Module UFMFJ9-30-1

Engineering Mathematics

MATLAB

Gary Atkinson, Benjamin Drew

September 2018

Department of Engineering Design and Mathematics
University of the West of England, Bristol

Contents

1 Introduction and Basic Concepts **3**
- 1.1 Introduction . . . 3
  - 1.1.1 MATLAB in Engineering Mathematics . . . 3
  - 1.1.2 Course Notes . . . 3
  - 1.1.3 Assessment . . . 4
  - 1.1.4 What is MATLAB? . . . 4
  - 1.1.5 Getting Help . . . 5
- 1.2 Getting Started . . . 5
  - 1.2.1 The MATLAB Environment . . . 5
  - 1.2.2 Getting Help in MATLAB . . . 7
- 1.3 Using MATLAB . . . 7
  - 1.3.1 Basic Calculations . . . 7
  - 1.3.2 Number Formats . . . 10
  - 1.3.3 Variables . . . 10
  - 1.3.4 Vectors, Matrices and Arrays . . . 13
  - 1.3.5 Performing Calculations with Vectors and Matrices . . . 18
  - 1.3.6 Indexing Values . . . 19
  - 1.3.7 Combining Vectors/Matrices . . . 22
  - 1.3.8 Variable Sizes and Statistical Data . . . 22
  - 1.3.9 Saving Data . . . 23

2 Script Files and Visualising Data **29**
- 2.1 Script Files . . . 29
- 2.2 Plotting . . . 31
  - 2.2.1 Basic Plotting Commands . . . 31
  - 2.2.2 Customising Your Plot . . . 33
  - 2.2.3 Multiple Plots . . . 36
- 2.3 Curve Fitting . . . 40
- 2.4 Saving Figures . . . 42
- 2.5 3D plots . . . 42

3 Functions and Conditional Statements **49**
- 3.1 The MATLAB Path . . . 49

3.2 Function Files . . . 50
3.2.1 Definition . . . 50
3.2.2 Writing Function Files . . . 50
3.2.3 Functions vs. Script Files . . . 53
3.3 Conditional Statements . . . 54
3.3.1 Relational and Logical Operations . . . 54
3.3.2 The `if-else` Statement . . . 57
3.4 Error Checking and Debugging . . . 59
3.4.1 Error and Warning Messages . . . 59
3.4.2 Debugging Programs . . . 60

4 Loops **69**
4.1 The `for` loop . . . 69
4.1.1 Syntax . . . 69
4.1.2 Simple Example . . . 70
4.1.3 Complex Example . . . 71
4.1.4 An Even More Complex Example . . . 72
4.2 The `while` loop . . . 74
4.2.1 Syntax . . . 74
4.2.2 Simple Example . . . 74
4.2.3 Complex Example . . . 76
4.2.4 An Even More Complex Example . . . 77
4.3 Which Loop Should You Use? . . . 78
4.4 Protected files . . . 78

1 Introduction and Basic Concepts

1.1 Introduction

Welcome to the MATLAB part of the Engineering Mathematics module (UFMFJ9-30-1). MATLAB is a scientific programming and software package used throughout the engineering industry and science in general. Not only is the ability to use MATLAB increasingly required by employers of graduate engineers in industry, its use may also help considerably in future modules and in your final year project.

1.1.1 MATLAB in Engineering Mathematics

While most of this Engineering Mathematics module will cover the mathematics upon which much of engineering is based, part of the module will cover an introduction to MATLAB, which is only taught in the first semester. It is comprised of two lectures and ten computer lab workshops. The lectures take place in weeks 9 and 15 while workshops take place all other weeks up to Christmas. The first lecture will be an introduction to MATLAB, workshops in weeks 10 to 14 will allow you to work through these notes and complete exercises. The lecture in week 15 covers more advanced topics, and the remaining workshops will allow you to complete this booklet and commence work on an assignment. For most of you, the assignment will be finished in your personal study time.

The first thing you should do in the workshop sessions is to work through the content in the chapter to review what was covered in the lecture and more. Have a go at entering the various commands (or editing and running the associated scripts) before working through the worksheet at the end of each chapter.

1.1.2 Course Notes

In these notes, the following conventions apply:

- Commands written in `typerwriter font` are commands you can use.
- The code examples have line numbers associated with them. In later chapters, line numbers will be referred to when explaining various functions.

These notes were initially developed by Benjamin Drew and subsequently edited and expanded by Gary Atkinson, both in the Department of Engineering Design and Mathematics, University of the West of England. Various sources were used for the content, including:

- MATLAB Documentation, available from `www.mathworks.com/help`
- Warren, Craig. (2012) *An Interactive Introduction to MATLAB*, University of Edinburgh. ©2012 University of Edinburgh. `www.eng.ed.ac.uk/teaching/courses/matlab`.

Examples and exercises adapted or based upon content from the notes from the University of Edinburgh are indicated by an asterisk (*), and are reprinted with permission of the author.

1.1.3 Assessment

This part of the Engineering Mathematics module will be assessed in two ways:

- Questions in the computer-based tests.
- A specific MATLAB assignment.

See the module handbook for more details of the computer-based tests, but be aware that there may be some questions associated with MATLAB in these tests.

The specific MATLAB assignment will involve programming in MATLAB. The coursework brief with specific details will be distributed midway through term and will involve writing several MATLAB functions to solve mathematical/engineering problems. This assignment will form half of the Component B (coursework) mark for the module (i.e. 12.5% of the whole module mark.)

The exercise sheets at the end of each chapter in these notes will help you learn the various topics that will be assessed in the computer-based tests and obtain the skills necessary to complete the separate assignment. As such, it is essential that you complete each exercise and make sure you fully understand what each line of code does.

1.1.4 What is MATLAB?

MATLAB is a software package from MathWorks, Inc. and is one of the most popular software packages for numerical computing and programming, especially in the engineering industry. MATLAB derives its name from MATrix LABoratory, as much of the mathematics underlying the software focuses around matrix operations. MATLAB provides an interactive environment for many engineering tasks, such as data analysis and visualisation, algorithm development and numerical computing.

One of the main reasons that MATLAB is used is that it relieves you of lot of the repetitive tasks associated with solving engineering problems, especially those that can only be solved numerically (as opposed to analytically, e.g. integrating symbolically). There are a large number of built-in functions, but its flexibility allows you to make your own custom functions to solve particular problems.

MATLAB itself is a core product and is augmented by additional software "toolboxes" which enhance particular features of the software, adding functions and capabilities, such

as the ability to analyse control systems, conduct symbolic analysis, carry out image processing tasks, etc. This module only covers the basics of the core MATLAB product, but you will no doubt encounter some of these toolboxes as you advance through your degree. MATLAB is available on all computers in the labs in the Department of Engineering Design and Mathematics.

1.1.5 Getting Help

This course is designed to be self-contained and everything you need to know for the module will be contained within these notes. If, however, you need help and your questions cannot be answered in the tutorials, the internet has a wealth of resources available to help you. Often, typing your question into a search engine will result in an answer to your question. You should also find the built-in help useful.

1.2 Getting Started

In the EDM computer labs, MATLAB is available via the Windows Start Menu (choose the latest version if there are multiple MATLAB installations available – e.g. MATLAB R2016a).

Launching MATLAB will bring up the MATLAB Desktop shown in Figure 1.1. If for some reason the default desktop is not shown when you start MATLAB, then click *Layout* near the top of the MATLAB window, then select *Default*. It is also recommended that you click *Layout* again, point to *Command History* and ensure that the *Docked* option is checked. There may be slight differences depending on the MATLAB version, but each of the five areas from the figure should be easily accessible to you.

1.2.1 The MATLAB Environment

The default layout of the MATLAB desktop divided into several panes, as shown in Figure 1.1. The most important are:

- The Command Window (1)
- Current Path (2)
- The Current Folder file list (3)
- The Workspace (4)
- The Command History (5)

1. Command Window

This window is where you can type commands at the command prompt >> and where most output data is displayed. When you type a command at the prompt, you must follow up by pressing Enter on the keyboard to execute the command.

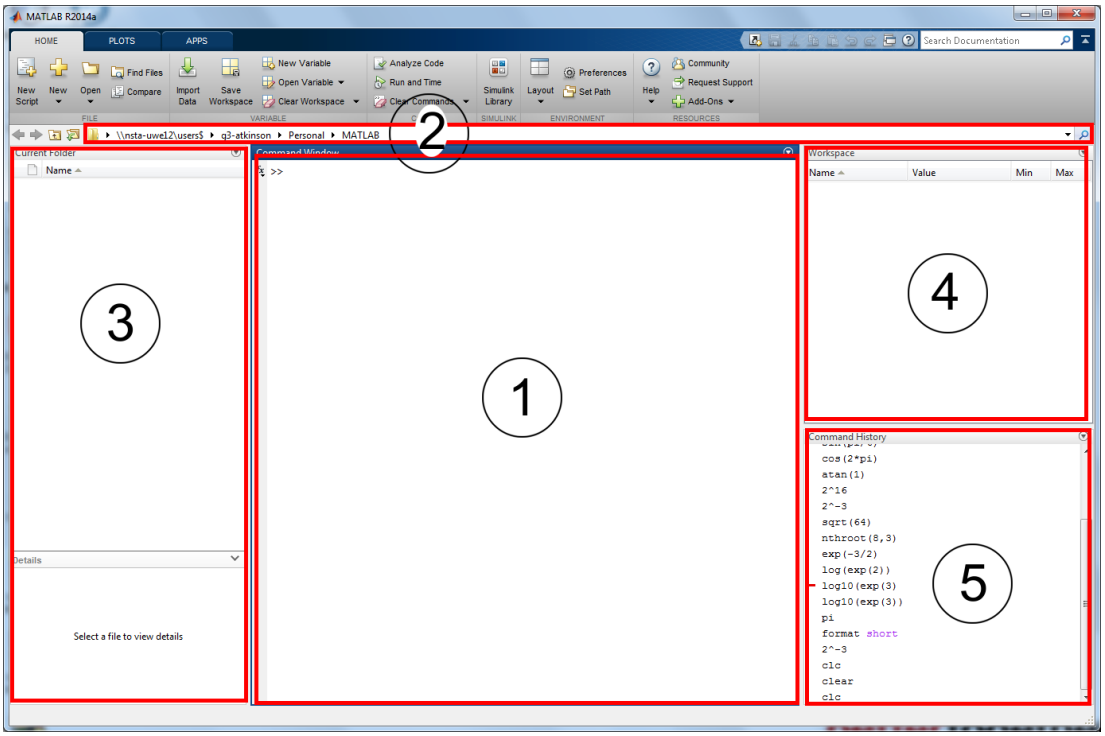


Figure 1.1: The MATLAB desktop (there may be slight differences depending on the MATLAB version you are using).

2. Current Path

This is the current Windows directory in which MATLAB is working. You can set this to whichever Windows folder you like, but should leave it as the default for now.

3. Current Directory List

This window shows the files and folders in the current path. You may find this useful in later chapters when working with files but can safely ignore it for now.

4. Workspace

This window lists all the variables created as part of your MATLAB session. As you perform calculations involving specific variables, the results will be shown here.

5. Command History

This window displays previous commands entered into the command window: in Figure 1.1 for example, you can see that the last command entered was `clc` which clears the command window and places the command prompt at the top of the screen.

1.2.2 Getting Help in MATLAB

One of the best ways of getting help is to use the internal help system in MATLAB. To access the help system you can:

- Click on the question mark symbol near the top of the MATLAB window.
- Get brief help about a certain function by typing `help topic` into the Command Window, where `topic` is the function or command in question. This method shows the help file in the command window.
- Get detailed help on a function (including the mathematics behind certain functions) in the formal help documentation by typing `doc topic` in the command window.

1.3 Using MATLAB

1.3.1 Basic Calculations

In the first instance, you can consider MATLAB to be a powerful calculator, and you can enter commands in the command prompt as you would expect. Simply type the command you require, and finish by pressing the Enter key. The abilities are extensive and a few are shown below. Remember to try out the commands in the boxes below and feel free to experiment.

Arithmetic

The basic operations are `+ - * /` and parentheses (round brackets) `( )`:

```
1 >> 2+3*4
2 ans =
3      14
4 >> (2+3)*4
5 ans =
6      20
```

The command in line 1 asks to evaluate $2 + 3 \times 4$, while the command in line 4 asks for the result of $(2 + 3) \times 4$.

Note that MATLAB follows the standard order of operations. This is (1) parentheses; (2) orders (powers, roots); (3) multiplication and division; (4) addition and subtraction.

For very large or very small values, scientific notation is used. For example,

```
1 >> 1/10000
2 ans =
3      1.0000e-4
```

where `1.0000e-4` means 1×10^{-4}.

Trigonometry

```
>> sin(pi/6)
ans =
         0.5000
>> cos(2*pi)
ans =
         1
>> atan(1)
ans =
     0.7854
```

Note that:

- MATLAB has a few pre-defined constants, e.g. π is typed as `pi`.
- you must explicitly type all operations: e.g. `sin(2*pi)`, not `sin(2pi)`.
- To get $\sin^{-1}$, $\cos^{-1}$ and $\tan^{-1}$, you use the functions `asin`, `acos` and `atan` respectively.
- Functions always take their arguments (the inputs to the function) in parentheses, e.g. `sin(pi)`, not `sin pi`.
- MATLAB normally deals in radians, not degrees. However, there are alternative functions for the most basic trigonometric operations that use degrees: try `sind`, `tand`, `acosd`, etc[1].

Exponential functions and logarithms

```
1 >> 2^16
2 ans =
3     65536
4 >> 2^(-3)
5 ans =
6     0.1250
```

The caret symbol, `^`, is used to raise a number to a power. So line 1 is used to evaluate 2^{16} and line 4 calculates 2^{-3}. The parentheses around the -3 term are not absolutely necessary in this example, but represents good practice.

[1] Note however, that standard practice dictates that radians should normally be used, especially when using calculus where degrees do not apply.

The following is an example of the use of a built-in *function*[2]. Functions generally require *input arguments* which are the values in the parentheses after the function name. The lines below demonstrate the `sqrt` function, which determines the square root of a value.

```
>> sqrt(64)
ans =
     8
```

To find the cube root, or in fact any root, use `nthroot`:

```
>> nthroot(8,3)
ans =
     2
```

Note that the function `nthroot` requires two input arguments separated by a comma. The first is the number you want to take the root of and the second is the order of the root. So the above evaluates $\sqrt[3]{8}$, i.e. the cube root of 8 (what does `nthroot(16,4)` calculate?).

```
>> exp(-3/2)
ans =
     0.2231
>> log(exp(3))
ans =
     3
>> log10(exp(3))
ans =
     1.303
```

Note that:

- `exp(x)` corresponds to e^x.
- `log` is used for the *natural* logarithm.
- `log10` is used for logarithm to the base 10 (also known as the "common logarithm").
- Confusingly, this is somewhat in contrast with standard mathematical notation, where "ln" is used for natural logarithm and "log" or "$\log_{10}$" for common logarithm.

There are many other elementary mathematical functions. To see a list, type `doc elfun` in the command window.

[2] Technically, `sin`, `cos`, etc are functions also, but let's not dwell on the precise definition of "function" until Chapter 3.

1.3.2 Number Formats

Although it seems that MATLAB only works to four decimal places, it is actually far more precise than that. To show all values, use the `format` command to change how MATLAB displays values. For example:

```
>> pi
ans =
     3.1416
>> format long
>> pi
ans =
     3.141592653589793
```

As an engineer, you might want to work in scientific notation, in which case you can use:

```
>> format short e
>> pi*1000000
ans =
     3.1416e+6
>> format long e
>> pi*0.000001
ans =
     3.141592653589793e-6
```

Type `help format` for more options or `format short` to return to the default representation.

1.3.3 Variables

You may have noticed that as you entered these commands, an item appeared in the Workspace window (see Fig. 1.1) called `ans`. This is the variable name that MATLAB gives to an undefined output in the command window. This can be used to perform follow on calculations. For example:

```
>> 2+3*4
ans =
     14
>> ans/7
ans =
     2
```

You can, of course, define your own variables by *assigning* a value to a variable. For example:

```
>> x = 2+3*4
x =
   14
>> y = 2*3
y =
   6
>> z = x+y
z =
   20
```

The format for assigning variables always follows `Variable_Name = value`, so

```
>> 3 + 4 = x
```

will *not* work. Try it anyway and observe the MATLAB error message.

Variable names may consist of any combination of letters and digits, starting with a letter. You cannot have spaces, but the underscore character is accepted. These are valid:

```
AverageValue, Net_Cost, FuelCons, x3, y34xc7
```

whereas these are not:

```
Average-Value, 2x, @fuel
```

Important: variable names are also case sensitive, so `radius`, `Radius` and `RADIUS` are all different variables.

Obviously, you should try and avoid redefining some of MATLAB's internal variables such as `pi`. You should also avoid using function names as variable names such as `sqrt` or `sin`.

It is also perhaps useful to note at this point that you can define variables without MATLAB confirming the fact by suppressing the output. To do this, type a semi-colon after your command. For example:

```
>> FuelConsumption = 45.6
FuelConsumption =
                  45.6
>> AverageCons = 34.5;
>>
```

On line four, the semi-colon was used meaning that MATLAB will perform the command, but the next line is simply the command prompt; the output has been suppressed.

Of course, you can use variables to define other variables, for example:

```
>> radius = 2.5;
>> Area = pi*radius^2
Area =
      19.6350
```

Note the capitalisation of the variable `Area`. This is to avoid confusion with the MATLAB function `area` (type `help area` to check this).

If you have been following these commands and entering them in MATLAB, there will now be quite a few variables shown in the Workspace window. Another way of seeing what variables you have used (and are still using) involves the `whos` command. To clear the workspace of all variables, type `clear`. Try:

```
>> whos
>> clear
>> whos
```

You should also notice that the Command History window has been filling up as you have been typing. If you double-click on one of the previous commands, MATLAB will re-execute it. Another way to re-use previous commands (or even similar ones), is to press the up-arrow (↑) on the keyboard to cycle through previous commands. You also have the opportunity to make changes to them before pressing Enter in order to execute the new edited version. If you know the first letter (or first few letters) of a previous command, you can type that to quickly jump back to the command you need. Let us work out the volume of a cylinder, then change its dimensions:

```
>> radius = 2.5;
>> height = 4;
>> crossarea = pi*radius^2;
>> Volume = height*crossarea
Volume =
    78.5398
```

To change the radius, type `r` then press the up-arrow key. Set the radius to a different value then recalculate the area (by typing `c` then up-arrow), and then the volume (by typing `V` then up-arrow) of the cylinder.

Non-standard variables and computations

MATLAB is able to handle cases where the output of a computation is infinite. For example:

```
>> x = 2;
>> y = 0;
>> z = x/y
z =
    Inf
```

Indeed, it is, occasionlly, useful to define a variable as infinity (or minus infinity):

```
>> x = -inf
    x = -Inf
>> y = x/2
y =
    -Inf
>> z = x/inf
    z = NaN
```

The last computation here has returned *not a number* (`NaN`) since the mathematical concept of $-\infty/\infty$ is ill-defined. Again, it is occasionally useful to deliberately set a variable to `NaN` (e.g. `x = nan`) as you may discover with experience.

Finally, it is worth noting that variables in MATLAB can take many different forms such as complex numbers, letters, words, structures and boolean (true or false) values. Most of these forms are beyond the scope of the Engineering Mathematics module. As a taster however, try typing `newVar = 'I am text'` into MATLAB[3] and observing the symbol for this variable in the Workspace window.

1.3.4 Vectors, Matrices and Arrays

As mentioned in the introduction, the name MATLAB is derived from MATrix LABoratory as much of the power of MATLAB comes from its ability to manipulate matrices. So, what is a matrix? First, let us consider what a **vector** is.

Vectors

In general science, a vector is most commonly defined as a two or three element array of numbers that represents a quantity with both direction and magnitude. A scalar by contrast, has only magnitude. In higher-level mathematics, there is no limit to the number of elements (or *dimensions*) that a vector can have, even though we can only experience three geometrical dimensions in normal life. In MATLAB, a vector is simply defined as a list of numbers that can be arranged either horizontally (known as a row

[3] Note that single quotation marks must be used here, which is the equivalent symbol to an apostrphe and is on the same key as the @ sign on a standard UK keyboard

vector) or vertically (a column vector). If you have used a spreadsheet software package before such as Microsoft Excel, and you have a table of values, you can think of each row or column of data as a vector.

In MATLAB there are a variety of ways to enter a vector. To create a vector with four elements in a single row, the values should be entered in square brackets separated by a space or comma:

```
>> a1 = [1 2 3 4]
a =
       1    2    3    4
>> a2 = [sin(pi/4), sin(pi/2), sin(3*pi/4), sin(pi)]
a2 =
       0.7071     1.0000     0.7071     0.0000
```

To create a column vector, do this the same way, but separate the elements with a semi-colon (or by typing each element on a new line):

```
>> b = [5; 6; 7; 8]
b =
       5
       6
       7
       8
```

To convert a vector between a row and a column arrangement, you *transpose* it, i.e. the rows and columns are interchanged. To do this in MATLAB you use the single quotation mark after the variable name:

```
>> a1'
ans =
       1
       2
       3
       4
```

Note that the standard mathematical notation for a transposed matrix is a^{T}. Note also that two single quotes will result in the transpose of a transpose, which is the original vector:

$$\texttt{a''} = \left(a^{\mathrm{T}}\right)^{\mathrm{T}} = \texttt{a}$$

Another way to create vectors is to specify a start value and an end value. This will create a vector with data points between the two values (inclusive) with an increment of one. For example:

```
>> c = 2:5
c =
    2    3    4    5
```

A different gap can be specified using the notation of `start_value : gap : end_value`. For example:

```
>> d = 2:0.5:4
d =
    2.0000    2.5000    3.0000    3.5000    4.0000
```

The end value does not need to be a multiple of the increment plus the starting value. For example:

```
>> 0.32:0.1:0.6
ans =
    0.3200    0.4200    0.5200
```

Yet another way to create a vector is to specify the start value, the end value and the number of (linear) data points required. This is achieved using the `linspace` function, which takes three input arguments: the start value, the end value and the number of data points. For example:

```
>> e = linspace(0,10,5)
e =
    0.0000    2.5000    5.0000    7.5000    10.0000
```

which creates a vector with 5 values between 0 and 10. For more information, type `help linspace` (there is also a `logspace` function). It should be noted that to obtain uniformly spaced values in a vector, you can always use *either* `linspace` or the previously mentioned colon notation. The choice of which to use depends on the particular application and, to an extent, personal preference (see the worksheet at the end of this chapter).

There are also some functions that can create "special" vectors (and matrices and higher-dimensional arrays). These include `ones` and `zeros`:

```
>> f = ones(1,3)
f =
       1    1    1
>> g = zeros(1,3)
```

```
g =
       0     0     0
```

The input arguments in those functions is the number of rows and columns. These commands therefore create a 1×3 matrix (a vector) of ones or zeros. Any matrix with one of the dimensions of 1 will be a vector.

A word of warning: if you are creating a particularly large vector, don't forget to use the semi-colon to suppress the output, as MATLAB will happily display the output of a huge vector for you, and this can take a while. As a demonstration, try:

```
h = 0:100
```

Now imagine that you wanted a vector from zero to 10000 with an increment of 0.001. Type it like this:

```
k = 0:0.001:10000;
```

With the semi-colon, MATLAB took 0.064 seconds to process the command. Without the semi-colon, the same operation took 42.26 seconds[4]! If you did forget the semi-colon and MATLAB is taking a while showing all the values in the command window, type Ctrl+C, which cancels whatever MATLAB is doing at the time.

Matrices

A matrix is made up of multiple juxtaposed vectors of the same length. Like a table in a spreadsheet program, it contains multiple rows of values (or multiple columns of values) and has two dimensions: number of rows × number of columns.

In MATLAB, a matrix is entered as a combination of what you learnt above.

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

`A` in this case is a 3×3 matrix.

You can also transpose a matrix in a similar fashion to a vector, i.e. by using the single quotation mark:

[4] This will depend on the speed of your computer, obviously.

```
>> A'
ans =
      1    4    7
      2    5    8
      3    6    9
```

You may also use colon notation:

```
>> B = [0:0.5:1; 4:0.5:5; 10:0.5:11]
B =
      0.0000     0.5000     1.00000
      4.0000     4.5000     5.00000
     10.0000    10.5000    11.00000
```

The functions `zeros` and `ones` also work for matrices:

```
>> D = ones(3,2)
D =
      1    1
      1    1
      1    1
>> E = zeros(2,4)
E =
      0    0    0    0
      0    0    0    0
```

There are other functions that can make matrices (and vectors) with special values, for example random numbers. Type `help rand` and `help randi` for more information.

One particularly interesting application of MATLAB matrices is in image processing where each pixel of the image is stored in a single element of a matrix. As a taster, try typing `load clown` into the command window. This will load a 200×320 element matrix called `X` into MATLAB. You can then view the image by typing `imagesc(X)`, followed by `colormap(gray)` (note American spelling).

Arrays

If a vector is a single row (or column) and a matrix is made up multiple rows (or columns), what is an array? In the first instance, an array is made up multiple matrices to create a three-dimensional "block" of numbers[5]. This is where MATLAB extends

[5] MATLAB actually considers both vectors and matrices as arrays, with vectors simply being one dimensional arrays and matrices being two dimensional arrays.

beyond the two-dimensional layout of a spreadsheet program. In fact, MATLAB has the capability to have arrays of more than three-dimensions (very hard to visualise, and beyond what you need to know for this course).

1.3.5 Performing Calculations with Vectors and Matrices

You saw previously that when performing basic calculations using single values, you can use the standard commands such as `* /` and `^`. Since the technology underlying MATLAB deals with matrices, simply using these same commands with vectors and matrices may not produce the results you expect. This is because MATLAB will be doing matrix calculations.

At this stage, we are not interested in matrix multiplication, inversion and so on, as this will be covered in the pure mathematics part of this module in Semester 2. What we are interested in though, is *element-by-element* multiplication, division and exponentiation, where individual elements of vectors or matrices are multiplied or divided by individual elements of other vectors. To do this in MATLAB, we should introduce the commands in Table 1.1.

Command	Description
`.*`	Element-by-element multiplication
`./`	Element-by-element division
`.^`	Element-by-element exponentiation

Table 1.1: Array commands

Let us say we have two vectors:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix}$$

and we wish to multiply elements together, so that we end up with one vector containing four elements:

$$C = \begin{bmatrix} (1 \times 5) & (2 \times 6) & (3 \times 7) & (4 \times 8) \end{bmatrix}$$

To achieve this result in MATLAB, we can type:

```
>> A = [1 2 3 4];
>> B = [5 6 7 8];
>> C = A.*B
C =
     5    12    21    32
```

Try typing `A*B`. What happens?

To divide, let us perform A divided by B:

$$D = \begin{bmatrix} (1/5) & (2/6) & (3/7) & (4/8) \end{bmatrix}$$

and in MATLAB :

```
>> D = A./B
D =
    0.2000    0.3333    0.4286    0.5000
```

Be careful with these commands as * and / (i.e. without the full stop) may result in real values, but not necessarily what you are looking for.

Another common computation is to square a each element of a vector or matrix. For example, squaring A:

$$E = \begin{bmatrix} (1^2) & (2^2) & (3^2) & (4^2) \end{bmatrix}$$

and in MATLAB :

```
>> E = A.^2
E =
    1    4    9    16
```

All of the above applies equally to both vectors and matrices. Note that when multiplying and dividing, the two vectors or matrices must have the same dimensions (number of rows and columns).

Important note: The error of forgetting to include the dot where element-wise arithmetic operations is required is probably the most common mistake made by novice MATLAB programmers. **Please remember this!** It is worth noting however, that the dot is optional for scalar operations (as discussed in Section 1.3.1) or when you wish to multiply all elements in a vector or matrix by the same value:

```
>> F = 0:5
F =
        0    1    2    3    4    5
>> G = 3
G =
        3
>> F*G
ans =
        0    3    6    9    12    15
>> F.*G
ans
        0    3    6    9    12    15
```

1.3.6 Indexing Values

Say we have a large matrix, but we want to know the value of one particular element, or the values in one particular column or row. How can we extract these?

Let us quickly create a 4×4 matrix:

```
>> C = magic(4)
C =
     16     2     3    13
      5    11    10     8
      9     7     6    12
      4    14    15     1
```

What do you think the `magic`[6] function does?

To refer to a particular element, you have to specify the row and then the column, in that order, such as:

```
>> C(4,2)
ans =
      14
```

To extract just certain values of the second row, for example, you can type:

```
l = C(2,2:4)
```

which will take the second, third and fourth values from the second row (remember that the result of 2:4 is a vector with values of 2, 3 and 4).

```
l =
     11    10     8
```

To extract a whole row or column, you use the colon operator. Let us extract a whole column:

```
>> m = C(:,3)
```

This extracts every value of the third column, hence:

```
m =
      3
     10
      6
     15
```

[6] Hint: what is the sum of each row? What is the sum of each column? What is the sum of each diagonal?

Similarly, you may use

```
n = C(3,:)
```

to extract every value of the third row, hence:

```
n =
      9     7     6    12
```

Note that row and column indices start at 1, so the value in the upper left hand corner is `C(1,1)`. Note also that only one index is technically required for a vector, regardless of whether it is of row or column format:

```
>> v = [5 4 3];
>> v(2)
ans =
    4
>> v(1,2)
ans =
    4
```

As well as using indexing to extract elements from a variable, you can also modify certain elements. For example, define this matrix:

```
>> M = [1 2 3;4 5 6]
M =
      1     2     3
      4     5     6
```

Set the top-right value to zero:

```
>> M(1,3) = 0
M =
     1     2     0
     4     5     6
```

Set the first column to be equal to the second column:

```
>> M(:,1) = M(:,2)
M =
     2     2     0
     5     5     6
```

1.3.7 Combining Vectors/Matrices

Notice that in the previous section, you can use various indexing commands to produce a vector from a matrix. For example `m = C(:,3)` gives a vector of the third column and `p = C(:,2)` will result in a vector of the second column. Say that we now want a new matrix with these two columns. We do this by typing:

```
>> q = [m p]
q =
      3     2
     10    11
      6     7
     15    14
```

So, by placing the two vectors within a set of square brackets, you obtain a new matrix with the two vectors juxtaposed. A couple notes: to achieve this, the vector lengths must be the same (here, they are both four); and you can use a comma to separate the vectors if you want, i.e. `q = [m,p]`.

The same can happen if you had two matrices. Let's do the following:

```
>> D = magic(3);
>> E = D.*2;
>> F = [D E]
F =
      8     1     6    16     2    12
      3     5     7     6    10    14
      4     9     2     8    18     4
```

You can see that the last three columns are two times the first three columns, a matrix produced using the `magic` command.

1.3.8 Variable Sizes and Statistical Data

It is often obvious what the size (dimensionality) of your variables will be. For example, if you entered `x = [1 2;3 4]` into the command window, then the dimensions are clearly 2×2. However, for reasons that will become apparent as you gain experience with MATLAB, it will not always be so obvious, and so the functions `length` and `size` come in handy.

The function `length` returns a scalar corresponding to the number of elements in a vector, while size returns a two-element vector corresponding to the dimensions of a matrix:

```
>> a = 0:0.1:1;
>> length(a)
```

```
ans =
      11
>> b = [1 2 3;4 5 6];
>> size(b)
ans =
     2    3
```

It is also often useful to calculate statistical data about a vector or matrix. Of course, MATLAB has many powerful tools for statistics, but three of the most commonly used are `min`, `max` and `mean`. These can be used in several ways but most commonly to simply calculate the minimum, maximum or mean value of the elements in a vector:

```
>> d = [2 -5 6 3 10 -1];
>> min(d)
ans =
      -5
>> mean(d)
ans =
      2.5000
>> x = -1:0.1:1;
>> y = x - x.^2;
>> max(y)
ans =
     0.2500
```

As with all MATLAB functions, type `help min`, `help length`, etc. for more details (including how these can be applied to matrices). Other related functions that you may wish to investigate are `sum`, `diff`, `std`, `median` and `range`.

1.3.9 Saving Data

Now that you know how to create variables, you might want to save them for later usage. Remember that although commands in the Command History are retained between sessions, variables in the Workspace are not. So, if you have data that you wish to retain for later use, run the following command in the Command Window:

```
>> save myfile.mat
```

This creates a file called *myfile.mat* in the current path (see area 2 of Figure 1.1) storing all your variables. The .mat file extension indicates that it is a MATLAB MAT-file containing MATLAB variables.

To restore data from a MAT-file into the workspace, either double-click the file in the Current Directory list window or use the `load` command. Observe the variables in the Workspace as you type the following:

```
>> clear all
>> A = 1:5;
>> B = A*2;
>> save myfile.mat
>> clear all
>> load myfile.mat
```

Obviously, you do not have to call the file 'myfile', but keep the .mat extension. MATLAB offers the facility to save and load data in a variety of formats including CSV (comma separated variables) and Microsoft Excel. Explore the help system to find out more. If you only want to save certain variables then these should be listed after in the following format:

```
>> save myfile.mat a b
```

This will save variables `a` and `b` only (assuming they exist). All variables will remain in the workspace in MATLAB until you type `clear` or restart MATLAB, but only those listed will be permanently saved to the file.

Chapter 1: Commands

To find out more information on these commands, type `help` *command* or `doc` *command*.

Command	Description
`clc`	Clear Workspace
`help` *topic*	Get help on a function called *topic*
`+ - * / ^`	Mathematical operators
`sin`, `cos`, `tan`	Trigonometric operators (radians)
`asin`, `acos`, `atan`	Inverse trigonometric operators (radians)
`sind`, `acosd`, etc.	Trigonometric operators (degrees)
`pi`	The predefined variable for π
`sqrt`, `nthroot`	Functions to find roots
`exp`, `log`, `log10`	Functions for exponentials and logarithms
`format long`, `format short`	Number formats for Command Window output
`whos`	Show current variables
`clear`	Clear workspace (erasing all variables)
`ones`	Create a matrix of ones
`zeros`	Create a matrix of zeros
`rand`	Create a matrix of random numbers
`randi`	Create a matrix of random integers
`.* ./ .^`	Element-by-element operators
`magic`	Creates a magic square
`M(r,c)`	Returns a value in row `r` and column `c` of matrix `M`
`M(:,c)`	Returns *all* rows of column `c` of matrix `M`
`M(r,:)`	Returns *all* columns of row `r` of matrix `M`
`length`	Returns the length of a vector
`size`	Returns the dimensions of a matrix
`min`	Returns the minimum value in a vector
`max`	Returns the maximum value in a vector
`mean`	Returns the mean of all values in a vector
`save` *filename*	Saves a workspace (all variables)
`load` *filename*	Loads a workspace
Ctrl+C	Special keystroke for force MATLAB to stop all computations

Table 1.2: Commands introduced in Chapter 1.

Chapter 1: Worksheet

1. *Arithmetic Operations: Compute the following:
   a) $\sin(2\pi) + e^{\frac{-3}{2}}$
   b) $10 \log_{10} 0.5$
   c) $\dfrac{2^5}{2^5 - 1}$
   d) $\left(1 - \dfrac{1}{2^5}\right)^{-1}$
   e) $\dfrac{\sqrt{5} - 1}{(\sqrt{5} + 1)^2}$

   (Answers: 0.2231; −3.0103; 1.0323; 1.0323; 0.1180)

2. *Exponentials and Logarithms: Compute the following:
   a) e^3
   b) $\ln(e^3)$
   c) $\log_{10}(e^3)$
   d) $\log_{10}(10^5)$

   (Answers: 20.0855; 3; 1.3029; 5)

3. *Trigonometric Functions: Compute the following:
   a) $\sin(\frac{\pi}{6})$
   b) $\cos(\pi)$
   c) $\tan(\frac{\pi}{2})$
   d) $\sin^2(\frac{\pi}{6}) + \cos^2(\frac{\pi}{6})$

   Does your answer to part (c) seem reasonable? Think about the graph of a tan function.

   (Answers: 0.5; −1; 1.6331e+016, 1)

4. *Create a vector `t` that ranges from 1 to 10 in steps of 1, and a vector `theta` that ranges from 0 to π and contains 32 elements. Make sure you can do this using both the colon notation and the `linspace` function as described in Section 1.3.4. Then

do the element-wise calculation of the following (note that your answer should be 32 elements long for x and z and ten elements long for y):

$$x = 2\sin(\theta)$$
$$y = \frac{t-1}{t+1}$$
$$z = \frac{\sin(\theta^2)}{\theta^2}$$

5. *First create the variables to represent the following matrices:

$$A = \begin{bmatrix} 12 & 17 & 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 8 & 3 \\ 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \qquad C = \begin{bmatrix} 22 \\ 17 \\ 4 \end{bmatrix}$$

Note that one may assign a variable, `x0`, by indexing such that it contains the second and third elements of A. This can be done using `x0=A(2:3)`. Use indexing of A, B and C to do the following:

a) Assign to the variable `x1` the second value of vector A.

b) Assign to the variable `x2` the third column of matrix B.

c) Assign to the variable `x3` the third row of matrix B.

d) Assign to the variable `x4` the first three values of vector A as the first row, and all the values in matrix B as the second, third and fourth rows.

e) Modify `x4` so that the top-left and bottom-right elements are set to -1.

6. *If matrix A is defined using the MATLAB code:

```
>> A = [1 3 2; 2 1 1; 3 2 3];
```

which command will produce the following matrix?

$$B = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}$$

7. Use the `rand` function to generate a *vector* called `data` that consists of ten random numbers. Note that `rand(n)` generates a matrix of `n` × `n` numbers, while `rand(1,n)` gives a vector (i.e. a 1 × `n` matrix). Verify that the variable has the expected dimensionality using both the `length` and `size` functions. Then use the `min`, `max` and `mean` functions to compute the minimum, maximum and mean of the values in the vector.

8. Assign variables A and B such that A = [1 2 3 4 5 6 7 8 9 10] and B = [10 9 8 7 6 5 4 3 2 1]. Create a new variable, C, such that its first five elements consist of element-wise products of A and B and the other five elements are the sum of A and B.

Hint: you can change values in arrays via indexing, e.g.

```
>> X = 10:-1:0;
>> X(5:7) = 0
```

9. Assign the variable A with any integer elements, which has dimensions 1×5 (i.e. a 5-element row vector), e.g. $A = [3\ 5\ 1\ 2\ 7]$. Write a single line of code to calculate the mean of the first, third and fifth elements.

10. Generate a 5×4 matrix, M, where the first two columns consist of random numbers between 0 and 1, the third column consists entirely of 1's and the fourth column entirely of 2's.

    Hint: see what happens if you type the following

```
>> X = [1 2;3 4];
>> Y = [5;6];
>> Z = [X Y]
```

    Repeat the above but with the random numbers in the range 0 to 10.

11. Create a 3×3 magic matrix, N. Write a line of code that stacks all the elements into a 9×1 column vector, P.

Optional, more challenging questions:

12. The MATLAB function `eye(n)`, generates an identity matrix of dimension n×n. For example,

```
>> I = eye(3)
```

    generates the following:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

    Write MATLAB code to generate a new matrix, J, of dimension 5×5 such that the 1's of an identity matrix are replaced by counting numbers from top-left to bottom-right.

13. With help from the MATLAB function, `repmat`, repeat the previous question with a 100×100 identity matrix. Type `help repmat` for more information.

2 Script Files and Visualising Data

You should now be familiar with using MATLAB as an advanced calculator, being able to perform basic calculations, assign variables, define vectors and matrices, and manipulate data.

This chapter builds upon Chapter 1 by first introducing you to script files, which allow you to enter a series of commands and run them in one go, before moving on to showing you how to plot data in MATLAB.

2.1 Script Files

Up to now, you have been entering commands in the command window one at a time. As your use of MATLAB gets more sophisticated, this method of entering commands can become rather cumbersome and time consuming, especially if you have to repeat a series of commands with a small change every time.

Fortunately, *script* files can be written that contain a series of MATLAB commands that you would otherwise type at the command prompt. By writing a script file, you are also saving all your work for future reference and you can begin to conduct relatively advanced calculations, all based inside script files. A script file is essentially just a plain text document, but has a `.m` file extension (and, hence are often called M-files).

Basic Script Files

To make a new script file, choose *New Script* near the top-left corner of the main MATLAB window, or type `edit` into the command window. This will bring up a new empty *Editor* window in which you enter commands.

As good practice, there are a few ground rules that are worth following when writing scripts. These are:

- **Comments:** It is extremely helpful both for yourself and for others reading your scripts to place effective comments in your script file. To do this, use the percent sign (`%`), which tells MATLAB to ignore anything after the percent sign on the line.

- **Header:** It is always a good idea to include a header (also known as a preamble) in your script at the beginning which gives whoever is reading the script some information about what it contains. Useful information would be the author, the date of the last update (sometimes even a list of dates with details of each update), and a variable dictionary, which details what variables are being used.

- **Clear:** It is also useful to start a script with `clear` and `clc`. The `clear` command deletes all variables in the current workspace (you would normally either create variables within the script, or import them from a MAT-file using the `load` command). The `clc` command cleans up the command window, which can help you with any errors that occur within the script (they will appear at the top of the window).
- **Script file names:** When saving your scripts for further use, they must not contain spaces, start with a number, be names of built-in functions or be variable names. If you wish to imitate a space within the name, use the underscore character.

As a demonstration, we shall make a simple script that calculates the cube roots of the numbers from 1 to 5.

```
% cuberoots.m
% Script to calculate some cube roots
%
% Benjamin Drew. Last update: 11/06/2013

% Variable Dictionary
% x          values from which to calculate cube roots
% y          output values

clear; % Clear all variables from workspace
clc; % Clear command window

% Define range for x
x = 1:5;

% Calculate y
y = nthroot(x,3) % No semi-colon, therefore output shown

```

Once you have written your script, save it (click *Save As...* under the *EDITOR* tab at the top of the main MATLAB window). There are a few ways to run it:

- In the command window, type the name of your script (without the `.m` extension)
- Under the *EDITOR* tab choose *Run*. If MATLAB asks you to change folder, then do so.
- In the editor window, press F5 on the keyboard. Again, change folder if MATLAB asks.

It should be noted that the last two methods listed will automatically save the script and run it. Conversely, entering the script name in the command window will instead run the last saved version. Furthermore, if you are using the first method, make sure the current path (shown in area 2 of Figure 1.1) is set to the file location of the script file (see Section 3.1 for further details.)

Run the code. There will be two variables in the workspace, `x` and `y`, and the command window will display `y`, which should be as follows:

```
y =
    1.0000    1.2599    1.4422    1.5874    1.7100
```

Note that any command you can type in the command window can be used in a script.

2.2 Plotting

The next step is to use MATLAB to visualise data with plots. MATLAB is very powerful for producing both two-dimensional and three-dimensional plots. It allows two methods to produce them: via the command line or using an interactive tool (or "wizard"). The are many wizard-type methods in MATLAB to plot and manipulate data and it is well worth investigating some of them. In these notes however, we concentrate primarily on the more standard method to manipulate the data and plots using commands. Furthermore, this section will focus on two-dimensional plots, with some three-dimensional plots used later as demonstrations.

Plots can be exported in a variety of formats for your needs including JPG, TIF, PNG, EPS and PDF, so it is straightforward to insert graphics into a report, for example.

Note that many of the pieces of code presented here are entered using the command window, but as discussed above, they would work equally well, if not better, if put inside a script.

2.2.1 Basic Plotting Commands

The basic plotting command in MATLAB is `plot(x,y)`, where `x` and `y` are your data, which need to be defined beforehand. When using the `plot` command, the variables `x` and `y` are normally vectors containing the x and y coordinates of the data to be plotted. Let us start with a very simple straight line plot in the form of:

$$y = mx + c$$

where, in this case, $m = 2$ and $c = 5$.

To plot the graphic, try typing the following in the command window:

```
>> x = -5:0.1:5;
>> y = 2.*x + 5;
>> plot(x,y)
```

This should now bring up a *figure* window in which your plot is displayed. Notice that MATLAB is currently automatically controlling the scale of the axes (and notice the position of the origin).

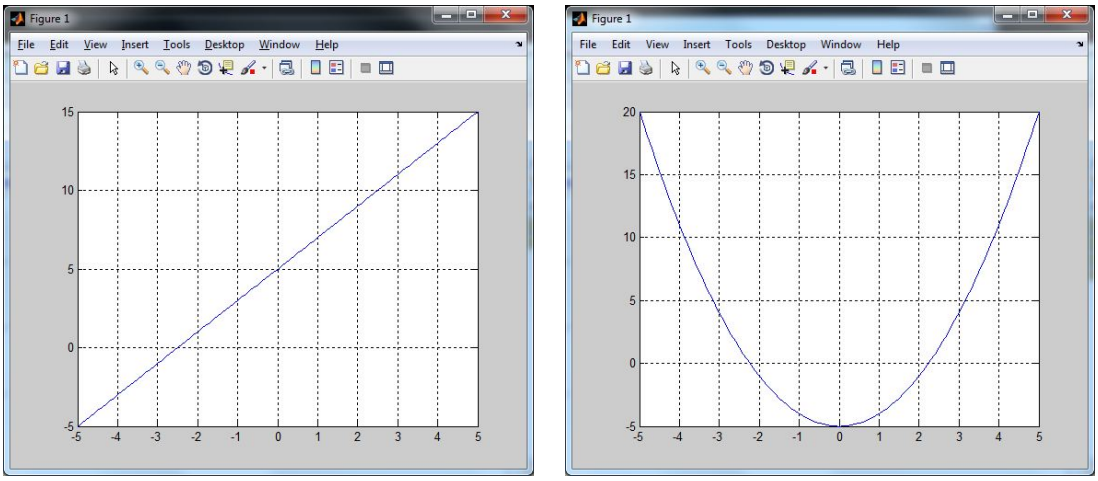
What is happening? Let's go through the code above:

- **Line 1:** Here, we are defining our x data as the numbers between -5 and 5 with an increment of 0.1 (the *independent variable*).
- **Line 2:** Here, we are defining our y data (known as *dependent variable*), using the equation of a straight line.
- **Line 3:** Runs the plot command.

To help visualise what is going on, type the following to show some gridlines:

```
1 >> grid on
```

You should now have something that looks like Figure 2.1(a).



(a) (b)

Figure 2.1: (a) Plot of straight line; (b) plot of parabola.

Let's try a plot of a quadratic function (i.e. a parabola), using the following equation:

$$y = x^2 - 5$$

To plot this, type the following:

```
1 >> y = x.^2 - 5;
2 >> plot(x,y)
```

Notice, since x was defined earlier, we don't need to redefine it (unless you want to change the range of values). In line 1, above, we have redefined our y data with the quadratic equation and the plot now shows the parabola. Notice that we need to place a full-stop ahead of the operator `^` so that MATLAB takes each value of x and squares it (in this case, `x^2` $\neq$ `x.^2`). Let us again, turn the gridlines on using the `grid on` command, resulting in Figure 2.1(b).

2.2.2 Customising Your Plot

Labels

The plots shown above give us the line defined in the equation, but do not really tell us anything without labels and titles. These can be added as follows:

```
1 >> xlabel('x');
2 >> ylabel('y = x^2 - 5')
3 >> title('A parabola')
```

Notice that the input arguments for the function `xlabel`, `ylabel` and `title` need to be in quotation marks.

Line Specification

By default, using the `plot` function produces plots shown as solid blue lines. This can be customised in a variety of ways to suit the purpose. The basics are shown here but more information can be found by accessing the help system. Common line styles and colours are shown in Table 2.1.

String Specifier	Line Colour	String Specifier	Line Style
`b`	Blue (default)	`-`	Solid line (default)
`r`	Red	`--`	Dashed line
`y`	Yellow	`:`	Dotted line
`g`	Green	`-.`	Dash-dot line
`k`	Black		
`m`	Magenta		

Table 2.1: Line styles and colours for MATLAB graphics.

The term "String Specifier" in the tables refer to the item you need to place within the plot command to specify the line style or colour. So, to plot data with a dashed red line, the code looks like:

```
1 >> plot(x,y,'r--')
```

Note that you specify the colour and the line style within quotation marks. To plot data with a dotted black line, the code looks like:

```
>> plot(x,y,'k:')
```

If you wish to place data points on the graph, a similar method is employed for the markers using the string specifiers in Table 2.2:

String Specifier	Marker Type	String Specifier	Marker Type
`+`	Plus sign	`o`	Circle
`*`	Asterisk	`x`	Cross
`s`	Square	`d`	Diamond
`.`	Dot		

Table 2.2: Marker types for data points.

So, to plot some data in green with diamond data points and a dash-dot line, the code is:

```
>> plot(x,y,'gd-.')
```

A particularly useful format is:

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

which connects dots using a solid line in the default colour (blue).

Many other options are available, allowing you to change the line width, the marker size, and much more. For example, try typing the following into the command window:

```
>> plot(0:10,(0:10).^2,'mo--','markersize',5,'linewidth',3)
```

To see further line specification options, you can type `doc LineSpec`.

Axes

Another way to customise your graph is to set the scale of the axes. MATLAB normally scales the axes automatically so that "nice" limits are chosen based on the extents of the variables being plotted. There are various other automatic choices set using the `axis` command, including:

- `axis tight` which sets the axis limits to the range of data.
- `axis equal` which sets the aspect ratio so that tick mark increments on each axis are equal in size on the screen.

- `axis square` which makes the axis box square in shape (instead of a rectangle).

Typing `help axis` gives you various other options.

You can also set your own limits using the `axis` command. For example, try the following and observe the axis change after you enter the last line:

```
>> x = 0:0.1:10;
>> plot(x,x.^2 - 5);
>> grid on
>> axis([0 3 -5 10])
```

To change the scale for just one axis, you can use the `xlim` or `ylim` options. For example, try the following after the lines above.

```
>> ylim([-5 5])
```

Plot Tools

Plot properties can also be manipulated interactively (without having to issue commands) by clicking on the *Show Plot Tools* icon in the Figure Window toolbar, as shown in Figure 2.2

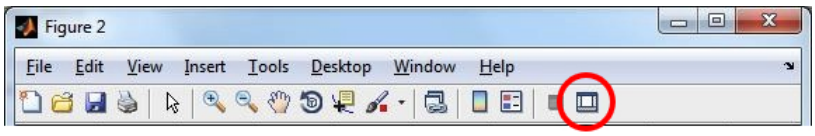


Figure 2.2: *Show Plot Tools* toolbar icon in the Figure Window toolbar.

Many of the options under plot tools can also be accessed by right-clicking on different parts of the figure window (most notably the data points/line).

Plot wizards

It is worth noting at this point that there are many other types of plots available. Depending on the plot type, the data may need to be two-, three- or even higher-dimensional. To see what plot types are available for a particular variable (or variables), select the variable(s) from the workspace and then click on the *PLOTS* tab at the top of the main MATLAB window. Feel free to experiment with some of the options available but avoid spending too much time looking at these as the most important features are specifically discussed in these notes.

2.2.3 Multiple Plots

Multiple Plots on the Same Axes

There are two ways of plotting two sets of data on the same axes. One way is to simply place two sets of data as the input arguments in the `plot` command. Say, for example, we have two sets of data, (x, y_1) and (x, y_2). To plot them both in the same axes, you could type:

```
>> plot(x,y1,x,y2)
```

MATLAB is clever enough to realise that these are two sets of data and will differentiate the two using different colours.

The alternative method makes use of the `hold` command. The `hold` command keeps what is currently on the graph while plotting a new set of data, and works in the following fashion:

```
>> plot(x,y1)
>> hold on
>> plot(x,y2)
```

and you could continue applying new sets of data until the `hold off` command is given or you close the figure window. Note that with this method, you should ideally give different line specifications to be able to distinguish between the different sets of data.

As an example, let us plot three sinusoids between 0 and 2π using the same axes. The three sinusoids are:

$$\begin{aligned} y_1 &= \sin(t) \\ y_2 &= \sin\left(t - \frac{\pi}{2}\right) \\ y_3 &= \sin(t - \pi) \end{aligned}$$

The code, presented here as it would be in a script, is therefore:

```
% sinewaves.m
% Script to plot sinewaves
%
% Benjamin Drew. Last update: 17/07/2012

% Variable Dictionary
% t          Independent variable (time)
% y1,y2,y3   Dependent variables

clear; % Clear all variables from workspace
```

```
clc; % Clear command window
close all; % Close all previous figure windows

% Define range for t
t = 0:pi/20:2*pi;

% Calculate y1, y2 and y3
y1 = sin(t);
y2 = sin(t - pi/2);
y3 = sin(t - pi);

% Plot data
plot(t,y1,'-.r*')
hold on
plot(t,y2,'--mo')
plot(t,y3,':bs')

% Add labels and legend
xlabel('t');
legend('sin(t)','sin(t-pi/2)','sin(t-pi)');
```

This produces the graph shown in Figure 2.3.

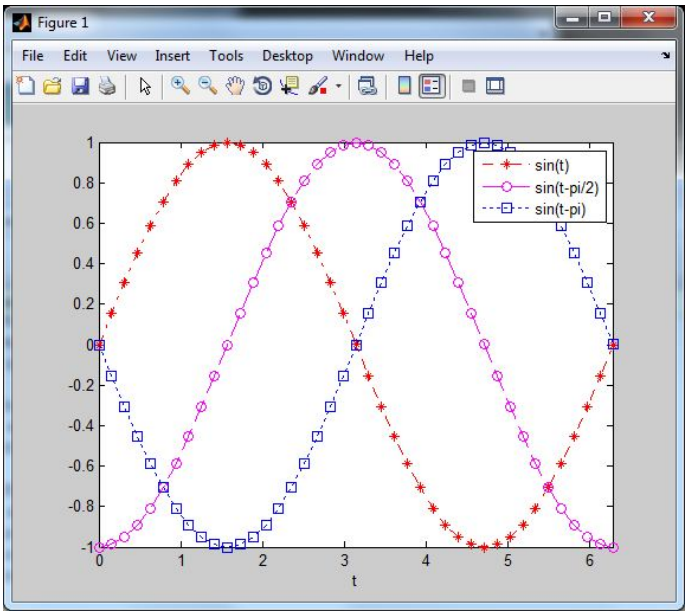


Figure 2.3: Example showing multiple plots on same axes.

The following explains the code, line by line:

- **Line 15:** Setting a suitable range of values for t, the independent variable (this could have been equally x). Here, the range is 0 to 2π in steps of $\frac{\pi}{20}$.
- **Lines 18–20:** Using the equations to calculate vectors for y_1, y_2 and y_3.
- **Line 23:** Plotting the first set of data, (t, y_1).
- **Line 24:** Turning `hold` on, meaning that plotting data will not replace what has already been plotted.
- **Lines 25–26:** Plotting the second and third sets of data.
- **Line2 29–30:** Adds a label to the t-axis and a legend (see below).

Note we have added the command `close all` to the preample, which closes all previous figures. If this command wasn't added, this script would add new lines to any existing axes—this can soon get out of control if you run the script a number of times.

Note also that the great advantage of writing scripts comes when making a modification. Say we now wanted to plot the three curves using the cosine function instead of the sine function, we can make the small changes to lines 18–20, and run the code again, without having to re-enter the other commands.

Legends

When plotting multiple series of data on the same axes, it is often useful to use a legend to identify the individual lines. To add a legend to Figure 2.3, you would type:

```
1 >> legend('sin(t)','sin(t-pi/2)','sin(t-pi)')
```

as shown in line 30 above.

The arguments for the `legend` command are the names (in quotes) you wish to give to the lines. Other arguments can be added to customise the location and orientation of the legend (among other options). To learn more, type `help legend`.

Multiple Plots on Different Axes

Another way of producing multiple plots is to produce a number of different plots in a single figure window. This is done using the `subplot` command. This command divides the figure window into an array (defined in the command) of rows and columns, and can place a different plot in each "tile" of the array. For example, let us plot four functions in a single window. The functions are:

$$y_1 = x \qquad y_2 = x^2 \qquad y_3 = x^3 \qquad y_4 = x^4$$

The code below will set up the data for the plots:

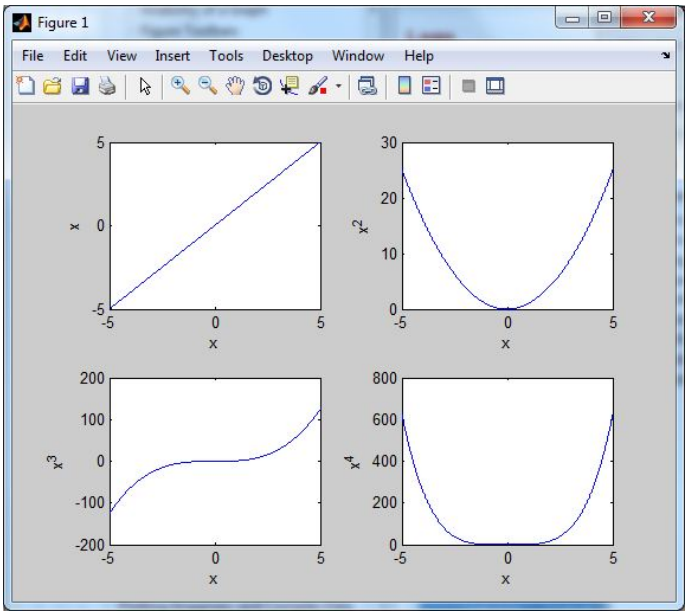


Figure 2.4: Example usage of the `subplot` command.

```
>> x = -5:0.01:5;
>> y1 = x;
>> y2 = x.^2;
>> y3 = x.^3;
>> y4 = x.^4;
```

So, the code firstly defines x between -5 and 5 with increments of 0.01. The variables `y1` to `y4` are the four dependent variables we wish to plot.

The `subplot` command normally takes three input arguments: the number of rows, the number of columns, and in which position the figure needs to appear. For example, let us plot the functions above:

```
>> figure;
>> subplot(2,2,1); plot(x,y1); xlabel('x'); ylabel('x');
>> subplot(2,2,2); plot(x,y2); xlabel('x'); ylabel('x^2');
>> subplot(2,2,3); plot(x,y3); xlabel('x'); ylabel('x^3');
>> subplot(2,2,4); plot(x,y4); xlabel('x'); ylabel('x^4');
```

This code should result in Figure 2.4.

Note:

- The `figure` command opens a new blank figure window.

- The arguments in the `subplot` command are the number or rows (2 rows in this case), the number of columns (again, 2 in this case) and the position. The plots are counted along the top row of the figure window, then the second row, etc.
- Four commands are entered on the same line, separated by semi-colons. Doing this enables all four commands to be run together (to save space in this document).

2.3 Curve Fitting

Accompanying these plot tools, MATLAB offers powerful tools for fitting curves and equations to data. Again, there is an interactive graphical user interface available by selecting the *Basic Fitting* command from the *Apps* menu in a figure window. In the resulting dialogue box, a number of different types of fitting tools for two-dimensional plots are available.

Like many things in MATLAB, the alternative method of adding trendlines or curves of best fit is to use the command line (or commands in a script file), and in particular, the `polyfit` function. This function will fit a polynomial to the data.

The function `polyfit` is called as follows:

```
>> coefficients = polyfit(xdata,ydata,n)
```

(It won't work if you type it at this stage since the raw data is not yet in the workspace). This command defines the variable `coefficients` which is a vector containing the coefficients of the polynomial of best fit. The input arguments are your raw data (here called `xdata` and `ydata`) and the degree of the polynomial needed. For a straight line approximation, this value would be one ($n = 1$); for a quadratic, $n = 2$; for a cubic, $n = 3$ and so on.

Let us take some sample data. This data should be downloaded from the online course resources site for this module, and is called `linear_fit.mat`. Clear the workspace (type `clear`) and then load this data, either by double clicking on the file in the Current Folder window, or by typing:

```
>> load linear_fit.mat
```

You may need to set the current path (see Fig. 1.1) to that in which the `.mat` file is located.

You should now see two variables in the workspace, `x` and `y`. These represent your raw data to which you wish to apply a curve fitting. For this example, let us apply a straight line approximation. To do so, type the following:

```
>> coefficients = polyfit(x,y,1);
>> y_fit = x*coefficients(1) + coefficients(2);
```

```
3 >> plot(x,y,'ro');
4 >> hold on
5 >> plot(x,y_fit);
6 >> xlabel('x'); ylabel('y'); grid on; title('Curve-fitting');
7 >> legend('Raw data','Linear trendline');
```

This should result in Figure 2.5.

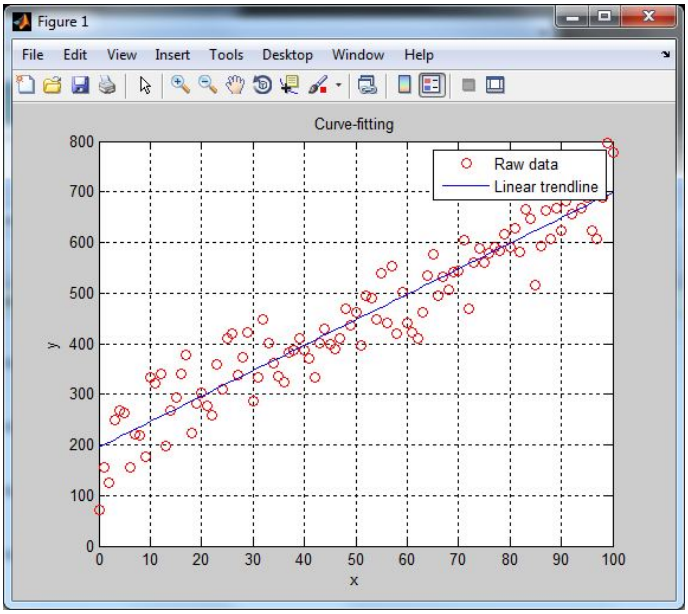


Figure 2.5: Plotting raw data and line of best fit.

Let us analyse the code, line-by-line:

- **Line 1:** The variable `coefficients` is defined by the `polyfit` command as the coefficients of a 1st order polynomial (equivalent to a straight line fit) based on the data in `x` and `y`.
- **Line 2:** Calculate `y` values for the given `x` values using the equation of the line of best fit and store in `y_fit`. An alternative line would use the `polyval` function, which is convenient for higher order polynomials:
  `y_fit = polyval(coefficients,x);`
- **Lines 3–6:** Plots the raw data, line of best fit, labels and title, and grid. Note that the `'ro'` listed in line 3 plots the raw data as red circles.
- **Line 7:** Adds a legend.

To determine the equation of the line of best fit, inspect the values stored in `coefficients` (e.g by typing `coefficients` in the command window or looking at the Workspace window). For this data, the coefficients come out to be:

```
>> coefficients
coefficients =
    5.0316   195.4123
```

So the equation of line of best fit is:

$$y = 5.0316x + 195.4123$$

2.4 Saving Figures

In the introduction, it was mentioned that figures can be saved in a variety of formats. To save a figure, in the figure window choose *File* → *Save As...*. Under the *Save as* box, you can choose the file format for the figure.

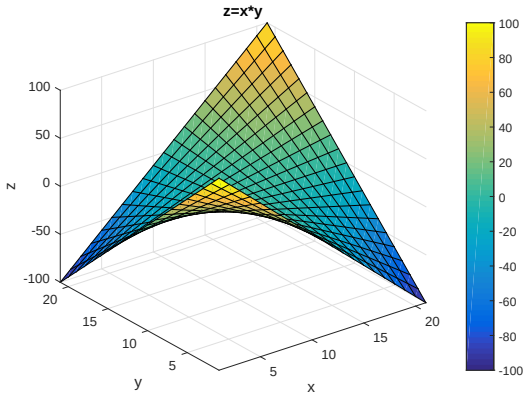
MATLAB's default file format for figures (denoted by the `.fig` file extension) saves the figure but not the data used to create the figure. Having saved the figure in this format though, you can still open it and add things like legends, axis labels and titles, as well as use the *Plot Tools* dialog box. You will only be able to open `.fig` files in MATLAB. To save for use in reports, it is recommended to save graphs in PNG format. To do this, simply select the PNG filetype from the *Save As* dialog box.

2.5 3D plots

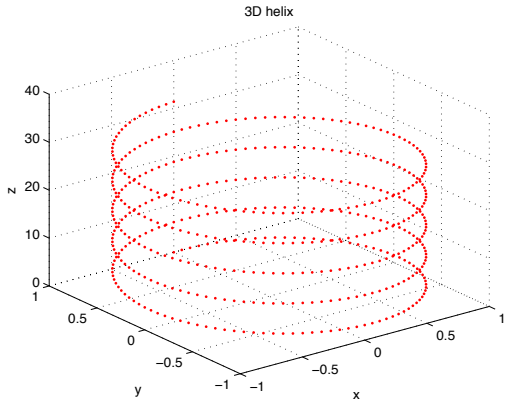
As stated previously, MATLAB has the ability to visualise data in three dimensions. The following is here to give you an idea of its capabilities but goes beyond what is being taught in this module. It may, however, give you some ideas of ways to present data in the future.

The following pages give you some examples, preceded by the code used to generate them. Can you decipher what is going on? (Hint: use the `help function_name` command if you need some ideas.) The second and third examples are taken from *An Interactive Introduction to MATLAB*, University of Edinburgh. Try typing the code into a *script* file and running them from there. This allows you to easily make a few changes and observe their effects.

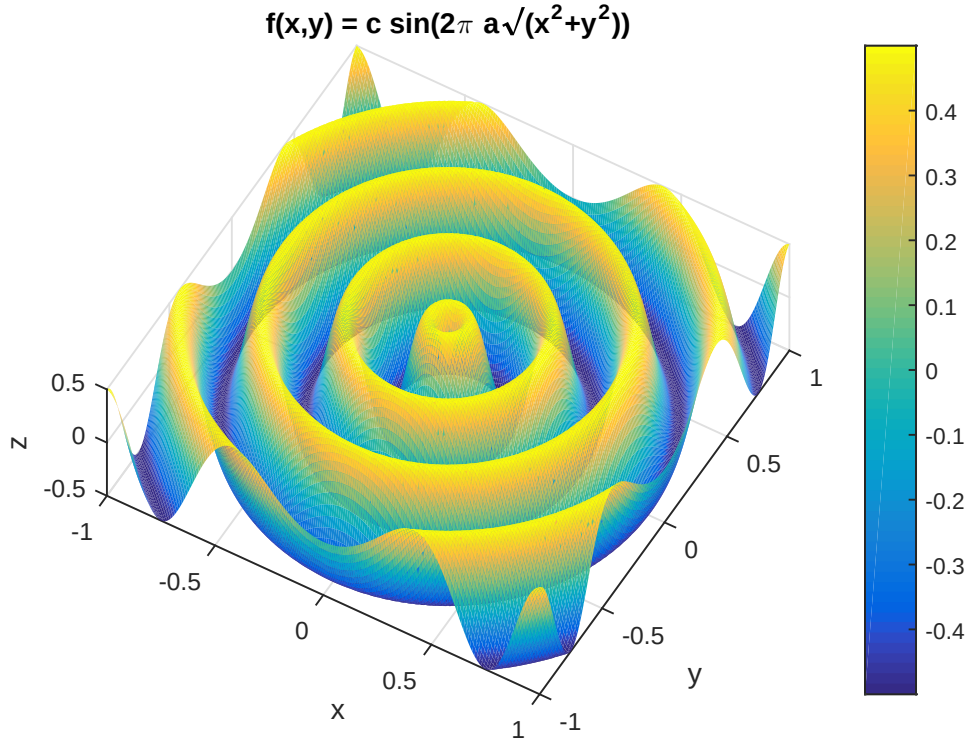
```
>> x = -10:10;
>> y = x;
>> z = x'*y;
>> surf(z);
>> colorbar; xlabel('x'); ylabel('y'); zlabel('z'); title('z = x*y')
```




```
>> t = 0:pi/50:10*pi;
>> plot3(sin(t),cos(t),t,'r.'), grid on, ...
xlabel('x'), ylabel('y'), zlabel('z'), title('3D helix');
```

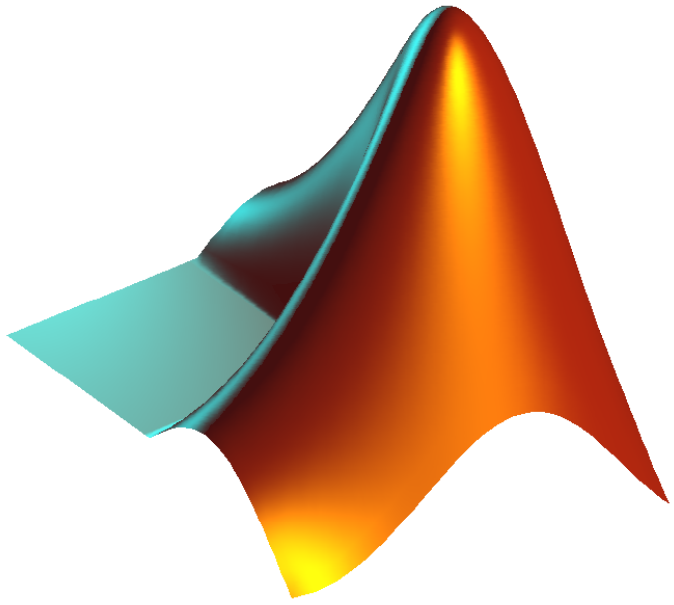


```
>> x = linspace(-1,1,200);
>> y = x;
>> a = 3;
>> c = 0.5;
>> [xx, yy] = meshgrid(x,y);
>> z = c*sin(2*pi*a*sqrt(xx.^2+yy.^2));
>> surf(xx,yy,z), colorbar, xlabel('x'), ylabel('y'), zlabel('z');
>> title('f(x,y) = c sin(2\pi a\surd(x^2+y^2))')
>> shading flat
>> view(30,70)
```



The code for this last example can be found by typing `edit logo` into the Command Window. You are not required to understand all the commands here; rather this is to demonstrate the graphics capabilities of MATLAB.

```
>> L = 40*membrane(1,25);
>> logoFig = figure('Color',[1 1 1]);
>> logoax = axes('CameraPosition', [-193.4013 -265.1546  220.4819],...
'CameraTarget',[26 26 10], 'CameraUpVector',[0 0 1], ...
'CameraViewAngle',9.5, 'DataAspectRatio', [1 1 .9], ...
'Position',[0 0 1 1], 'Visible','off',...
'XLim',[1 51], 'YLim',[1 51],'ZLim',[-13 40], 'parent',logoFig);
>> s = surface(L, ...
'EdgeColor','none', 'FaceColor',[0.9 0.2 0.2], ...
'FaceLighting','phong', 'AmbientStrength',0.3, 'DiffuseStrength',0.6,...
'Clipping','off','BackFaceLighting','lit', 'SpecularStrength',1,...
'SpecularColorReflectance',1, 'SpecularExponent',7, ...
'Tag','TheMathWorksLogo', 'parent',logoax);
>> l1 = light('Position',[40 100 20], 'Style','local', ...
'Color',[0 0.8 0.8], 'parent',logoax);
>> l2 = light('Position',[.5 -1 .4], 'Color',[0.8 0.8 0], ...
'parent',logoax);
```



Chapter 2: Commands

Command	Description
`%`	Anything after the perecent sign is 'commented out' (ignored)
`clear`	Clears all variables from workspace
`close all`	Closes all previous figure windows
`plot`	Basic plot command
`grid`	Turns on or off the gridlines
`xlabel`, `ylabel`	Labels for axes
`title`	Title for figure
`axis`	Axis commands
`xlim`, `ylim`	Defining axis scales
`hold on`	Keeps previously plotted data in figure
`legend`	Adds a legend to the figure
`figure`	Opens a new blank figure window
`subplot`	Creates subplots in figure windows
`polyfit`	Creates coefficients for a curve fit
`polyval`	Creates curve fit based on values from `polyfit`

Table 2.3: Commands introduced in Chapter 2.

Chapter 2: Worksheet

1. *Plot the following functions by entering suitable commands into the Command Window (you will need to decide upon appropriate ranges for x):

   a) $y = \dfrac{1}{x}$, with a blue dashed line.

   b) $y = \sin(x)\cos(x)$, with a red dotted line.

   c) $y = 2x^2 - 3x + 1$, with red cross markers.

   Turn the grid on in all your plots, and remember to label axes and use a title.

2. *Given the following function:

$$s = A\cos(\phi) + \sqrt{b^2 - (A\sin(\phi) - c)^2}$$

   write a *script file* to plot s as a function of angle ϕ on a well-labelled graph using a solid blue line when:

$$A = 1 \qquad b = 1.5 \qquad c = 0.3 \qquad 0 \leq \phi \leq 2\pi \text{ radians}$$

   Add a second line to the graph where $A = 0.5$ using a broken red line.

3. The following table gives climate data for Bangkok, Thailand. You can download the data from the online course tools (Bangkok.mat). Save the data into the current directory and use the command `load Bangkok` to import into MATLAB.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Mean T (°C)	25.9	27.4	28.7	29.7	29.2	28.7	28.3	28.1	27.8	27.6	26.9	25.6
High T (°C)	32.0	32.7	33.7	34.9	34.0	33.1	32.7	32.5	32.3	32.0	31.6	31.3
Low T (°C)	21.0	23.3	24.9	26.1	25.6	25.4	25.0	24.9	24.6	24.3	23.1	20.8
Mean P (mm)	9	30	29	65	220	149	155	197	344	242	48	10

   Write a script file to plot the following, each inside a separate figure window:

   a) One plot for the mean temperature

   b) One plot for the high temperature and low temperature on the same axes

   c) One bar graph for the precipitation (type `help bar` if in doubt).

4. Using the data from the previous question, use the `yyaxis` command to plot the mean temperature and the precipitation on the same axes with different y-values. (Hint: type `help yyaxis` or `doc yyaxis` for more information on the `yyaxis` function). Answer this question in a script file.

5. Download the `raw_data.mat` file from the online course resources and find the equation for:

   a) A straight line fit

   b) A quadratic fit

   You may answer this either in a script file (recommended) or directly using the Command Window.

   Time permitting, plot the two best fits and the raw data on the same axes.

   (Answer: $3.0637x - 5.2980$; $0.3702x^2 - 0.6383x + 0.8102$)

3 Functions and Conditional Statements

We introduced script files in Chapter 2, but we will discuss them a bit more in this chapter. In addition to basic script files, there is another type of M-file called a *function*. The main technical difference between the two is that script files work with variables in the current Workspace, while function files only work with *internal* variables. Do not worry too much about these details at the moment, as they will become clearer shortly.

Later in this chapter, we will consider how conditional statements can be introduced to both script and function files so that MATLAB will take a different action depending on the status of certain variables.

As before, the files used as demonstrations in this chapter can be downloaded from the online course resources.

3.1 The MATLAB Path

Before introducing functions, it is beneficial to define the concept of the MATLAB path (also called "current directory/path/folder" or "working directory"). The MATLAB path is the Windows directory in which MATLAB is currently "working" and is always visible near the top of the main window (area 2 in Fig. 1.1).

For the purposes of this module, the main thing you need to know about the path is that it tells MATLAB where to look for any non-standard files (including scripts and functions that you wrote). To illustrate the concept of the path, first open one of the script files you wrote for Chapter 2 and save it in a new folder called `H:\quickdemo` using the file name of `quickfile.m`. Next, change the MATLAB path to `H:\quickdemo`. Type the name of the script file (without the `.m` extension) into the Command Window and press Enter. Your program should run successfully because you have told MATLAB where to look for the file. Now change the current path to another folder and try to run the script file from the Command Window again. You will probably get an error message this time as MATLAB will be unable to locate the file. Finally, run the script again, but this time do so by clicking somewhere in the script itself in the editor window and pressing F5. MATLAB should ask you if you would like to change the folder to that where the script file is located. If you choose "yes" then the path will be changed accordingly and your code should run successfully.

To complete this discussion of the MATLAB path, select the *HOME* tab at the top of the main MATLAB window, then click *Set Path*. You should see a small window with a list of folders called the *search path list*. Whenever you try and call a script (or function – see below) from the Command Window and MATLAB cannot locate it in the current folder, it will search through these. If MATLAB can still not find the required file, then it will return an error message. You can add new folders to this list, although you probably won't need to do so for this module.

3.2 Function Files

3.2.1 Definition

In mathematics, a function can be regarded as an equation (or other algorithm) that takes one or more values as an input and returns a corresponding output. For example, the function $f(x) = 2x^3 - 4$ takes a value for x (the only input), then cubes it, multiplies by two and subtracts 4 to obtain $f(x)$ (the output). Similarly, the function $g(x, y) = x^2 + y$ takes x (the first input), squares it and adds y (the second input) to obtain the output $g(x, y)$.

This concept can be extended to MATLAB (and other programming languages) where any number of inputs and outputs are allowed. Furthermore, it may be that the function does much more than apply an equation, but rather complete a long sequence of commands. The key point here is that a MATLAB function takes one or more inputs, does something with them and returns one or more outputs[1]. Furthermore, each input does not need to be a single numeric value but could be vectors, matrices or even character strings.

3.2.2 Writing Function Files

As you are now aware, MATLAB has many built-in functions, such as `sqrt`, `sin`, `plot` and so on, but MATLAB also offers the ability for you to write your own functions to perform specific tasks.

A function file *always* begins with a function definition line which looks like the following:

```
1 function [ output_args ] = function_name( input_args )
2 %FUNCTION_NAME Summary of this function goes here
3 %   Detailed explanation goes here
4
5
6 end
```

You can start writing a function file in the same way as a script file (i.e. by clicking *New Script* under the *HOME* tab of the main MATLAB window) and entering the text into the editor window. The file is saved with a `.m` extension in the same way as script files but MATLAB knows the file is a function by the presence of the declaration at the start of the file.

The various parts of a basic function are as follows:

- `function` This is mandatory, and tells MATLAB that the M-file is a function. Where the word `function` is not present, MATLAB will assume the file is a script.

[1] Strictly speaking, it may be possible to have no inputs or outputs, but let's not dwell on that technicality for now.

- `[ output_args ]` are the output variables from the function. When you run the function, these are the outputs of the function. For example, when you run the function `sqrt(16)`, the output is 4. The square brackets are there to accommodate more than one output, which are separated by a comma.
- `function_name` is where the name of the function should go. Standard practice dictates that you should save the M-file with the same name you use here.
- `( input_args )` contains the input variables, comma-separated.
- `%FUNCTION_NAME Summary of this function goes here` is where you should put some comments after the function definition line to explain what your function does and what the input and output variables should be. This should be in addition to other comments you would normally add to your script files.

As an example, let us write a function to work out the volume of a cylinder. We pass two values to the function, the diameter and the height of the cylinder. The function would therefore be written as:

```
function [ cylvol ] = calcvol( diameter,height )
%CALCVOL Calculates cylinder volume
%   CALCVOL(D,H) is the volume of a cylinder with diameter D and
%   height H.

%   Benjamin Drew, modified by Gary Atkinson. Last Update: 15/08/2013
%
%   Variable Dictionary:
%   diameter   cylinder diameter (input)
%   height     cylinder height (input)
%   csarea     cross sectional area (internal)
%   cylvol     cylinder volume (output)

csarea = pi*diameter^2/4; % Calculates the cross sectional area.
cylvol = csarea * height; % Calculates the volume.

end
```

This is saved as `calcvol.m`.
So in the MATLAB Command Window, if you entered:

```
>> cylvolume = calcvol(3,5)
```

MATLAB would go off to the function, assign the value of 3 to the first input argument (`diameter`) and 5 to the second (`height`), perform the calculation with those input arguments and return:

```
cylvolume =
   35.3429
```

Check this on your calculator. The function has calculated that a cylinder of diameter 3 and height 5 has a volume of 35.3429 (arbitrary units). N.B. if you get an error message, then you may need to change the current path to that where `calcvol.m` is located (see Section 3.1).

You can see after running this function that the variables defined in the function do not appear in the workspace, as these variables were *internal* to the function.

Entering the following code in the command window would generate the same outcome:

```
>> d = 3
>> h = 5
>> cylvolume = calcvol(d,h)
```

Two points to note here. Firstly, we are defining variables and using those in our call to the function – you do not always need to "hard-code" the values in the call. Secondly, if we decide to call the function using variables for the input arguments, they do not need to have the same name as the variables used in the function itself. The same is true for output variables.

To Reiterate...

It is worth reiterating exactly how the function file works. You call a function file by entering its name followed by any input arguments required enclosed by parentheses. For example, to find the square root of 8, you would use `sqrt(8)`, where `sqrt` is the function name, and `8` is the input argument. Sometimes function files require more than one input argument, and these are separated by a comma, for example `nthroot(8,3)` calculates the cube root of 8.

When the function runs, it returns (outputs) whatever is in the square brackets in the function declaration. As such, the variable in said square brackets needs to be given a value somewhere within the function. So, in the example above, notice that `cylvol` is given a value in the function, and since `cylvol` is the variable name in the square brackets in the first line, this is the value that will be returned to the workspace.

There can be more than one output from a function, and these too are separated by a comma in the function line. For example, a function that determines the square root, cube root and fourth root of a number might look something like the following.

```
function [ sqroot, cubroot, fouroot ] = rootsgalore( x )
%ROOTSGALORE Calculates square, cube and fourth roots of a number
```

```
%   ROOTSGALORE(X) Calculates square, cube and fourth roots of a
%   number, X.

%   Benjamin Drew. Last Update: 11/06/2013
%
%   Variable Dictionary:
%   x           value to calculate the roots of (input)
%   sqroot      square root (output)
%   cubroot     cube root (output)
%   fouroot     fourth root (output)

sqroot = sqrt(x);
cubroot = nthroot(x,3);
fouroot = nthroot(x,4);

end
```

So this function requires *only one* input argument, but returns *three outputs*. To call this function, you could enter something like the following into the Command Window:

```
>> [a, b, c] = rootsgalore(15);
```

which would fill the varibales `a`, `b` and `c` with the square root, cube root and fourth root of 15 respectively.

3.2.3 Functions vs. Script Files

This is a good point to reflect on the differences between script files and functions. Both of these are text files with a `.m` extension that contain a series of MATLAB commands. The difference is that a script file runs each line in turn and stores each of the computed variables in the workspace. A function, on the other hand, is always called from elsewhere (either the command window, a script file or another function) with given inputs. It then computes some combination of outputs and discards any intermediate variables.

Which should you use? That depend on the application and, to a degree, personal preference. As a general rule, script files tend to act as "umbrella" code for a project, while functions are written to perform the various smaller tasks that constitute the project and are called from the script. Functions are particularly useful when they need to be called many times with different input values. With experience, the various pros and cons of scripts and functions will quickly become apparent.

One final, but important, note on functions

As mentioned previously, it is often a good idea to start *script* files with a `clear` command to remove any old variables from the workspace to avoid confusion. However, you should **NEVER** use `clear` in a function. This will delete the input variables so MATLAB will not know what values to use in computing the function's output(s).

3.3 Conditional Statements

Up to now, most of the programming you have done in MATLAB has involved simple step-wise running: MATLAB just reads your scripts line-by-line performing each function as it comes across it. For instance, a piece of code will define variables, perform calculations and plot results.

Decision making is an important concept in programming and allows certain commands to run depending on conditions. The flow of the program then skips commands that do not meet these conditions while performing commands when the conditions are met.

3.3.1 Relational and Logical Operations

Before implementing conditional statements in your program, you need to understand how MATLAB makes decisions. The symbols in Table 3.1 show the relational operators that MATLAB uses to compare values. The results of these operators will always be either true, denoted by 1 or false, denoted by 0.

Operator	Mathematical Symbol	MATLAB Symbol
Equal	$=$	`==`
Not Equal	$\neq$	`~=`
Greater than	$>$	`>`
Greater than or equal to	$\geq$	`>=`
Less than	$<$	`<`
Less than or equal to	$\leq$	`<=`

Table 3.1: Relational Operators

Relational operators can be combined using logical operators (for example, to demonstrate a number is within a range—greater than one value *and* less than another). These are listed in Table 3.2.

Operator	MATLAB Symbol
AND	&
OR	|
NOT	~

Table 3.2: Logical Operators (N.B. The "OR" (|) symbol often confusingly appears as "¦" on a standard UK keyboard).

As an example, let us compare some numbers:

```
1  >> a = 10;
2  >> b = 15;
3  >> c = 12;
4  >> a<b
5  ans =
6       1
7  >> a>b
8  ans =
9       0
10 >> c>(a+b)
11 ans =
12      0
13 >> (c>a) & (c<b)
14 ans =
15      1
```

Lines 4, 7, 10 and 13 are known as *logical expressions* as their results can only be either true or false (either 1 or 0 respectively). Note that you can *not* use `a<c<b` instead of line 13; you can only compare two values at a time.

Similar expressions can be used on vectors to compare the values, element-by-element. Try the following*:

```
1  >> x = [1 5 3 7];
2  >> y = [0 2 8 7];
3  >> k = x<y
4  k =
5      0     0     1     0
6  >> k = x<=y
7  k =
8      0     0     1     1
9  >> k = x>y
10 k =
```

```
    1    1    0    0
>> k = x>=y
k =
    1    1    0    1
>> k = x==y
k =
    0    0    0    1
>> k = x~=y
k =
    1    1    1    0
```

Note the difference between the symbol used to assign variable (=) and that to check equality (==).

And with logical operators:

```
>> x = [1 5 3 7];
>> y = [0 2 8 7];
>> k = (x>y) & (x>4)
k =
    0    1    0    0
>> k = (x>y) | (x>4)
k =
    1    1    0    1
>> k = ~((x>y) | (x>4))
k =
    0    0    1    0
```

You can see that when comparing two vectors, the result is a *logical* vector. Note that logical variables have different icons in the Workspace. As you may expect, all of the above discussion applies equally to matrices.

Relational and logical operators can also be used to modify variables. This is best explained via example. Let's set up a matrix:

```
>> M = [2 4 6;8 6 4]
M =
     2     4     6
     8     6     4
```

To set all elements with a value less than 5 to zero:

```
>> M(M<5) = 0
M =
```

```
3      0     0     6
4      8     6     0
```

which you might think of as "M where M is less than 5 becomes zero".

Negate all values greater than 7:

```
1  >> M(M>7) = -M(M>7)
2  M =
3       0     0     6
4      -8     6     0
```

3.3.2 The `if-else` Statement

What are these relational and logical operations used for? The `if`, `elseif` and `else` commands in MATLAB allow you to control which parts of your script or function should run based on given conditions. The construction is simply: "if a condition is true, run this command":

```
1  if logical_expression
2      commands
3  end
```

On line 1, the `if` statement is followed by a logical expression, and if the condition is true, then MATLAB moves on to line 2 to run the command (or series of commands). If the condition is not met, i.e. it is false, then MATLAB skips line 2 and moves on to line 3, which ends the conditional statement. You must always include an `end` command to close an `if`-statement. As a working example, take a look at the script file below:

```
1  % simple_if_script1.m
2  % Script demonstrating if-statement
3  %
4  % Benjamin Drew, modified by Gary Atkinson. Last update: 14/08/2014
5
6  % Variable dictionary
7  % x     Random number between 0 and 1
8
9  clear;
10 clc;
11
12 x = rand(1)
13
14 if x>0.5 % Check whether x is less than 0.5.
```

```
15      disp('The number generated is greater than 0.5.')
16  end
```

This script generates a random number between 0 and 1, and then goes on to check whether the number is greater than 0.5. If it is, it displays a statement in the Command Window saying so using the `disp` command. If it is not, it ignores the statement. Try running the code several times.

It is relatively simple to expand the `if` statement to include other conditions if the first condition is not met. This can be done by using the `elseif` and `else` commands. For example, let us expand the code from above to check for other conditions:

```
1   % simple_if_script2.m
2   % Script demonstrating if-else statements
3   %
4   % Benjamin Drew, modified by Gary Atkinson. Last update: 14/08/2014
5
6   % Variable dictionary
7   % x     Random number between 0 and 1
8
9   clear;
10  clc;
11
12  x = rand(1)
13
14  if x<0.2 % Check whether x is less than 0.2.
15      disp('A very low value for x.')
16  elseif x<0.8 % Check whether x is less than 0.8.
17      disp('A medium value for x')
18  else % Otherwise, x must be greatergreater than 0.8.
19      disp('A very high value for x.')
20  end
```

Note how the code is indented, as this aids readability. MATLAB will usually do this for you automatically, but if not, highlight your code and choose *Indent* → *Smart* from the *EDITOR* tab or press Ctrl+I.

Line 14 demonstrates the straightforward `if` statement, as above. If this condition is false, MATLAB moves on to the `elseif` statement on line 16, checking this condition. If the result of this is also false, then it moves on to the `else` command, which catches *all* other possibilities (notice that there is no logical expression here). You are free to use as many `elseif` statements as you like before you use an `else` or `end`. Furthermore, you are not obliged to use an `else` at all, but must end the sequence of commands with an `end`.

3.4 Error Checking and Debugging

This section introduces some methods and features of MATLAB to help make your functions "fool-proof", to help avoid coding errors and to find and correct mistakes. While it is strongly recommended that you spend some time working through this section, it is permissible to omit it for now and return to it at a later date if you feel that you are falling behind in the module. Please note however, that the material in this section (especially Section 3.4.2) will help to minimise the likelihood of making an error in your assignment.

3.4.1 Error and Warning Messages

It is often a good idea to add a few conditional statements to the start of functions to check that the correct form of inputs have been used. As an example, consider the function to calculate the volume of a cylinder given in Section 3.2.2. The function takes the diameter and height of the cylinder as inputs and returns the volume as the output. Clearly, it would be inappropriate to enter negative values for the inputs and a well-written function would check this and display an appropriate error message as necessary.

The code below is the same as the previous cylinder function, but with error checking added (and comments removed for the sake of brevity):

```
1  function [ cylvol ] = calcvol( diameter,height )
2  % CALCVOL Calculates cylinder volume
3  % Gary Atkinson, last update 15/08/2013
4
5  if ~isnumeric(diameter) | ~isnumeric(height) % Error check format
6      error('Inputs must be numeric. Function terminating');
7  end
8  if diameter <= 0 | height <= 0 % Error check range
9      error('Dimensions must be greater than zero. Function terminating');
10 end
11
12 csarea = pi*diameter^2/4; % Calculates the cross sectional area.
13 cylvol = csarea * height; % Calculates the volume.
14
15 end
```

Consider lines 8 to 10 first. These lines check if either of the two input variables are negative (i.e. invalid). For the case where the inputs are acceptable, MATLAB moves on to the next lines of code. Otherwise the `error` function on line 9 is used to stop MATLAB and display the error text in the Command Window in red. Can you work out what the other error check on lines 5 to 7 does?

In contrast to a MATLAB error message, a *warning* is used when a non-critical problem is encountered. Consider replacing lines 8 to 10 in the previous code with the following:

```
if diameter < 0 | height < 0 % negative values entered
    diameter = abs(diameter);
    height = abs(height);
    warning('Negative dimension(s) entered. Using absolute values.');
end
```

Here the code checks for negative values as before. However, instead of terminating the program, the `abs` function is used to replace the variables with their absolute (positive) values[2]. The `warning` command then displays the given text in the Command Window in orange, but allows the program to continue.

3.4.2 Debugging Programs

The above error and warning features are useful to include in codes to make sure your scripts and functions are being used correctly. But what about where there are errors in your code itself? Like all programming languages, MATLAB requires very precise coding in order for functions and scripts to operate correctly. Even experienced programmers commonly make errors in coding and struggle to correct ("debug") their code. MATLAB has a wide range of debugging tools available to help spot errors in code, the most important of which are introduced here.

Imagine that we wish to write code able to compute the first six values of the following sequence for a given value of y_1:

$$y_n = {y_{n-1}}^{1.5}$$

In an attempt to do this we might write the following code (you will learn of a more compact means to do this in the next chapter):

```
function y = powerseq(yFirst)
% POWERSEQ(yFirst) Calculates the first six terms of the sequence
%   y_n = y_(n-1) ^ 1.5 for a given y_1 value
%   Gary Atkinson. Last update: 13/08/2014
%
%   Variable dictionary:
%   y1    First value for y
%   y     Output, corresponding to first six elements of the sequence

y = zeros(1,6); % initialise y vector
```

[2] It is not *generally* recommended to set input variables to absolute values in this way – negative inputs are often perfectly valid.

```
11 y(1) = yFirst; % set first element to same value as input (y_1)
12 y(2) = y(1).^1.5; % compute second value of sequence
13 y(3) = y(2).^1.5; % third value, etc...
14 y(4) = y(2).^1.5;
15 y(5) = y(4).^1.5;
16 y(6) = y(5).^1.5;
```

Observe the deliberate mistake on line 14 but suppose we did not notice this as we wrote it. We may pose two questions: (1) how to avoid such an error in the first place?, and (2) given such an error occurred, how can it be detected?

Avoiding Mistakes

Regarding the first of these questions, it is generally a good idea to write your code and test it in stages. In this case for example, you might want to first write code that only computes the first two values of the sequence and output those. It is also a good idea to omit the semi-colons at this stage in order to see the output of each line in the Command Window. Once you are happy that the first two values are correct, you can then add the semicolons and the code for the other values. The key point here is: **test your code continually as you write it**, rather that write the entire program and test it at the end. This advice will become progressively more useful as your programs become longer and more sophisticated.

Furthermore, once your code (or a portion thereof) is complete, be sure to test it for a range of known parameters. For example, if you have written code to compute the roots of a quadratic (see the worksheet at end of the chapter) you should test your program works with several different quadratic coefficients covering cases where one, two or no roots are present and verify the results using a calculator.

Another piece of advice is to maintain neat indents as discussed in Section 3.3.2.

Detecting and Correcting Mistakes

Inevitably, there will be times when MATLAB is not giving the results you expect and you need to debug. Broadly speaking, programming errors can be classified into three types:

- Syntactic error
- Runtime error
- Semantic error

A **syntactic error** is where invalid/meaningless syntax is used. An example in MATLAB would be as follows since this sequence of characters has no defined meaning:

```
1 a = 2 ][ 3}
```

Syntactic errors are usually the easiest type of error to deal with. If you run a script containing such an error then MATLAB will terminate when it reaches this line and display a red error message with a reference/link to the line of code containing the error. It is usually a case of simply editing that line of code to correct the error.

A **runtime error** by contrast contains code that makes sense to MATLAB but the culprit line causes an error for some other reason. An example would be:

```
1 count1to6 = [1 2 3 4 5];
2 sixthVal = a(6);
```

Both of these lines are syntactically correct, but the second line generates an error as it is trying to access the sixth element of a five-dimensional vector. As before, MATLAB will terminate the program and display an error message linking to the line that caused it. **Important note:** the actual mistake may not be on the line highlighted by MATLAB. In the example above, the mistake is on Line 1 since the sixth value is not added to the vector. However, the error would be generated by line 2.

Finally, a **semantic error** (sometimes called *logic error*) is where MATLAB is able to run all the code without generating an error message but does not produce the required result. For example:

```
1 function vel = calcVelocity(dist,time)
2 vel = dist*time;
```

Clearly, the second line should be `vel = dist/time`, but since MATLAB has no way of knowing this, it will run the code anyway and return the wrong result from the function. Semantic errors have potential to be the most problematic since the programmer does not always notice that (s)he has made such a mistake (the advice in the "avoiding mistakes" section above should help minimise the number of semantic errors you make).

Breakpoints

The error in the `powerseq` function above is of the semantic type. Suppose we wrote the code, checked the outputs with some manually calculated values and realised a semantic error was present. At this stage we would examine the code to try and find the error. In this case, you can probably identify the error quickly by simple visual inspection but in more complicated functions and scripts it can often be very difficult to find the mistake and so "breakpoints" can be useful.

To illustrate breakpoints, first copy the `powerseq` code above into the MATLAB editor and save as `powerseq.m` (you do not need to copy the comments). Type `powerseq(2)` into the Command Window. You should see the following:

```
1 >> powerseq(2)
2 ans =
3     2.0000    2.8284    4.7568    4.7568   10.3747   33.4168
```

Because of the deliberate error on line 14, the third and fourth values are identical. To investigate the error in more detail we will place a breakpoint on the line that starts "`y(3) = y(2)...`" (line 13 in the above code but may be different in your own version). To do this, either position the cursor on this line and press F12 or click the "—" symbol at the start of the line (just to the right of the line number in the editor window). The "—" symbol should turn into a red disc.

Type `powerseq(2)` into the Command Window again. This time, MATLAB will only run the function up to the breakpoint and then pause, retaining the variables that the function is currently using in the Workspace. A green arrow appears next to the breakpoint indicating that the execution of the code is paused at the start of that line and the usual "`»`" symbol in the Command Window is replaced by "`K»`".

At this point, you can do several things to investigate the code. Firstly, you can refer to the variables in the Workspace to see if they have the correct values at this point. Secondly, you can type commands into the Command Window (e.g. simple computations to help verify what the variables should be or even to modify their values). Thirdly, you can press the *Step* button under the *EDITOR* tab near the top of the main MATLAB window to move to the next line and continue your investigation. Fourthly, you can press the *Continue* button to let MATLAB finish running the code (or move to the next breakpoint if you added more than one). Fifthly, you can click *Quit Debugging* to terminate the program. There are several other features available to you which you can investigate if you wish but are not essential for the time being.

Chapter 3: Commands

Command	Description
`function`	Defines an M-file as a function
`==, ~=, >, >=, <, <=`	Relational operators
`&, |, ~`	Logical operators
`if`, `elseif`, `else`	Logical constructions
`disp`	Display specific output string
`error`	Terminate a program and display an error message in red text
`warning`	Display a warning in the command window in orange text
`abs`	Returns the positive value of a variable

Table 3.3: Commands introduced in Chapter 3.

Chapter 3: Worksheet

1. Write a *function* that will convert a value from feet to metres given that 1 foot is 0.3048 metres. Your function should take one input argument (value of feet) and convert this to metres (your output argument). Call your function ft2m so that you can enter the following into the *Command Window* and get the correct answer.

```
>> ft = 35;
>> metres = ft2m(ft)
>> metres =
           10.668
```

   Your function should include the header section with appropriate information and a variable dictionary.

   N.B. Your function **must** work when called from the Command Window. Otherwise, you have not written a function but a script.

2. *The absolute pressure at the bottom of a tank that is open to the atmosphere is governed by the equation:

$$p_{\text{abs}} = \rho g h + p_{\text{atmos}}$$

   where

$$\begin{aligned} p_{\text{abs}} &= \text{absolute pressure at the bottom of a tank} \\ \rho &= \text{density of the fluid} \\ g &= \text{gravitational constant} = 9.81\,\text{N/kg}\ (= 9.81\,\text{m/s}^2) \\ h &= \text{height of liquid level in tank} \\ p_{\text{atmos}} &= \text{atmospheric pressure} \end{aligned}$$

   Write a function to determine p_{abs} that takes ρ, h and p_{atmos} as inputs. That is, your function should accept the syntax:

```
>> pabs = calculatePressure(rho,h,patm)
```

   Again, and as with all functions, your code should work when called from the Command Window. Note also that there is no need to include an input argument for g since this is a constant that can be "hard-coded" into the function.

   Time permitting, add one or two error checks at the start of the code.

3. *What will the following pieces of code print? Determine by hand before checking your answer in MATLAB:

a)
```
a = 10;
if a ~= 0
    disp('a is not equal to zero')
end
```

b)
```
a = 10;
if a > 0
   disp('a is positive')
else
   disp('a is negative')
end
```

c)
```
a = 5;
b = 3;
c = 2;
if a < b*c
    disp('Hello world')
else
    disp('Goodbye world')
end
```

4. *For what values of the variable `a` will the following pieces of code print `Hello world`?

a)
```
if a >= 7 & a < 7
    disp('Hello world')
else
    disp('Goodbye world')
end
```

b)
```
if a < 7 | a >= 3
    disp('Hello world')
else
    disp('Goodbye world')
end
```

5. Write a function that accepts two input scalars that represent the x and y coordinates of a point, respectively. The function should determine if the point lies within a circle of radius equal to 1 (arbitrary units). Where this is the case, the (only) output of the function should be set to 1. Otherwise the output should

be set to 0. In addition, the function should add text to the command window indicating whether the point was inside or outside of the circle.

As an optional more challenging problem, you may wish to get MATLAB to plot the circle and the point on a graph.

6. Write a function to solve a quadratic by accepting the three coefficients as inputs and returning the roots in an output.

   Specifically, your function should do the following:

   a) Accept the three coefficients of a quadratic equation as inputs.

   b) Determine the value of the discriminant.

   c) If the value of the discriminant is positive, use the `disp` command to tell the user that the equation has two real roots, and output them in a two-element column vector.

   d) If the value of the discriminant is zero, tell the user that the equation has one real root, and output the value as a scalar.

   e) If the value of the discriminant is negative, tell the user that no real roots exist and output `NaN`.

   f) Plot the graph of the equation with labels, a title and with the grid on.

4 Loops

The construct of loops (also known as iterations) offer a means to alter the way MATLAB flows through a script or function, and is particularly useful for repeating certain commands. You might want to repeat the same commands, changing the value of a variable each time, for a fixed number of iterations. An example would be to use the trapezoid rule to numerically determine the approximate area under a curve (numerical integration). In MATLAB, this involves a `for` loop.

Alternatively, you might want to repeat the same commands, changing the value of a variable until a certain condition is reached. For example, for turbulent flow in a pipe, to find the friction factor when the flow rate or pipe diameter is unknown involves an iteration until the value of the friction factor changes by a small amount (you may have covered this in fluid dynamics). Other examples are numerical methods to determine the roots for an equation. This can be achieved in MATLAB using a `while` loop.

The scripts used in this chapter can be downloaded from the online course resources.

4.1 The `for` loop

4.1.1 Syntax

A `for` loop is used to repeat a command or series of commands for a fixed number of times. The basic syntax for a `for` loop is as follows:

```
1 for variable = first:increment:last
2     commands
3 end
```

- **Line 1:** This line contains the mandatory `for` command followed by the *loop counter*. The loop counter works in a similar manner to defining a vector: `first` indicates the starting number of the counter; `increment` indicates the increment (and can be neglected if the increment is 1); and `last` is the ending value for the counter. So, as an example*, we could type `for k = 0:5:15`. In this case, for the first iteration, `k = 0`; on the second loop, `k = 5`; on the third iteration, `k = 10`; and on the fourth iteration, `k = 15`. At this point, MATLAB would finish iterating and move on to following commands (after the `end` statement). As another example, if we wanted the loop to run 50 times, we could write: `for n = 1:50`, and the loop would iterate, with `n` increasing in steps of 1, from 1 to 50. This is particularly useful for filling a vector with values resulting from a series of commands (see the complex example below.)

- **Line 2:** Contains the command or series of commands to run during each iteration of the loop.
- **Line 3:** As with `if-else` statements, MATLAB needs to know where the loop finishes, and this is indicated by the `end` statement.

4.1.2 Simple Example

As a very simple example*, let us write a `for` loop which simply displays the value of `x`:

```
1 for x = 1:1:9
2     x
3 end
```

In line 1 the `for` loop is defined to run with the counter `x` to start at 1, and increment up in steps of 1 until `x = 9`. Line 2 displays the value of `x` in the command window, and line 3 ends the loop.

Running this code would produce the following:

```
1  x =
2       1
3  x =
4       2
5  x =
6       3
7  x =
8       4
9  x =
10      5
11 x =
12      6
13 x =
14      7
15 x =
16      8
17 x =
18      9
```

So, when the loop is executed, initially the value of 1 is assigned to `x`, and then the body of the loop is executed. The value of `x` is then incremented, the body of the loop is executed again, and so on until `x` is 9. At this point, the loop is executed for the final time and then ends.

4.1.3 Complex Example

Let us use a `for` loop to come up with the data necessary to plot the equation

$$y = x^2.$$

You are probably thinking that we don't need to use a loop to do this—we did this in Chapter 2 when we looked at how to visualise data. You are right! This is a particularly poor way of doing the same thing (it is taking much more computational time), but illustrates the way loops can be used.

The script follows:

```
1  % complex_for_example.m
2  % Script to plot a quadratic equation
3  %
4  % Benjamin Drew. Last update: 21/07/2012
5
6  % Variable Dictionary
7  % x          Independent variable
8  % y          Dependent variable
9  % k          Loop counter
10
11 clear; % Clear all variables from workspace
12 clc; % Clear command window
13 close all; % Close all previous figure windows
14
15 x = -10:0.01:10; % Define range for x
16
17 for k = 1:length(x)
18      y(k) = x(k)^2;
19 end
20
21 % Plot data
22 plot(x,y)
23
24 % Add labels
25 xlabel('x'); ylabel('y');
```

Let's examine what is going on line-by-line.

- **Lines 1–13:** These lines are the standard header, including description, author, date and the clearing commands. See previous chapters for more details.
- **Line 15:** Here, we define our independent variable (the horizontal axis for our plot), as a range from -10 to 10 with increments of 0.01.

- **Line 17:** We now start the `for` loop. We specify a counter variable, in this case `k` (it doesn't matter what you use—you could have picked `n`, or `t`, or even `counter`). In this example, we want the commands to be performed for each value of `x`. Referring to line 15, we can deduce that there are 2001 elements of `x`, so we could have written `for k = 1:2001`. However, it is better to use `length(x)` (see Section 1.3.8) so that if we later change the step size or range of `x` on line 15, we do not need to make a related change to line 17.
- **Line 18:** This is the command that we want to iterate. Here, we define our dependent variable `y` as `x^2`. But since we are doing this in a loop, we can do it in an element-by-element manner, indicated by the `k` term in parentheses after the variable name[1]. In the first iteration, `k = 1`, so the first element of `y` will be set to the first element of `x` squared. In the second iteration, it sets the second element of `y` to be the second element of `x` squared, and so on, until it reaches the final value where the last element of `y` is set to the last element of `x` squared.
- **Line 19:** Marks the end of the loop, i.e. the portion of code that will be repeated.
- **Lines 21–25:** ... plot the graph using the standard plotting commands (see Chapter 2 for more details).

4.1.4 An Even More Complex Example

For this example, you may wish to refer to the section of your mathematics notes on series, in particular the sum of an infinite series. One example given is:

$$\sum_{k=1}^{\infty} \frac{5}{k} = 5 + \frac{5}{2} + \frac{5}{3} + \frac{5}{4} + \dots$$

Your maths notes go on to say that it is beyond the scope of this course to determine whether or not an infinite series has a finite sum. Well, let's ignore that, and calculate what the first thousand terms add up to[2]. So, we revise the equation to be:

$$\sum_{k=1}^{1000} \frac{5}{k} = 5 + \frac{5}{2} + \frac{5}{3} + \frac{5}{4} + \dots + \frac{5}{999} + \frac{5}{1000}$$

To compute this in MATLAB, we could write the following:

```
1 % even_more_complex_for_example.m
2 % Script to determine the finite value of a sum of a series.
3 %
4 % Benjamin Drew. Last update: 21/07/2012
```

[1] Don't forget, this is how we index numbers in vectors and matrices. See Section 1.3.6.
[2] It turns out that for this equation, the sum of the infinite series does not converge to a single value.

```
5
6  % Variable Dictionary
7  % total      Final value
8  % k          Loop counter
9
10 clear; % Clear all variables from workspace
11 clc; % Clear command window
12
13 total = 0; % Initialise the variable total as 0.
14
15 for k = 1:1000
16      total = total + 5/k; % Add previous value of sum to current value.
17 end
18
19 format long; % Set MATLAB to show 15 digits of precision.
20 total % Display the value of sum to the command window.
```

Again, let's examine this line-by-line:

- **Lines 1–11:** Standard header, as before.
- **Line 13:** In this line, we want to initialise our sum variable, `total` to be zero. You'll see why in the description for line 16.
- **Line 15:** Here, we set up the `for` loop using the limits of our sum command in the equation: the counter, `k` increments from 1 to 1000 in steps of 1.
- **Line 16:** On this line, we set the value of `total` to be the previous value of `total` plus $5/\texttt{k}$. So, in the first iteration, $\texttt{total} = 0 + \frac{5}{1} = 5$ (hence why we needed to initialise `total` in line 13). In the second iteration, the equation would be $\texttt{total} = 5 + \frac{5}{2} = 7.5$. In the third iteration the equation would be $\texttt{total} = 7.5 + \frac{5}{3} = 9.1667$, and so on, until `total` becomes the previous value $+ \frac{5}{1000}$.
- **Line 17:** Marks the end of the loop.
- **Line 19:** We want a precise answer, so we set MATLAB to display full 15 digit precision using the `format long` command.
- **Line 20:** Displays our sum total in the command window.

Suggestion: time-permitting, you may wish to try some of the debugging features on this code, as discussed in Section 3.4.2. In particular, you could remove the semicolon from Line 16 to observe the progression of `total` and/or add a breakpoint at the start of this line.

4.2 The `while` loop

A `while` loop is similar to the `for` loop in that it lets you repeatedly execute a series of commands. The difference is that whereas a `for` loop needs to you define the *number of iterations required*, a `while` loop continues until a *certain condition is met.*

4.2.1 Syntax

The general syntax is as follows:

```
1 while condition_is_true
2     commands
3 end
```

- **Line 1:** This line contains the mandatory `while` command followed by the condition that must be true for the loop to operate. As such, this condition must be some form of conditional statement or logical expression, as discussed in Chapter 3. For example, the loop could run while `x` is less than 10 (assuming that x is increasing within the loop.) The point at which `x` becomes greater than or equal to 10, the loop terminates and MATLAB continues with any commands following the loop. Any variables used in this statement must be defined *before* the loop starts.
- **Line 2:** This line contains the command or series of commands to run during each iteration of the loop. For the loop to function correctly (and end!—see later), the variable used in the logical expression in line 1 needs to be modified.
- **Line 3:** This line contains the `end` command which must always be used at the end of a loop to close it.

4.2.2 Simple Example

As a very simple example, let us write a `while` loop for which the value of variable `x` is incremented. (Compare this to the simple example we used for the `for` loop in Section 4.1.2.)

```
1 x = 1;
2 while x<10
3    x
4    x = x + 1;
5 end
```

We shall now go through this code line-by-line:

- **Line 1:** On this line, we initialise (assign for the first time) a value of 1 to the variable `x`. This is outside of the `while` loop as firstly, `x` is used as the condition for the loop to run (it needs to know what `x` is), and on line 4, `x` is used in an equation, so again, a value needs to be defined.

- **Line 2:** Here, the `while` loop starts, and the condition of `x<10` is specified as the logical expression that must remain true for the loop to run. As soon as `x` fails to meet this condition at the start of an iteration (i.e. when it is 10 or greater), the loop will stop running and MATLAB will continue with any commands that come after the loop.

- **Line 3:** This line simply displays the current value of `x` in the command window.

- **Line 4:** On this line, the value held in `x` is increased by 1. This value must increase every time, otherwise the condition of `x<10` will always be true, and the loop will run indefinitely.

- **Line 5:** Indicates the end of the loop.

So, running the above code, the output should look like:

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

Note that if you have accidentally written some code that has resulted in an infinite loop, then you can stop MATLAB from calculating by pressing Ctrl+C.

4.2.3 Complex Example

As this next example, let us use a while loop to produce the same graph of

$$y = x^2$$

Again, using a `while` loop is an inefficient way or producing this graph, but it demonstrates how it works, and ways that it differs from a `for` loop. Compare this code with that shown in Section 4.1.3:

```
1  % complex_while_example.m
2  % Script to plot a quadratic equation
3  %
4  % Benjamin Drew. Last update: 23/07/2012
5
6  % Variable Dictionary
7  % x          Independent variable
8  % y          Dependent variable
9  % k          Loop counter
10
11 clear; % Clear all variables from workspace
12 clc; % Clear command window
13 close all; % Close all previous figure windows
14
15 x = -10:0.01:10; % Define range for x
16 k = 1;
17
18 while k <= length(x)
19     y(k) = x(k)^2;
20     k = k + 1;
21 end
22
23 % Plot data
24 plot(x,y)
25
26 % Add labels
27 xlabel('x'); ylabel('y');
```

Let's go through this line-by-line, as before. The lines highlighted below are the differences between this code and that used with a `for` loop.

- **Lines 1–13:** Header and clearing commands.
- **Line 15:** Here, we define our independent variable as ranging from -10 to 10 with an increment of 0.01.

- **Line 16:** Here, we have to initialise our counter `k`, as we refer to it in the `while` loop statement.
- **Line 18:** Here, we start the `while` loop, stating that we want it to run only while the value of `k` is *less than or equal to* the length of `x`. Remember, the `length` command returns the number of elements in the vector.
- **Line 19:** This is the command we want to iterate. Again, we want to do this in an element-by-element manner, so in the first iteration (when `k = 1`), the first element of `y` will be the first element of `x` squared, and so on.
- **Line 20:** For `k` to increase every iteration, we need a command within the loop to increment `k` by 1 everytime the loop runs.
- **Line 21:** Indicates the end of the loop.
- **Lines 23–27:** These lines plot the graph using the standard plotting commands.

4.2.4 An Even More Complex Example

Again, we shall use the `while` loop to replicate the same example used in Section 4.1.4, where we calculated the sum series:

$$\sum_{k=1}^{1000} \frac{5}{k}$$

In MATLAB the code to do this using a `while` loop is:

```
1  % even_more_complex_while_example.m
2  % Script to determine the finite value of a sum of a series.
3  %
4  % Benjamin Drew. Last update: 23/07/2012
5
6  % Variable Dictionary
7  % total      Final value
8  % k          Loop counter
9
10 clear; % Clear all variables from workspace
11 clc; % Clear command window
12
13 total = 0; % Initialise the variable sum as 0.
14 k = 1;
15
16 while k<=1000
17      total = total + 5/k; % Add previous value of sum to current value.
18      k = k + 1;
```

```
19 end
20
21 format long; % Set MATLAB to show 15 digits of precision.
22 total % Display the value of sum to the command window.
```

- **Lines 1-13:** These lines are the header, the clearing commands, and initialising the variable `total` as 0.
- **Line 14:** We have to give our counter, `k` an initial value, as it is being used in the `while` loop statement.
- **Line 16:** This starts the loop, which will run while `k` is *less than or equal* to 1000.
- **Line 17:** This computes the value for `total` which is the sum of the previous value of `total` and $\frac{5}{k}$. As with the `for` loop, in the first iteration this is $0 + 5$; the second will be $5 + \frac{5}{2}$, and so on.
- **Line 18:** Like the previous example, `k` must increment up by 1 every iteration of the loop, and will do this until the loop terminates (i.e. when $k > 1000$.)
- **Lines 19:** Indicates the end of the loop.
- **Lines 21-22:** Sets MATLAB to display 15 digits of precision and displays the value of `total` in the command window.

You should see that the final value is the same as with the example shown in Section 4.1.4.

4.3 Which Loop Should You Use?

The examples shown above have solved the same problems using both a `for` and a `while` loop. In some cases, a `for` might be more suitable (if you know how many iterations you require), whereas a `while` loop might be more suitable for other problems (particularly if you are unsure of how many iterations are required). The examples above are generally best suited to a `for` loop. However, the example questions in the worksheet should help to illustrate the relative merits of each type: including some cases where the `while` loop is superior.

4.4 Protected files

There are occasions where you have written a MATLAB program (script or function) that you want to allow others to use but would prefer them not to see the actual text of the code itself (e.g. to protect intellectual property). On these occasions, the p-code feature is useful.

To illustrate p-code, enter the following code into the editor and save as `powerseq.m` (don't bother about the comments). Note that this function serves the same purpose as that used in Section 3.4.2 but is written using a `for` loop. Make sure you are clear about how this code works.

```
function y = powerseq(yFirst)
% POWERSEQ(yFirst) Calculates the first six terms of the sequence
%   y_n = y_(n-1) ^ 1.5 for a given y_1 value
%   Gary Atkinson. Last update: 14/08/2014
%
%   Variable dictionary:
%   y1    First value for y
%   y     Output, corresponding to first six elements of the sequence

y = zeros(1,6); % initialise y vector
y(1) = yFirst; % set first element to same value as input (y_1)
for k = 2:6 % go through each other element
    y(k) = y(k-1).^1.5; % compute kth value
end
```

Make sure the MATLAB path is set to the location of the `powerseq.m` file and type

```
pcode powerseq
```

into the Command Window. This should generate a file called `powerseq.p` in the same folder. You (or anyone else) can now call the p-code file from the Command Window or from another function or script and it should work in the same way as the `.m` file, provided that it is located in the MATLAB path. You can therefore distribute the p-code without anyone being able to read the MATLAB commands that it contains.

Do not try to open a p-code file that someone has given you as they are, by there very nature, uneditable. Instead, just place it in the same directory as the other code you are working on and use it as you would any other function.

Chapter 4: Worksheet

1. *Write a script to perform the following:
   a) A `for` loop that multiplies all even numbers from 2 to 10.
   b) A `while` loop that multiplies all even numbers from 2 to 10.
   c) A `for` loop that assigns the values 10, 20 , 30, 40 and 50 to a vector.
   d) A `while` loop that assigns the values 10, 20 , 30, 40 and 50 to a vector.
   e) Is there a simpler way to do c) and d) avoiding loops?

2. *The factorial of a positive integer is defined as:

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 1$$

   where $n! = 1$ when $n = 0$. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

   Use a `for` loop to write a *function* able to compute and print factorials. Your function should take a single input corresponding to the integer that we wish to calculate the factorial of. Your function should return the value of the factorial in the case where the input is a non-negative integer, return 1 for the case where the input is exactly zero, or an error message otherwise. (Give your function a name other than `factorial.m`, as there is a built-in function called `factorial`.)

   [Hint: you may find it easier to write a script file first and convert to a function when that is working.]

3. An infinite number of engineers walk into a bar. The first orders a beer. The second orders half a beer. The third orders quarter of a beer. The fourth orders one eighth of a beer. The bartender interrupts saying "You're all idiots" and pours the engineers two beers. Use MATLAB to show how the bartender knew to pour just two beers. [Hint: think about this as an infinite sum problem.]

4. Consider the geometric series below[3]:

$$\sum_{k=0}^{\infty} 4^{-k} = 1 + \frac{1}{4} + \frac{1}{4^2} + \frac{1}{4^3} \ldots$$

   Analytically, the solution to this sum is:

$$\frac{1}{1-r} = \frac{1}{1-1/4} = \frac{4}{3}$$

   where r is the common ratio of the series.

[3] This is called the Archimedes Quadrature and relates to the area between a line and a parabola. See `http://en.wikipedia.org/wiki/The_Quadrature_of_the_Parabola`

The code below proves this numerically. Examine the code carefully and make sure you understand each line. The code brings together various techniques developed during the course and contains one or two new concepts you can work out for yourself. Note also that the loop here is a `while` loop. This is much more convenient than a `for` loop in this case. Make sure you understand why.

(The code may be downloaded from the online course resources.)

```
function [total,k] = archimedes(tol)
% [total,k] = archimedes(tol)
%  Calculate the sum, total, of the archimedes quadrature up to a
%  tolerance of tol (recommended tol for testing = 0.00001)
%  Most comments have been removed for the sake of an exercise
%
%  Gary Atkinson. Last update: 20/08/2013

total = 0;
done = false;
k = 0;
series = [];

while ~done
    totalnew = total + 1/(4^k);
    if (totalnew-total)/totalnew < tol
        done = true;
    else
       total = totalnew;
    end
    k = k + 1;
    series = [series total];
end

plot(0:k-1,series,'o-','linewidth',2);
grid
xlabel('Iteration number, \itk');
ylabel('Sum to \itk\rm^t^h iteration');
title('\bfConvergence of the Archimedes'' quadrature of parabola');
```

5. The Fibonacci sequence is defined such that:

$$F_n = F_{n-1} + F_{n-2} \qquad \text{where} \quad F_0 = 0 \quad \text{and} \quad F_1 = 1$$

Write a script that computes and displays the first N Fibonacci numbers, where N is defined as an integer of your choice near the start of the script.