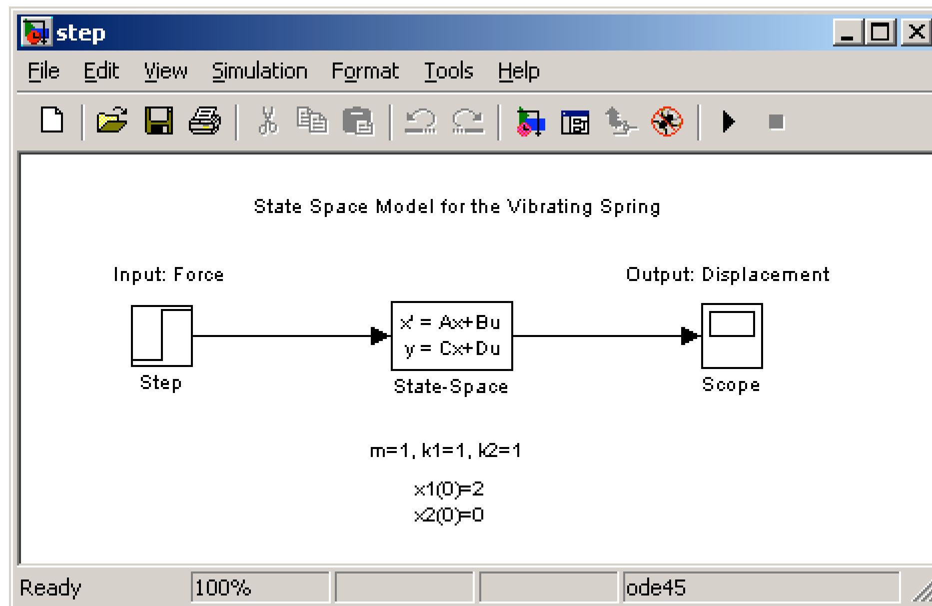


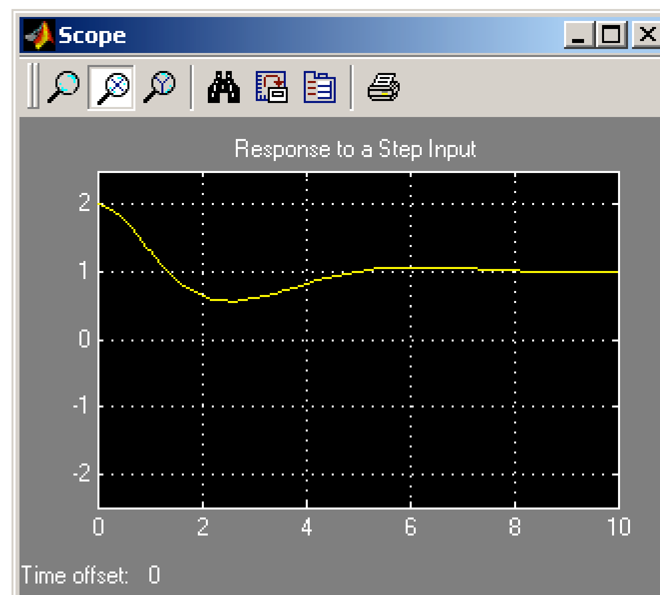
We observe that after time t approximately equal to 9, the displacement remains constant. Next we change the system's input to a ramp by following the previous procedure. We specify the ramp input parameters by double-clicking on the **Ramp** icon and choosing slope=1, start time=0 and initial output=0. In the following figures we present the modified system and its response.



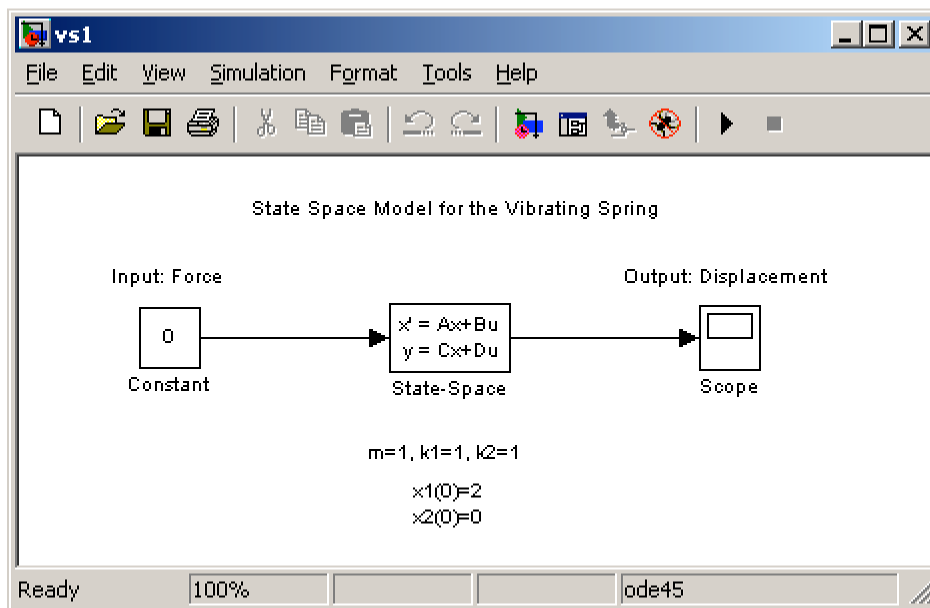



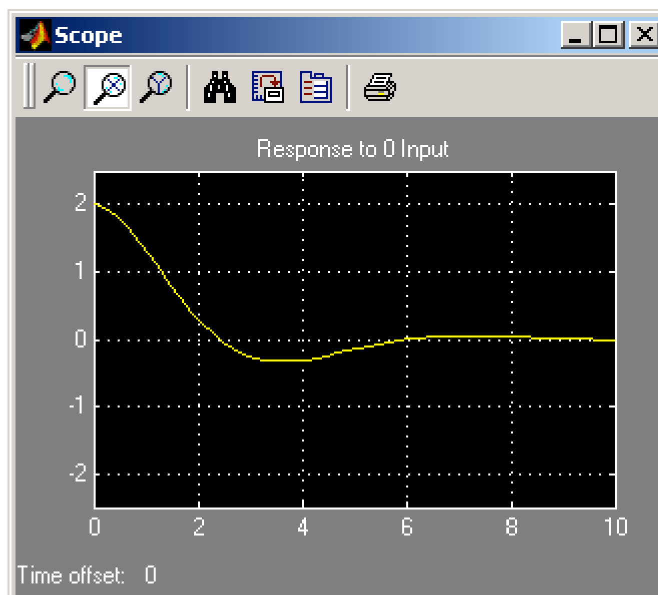

As expected, since the force acting on the body is increasing in magnitude and doesn't change direction, the distance from the rest point increases. Also, we consider the case of a step input, that is a force spontaneously changing its magnitude at some time $t_0$. Again the step function parameters, are specified by double-clicking on the **Step** icon and choosing step time=1, initial value=0 and final value=1



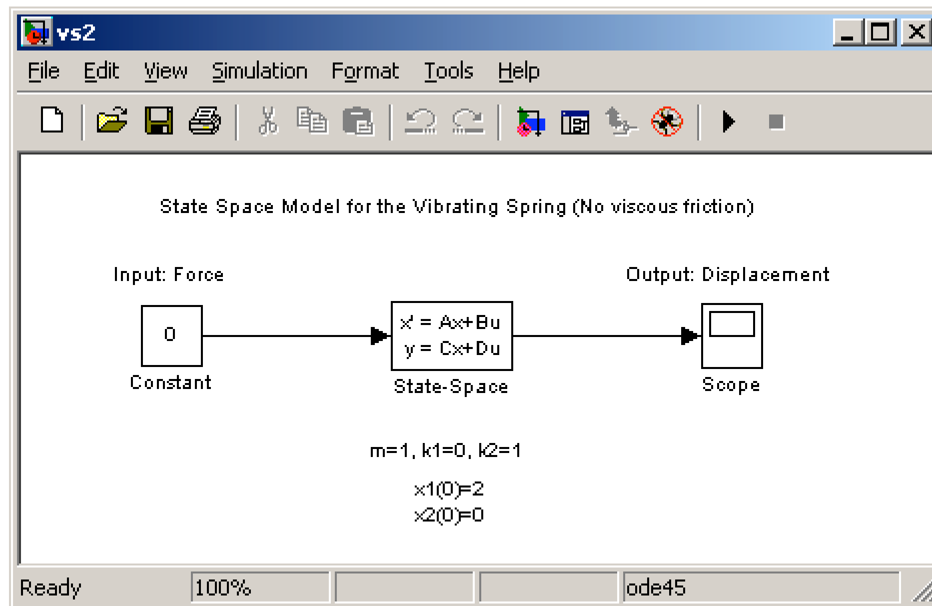



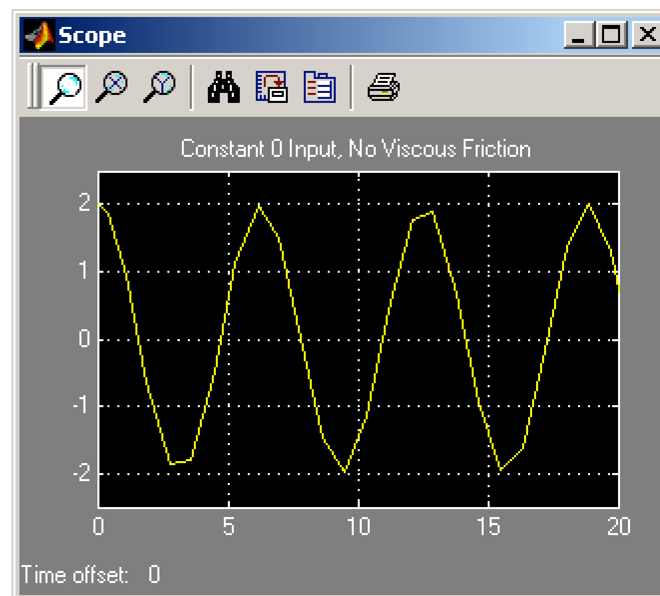

In the sequel, lets consider the case of zero force acting on the body, that is we have a constant input equal to zero. We expect that the body will eventually return to the rest position $x_1 = 0$ and indeed this is what our model predicts.




In all previous cases except the ramp input's case, the system's output eventually approaches a constant value. Let's examine the case where viscous friction is negligible, that is $k_1 = 0$ and again the input force is the constant 0 (double click on **State-Space** icon and adjust A to [0 1; -1 0]). We expect that the body will be moving internally, since no force exists to counterbalance the spring's force acting on it. Indeed, the prediction of our model is consistent with the physical intuition. As we can see in the latter of next two figures the output of the system (displacement of the body) oscillates between the extreme values –2 and 2.




In our last example we try to correct this behavior and actually make the system's output approach the value 0 (that is to make the body return to the equilibrium position) by introducing a so-called feedback control law. More precisely we define a new input to the system which is not user supported as in the previous examples, but depends on the system's output. In other words, the system's output simultaneously defines its input. Indeed, we define:

$u = K \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, where $K = [0 - 1]$. The selection of the feedback gain matrix $K$ is of course not arbitrary but we will not comment on its derivation. In fact $K$ can be selected

in many different ways and it is the task of the control engineer to determine the best one, depending on design requirements and limitations.

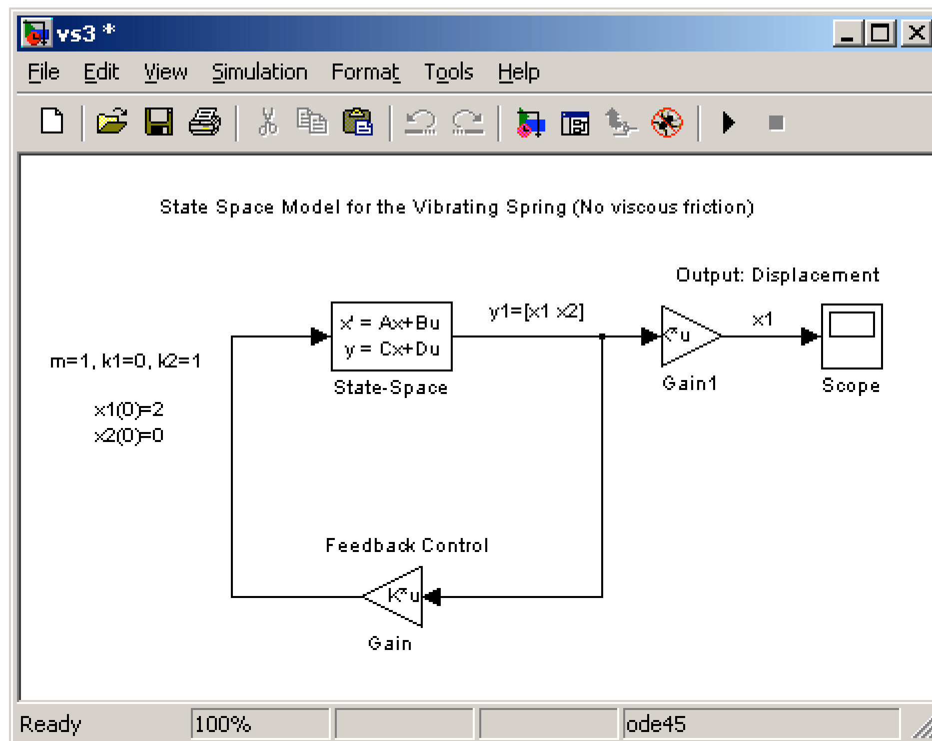
Then our system becomes:

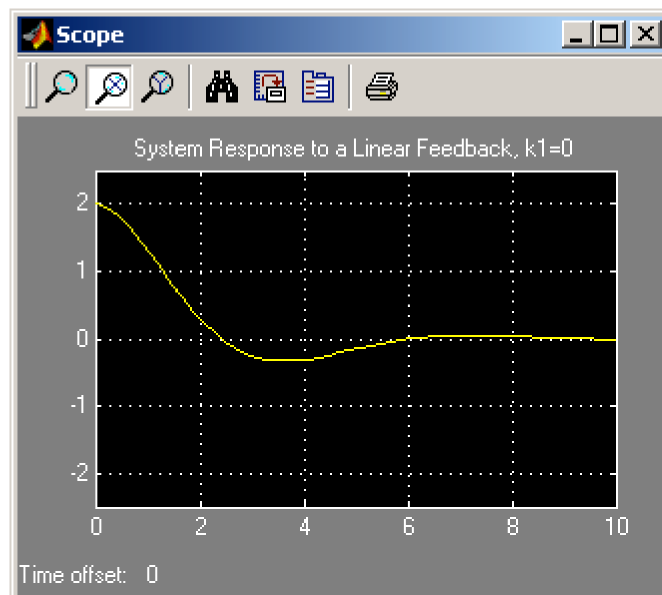
$$\dot{x} = (A + B \cdot K) \cdot x$$
$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot x + \begin{bmatrix} 0 \end{bmatrix} \cdot u$$

but using Simulink we actually develop  the equivalent formulation

$$\dot{x} = (A + B \cdot K) \cdot x$$
$$y_1 = x$$
$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot y_1$$

In the next two figures the feedback controlled system is presented along with its output. To construct this model drag the **Gain** blocks from the **Math** library (and again double click on them to specify the gain matrices). Also, in order to rotate a **Gain** block simply right-click on it, choose the **Format** option from the pull-down menu that appears and then select the options **Rotate block** or **Flip block**. Finally, note that the body returns to its equilibrium position as it was desired.




Scope
System Response to a Linear Feedback, k1=0
2
1
0
-1
-2
0
2
4
6
8
10
Time offset: 0