# 2 Writing Scripts and Functions

## 2.1 Creating Scripts and Functions

With the preliminaries out of the way we now turn our attention to actually using MATLAB by writing a short piece of code. Most of the commands in this section have purposely been written so they can be typed at the prompt, `>>`. However, as we develop longer codes or ones which we will want to run many times it becomes necessary to construct scripts. A *script* is simply a file containing the sequence of MATLAB commands which we wish to execute to solve the task at hand; in other words a script is a computer program written in the language of MATLAB.

To invoke the MATLAB editor[1] we type `edit` at the prompt. This editor has the advantage of understanding MATLAB syntax and producing automatic formatting (for instance indenting pieces of code as necessary). It is also useful for colour coding the MATLAB commands and variables. Both of these attributes are extremely useful when it comes to debugging code. The MATLAB editor also has the feature that once a piece of code has been run the values of variables can be displayed by placing the mouse close to the variable's location within the editor. This is extremely useful for seeing what is going on and provides the potential to identify where we might have made a mistake (for instance, if we had set a variable to be the wrong size).

[1] You can of course make use of any other editor you have available on your computer. We have chosen to use the built-in MATLAB editor. Its implementation may differ slightly from platform to platform. If you are unsure of its use try typing `help edit` at the MATLAB prompt.

**Example 2.1** *We begin by entering and running the code:*

```
a = input('First number ');
b = input('Second number ');
disp([' Their sum is ' num2str(a+b)])
disp([' Their product is ' num2str(a*b)])
```

*This simple code can be entered at the prompt, but that would defeat our purpose of writing script files. We shall therefore create our first script and save it in a file named `twonums.m`. To do this, first we type `edit` at the MATLAB prompt to bring the editor window to the foreground (if it exists) or invoke a new one if it doesn't. Along the base of the typing area are a set of tabs. These allow you to switch between multiple codes you may be simultaneously working on. Since this is our first use of the editor, MATLAB will have given this code the default name `Untitled.m`.*

*To proceed we type the above code into the editor and then use the File Menu (sub item Save As) to change the name of the code and Save it as `twonums.m`. We will need to erase the current default name (`Untitled.m`) and type the new filename. After a code has been named we can use the save icon (a little picture of a disc) to save it, without the need to use the File Menu. If we now return to the MATLAB window and enter the command `twonums` at the prompt, our code will be executed; we will be asked to enter two numbers and MATLAB will calculate, and return, their product and sum. The contents of the file can be displayed by typing `type twonums`.*

This simple example has introduced two new commands, `input` and `num2str`. The `input` command prompts the user with the flag contained within the quotes `' '` and takes the user's response from the standard input, in this case the MATLAB window. In the first example it then stores our response in the variable `a`. The second command `num2str` stands for *number-to-string* and instructs MATLAB to convert the argument from a number, such as the result of `a+b`, to a character string. This is then displayed using the `disp`.

Important Point

It is very important you give your files a meaningful name and that the files end with `.m`. You should avoid using filenames which are the same as the variables you are using and which coincide with MATLAB commands. Make sure you do not use a dot in the body of the filename and that it does not start with a special character or a number.

For instance `myfile.1.m` and `2power.m` are not viable filenames (good alternatives would be `myfile_1.m` and `twopower.m` respectively). Filenames also have the same restrictions which we met earlier for variable names (see page 4). When processing a command, MATLAB searches to see if there is a user-defined function of that name by looking at all the files in its search path with a `.m` extension. If you are in any doubt whether something is a MATLAB command use either the command `help` or the command `which` in combination with the filename, for instance `help load` or `which load` for the MATLAB command `load`.

You should consider creating a directory for your MATLAB work. For instance on a Unix machine the sequence of commands (at the Unix prompt)

```
mkdir Matlab_Files
cd Matlab_Files
matlab
```

will create a directory (which obviously only needs to be done once), the second command changes your working directory to `Matlab_Files` and the third command invokes MATLAB (to check that you are in the correct directory use the command `pwd` to 'print working directory'). In a Windows environment you can use Explorer to create a new folder and then invoke MATLAB by clicking on the MATLAB icon. On a Macintosh you can simply create a new directory as you would a new folder[2]. Depending upon your computing platform you will probably need to change to your new folder; the `cd` within MATLAB allows you to change directories. In MATLAB6 you can use the `...` symbol at the top of the control environment to change the working directory and this is displayed to the left of this symbol. It is also possible to access files from other directories by augmenting MATLAB's search path. Again, this is platform-specific; the following works on our Unix platforms

```
path(path,'/home/sro/MyMatlabFiles')
```

On a Macintosh you can set the path by choosing `Set Path` from the MATLAB `File` menu. With the search paths set it is possible to create and manage a central resource of user written library functions which can be accessed from any directory on your computer.

One of the most common sources of problems for the novice programmer occurs when the wrong script is being run or when the computer cannot find the program you have just entered. This is usually because the file isn't in the correct directory, or is misnamed. You can check that a file is the correct

[2] Under MacOS X you can use the standard Unix commands given above.

one by using the command `type`. The syntax for this command is simply `type file1`; this will produce a listing of the MATLAB file `file1.m`. If the text is not what you are expecting you may well be using a filename which clashes with an existing MATLAB command. In order to see which file you are looking at, type the command `which file1`. This will tell you the full pathname of the listed files which can be compared with the current path by typing `pwd`. You can also list the files in the current directory by typing `dir` or alternatively all the available MATLAB files can be listed by using `what`: for more details see `help what`.

**Example 2.2** *If we create a MATLAB file called `power.m` using the editor it can be saved in the current directory: however the code will not work. The reason for this can be seen by typing `which power` which produces the output*

```
>> which power
power is a built-in function.
```

*So MATLAB will try to run the built-in function.*

### 2.1.1 Functions

In the previous sections we wrote codes which could be legitimately run at the MATLAB prompt. We now discuss the important class of codes which actually act as functions. These codes take inputs and return outputs. We shall start with a very simple example:

```
function [output] = xsq(input)
output = input.^2;
```

which we will save as `xsq.m`. It is important that we get the syntax for functions correct so we'll go through this example in some detail.

- The first line of `xsq.m` tells us this is a function called `xsq` which takes an input called `input` and returns a value called `output`. The input is contained in round brackets, whereas the output is contained within square brackets. It is crucial for good practice that the name of the function `xsq` corresponds to the name of the file `xsq.m` (without the `.m` extension), although there is some flexibility here.

- The second line of this function actually performs the calculation, in this case squaring the value of the input, and storing this result in the variable `output`. Notice that the function uses dot arithmetic `.^` so that this function will work with both vector and matrix inputs (performing the operation element by element). Notice also that we have suppressed the output of the calculation by using a semicolon; in general all communication between a function subroutine and the main calling program should be done through the input and output.

Having written our function we can now access it from within MATLAB. Consider the following:

```
x = 1:10;
y = xsq(x)
```

Here we have initialised the vector `x` to contain the first ten integers. We call the function in the second line and assign the result, in this case `x.^2`, to the variable `y`. Notice that we have called the function with the argument `x` and not something called `input`. The variables `input` and `output` are local variables that are used by the function; they are not accessible to the general MATLAB workspace[3]. When the function is run it merely looks at what is given as the argument. It is therefore important the function has the correct input; in our example scalar, vector or matrix inputs are allowed. In other cases, if the function expects an argument of a certain type then it must be given one otherwise an error will occur (which will not always be reported by MATLAB). Of course it is not possible to call the function just using `xsq` since the code cannot possibly know what the input is. MATLAB will return an error stating that the `Input argument 'input' is undefined`. As noted above this function can also be used for scalars, for instance `xsq(2)` returns the value 4, and for vectors

```
>> A = [1 2 3 4 5 6];
>> y = xsq(A)

y =

     1     4     9    16    25    36
```

Functions can also take multiple inputs and give multiple outputs. Consider the following examples:

**Example 2.3** *Suppose we want to plot contours of a function of two variables* $z = x^2 + y^2$. *We can use the code*

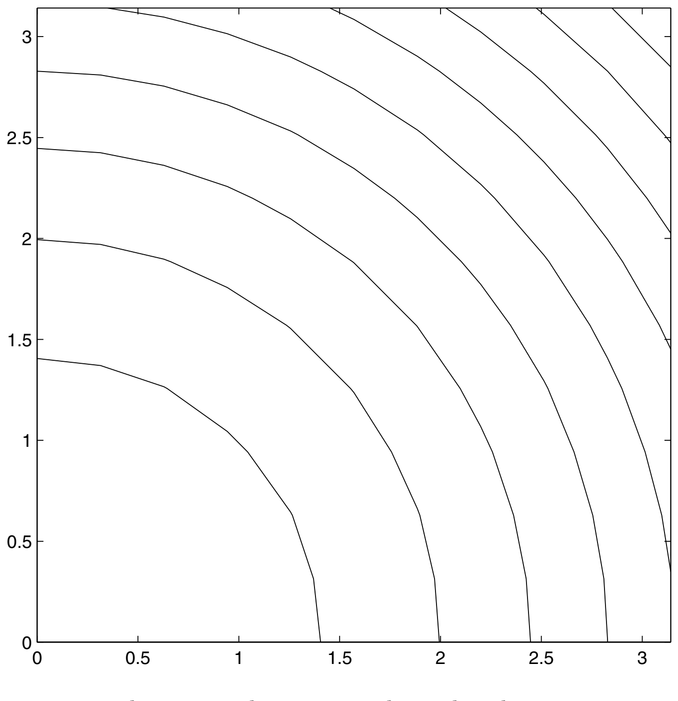
[3] You can find out what variables are in use by MATLAB by typing `who`, which lists all variables in use, or `whos`, which lists all variables along with their size and type.

```
function [output] = func(x,y)
output = x.^2 + y.^2;
```

*which should be saved in the file* `func.m`*. The first line indicates this is a function which has two inputs* `x` *and* `y`*, and returns a single output* `output`*. The next line calculates the function $x^2 + y^2$; again we have used dot arithmetic to allow for the possibility of vector or matrix arguments. For the calculation to be valid the vectors* `x` *and* `y` *must have the same size. To plot the contours (that is the level curves) of the function $z = x^2 + y^2$ we would proceed as follows:*

```
x = 0.0:pi/10:pi;
y = x;
[X,Y] = meshgrid(x,y);
f = func(X,Y);
contour(X,Y,f)
axis([0 pi 0 pi])
axis equal
```

*This gives us the plot*



*For the moment we do not need to worry about the plotting commands used to display the curves: we will return to plotting in more detail later in this chapter.*

Important Point

It is just as important that the function receives the correct number of inputs as it is that they are of the correct type. For example, we cannot call our function using `func` or `func(1)` (or with arguments of different size). In all these cases MATLAB will give an error message.

**Example 2.4** *Suppose we now want to construct the squares and cubes of the elements of a vector. We can use the code*

```
function [sq,cub] = xpowers(input)
sq = input.^2;
cub = input.^3;
```

*Again, the first line defines the function* ***`xpowers`****, which has two outputs* ***`sq`*** *and* ***`cub`*** *and one input. The second and third lines calculates the values of* ***`input`*** *squared and cubed. This function file must be saved as* ***`xpowers.m`*** *and it can be called as follows:*

```
x = 1:10;
[xsq,xcub] = xpowers(x);
```

*This gives*

```
>> xsq

xsq =

     1     4     9    16    25    36    49    64    81   100

>> xcub

xcub =

  Columns 1 through 6

           1           8          27          64         125         216

  Columns 7 through 10

         343         512         729        1000
```

*The output is two row vectors, one containing the values of the first ten integers squared and the second the values of the first ten integers cubed. Notice that when the function is called we must know what form of output we expect,* *whether it be a scalar, a vector or a matrix. The expected outputs should be placed within square brackets.*

**Example 2.5** *As you might expect a function can have multiple inputs and outputs:*

```
function [out1,out2] = multi(in1,in2,in3)
out1 = in1 + max(in2,in3);
out2 = (in1 + in2 + in3)/3;
```

*which should be saved as `multi.m`. This gives us a function called `multi` that takes three inputs `in1`, `in2` and `in3` and returns two outputs `out1` and `out2`. The first output is the sum of the first input and the maximum of the latter two, calculated using the MATLAB intrinsic function `max`. The second output is simply the arithmetic mean (the average) of the inputs. We can call this function in the following way*

```
x1 = 2; x2 = 3; x3 = 5;
[y1,y2] = multi(x1,x2,x3);
y1, y2
```

*For this example we obtain `y1=7` and `y2=3.3333`.*

The input and output of a function do not have to be the same size (although in most cases they will be). Consider the following example:

**Example 2.6** *Consider a code which returns a scalar result from a vector input. For example*

```
function [output] = sumsq(x)
output = sum(x.^2);
```

*Our function `sumsq` takes a vector (or potentially a scalar) as an input and returns the sum of the squares of the elements of the vector. The MATLAB intrinsic function `sum` calculates the sum of its vector argument. For instance*

```
x = [1 2 4 5 6];
y = sumsq(x)
```

*sets `y` equal to the scalar* $1^2 + 2^2 + 4^2 + 5^2 + 6^2 = 82$.

### 2.1.2 Brief Aside

For those of you familiar with matrices we pause here and note that the command in the previous example will also work with matrices:

```
>> A=[1 2 3; 4 5 6];
>> sumsq(A)

ans =

    17    29    45
```

The command has squared (and summed) the elements of the matrix `A`, which is two-by-three. This has exploited the property that the `sum` command sums the columns of a matrix. If we want to sum the rows of a matrix we use `sum(A,2)`, so that we have

```
>> sum(A,1)  % which is equivalent to sum(A)

ans =

     5     7     9

>> sum(A,2)

ans =

     6
    15
```

Notice that the answers are the shape we would expect: the first is a row vector whereas the second is a column vector. Many of the MATLAB commands we shall meet in this text, in general those commands which reduce the dimension of the input object by one, can operate along the rows or the columns; which is specified using an additional argument.

Many of MATLAB's intrinsic commands work in the same way and so care is needed to ensure that the correct number and form of inputs to functions are used.

## 2.2 Plotting Simple Functions

One of the most powerful elements of MATLAB is its excellent plotting facilities which allow us to easily and rapidly visualise the results of calculations. We have already met some examples of plotting (the line graph plotting of the functions and the contour plot on page 32). We pause here to try some examples of the plotting facilities available within MATLAB. We start with the simplest command `plot` and use this as an opportunity to revisit the ways in which functions can be initialised. We start with initialising an array, in this case `x`

```
x = 0:pi/20:pi;
```

which as we know sets up a vector whose elements are

$$\left(0, \frac{\pi}{20}, 2\frac{\pi}{20}, \cdots, 19\frac{\pi}{20}, \pi\right),$$

(that is, a vector whose elements range from 0 to $\pi$ in steps of $\pi/20$). This array is of size one-by-twenty one, which can be confirmed by using the command `size(x)` or, if we know it is a one-dimensional array as is the case here, by using the command `length(x)` (in general `length` gives the maximum of the dimensions of a matrix). We can plot simple functions, for instance `plot(x,sin(x))` or more complicated examples such as

```
plot(x,sin(3*x),x,x.^2.*sin(3*x)+cos(4*x))
```

(this plots $\sin 3x$ and $x^2 \sin 3x + \cos 4x$). Try these examples out for yourself.

Now we have an array `x` we begin using it as an argument to other functions. We start with calculating the point on a straight line $3x - 1$ using

```
y = 3*x-1;
```

Again `size(y)`, or `length(y)`, confirms that this array has the same dimensions as the vector `x`. We can plot $y$ versus $x$ using the command `plot(x,y)` to produce a straight line (in the default colour blue). You can change the colour or style of the line, or force the individual data points to be plotted, using a third argument for the plot command; more details will be given later on page 45.

Proceeding to polynomials, the code

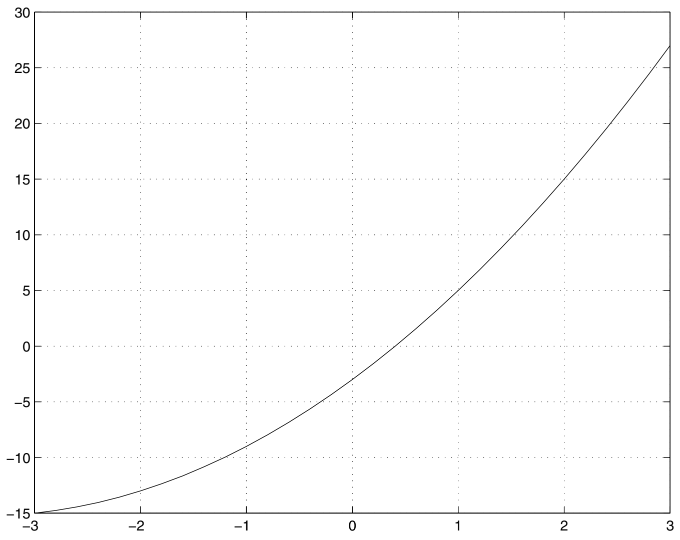
```
y = x.^2+3;
```

produces a vector `y` whose elements are given by the quadratic $x^2 + 3$. Notice that by using the dot before the operator (here exponentiation ^) we are performing the operation element by element on the array `x`.

**Example 2.7** *To plot the quadratic* $x^2 + 7x - 3$ *from* $x$ *equals* $-3$ *to* $3$ *in steps of* $0.2$ *we use the code*

```
x = -3:0.2:3;
y = x.^2+7*x-3;
grid on
plot(x,y)
```

*The resulting plot is given below*



We have given the plot a grid by using the command `grid on`; this can be removed using the command `grid off`.

MATLAB provides an excellent computing environment for producing results which can be viewed quickly and easily. This is essential when we come to analyse the results of our calculations, a task that is usually necessary in order to obtain useful information from an otherwise mathematical calculation. MATLAB is capable of producing very intricate and clear plots, as the following example illustrates.

**Example 2.8** *Consider the following code:*

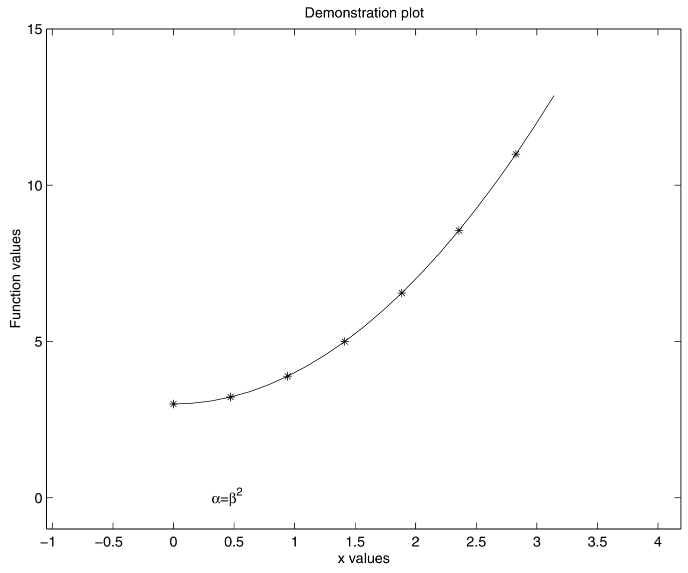
```
x = 0:pi/20:pi;
n = length(x);
r = 1:n/7:n;
y = x.^2+3;
plot(x,y,'b',x(r),y(r),'r*')
axis([-pi/3 pi+pi/3 -1 15])
xlabel('x values')
ylabel('Function values')
title('Demonstration plot')
text(pi/10,0,'\alpha=\beta^2')
```

*If we dissect this piece of code we see the first line initializes the vector* `x`*. The second and third lines pick out every third integer value between 1 and* `length(x)`*. The fourth line computes the values of the quadratic $y = x^2 + 3$ at the points in the vector* `x` *(using dot arithmetic to achieve this). We then reach the* `plot` *command. Here we are telling MATLAB to plot the curve $y$ versus $x$ and to colour the plot blue (using the flag* `'b'`*). The second part of the* `plot` *command tells MATLAB to plot every third point on the curve (here represented by* `x(r)` *and* `y(r)`*) as points which are labelled with a red asterisk using* `'r*'`*. The commands directly following the* `plot` *command are used to manipulate the final look of the plot. The command* `axis` *sets the start and end points of the horizontal (from $-\pi/3$ to $\pi + \pi/3$) and then the vertical axis (from $-1$ to $15$). The commands* `xlabel` *and* `ylabel` *add labels to the horizontal and vertical axes and the command* `title`*, not surprisingly adds the title to the figure. The command* `text` *allows the user to add text to the figure at a coordinate specified within the units of the plot. The arguments of these commands include a string which starts and ends with a quotation mark. The text can include many characters but here we have included Greek letters, using the LaTeX[4] construction \alpha for $\alpha$ and \beta for $\beta$. The properties of a figure can be edited using the drop down menus on its window. This includes being able to increase the size of the characters used in labels. By changing the line which sets the title to*

```
title('Demonstration plot','FontSize',24)
```

*we see quite a dramatic change in the size of the characters in the title.*

[4] LaTeX is a language that is almost universally used for typesetting mathematics. It is available for almost all computing platforms and operating systems. Further details can be found by consulting any good LaTeX text.

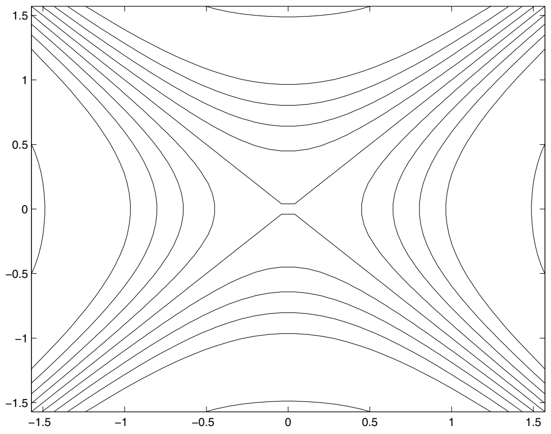


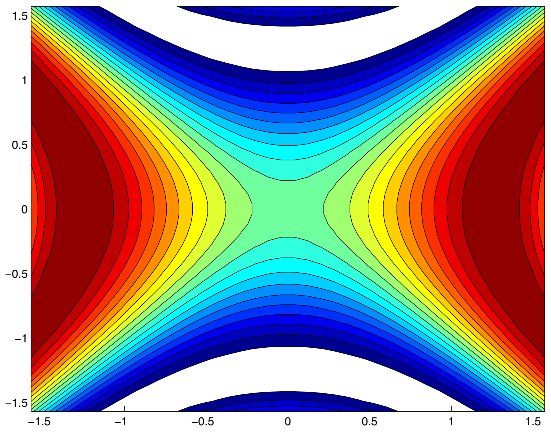
There are a wide variety of other plotting options available. For instance `loglog(x,y)` produces a log-log plot. Similarly `semilogx` and `semilogy` produces a log plot for the $x$ and $y$-axis, respectively. You should also be aware of the commands `clf` which clears the current figure and `hold` which holds the current figure. We will explore the use of these commands in the tasks at the end of this section.

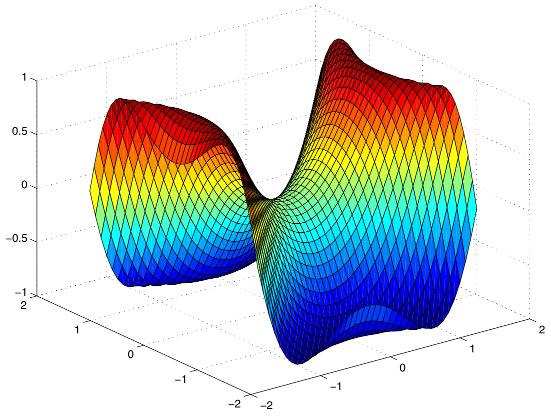
One of the excellent features of MATLAB is the way in which it handles two and three-dimensional graphics. Although we will have little need to exploit the power of MATLAB's graphical rendering we should be aware of the basic commands. Examples serve to highlight some of the many possibilities:

```
x = linspace(-pi/2,pi/2,40);
y = x;
[X,Y] = meshgrid(x,y);
f = sin(X.^2-Y.^2);
figure(1)
contour(X,Y,f)
figure(2)
contourf(X,Y,f,20)
figure(3)
surf(X,Y,f)
```

This gives the three figures







These figures are, respectively, a contour plot (created using `contour(X,Y,f)`), a filled contour plot with 20 contour levels (created using `contourf(X,Y,f,20)` and a surface plot (created using `surf(X,Y,f)`). The function `f` is plotted on a grid generated using the command `meshgrid(x,y)`. As the name suggests meshgrid sets up a grid of points by generating copies of the x and y values; see help meshgrid for details of this command. To further explore the potential of MATLAB's plotting facilities see the MATLAB demo topics Visualization and Language/Graphics.

### 2.2.1 Evaluating Polynomials and Plotting Curves

We return to the topic of functions by first constructing a simple program to evaluate a quadratic. We shall start with a code to generate the value of a specific quadratic at a specific point:

```
x = 3;
y = x^2+x+1;
disp(y)
```

This code sets x equal to 3, calculates $x^2 + x + 1$ and then displays the answer. We now expand the above code so that the user can enter the point and the coefficients of the quadratic. In general suppose we have the general quadratic

$$y = a_2x^2 + a_1x + a_0.$$

Firstly we need to define the quadratic and this is done by fixing the three coefficients $a_0, a_1$ and $a_2$. This can be done using the following script, which we will call quadratic.m

```
% quadratic.m
% This program evaluates a quadratic
% at a certain value of x
% The coefficients are stored in a2, a1 and a0.
%                                                    SRO & JPD
%
str = 'Please enter the ';
a2 = input([str 'coefficient of x squared: ']);
a1 = input([str 'coefficient of x: ']);
a0 = input([str 'constant term: ']);
x = input([str 'value of x: ']);
y = a2*x*x+a1*x+a0;
% Now display the result
disp(['Polynomial value is: ' num2str(y)])
```

Again we have a code where the first few lines of the code start with a percent sign %. MATLAB treats anything coming after a % sign as a comment.

Comments at the start of a code have a special significance in that they are used by MATLAB to provide the entry for the help manual for that particular script. The manual entry can then be accessed by typing `help quadratic` at the MATLAB prompt to produce

```
quadratic.m
This programme evaluates a quadratic
at a certain value of x
The coefficients are stored in a2, a1 and a0.
                                        SRO & JPD
```

Notice the manual entry terminates once MATLAB reaches a line in the file `quadratic.m` in which the first character is **not** a `%` sign. The line later on in the code which starts with a percent sign is again a comment. In this case we have inserted this line to provide information to the user about the calculation which is to be performed; in this case display the result of our calculation. Judicious use of comments throughout a code makes it readable by a person who may not be fully conversant with the precise details of the calculations to be performed. It is good programming practice to insert text as a manual entry at the start of any code as well as including comments on particular aspects of the calculation which may not be transparent to the user of the code. Note it is also possible to obtain a complete listing of the code from within MATLAB by typing `type quadratic` at the prompt.

Back to our code. The first three non-comment lines use the `input` command to prompt the user to enter the values of the constants `a2`, `a1` and `a0`. As we saw earlier the `input` command takes an argument in the form of a character string, contained between quotes, that acts as the prompt which appears on the screen. Here we have introduced a variable `str` which is equal to `'Please enter the '`, a phrase which is common to each of the input statements. The argument to `input` is then a vector containing strings. Notice in this example we have appended to the prompt a colon followed by a space so that when the user types the values, rather than them being flush against the final character, there is a space. This is not necessary but simply a style convention we will usually adopt.

For the moment we shall assume the user responds to the prompts by entering values which are reasonable. The user is then prompted for the value of $x$ at which they wish to evaluate the polynomial. The following line actually does the calculation of evaluating the polynomial; at this stage we have used the fact that $x^2 = x \times x$, but of course we could use `a2*x^2+a1*x+a0`. There are many ways of writing polynomials, for instance this could have been written recursively as

$$a_0 + x(a_1 + xa_2). \tag{2.1}$$

This general recursive form goes under the name of Horner's method. With the function evaluated the answer is displayed, using the function `num2str` ('number-to-string') which takes as input a number and returns a character string which is then appended to the phrase `'Polynomial value is: '`. The concatenation (joining together) of strings is achieved by writing them as elements in a row vector and then using the `disp` function to write them to the screen.

There is no obvious reason why we couldn't type the commands given above at the MATLAB prompt and enter all the values required for the calculation to proceed. However, it is preferable to carry out the calculation by simply typing the command `quadratic`; this also allows us to have access to a general program which calculates the value of a quadratic. An even better approach is to write a user callable function. We will use this example to build up the level of complexity of the function until we have one which is as general as we can possibly make it.

Firstly let's suppose the values of the coefficients in the quadratic are known *a priori*[5]. For example, suppose we want to evaluate the quadratic $y = 3x^2 + 2x + 1$. We could then use the function

```
%
% evaluate_poly.m
%
function [value] = evaluate_poly(x)
value = 3*x.^2+2*x+1;
```

Here we have again used dot arithmetic in the construction `x.^2`, thus making this function able to take both a scalar or vector (or even a matrix) argument. As we have seen earlier if this function is called with a vector, then a vector is returned as output.

Now we can use the function `evaluate_poly`, in the form `evaluate_poly(2)` or `x = 2;y = evaluate_poly(x)`. Note within MATLAB we are able to call a function with a variety of different inputs; whether this is valid depends upon the structure of the function[6]. With this in mind we can now use our function to generate a vector containing the values of the polynomial at a number of $x$ points and plot the result using

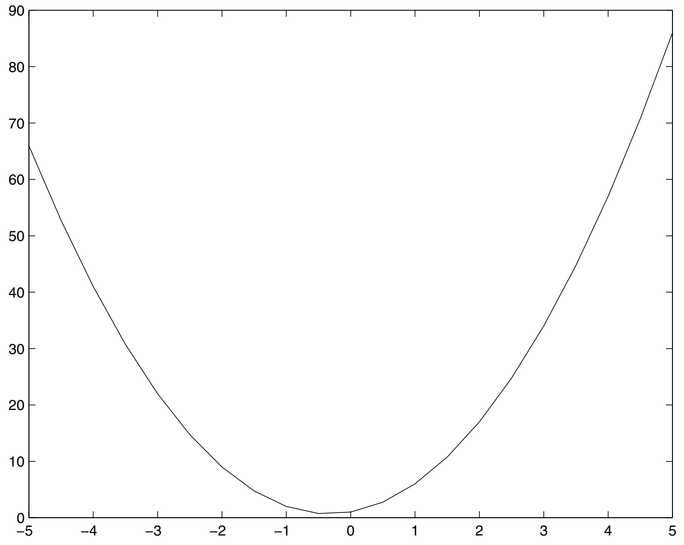
```
>> x = -5:0.5:5;
```

[5] A very similar command already exists in MATLAB, namely `polyval`, but this function is developed to demonstrate the construction.

[6] This is similar to the idea of overloading which is intrinsic to object orientated languages like C++ and Java. However, in those languages a different function is called depending on the type of input, whereas here it is one function which apparently deals with all the cases.

```
>> y = evaluate_poly(x);
>> plot(x,y)
```

which gives



### 2.2.2 More on Plotting

At this point we briefly return to the problem of generating plots using MATLAB. The simple code given above first sets up a vector which runs from $x = -5$ to $x = 5$ in steps of 0.5. The value of the quadratic at each point is then calculated, using our newly defined function. The final `plot(x,y)` command 'simply' draws a line through these points. The way the plot is constructed can best be seen by making the $x$-grid coarser (that is, with less points); we then find the plot is constructed by drawing a straight line through each pair of points. MATLAB is slightly more subtle than this in that it checks whether there is a window open ready for plotting, and if not it opens one. The axes are automatically rescaled and tick marks are drawn. The default settings are generally fine, although at some points you may want to look at certain regions of the plot. At the moment we will not worry about the details of how this is achieved[7]. In general we shall introduce the features of the plotting package as we need them. Here we extend our plotting capability by considering the impact of a

[7] Try the commands `h = gca`. This returns what MATLAB calls a 'handle' to the current axis, and then the function `get(h)` lists all the properties of the axis. There is also a command `gcf` which returns a handle to the entire figure.

third argument of the `plot` command, such as in `plot(x,y,'r.')`. The string which is passed as the third argument conveys information as to how the plot should be presented: here the first character `r` is the colour (red) and the second is the symbol (in this case a dot `.`) to be used at each point (as opposed to a line joining the points). The colour options are

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

and the choice of symbols are

| | | | |
|---|---|---|---|
| . | point | v | triangle (down) |
| o | circle | X | triangle (up) |
| x | x-mark | < | triangle (left) |
| + | plus | > | triangle (right) |
| * | star | p | pentagram |
| s | square | h | hexagram |
| d | diamond | | |

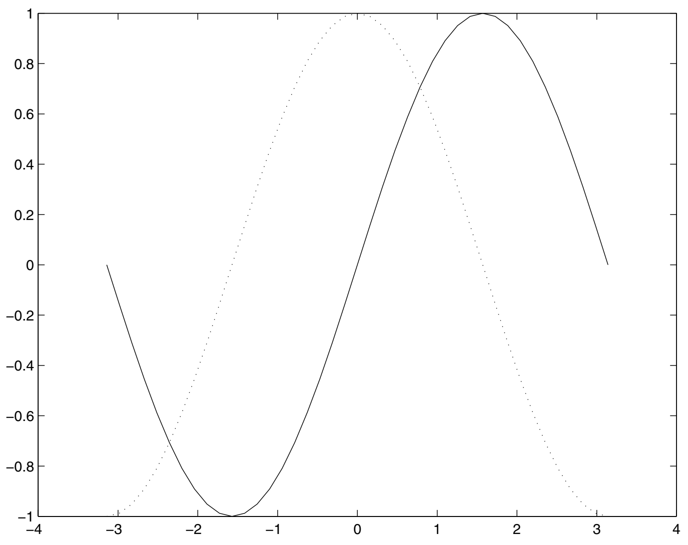
It is also possible to control the line style. Instead of using a symbol, as in the previous command, we can draw the line using one of the following options:

| | |
|---|---|
| - | solid |
| : | dotted |
| -. | dashdot |
| - - | dashed |

As we saw earlier (on page 38) we can also plot more than one curve on the same graph. One simple way of achieving this is through the following code

```
>>x = -pi:pi/20:pi;
>>plot(x,sin(x),'r-',x,cos(x),'b:')
```

This generates the plot



from which the command syntax can be deduced; a plot of $\sin(x)$ versus $x$ using a solid red line and a plot of $\cos(x)$ versus $x$ using a blue dotted line (if the plot were to be viewed in colour). This could have been achieved in another way

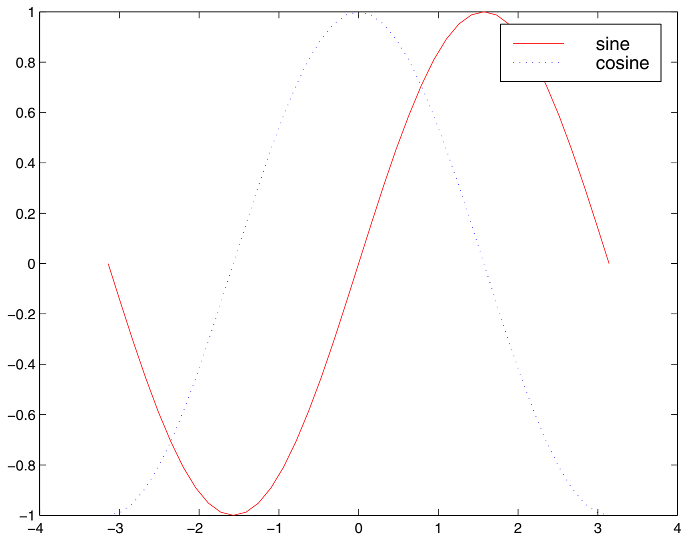
```
x = -pi:pi/20:pi;
clf
plot(x,sin(x),'r-')
hold on
plot(x,cos(x),'b:')
hold off
```

As you might infer from this, using the command `plot` clears the current plot and replaces it with the new figure, unless the figure is being "held". So here we plot $\sin(x)$, hold this current figure using `hold on` and then plot $\cos(x)$. The current figure is released using `hold off`.

We could add a legend to this plot to help us identify which line is which. This is accomplished with the command

```
legend('sine','cosine')
```

which now gives



In this case the legend has been placed in the top right corner, however its location can be changed, see `help legend` for details.

We now return to our discussion of functions and consider in more detail functions which can be called with more than one argument and return as many arguments as we wish. In constructing functions we do need to take some care with how our outputs are returned; we have already seen that if the function is called with a vector then the “output” is, in general, a vector. However, there is no reason why there should not be two (or more) outputs returned. Consider the following modification to the function `evaluate_poly` from page 43

```
%
% evaluate_poly2.m
%
function [f, fprime] = evaluate_poly2(x)
f = 3*x.^2+2*x+1;
fprime = 6*x+2;
```

This MATLAB function calculates the values of the polynomial and its derivative. This could be called using the sequence of commands

```
x = -5:0.5:5;
[func,dfunc] = evaluate_poly2(x);
```

We can further generalise our code by passing the coefficients of the quadratic to the function, either as individual values or a vector. The code then becomes

```
%
% evaluate_poly3.m
%
function [f,fprime] = evaluate_poly3(a,x)
f = a(1)*x.^2+a(2)*x+a(3);
fprime = 2*a(1)*x+a(2);
```

which can be called using

```
x = -5:0.5:5;
a = [1 2 1];
[f,fp] = evaluate_poly3(a,x);
```

Before proceeding, we make one final comment regarding the general form for function files. The only communication between the function and the calling program is through the input and output variables; the function should not prompt a user for input nor should it echo the result of any calculation. This is not a necessary constraint on MATLAB functions but more a useful programming convention which is worth following.

This fairly simple routine `evaluate_poly3(x)` was chosen to demonstrate some of the particulars of MATLAB function files because MATLAB has a built-in function, called `polyval`, which also evaluates polynomials. The extract from `help polyval` yields the information

```
Y = POLYVAL(P,X), when P is a vector of length N+1 whose elements
    are the coefficients of a polynomial, is the value of the
    polynomial evaluated at X.

        Y = P(1)*X^N + P(2)*X^(N-1) + ... + P(N)*X + P(N+1)
```

Care needs to be taken with the order in which the coefficients of the polynomial are presented. In fact these are in the same order as in our routine. This example provides a good opportunity to investigate the full code for this built-in function; the code listing can be obtained using `type polyval`. Compare this with our three-line function `evaluate_poly3(x)`.

**Example 2.9** *We can use* `polyval` *to evaluate polynomials. We will use it to test a hypothesis that the values of* $n^2 + n + 41$, *where* $n$ *is an integer, are prime (at least for the first few integer values of* $n$*: obviously this is not so for* $n = 41$*). Consider*

```
p = [1 1 41];
x = 1:40;
f = polyval(p,x);
isprime(f)
```

*the function `isprime(f)` returns a value of* 1 *if the element of `f` is a prime and* 0 *if it is not. The result of our calculation is a string of* 39 *ones demonstrating that all but the value for* $n = 40$ *are prime* ($40^2+40+41 = 40\times(40+1)+41 = 41\times(40+1) = 41^2$*). We note that the value of quadratic* $n^2 - n + 41$ *is also prime for all integers up to and including* $n = 40$.

## 2.3 Functions of Functions

One MATLAB command which provides us with considerable freedom in writing versatile code is the command `feval` which loosely translates as *function evaluation.* The simplest use for this command is

```
    y= feval('sin',x);
```

evaluate the function `sin` at `x`. This is equivalent to `sin(x)`. In general the arguments for `feval` are the name of the function, which must be either a MATLAB built-in function or a user defined function, contained between quotes, and the value (or values in the case of a vector) at which the function is to be evaluated. The utility of `feval` is that it allows us to use function names as arguments. By way of introduction we provide an example. Consider the function

$$h(x) = 2\sin^2 x + 3\sin x - 1.$$

One way of writing code that would evaluate this function is

```
function [h] = fnc(x)
h = 2*sin(x).*sin(x)+3*sin(x)-1
```

However we can recognise this function as a composition of two functions $g(x) = \sin(x)$ and $f(x) = 2x^2 + 3x - 1$ so that $h(x) = f(g(x))$. We can easily write a function file `f.m` to evaluate $f(x)$

```
function [y] = f(x)
y = 2*x.^2+3*x-1;
```

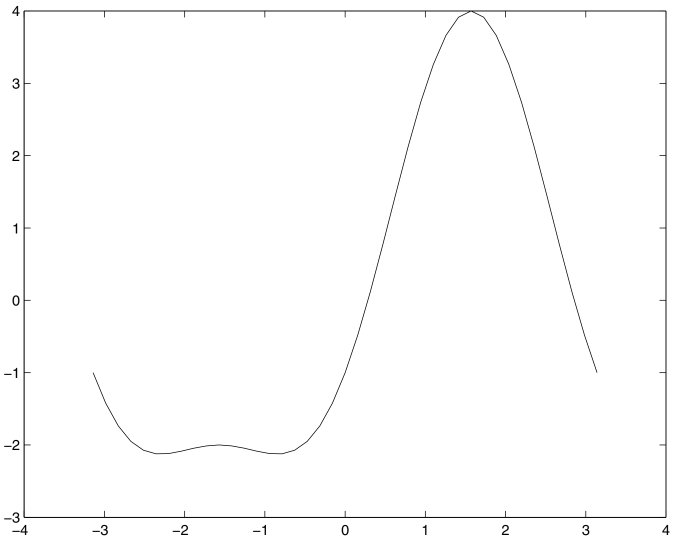
(using dot arithmetic to allow for the possibility of vector arguments `x`). To use this in calculating the value of the composite function $h(x)$ we need to be able to pass the function name to `f` as an argument. We can do this with the following modification to our code

```
function [y] = f(fname,x)
z = feval(fname,x);
y = 2*z.^2+3*z-1;
```

Calculating the function $h(x)$ is now as simple as

```
x = -pi:pi/20:pi;
y = f('sin',x);
plot(x,y)
```

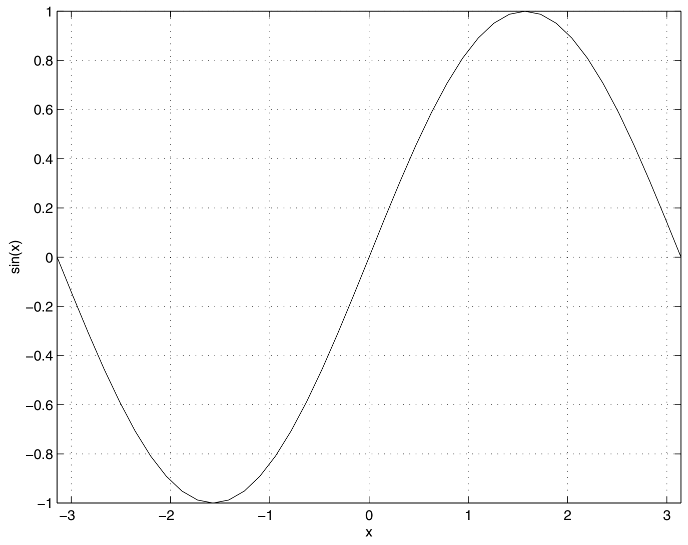
This gives a plot of the function $h(x)$ as



A more useful example is given by the function `plotf`

```
function plotf(fname,x)
y = feval(fname,x)
plot(x,y)
grid on
axis([min(x) max(x) min(y) max(y)])
xlabel('x')
ylabel([fname '(x)'])
```

This function takes as input a function name and a vector `x` and produces a labelled plot. Notice although there are two inputs for this function there are no values output; the figure is the output.



The command `feval` has a wide range of uses, some of which we will exploit later throughout the text.

## 2.4 Errors

### 2.4.1 Numerical Errors

We mentioned right at the start of this text that numerical methods are usually not exact (that is, they are approximate methods). In fact it is very hard to get computers to perform exact calculations. If we add (or subtract) integers then a computer can be expected to get the exact answer, but even this operation has its limits. Once we try to perform the division operation we run into trouble. We are happy with the fact that one divided by three is a third, which we can write as $1/3$, but if we need to store this on a computer we run into difficulties.

For the moment let's suppose we have a virtual computer that works in decimal. By this we mean that any number which can be written as a terminating decimal (that is, one which stops), can be stored 'exactly' (provided we do not require too many digits). We know that a third does not have a finite representation, so there is no way we can store this number in our imaginary computer other than by truncating the sequence. We would use the notation $1/3 \approx 0.3333$. Obviously the more three's we retain the more accurate the answer.

Almost all numerical schemes are prone to some kind of error. It is important to bear this in mind and understand the possible extent of the error. Intrinsic to this has to be the knowledge of how much we can trust the results which are produced. Errors can be expressed as two basic types:

**Absolute error:** This is the difference between the exact answer and the numerical answer.

**Relative error:** This is the absolute error divided by the size of the answer (either the numerical one or the exact one), which is often converted to a percentage.

(In each case we take the modulus of the quantity.)

**Example 2.10** *Suppose an error of £1 is made in a financial calculation of interest on £5 and on £1,000,000. In each case the absolute error is £1, whereas the relative errors are 20% and 0.0001% respectively.*

**Example 2.11** *Suppose a relative error of 20% is made in the above interest calculations on £5 and £1,000,000. The corresponding absolute errors are £1 and £200,000.*

**Example 2.12** *Estimate the error associated with taking 1.6 to be a root of the equation* $x^2 - x - 1 = 0$.

*The exact values for the roots are* $(1 \pm \sqrt{5})/2$ *(let us take the positive root). As such the absolute error is*

$$\left| \frac{1+\sqrt{5}}{2} - 1.6 \right| \approx 0.01803398874989$$

*and the relative error is the absolute error divided by the value* 1.6 *(or alternatively the exact root) which is approximately equal to* 0.01127124296868 *or 1.127%.*

*We could also substitute* $x = 1.6$ *into the equation to see how wrong it is:* $1.6^2 - 1.6 - 1 = -0.04$. *Although it is difficult to understand how this can be used, it is often the only option (particularly if the exact answer cannot be found).*

This method can be used to determine roots of a function. We now discuss an example which uses this technique:

**Example 2.13** *Determine a value of $x$ such that*

$$f(x) = x^2 + 4x = 40.$$

*We start by guessing that $x = 6$ is the root we require:*

$x = 6,\ f(6) = 60 > 40$ *which is too big, try $x = 5$.*

$x = 5,\ f(5) = 45 > 40$ *which is still too big, try $x = 4$.*

$x = 4,\ f(4) = 32 < 40$ *now this is too small, so we shall try $x = 4.5$.*

$x = 4.5,\ f(4.5) = 38.25 < 40$ *a bit too small, try $x = 4.75$*

$x = 4.75,\ f(4.75) = 41.5625 > 40$ *a bit too big, back down again to $x = 4.625$*

$x = 4.625,\ f(4.625) = 39.890625 < 40$ *a bit too small, back up again to $x = (4.75 + 4.625)/2$*

$x = 4.6875,\ f(4.6875) = 40.72265625 > 40$ *and we can continue this process.*

*Here we have just moved around to try to find the value of $x$ such that $f(x) = 40$, but we could have done this in a systematic manner (actually using the size of the errors).*

It appears that in order to work out the error one needs to know the exact answer in which case the whole idea of an error would seem to be superfluous. In fact when we focus on errors we are merely trying to work out their size (or magnitude) rather than their actual numerical value.

Before we move on we revisit the MATLAB variable `eps`. This is defined as the smallest positive number such that `1+eps` is different from `1`. Consider the calculations:

```
x1 = 1+eps;
y1 = x1-1
x2 = 1+eps/2;
y2 = (x2-1)*2
```

In both cases using simple algebra you would expect to get the same answer, namely `eps`; but in fact `y2` is equal to zero. This is because MATLAB cannot distinguish between `1` and `1+eps/2`. The quantity `eps` is very useful, especially when it comes to testing routines.

**Example 2.14** *Calculate the absolute errors associated with the following calculations:*

```
sin(15*pi)
(sqrt(2))^2
1000*0.001
1e10*1e-10
```

*To calculate the absolute errors we need to know the exact answers which are* $0$, $2$, $1$ *and* $1$ *respectively. We can use the code:*

```
abs(sin(15*pi))
abs((sqrt(2))^2-2)
abs(1000*0.001-1)
abs(1e10*1e-10-1)
```

*(notice in the last case the exponent form of the number takes precedence in the calculation: we could of course make sure of this using brackets). The errors are* $10^{-15}$, $10^{-16}$ *and zero (in the last two cases).*

### 2.4.2 User Error

The elimination of user error is critical in achieving accurate results. In practice user error is usually more critical than numerical errors, and after some practise their identification and elimination becomes easier. User error is avoidable, whereas many numerical errors are intrinsic to the numerical techniques or the computational machinery being employed in the calculation.

The most severe user errors will often cause MATLAB to generate error messages; these can help us to understand and identify where in our code we have made a mistake. Once all the syntax errors have been eliminated within a code (that is MATLAB is prepared to run the code), the next level of errors are harder to find. These are usually due to:

1. *An incorrect use of variable names.*
   This may be due to a typographical error which has resulted in changes during the coding.

2. *An incorrect use of operators.*
   The most common instance of this error occurs with dot arithmetic.

3. *Syntactical errors which still produce feasible MATLAB code.*
   For instance in order to evaluate the expression $\cos x$, we should use `cos(x)`: unfortunately the expression `cos x` also yields an answer (which is incor-

rect); somewhat bizarrely `cos pi` yields a row vector (which has the cosines of the ASCII values of the letters `p` and `i` as elements).

4. *Mathematical errors incorporated into the numerical scheme the code seeks to implement.*
   These usually occur where the requested calculation is viable but incorrect (see the example on page 181).

5. *Logical errors in the algorithm.*
   This is where an error has occurred during the coding and we find we are simply working out what is a wrong answer.

Avoiding all of these errors is something which only comes with practice and usually after quite a lot of frustration. The debugging of any program is difficult, fortunately in MATLAB we have the luxury of being able to access variables straight away. This allows us to check results at any stage. Simply removing the semicolons from a code will cause the results to be printed. Although this is not recommended for codes which will produce a deluge of results, it can be very useful.

In Task 2.10 at the end of this chapter and similarly in other chapters[8], we have included some codes with deliberate errors. They also contain commands from future chapters: however most of the errors do not rely on understanding the nuances of these unfamiliar commands. Hopefully by trying to locate the errors in these you should start to develop the debugging skills which will eventually ensure you become a competent programmer.

We shall start with a couple of examples which demonstrate some of the most common sources of problems. Of course it is impossible to predict all possible errors which could be made, but the more you are aware of the better!

**Example 2.15** *This code purports to obtain three numbers $a$, $b$ and $c$ from a user and then produce the results $a+b+c$, $a/((b+c)(c+a))$ and $a/(bc)$.*

[8] These tasks are marked with **(D)**.

```
a = input(' Please entere a )
b = 1nput(' Please enter b )
a = Input ' Please enter c '

v1 = a+ B+d
v2 = a/((b+c)(c+a)
v3 = a/b*c

disp(' a + b + c= ' int2str(v1)])
disp(['v2 = ' num2str v2 ]
disp(['v3 =  num2str(v4) ]);
```

*We shall now work through the lines one-by-one detailing where the errors are and how they can be fixed.*

`a = input(' Please entere a )` *Firstly the single left-hand quote mark should be balanced by a corresponding right-hand quote mark. The command should also end with a semicolon. This will stop the value of* `a` *being printed out after it is entered. The misspelling of the word enter is annoying but will not actually affect the running of the code: however it is worth correcting if only for presentation's sake.*

`b = 1nput(' Please enter b )` *Again we have an imbalance in the single quote mark and the lack of a semicolon. We also have the fact that* `input` *has become* `1nput`*.*

`a = Input ' Please enter c '` *Here we have a capitalised command (which MATLAB will actually point out in its error message – try typing* `Input` *in MATLAB). Again we are missing the semicolon, but the more severe errors here are the lack of brackets around the message and the fact that we are resetting the variable* `a`*. This erases the previous value of* `a` *and also does not set the value of* `c` *as we intended.*

`v1 = a+ B+d` *The spacing is unimportant and is merely a matter of style. Here again we should add a semicolon. The main problem here is that the variable* `b` *has been capitalized to* `B` *and since MATLAB is case sensitive these are treated as different variables. Another new variable* `d` *has also slipped in. This means MATLAB will either complain that the variable has not been initialised or will use a previous value which has nothing to do with this calculation. To avoid this it is a good idea to start the code with* `clear all` *to clear all variables from memory.*

`v2 = a/((b+c)(c+a)` *This command presents a number of errors in the evaluation of the denominator of the fraction. The first of these is the missing* *asterisk between the two factors and secondly we have a unbalanced bracket.*

*v3 = a/b*c* *This has one error but it is one of the most common ones. As mentioned previously this evaluates* $\left(\frac{a}{b}\right)c$*. We need to force the calculation* $bc$ *to be performed first by using brackets. With this (and the previous two commands we might want to elect to add semicolons to stop extra output).*

*disp(' a + b + c= ' int2str(v1)])* *This command is missing a left square bracket to show that we are going to display a vector (with strings as elements). We have used the command* ***int2str*** *which takes an integer and returns a character string. However at no point have we specified that a, b and c are integers. We should instead use the command* ***num2str*** *which converts a general number to a character string.*

*disp(['v2 = ' num2str v2 ]* *This command is missing a round bracket from the end of the expression and a pair of brackets to show that* ***num2str*** *is being applied to the variable* ***v2****.*

*disp(['v3 = num2str(v4) ]);* *Here we are missing a single quote to show that the string has ended (after the equals sign). We have also erroneously introduced a new variable* ***v4*** *which should be* ***v3****. The semicolon on the end of this line is superfluous since the* ***disp*** *command displays the result.*

*The corrected code looks like:*

```
clear all
a = input(' Please enter a ');
b = input(' Please enter b ');
c = input(' Please enter c ');

v1 = a+b+c;
v2 = a/((b+c)*(c+a));
v3 = a/(b*c);

disp([' a + b + c= ' num2str(v1)])
disp([' v2 = ' num2str(v2)])
disp([' v3 = ' num2str(v3)])
```

## 2.5 Tasks

**Task 2.1** *Enter and run the code*

```
a = input('Enter a : ');
b = input('Enter b : ');
res = mod(a,b);
str1 = 'The remainder is ';
str2 = '  when ' ;
str3 = ' is divided by ';
disp([str1 num2str(res) str2 ...
  num2str(a) str3 num2str(b)]);
```

*which should be saved as* `remainders.m`*. Experiment with this code by running it with various values for* `a` *and* `b`*. Make sure you understand how this code works, in particular the* `mod` *command and the* `disp` *command.*

**Task 2.2** *Modify the code in Task 2.1 so it returns the answer to $a^b$ and change the character strings* `str1`*,* `str2` *and* `str3` *so that the format of the answer is 'The answer is $a^b$ when a is raised to the power b'.*

**Task 2.3** *Write a code which takes a variable $x$ and returns the value of $2^x$. Make sure that your code works for variables which are scalars, vectors and matrices.*

**Task 2.4** *By modifying the function* `func.m` *(given earlier on page 32) repeat the example for the functions $x^2 - y^2$ and then $\sin(x + y)$ (extending the range to $[0, 2\pi]$ in the latter case).*

**Task 2.5** *Modify the code* `xpowers.m` *to simultaneously give the values of the functions $\sin x$, $\cos x$ and $\sin^2 x + \cos^2 x$.*

**Task 2.6** *Modify the code* `multi.m` *in Example 2.5 to work out the values of the map:*

$$\begin{aligned} x \mapsto &\quad (x+y)|1 \\ y \mapsto &\quad (x+2y)|1, \end{aligned}$$

*where $(a|b)$ is the remainder when $a$ is divided by $b$ and can be calculated using the MATLAB function* `mod`*. If $b$ is unity this is the fractional part. This new function will take two inputs and return two outputs.*

**Task 2.7** *The code*

```
clear x y
x = -2:0.1:2;
y = 9-x.^2;
plot(x,y)
```

*plots the function $y = 9 - x^2$ for $x \in [-2, 2]$ in steps of $1/10$. Modify the code so the function $y = x^3 + 3x$ is plotted between the same limits and then for $x \in [-4, 6]$ in steps of $1/4$. (If you can't see the current figure, type `figure(1)` which should bring it to the foreground). This code can be run from the prompt or can be entered using `edit` and then saved.*

**Task 2.8** *Consider the quartic $y = x^4 + x^2 + a$. For which values of $a$ does the equation have two real roots?*

**Task 2.9** *Plot on the same graph the functions $f(x) = x + 3$, $g(x) = x^2 + 1$, $f(x)g(x)$ and $f(x)/g(x)$ for the range $x \in [-1, 1]$. (You need to decide how many points to use to get a smooth curve, and whether to set up the vector `x` either using the colon construction or `linspace`). You also need to remember to use dot arithmetic since you are operating on vectors.*

**Task 2.10 (D)** *This task contains codes which are written with deliberate mistakes. You should try to debug the codes so that they actually perform the calculations they are supposed to:*

1. *Perform the calculations*

$$\begin{aligned} x &= 4 \\ x + 2 &= y \\ z &= \frac{1}{y^2 \pi} \end{aligned}$$

```
x=4
x+2=y
z=1/y^2Pi
```

2. *Calculate the sum*

$$\sum_{i=1}^{N} \frac{1}{i} + \frac{1}{(i+2)(i+3)}$$

*where the user inputs $N$.*

```
N=input('Enter N )

for i=1:n
    sum = 1/j + 1/(j+2)*(j+3)
end

disp( ' The answer is ' s])
```

*Make sure the code gives the correct answer, for instance for $N = 1$.*

3. *Calculate the function*

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

*for $x$ from 0 to 1 in steps of 0.1.*

```
x==0.0:0.1:1.0;
f=xcos x/*([x^2+1]*(x+2)
```

4. *Set up the vector 1 3 3 3 5 3 7 3 9*

```
w = ones(9);

w(1) = 1;

for j = 1:4
    w(2j) = 3:
    w(2j+1) = 2j+1:
```

*Make sure that the code returns the correct values of the entries of* `w`*.*

**Task 2.11** *The following codes should be written to produce conversion between speeds in different units.*

*(a) Construct a code which converts a speed in miles per hour to kilometres per hour.*

*(b) Write a code which converts metres per second to miles per hour and use it to determine how fast a sprinter who runs the 100 metres in 10 seconds is travelling in miles per hour (on average).*

*(c) Rewrite your code from part (a) so that it is now a function that takes a single input, the speed in miles per hour, and returns a single output, the speed in kilometres per hour.*

*(d) Now modify the code you wrote in part (b) to determine the sprinter's speed in kilometres per hour by calling the function from the previous part.*

*(***NB***: 1 mile = 1760 yards; 1 yard = 36 inches; 1 inch = 2.54 cm; 1 m = 100 cm)*

**Task 2.12** *The functions $f(x)$ and $g(x)$ are defined by*

$$f(x) = \frac{x}{1+x^2} \text{ and } g(x) = \tan x.$$

*Write MATLAB codes to calculate these functions and plot them on the interval $(-\pi/2, \pi/2)$. Also plot the functions $f(g(x))$ and $g(f(x))$ on this interval.*

**Task 2.13** *Write a code which enables a user to input the coefficients of a quadratic $q(x) = ax^2 + bx + c$ and plot the function $q(x)$ for $x = \sin y$ where $y \in [0, \pi]$.*