Basics of MATLAB

1 First Steps in MATLAB

1.1 Starting MATLAB

MATLAB is a software package that lets you do mathematics and computation, analyse data, develop algorithms, do simulation and modelling, and produce graphical displays and graphical user interfaces.

To run MATLAB on a PC double-click on the MATLAB icon. To run MATLAB on a UNIX system, type `matlab` at the prompt.

You get MATLAB to do things for you by typing in commands. MATLAB prompts you with two greater-than signs (`>>`) when it is ready to accept a command from you.

To end a MATLAB session type `quit` or `exit` at the MATLAB prompt.

You can type `help` at the MATLAB prompt, or pull down the Help menu on a PC.

When starting MATLAB you should see a message:

```
  To get started, type one of these commands: helpwin,
  helpdesk, or demo
>>
```

The various forms of help available are

`helpwin`	Opens a MATLAB help GUI
`helpdesk`	Opens a hypertext help browser
`demo`	Starts the MATLAB demonstration

The complete documentation for MATLAB can be accessed from the hypertext helpdesk. For example, clicking the link Full Documentation

Set → Getting Started with MATLAB will download a portable document format (PDF) version of the *Getting Started with MATLAB* manual.

You can learn how to use any MATLAB command by typing `help` followed by the name of the command, for example, `help sin`.

You can also use the `lookfor` command, which searches the help entries for all MATLAB commands for a particular word. For example, if you want to know which MATLAB functions to use for spectral analysis, you could type `lookfor spectrum`. MATLAB responds with the names of the commands that have the searched word in the first line of the help entry. You can search the entire help entry for all MATLAB commands by typing `lookfor -all` *`keyword`*.

1.2 First Steps

To get MATLAB to work out $1 + 1$, type the following at the prompt:

```
1+1
```

MATLAB responds with

```
ans  =
     2
```

The answer to the typed command is given the name `ans`. In fact `ans` is now a variable that you can use again. For example you can type

```
ans*ans
```

to check that $2 \times 2 = 4$:

```
ans*ans
ans  =
     4
```

MATLAB has updated the value of `ans` to be 4.

The spacing of operators in formulas does not matter. The following formulas both give the same answer:

```
1+3 * 2-1 / 2*4
1 + 3 * 2 - 1 / 2 * 4
```

The order of operations is made clearer to readers of your MATLAB code if you type carefully:

```
1 + 3*2 - (1/2)*4
```

1.3 Matrices

The basic object that MATLAB deals with is a matrix. A matrix is an array of numbers. For example the following are matrices:

$$\begin{pmatrix} 12 & 3 & 9 \\ -1200 & 0 & 1\text{e}6 \\ 0.1 & \text{pi} & 1/3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}, \begin{pmatrix} i \\ -i \\ i \\ -i \end{pmatrix}, 42.$$

The size of a matrix is the number of rows by the number of columns. The first matrix is a 3×3 matrix. The (2,3)-element is one million—1e6 stands for 1×10^6—and the (3,2)-element is pi $= \pi = 3.14159 \ldots$. The second matrix is a row-vector, the third matrix is a column-vector containing the number i, which is a pre-defined MATLAB variable equal to the square root of -1. The last matrix is a 1×1 matrix, also called a scalar.

1.4 Variables

Variables in MATLAB are named objects that are assigned using the equals sign `=` . They are limited to 31 characters and can contain upper and lowercase letters, any number of '_' characters, and numerals. They may not start with a numeral. MATLAB is case sensitive: `A` and `a` are different variables. The following are valid MATLAB variable assignments:

```
a = 1
speed = 1500
BeamFormerOutput_Type1 = v*Q*v'
name = 'John Smith'
```

These are invalid assignments:

```
2for1 = 'yes'
first one = 1
```

To assign a variable without getting an echo from MATLAB end the assignment with a semi-colon `;`. Try typing the following:

```
a = 2
b = 3;
c = a+b;
d = c/2;
d
who
whos
clear
who
```

1.5 The Colon Operator

To generate a vector of equally-spaced elements MATLAB provides the colon operator. Try the following commands:

```
1:5
0:2:10
0:.1:2*pi
```

The syntax *x*:*y* means roughly "generate the ordered set of numbers from *x* to *y* with increment 1 between them." The syntax *x*:*d*:*y* means roughly "generate the ordered set of numbers from *x* to *y* with increment *d* between them."

1.6 Linspace

To generate a vector of evenly spaced points between two end points, you can use the function `linspace(`*`start`*`,`*`stop`*`,`*`npoints`*`)`:

```
>> x = linspace(0,1,10)
x  =
  Columns 1 through 7
       0  0.1111  0.2222  0.3333  0.4444  0.5556  0.6667
  Columns 8 through 10
  0.7778  0.8889  1.0000
```

generates 10 evenly spaced points from 0 to 1. Typing `linspace(`*`start`*`,` *`stop`*`)` will generate a vector of 100 points.

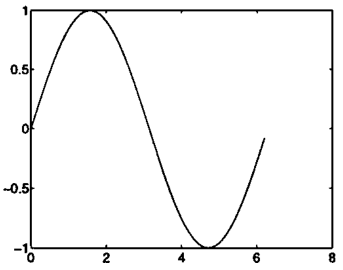
1.7 Plotting Vectors

Whereas other computer languages, such as FORTRAN, work on numbers one at a time, an advantage of MATLAB is that it handles the matrix as a single unit. Let us consider an example that shows why this is useful. Imagine you want to plot the function $y = \sin x$ for x between 0 and 2π. A FORTRAN code to do this might look like this:

```
DIMENSION X(100),Y(100)
PI = 4*ATAN(1)
DO 100 I = 1,100
    X(I) = 2*PI*I/100
    Y(I) = SIN(X(I))
100 CONTINUE
PLOT(X,Y)
```

Here we assume that we have access to a FORTRAN plotting package in which `PLOT(X,Y)` makes sense. In MATLAB we can get our plot by typing:

```
x = 0:.1:2*pi;
y = sin(x);
plot(x,y)
```



The first line uses the colon operator to generate a vector x of numbers running between 0 and 2π with increment 0.1. The second line calculates the sine of this array of numbers, and calls the result y. The third line produces a plot of y against x. Go ahead and produce the plot. You should get a separate window displaying this plot. We have done in three lines of MATLAB what it took us seven lines to do using the FORTRAN program above.

2 Typing into MATLAB

2.1 Command Line Editing

If you make a mistake when entering a MATLAB command, you do not have to type the whole line again. The arrow keys can be used to save much typing:

↑	ctrl-p	Recall previous line
↓	ctrl-n	Recall next line
←	ctrl-b	Move back one character
→	ctrl-f	Move forward one character
ctrl-→	ctrl-r	Move right one word
ctrl-←	ctrl-l	Move left one word
home	ctrl-a	Move to beginning of line
end	ctrl-e	Move to end of line
esc	ctrl-u	Clear line
del	ctrl-d	Delete character at cursor
backspace	ctrl-h	Delete character before cursor
	ctrl-k	Delete (kill) to end of line

If you finish editing in the middle of a line, you do not have to put the cursor at the end of the line before pressing the return key; you can press return when the cursor is anywhere on the command line.

2.2 Smart Recall

Repeated use of the ↑ key recalls earlier commands. If you type the first few characters of a previous command and then press the ↑ key

MATLAB will recall the last command that began with those characters. Subsequent use of ↑ will recall earlier commands that began with those characters.

2.3 Long Lines

If you want to type a MATLAB command that is too long to fit on one line, you can continue on to the next by ending with a space followed by three full stops. For example, to type an expression with long variable names:

```
Final_Answer = BigMatrix(row_indices,column_indices) + ...
               Another_vector*SomethingElse;
```

Or to define a long text string:

```
Mission = ['DSTO''s objective is to give advice that' ...
'is professional, impartial and informed on the' ...
'application of science and technology that is best' ...
'suited to Australia''s defence and security needs.'];
```

2.4 Copying and Pasting

Your windowing system's copy and paste facility can be used to enter text into the MATLAB command line. For example all of MATLAB's built-in commands have some helpful text that can by accessed by typing `help` followed by the name of the command. Try typing `help contour` into MATLAB and you will see a description of how to create a contour plot. At the end of the help message is an example. You can use the mouse to select the example text and paste it into the command line. Try it now and you should see a contour plot appear in the figure window.

3 Matrices

3.1 Typing Matrices

To type a matrix into MATLAB you must

- begin with a square bracket `[`
- separate elements in a row with commas or spaces
- use a semicolon `;` to separate rows
- end the matrix with another square bracket `]`.

For example type:

```
a = [1 2 3;4 5 6;7 8 9]
```

MATLAB responds with

```
a  =
     1     2     3
     4     5     6
     7     8     9
```

3.2 Concatenating Matrices

Matrices can be made up of submatrices: Try this:

```
>> b = [a 10*a;-a [1 0 0;0 1 0;0 0 1]]
b  =
     1     2     3    10    20    30
     4     5     6    40    50    60
     7     8     9    70    80    90
    -1    -2    -3     1     0     0
    -4    -5    -6     0     1     0
    -7    -8    -9     0     0     1
```

The `repmat` function can be used to replicate a matrix:

```
>> a = [1 2; 3 4]
a  =
     1     2
     3     4
>> repmat(a,2,3)
ans  =
     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4
```

3.3 Useful Matrix Generators

MATLAB provides four easy ways to generate certain simple matrices. These are

`zeros`	a matrix filled with zeros
`ones`	a matrix filled with ones
`rand`	a matrix with uniformly distributed random elements
`randn`	a matrix with normally distributed random elements
`eye`	identity matrix

To tell MATLAB how big these matrices should be you give the functions the number of rows and columns. For example:

```
>> a = zeros(2,3)
a  =
     0     0     0
     0     0     0

>> b = ones(2,2)/2
b  =
    0.5000    0.5000
    0.5000    0.5000

>> u = rand(1,5)
u  =
    0.9218    0.7382    0.1763    0.4057    0.9355

>> n = randn(5,5)
n  =
   -0.4326    1.1909   -0.1867    0.1139    0.2944
   -1.6656    1.1892    0.7258    1.0668   -1.3362
    0.1253   -0.0376   -0.5883    0.0593    0.7143
    0.2877    0.3273    2.1832   -0.0956    1.6236
   -1.1465    0.1746   -0.1364   -0.8323   -0.6918

>> eye(3)
ans  =
     1     0     0
     0     1     0
     0     0     1
```

3.4 Subscripting

Individual elements in a matrix are denoted by a row index and a column index. To pick out the third element of the vector `u` type:

```
>> u(3)
ans  =
    0.1763
```

You can use the vector `[1 2 3]` as an index to `u`. To pick the first three elements of `u` type

```
>> u([1 2 3])
ans  =
    0.9218    0.7382    0.1763
```

Remembering what the colon operator does, you can abbreviate this to

```
>> u(1:3)
ans  =
   0.9218    0.7382    0.1763
```

You can also use a variable as a subscript:

```
>> i = 1:3;
>> u(i)
ans  =
   0.9218    0.7382    0.1763
```

Two dimensional matrices are indexed the same way, only you have to provide two indices:

```
>> a = [1 2 3;4 5 6;7 8 9]
a  =
     1     2     3
     4     5     6
     7     8     9
>> a(3,2)
ans  =
     8
>> a(2:3,3)
ans  =
     6
     9
>> a(2,:)
ans  =
     4     5     6
>> a(:,3)
ans  =
     3
     6
     9
```

The last two examples use the colon symbol as an index, which MATLAB interprets as the entire row or column.

If a matrix is addressed using a single index, MATLAB counts the index down successive columns:

```
>> a(4)
ans  =
     2
>> a(8)
ans  =
     6
```

Exercise 1 *Do you understand the following result? (Answer on page 183.)*

```
>> [a a(a)]
ans  =
     1     2     3     1     4     7
     4     5     6     2     5     8
     7     8     9     3     6     9
```

The colon symbol can be used as a single index to a matrix. Continuing the previous example, if you type

```
a(:)
```

MATLAB interprets this as the columns of the a-matrix successively strung out in a single long column:

```
>> a(:)
ans  =
     1
     4
     7
     2
     5
     8
     3
     6
     9
```

3.5 End as a subscript

To access the last element of a matrix along a given dimension, use `end` as a subscript (MATLAB version 5 or later). This allows you to go to the final element without knowing in advance how big the matrix is. For example:

```
>> q = 4:10
q  =
     4     5     6     7     8     9    10
>> q(end)
ans  =
    10
>> q(end-4:end)
ans  =
     6     7     8     9    10
>> q(end-2:end)
ans  =
     8     9    10
```

This technique works for two-dimensional matrices as well:

```
>> q = [spiral(3) [10;20;30]]
q  =
     7     8     9    10
     6     1     2    20
     5     4     3    30
>> q(end,end)
ans  =
    30

>> q(2,end-1:end)
ans  =
     2    20

>> q(end-2:end,end-1:end)
ans  =
     9    10
     2    20
     3    30

>> q(end-1,:)
ans  =
     6     1     2    20
```

3.6 Deleting Rows or Columns

To get rid of a row or column set it equal to the empty matrix `[]`.

```
>> a = [1 2 3;4 5 6;7 8 9]
a  =
     1     2     3
     4     5     6
     7     8     9
>> a(:,2) = []
a  =
     1     3
     4     6
     7     9
```

3.7 Matrix Arithmetic

Matrices can be added and subtracted (they must be the same size).

```
>> b = 10*a
b  =
    10    30
    40    60
    70    90
```

```
>> a + b
ans  =
    11    33
    44    66
    77    99
```

3.8 Transpose

To convert rows into columns use the transpose symbol `'`:

```
>> a'
ans  =


     1     4     7
     3     6     9

>> b = [[1 2 3]' [4 5 6]']
b  =
     1     4
     2     5
     3     6
```

Be careful when taking the transpose of complex matrices. The transpose operator takes the complex conjugate transpose. If `z` is the matrix:

$$\begin{pmatrix} 1 & 0-i \\ 0+2i & 1+i \end{pmatrix}$$

then `z'` is:

$$\begin{pmatrix} 1 & 0-2i \\ 0+i & 1-i \end{pmatrix}.$$

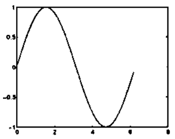
To take the transpose without conjugating the complex elements, use the `.'` operator. In this case `z.'` is:

$$\begin{pmatrix} 1 & 0+2i \\ 0-i & 1+i \end{pmatrix}.$$

4 Basic Graphics

The bread-and-butter of MATLAB graphics is the `plot` command. Earlier we produced a plot of the sine function:

```
x = 0:.1:2*pi;
y = sin(x);
plot(x,y)
```



In this case we used `plot` to plot one vector against another. The elements of the vectors were plotted in order and joined by straight line segments. There are many options for changing the appearance of a plot. For example:

```
plot(x,y,'r-.')
```

will join the points using a red dash-dotted line. Other colours you can use are: `'c'`, `'m'`, `'y'`, `'r'`, `'g'`, `'b'`, `'w'`, `'k'`, which correspond to cyan, magenta, yellow, red, green, blue, white, and black. Possible line styles are: solid `'-'`, dashed `'--'`, dotted `':'`, and dash-dotted `'-.'`. To plot the points themselves with symbols you can use: dots `'.'`, circles `'o'`, plus signs `'+'`, crosses `'x'`, or stars `'*'`, and many others (type `help plot` for a list). For example:

```
plot(x,y,'bx')
```

plots the points using blue crosses without joining them with lines, and

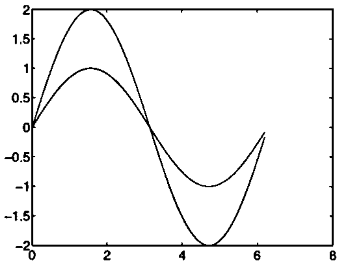
```
plot(x,y,'b:x')
```

plots the points using blue crosses and joins them with a blue dotted line. Colours, symbols and lines can be combined, for example, `'r.-'`, `'rx-'` or `'rx:'`.

4.1 Plotting Many Lines

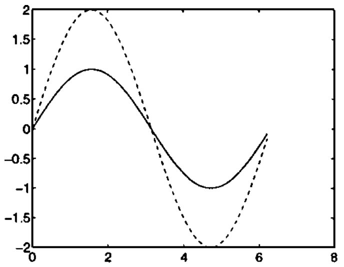
To plot more than one line you can specify more than one set of x and y vectors in the `plot` command:

```
plot(x,y,x,2*y)
```



On the screen MATLAB distinguishes the lines by drawing them in different colours. If you need to print in black and white, you can differentiate the lines by plotting one of them with a dashed line:

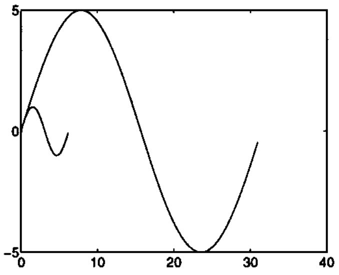
```
plot(x,y,x,2*y,'--')
```



4.2 Adding Plots

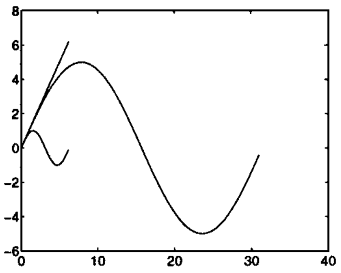
When you issue a `plot` command MATLAB clears the axes and produces a new plot. To add to an existing plot, type `hold on`. For example try this:

```
plot(x,y)
hold on
plot(5*x,5*y)
```



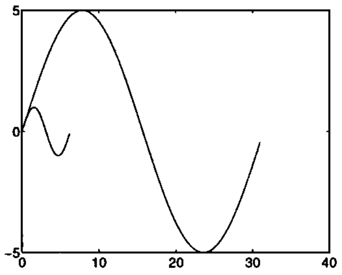
MATLAB re-scales the axes to fit the new data. The old plot appears smaller. Once you have typed `hold on`, all subsequent plots will be added to the current axes:

```
plot(x,x)
```



Companion M-Files Feature 1 *If you decide you want to remove the last thing you plotted on a plot with* `hold on` *in force, you can type:*

```
undo
```



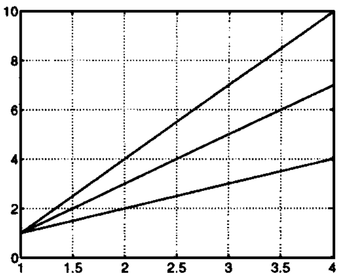
to get back to where you were before.

To switch off the `hold` behaviour, type `hold off`. Typing `hold` by itself toggles the hold state of the current plot.

4.3 Plotting Matrices

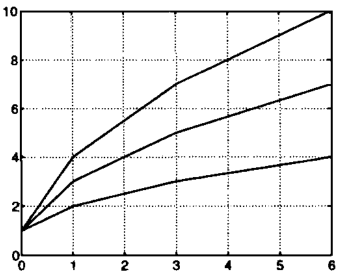
If one of the arguments to the `plot` command is a matrix, MATLAB will use the columns of the matrix to plot a set of lines, one line per column:

```
>> q = [1 1 1;2 3 4;3 5 7;4 7 10]
q =
     1     1     1
     2     3     4
     3     5     7
     4     7    10
>> plot(q)
>> grid
```



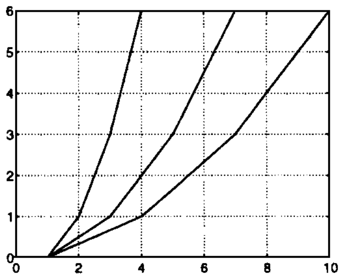
MATLAB plots the columns of the matrix `q` against the row index. You can also supply an x variable:

```
>> x = [0 1 3 6]
x =
     0     1     3     6
>> plot(x,q)
>> grid
```



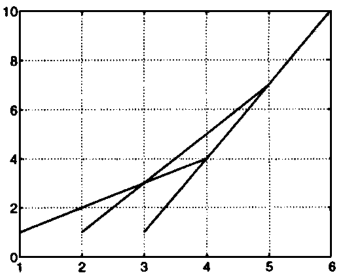
Here the x values are not uniformly spaced, but they are the same for each column of `q`. You can also plot a matrix of x values against a vector of y values (be careful: the y values are in the vector `x`):

```
plot(q,x)
grid
```



If both the x and y arguments are matrices, MATLAB will plot the successive columns on the same plot:

```
>> x = [[1 2 3 4]' [2 3 4 5]' [3 4 5 6]']
x  =
     1     2     3
     2     3     4
     3     4     5
     4     5     6
>> plot(x,q)
>> grid
```



4.4 Clearing the Figure Window

You can clear the plot window by typing `clf`, which stands for 'clear figure'. To get rid of a figure window entirely, type `close`. To get rid of all the figure windows, type `close all`. New figure windows can be created by typing `figure`.

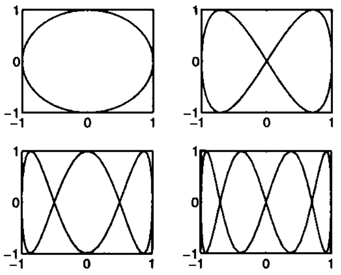
4.5 Subplots

To plot more than one set of axes in the same window, use the `subplot` command. You can type

```
subplot(m,n,p)
```

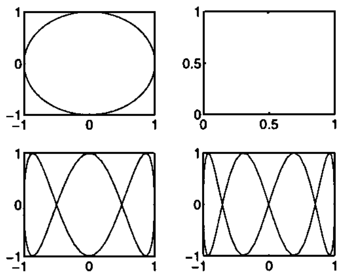
to break up the plotting window into `m` plots in the vertical direction and `n` plots in the horizontal direction, choosing the `p`th plot for drawing into. The subplots are counted as you read text: left to right along the top row, then left to right along the second row, and so on. Here is an example (do not forget to use the ↑ key to save typing):

```
t = 0:.1:2*pi;
subplot(2,2,1)
plot(cos(t),sin(t))
subplot(2,2,2)
plot(cos(t),sin(2*t))
subplot(2,2,3)
plot(cos(t),sin(3*t))
subplot(2,2,4)
plot(cos(t),sin(4*t))
```



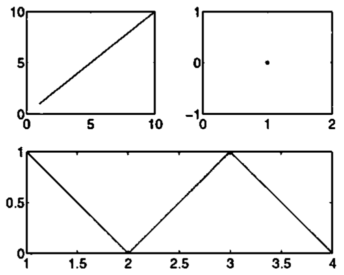
If you want to clear one of the plots in a subplot without affecting the others you can use the `cla` (clear axes) command. Continuing the previous example:

```
subplot(2,2,2)
cla
```



As long as your subplots are based on an array of 9 × 9 little plots or less, you can use a simplified syntax. For example, `subplot(221)` or `subplot 221` are equivalent to `subplot(2,2,1)`. You can mix different subplot arrays on the same figure, as long as the plots do not overlap:

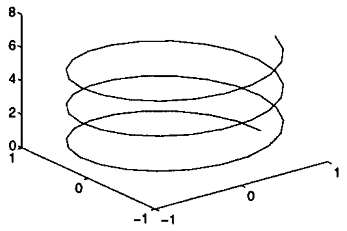
```
subplot 221
plot(1:10)
subplot 222
plot(0,'*')
subplot 212
plot([1 0 1 0])
```



4.6 Three-Dimensional Plots

The `plot3` command is the 3-d equivalent of `plot`:

```
t = 0:.1:2*pi;
plot3(cos(3*t),sin(3*t),t)
```



The three dimensional spiral can be better visualised by changing the orientation of the axes. You can invoke a mouse-based 3-d axis mover by typing:

```
rotate3d
```

If you click the mouse button down on the plot and drag, you can move the axes and view the plot from any angle. Release the mouse button to redraw the data. Type `rotate3d` again to turn off this behaviour.

4.7 Axes

So far we have allowed MATLAB to choose the axes for our plots. You can change the axes in many ways:

`axis([`*xmin xmax ymin ymax*`])`	sets the axes' minimum and maximum values
`axis square`	makes the axes the same length
`axis equal`	makes the axes the same scale
`axis tight`	sets the axes limits to the range of the data
`axis auto`	allows MATLAB to choose axes limits
`axis off`	removes the axes leaving only the plotted data
`axis on`	puts the axes back again
`grid on`	draws dotted grid lines
`grid off`	removes grid lines
`grid`	toggles the grid
`box`*	toggles the box
`zeroaxes`*	draws the x-axis at $y = 0$ and vice-versa

The functions marked with an asterisk * are nonstandard features, implemented in this book's companion m-files.[1]

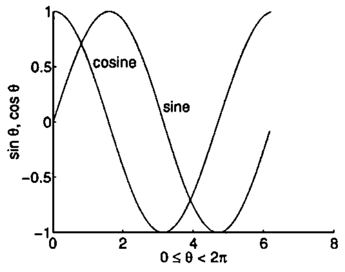
4.8 Labels

You can put labels, titles, and text on a plot by using the commands:

`xlabel('`*text*`')`	
`ylabel('`*text*`')`	
`zlabel('`*text*`')`	
`title('`*text*`')`	
`text(`*x*`,`*y*`,'`*text*`')`	places text at position x, y
`gtext('`*text*`')`	use mouse to place text

To put mathematics in labels you can use MATLAB's backslash notation (familiar to users of the TEX typesetting system):

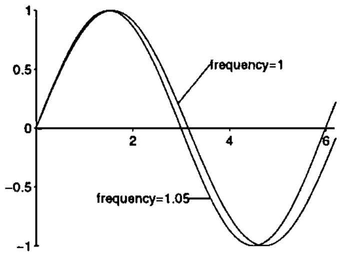
```
t = 0:.1:2*pi;
y1 = cos(t);
y2 = sin(t);
plot(t,y1,t,y2)
xlabel('0 \leq \theta < 2\pi')
ylabel('sin \theta, cos \theta')
text(1,cos(1),' cosine')
text(3,sin(3),' sine')
box
```



[1]MATLAB version 5.3 implements its own version of the `box` command.

Companion M-Files Feature 2 *To label many curves on a plot it is better to put the text close to the curves themselves rather than in a separate legend off to one side. Legends force the eye to make many jumps between the plot and the legend to sort out which line is which. Although* MATLAB *comes equipped with a* ***legend*** *function, I prefer to use the companion m-file* ***curlabel****, which is good especially for labelling plots which are close together:*

```
t = 0:.1:2*pi;
plot(t,sin(t),t,sin(1.05*t))
curlabel('frequency = 1')
curlabel('frequency = 1.05')
axis([0 max(t) -1 1])
zeroaxes
```



You must use the mouse to specify the start and end points of the pointer lines. The echo from the function can be pasted into an m-file for future use.

5 More Matrix Algebra

You can multiply two matrices together using the * operator:

```
>> a = [1 2;3 4]
a =
     1     2
     3     4
>> b = [1 0 1 0;0 1 1 0]
b =
     1     0     1     0
     0     1     1     0
>> a*b
ans =
     1     2     3     0
     3     4     7     0

>> u = [1 2 0 1]
u =
     1     2     0     1
>> v = [1 1 2 2]'
```

```
v  =
     1
     1
     2
     2
>> v*u
ans  =
     1     2     0     1
     1     2     0     1
     2     4     0     2
     2     4     0     2
>> u*v
ans  =
     5
```

The matrix inverse can be found with the `inv` command:

```
>> a = pascal(3)
a  =
     1     1     1
     1     2     3
     1     3     6
>> inv(a)
ans  =
     3    -3     1
    -3     5    -2
     1    -2     1
>> a*inv(a)
ans  =
     1     0     0
     0     1     0
     0     0     1
```

To multiply the elements of two matrices use the `.*` operator:

```
>> a = [1 2;3 4]
a  =
     1     2
     3     4
>> b = [2 3;0 1]
b  =
     2     3
     0     1

>> a.*b
ans  =
     2     6
     0     4
```

To raise the elements of a matrix to a power use the `.^` operator:

```
>> a = pascal(3)
a  =
     1     1     1
     1     2     3
     1     3     6
>> a.^2
ans  =
     1     1     1
     1     4     9
     1     9    36
```

6 Basic Data Analysis

The following functions can be used to perform data analysis functions:

`max`	maximum
`min`	minimum
`find`	find indices of nonzero elements
`mean`	average or mean
`median`	median
`std`	standard deviation
`sort`	sort in ascending order
`sortrows`	sort rows in ascending order
`sum`	sum of elements
`prod`	product of elements
`diff`	difference between elements
`trapz`	trapezoidal integration
`cumsum`	cumulative sum
`cumprod`	cumulative product
`cumtrapz`	cumulative trapezoidal integration

As we have seen with the `plot` command, MATLAB usually prefers to work with matrix columns, rather than rows. This is true for many of MATLAB's functions, which work on columns when given matrix arguments. For example:

```
>> a = magic(3)
a  =
     8     1     6
     3     5     7
     4     9     2
>> m = max(a)
m  =
     8     9     7
```

`max` returns a vector containing the maximum value of each column. When given a vector, `max` returns the maximum value:

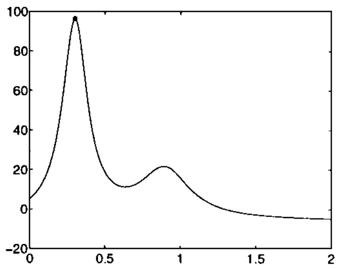
```
>> max(m)
ans  =
     9
```

To find the index corresponding to the maximum value, supply two output arguments to `max`:

```
>> [v,ind] = max(m)
v  =
     9
ind  =
     2
```

The first argument is the maximum value and the second is the index of the maximum value. Another example is

```
>> x = 0:.01:2;
>> y = humps(x);
>> plot(x,y)
>> [v,ind] = max(y)
v  =
   96.5000
ind  =
    31
>> hold on
>> plot(x(ind),y(ind),'ro')
>> x(ind)
ans  =
    0.3000
>> y(ind)
ans  =
   96.5000
```



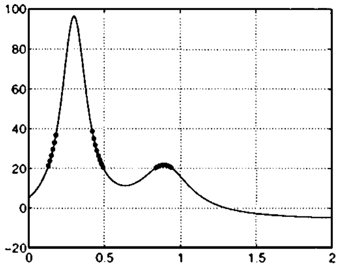
The `find` function is often used with relational and logical operators:

Relational operators	`==`	equal to
	`~=`	not equal to
	`<`	less than
	`>`	greater than
	`<=`	less than or equal to
	`>=`	greater than or equal to

Logical operators	`&`	AND
	`|`	OR
	`~`	NOT
	`xor`	EXCLUSIVE OR
	`any`	True if any element is non-zero
	`all`	True if all elements are non-zero

We continue the previous example and use `find` to plot the part of the peaks function that lies between $y = 20$ and $y = 40$:

```
clf
ind = find(20<=y & y<=40);
plot(x,y,x(ind),y(ind),'o')
grid
```



When used with one output argument, `find` assumes that the input is a vector. When the input is a matrix `find` first strings out the elements as a single column vector and returns the corresponding indices. As an example we consider the spiral matrix:

```
>> s = spiral(3)
s  =
     7     8     9
     6     1     2
     5     4     3
```

We find the elements of `s` less than 6:

```
>> s<6
ans  =
     0     0     0
     0     1     1
     1     1     1
>> find(s<6)
ans  =
     3
     5
     6
     8
     9
```

The result of `find` is a vector of indices of `s` counted down the first column, then the second, and then the third. The following example shows how the results of the find command can be used to extract elements from a matrix that satisfy a logical test:

```
>> s = 100*spiral(3)
s  =
   700   800   900
   600   100   200
   500   400   300
>> ind = find(s>400)
ind  =
     1
     2
     3
     4
     7
>> s(ind)
ans  =
   700
   600
   500
   800
   900
>> s(s>400)
ans  =
   700
   600
   500
   800
   900
```

After introducing graphics of functions of two variables in the next section, we will see how the `find` command can be used to do the three-dimensional equivalent of the plot shown on page 23, where the domain of a curve satisfying a logical test was extracted.

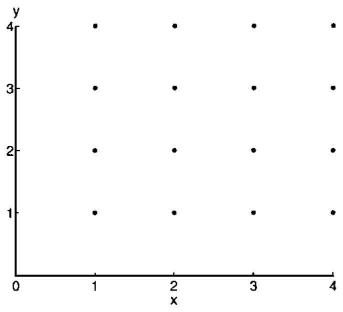
7 Graphics of Functions of Two Variables

7.1 Basic Plots

A MATLAB surface is defined by the z coordinates associated with a set of (x, y) coordinates. For example, suppose we have the set of (x, y) coordinates:

$$(x,y) = \begin{pmatrix} 1,1 & 2,1 & 3,1 & 4,1 \\ 1,2 & 2,2 & 3,2 & 4,2 \\ 1,3 & 2,3 & 3,3 & 4,3 \\ 1,4 & 2,4 & 3,4 & 4,4 \end{pmatrix}.$$

The points can be plotted as (x, y) pairs:



The (x, y) pairs can be split into two matrices:

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix}; \quad y = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{pmatrix}.$$

The matrix x varies along its columns and y varies down its rows. We define the surface z:

$$z = \sqrt{x^2 + y^2};$$

which is the distance of each (x, y) point from the origin $(0, 0)$. To calculate z in MATLAB for the x and y matrices given above, we begin by using the `meshgrid` function, which generates the required `x` and `y` matrices:

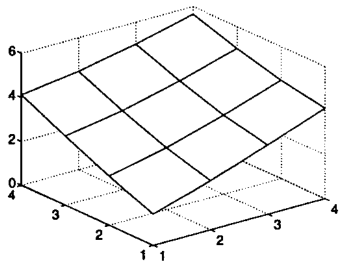
```
>> [x,y] = meshgrid(1:4)
x  =
      1       2       3       4
      1       2       3       4
      1       2       3       4
      1       2       3       4
y  =
      1       1       1       1
      2       2       2       2
      3       3       3       3
      4       4       4       4
```

Now we simply convert our distance equation to MATLAB notation; $z = \sqrt{x^2 + y^2}$ becomes:

```
>> z = sqrt(x.^2 + y.^2)
z  =
     1.4142     2.2361     3.1623     4.1231
     2.2361     2.8284     3.6056     4.4721
     3.1623     3.6056     4.2426     5.0000
     4.1231     4.4721     5.0000     5.6569
```

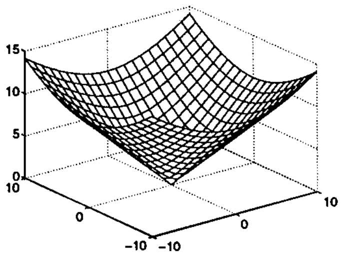
We can plot the surface `z` as a function of `x` and `y`:

```
mesh(x,y,z)
```



We can expand the domain of the calculation by increasing the input to `meshgrid`. Be careful to end the lines with a semicolon to avoid being swamped with numbers:

```
[x,y] = meshgrid(-10:10);
z = sqrt(x.^2 + y.^2);
mesh(x,y,z)
```



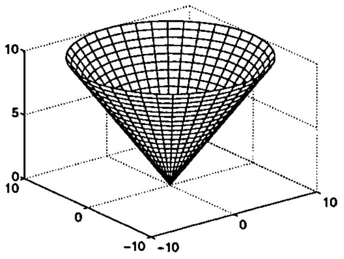
The surface is an inverted cone, with its apex at $(0, 0, 0)$.

Companion M-Files Feature 3 *A clearer plot can be produced using a polar grid, instead of a rectilinear grid. We can use the companion function* `polarmesh` *to produce such a plot. First we define a polar grid of points:*

```
[r,th] = meshgrid(0:.5:10,0:pi/20:2*pi);
```

Then display the surface defined by $z = r$*:*

```
polarmesh(r,th,r)
```



A more interesting surface is

$$z = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10(\tfrac{1}{5}x - x^3 - y^5)e^{-x^2-y^2} \cdots$$
$$- \tfrac{1}{3}e^{-(x+1)^2-y^2} .$$

In MATLAB notation you could type:

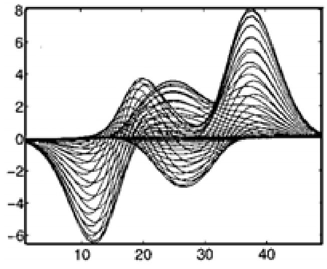
```
z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

but you do not have to type this because it is already defined by the function `peaks`. Before plotting we define the data and set the colour map to gray:

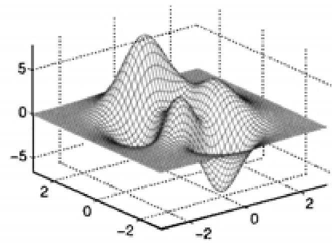
```
[x,y,z] = peaks;
colormap(gray)
```

The following plots show 10 different ways to view this data.

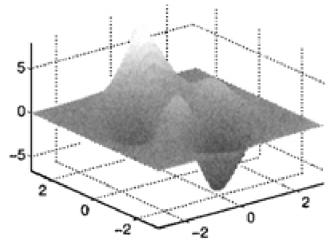
```
clf
plot(z)
```



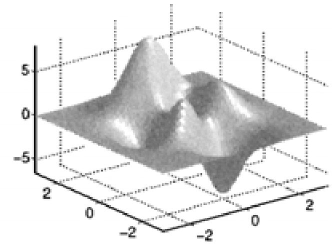
```
mesh(x,y,z)
```



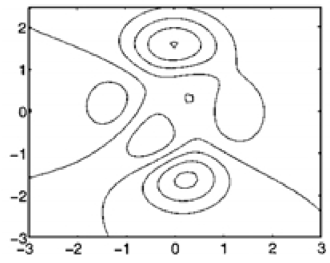
```
surf(x,y,z)
shading flat
```



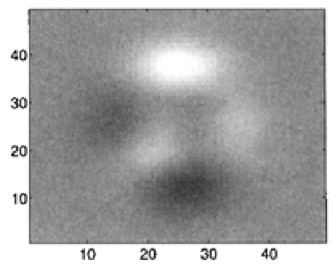
```
surfl(x,y,z)
shading flat
```



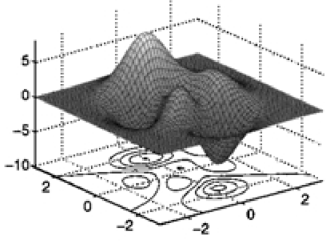
```
contour(x,y,z)
```



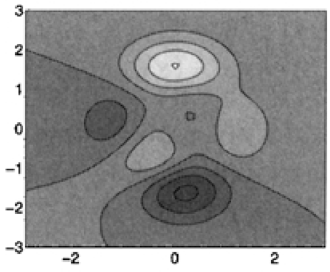
```
imagesc(z)
axis xy
```



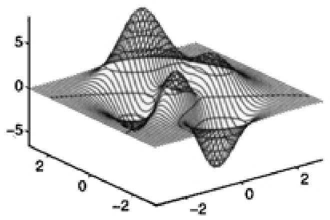
```
surfc(x,y,z)
```



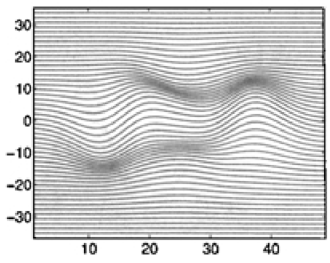
```
contourf(x,y,z)
```



```
plot3(x,y,z,'k')
hold on
contour3(x,y,z,'k')
```



```
clf
spanplot(z)
```



The `contour` function plots the contours using the current colour map's colours (see next section). Adding the specifier `'k'` to the end of the argument list draws the contours in black. The `spanplot` function is nonstandard and is included in the companion software.

You should experiment with these plots. Try typing `help` for each of these plot commands. Explore the various ways of shading a surface, try using different colour maps (see next section) or viewing angles (`help view`), or try modifying the surface and replotting. Remember that `rotate3d` can be used to switch on a click-and-drag three-dimensional view changer: click down on the plot and drag it to alter the viewing angle; release the mouse to redraw the plot. (If `rotate3d` is already switched on, typing `rotate3d` again will switch it off.)

7.2 Colour Maps

MATLAB uses a matrix called a colour map to apply colour to surfaces and images. The idea is that different colours will be used to draw various parts of the plot depending on the colour map. The colour map is a list of triplets corresponding to the intensities of the red, green, and blue video components, which add up to yield other colours. The intensities must be between zero and one. Some example colours are shown in this table.

Red	Green	Blue	Colour
0	0	0	Black
1	1	1	White
1	0	0	Red
0	1	0	Green
0	0	1	Blue
1	1	0	Yellow
1	0	1	Magenta
0	1	1	Cyan
.5	.5	.5	Gray
.5	0	0	Dark red
1	.62	.4	Dark orange
.49	1	.83	Aquamarine
.95	.9	.8	Parchment

Yellow, for example, consists of the combination of the full intensities of red and green, with no blue, while gray is the combination of 50% intensities of red, green, and blue.

You can create your own colour maps or use any of MATLAB's many predefined colour maps:

```
hsv     hot      gray     bone        copper   pink
white   flag     lines    colorcube   jet      prism
cool    autumn   spring   winter      summer
```

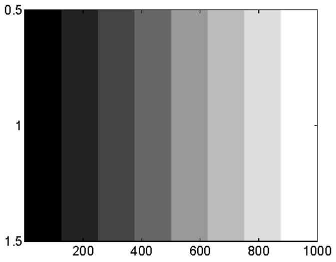
Two nonstandard colour maps that are supplied in the companion software include `redblue` and `myjet`. The first consists of red blending to blue through shades of gray. The second consists of a modification of the `jet` colour map that has white at the top instead of dark red.

These functions all take an optional parameter that specifies the number of rows (colours) in the colour map matrix. For example, typing `gray(8)` creates an 8×3 matrix of various levels of gray:

```
>> gray(8)
ans  =
         0         0         0
    0.1429    0.1429    0.1429
    0.2857    0.2857    0.2857
    0.4286    0.4286    0.4286
    0.5714    0.5714    0.5714
    0.7143    0.7143    0.7143
    0.8571    0.8571    0.8571
    1.0000    1.0000    1.0000
```

To tell MATLAB to use a colour map, type it as an input to the `colormap` function:

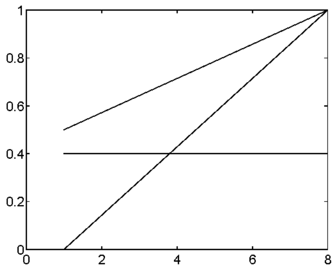
```
m = gray(8);
colormap(m)
imagesc(1:1000)
```



Most of MATLAB's surface viewing functions use the colour map to apply colour to the surface depending on the z-value. The `imagesc` function produces a coloured image of the matrix argument, colouring each element depending on its value. The smallest element will take the colour specified in the first row of the colour map, the largest element will take the colour specified in the last row of the colour map, and all the elements in between will take linearly interpolated colours.

To get a plot of the levels of red, green, and blue in the current colour map use `rgbplot`:

```
colormap(summer)
rgbplot(colormap)
```

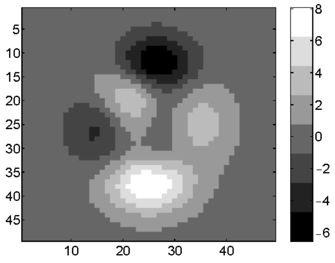


On the screen the lines corresponding to the red, green, and blue components of the colour map are coloured red, green, and blue, respectively.

7.3 Colour Bar

To display the current colour map use the `colorbar` function:

```
z = peaks;
colormap(gray(8))
imagesc(z)
colorbar
```



7.4 Good and Bad Colour Maps

Much research has been done on human perception of colours and, in particular, how different viewers interpret coloured images as value-scales.

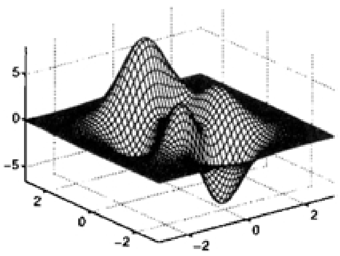
The conclusion is that most viewers find it very difficult to interpret these sorts of images; the cognitive switch from, for example, ROYGBIV to amplitude is very slow and nonintuitive. A way out of this is to use a palette of slightly varying, nonsaturated colours. These sorts of colours have been used to create high-quality geographic maps for many years. Most of MATLAB's colour maps consist of highly saturated colours (including the default colour map, which is `jet(64)`). It is better to forgo these sorts of colour maps and stick with the calmer ones such as `gray`, `bone`, or `summer`. The `gray` colour map has the added advantage that printed versions will reproduce easily, for example, on a photocopier.[2] The companion m-files include some other colour maps: `redblue`, `myjet`, `yellow`, `green`, `red`, and `blue`.

To distinguish adjacent patches of subtly different colours, the eye can be helped by enclosing the patches with a thin dark edge. The `contourf` function, therefore, is an excellent way of displaying this sort of data.[3]

7.5 Extracting Logical Domains

Let us look again at the peaks function:

```
[x,y,z] = peaks;
surfl(x,y,z)
axis tight
colormap(gray(64))
```



Suppose we want to extract the part of this surface for which the z values lie between 2 and 4. We use exactly the same technique as was given on page 23. The `find` command is used first to find the indices of the `z` values that satisfy the logical test:

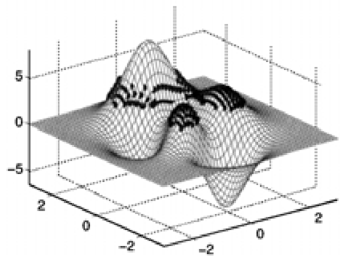
```
>> ind = find(2<=z & z<=4);
>> size(ind)
ans  =
   234     1
```

There are 234 elements in `z` that satisfy our condition. We can plot these elements over the surface as follows:

[2]Edward R. Tufte, *Visual Explanations* (Graphics Press, Cheshire Connecticut, 1997), pp. 76–77.

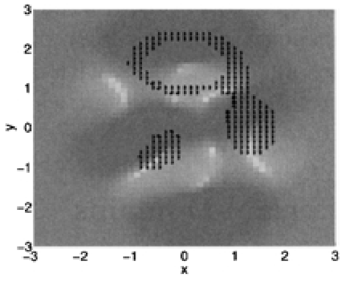
[3]Edward R. Tufte, *Envisioning Information* (Graphics Press, Cheshire Connecticut, 1990), pp. 88ff.

```
hold on
plot3(x(ind),y(ind),z(ind),'.')
```



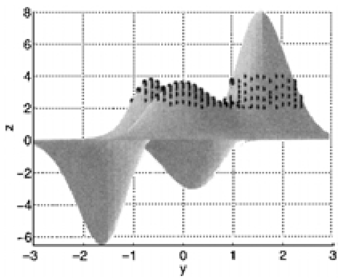
The x, y domain of the extracted points can be shown clearly with an overhead view:

```
view(2)
xyz
shading flat
```



The associated z values can be shown with a side view:

```
view(90,0)
grid
```



7.6 Nonrectangular Surface Domains

The `polarmesh` function given on page 26 showed a conical function defined over a circular domain of x and y points. Let us now look a bit more generally at how to define such nonrectangular domains for surfaces.

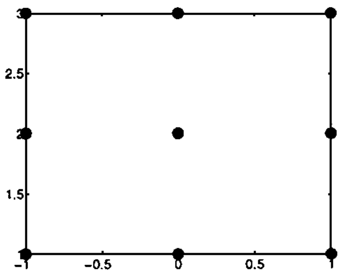
The standard MATLAB functions, including graphics functions, tend to like working with rectangular matrices: each row must have the same number of columns. For surfaces, this requirement applies to the x, y and z matrices that specify the surface. Let us demonstrate by way of an example. First we generate a rectangular domain of x and y points, with x going from -1 to 1, and y going from 0 to 2:

```
>> [x,y] = meshgrid(-1:1,1:3)
```

```
x  =
    -1     0     1
    -1     0     1
    -1     0     1
y  =
     1     1     1
     2     2     2
     3     3     3
```

This set of points defines a rectangular domain because the rows of x are identical and the columns of y are identical. We can make a plot of the points (as we did on page 25):

```
clf
plot(x,y,'.')
```

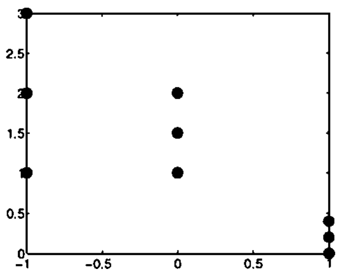


Now let us change the y matrix a bit:

```
>> y = [[1; 2; 3] [1; 1.5; 2] [0; .2; .4]]
y  =
    1.0000    1.0000         0
    2.0000    1.5000    0.2000
    3.0000    2.0000    0.4000
```

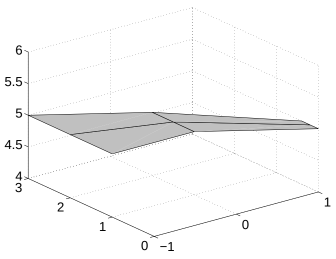
The plot of this data looks like a bent triangle:

```
plot(x,y,'.')
```



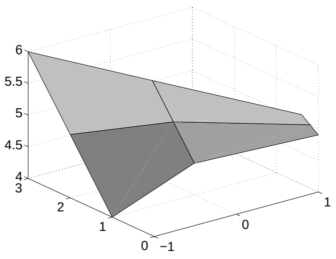
To define a surface over this domain we simply have to supply the z values. We can start by simply defining a constant z:

```
>> z = 5*ones(3,3)
z  =
     5     5     5
     5     5     5
     5     5     5
>> surf(x,y,z)
```



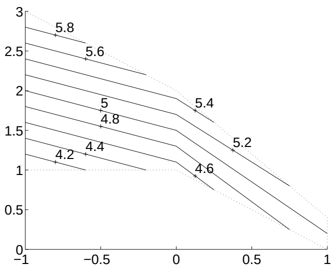
But, of course, the z values need not be constant:

```
>> z = [[4; 5; 6] [4.5; 5; 5.5] [4.9; 5; 5.1]]
z  =
    4.0    4.5    4.9
    5.0    5.0    5.0
    6.0    5.5    5.1
>> surf(x,y,z)
```



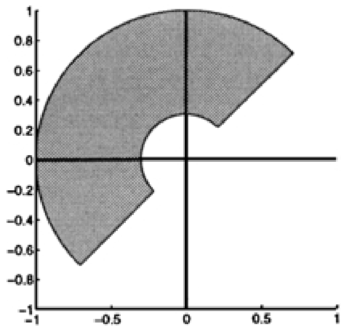
Other graphics functions can also handle nonrectangular grids. Here is an example using the `contour` function:

```
cs = contour(x,y,z,'k');
clabel(cs)
i = [1 4 7 9 6 3 1];
hold on
plt(x(i),y(i),':')
```



The contour levels are labelled using the `clabel` command, and the region defined by the x and y points is outlined by the dotted line. The contours that the labels refer to are marked by small plus signs '`+`'. The outline around the bent domain is drawn using the `x` and `y` matrices indexed using the vector `i`. The vector `i` extracts the appropriate points from the x and y matrices using the columnar indexing described in section 3.4 on page 9. The other surface graphics functions—`mesh`, `surfl`, `surfc`, and `contourf`—can handle such nonrectangular grids equally well. The `image` and `imagesc` functions assume equally spaced rectangular grids and cannot handle anything else. (The `pcolor` function draws a surface and sets the view point to directly overhead, so it is not discussed separately.)

Let us now do another example of a surface defined over a nonrectangular grid. We want to define a set of points that cover the semi-annular region as shown in the diagram at right. To define such a set of points we use a polar grid based on radial and angular coordinates r and θ. We use the following limits on these coordinates:



$$.3 \le r \le 1$$
$$\pi/4 \le \theta \le 5\pi/4$$

These are set up in MATLAB as follows:

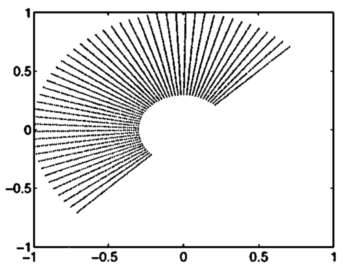
```
rv = linspace(.3,1,50);
thv = linspace(pi/4,5*pi/4,50);
[r,th] = meshgrid(rv,thv);
```

where the calls to `linspace` produce vectors of 50 points covering the intervals. The x and y points are defined by the following trigonometric relations:

```
x = r.*cos(th);
y = r.*sin(th);
```

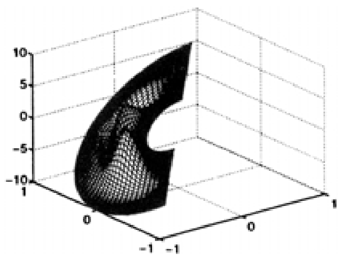
Now our semi-annular region is defined. To prove it, let us plot the points:

```
plot(x,y,'.')
```



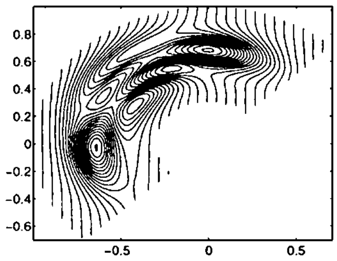
Again, we can define any z matrix we like. Just for fun, we use the peaks function of the right size and add a linear ramp:

```
z = peaks(50) + 10*x;
surf(x,y,z)
```



As we did in the previous example, we check that the `contour` function works (omitting the labels this time, and upping the number of contours drawn to 30):

```
contour(x,y,z,30);
```



You may have noticed that the semi-annular region does not appear as a circular segment in our plots. That is because the axes are not square. To get square axes you can use the `axis square` command as described on pages 18 and 120.

In this section we have looked at surfaces having domains that could be defined in terms of rectangular x and y data matrices. Domains that cannot be defined with such matrics are discussed in section 36 on page 157. For example all x values may not have the same number of y values, or the x, y points could be scattered about in an irregular way.

8 M-Files

Until now we have driven MATLAB by typing in commands directly. This is fine for simple tasks, but for more complex ones we can store the typed input into a file and tell MATLAB to get its input from the file. Such files must have the extension "`.m`". They are called m-files. If an m-file contains MATLAB statements just as you would type them into MATLAB, they are called *scripts*. M-files can also accept input and produce output, in which case they are called *functions*.

8.1 Scripts

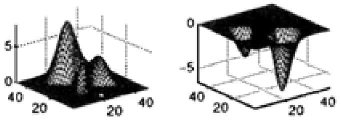
Using your text editor create a file called `mfile1.m` containing the following lines:

```
z = peaks;
zplot = z;

% Do the peaks:

clf
subplot(221)
ind = find(z<0);
zplot(ind) = zeros(size(ind));
mesh(zplot)
axis tight
```

```
% Do the valleys:

subplot(222)
ind = find(z>0);
zplot = z;
zplot(ind) = zeros(size(ind));
mesh(zplot)
axis tight
```

Now try this in the MATLAB window:

```
mfile1
```



MATLAB has executed the instructions in `mfile1.m` just as if you had typed them in. The lines beginning with the percent sign `%` are ignored by MATLAB so they can be used to put comments in your code. Blank lines can be used to improve readability.

Any variables created by a script m-file are available in the command window after the m-file completes running:

```
>> clear
>> whos
>> mfile1
>> whos
  Name          Size            Bytes  Class
  ind        1544x1             12352  double array
  z            49x49            19208  double array
  zplot        49x49            19208  double array

Grand total is 6346 elements using 50768 bytes
```

These variables are said to exist in the MATLAB *workspace*. Scripts can also operate on variables that already exist in the workspace.

You can type the name of a script file within another script file. For example you could create another file called `mfile2` that contains the text line `mfile1`; the contents of `mfile1` will then be executed at that point within `mfile2`.

8.2 Functions

Functions are m-files that can be used to extend the MATLAB language. Functions can accept input arguments and produce output arguments. Many of MATLAB's own commands are implemented as m-files; try typing `type mean` to see how MATLAB calculates the mean. Functions use variables that are local to themselves and do not appear in the main workspace. This is an example of a function:

```
function x = quadratic(a,b,c)

% QUADRATIC Find roots of a quadratic equation.
%
% X = QUADRATIC(A,B,C) returns the two roots of the
% quadratic equation
%
%                  y = A*x^2 + B*x + C.
%
% The roots are contained in X = [X1 X2].

% A. Knight, July 1997
delta = 4*a*c;
denom = 2*a;
rootdisc = sqrt(b.^2 - delta); % Root of the discriminant
x1 = (-b + rootdisc)./denom;
x2 = (-b - rootdisc)./denom;
x = [x1 x2];
```

Function m-files must start with the word `function`, followed by the output variable(s), an equals sign, the name of the function, and the input variable(s). Functions do not have to have input or output arguments. If there is more than one input or output argument, they must be separated by commas. If there are one or more input arguments, they must be enclosed in brackets, and if there are two or more output arguments, they must be enclosed in square brackets. The following illustrate these points (they are all valid function definition lines):

```
function [xx,yy,zz] = sphere(n)
function fancyplot
function a = lists(x,y,z,t)
```

Function names must follow the same rules as variable names. The file name is the function name with "`.m`" appended. If the file name and the function name are different, MATLAB uses the file name and ignores the function name. You should use the same name for both the function and the file to avoid confusion.

Following the function definition line you should put comment lines that explain how to use the function. These comment lines are printed in the command window when you type `help` followed by the m-file name at the prompt:

```
>> help quadratic
  QUADRATIC Find roots of a quadratic equation.
  X = QUADRATIC(A,B,C) returns the two roots of the
  quadratic equation
                       y = A*x^2 + B*x + C.
  The roots are contained in X = [X1 X2].
```

MATLAB only echoes the comment lines that are contiguous; the first non-comment line, in this case the blank line before the signature, tells MATLAB that the help comments have ended. The first line of the help comments is searched and, if successful, displayed when you type a `lookfor` command.

Comment lines can appear anywhere in the body of an m-file. Comments can be put at the end of a line of code:

```
rootdisc = sqrt(b.^2 - delta); % Root of the discriminant
```

Blank lines can appear anywhere in the body of an m-file. Apart from ending the help comment lines in a function, blank lines are ignored.

8.3 Flow Control

MATLAB has four kinds of statements you can use to control the flow through your code:

`if`, `else` and `elseif` execute statements based on a logical test
`switch`, `case` and `otherwise` execute groups of statements based on a logical test
`while` and `end` execute statements an indefinite number of times, based on a logical test
`for` and `end` execute statements a fixed number of times

If, Else, Elseif

The basic form of an `if` statement is:

```
if test
   statements
end
```

The ***test*** is an expression that is either 1 (true) or 0 (false). The ***statements*** between the `if` and `end` statements are executed if the ***test*** is true. If the ***test*** is false the ***statements*** will be ignored and execution will resume at the line after the `end` statement. The ***test*** expression can be a vector or matrix, in which case *all* the elements must be equal to 1 for the ***statements*** to be executed. Further tests can be made using the `elseif` and `else` statements.

Exercise 2 *Write a function m-file that takes a vector input and returns 1 if all of the elements are positive, −1 if they are all negative, and zero for all other cases. Hint: Type* `help all`*. (Answer on page 183.)*

Switch

The basic form of a `switch` statement is:

```
    switch test
           case result1
                statements
           case result2
                statements
           .
           .
           .
           otherwise
                statements
    end
```

The respective *statements* are executed if the value of *test* is equal to the respective *result*s. If none of the `case`s are true, the `otherwise` statements are done. Only the first matching case is carried out. If you want the same *statements* to be done for different `case`s, you can enclose the several *result*s in curly brackets:

```
switch x
  case 1
    disp('x is 1')
  case {2,3,4}
    disp('x is 2, 3 or 4')
  case 5
    disp('x is 5')
  otherwise
    disp('x is not 1, 2, 3, 4 or 5')
end
```

While

The basic form of a `while` loop is

```
    while test
          statements
    end
```

The *statements* are executed repeatedly while the value of *test* is equal to 1. For example, to find the first integer n for which $1+2+\cdots+n$ is is greater than 1000:

```
n = 1;
while sum(1:n)<=1000
  n = n+1;
end
```

A quick way to 'comment out' a slab of code in an m-file is to enclose it between a `while 0` and `end` statements. The enclosed code will never be executed.

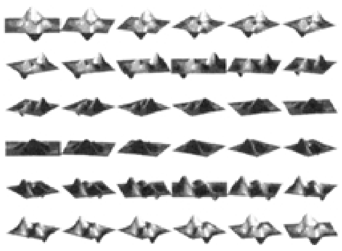
For

The basic form of a `for` loop is:

```
for index = start:increment:stop
    statements
end
```

You can omit the increment, in which case an increment of 1 is assumed. The increment can be positive or negative. During the first pass through the loop the *index* will have the value *start*. The *index* will be increased by *increment* during each successive pass until the *index* exceeds the value *stop*. The following example produces views of the `peaks` function from many angles:

```
clf
colormap(gray)
plotnum = 1;
z = peaks(20);
for az = 0:10:350
  subplot(6,6,plotnum)
  surfl(z),shading flat
  view(az,30)
  axis tight
  axis off
  plotnum = plotnum + 1;
end
```



The index of a `for` loop can be a vector or a matrix. If it is a vector the loop will be done as many times as the number of elements in the vector, with the index taking successive values of the vector in each pass. If the index is a matrix, the loop will be done as many times as there are columns in the matrix, with the index taking successive columns of the matrix in each pass. For example:

```
>> q = pascal(3)
q  =
     1     1     1
     1     2     3
     1     3     6
>> for i = q,i,end
i  =
     1
     1
     1
```

```
i  =
     1
     2
     3
i  =
     1
     3
     6
```

Vectorised Code

MATLAB is a matrix language, and many of its algorithms are optimised for matrices. MATLAB code can often be accelerated by replacing `for` and `while` loops with operations on matrices. In the following example, we calculate the factorial of the numbers from 1 to 500 using a `for` loop. Create a script m-file called `factorialloop.m` that contains the following code:

```
for number = 1:500
  fact = 1;
  for i = 2:number
    fact = fact*i;
  end
  y(number) = fact;
end
```

We can time how long this program takes to run by using the stopwatch functions `tic` and `toc`:

```
>> tic;factorialloop;toc
elapsed_time  =
    4.6332
```

which is the time in seconds. The same calculation can be done in much less time by replacing the internal `for` loop by the `prod` function. Create an m-file called `factorialvect.m`:

```
for number = 1:500
  y(number) = prod(1:number);
end
```

This version takes about a tenth of the time:

```
>> clear
>> tic;factorialvect;toc
elapsed_time  =
    0.4331
```

Further increases in speed can be achieved by pre-allocating the output matrix y. If we have an m-file called factorialpre.m:

```
y = zeros (1,500);
for number = 1:500
   y(number) = prod(1:number);
end
```

the execution time is about 10% faster:[4]

```
>> clear
>> tic;factorialpre;toc
elapsed_time  =
    0.3752
```

More on vectorising code is given in Part II (see page 175).

8.4 Comparing Strings

The tests in flow control statements often involve strings (arrays of characters). For example you may want to ask the user of an m-file a question which has a "yes" or "no" response, and adjust the flow accordingly. Although MATLAB has sophisticated menu utilities, the following is often sufficient to get a user input:

```
input('Do you want to continue (y or n) ? ','s');
```

The 's' at the end tells MATLAB to expect a string response, rather than a numerical response. The following MATLAB code tests for a 'y' response:

```
if strcmp(lower(ans(1)),'y')
   go_ahead
else
   return
end
```

The strcmp function compares strings, lower converts to lower-case characters and ans(1) selects the first letter of the response. Type help strcmp for more information. The return command returns to the invoking function or to the MATLAB prompt.

9 Data Files

Many techniques are available to read data into MATLAB and to save data from MATLAB. The load and save functions can load or save MATLAB format binary or plain ASCII files, and low-level input-output routines can be used for other formats.

[4]See MATLAB's gamma function if you are interested in computing factorials.

9.1 MATLAB Format

To save all the variables in the workspace onto disk use the `save` command. Typing `save keepfile` will save the workspace variables to a disk file called `keepfile.mat`, a binary file whose format is described in the MATLAB documentation. This data can be loaded into MATLAB by typing `load keepfile`.

To save or load only certain variables, specify them after the filename. For example, `load keepfile x` will load only the variable `x` from the saved file. The wild-card character `*` can be used to save or load variables according to a pattern. For example, `load keepfile *_test` loads only the variables that end with `_test`.

When the filename or the variable names are stored in strings, you can use the functional forms of these commands, for example:

```
save keepfile      is the same as   save('keepfile')
save keepfile x    ...              save('keepfile','x')
load keepfile      ...              A = 'keepfile'
                                    load(A)
```

Exercise 3 *The file* `clown.mat` *contains an image of a clown. What colour is his hair? (Answer on page 183.)*

9.2 ASCII Format

A file containing a list or table of numbers in ASCII format can be loaded into MATLAB. The variable containing the data is given the same name as the file name without the extension. For example, if a file `nums.dat` contained ASCII data, `load nums.dat` would load the data into a variable called `nums`. If the ASCII file contained a table of numbers, the variable would be a matrix the same size as the table.

Other functions are available to read various forms of delimiter-separated text files:

`csvread`	Read a comma separated value file
`csvwrite`	Write a comma separated value file
`dlmread`	Read ASCII delimited file
`dlmwrite`	Write ASCII delimited file

9.3 Other Formats

MATLAB's low-level input/output routines can be used to access more unusual data formats. They are listed here for reference:

File Opening and Closing:	`fclose`	`fopen`
Unformatted I/O:	`fread`	`fwrite`
Formatted I/O:	`fgetl`	`fprintf`
	`fgets`	`fscanf`
File Positioning:	`feof`	`fseek`
	`ferror`	`ftell`
	`frewind`	
String Conversion:	`sprintf`	`sscanf`

10 Directories

When you type a string of characters, say `asdf` at the MATLAB prompt and press return, MATLAB goes through the following sequence to try to make sense of what you typed:

1. Look for a variable called `asdf`;
2. Look for a built in MATLAB function called `asdf`;
3. Look in the current directory for an m-file called `asdf.m`;
4. Look in the directories specified by the MATLAB search path for an m-file called `asdf.m`.

The following commands are useful for working with different directories in MATLAB:

`cd`	Change to another directory
`pwd`	Display (print) current working directory
`dir`	Display contents of current working directory
`what`	Display MATLAB-relevant files in current working directory
`which`	Display directory containing specified function
`type`	Display file in the MATLAB window
`path`	Display or change the search path
`addpath`	Add directory to the search path
`rmpath`	Remove directory from the search path

If the directory name contains a blank space, enclose it in single quotes:

```
dir 'my documents'
```

(On PCs or Macintoshes you can use the Path Browser GUI to manipulate the path. Select 'File'→'Set Path' or click the Path Browser button on the tool bar.)

11 Startup

Each time you start MATLAB it looks for a script m-file called `startup.m` and, if it finds it, does it. Thus, you can use `startup.m` to do things like set the search path, set command and figure window preferences (e.g., set all your figures to have a black background), etc.

On PCs you should put the `startup.m` file in the directory called `C:\MATLAB\toolbox\local`. On UNIX workstations you should put your startup file in a directory called `matlab` immediately below your home directory: `~/matlab`.

12 Using MATLAB on Different Platforms

A MATLAB format binary (`.mat`) file that is saved on one platform (say, a PC or a Macintosh) can be transferred to a different platform (say, a Unix or VMS box) and loaded into MATLAB running on that platform. The mat-file contains information about the platform that saved the data. MATLAB checks to see if the file was saved on a different platform, and performs any necessary conversions automatically.

MATLAB m-files are ordinary ASCII text, and are immediately transportable between platforms. Different platforms may use different characters to terminate lines of text (with CR and LF characters), but MATLAB handles them all. However, the text editor you use must be able to handle the end-of-line characters correctly.

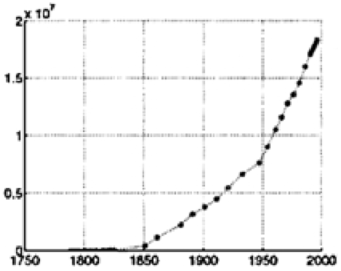
The program you use to transfer m-files or mat-files, for example, FTP or mail, must do so without corrupting the data. For FTP, for example, mat-files must be transmitted in *binary* mode and m-files must be transmitted in *ASCII* mode.

13 Log Scales

When dealing with data that varies over several orders of magnitude a plain linear plot sometimes fails to display the variation in the data. For example, consider the census estimates[5] of Australia's European population at various times. If this data is contained in the file `population.dat`, we can load and plot it as follows:

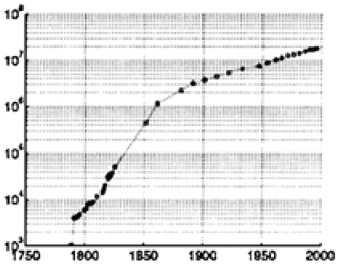
[5]Australian Bureau of Statistics Web Page, `http://www.statistics.gov.au`, and *Australians: A Historical Library, Australians: Historical Statistics*, Fairfax, Syme & Weldon Associates, 235 Jones Street, Broadway, New South Wales 2007, Australia, 1987, pp. 25,26.

```
load population.dat
year = population(:,1);
P = population(:,2);
plot(year,P,':o')
box;grid
```



The European population prior to 1850 was very low and we are unable to see the fine detail. Detail is revealed when we use a logarithmic y-scale:

```
semilogy(year,P,':o')
box;grid
```



The following functions implement logarithmic axes:

`loglog`	Both axes logarithmic
`semilogx`	logarithmic x-axis
`semilogy`	logarithmic y-axis

14 Curve Fitting—Matrix Division

We continue with the example of Australian population data given in the previous section. Let us see how well a polynomial fits this data. We assume the data can be modelled by a parabola:

$$p = c_0 + c_1 x + c_2 x^2$$

where x is the year, c_0, c_1, and c_2 are coefficients to be found, and p is the population. We write down this equation substituting our measured data:

$$\begin{aligned} p_1 &= c_0 + c_1 x_1 + c_2 x_1^2 \\ p_2 &= c_0 + c_1 x_2 + c_2 x_2^2 \\ &\vdots \\ p_N &= c_0 + c_1 x_N + c_2 x_N^2 \end{aligned}$$

Where p_i is the population for year x_i, and $i = 1, 2, \ldots N$. We can write this series of equations as a matrix equation:

$$\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ & \vdots & \\ 1 & x_N & x_N^2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} .$$

Or, defining matrices,

$$\mathbf{P} = \mathbf{X} \cdot \mathbf{C} .$$

In MATLAB the **X** matrix is calculated as follows:

```
>> X = [ones(size(year)) year year.^2]
X  =
           1        1788      3196944
           1        1790      3204100
           .
           .
           .
           1        1993      3972049
           1        1994      3976036
           1        1995      3980025
```

The backslash operator solves the equation for the coefficient matrix **C**:

```
>> C = X\P
C  =
   1.0e+09 *
    2.0067
   -0.0022
    0.0000
```

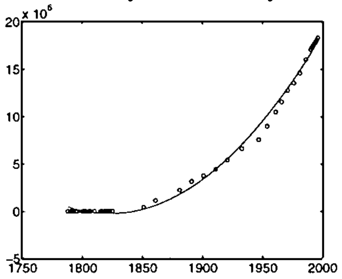
The third coefficient is not really zero; it is simply too small (compared to 2.0×10^9) to show in the default output format. We can change this by typing:

```
>> format long e
>> C
C  =
     2.006702229622023e+09
    -2.201930087288049e+06
     6.039665477603122e+02
```

The backslash operator does its best to solve a system of linear equations using Gaussian elimination or least-squares algorithms, depending on whether the system is exact, or over- or under-determined. We can display the resulting fit to the data by calculating the parabola. We use matrix multiplication to calculate the polynomial over a fine set of points separated by half a year:

```
year_fine = (year(1):0.5:year(length(year)))';
Pfine = [ones(size(year_fine)) year_fine year_fine.^2]*C;
```

```
plot(year,P,'o',...
     year_fine,Pfine)
```



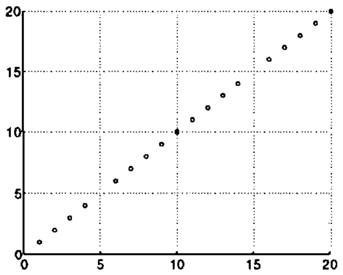
This technique can be used to fit any function that is linear in its parameters. (MATLAB provides the functions `polyfit` and `polyval` as easy interfaces to the functionality that we have just illustrated using matrix multiplication and division.)

> **Exercise 4** *Use this technique to fit an exponential curve to the population data. Hint: Take logs. (Answer on page 183.)*

15 Missing Data

Real-world measurements are often taken at regular intervals; for example, the position of a comet in the sky measured each night, or the depth of the sea along a line at 1 metre increments. Environmental effects or equipment failure (a cloudy night or a failed depth meter) sometimes result in a set of data that has missing values. In MATLAB these can be represented by `NaN`, which stands for "not-a-number". `NaN` is also given by MATLAB as the result of undefined calculations such as 0/0. MATLAB handles `NaN`s by setting the result of any calculation that involves `NaN`s to `NaN`. Let us look at an example:

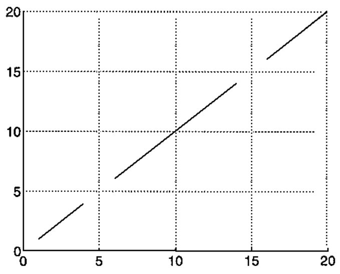
```
y = [1:4 NaN 6:14 NaN 16:20];
plot(y,'o')
grid;box
```



In everyday language we would say that the fifth and the fifteenth values of the y-vector are missing. MATLAB's graphics functions usually handle

NaNs by leaving them off the plot. For example, if we allow `plot` to try to join the points with a straight line, the values on either side of the NaNs terminate the line:

```
plot(y)
grid;box
```



If we calculate the difference between y-values, the results involving NaNs are themselves NaN:

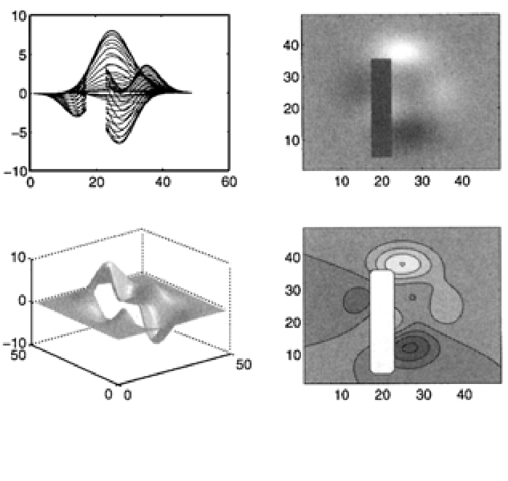
```
>> diff(y)
ans  =
  Columns 1 through 12
     1    1     1   NaN   NaN   1  1  1  1  1  1  1
  Columns 13 through 19
     1  NaN   NaN     1     1   1  1
```

If we calculate the cumulative sum of y, everything from the first NaN onwards is NaN:

```
>> cumsum(y)
ans  =
  Columns 1 through 12
     1     3     6    10   NaN   NaN   NaN   ...   NaN
  Columns 13 through 20
   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
```

MATLAB's surface plotting functions handle NaNs in a similar way:

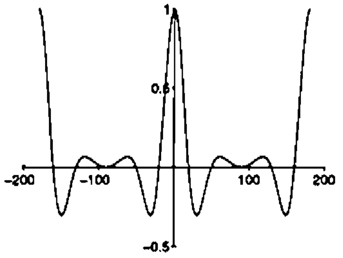
```
z = peaks;
z(5:35,18:22) = NaN;
subplot(221)
plot(z')
subplot(222)
colormap(gray(64))
imagesc(z)
axis xy
subplot(223)
surfl(z)
shading flat
subplot(224)
contourf(z)
```



16 Polar Plots

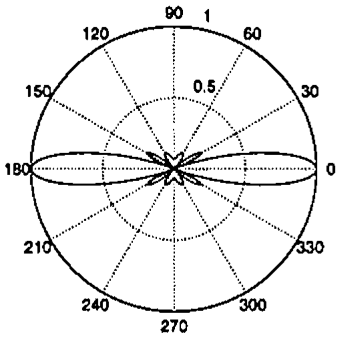
When displaying information which varies as a function of angle, it is often beneficial to use a polar diagram in which conventional (x, y) values are interpreted as angle and radius. Compare the following two displays. First the conventional (x, y) plot:

```
clf
t = linspace(-pi,pi,201);
g = sinc(2.8*sin(t));
plot(t*180/pi,g)
zeroaxes
```



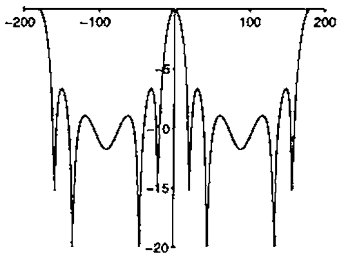
(The command `zeroaxes` is part of the companion software to this book.) Then the polar diagram indicating the directional variation in the quantity `g`:

```
clf
polar(t,g)
```



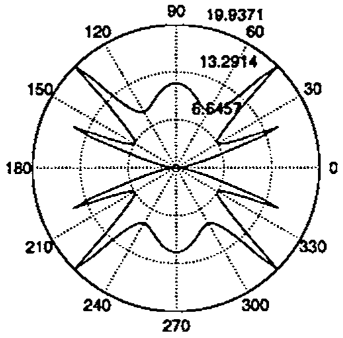
Plots such as these are sometimes displayed in decibel units:

```
gdb = 10*log10(abs(g));
plot(t*180/pi,gdb)
zeroaxes
```



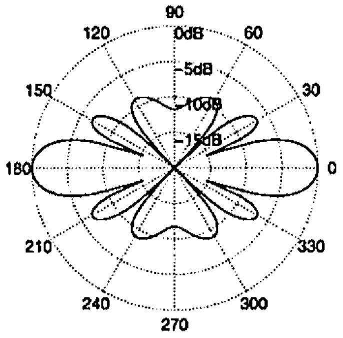
But the polar diagram in this case gives rubbish because it is interpreting the negative decibel values as negative radii:

```
clf
polar(t,gdb)
```



In this case you must use a modified version of `polar` that interprets a zero radius as a 0 dB value which should go at the outer limit of the plot. Negative decibel values should appear at smaller radii. I have implemented these ideas in the m-file `negpolar` (see companion software):

```
negpolar(t,gdb)
```



The `negpolar` function also omits the solid outer circle which, like the box drawn around MATLAB's default `plot` output, unnecessarily frames the plot and can obscure the data that you are trying to display. A faint dotted grid is enough to put the plotted points in context. I will say more about this in the section on Handle Graphics later (see page 65).

17 Fourier Transform

A theorem of mathematics says, roughly, that any function can be represented as a sum of sinusoids of different amplitudes and frequencies. The Fourier transform is the mathematical technique of finding the amplitudes and frequencies of those sinusoids. The Discrete Fourier Transform (DFT) is an algorithm that calculates the Fourier transform for numerical data. The Fast Fourier Transform is an efficient implementation of the DFT. The following functions are available in MATLAB to do Fourier transforms and related operations:

`fft`	One-dimensional fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform
`fftn`	N-dimensional fast Fourier transform
`fftshift`	Move zeroth lag to centre of transform
`ifft`	Inverse one-dimensional fast Fourier transform
`ifft2`	Inverse two-dimensional fast Fourier transform
`ifftn`	inverse N-dimensional fast Fourier transform
`abs`	Absolute value (complex magnitude)
`angle`	Phase angle
`cplxpair`	Sort complex numbers into complex conjugate pairs
`nextpow2`	Next power of two
`unwrap`	Correct phase angles

The FFT of the column vector

```
y = [2 0 1 0 2 1 1 0]';
```

is

```
>> Y = fft(y)
Y  =
   7.0000
  -0.7071+ 0.7071i
   2.0000- 1.0000i
   0.7071+ 0.7071i
   5.0000
   0.7071- 0.7071i
   2.0000+ 1.0000i
  -0.7071- 0.7071i
```

The first value of `Y` is the sum of the elements of `y`, and is the amplitude of the "zero-frequency", or constant, component of the Fourier series. Terms 2 to 4 are the (complex) amplitudes of the positive frequency Fourier components. Term 5 is the amplitude of the component at the Nyquist frequency, which is half the sampling frequency. The last three terms are the negative frequency components, which, for real signals, are complex conjugates of the positive frequency components.

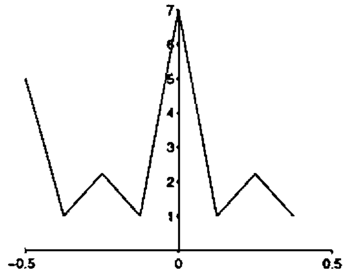
The `fftshift` function rearranges a Fourier transform so that the negative and positive frequencies lie either side of the zero frequency.

Companion M-Files Feature 4 *The function `fftfreq` gives you a two-sided frequency vector for use with `fft` and `fftshift`. For example, the frequency vector corresponding to an 8-point FFT assuming a Nyquist frequency of 0.5 is*

```
>> fftfreq(.5,8)'
ans  =
      -0.5000
      -0.3750
      -0.2500
      -0.1250
            0
       0.1250
       0.2500
       0.3750
```

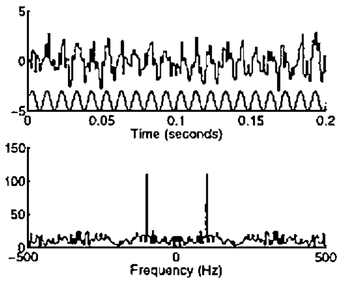
We combine `fftshift` and `fftfreq` to plot the two-sided FFT:

```
plot(fftfreq(.5,8),fftshift(abs(Y)))
axis([-.5 .5 0 7])
zeroaxes
```



Let us do a slightly more realistic example. We simulate some data recorded at a sampling frequency of 1 kHz, corresponding to a time step `dt = 1/1000` of a second. The Nyquist frequency is, therefore, 500 Hz. Suppose there is a 100 Hz sinusoid contaminated by noise. We simulate the data, calculate the FFT, and plot the results as follows:

```
dt = 1/1000;
t = dt:dt:200*dt;
sine = sin(2*pi*100*t);
y = sine + randn(size(t));
Y = fft(y);
f = fftfreq(500,length(Y));

clf
subplot(211)
stairs(t,y)
hold on
stairs(t,sine-4)
box
xlabel('Time (seconds)')

subplot(212)
stairs(f,fftshift(abs(Y)))
box
xlabel('Frequency (Hz)')
```



The top trace in the top plot is the noisy data, and the bottom trace is the original pure sinusoid. The lower plot clearly shows the frequency at 100 Hz.

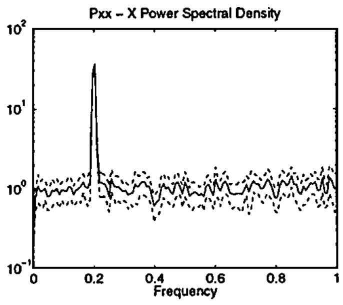
Two GUI-based FFT demos can be accessed by typing `demo` at the prompt. Select the "Signal Processing" option, then choose the "Discrete Fourier Transform" or the "Continuous Fourier Transform".

> **Exercise 5** *Extend the ideas in the previous example to two dimensions, as would be the case, for example, if you made measurements in space and time, rather than time alone. Generate a two-dimensional sinusoid and explore its FFT. (Answer on page 185.)*

18 Power Spectrum

The power spectrum (or power spectral density, or PSD) is a measure of the power contained within frequency intervals. The problem is that we only have a finite set of samples of the true signal so we can never have perfect knowledge about its power spectrum. A common way to estimate a PSD is to use the square of the FFT of the samples. The square of the FFT is called the *periodogram*. The workhorse of MATLAB's periodogram-based spectral estimation is the `spectrum` function (in the Signal Processing Toolbox). We illustrate using data similar to the previous example of a noisy sinusoid, but we take more samples. A PSD estimate can be found by typing:

```
dt = 1/1000;
t = dt:dt:8192*dt;
sine = sin(2*pi*100*t);
y = sine + randn(size(t));
clf
spectrum(y)
```

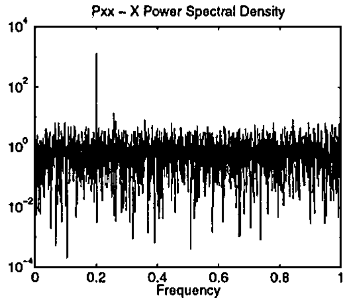


The frequency scale is normalised to the Nyquist frequency. The middle line is the PSD estimate and the two dashed lines are the 95% confidence intervals. Typing `help spectrum` reveals that there are many parameters that you can adjust when calculating the power spectrum. MATLAB's `spectrum` function uses the Welch method of PSD estimation,[6] which divides a long signal into a number of smaller blocks, calculates

[6]See Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975, p. 553. An excellent general treatment of PSD estimation is also given in William Press, Brian Flannery, Saul Teukolsky and William Vetterling, *Numerical Recipes*, Cambridge University Press, 1989.

the periodograms of the blocks, and averages the periodograms at each frequency. This is a technique commonly used to reduce the variance of the PSD. For example, we can compare the variance of the above estimate to that of a single periodogram by telling spectrum to use a block length equal to the length of the signal:

```
spectrum(y,8192)
```

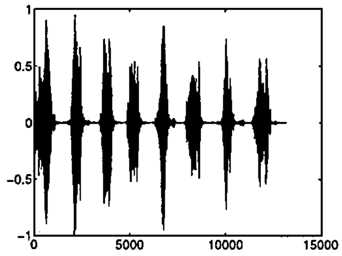


You can also specify windows to reduce spectral leakage, sampling frequencies to get correct frequency scales and overlapping blocks. If you are interested in PSD estimation, the Signal Processing toolbox contains other methods of PSD estimation including Welch's method, MUSIC, maximum entropy and multitaper. MATLAB also provides a graphical user interface for spectral estimation as part of its interactive signal processing environment `sptool`. The System Identification toolbox also contains algorithms for PSD estimation (type `iddemo` and choose option 5 for a demonstration).

19 Sounds in MATLAB

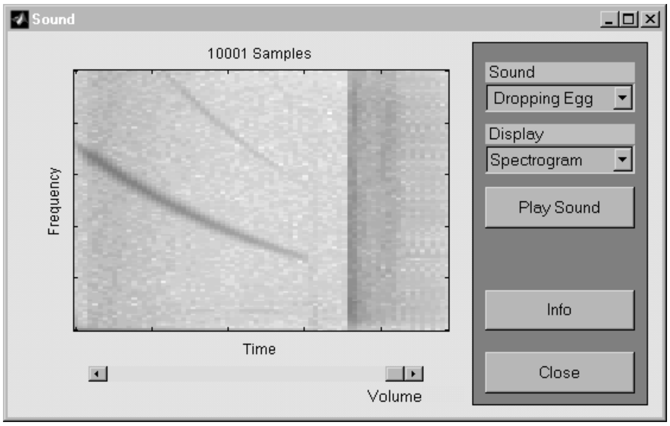
MATLAB can send data to your computer's speaker, allowing you to visually manipulate your data, and listen to it at the same time. A digitised recording of an interesting sound is contained in the mat-file `chirp.mat`. Load this data, do a plot, and listen to the sound by typing:

```
load chirp
plot(y)
sound(y)
```



The volume of the sound can be controlled from within MATLAB using the `soundsc` function and supplying an upper and lower limit. Or if you wish, you can use your computer's system software to control the volume. On UNIX the volume of the sound can be controlled with the audiotool. On a PC the volume can be controlled from the "properties" panel of the sound recorder.

You can invoke a sound demo GUI by typing `xpsound`. This GUI includes these bird chirps plus a few other sounds, three different display types, a volume slider, and a play button.



20 Time-Frequency Analysis

Signals, such as the sound data of the previous section, often consist of time series data with a time-varying frequency content. The `specgram` function allows you to analyse this kind of time-frequency data. As an example we generate a frequency modulated carrier and analyse its frequency variation with time. The `modulate` and `vco` function can be used to produce signals with many different modulation types.[7] We begin with a linear frequency sweep from 0 to 500 Hz sampled at 1 kHz. First, you must prepare a frequency control vector, which is normalised between −1 and 1, where −1 corresponds to the minimum frequency and 1 corresponds to the maximum frequency. Here we use a linear frequency control and 8192 points:

```
x = linspace(-1,1,8192);
```

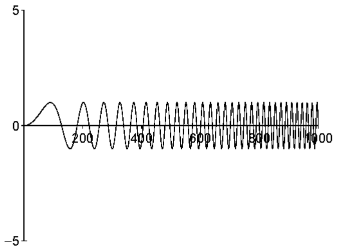
Now use the `vco` function (in the Signal Processing Toolbox) to convert this to a frequency modulated signal:

```
Fs = 1000;
y = vco(x,[0 500],Fs);
```

The input vector `[0 500]` says that our frequency sweep will go from 0 Hz to 500 Hz and the sampling frequency is `Fs = 1000` Hz. The first thousand points of this signal reveal the steady increase in frequency:

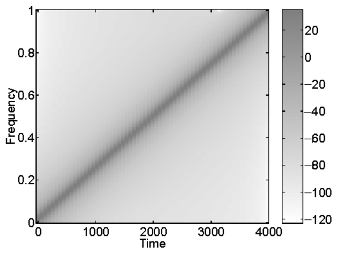
[7]In fact what we are doing here could also be done with the m-file `chirp.m` (not to be confused with the data file `chirp.mat`).

```
plot(y(1:1000))
axis([0 1000 -5 5])
zeroaxes
```



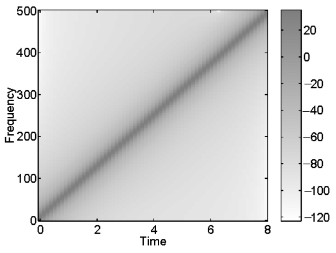
The frequency content of this signal as a function of time can be calculated using the `specgram` function. This function uses the Short Time Fourier Transform (STFT) technique. The STFT chops up the signal into a series of short segments and calculates the FFT of each segment. Each FFT becomes the estimate of the frequency content at that time. For our example we can get a quick display by typing:

```
clf
specgram(y)
colormap(flipud(gray/2+.5))
colorbar
```



The linear increase in frequency with time is clearly displayed, although here we have not told `specgram` what the sampling frequency is, so it has plotted a normalised frequency scale. If we include the sampling frequency as an input, we get the true frequencies. If you type `help specgram` you will see that the inputs are such that the sampling frequency comes third in the list, after the signal itself and the FFT size. Here we do not want to bother about specifying the FFT size, so we can just specify the empty matrix for that input and `specgram` will use its default value of `NFFT = 256`:[8]

```
specgram(y,[],Fs)
colormap(flipud(gray/2+.5))
colorbar
```



The frequency now goes from zero to 500 Hz.

[8]Many of MATLAB's functions behave this way: specifying the empty matrix will tell the function that you want to use its default value for that input.

Exercise 6 *Try a more complicated modulation function; for example, a sinusoidal rather than a linear frequency variation. Try plotting the results as a surface instead of an image. (Answer on page 186.)*

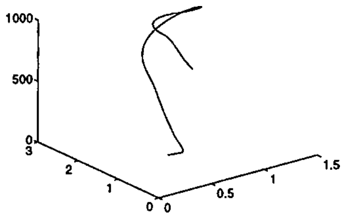
21 Line Animation

MATLAB's `comet` function can be used to produce an animation on the screen of a trajectory through either two-space or three-space. For example, we use some recorded aircraft GPS data in the file `gps.mat`.

```
>> clear
>> load gps
>> whos
  Name       Size           Bytes  Class
  t        500x1             4000  double array
  x        500x1             4000  double array
  y        500x1             4000  double array
  z        500x1             4000  double array
Grand total is 2000 elements using 16000 bytes
```

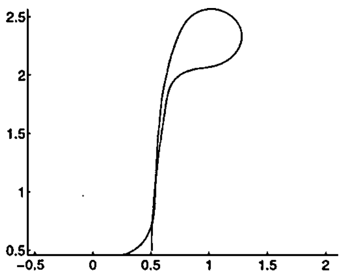
A simple 3-d plot is difficult to interpret:

```
>> plot3(x,y,z)
```



The floating thread has too few visual clues for the eye to interpret, and the altitude variation further clutters the display. A two-dimensional plot tells us that the aircraft was doing turns (but not how high it was):

```
plot(x,y)
axis equal
box
```



This is an improvement, but we still do not know where the aircraft started, where it finished, and how it went in between. We can see an animation of the trajectory by typing:

```
comet(x,y)
```

(You can get a three-dimensional version by using `comet3`.) You can see it on your screen. But we have just illustrated a disadvantage of such a display: you have to be there. I cannot communicate to you what it looks like on paper. For that you need to resort to, say, an array of two-dimensional plots strung out along the third time dimension. This gets us into the subject of plot arrays, which is discussed in Section 32.3 on page 123.

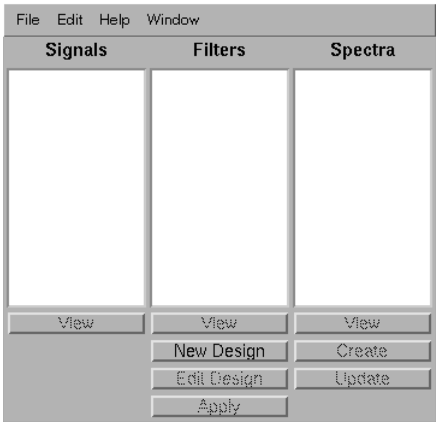
22 SPTool

SPTool (in the Signal Processing Toolbox) is a graphical user interface to many of MATLAB's signal processing functions. The idea is to import signals from the MATLAB workspace into the SPTool environment where they can be manipulated in a great variety of ways. As an example, load some data into your workspace by typing:

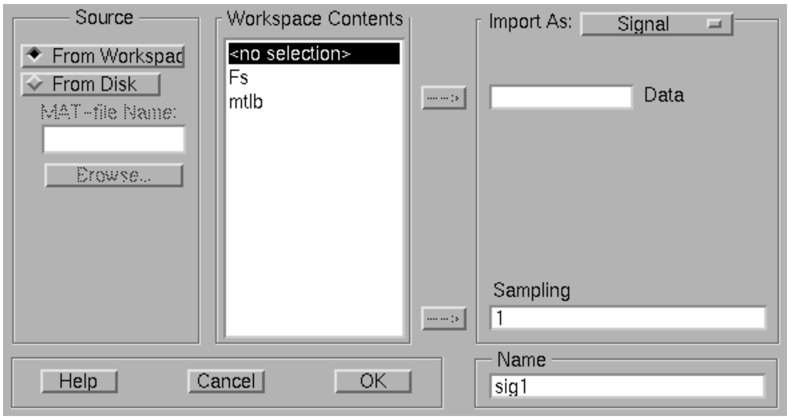
```
load mtlb
```

We will use SPTool to look at this time-series data and calculate various power spectra. Invoke SPTool by typing:

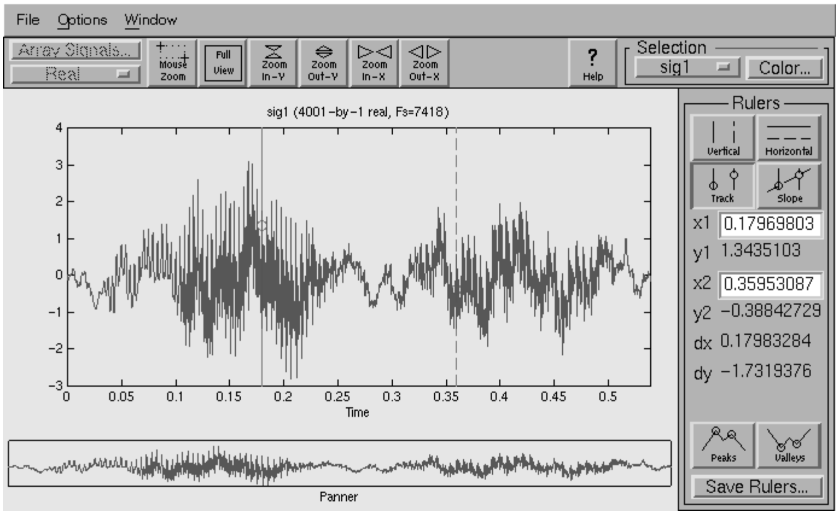
```
sptool
```



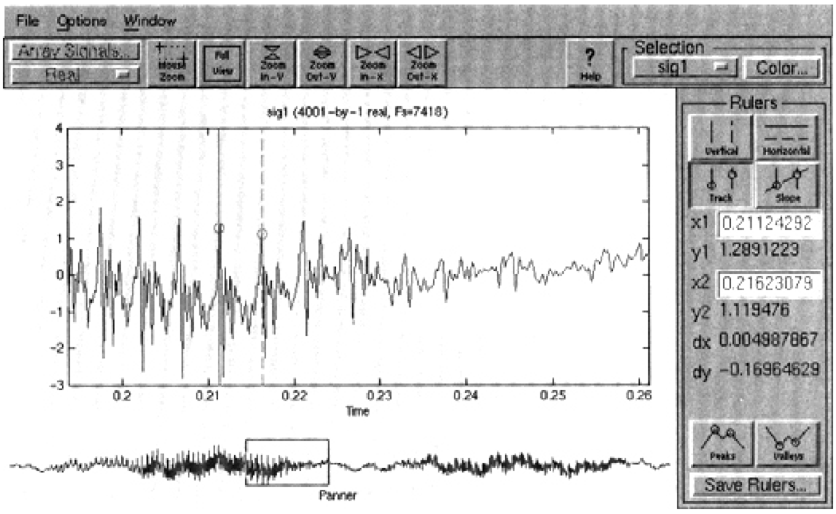
Choose the `File`→`Import` menu item to open the import panel, which allows you to control the variables that `sptool` can "see":



Click on the variable `mtlb` and the arrow button (`-->`) to get `mtlb` to appear in the `Data` box (or just type `mtlb` there). Do the same to make `Fs` appear in the `Sampling` box. Then press `OK`. A signal called `sig1` appears in the `Signals` box in the main `SPTool` panel. Clicking on the `View` button at the bottom of the `Signals` box opens the signal browser panel:



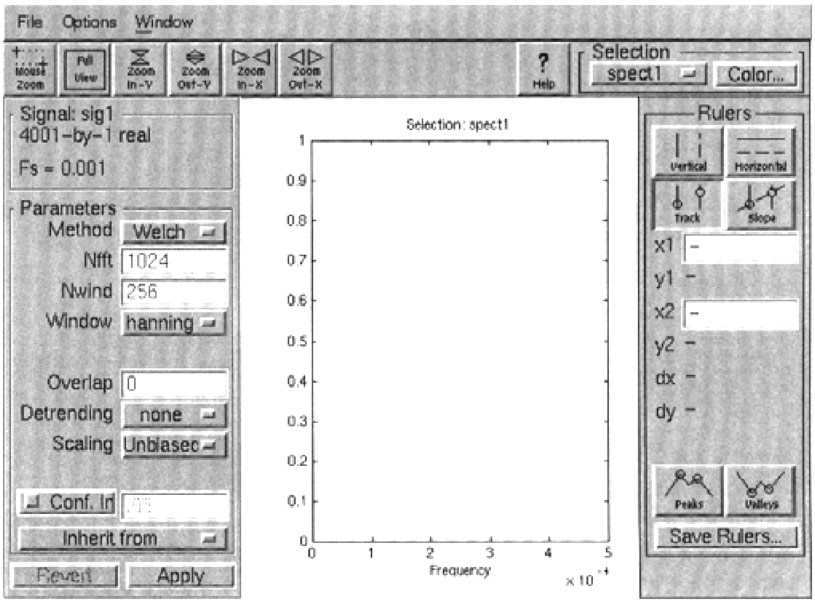
Here you have a plot of the time series with two "rulers". The rulers can be used to pick values out of the data, as well as to calculate intervals and slopes. The data in the `Rulers` box at the right of the display shows this information. At the bottom is a "panner". If you click on the `Zoom In-X` button a couple of times, the top plot shows an expanded portion of the data, and the panner at the bottom shows the location of the top box within the entire signal.



x1 0.21124292
y1 1.2891223
x2 0.21623079
y2 1.119476
dx 0.004987867
dy -0.16964629

By clicking within the panner box and dragging, you can change the location of the zoomed window. You can listen to this time series by selecting `Options`→`Play`.

To calculate the power spectrum of this signal, go back to the main `SPTool` panel and click the `Create` button at the bottom of the `Spectra` box. Doing this will open the Spectrum Viewer:

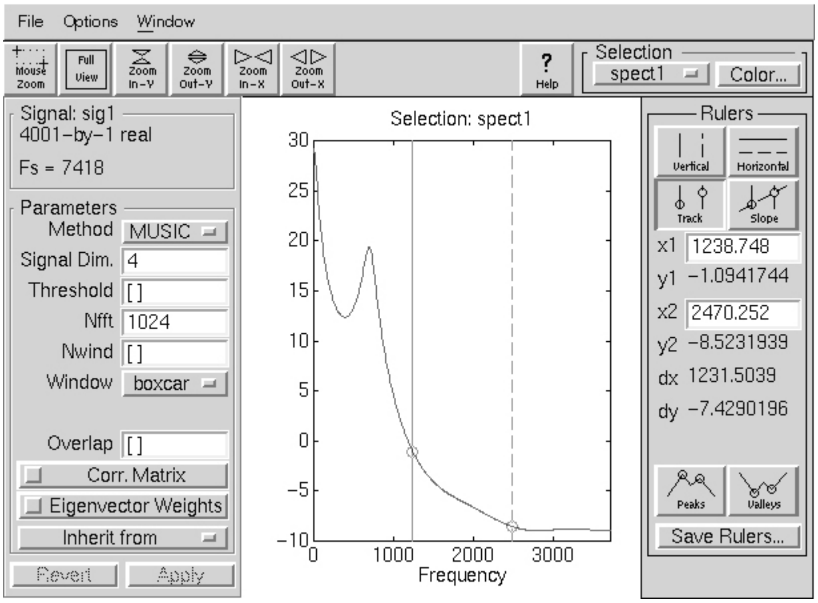


Signal: sig1
4001-by-1 real
Fs = 0.001

Parameters
Method Welch
Nfft 1024
Nwind 256
Window hanning
Overlap 0
Detrending none
Scaling Unbiased

x1 -
y1 -
x2 -
y2 -
dx -
dy -

Choose a method with the parameters you like to get a plot of a spectral estimate:



Signal: sig1
4001-by-1 real
Fs = 7418

Parameters
Method MUSIC
Signal Dim. 4
Threshold []
Nfft 1024
Nwind []
Window boxcar
Overlap []

Selection spect1

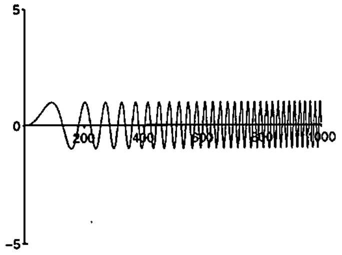
x1 1238.748
y1 −1.0941744
x2 2470.252
y2 −8.5231939
dx 1231.5039
dy −7.4290196

You can design and apply filters to data in a similar way.

23 Handle Graphics

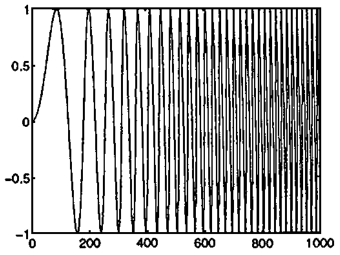
So far in this book, we have only used MATLAB's high-level plotting function (`plot`, `surf`, etc.). High-level plotting functions produce simple graphs and automate the many mundane decisions you might make in producing a plot, such as the position of the plot, the colour of the axes, the font size, the line thickness, and so on. MATLAB's system of *Handle Graphics* allows you to control a great many of these "mundane" aspects of plotting, to produce plots that are optimised for communicating the data at hand. The idea behind Handle Graphics is that every object in the figure window (axes, lines, text, surfaces, etc.) has a set of properties. These properties can be examined using the `get` command and set to new values using the `set` command. Every object in the figure window also has a unique identifier (a number) called a *handle*. The object's handle tells `get` and `set` what object you are interested in. As an introductory example, consider the plot shown on page 58 of the frequency modulated sinusoid:

```
x = linspace(-1,1,8192);
Fs = 1000;
y = vco(x,[0 500],Fs);
plot(y(1:1000))
axis([0 1000 -5 5])
zeroaxes
```



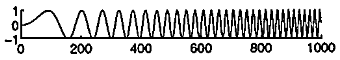
We used the axis command to set the y-axis limits to [-5 5] instead of the default limits, in this case, of [-1 1]

```
clf
plot(y(1:1000))
```



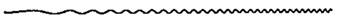
which makes the variation in frequency slightly less apparent, and is just too grandiose. The eye can pick up very subtle variations in line straightness, but here the variation is so huge that the lines become parallel and begin to produce the optical illusion of vibration. Also, lines that are very nearly vertical or horizontal begin to be affected by the finite resolution of dot printers. Using Handle Graphics we can achieve a more elegant result by reducing the height of the y-axis. We do this by setting the position property of the current axes:

```
set(gca,'Position',[.1 .5 .8 .1],'box','off')
```



The gca input is itself a function, which returns the handle to the current set of axes. We are saying that we want to set the position of the current axes to be equal to the vector [.1 .1 .8 .1]. The position vector has the form [*left, bottom, width, height*], in units normalised to the figure window; $(0, 0)$ is the bottom left and $(1, 1)$ is the top right. But perhaps we should shrink it even further, and dispense with the ever-present axes:

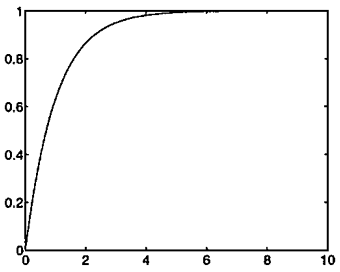
```
set(gca,'Position',[.1 .5 .8 .01],'visible','off')
```



23.1 Custom Plotting Functions

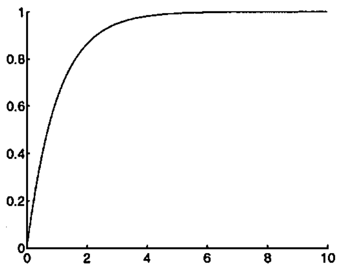
Handle Graphics can be used to write your own graphics m-files that are fine-tuned to your requirements. For example, the box around the graph produced by the default `plot` command can obscure the data:

```
clf
t = linspace(0,10);
y = 1 - exp(-t);
plot(t,y)
```



To avoid this problem (which I have found occurs frequently), I use my own personal version of the `plot` command, called `plt`, which omits the box:

```
plt(t,y)
```



The m-file for `plt` (see companion software) simply passes all the input parameters directly to the `plot` command and then sets the `'box'` property of the current plot to `'off'`.

23.2 `Set and Get`

Typing

```
get(H)
```

where `H` is an object handle, displays all of the property names associated with the object. Typing

```
set(H)
```

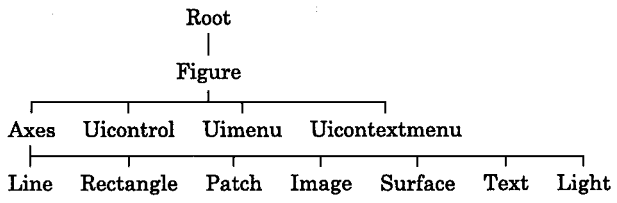
displays all of the possible values that can be taken by every property associated with the object. Typing

set(H,'*Property*')

displays all of the possible values for the ***Property*** associated with the object.

23.3 Graphical Object Hierarchy

MATLAB graphical objects are arranged according to the hierarchy shown here.



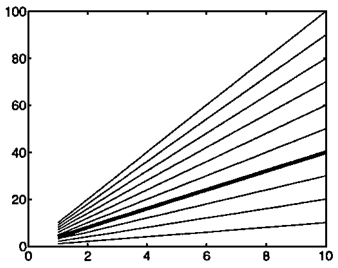
The object immediately above another is called a *parent*, and the objects below are called *children*. In general, children inherit their handle graphics properties from their parent. For example the position of a line on a plot depends on the position of the axes that it goes in, which, in turn, depends on the position of the figure window.

The Root object is the computer screen. There can only be one Root object. You can see the properties of the Root object and the allowable options by typing `set(0)` (the handle of the Root is always equal to zero).

The Uicontrol, Uimenu, and Uicontextmenu objects are graphical user interface elements that are discussed in Part II of this book (see page 133).

A parent can have any number of children. For example the Root can have many Figures, a Figure can have many Axes, and a set of Axes can have many Lines, Surfaces, and so on. If a parent has many children, one of them is designated the *current* one. For example the current set of axes is the one that will be updated the next time you do a `line` command. You can make an object current by clicking on it with the mouse. For example, I clicked on the fourth line from the bottom before setting its `linewidth` property to 5 (the default linewidth is 0.5):

```
plot([1:10]'*[1:10])
set(gco,'linewidth',5)
```



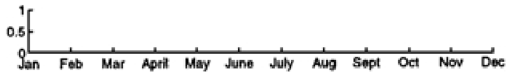
The following functions return the handles of current objects:

`gcf` Get Current Figure
`gca` Get Current Axes
`gco` Get Current Object

The handle of a Figure is the number (1, 2, etc.) that normally appears in the Figure's title bar (supplied by the windowing system).

All of the graphical objects, except the Root object, have low-level creation functions in which you can specify their properties. For example, here is how to create a set of axes with the x-axis tick marks labelled by months of the year:

```
lbls = ['Jan|Feb|Mar|April|May|June|'...
        'July|Aug|Sept|Oct|Nov|Dec'];
clf
axes('position',[.1 .5 .8 .1],'xlim',[1 12],...
     'xtick',1:12,'xticklabel',lbls)
```



The general format of object creation functions is

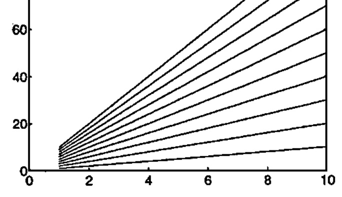
```
handle = function('propertyname','propertyvalue')
```

The output of the function is the handle of the object. This handle can then be used in subsequent calls to `get` and `set` to modify the properties of the object. The *`propertyname`*s are displayed by MATLAB with capitalisation to make them easier to read; for example, the `VerticalAlignment` text property or the `YAxisLocation` axes property. When you are typing property names, you do not need to use the full name or any capitalisation; you need only use enough letters of the property name to uniquely specify it, and MATLAB does not care what capitalisation you use. Nevertheless, when writing m-files, it is a good idea to use the full property name because abbreviated names may no longer be unique if extra properties are added in future releases of MATLAB.

Example: Line Width

The default way to plot a matrix is to draw one line for each column of the matrix, with the lines differentiated by colour. Suppose instead that we want to differentiate the lines by their thicknesses. One way to do it is as follows. First generate the data and plot it:

```
y = [1:10]'*[1:10];
clf
plot(y)
```



Now we need to get the handles of all the lines. We could have said `h = plot(y)` to get them, but for now we use the `get` function:

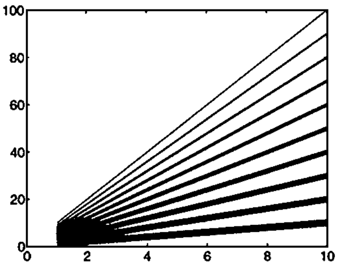
```
h = get(gca,'children')
```

The `gca` function returns the handle of the current axes, and `get(gca,'children')` returns the handles of all the current axes' children (the lines on the plot). Now we want to change the thicknesses of the lines. We set up a vector of line widths with as many elements as there are lines:

```
widths = linspace(.1,10,length(h));
```

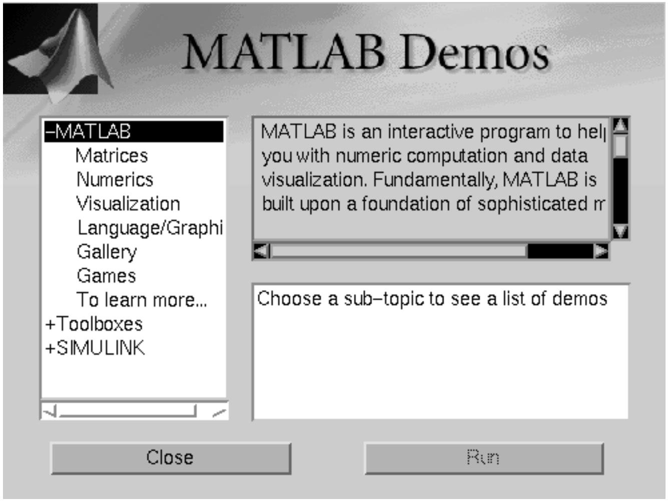
The widths of the lines will vary from a minimum of 0.1 to a maximum of 10. We use a `for`-loop to change the width of each of the lines:

```
for i = 1:10
  set(h(i),'linewidth',widths(i));
end
```



24 Demos

The MATLAB demos are well worth browsing. You can learn about a subject (often reading references are given), as well as learning about MATLAB's capabilities. Of interest to sonar and radar signal processors is MATLAB's Higher Order Spectral Analysis toolbox containing, for example, functions for direction of arrival estimation (beamforming plus other methods), time-frequency distributions, and harmonic estimation. Type `help hosa` for a list of functions in the Higher Order Spectral Analysis toolbox. Browsing the demos or doing a keyword search may save you from writing your own MATLAB code and re-inventing the wheel. Type `demo` to get the panel:



MATLAB is an interactive program to hel
you with numeric computation and data
visualization. Fundamentally, MATLAB is
built upon a foundation of sophisticated m

Choose a sub–topic to see a list of demos