MATLAB Introduction

MATLAB™ is an interactive program that can be used to calculate various scientific and engineering functions, the results of which can often be plotted as graphs. In fact, there is a family of MATLAB programs, including various toolboxes. For instance, the Control Systems Toolbox allows the user to design and test control systems. The Symbolic Maths Toolbox allows MATLAB to call Maple, a program that allows mathematical expressions to be processed symbolically.

MATLAB stands for 'Matrix Laboratory', and so it provides facilities for handling matrices, which are elements which can contain many data values. However, a special case of a matrix, namely one with only one element, is a scalar value, which can be an ordinary number such as 2 or -4.5, or it can be a complex number. Hence MATLAB can be used to process ordinary numbers, complex numbers as well as matrices.

The program allows the user to interactively enter simple commands, which the program obeys. It also allows the user to write their own programs, which MATLAB will obey. In effect, therefore, the user can define extra commands for MATLAB. User programs or commands are called scripts or M-files.

When MATLAB is invoked few help messages are shown followed by the prompt

```
>>
```

The user can type in commands after the prompt. In the following, commands are illustrated by being shown after the >> prompt.

From the MATLAB environment it is possible to invoke an editor which allows the user to type in a script or a function in an M-file. Once such a file has been written, the user can invoke that command by typing something after the prompt.

Help

MATLAB provides help facilities. For instance if at the prompt you type

```
>>help
```

Then a variety of topics are listed about which help is available. One in the list is:

```
MATLAB\elmat    - Elementary matrices and matrix manipulation.
```

This is saying that details about elementary matrices are available in the MATLAB directory under the title elmat. To find out about these details you thus type

```
>>help elmat
```

If the Control Systems toolbox is installed, the original help command will also list

```
toolbox\control    - Control System Toolbox.
```

To find out more details about the functions available here, you type

```
>>help control
```

Demonstration

MATLAB also comes with various helpful demonstrations.

```
>> demo
```

invokes the demonstrations which give examples of what MATLAB can do.

Simple commands

The basic command in MATLAB is of the form:

```
>> variable = expression
```

In this, the expression is evaluated and the result is assigned to the variable.

For instance,

```
>> A = 2
```

This assigns 2 to the variable A. If this command is invoked, the system will obey this command and print the result. That is, the screen will contain:

```
A =
    2
```

If the user does not wish to see the result displayed, the command should be followed by a semicolon. Thus:

```
>> A = 2.5;
```

assigns 2.5 to the value A, but does not show the result.

If the user wishes to see the contents of A, then the following command may be used:

```
>> A
```

The above command is a special case of the command variable = expression. Here the variable is not specified, so the expression is evaluated but not stored anywhere.

Expressions can contain more complicated values:

```
>> 2+5
```

will cause the expression 2 + 5 to be found. Standard operators such as +, -,* and / can be used, though there are some restrictions when these are applied to matrices. The ^ operator means raise to a power.

```
>>5^2
```

will calculate 5 squared = 25.

Functions are also provided, for instance

```
>> sin(5)
```

will calculate the sin of 5. Note 5 is in radians. To find the sin of 45 degrees we use:

```
>> sin (45*pi/180)
```

Note, the constant pi is defined in the language. Also defined are i and j for $\sqrt{-1}$ and Inf for infinity.

Note also that MATLAB is case-sensitive. It understands sin, but not Sin. If you type:

```
>> Sin(4)
```

you get the error:

```
??? Sin(
        |
    Missing operator, comma, or semi-colon.
```

Variables

Variables can be integers, floating point numbers, complex numbers and matrices. Names can be assigned to variables, as shown by the use of the variable A earlier. Variable names can comprise many letters. Note that names are case sensitive.

Integers are whole numbers, positive or negative, such as 1, 45, -6456.

Floating point numbers can have fractional parts, such 7.071 or -12.678.

Complex numbers are ones that include the square root of -1. This is denoted by i or j. When MATLAB prints out complex numbers it used i for √-1. For example, the complex number 3.4 + 5.7j can be entered by:

```
>> x = 3.4 + 5.7*j
```

Matrices are series of values, each value could be an integer, floating point number or a complex number. A simple matrix consists of one row of numbers, separated by spaces or commas:

```
>> x = [1 2 3]
```

This prints out as

```
x =
     1 2 3
```

If the numbers are entered with semicolons, the matrix contains one column of numbers

```
>> x = [3;8;2]
```

This is printed as:

```
x =
     3
     8
     2
```

Individual elements of a matrix can be accessed. For instance, in the above,

x(1) is 3; x(2) is 8 and x(3) is 2.

Part of a matrix can be accessed, for instance if x is [1 5 7 3 9 0],

```
>> x(2:4)
```

will return a matrix containing elements 2, 3 and 4 of the matrix x, that is

```
   5 7 3
```

A two dimensional matrix is entered as follows, each row being separated by semicolons, with spaces between elements in each row:

```
>> x = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

This prints out as

```
x =
    1    2    3    4
    5    6    7    8
    9   10   11   12
```

Note, each row should have the same number of elements. In the above there are three rows, each of which has four elements. The matrix has four columns. The size of matrix can be found, for instance

```
>> size(x)
```

returns

```
3 4
```

This indicates that x has 3 rows and 4 columns. If you want to store the size of the matrix in variables, type:

```
>> [ r c ]  = size(x)
```

The size command returns a matrix result, so by saying [r c] one is saying that the result of size is put in a matrix whose two elements are labelled r and c. As a result, this stores the number of rows in variable r and the number of columns in variable c.

It is sometimes required that a matrix is needed containing, say, all the integers from 1 to 10. This can be achieved easily by:

```
>> x = [1:10]
```

The result of this is:

```
x =
     1    2    3    4    5    6    7    8    9   10
```

If three values are specified, separated by colons, all numbers are generated between the first and last number where the difference between each value is set by the second number. For instance,

```
>> x = [0:0.2:2]
```

Will result in:

```
x =
     0  0.2  0.4  0.6  0.8  1.0  1.2  1.4  1.6  1.8.  2.0
```

To get a matrix containing 11 equally spaced values between 0 and pi, one can say:

```
>> x = pi * [0:0.1:10]
```

This calculates a matrix containing 11 values between 0 and 1, namely 0, 0.1, 0.2 up to 1, and then multiplies each element by pi.

This demonstrates the general rule that a scalar k times a matrix generates a matrix of the same size, where each element in the matrix is k times the corresponding element in the original matrix. For instance,

```
>>3 * [1 2 3]
```

Generates

```
3 6  9
```

One can find the minimum or maximum values in a matrix, for instance

```
>> max [1 5 3 8 2]
```

returns the maximum value, in this case 8, in the array. The function min also exists.

Operators

The operators +, -, *, / and ^ are some of the operators available. For more details see

>>help ?	This gives an overview of all operators
>>help arith	This describes many of the arithmetic operators
>>help slash	This describes the division operators
>>help relop	This describes the relational and logical operators

Complex Numbers

+, -, * and ^ work with complex numbers as one would expect. The division operator can also be used, though this works by using the complex conjugate. For instance, for the given value of x above:

$$1/x = \frac{3 - j4}{3^2 + 4^2} = 0.12 - j0.16$$

Matrix Operators

The standard operators can be used with two matrices, though these matrices must be of the right size. If they are not then a suitable error message is generated.

A + B	is legal if A and B have the same size; the result is also of that size Each element in the result is the sum of the corresponding elements in A and B
A – B	is similar except the result is the difference between A and B
A * B	is legal if the number of columns in A equals the number of rows in B The result has the same number of rows as A and the same number of columns as B
A / B	is achieved by something similar to A times the inverse of B, though A and B should be of the same size and the inverse must exist.

See the help command on slash for more details about division.

The relational operators also work, for instance

A > B	is a matrix of the same size with all 1s if A>B and all 0s otherwise.

There are other operators, for instance

A .* B	which requires A and B to be the same size Each element in the answer is the product of the corresponding elements in A and B.
A . /B	is like A.* B except each element in A is divide that in B.
A.^3	produces a matrix where each element is the cube of the corresponding element in A.
A’	is the transpose of A.
A(r,:)	returns the r’th row of A
A(:,c)	returns the c’th column of A

Common Functions

The following lists some of the basic functions. A full list is available by typing

>>help elfun

Function	***This returns:***
abs(x)	the absolute value of x (the size of the number)
sign(x)	the sign of x
rem(x,y)	the remainder of x/y
sqrt(x)	the square root of x
round(x)	the number x to the nearest integer
ceil(x)	x rounded up to the next integer
fix(x)	x rounded down to the next integer
cos(x)	the cosine of x, x in radians
sin(x)	the sine of x, x in radians
tan(x)	the tangent of x, x in radians
acos(x)	the angle a such that cos(a) = x
asin(x)	the angle a such that sin(a) = x
atan(x)	the angle a such that tan(a) = x
log(x)	the natural log of x, this function sometimes called ln
exp(x)	returns e raised to the power of ; sometimes written e^x
log10(x)	returns the common base 10 log

To find out more details on each function, you can type, for instance,

>>help atan

These functions can be used on matrices. For instance:

>> x = pi*[0:0.1:2];

>> s = sin(x);

The first line generates an array of 21 equally spaced numbers between 0 and 2*pi. The second command generates a matrix s, with the same number of elements as x, and each value in s is the sin of the corresponding value in x. For instance,

x(1) is 0 so s(1) is sin(0);

x(2) is 0.1, so s(2) is sin(0.1*pi);

x(3) is 0.2, so s(3) is sin (0.2*pi)

The functions for processing complex numbers are shown below, where the Result column shows the value generated if x = 3+j*4

Comma	***This returns the***	***Result***
real(x)	real part of a complex number	3
imag(x)	imaginary part of a complex number	4
abs(x)	magnitude of the complex number	$\sqrt{(3^2+4^2)}=5$
angle(x)	angle of a complex number x	$\tan^{-1}(4/3)=0.9273$
conj(x)	complex conjugate of the complex number x	3-j*4

A number of matrix functions are also available. More details about these and other functions are available in:

>> help elmat

>> help matfun

Some of the common functions are as follows

Function	***Action***
inv(A)	returns the inverse of A
det(A)	returns the determinant of A
eig(A)	returns the eigenvalues of A
rank(A)	returns the rank of A
linsolve(A,b)	returns x where A x = b, using Gaussian Elimination
eye(n)	returns a n*n identity matrix
zeros (n)	returns a n*n zero matrix

Matrices as polynomials

A vector containing a series of values can be used to represent a polynomial. For instance, [10 7 1] can represent

$10x^2 + 7x + 1$

Note, the first element in the polynomial is the coefficient of the highest order term in the polynomial.

A polynomial can be evaluated, for instance

>>polyval([10 7 1], 3)

calculates

$10*3^2 + 7*3 + 1 = 922$

Function polyal can be used with complex numbers, as may be required in frequency response analysis.

>>polyval ([10 7 1], j*2)

Evaluates

$10*(j2)^2 + 7*j2 + 1 = -39+14j$

Suppose there are two polynomials, a = [10 7 1] and b = [5 1].

These cannot be added together directly, as they are vectors of different sizes. However, it is possible to add them by

>>[0 b] + a

[0 b] generates the vector comprising 0 followed by b. The result is the same size as a, so it is possible to add them.

Multiplication of polynomials is possible.

>> conv(a,b)

Generates the vector equivalent to the polynomial found by multiplying

$(10x^2 + 7x + 1)(5x+1) = 50x^3 + 45x^2 + 12x + 1$

The result is thus

[50 45 12 1]

The roots of a polynomial equation can be found as follows

>>roots([10 7 1])

Will return –0.5 and –0.2, given that $10x^2 + 7x + 1=(2x+1)(5x+1)$

MATLAB can find polynomials to fit data. For instance suppose

x = [1 2 3 4 5] and y = [4 13 26 43 64]

The command to fit the best second order polynomial to these x,y data is

>> polyfit(x,y,2)

The answer is:

2.0000 3.0000 -1.0000

This is correct, as $y = 2x^2+3x-1$

A system might be represented by two polynomials, for instance:

$$\frac{s+2}{(s+1)(s+3)} = \frac{s+2}{s^2+4s+3}$$

This could be represented by two polynomials [1 2] and [1 4 3]. The transfer function can be printed by

>>printsys([2 1], [1 4 3]

Simplification of polynomials is also available. For the values of a and b given earlier,

>>printsys(a,b)

would print

$$\frac{5s+1}{10s^2+7s+1}$$

But the denominator factorises as (5s+1)(2s+1), so the above simplifies to

$$\frac{1}{2s+1}$$

There is a command to do this:

>>[n d] = minreal(a,b);

This simplifies the numerator and denominator polynomials, cancelling any root that exists in both the numerator and denominator. It also ensures that the first term in the denominator is 1 (by dividing all terms by the first value in the denominator). n and d =

[0.5] and [1 0.5]

so that printsys(n,d) would output

$$\frac{0.5}{s+0.5}$$

Data file load and save

MATLAB has some simple facilities for loading and saving data.

For instance, suppose x = [1 2 3 4 5] and y = [4 13 26 43 64].

These can be saved to a file called xydata.dat by:

```
>> save xydata.dat x y -ascii
```

This stores the data x and y into a file containing ascii data (as opposed to binary data).

Your can check the data is there by typing:

```
>>type xydata.dat
```

This generates a file with two rows of five numbers, being the data for x then that for y.

To load data you type

```
>>load xydata.dat
```

and this will generate a 2D matrix called xydata. The first row will have the data which was in x, the second row will have the data which was in y. You can then access elements from the matrix variable xydata. Not xydata does not known about the variables x and y; it is just a matrix with data.

Vectors

Matrices can represent vectors, for instance [1 2] represents a 2D vector, [1 5 8] represents a 3D vector. Functions are available for vector products:

```
>>dot(x,y)       performs the vector dot product of 2 equal sized vectors
>>cross(x,y)     performs the vector cross product, where x and y have 3 elements.
```

The result of the former is a scalar, that of the latter is a three element vector.

The above can be extended to handle many vectors. For instance,

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

a and b are two 3 element column vectors, which can be processed by dot and cross.

```
>>dot (a,b)
```

produces the result

```
    5736
```

The first is the result of the dot products of the first column in a and b, the second is the dot product of the second column in a and b.

```
>>cross(a,b)
```

produces the result

```
    17    6
     21   36
    -16  -26
```

The first column is the cross product of the first columns in a and b; the second is the cross product of the second columns of a and b. Note a and b must have 3 rows.

Graph Plotting

MATLAB has quite a powerful set of graph plotting routines, for both 2D and 3D graphs. The following two commands illustrate how a simple graph could be plotted.

```
>>t=pi*[0:0.1:2];
>>plot(t,sin(t));
```

The first command fills the array with values at suitable intervals. The second sends to the plot command two arrays, the first has the values for co-ordinates in the x direction, the second the values in the y direction. These values will be plotted on a graph, suitably scaled, with axes marked with appropriate scales. Information can be added:

```
>>title('Sine graph');  adds the string 'Sine graph' as a title to the graph
>>xlabel('t');          labels the x axis as 't'
>>ylabel('func');       labels the y axis as 'func'
```

Two graphs can be plotted superimposed. For instance,

```
>> plot(t,sin(t),t,cos(t));
```

Plots the graph of sin(t) against t, and the graph of cos(t) against t. Note there must be pairs of x and y co-ordinates for each graph. Three graphs can be plotted by including three pairs of co-ordinates, etc.

MATLAB chooses suitable colours and ways of representing the data on the graphs. By default, graphs are series of lines joined up, and different colours are used for each plot. These can be overriden, for instance,

```
>>plot(t,sin(t),'w-',t,cos(t),'y*');
```

Specifies that the sine graph should be lines in white and the cosine graph should be a series of yellow asterixes. A full list of the available colours and means of representing the lines is available by typing

```
>>help plot
```

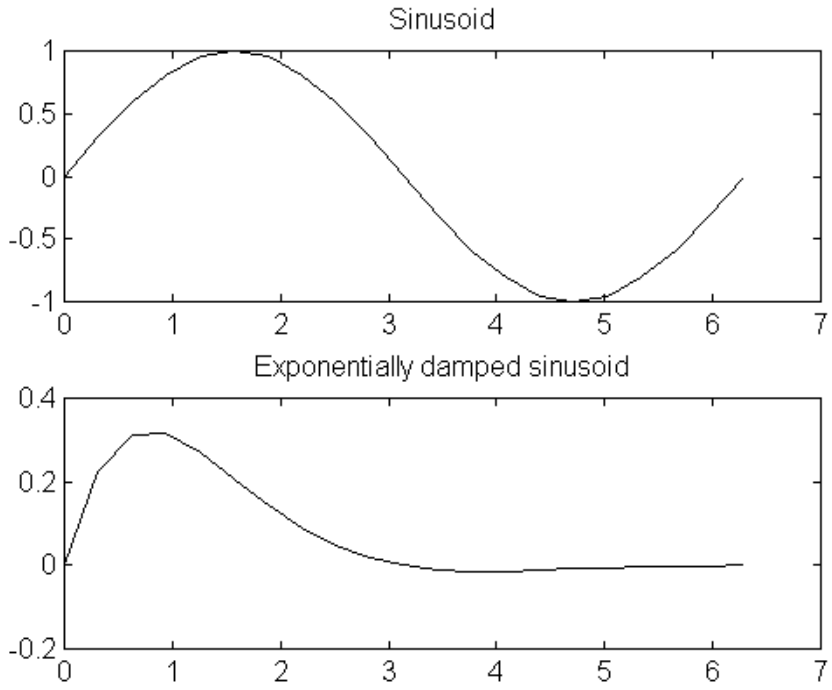
The subplot function allows two or more graphs to be drawn on separate axes.

```
>>subplot(r,c,g);
```

specifies that there will be r*c graphs, arranged in r rows and c columns, and that the next graphics command will apply to graph number g. The following shows how two graphs are shown, one of sin(t) at the top, and one of exp(-t)*sin(t) on the bottom.

```
>>t=pi*[0:0.1:2];
>>subplot(2,1,1);
>>plot(t,sin(t));
>>title('Sinusoid');
>>subplot(2,1,2);
>>plot(t,exp(-t).*sin(t));
>>title('Exponentially damped sinusoid');
```

The first call to plot and to title apply to the top graph region. The second call to plot and title apply to the bottom graph region. Note the use of the '.*' operator to calculate exp(-t)*sin(t). The figure below shows the result produced.



The function clf clears the graphics window and sets it to having region for one graph.

More details about 2D graphs are available on

>>help plotxy

This help message includes the routines semilogx which plots graphs where the x axis is scaled logarithmically; semilogy, where the y axis is scaled logarithmically and loglog where the graph is plotted where both axes are scaled logarithmically. It is also possible to plot bargraphs, histograms, etc. Details about each specific function are available, for instance, to learn more about loglog, you type

>>help loglog

The command grid can be used to draw a grid on the graph.

When showing two graphs on the same axes it is convenient to write a label for each. The command text (x, y, str) writes the string str at position x,y.

3D plotting

Three dimensional graphs are possible, more details being available using

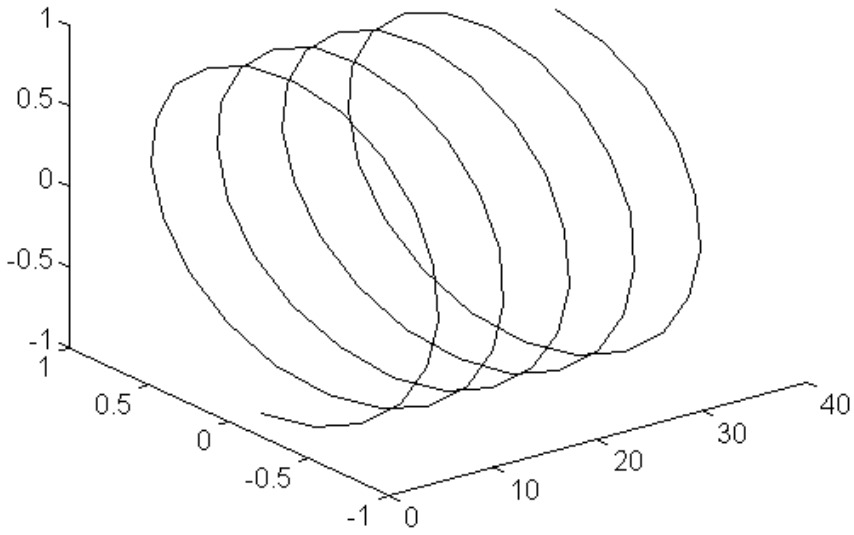
>>plotxyz

The simplest plot function for three dimensions is plot3(x,yz) where x, y and z are matrices cntaining data in the x, y and z directions. For instance, consider

>>t=pi*[1:0.1:10];

>>plot3(t,sin(t),cos(t))

These cause a helix to be plotted, as is shown in the figure below. Note, the x, y and z axes can be labelled by the commands xlabel, ylabel and zlabel.



A three dimensional plot of x,y against some function of (x,y), can be achieved as follows. x should have m values in a suitable range, y should have n values in a suitable range, and z should be an array containing m*n values, each value being f(x,y). In the example below, x and y both contain values in the range -2 to 2. To plot the 3D graph:

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

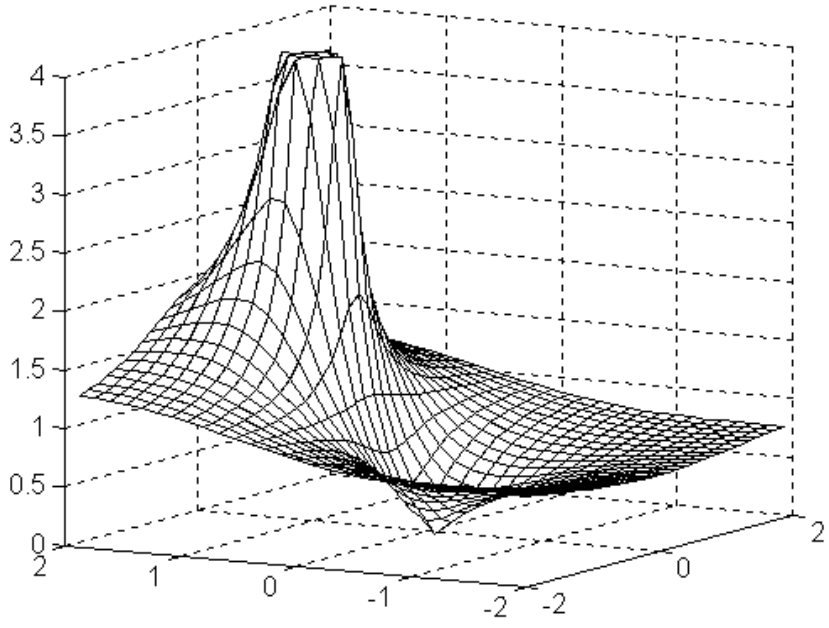
The colour of the data plotted is determined by the range of values in z. If just a black white graph is needed, the colormap must be set to black. There is no command 'black', but there is one called 'white' which produces a vector of '1's. Thus black is 1-white: so

```
>> colormap(1-white)
```

The position from which the graph is viewed is set by

```
>>view(az, el)
```

where az is the azimuth rotation and el is the vertical elevation, both in degrees. In the figure below, az = -60 and el = 10.



Scripts and M files

As we have seen, MATLAB allows the user to type in commands interactively. It also allows the user to prepare a set of commands in an editor, save those commands in a file, called <name>.m, and then those commands can be entered by typing

>><name>

where <name> is the name of the file.

For instance, the following is the file sploteg.m which was used to generate the earlier example of two graphs.

```
% Program to show how two graphs can be plotted
% One graph is of sin(t)
% Below which is graph of exp(-t)*sin(t)
% Dr Richard Mitchell
t=pi*[0:0.1:2];                          % set range of time
subplot(2,1,1);                          % set to plot in the top of two graph areas
plot(t,sin(t));                        % plot sin graph
title('Sinusoid');                       % add title to it
subplot(2,1,2);                          % set to plot in lower area
plot(t,exp(-t).*sin(t));                 % plot exp(-t)*sin(t)
title('Exponentially damped sinusoid'); % add title to second graph
```

This is run by typing

>>sploteg

Note anything after % is a comment. The comments at the start of the file form a header. This header is printed if you type

>>help sploteg

Note, the rule about semicolons at the end of a command applies in these files. Thus, if you don't want the result of each operation printed on the screen, put a semicolon at the end of each command (as is done in the above example).

Commands if, while and for

One can make these files quite powerful using the if, while and for commands. The first allows conditional operation. The syntax of the if statement is:

```
if condition1
    statement1
 elseif condition2
    statement2
else
    statement3
end
```

If the first condition is true, then statement1 is obeyed, otherwise if the second condition is true then statement2 is obeyed, and if neither condition is true, statement3 is obeyed.

Note, the elseif and else parts are optional, though an end is needed. An if statement can contain many elseif commands. Each statement could be one or many statements (including if statements).

A condition is true if the variable specified is non zero, or some relational operator returns true. For instance

```
if b
   x = [1 2];
end
```

will set x to [1 2] if b is not zero.

Alternative conditions, and their meaning, are shown below

```
if a > 3              % this tests if a is greater than 3
if a == 5             % this tests if a is equal to 5
if a ~= 6             % this tests if a does not equal 6
```

Ands and ors are possible

```
if (a > 2) & (a <= 5)    % true of a is greater than 2 and less than or equal to 5
if (a <=2) | (a>5)       % true if a is less than or equal to 2 or greater than 5.
```

The while command allows loops: its command syntax is

```
while condition
   statement
end
```

All the time the condition is true, the statement is obeyed. The statement can be one or many statements. Note, it is important that these statements include some action which will eventually result in the condition becoming false.

As an example, recall that for two polynomials to be added together, they must first be changed to being the correct size. Suppose p1 and p2 are polynomials, and p1 is smaller than p2. The following makes them the same size by inserting leading zeroes:

```
while length(p1) < length(p2)
   p1 = [0 p1];
end
```

Another way of looping is to use the for statement: its syntax is:

```
for x=p:q
   statement
end
```

The statement is obeyed when x takes each of the integer values between p and q. e.g.

```
a = [];
for x=1:3
   a=[a x];
end
```

Will obey the command a = [a x] when x is 1, 2 and 3. The overall result is [1 2 3].

The variable range can be changed to count down, or to change by values other than 1 by having a third value

```
for x=3:-1:1
```

will set x to be 3 then 2 then 1

```
for x:1:0.5:3
```

will set x to be 1, 1.5, 2, 2.5, 3

Example

The following illustrates the use of if and for, producing the 3D plot shown earlier. To explain, for values of x and y in the range -2 to 2, a value is calculated being:

$$z[x, y] = \frac{|x + jy|}{|-1 + x + jy|}$$

Note, the denominator could be zero, so a limit is placed on z[x,y]: if the value exceeds the limit, then the limit is stored in z[x,y]. The data are in a file Nyq3D.m, as follows:

```
% Nyq3D, plots closed loop gain of system
% If forward transfer function is A, and feedback gain is -1
% plot mod(A/(1+A)), where A is x+jy
% Note, A/(1+A) is infinite at A = -1, thus the plot is truncated at suitable value
% Dr Richard Mitchell 9/12/99
%
m = 15;                     % set number of points found
l = 4;                      % limit for a/1+a
z = l * ones(2*m,2*m);        % fill matrix with limit
r=[-m:m-1]*2/m;               % x,y values in range -2..2
for x= 1:2*m,                 % x range
  for y = 1:2*m,
                            % calc data at point r(x),r(y)
    a = abs(r(x) + j*r(y));     % find mod (xa + j ya)
    b = abs(r(x) - 1 + j*r(y)); % find mod (-1 + this)
    if (b ~= 0) & (a < b*l),    % if not div by 0, and not too big
      z(x,y)=a/b;           % store a/b
    end
  end
end;
mesh(r, r, z);              % draw mesh
view(-60,10);                 % set point from which graph viewed
grid;                       % add grid
colormap(1-white);            % set so drawn in black only
```

Note, x and y are variables in the range 1 to 2*m, but the values used to set z, are in the range -2 to 2. This conversion is achieved by using x and y to index into the matrix r which contains 2m values from -2..2.

Functions

The above is an example of a file containing a series of commands which are obeyed. An extra feature in MATLAB is to provide a file like the above, but one which returns a result. This in effect is defining a new function. The following is an example of such a function, one which computes the sin(x), but x is in degrees.

```
function ans = AngleSin (x)
% returns Sin(x), but x is in degrees
  ans = sin(x*pi/180);
```

This is stored in a file anglesin.m.

The syntax of the first line is

```
function result_variable = function_name (parameters)
```

The comments after this line form the header which is what is printed if you type

```
>>help function_name
```

The result from a function can be a single variable, being a scalar, complex number or matrix. If two variables are to be returned, one would have a function of the form

```
function [x, y] = fname (a, b, c)
```

in which there would be suitable statements assigning values to x and y.

For instance, a function to return the modulus and phase of the complex number whose real and imaginary components are x and y, is:

```
function [r, a] = Polar (x, y)
% Converts Cartesian representation of number x+jy
% to polar , modulus r, argument a
% Dr Richard Mitchell
  z = x + j*y;
  r = abs(z);
  a = angle(z);
```

This could be called by

```
>>[m, t] = Polar (3, 4)
```

A function with parameters and no return value is possible. For instance, for the Nyq3D example, it is more useful if the user can specify the number of points and the limit value when the function is called. In which case, the earlier example should have as its first line

```
function Nyq3D (m, l)
```

And the commands to assign values to m and l should be removed.

Then the figure can be drawn by a command of the form:

```
>>Nyq3D(25,5)
```

Note, to cause a function to end at any time, use the command

```
return
```

Invoking functions with different numbers of arguments

Suppose a function fred is one which returns 2 values and which is passed 4 arguments. One would call such a function by, for instance,

```
[x,y] = fred (1, 2, 3, 4);
```

It is also possible to call fred with different numbers of arguments, and commands in the function can be used to determine how to respond. These commands are nargin and nargout, which report the number of input and output arguments. The following example shows how this functionality could be employed.

```
function [x,y] = Ellipse (xo, yo, a, b)
% [x,y] = Ellipse(xo, yo, a, b)
% This function calculates an ellipse origin xo,yo and radii a, b
% If b not specified, then a circle is drawn.
% If called with no output arguments, Ellipse is plotted
% If called with output arguments, co-ordinates of ellipse returned in x,y
% Dr Richard Mitchell, 28.4.00
if nargin < 3
    error('Not enough arguments specified');
elseif nargin < 4
    b = a;                          % only a radius specified, so make b equal to a
end;
z = pi*[0:0.01:2];                  % calc array of values from 0 to 2*pi
x = a*cos(z)+xo;                    % calc x values
y = b*sin(z)+yo;                    % and y values
if nargout == 0                     % if no output arguments, plot ellipse
  plot (x,y,[max(x) min(x)], [0 0], [0, 0],[max(y), min(y)]);
end;                                % note plotted ellipse plus x and y axes
```

If invoked by

```
>> ellipse (-1, 5, 7);
```

It will calculate and plot a circle of radius 7 at origin -1,5.

If invoked by

```
>> ellipse (-1, 5, 7, 3);
```

It will calculate and plot an ellipse with radii 7 and 3 at origin -1,5.

If invoked by

```
>> [x,y] = ellipse (3, 5, 8);
```

It will calculate the co-ordinates of a circle radius 8 at origin 3,5, and store these co-ordinates in the vectors x and y.

Strings and Input

There are numerous other facilities in MATLAB. Much can be learnt of these by the use of the help command and by viewing the demonstrations. For instance

```
>>help strfun
```

tells the user about the various string commands. These allow, for example, numbers to be converted to stings and vice-versa, etc. It should be noted that MATLAB allows string variables. For instance

```
>> x = 'my string';
```

sets x to be the characters *my string*.

```
>> s = sprintf ('x is %d and y is %d', x, y)
```

Uses the sprintf function, like that provided in C, to generate a string with a message and values interspersed. If x = 100 and y = 50, this command outputs:

```
x is 100 and y is 50
```

More details can be found by typing

```
>> help sprintf
```

Another simple command is the Input command. This can be used to let the user enter a value to a function at run time. The following is an example of its use.

```
>> r = input('Enter value for radius r > ');
```

As a result, when this command is encountered MATLAB will prompt:

```
Enter value for radius r >
```

The user can then type in a value, and that value will be stored in the variable 5.

Returning different types of result

One can write a function which returns different types of result. In the following example, under certain conditions the function returns a numerical vale, at other times it returns a string. This function, findmax, processes two related, equal sized arrays of data. These data could, for instance, be a series of values in one array and the times when each of these values occurred is in the second array.

The function searches the first array and finds its largest value. It can then report various aspects. One, it can state what this maximum value is. Two, it can report the corresponding value in the second array: if the second array represents time, then the function is reporting the time at which the maximum value occurred. Three, the function can report the position in the arrays where the maximum value it occurs. The final option is to report all three.

For the first three options, the function returns a numerical value; but for the final option it returns a string, using the sprintf function.

The function findmax is as follows:

```
function ans = findmax (a1, a2, choice);
% ans = findmax (a1, a2, choice);
% function finds the maximum in a1: suppose this is at index cpos
% case choice of
%    1: ans = a2(cpos)     2: ans = a1(cpos); 3: ans = cpos;     0: ans = all 3.
% Dr Richard Mitchell
ct = 2;                          %  count from position 2
cpos = 1;                        %  guess maximum is at position 1 initially
while ct <= length (a1),         %  keep scanning til get to end of array
   if a1(ct) > a1(cpos),         %  if element greater than current guess of maximum
      cpos = ct;                 %  make current position the current guess of maximum
   end;
   ct = ct + 1;
end;
if choice == 1
   ans = a2(cpos);
elseif choice == 2
   ans = a1(cpos);
elseif choice == 0
   ans = cpos
else
   ans = sprintf('Max value %f at %f, position %d',a1(cpos),a2(cpos),cpos);
end;
```

Directory commands

MATLAB stores data in files in appropriate directories. Some commands exist which allow the user to handle these files, to identify them. For instance

```
>>dir
```

lists the current directory

```
>>cd newdir
```

Causes MATLAB to move to the directory newdir.

To go up a level newdir is specified by '..'.

As one might expect, the command to list the contents of file myfile.m is

```
>> type myfile
```

More details about these can be found by typing commands like

```
>>help dir
```

Debugging

Given that it is possible to write MATLAB functions, it is important to be able to debug those programs. This section considers a simple way of doing this. Note, this applies only to M files that contain a function. More details are available by typing:

```
>>help debug
```

Essentially the idea is that the user can specify points in these files where breakpoints can be put. Then the user can run the M file and MATLAB will obey all the commands until the break point is encountered. At this point the program stops and the user can find out the values of variables, set and then run to another breakpoint, single step through the program, or finish the debugging. Given that a script file can call other script files, special commands are provided for moving between the different levels.

Example Program

The following example illustrates debugging in action. It comprises a function sinorcos which calls another, anglesin. The functions are:

```
function sinorcos (iss)
% plots sin or cos function
% is iss = 0, plots cos, else plots sin
t = [0:360];
if iss
  x = anglesin(t);
else
  x = anglesin(t+90);
end;
plot (t,x);
```

where, anglesin is the following function in anglesin.m

```
function ans = AngleSin (x)
% returns Sin(x), but x is in degrees
   ans = sin(x*pi/180);
```

The first stage in debugging is to find out where breakpoints should be put. By typing

```
>> dbtype sinorcos
```

MATLAB will print out the function sinorcos with line numbers. The result is:

```
1   function sinorcos (iss)
2   % plots sin or cos function
3   % is iss = 0, plots cos, else plots sin
4   t = [0:360];
5   if iss
6     x = anglesin(t);
7   else
8     x = anglesin(t+90);
9   end;
10   plot (t,x);
```

This shows the line beginning with the if, is on line 5. To put a breakpoint there, type

```
>> dbstop at 5 in sinorcos
```

Then one can run the function in the normal way:

```
>>sinorcos(1)
```

and the program will stop at line 5 of sinorcos. The prompt is slightly different now

```
k>>
```

One can find out which variables exist:

```
k>>who
```

One of these variables is t. Its value can be found by:

```
k>>t
```

One can then get the system to then obey the next instruction

```
k>>dbstep
```

This will obey the if iss command. Then

```
k>>dbstep
```

Will cause then next line to be obeyed, namely x = anglesin(t).

Note, the program will not stop inside the call to anglesin, it will just obey the call to this function and set x accordingly. If one wanted the program to single step into anglesin, the command would be:

```
k>>dbstep in
```

This will cause the program to stop at the line

```
ans = sin(x*pi/180)
```

A useful command at this point is

```
k>>dbstack
```

which tells you whereabouts one is in the debugging. The answer in this case is:

```
In c:\matlab\rjm\anglesin.m at line 4
In c:\matlab\rjm\sinorcos.m at line 6
```

This means that the program is currently stopped at line 4 in anglesin, and the call to anglesin was from line 6 of sinorcos.

One can continue stepping through the program using dbstep. Or, the command dbcont, could be used to continue from the current breakpoint. Breakpoints can be cleared by:

```
>>dbclear at 5 in sinorcos
```

The debugging session can be quit by

```
>> dbquit
```

Other facilities are provided, such as putting a breakpoint which is activated only if there is an error. More details can be found by typing

```
>> help dbstop
```

Control and polynomials for systems

For simulation purposes, in the Control Toolbox, there are commands like step and impulse. For instance,

```
>> step ([1], [10 7 1])
```

computes and then plots the step response of a system defined by the transfer function

$$\frac{1}{10s^2 + 7s + 1}$$

One might want to calculate the step response, but not have it plotted:

```
>> [x, y, t] = step ([1], [10 7 1])
```

Calculates the step response and stores the response and time data in variables x and t.

If one wants to specify the time period over which the simulation takes place, do something like the following:

```
>> t = [0:0.1:10];
>> step ([1], [10 7 1], t)
```

The step command can also be invoked using state space models. For more details type

```
>> help step
```

Commands also exist for Bode, Nyquist, and Root Locus plots. These can be invoked in various ways, including:

```
>> Bode ([1], [10 7 1]);
```

Plots Bode diagram of system with the given transfer function

```
>> [m,p,w]=Bode ([1], [10 7 1]);
```

Calculates Bode diagram with the transfer function and returns the magnitude values, the phase values and the angular frequencies in m,p,w.

```
>> Nyquist ([1], [10 7 1]);
```

Plots the Nyquist diagram.

```
>> [rp,ip,w]=Nyquist ([1], [10 7 1]);
```

Calculates the Nyquist diagram and returns the real part, imaginary part and angular frequency of each part of the locus in rp, ip and w.

More details are available using the help facility.

One reason why one might want to have the data for a Nyquist diagram calculated but not plotted, is to plot a Nyquist diagram containing the locus and say the unit circle centred on the -1,j0 point, and some axes. This could be achieved by the following:

```
>> [rpart,ipart,w]=Nyquist ([1], [10 7 1]);
>> [cx,cy] = ellipse (-1, 0, 1);
>> xmm = [max([max(rpart) max(mx)]), min([min(rpart) min(mx)])];
>> ymm = [max([max(ipart) max(my)]), min([min(ipart) min(my)])];
>> plot (rpart,ipart, cx,cy, xmm, [0 0], [0 0], ymm);
```

This uses the function ellipse defined before to calculate the points of the circle.

The assignment to xmm is a vector containing the largest and smallest x values of the data. The largest x value is the larger of the largest value in rpart and the largest value in cx. The max command is used to effect here. The min command finds the smallest value. Similar commands are used to find the values in the y direction. The arrays xmm, and ymm and [0 0] are then used to plot the axes.

If a system comprises two blocks in a feedback circuit, where the top block is defined by the polynomials An/Ad and the feedback block is defined by the polynomials Bd/Bn, then the closed loop transfer function can be found using the command.

```
>>[n d] = feedback (An,Ad,Bn,Bd,-1)
```

will return in n and d the transfer function

$$\frac{An/Ad}{1 - - AnBn/AdBd}$$

Solution of Differential Equations

The ode45 function, for example, can be used to solve the equation $\frac{dy}{dt} = f(t, y)$

First one writes an M file for the function f. For instance, odefunc.m is:

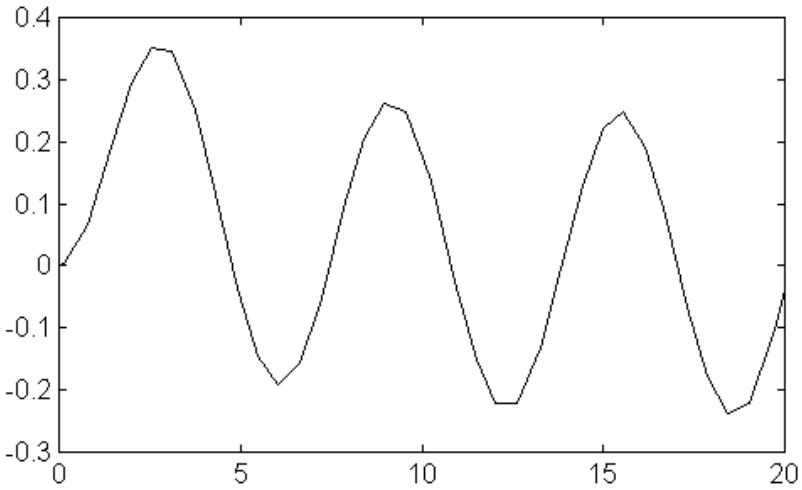
```
function ans = odefunc(t,y)
   ans(1) = (sin(t)-y(1)) / 4;
```

This is the function for f(t,y) = (sin(t)-y) / 4.

Then this can be solved and plotted by:

```
>>[t, y]=ode45('odefunc',0, 20, 0);
>>plot (t,y);
```

Here ode45 returns an array with the time values t and the computed values y. The values passed to ode45 are the function being solved, the initial and final values of t over which the function is processed, and the initial value of y. The result of the above is as follows.



More details on this and other functions can be found using the help system.

Fourier Transforms

Commands are available for finding the Fourier Transform of a set of data. Suppose x is a vector containing values of a signal at set times. Then

```
>> fft(x)
```

returns the Fast Fourier Transform of x, as a vector of complex numbers. The amplitude of these numbers gives the amplitude spectrum of the transformed data. These could be plotted as a bar graph by:

```
>> bar(abs(fft(x)));
```

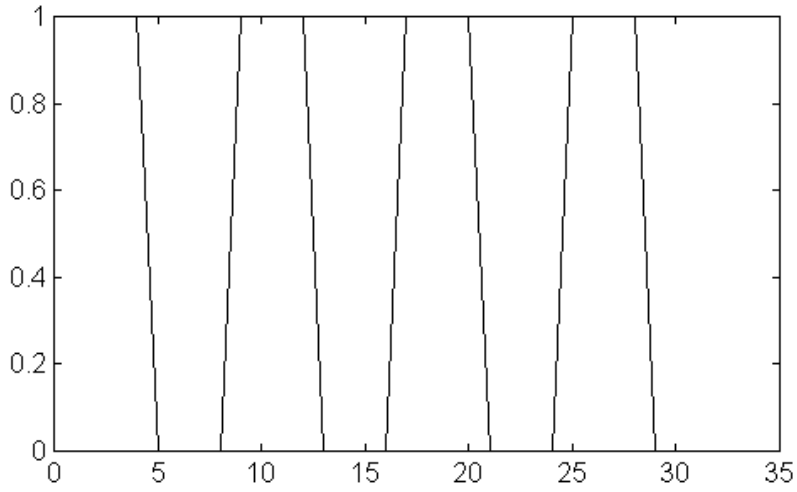
The phase spectrum is could be plotted by:

```
>> bar(angle(fft(x)));
```

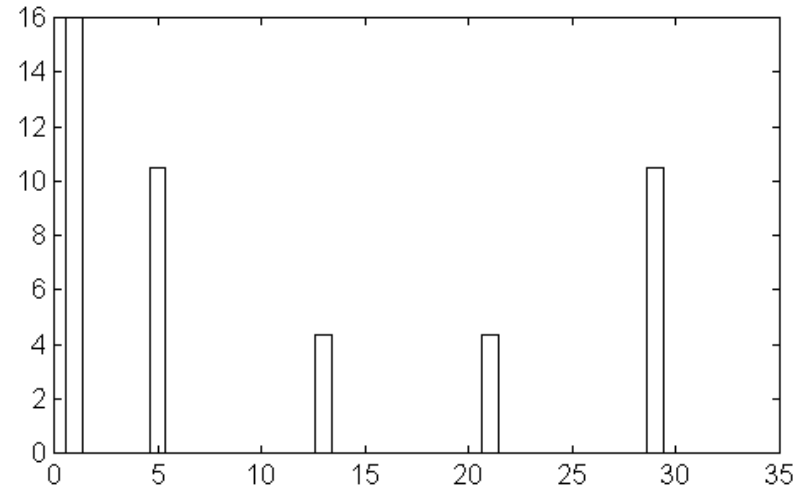
Suppose x represents a square wave. This can be set by

```
>> x = [1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0];
```

This gives a square wave looking as follows:



Then, for instance, the amplidude spectrum is as follows.



More details on this and related functions are available on the help system.