MATLAB Basics

MATLAB numbers and numeric formats

All numerical variables are stored in MATLAB in **double precision floating-point** form. (In fact it is *possible* to force some variables to be of other types but not easily and this ability is not needed here.) *Floating-point* representation of numbers is essentially equivalent to the "scientific notation" of your calculator. Specifically a (real) number x is stored in *binary* floating point form as

$$x = \pm f * 2^E$$

where f is called the *mantissa* and E the *exponent*. The exponent is an integer (positive or negative) and the mantissa lies in the range $1 \leq f < 2$. This representation is entirely *internal* to the machine and its details are not important here. (They are important when analyzing carefully the performance of numerical algorithms.)

Similar formats using the conventional *decimal* system are available for MATLAB input and output. These formats can be set in the **F**ile – **P**references menu in the MATLAB command window. They can also be changed using MATLAB commands.

```
» format short
» pi
ans =
 3.1416
» format long
» pi
ans =
 3.14159265358979
» format short e
» pi
ans =
 3.1416e+000
» 100*pi
ans =
 3.1416e+002
```

What does this e *mean?* The result 3.1416e+000 just means 3.1416×10^0 where three digits are allowed for the *decimal* exponent. Similarly the final answer 3.1416e+002 means 3.1416×10^2 or, approximately, 314.16. In the same way, 3.1416e–002 means 3.1416×10^{-2} or, approximately, 0.031416

One final note on this first MATLAB session: clearly the constant π is known and is built into MATLAB to high accuracy with the name pi. You should therefore avoid using pi as a variable name in your MATLAB sessions!

Strings and printing

In more advanced applications such as symbolic computation, string manipulation is a very important topic. For our purposes, however, we shall only need very limited skills in handling strings initially. One most important use might be to include Your Name and the Course as part of your MATLAB workspace in a simple, and automatic, way.

This is easily achieved by using strings and the MATLAB print function fprintf in a special file called startup.m which will be executed automatically when you start MATLAB.

Strings can be defined in MATLAB by simply enclosing the appropriate string of characters in single quotes such as

```
≫ s='My Name'
```

results in the output

```
s =
My Name
```

More complicated strings can be printed using the MATLAB function fprintf. This is essentially a C programming command which can be used to obtain a wide-range of printing specifications. Very little of its detail is needed at this stage. We use it here only to illustrate printing strings. If you want more details on advanced printing, full details are provided in your MATLAB manual. The use of fprintf to print the Name and Course information is illustrated by

```
≫ fprintf(' My Name \n Course \n')
 My Name
 Course
≫
```

where the \n is the new-line command. The final one is included to ensure that the next MATLAB prompt occurs at the beginning of the next line rather than immediately at the end of the printed string.

To make this appear automatically every time you start MATLAB, simply create a file called startup.m which has this one command in it. To create this m-file, click the "New m-file" icon in the MATLAB window. Type the single line

```
fprintf(' My Name \n Course \n')
```

and then save this file with the name "startup". By default, MATLAB will save this in a folder called Work which is where you want it to be. (In older versions of MATLAB, the default location may be different, but is still acceptable.)

The function fprintf can be used for specifying the format of MATLAB's numeric output, too. However, for our purposes, the various default numerical formats are usually adequate.

Editing in the Command Window

Although we have not yet done very much in the MATLAB Command Window, it is worth summarizing some of the basic editing operations available there. Most of the elementary editing operations are completely standard:

BackSpace and Delete work just as in any other Windows-based program;
Home moves the cursor to the beginning of the current line;
End moves the cursor to the end of the current line;
→ and ← move the cursor one place right or left;
Ctrl → and Ctrl ← move one *word* right or left.
Ctrl-C and Ctrl-V have their usual effect of copying and pasting.

There are some nonstandard editing shortcuts which are useful:
Esc deletes the whole of the current line;

Ctrl-K deletes from the cursor to the end of the current line.

The ↑ and ↓ keys have special roles:

↑ recalls the previous line (repeatedly up to many lines) so that earlier lines can be reproduced. This is often useful when you want to correct errors and typos in earlier code without having to retype complicated instructions — or simply to repeat an operation.

↓ recalls the next line. (Obviously if the current line is the last one, then there is no next line.)

These two keys are especially important because of the fact that MATLAB is interactive and each line is executed as it is completed so that you cannot go back up the Workspace to correct an earlier mistake.

Arithmetic operations

Arithmetic in MATLAB follows all the usual rules and uses the standard computer symbols for its arithmetic operation signs.

Thus we use

Symbol	Effect
+	Addition
−	Subtraction
∗	Multiplication
/	Division
^	Power

In our present context we shall consider these operations as *scalar* arithmetic operations which is to say that they operate on two numbers in the conventional manner. MATLAB's arithmetic operations are actually much more powerful than this. We shall see just a little of this extra power later.

The conventional algebraic order of precedence between the various operations applies. That is,

$$\left(\text{expressions in parentheses}\right)$$

take precedence over

$$\text{powers, } \wedge$$

which take precedence over

$$\text{multiplication and division, } *, /$$

which, in turn, take precedence over

$$\text{addition and subtraction, } +, -$$

We have in fact already seen examples of some of these operations.

Assignment of values — whether direct numerical input or the result of arithmetic — is achieved with the usual = sign. Therefore, within MATLAB (and other programming languages) we can legitimately write "equations" which are mathematically impossible. For example, the assignment statement

```
≫ x = x+0.1
```

has the effect of incrementing the value of the variable x by 0.1 so that if the current value of x before this statement is executed is 1.2 then its value after this is executed is 1.3. Such actions are often referred to as overwriting the value of x with its new value.

Similarly the MATLAB commands

```
≫ i=1;
≫ i=i+2
```

result in the output

```
i = 3
```

Note the use of the ; to suppress the output from the first line here.

There are some arithmetic operations which require great care. The order in which multiplication and division operations are specified is especially important.

What is the output from the following MATLAB commands?

```
» a=2; b=3; c=4;
» a/b*c
```

Here the absence of any parentheses results in MATLAB executing the two operations (which are of *equal* precedence) from left-to-right so that

- First a is divided by b,

and then

- The *result* is multiplied by c.

The result is therefore

```
ans =
 2.6667
```

Note here the default "variable" ans is used for any arithmetic operations where the result is not assigned to a named variable.

This arithmetic is equivalent to $\frac{a}{b}c$ or as a MATLAB command

```
» (a/b)*c
```

Similarly, a/b/c yields the same result as $\frac{a/b}{c}$ or $\frac{a}{bc}$ which could (alternatively) be achieved with the MATLAB command

```
» a/(b*c)
```

Use parentheses to be sure MATLAB does what you intend!

MATLAB's mathematical functions

All of the standard mathematical functions — often called the *elementary* functions — that you will meet in your Calculus courses are available in MATLAB using their usual mathematical names. Many other functions — the *special* functions — are also included; you will most likely come across some of these in later mathematics and, more especially, engineering courses.

The elementary functions are listed in your User's Guide. This listing includes several functions which will not be familiar to you yet, and several that we shall not deal with in this book. The important functions for our purposes are:

abs (x)	Absolute value
sqrt (x)	Square root
sin (x)	Sine
cos (x)	Cosine
tan (x)	Tangent
log (x)	Natural logarithm
exp (x)	Exponential function, e^x
atan (x)	Inverse tangent, or arctan
asin (x)	Inverse sine, or arcsin
acos (x)	Inverse cosine, or arccos
cosh (x)	Hyperbolic cosine
sinh (x)	Hyperbolic sine

Note that the various trigonometric functions expect their argument to be in *radian* (or pure number) form — **NOT** in degrees which are an artificial unit based on the ancient Babylonians' belief that a year was 360 days long! Forexample,

```
» sin(pi/3)
```

gives the output

```
ans =
 0.8660
```

Shortly we shall see how to use these functions to generate both tables of their values and plots of their graphs. Also we shall see how to define other functions either as strings in the MATLAB command window, or more usefully, as function m-files which can be saved for repeated use.

Vectors and Matrices

In MATLAB the word *vector* should really be interpreted simply as "list of numbers". Strictly it could be a list of other objects than numbers but "list of numbers" will fit our needs for now. These *can* be used to represent physical vectors but are much more versatile than that as we shall see.

There are two basic kinds of MATLAB vectors: *Row* and *Column* vectors. As the names suggest, a row vector stores its numbers in a long "horizontal list" such as

$$1, 2, 3.4, 1.23, -10.3, 2.1$$

which is a row vector with 6 components. A column vector stores its numbers in a vertical list such as

$$\begin{array}{c} 1 \\ 2 \\ 3.4 \\ 1.23 \\ -10.3 \\ 2.1 \end{array}$$

which is a column vector with (the same) 6 components. In mathematical notation these arrays are usually enclosed in brackets [ ].

There are various convenient forms of these vectors, for allocating values to them, and accessing the values that are stored in them. The most basic method of accessing or assigning individual components of a vector is based on using an *index*, or *subscript*, which indicates the position of the particular component in the list. The MATLAB notation for this subscript is to enclose it in parentheses ( ).

For assigning a complete vector in a single statement, we can use the [ ] notation. These two are illustrated for the above row vector in the following MATLAB session:

```
» x=[1,2,1.23,3.4,-8.7,2.3]
x =
 1.0000 2.0000 1.2300 3.4000 -8.7000 2.3000
» x(2)=x(1)+2*x(3)
x =
 1.0000 3.4600 1.2300 3.4000 -8.7000 2.3000
```

The first command simply initializes the vector with the given components. The second performs arithmetic to set the second component equal to the first plus twice the third. Note that the full vector is output, and that the other components are unchanged.

In entering values for a row vector, spaces could be used in place of commas. For the corresponding column vector simply replace the commas with semi-colons.

- **All MATLAB vectors have their index or subscript begin at 1.**
  This is *NOT* something that the user can vary. For most applications this causes little difficulty but there are times when we must take special precautions in MATLAB programming to account for this.
- **To switch between column and row format for a MATLAB vector we use the** *transpose* **operation denoted by '.**
  This is illustrated for the row vector above by:

```
» x=x'
x =
 1.0000
 3.4600
 1.2300
 3.4000
 -8.7000
 2.3000
```

To switch back to row form: just use the transpose operator again.

MATLAB has several convenient ways of allocating values to a vector where these values fit a simple pattern.

A *matrix* is simply a rectangular array of numbers (or other objects). Thus a matrix is a row of columns in which each entry is a column vector of the same length, or a column of rows in which each entry is a row of teh same length. The entries are indexed in a similar way to vectors: A(2,3) refers to the element in Row2 and Column3. Thus the elements of a 3×4 matrix are labeled in MATLAB as follows:

$$\begin{bmatrix} A(1,1) & A(1,2) & A(1,3) & A(1,4) \\ A(2,1) & A(2,2) & A(2,3) & A(2,4) \\ A(3,1) & A(3,2) & A(3,3) & A(3,4) \end{bmatrix}$$

Mathematically this same array is usually written as

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}$$

The same matrix is a row of columns

$$\left[A(:,1),\ A(:,2),\ A(:,3),\ A(:,4)\right]$$

or a column of rows:

$$\begin{bmatrix} A(1,:) \\ A(2,:) \\ A(3,:) \end{bmatrix}$$

where in each case the : denotes a "wild card" meaning all entries. The use of : in MATLAB is more powerful than this suggests. The next section explains MATLAB's : notation more fully. This : notation can also be used within teh subscripts of a vector or matrix. We shall see this in examples.

MATLAB's : notation

The *colon* : has a very special and powerful role in MATLAB. Basically, it allows an easy way to specify a vector of *equally-spaced* numbers.

There are two basic forms of the MATLAB : notation.

- **Two arguments separated by a : as in**
  ```
  » v=–2 : 4
  ```
  generates a row vector with first component –2, **last one** 4, **and others spaced at unit intervals.**
- **Three arguments separated by two :'s has the effect of specifying the starting value : spacing : final value.**

For example, the MATLAB command

```
» v=–1:0.2:2;
```

generates the row vector

v = −1.0, −0.8, −0.6, −0.4, −0.2, 0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0.

Remember the syntax is start:step:stop. Also the step can be negative.

There are no practical restrictions on the length of such a vector (or, therefore, on the range or spacing which can be used). Of course very long vectors may have two negative effects:

Computing times are likely to rise dramatically, and

Output may take up so much room in the Window that you have no way of fathoming it — or printing it in reasonable time.

- **Don't forget the use of a semi-colon ; to suppress output in the Command Window**.

There is another very useful aspect of MATLAB's colon notation. It can also be used to specify a *range* of subscripts for a particular operation. We see this in the following example.

```
» w=v(3:8)
w =
 –0.6000 –0.4000 –0.2000 0 0.2000 0.4000
```

Note that the index of w still starts at 1, so that, for example,

```
» w(3)
ans =
 –0.2000
```

linspace and logspace

MATLAB has two other commands for specifying vectors conveniently.

- linspace **is used to specify a vector with a given number of equally-spaced elements between specified start and finish points**

  This needs some care with counting to get convenient spacing.

  For example:

  ```
  » x=linspace(0,1,10)
  ```

  results in the vector [0, 0.1111, 0.2222, 0.3333, 0.4444, 0.5556, 0.6667, 0.7778, 0.8889, 1.0000]. Using 10 points results in just 9 *steps.*

  ```
  » x=linspace(0,1,11)
  ```

  gives the vector [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0].
- logspace **has a similar effect — except that the points are spaced on a logarithmic scale**.

Tables of function values

We can use MATLAB's vectors to generate tables of function values. For example:

```
» x=linspace(0,1,11);
» y=sin(x);
» [x',y']
```

generates the output

```
ans =
 0 0
 0.1000 0.0998
 0.2000 0.1987
 0.3000 0.2955
 0.4000 0.3894
 0.5000 0.4794
```

```
 0.6000 0.5646
 0.7000 0.6442
 0.8000 0.7174
 0.9000 0.7833
 1.0000 0.8415
```

Note the use of the transpose to convert the row vectors to columns, and the separation of these two columns by a comma.

Note also that all the standard MATLAB functions are defined to operate on vectors of inputs in an element-by-element manner. The following example illustrates the use of the : notation and arithmetic within the argument of a function.

```
» y=sqrt(2+3*(0:0.1:1)')
y =
 1.4142
 1.5166
 1.6125
 1.7029
 1.7889
 1.8708
 1.9494
 2.0248
 2.0976
 2.1679
 2.2361
```

Array arithmetic

Array arithmetic allows us to perform the equivalent arithmetic operations on all the components of a vector. In most circumstances the standard arithmetic symbols achieve what is wanted and expected. However, especially for multiplication, division and powers involving MATLAB vectors (or matrices) a dot . is needed in conjunction with the usual operation sign.

These various operations are best illustrated with some simple MATLAB examples. For these examples we shall use the following data vectors a and b, and scalar (number) c.

```
» a=[1,2,3,4]; b=[5,3,0,2]; c=2;
```

Since the vectors a and b have the same size, they can be added or subtracted from one another. They can be multiplied, or divided, by a scalar, or a scalar can be added to each of their components. These operations are achieved using the following commands which produce the output shown:

```
» a+b
ans =
 6 5 3 6
» a-2*b
ans =
 -9 -4 3 0
» c*a
ans =
 2 4 6 8
```

```
» a+c
ans =
 3 4 5 6
» b/c
ans =
 2.5000 1.5000 0 1.0000
```

Mathematically the operation of division by a vector does not make sense. To achieve the corresponding componentwise operation, we use `c./a`. Similarly, for powers we use .^as follows

```
» c./a
ans =
 2.0000 1.0000 0.6667 0.5000
» a.^c % squares of components of a
ans =
 1 4 9 16
» c.^a
ans =
 2 4 8 16 % powers of c=2
```

Similarly the mathematical operation $a * b$ is not defined but we may wish to generate the vector whose components are the products of corresponding elements of a and b.

```
» a.*b
ans =
 5 6 0 8
» b.^a
ans =
 5 9 0 16
» a./b
Warning: Divide by zero.
ans =
 0.2000 0.6667 Inf 2.0000
```

Note the warning created by the division by zero in the third element of the final operation here.

String functions

The simplest user-defined functions in MATLAB are created as strings in the command window. If we may need to evaluate these for arrays of arguments, remember to use the "dot operations" where necessary.

As an example, evaluate the function

$$f(x) = 2x^3 - \frac{3x}{1+x^2}$$

for $x = 0, 0.3, 0.6, \ldots, 3$.

```
» x=0:0.3:3;
» f='2*x.^3-3*x./(1+x.^2)';
» y=eval(f);
» [x',y']
```

```
ans =
 0 0
 0.3000 -0.7717
 0.6000 -0.8915
 0.9000 -0.0337
 1.2000 1.9806
 1.5000 5.3654
 1.8000 10.3904
 2.1000 17.3575
 2.4000 26.5829
 2.7000 38.3889
 3.0000 53.1000
```

The only difficulty here is remembering that each time we form products, quotients or powers of vectors, the corresponding dot operation must be used.

However there is another major drawback to this simple approach. Every time we wish to evaluate the function f, we must ensure that the argument is stored in a variable called x. This may be inconvenient if we have several functions and arguments within a computation.

For this reason, function m-files are a much superior way of creating and saving user-defined functions.

Function m-files

To create a new m-file, click the "New m-file" icon in the MATLAB command window.

If the m-file is to be a function m-file, the first word of the file is `function`, we must also specify names for the function, its input and output. The last two of these are purely *local* variable names. These mimic the mathematical idea of defining a function

$$y = f(x)$$

but then assigning, for example, $z = f(2)$. This is just a shorthand for temporarily setting $x = 2$, evaluating the output y of the function and assigning this value to z. Function m-files work the same way.

To illustrate we again evaluate the function

$$f(x) = 2x^3 - \frac{3x}{1+x^2}$$

for $x = 0, 0.3, 0.6, \ldots, 3$ but this time using a function m-file.

The m-file could be:

```
function y=fun1(x)
y=2*x.^3-3*x./(1+x.^2);
```

Then the commands

```
» v=(0:0.3:3)';
» fv=fun1(v);
» [v,fv]
```

generate the same table of output as before.

Note that this time, the vector `v` is a column vector and, consequently, so is `fv`. There is no need for any transpose operations in the output line.

MATLAB m-files can be *much* more involved than this, as you will see during your course, but the principles are the same.

Script m-files

A script m-file is a MATLAB *program*. It consists of the set of MATLAB instructions for performing a particular task. It is run by simply typing its name in the MATLAB command window. The startup.m file created earlier is a script m-file.

As well as storing complete programs, perhaps the best way to save work if you are in the middle of an assignment is to use a “script” m-file. By copying the instructions you used into an m-file and saving it with some appropriate name, you can then recover to exactly the point at which you stopped work. Further instructions can then be added in the command window, or by editing the m-file to add the new instructions.

As a simple example of a script m-file, we could store the m-file containing the three lines

```
v=(0:0.3:3)’;
fv=fun1(v);
[v,fv]
```

with the name Ex_table.m and then the command

```
» ex_table
```

will again produce the same table of function values as we generated above.

Again, obviously, script m-files can be *much* more complicated than this!

Plotting

MATLAB has several methods for plotting – both in two- and three-dimensional settings. We shall concentrate on just one of the two-dimensional plotting functions, the most powerful, plot.

MATLAB’s plot function has the ability to plot many types of “linear” two-dimensional graphs from data which is stored in vectors or matrices. The user has control over the data points to be used for a plot, the line styles and colors and the markers for plotted points. All of these variations are detailed under plot in the User’s Guide. In this section we simply illustrate some of them with examples.

Markers

The markers used for points in plot may be any of

point	$\cdot$
circle	$\circ$
cross	$\times$
plus	$+$
star	$*$

- **If any of these markers is used the corresponding plot consists only of the discrete points**.

Line Styles

The linestyle can be a solid line as before or one of several broken lines. These are particularly useful in distinguishing curves for monochrome printing. These styles are denoted by

solid line	–
dotted line	:
dashed line	– –
dash-dot	–.

Colors

The colors available for MATLAB plots are

yellow	y	green	g
magenta	m	blue	b
cyan	c	white	w
red	r	black	k

The color choice may be combined with any of the line styles or markers.

The use of these is illustrated with the following commands. First the data vectors

```
» x=(-5:0.1:5)';
» y=twofuns(x);
```

are used for our illustrations. The file twofuns.m is

```
function y=twofuns(x)
y(:,1)=1./(1+x.^2);
y(:,2)=x./(1+x.^2);
```

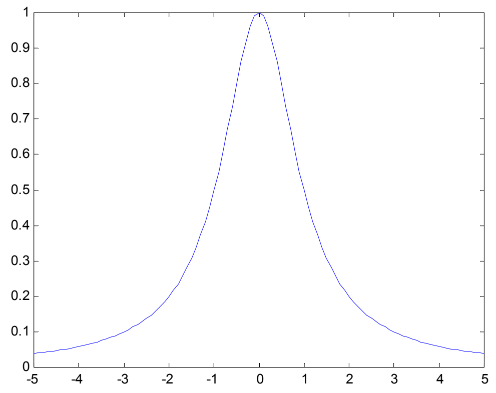
The matrix y has two columns the first of which has the values of $\frac{1}{1+x^2}$ and the second has the values of $\frac{x}{1+x^2}$.

Perhaps the simplest method of obtaining multiple plots on the same axes is to use the command hold on. The effect of this is that the next plot will be added to the current plot window. To cancel this in order to produce a new figure, hold off can be used. The command hold acts as a simple toggle switch between these two states.

To obtain a plot of just the first of these functions, we can use

```
» plot(x,y(:,1))
```

which produces a “blue solid line” graph like that below



Blue, solid line is the default first graph generated by plot. To change this we use syntax such as 'm+' which would be added to the command to force magenta plus signs to be plotted. The command

```
» plot(x,y(:,1),'k:')
```

yields the same curve plotted as a dotted black line.

It is worth noting that these effects are reliable screen effects, and are usually reliable for printing directly from MATLAB. Sometimes when graphics such as these are copied into other documents, the printer drivers will convert all line styles to solid lines!

To get the plots of both functions on the same axes — but with different linestyles — use a command such as

```
≫ plot(x,y(:,1),'k',x,y(:,2),'g--');
```

which results in the first curve being plotted as a solid black line, and the second as a dashed green line.

Note that the vector x is needed for both sets of data.

MATLAB program control

We consider the various ways of controlling the *flow* of a piece of MATLAB code. These include:

- for **loops** — which enable us to have an operation repeated a specified number of times.

  This may be required in summing terms of a series, or specifying the elements of a nonuniformly spaced vector such as the first terms of a sequence defined recursively.
- while **loops** — which allow the number of times the loop operation is performed to be determined by the results.

  This is often used in iterative processes such as obtaining approximations to the solution of an equation.
- if … else … — which is used when different actions are to be taken depending on the value of some variable.

  A simple example would be in obtaining the maximum of two quantities.

Often the else clause may not be needed when a command is to be performed only if some condition is satisfied while *nothing* is done if the condition is not satisfied. The *condition* itself may be a compound condition based on more than one "true-false" determination. We shall see examples of all these.

for loops

The simplest for loops in MATLAB are of the form

```
for var=start : finish
 first_command
 . . .
 last_command
end
```

where start and finish are integers usually with start < finish. (If finish < start, the loop is said to be *empty* and none of the commands will be performed.)

Each of the commands in the loop will be performed once for each value of var beginning with the start value and increasing by 1 each time until the commands are executed for the last time with var=finish.

As a simple example the following commands generate the sum

$$2 + 2/3 + 2/9 + \cdots + 2/3^{10}$$

The terms are generated recursively – each is just one-third of its predecessor.

```
» term=2;
» S=2;
» for k=1:10
 term=term/3;
 S=S+term;
end
» S
S =
 2.99998306491219
```

Note that both term and sum must be initialized before the loop.

for loops can be nested. For example if we wished to repeat the summation for common ratios other than 3, we might use the following loops.

```
» for n=2:5
 term=2;
 S(n)=2;
 for k=1:10
 term=term/n;
 S(n)=S(n)+term;
 end
end
» [(2:5)',S(2:5)']
ans =
 2 3.99805
 3 2.99998
 4 2.66667
 5 2.50000
```

for loops can use a step other than 1. The general format is

```
  for counter=start:step:stop
  . . .
end
```

while loops

For many situations a loop is required *but* we don't know in advance how many times its commands will need to be performed. One situation where this arises regularly is in obtaining repeated approximations to a quantity where we seek to achieve some specified accuracy in the final result. For example, if we set $x = 1$ and then repeat the instruction

```
» x=cos(x)
```

we will see that the values begin to settle down. Suppose we wish to compute these until they agree to within 4 decimal places.

To achieve this we can use a while loop which is controlled not by a simple counter but by a logical condition. The loop

```
» x1=1;x0=0;
» while abs(x1-x0)>1e-4
 x0=x1;
 x1=cos(x0);
end
```

yields the final result

```
x1 = 0.7391
```

Note that we need two values in order to make the comparison. Therefore two initial values must be specified, and we must update them each time through the loop.

Again while loops can be nested, while loops may contain for loops, and vice versa.

The control of while loops can be more complicated than here. It can contain compound logical conditions.

Logical and relational operators

There are three basic logical operators which are important in MATLAB programming:

&	AND
|	OR
~	NOT

The meanings of these should be fairly obvious. Just remember that the mathematical meaning of OR is *always* inclusive. The effects of these operators is simply summarized in the following *truth table* where T and F denote "true" and "false" respectively. In MATLAB these values are actually represented numerically using 0 for false and 1 for true. (Strictly, any *nonzero* equates to "true" so that some arithmetic can be done with logical values — *if you are very careful*!)

A	B	A&B	A|B	~A
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

There are other logical *functions* in MATLAB. The details can be found in the User's Guide.

Generally, you should use ( ) round the various components of compound logical conditions to be sure you have the precedence you intend. For example, (A&B)|C and A&(B|C) have very different meanings — just compare the two truth tables.

The relational operators which are often used in testing are mostly straightforward:

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

Note especially the *double* == for logical testing of equality. The single = is only used in MATLAB for assignments.

if ... else ...

Finally, we introduce the basic structure of MATLAB's logical branching commands. Frequently in programs we wish the computer to take different actions depending on the value of some variables. Strictly these are logical variables, or, more commonly, logical *expressions* similar to those we saw in defining while loops.

There are three basic constructions all of which begin with if and finish with end. The simplest has the form

```
if condition
  commands
end
```

in which the statements in the *commands* block are **only** executed if the *condition* is satisfied (true). If the condition is not satisfied (false) then the *commands* block is skipped.

The second situation is

```
if condition
  true_commands
else
  false_commands
end
```

in which the first set of instructions, the *true_commands* block, are executed if *condition* is true while the second set, the *false_commands* block, are executed if *condition* is false.

As a simple example the following code finds the maximum of two numbers:
```
» if a>=b
    maxab=a;
  else
    maxab=b;
end
```
The third, and most general, form of branch is
```
   if first_condition
     first_true_commands
   elseif second_condition
     second_true_commands
   else
     second_false_commands
   end
```
In this case, the *first_true_commands* block is executed if the *first_condition* is true. Otherwise (in which case *first_condition* is false) a *second_condition* is tested: if it is true the *second_true_commands* block is executed; if this second condition is *also* false then the *second_false_commands* block is executed.

In fact, if … elseif … elseif … ⋯ … else … end blocks can be *nested* arbitrarily deep**.** Only the statements associated with the ***first*** true condition will be executed**.**

The code below could be used to determine the nature of the roots of an arbitrary quadratic equation $ax^2 + bx + c = 0$.
```
» if a==0
      if b==0
        if c==0
          fprintf('Every x is a solution \n')
        else                    % else of c==0
          fprintf('No solutions \n')
        end
      else                      % else of b==0
        fprintf('One solution \n');
      end
    elseif b^2-4*a*c>0          % else of a==0,
      fprintf('Two real solutions \n');
    elseif b^2-4*a*c==0
      fprintf('One repeated real solution \n');
    else
      fprintf('Two complex solutions \');
end
```