Learning to use MATLAB for CATAM project work

Version 1.31

Faculty of Mathematics, University of Cambridge

This document can be downloaded from

http://www.maths.cam.ac.uk/undergrad/catam/MATLAB/manual/booklet.pdf

Please email suggestions, comments and corrections to catam@maths.cam.ac.uk

March 17, 2022

Contents

1 Introduction **1**

1.1 Suggestions, comments and corrections 1

1.2 Other documentation 1

2 Using the Computers on the Mathematics MCS **3**

2.1 Printing 3

2.2 Files and folders 4

2.2.1 Backing up your files 4

2.3 Further documentation 5

3 Introduction to MATLAB **6**

3.1 Starting MATLAB 6

3.2 The basics 7

3.3 Vectors 9

3.4 Plots 10

4 Programming in MATLAB **11**

4.1 A simple program 11

4.1.1 Programming tips 13

4.1.2 My program is running out-of-control or not responding 13

4.2 Improving the output 13

4.3 Reducing typing and a noddy guide to functions 15

4.3.1 Script files (a.k.a. M-Files) 15

4.3.2 Functions 18

4.4 Exercises 21

5 Help! **23**

6 Vectors and matrices **24**

6.1 Creating matrices 24

6.2 Manipulating matrices 26

6.2.1 Exercise 28

7 A few more functions **29**
7.1 Scalar functions . . . 29
7.2 Vector functions . . . 29
7.2.1 Exercise . . . 30
7.3 Matrix functions . . . 30
7.4 Random number generation . . . 30

8 Program flow control **31**
8.1 The `if-else` control structure . . . 31
8.1.1 Exercises . . . 33
8.2 The `while` control structure . . . 34
8.2.1 Exercise . . . 36
8.3 The `switch-case` control . . . 36
8.3.1 Exercise . . . 38

9 Elementary graph plotting **39**
9.1 The `plot` command . . . 39
9.1.1 Exercise . . . 41
9.2 Other 2D graphs . . . 42
9.2.1 Exercise . . . 43
9.3 Multiple figures and plots . . . 44
9.4 Saving your figures . . . 45
9.4.1 Exercise . . . 46
9.5 3D graphs . . . 46

10 Random number generation **48**

11 Symbolic manipulation **49**

12 Sets and set operations **51**

13 Troubleshooting **52**

13.1 Errors and debugging 52

13.1.1 Debugging from the Editor 52

13.1.2 Command line tools 53

13.2 Timing 54

14 This and that **55**

14.1 Programming style 55

14.1.1 Indentation 55

14.1.2 Vertical alignment 56

14.2 Some terminology 56

15 Sample project: Fibonacci numbers **57**

15.1 Definitions 57

15.2 Recursion versus iteration 57

15.3 The size of Fibonacci numbers 57

16 Acknowledgments **58**

A Using Windows: Basics **59**

A.1 Logging in 59

A.2 Windows basics 60

A.3 The start menu and task bar 60

A.4 Window elements 61

A.5 Files and folders 62

A.6 Logging out 62

B A generalised `printsquares` code **63**

C Index of functions in this booklet **64**

1 Introduction

This guide is intended to help you to learn how to program with MATLAB whether you are new to programming, or whether you have programmed before but would like to know more about MATLAB.

You should skim over or skip sections containing material that is already familiar to you. However, you should realise that writing programs is the only way to learn a programming language — only by typing in and running your own programs will you learn to translate mathematics into computer algorithms and thence into computer programs. Although it might be tedious to type in long programs, instead of just loading them in, you can learn a lot in the process of typing in, running and changing the programs in this guide.

If you are looking for a quick start to MATLAB, you may also skip sections which appear on a grey background. Such sections provide more advanced material on MATLAB and programming in general, and may be more useful on a second pass through the tutorial.

§2 covers the use of the Desktop Services including information about files and printing. Additional information for those who are new to Windows is included in Appendix A.

The remaining sections cover learning to use MATLAB. The early sections concentrate on programming techniques by deliberately using examples that are mathematically very simple. You are encouraged to modify the example programs and to write your own programs.

1.1 Suggestions, comments and corrections

Unfortunately there are likely to be a few infelicities in this booklet, not in the least because *The MathWorks*, the suppliers of MATLAB, tinker with the graphical interface. For the benefit of those who follow, **please email suggestions, comments and corrections** (no matter how minor) to `catam@maths.cam.ac.uk`. Thank you.

1.2 Other documentation

A short guide like this one can only cover a small subset of the MATLAB language. There are many other guides available on the net and in book form that cover MATLAB in far more depth. Further:

- MATLAB has its own built-in help and documentation.
- *The MathWorks* provide an introduction *Getting Started with MATLAB*. You can access this by 'left-clicking' on the `Getting Started` link at the top of a MATLAB *'Command Window'*. Alternatively there is an on-line version available at[1]

  `http://uk.mathworks.com/help/releases/R2017b/matlab/getting-started-with-matlab.html`x

[1] These links work at the time of writing. Unfortunately *The MathWorks* have an annoying habit of breaking their links.

with a printable version available from

`http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/getstart.pdf`

- Mathworks also provides MATLAB Onramp, an interactive tutorial on the basics which does not require MATLAB installation:

  `http://uk.mathworks.com/support/learn-with-matlab-tutorials.html`

- *The MathWorks* also provide links to a whole a raft of other tutorials

  `https://uk.mathworks.com/support/learn-with-matlab-tutorials.html`

  In addition their *MATLAB* documentation page gives more details on maths, graphics, object-oriented programming etc.; see

  `https://uk.mathworks.com/help/matlab/index.html`

- There is also a plethora of books on MATLAB. For instance:

  (a) *Numerical Computing with MATLAB* by Cleve Moler[2] (SIAM. 2nd Ed. 2008, ISBN 978-0-898716-60-3). This book can be downloaded for free from

  `http://www.mathworks.co.uk/moler/chapters.html`

  (b) *MATLAB Guide* by D.J. Higham & N.J. Higham (SIAM, 2nd Ed. 2005, ISBN 0-89871-578-4).

[2] Cleve Moler is chief mathematician, chairman, and cofounder of MathWorks.

2 Using the Computers on the Mathematics MCS

[This section is under review, and links and information relating to printing and use of MCS facilities may not apply in the current academic year.]

This section assumes that you are sitting in front of one of the Desktop Services (DS) computers, such as those in the CATAM room, GL.04, in the basement of Pavilion G at the CMS. It also assumes that you know your UIS username and password.[3] If you have forgotten your password see `http://www.ucs.cam.ac.uk/desktop-services/accounts/`. The following instructions should work during the **academic year 2017/18**.

The machines in GL.04 can run either Linux (Ubuntu) or Windows; the instructions here assume that you will be running Windows. Further details for logging in and out, as well as additional information mainly for users new to Windows, can be found in Appendix A.

2.1 Printing

Undergraduate mathematics students are given free print credit at the start of each academic year that allows them to print to the black-and-white and colour printers in GL.04. The names of the two Desktop Services print queues in GL.04 (for which there is free print credit) are:

Printer Name	Description
Maths_Pav-G_BW	An HP LJ4350 black-and-white A4 printer
Maths_Pav-G_Col	An HP LJ3700 colour A4 printer

For reasons of cost please only use the colour printer when colour output is essential. The cost of printing can be found at `https://help.uis.cam.ac.uk/devices-networks-printing/ds-print/users/ds-print-payment`. Note that your free credit only applies to the printers in GL.04, not to other printers on the Desktop Services network.[4]

The amount of credit allocated depends on the year of study and is enough to cover your yearly needs. However, if you should run out for some reason, you are asked to complete a form that can be found at `http://www.maths.cam.ac.uk/computing/mcs/MCS-print.html`, where you should explain why you have used up your allotted credit; the signature of your Director of Studies in support is also required. Your application will then be reviewed[5] and, if successful, extra credit will be added to your account.

Please note that the free print credit provided by the Faculty of Mathematics is different to the printing credit that can be bought through the Desktop Services common balance scheme (see `https://help.uis.cam.ac.uk/devices-networks-printing/ds-print/users/ds-print-payment`). If you use the DS printers in Faculty of Mathematics then credit

[3] Your DS username is your CRSid, which is the same as that for Raven and Hermes. If you joined the University before February 2014, when the combined UCS Password was introduced, your DS password will not be the same as your password(s) for Raven and for Hermes (unless you have changed them to be the same).

[4] In fact the free print credit also applies to the printers in the Part III room, but you will not need to use these.

[5] All print activity is logged, so please do not use your printing credit for anything other than your mathematical studies.

is, at first, deduced from your free print credit until it expires. However, you should be aware that after that credit is used up, future use is deducted from any DS common balance (since the DS printers in Faculty of Mathematics are also part of the DS common balance scheme).

If either of the printers have any problems, please email `printer@maths.cam.ac.uk` explaining the nature of the problem, the printer in question and any error messages that may be displayed on the screen.

2.2 Files and folders

The Desktop Services PCs have several disk drives for storing information — a USB drive (A:) which accepts USB memory sticks, a writeable DVD drive (D:), and an internal drive (C:) which contains Windows. Additional 'networked' drives are held on on fileserver computers. Drive U: holds your own files, while you will use drive X: for project submission at the beginning of the Lent and Easter terms.

When you are working with multiple files it is convenient to organise related files into groups called **folders** or directories. Applications access files by default in the current folder which can be changed using the application's File Menu. A file outside the current folder can be specified by a directory path preceding the file name; for instance

U:\MATLAB\Project_0-1\bisection.m

refers to a file called **bisection.m** that is in a sub-folder **Project_0-1** of the **MATLAB** folder.

On-line help for Windows (as opposed to MATLAB) is accessible by clicking on the **start** button then on **Help and Support**. Regrettably, it can then often be a challenge to track down precisely what you need to know. One good strategy is to type some relevant words into the **Search** window and click on the white arrow in the green square may get you some helpful information. In particular, you may wish to search for instructions on creating folders, listing folders' contents, copying and deleting files, formatting disks etc.[6]

To start MATLAB go to the **start menu**. Then click on **All Programs** and from the menu choose **Teaching Packages**, then on **Catam**. From the small menu which is then displayed click on **MATLAB** to start.

2.2.1 Backing up your files

As noted in `https://help.uis.cam.ac.uk/devices-networks-printing/managed-desktops/ds-filestore/ds-filestore-backup`, you are responsible for keeping backup copies of your files. The fileservers are backed up by the Computing Service, but not in such a way that individual files can be readily retrieved. It is very easy, as many people have found out the hard way, to lose a file, for instance by accidental deletion or overwriting.

[6] In fact Windows **Help** is so abysmal that it is often quicker to search for an answer using `Google` or your favourite search engine.

You cannot assume that a file you have moved to the *Recycle Bin* on a particular machine will still be available on that machine when you come back ten minutes later.

The easiest way to back up files from your filespace on the Desktop Services cluster is to make regular copies to a USB stick, DVD etc., and keep the backups in a safe place, labelled and dated.

Some of you may find a convenient alternative is to sign up for a free account on a cloud computing resource such as Dropbox at `http://www.dropbox.com`. This allows files to be transferred using 'drag and drop' via any internet connection.

It is also a good idea to make sure you make up-to-date (and dated) printouts of documents under development at intervals; in case of major disaster it is usually possible to use a scanner to recreate a document.

2.3 Further documentation

The Computer Service provides further information on the Desktop Services facilities that are available throughout the University, at
`https://help.uis.cam.ac.uk/devices-networks-printing/managed-desktops`.

3 Introduction to MATLAB

We recommend that you use **MATLAB** for the Computational Projects. However, you are not required to use MATLAB, and if you choose you could program in Mathematica, Maple, Scilab, C, C++, C#, Python, or any other language.

One of the advantages of MATLAB is that it has an 'environment' which includes an editor and a debugger. However, even if you decide to use MATLAB you need not use either of these (e.g. you could use the Emacs editor instead of the integrated editor).

MATLAB is available free of charge from the Faculty of Mathematics for installation on your own personal computer running Windows, Mac OS or Linux.[7] MATLAB is also pre-installed on the computers in the CATAM MCS room in the CMS, and on other MCS computers (including those at a number of Colleges and that in the Betty and Gordon Moore Library); for a list of University and College Desktop Services computer cluster sites see

`https://help.uis.cam.ac.uk/devices-networks-printing/managed-desktops`

3.1 Starting MATLAB

If you are sitting in front of a MCS Windows computer, you can start MATLAB from the start menu as described in §2.2. If you have installed MATLAB on your own machine there may be an icon on the desktop from which you can start MATLAB; alternatively there will be an entry for MATLAB in your 'start' or 'finder' menu. After a short while a window should open with three or four panes.[8]

1. One pane is labelled '*Command Window*'. This is the pane in which you will type MATLAB commands.
2. One pane is labelled '*Current Folder*'[9], and lists the files in that directory.
3. One pane is labelled '*Workspace*', and lists the variables that you have defined. Displayed variables may be viewed, manipulated, saved, and cleared.
4. One pane is labelled '*Command History*'. This pane lists your previous commands. *Inter alia* you can execute a previous command by double-clicking on it.

Henceforth, unless stated otherwise, you should type the MATLAB commands in this guide into the '*Command Window*'.

[7] You can install a copy of MATLAB for non-commercial use on your own personally-owned computer by downloading and the installation files from the Faculty website: see instructions at

`http://www.maths.cam.ac.uk/undergrad/catam/software/MATLAB.html`.

[8] Please note the 'should'. Depending on how MATLAB has been configured you may end up with one, two, three or four panes.

[9] In older versions of MATLAB this was the '*Current Directory*' pane.

3.2 The basics

MATLAB includes all the operations and functions you would find on a calculator. It will attempt to evaluate mathematical expressions that you input. Into the *'Command Window'* type

```
>> 2+2
```

and press the return key. MATLAB should print the line

```
ans = 4
```

Note that all inputs in MATLAB are terminated by hitting the return key. It is assumed you will do this from now on. Type

```
>> cos(pi)
```

MATLAB returns

```
ans = -1
```

as '`pi`' is the built-in expression for π. Now type

```
>> pi^2;
```

In this instance, MATLAB evaluates π^2 but the semi-colon at the end of the line causes the output to be suppressed. Typing

```
>> pi^2
```

gives the expected output

```
ans = 9.8696
```

Note that this is not the actual precision to which MATLAB has calculated the answer, it is only the output precision.

Of course, MATLAB is much more than a calculator. The first step to programming in MATLAB is learning how to define and use variables. Variables are identified by a name (often a mnemonic name) beginning with a letter. They are assigned values using the symbol `=` (often read as 'set equal to'.) The assignment statement takes the form

'variable' `=` 'value'

where ‘value’ can be a number or algebraic expression, and the algebraic expression can include other variables.

Next, into the *‘Command Window’* type

```
>> a = 2
```

MATLAB returns

```
a = 2
```

Now type

```
>> a = 3;
```

Although the output is suppressed by the semi-colon, the value of variable `a` has changed. Confirm this by typing

```
>> a*(a+1)
```

You should obtain

```
ans = 12
```

It is important to remember that `=` assigns values to variables. For example, `a=sqrt(a)` is not an equation nor a recursive definition; it simply assigns to variable ‘`a`’ the square root of the current value of `a`.

Unlike many programming languages, MATLAB does not require variables to be defined before they are used. MATLAB is of course aware of variable types, e.g. integer, real, string, array, logical, but variables are not forced into type-specific roles and they may change their type during the course of a program. MATLAB does not observe any naming conventions for variable types.

It is, however, good practice to initialize large arrays with null values before they are used. This allows MATLAB to set aside sufficient memory before the program begins.

The fluidity of variable type gives MATLAB great flexibility. For instance, as we will see in the next section, if `v` is a vector-valued variable then `v(1)` is its first element. However, `v(1.0)` also returns the first element, as does `v(pi>3)`. The latter case works because `pi>3` returns logical `1` and this is then used as an array index. On the other hand, `v(1.2)` continues to make no sense.

One drawback of not defining variable types is that when type-specificity is required, you will need to ensure the correct variable type is being used. If you make a mistake, however, MATLAB will generally alert you to this at runtime.

3.3 Vectors

MATLAB has been designed to reference and manipulate vectors extremely efficiently. It has an extensive library of operations and functions that can be applied to vectors taken as a whole or on an element-by-element basis. Utilizing these built-in features, will allow you to write streamlined and speedy code.

There is a fuller discussion of vectors and matrices in §6, but it will be helpful to have a small taster now.

Into the *'Command Window'* type

```
>> x = [-1 0 1 2]
```

The variable `x` is row vector with 4 elements. You can ask for its third element by typing

```
>> x(3)
```

Note that row (and column) indices in MATLAB start at 1.

A faster way to define `x` is by typing

```
>> x = -1:1:2
```

The right hand side of the assignment statement is interpreted as 'start at -1 then increment by 1 until 2 is exceeded'. In fact, MATLAB assumes an increment of 1 unless otherwise stated. Therefore, `x = -1:2` is even more succinct.

Now type

```
>> y = exp(x)
```

Note that the `exp` function acts on the vector `x` element-wise: it exponentiates each element of `x`. It follows that `y` will be the same length as `x`.

On the other hand, MATLAB operations tend not to act element-by-element. For instance, `y = x^2` requires `x` to be a scalar or a square matrix, and MATLAB will return an error if this is not the case. To force a MATLAB operation to act element-wise, one inserts a `.` before the operator. To see this, type

```
>> y = x.^2
```

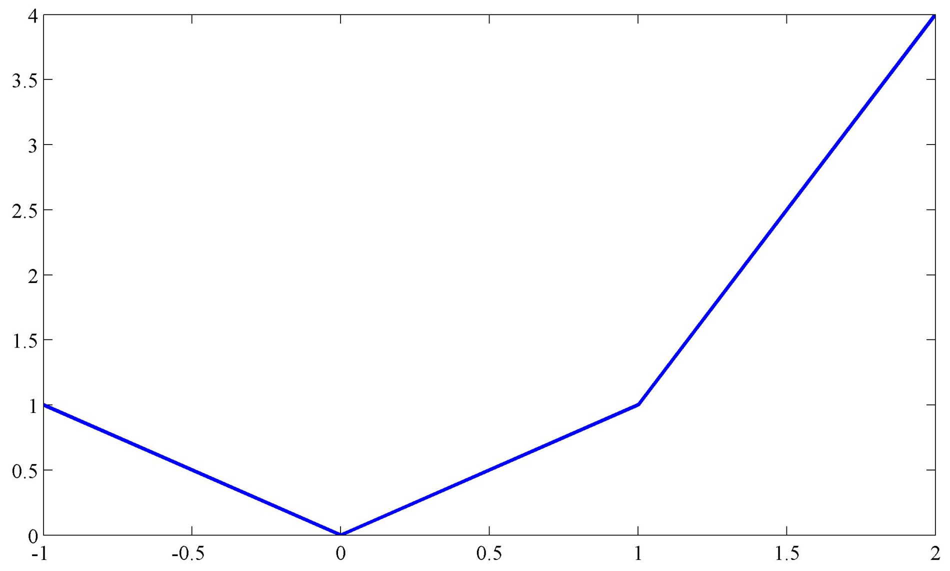

Figure 1: Simple plot of $y = x^2$.

3.4 Plots

The `plot` command is an example of MATLAB's use of vectors. The `plot` command takes vectors `x` and `y` of the *same length* and plots the points `(x(1),y(1)), (x(2),y(2)), ...` By default, `plot` also connects the points with straight line segments.

We will look at plotting in detail in §9. In the meantime, using `x` and `y` defined above, type

```
>> plot(x,y)
```

MATLAB should open a graphics window and display a very segmented-looking parabolic curve (see Figure 1). To make the curve more smooth, type

```
>> x = -1:0.1:2;
>> y = x.^2;
>> plot(x,y)
>> grid on
>> xlabel('x')
>> ylabel('y')
>> title('Plot of y = x^2')
```

Note that it is important to redefine `y` in the second line, otherwise `plot` will fail attempting to plot `x` of length 31 against `y` of length 4. The revised plot is in Figure 2.

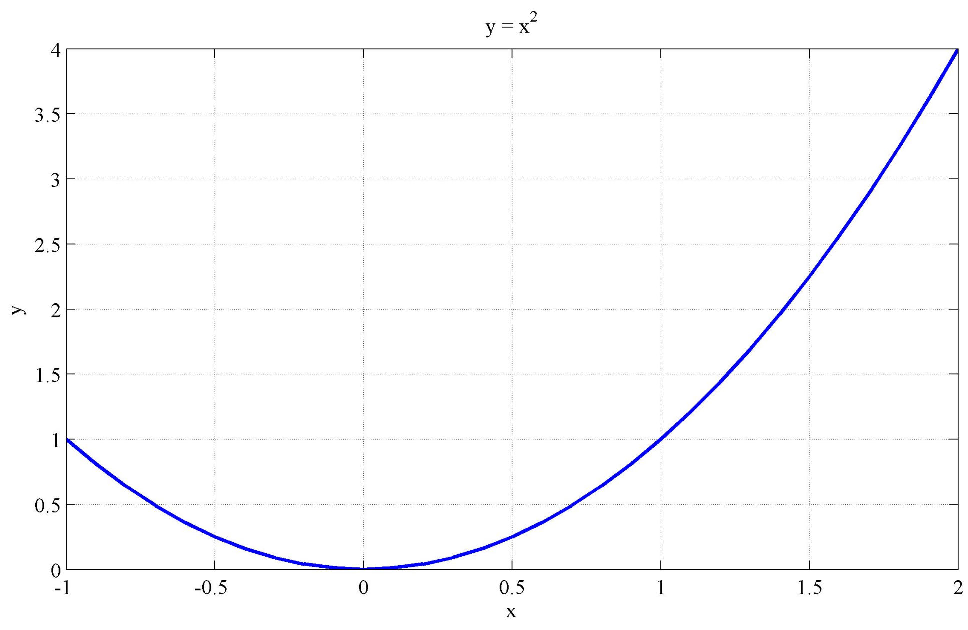

Figure 2: A better plot of $y = x^2$.

4 Programming in MATLAB

MATLAB is a high-level computer programming language and, like other ‘high-level’ programming languages, a MATLAB program is essentially a sequence of statements. However, unlike languages such as C and C++, MATLAB programs are not compiled before they are executed.[10]

4.1 A simple program

As an introduction we will write a simple program to write out a table of the squares of the first 10 natural numbers. To do this, we will introduce the concept of ‘loops’ (a concept that applies to many other ‘high-level’ programming languages).

Into the ‘*Command Window*’ type

```
>> Ilow = 1;
>> Ihigh = 10;
>> for I = Ilow : 1 : Ihigh
     Isquare = I * I
   end
```

In the above:

[10] This is not strictly true. MATLAB does have a ‘just-in-time’ compiler that is invoked if code is executed from a M-File (as described in §4.3.1).

The variables `Ilow` *and* `Ihigh`. The variables `Ilow` and `Ihigh` are assigned values 1 and 10 respectively by the assignment statements:

```
>> Ilow = 1;
>> Ihigh = 10;
```

As noted in the previous section, because a semi-colon has been added at the end of the line the values of `Ilow` and `Ihigh` are not printed out when they are assigned.

The `for` *loop.* The syntax of the `for` loop is as follows. The loop starts with the word `for`. The next statement is the loop counter condition `I=Ilow:1:Ihigh`,[11] where `I` is called the loop counter. The loop counter condition tells the computer to execute the loop once for each value of `I` from `Ilow` to `Ihigh`, adding 1 to `I` after each loop, i.e. it tells the computer to execute the loop once with $\texttt{I} = 1 = \texttt{Ilow}$, once with $\texttt{I} = 2 = \texttt{Ilow} + 1$, ..., once with $\texttt{I} = 10 = \texttt{Ihigh}$.[12] The `for` loop ends with `end`, and the statements between the loop counter condition and `end` are referred to as the body of the `for` loop.

Hence, on reaching the `for` loop, the computer sets $\texttt{I} = \texttt{Ilow}$ then it checks if $\texttt{I} \leqslant \texttt{Ihigh}$ and, if true, it executes the statements between the `for` statement and the `end` statement. Next, `I` is incremented by 1, the computer re-checks if $\texttt{I} \leqslant \texttt{Ihigh}$ and, if true, it re-executes the body of the `for` loop. This continues until $\texttt{I} > \texttt{Ihigh}$, at which point the loop ends and execution continues with the first statement, if any, after the loop.

Note that `Ilow:1:Ihigh` is in the form of a MATLAB vector. In MATLAB, a loop counter is simply assigned the elements of a vector sequentially.[13] Furthermore, the expression `Ilow:Ihigh` would also work in the loop counter condition since, as we saw earlier, an increment of 1 is assumed by default.

The body of the `for` *loop.* The mathematical work of the program is carried out in the statements that form the body of the loop. The assignment statement

```
Isquare = I*I
```

computes the square of `I`, assigns that value to the variable `Isquare`, and prints it out (since there is no final semi-colon). The '`*`' is an 'operator' that means 'multiply'. Operators include:

Operator	**Operation**
`+`	add
`-`	subtract
`*`	multiply
`^`	raise to the power
`/`	divide

For further information on arithmetic operators enter `help arith`, or `helpwin arith` within the MATLAB *'Command Window'*.

[11] Or, with spaces, `I = Ilow : 1 : Ihigh`. Whether you include spaces for clarity is a personal choice.

[12] Since the `for` loop increment is one, the `:1` in `Ilow:1:Ihigh` is optional. Note that integer increments other than one, including negative integers, are allowed; but an increment of zero is not wise.

[13] This means the loop counter does not have to be an integer.

4.1.1 Programming tips

Although MATLAB does not care whether or not you leave blank lines or blank space between variables and operators, it is a considerable help when checking the logic of your program if you get into the habit of indenting the statements in loops (and spacing out at least some of the elements in statements). Decide on your policy for indentation and the use of blanks, and try to stick to it. You will find that when you type in a program within many editors, the editor can automatically indent for you.

It is rather common programming practice to use the variable names `i` and `j` for loop counters. In the above example we have not followed this practice, because in MATLAB `i` and `j` are predefined to be the principal square root of -1. However, if you overwrite this prior definition and you subsequently want to use `i` and/or `j` as the imaginary unit, you can reset them by clearing the current value[s] with `clear i` and/or `clear j`.

4.1.2 My program is running out-of-control or not responding

Sometimes you will make a mistake in your programming, and your program will run out-of-control or not respond. Hitting `Ctrl+C` in the *'Command Window'* should restore normality.

4.2 Improving the output

The output from our program is not very readable! Matters can be improved slightly by asking MATLAB to produce compact formatting. Try

```
>> format compact
```

before executing the `for` loop again:

```
>> for I=Ilow:Ihigh, Isquare = I^2, end
```

where we have reduced the `for` loop to one line (at the expense of readability) by use of commas.[14]

However even after opting for a compact format, the output is still a little like drinking from a fire hydrant (especially if `Ihigh` was much greater than 10). Ideally we would like not to produce two lines of output when one line would do. Instead try the following (note the use of '`;`' after '`Isquare = I*I`'):

```
>> for I=Ilow:Ihigh
     Isquare = I*I;
     disp(Isquare)
   end
```

[14] Note that `format loose` returns you to the default formatting.

or the more compact

```
>> for I=Ilow:Ihigh
     disp(I*I)
   end
```

This is better, since the use of the display command `disp` has reduced the output for each value of `I` to one line. However the output is not as informative as it might be, since we do not know the significance of the numbers printed. So try one last change:

```
>> for I = Ilow:Ihigh
     disp(['I = ' num2str(I) ', I*I = ' num2str(I*I)])
   end
```

To see what is happening here, note that `['I = ' num2str(I) ', I*I= ' num2str(I*I)]` is a MATLAB vector with 4 elements (*separated by spaces*). The `disp` command puts the elements of the vector on the same line of output. The elements of the vector, however, are string variables not numerical variables. Single quotes are used to enclose the value of a string variable. Therefore, regardless of the value of the numerical variable `I`, `'I = '` is the literal text given in quotes. To insert a numerical variable into text as we have to do here, we use the built-in MATLAB function `num2str`, which converts a numerical variable into a string variable.

MATLAB incorporates many approaches to handling input and output (often based on the syntax of the C programming language). For instance, the following uses `fprintf` (which is MATLAB's version of C's `printf`) to improve the readability of the output:

```
>> for I = Ilow:Ihigh
     fprintf('I = %2g, I*I = %3g\n',I,I*I)
   end
```

The `fprintf` *statement.* The

```
fprintf('I = %2g, I*I = %3g\n',I,I*I)
```

component of the above code displays the values of the variables `I` and `I*I` on the same line. The *string of characters* `'I = %2g, I*I = %3g\n'`, i.e. the characters up to the first comma not inside quotes, tells MATLAB what characters to print, where to print [any] numbers, and in what 'format' to print these numbers. For instance the `"%2g"` tells MATLAB to print out a number with a width of 2 characters (specifying the width of a *field* ensures the numbers are printed out in neat columns), and the `"\n"` tells MATLAB to move to a new line. The next two arguments (separated by commas) specify the two numbers that are to be output.

In fact the usual form of calling this is `fprintf(fid,formatstr,arg1,...)`, where `fid` is a file handle, in which case, the output is appended to the file. When (as in our example) the `fid` is omitted, the output goes to `stdout` (usually the command window).

There are many more refinements to the `fprintf` statement; e.g. `"%s"` tells MATLAB to print a string of characters. However, this is probably *not* the stage at which to delve into the many options available with `fprintf`, so we will not. However, having been warned, if you are interested you can learn more about `fprintf` by entering

```
>> help fprintf
```

(or `helpwin fprintf`), or for some slightly more detailed documentation

```
>> doc fprintf
```

4.3 Reducing typing and a noddy guide to functions

As noted earlier, the suppliers of MATLAB, *The MathWorks*, tinker with the graphical interface. As a result in what follows there may be differences between the version of MATLAB on the MCS (version R2017a), and versions of MATLAB that the Faculty of Mathematics has made available for installation on your personal computers (for current students these range from version R2011a/7.12 to version R2017a). The convention adopted below is for the most part to follow version R2012b/8.0. There are some significant differences between version R2012b/8.0 onwards and earlier versions of MATLAB (which we try to note below).

4.3.1 Script files (a.k.a. M-Files)

Unless you have discovered the wonders of the arrow keys[15], backspace and delete within the *'Command Window'*, in the last section you will have typed the same code in a number of times. Really, what you would like to be able to do is to save your program somewhere so that you can re-run it after minor changes. This section explains how to do this.

We are going to store your programs in a file called a script file. However, so that your MATLAB programs do not get muddled up with other files, we will put the MATLAB programs in a separate folder. To create a such a folder move the mouse so that the pointer sits in an unmarked, i.e. white, part of the *'Current Folder'* pane. Right-click, and select `New Folder`. When a new folder appears in the pane, type in a name (e.g. `MATLAB`) and hit the return key. Now double click on your new folder to open that folder (it should be empty). Note that you may get a *Popup* telling you that the folder is not in your MATLAB path, that you can double-click to make the folder your current folder, and that you can add it to your path by selecting *Add to Path* from the context menu: you can access the context menu by right-clicking on the folder, but for the time being just double [left] click on it.

What to do next depends on the version of MATLAB you are using. You can check what version of MATLAB that you are running by entering `ver MATLAB` into the *'Command Window'*.

[15] I.e. up-arrow, left-arrow, right-arrow and down-arrow.

R2012b onwards. Make sure that the `HOME` tab is selected in the top line of the MATLAB window. Then either click on `New Script` immediately below the tab, or click on `New`, followed by `Script`.

R2012a or earlier. Click on `File` on the top line of the 'MATLAB' window, followed by `New` and `Script`

A new *'Editor'* window should open containing a cursor on line 1. Type in the following code:[16]

```
Ilow = 1;
Ihigh = 10;
for I = Ilow:Ihigh
  disp(['I = ' num2str(I) ', I*I = ' num2str(I*I)])
end
```

Or, if you prefer to use `fprintf`,

```
Ilow=1;
Ihigh=10;
for I=Ilow:Ihigh
  fprintf('I = %2g, I*I = %3g\n',I,I*I)
end
```

Once you have typed in the above code it is necessary to save it in a file before it can be run. Again the precise instructions depend on the version of MATLAB that you are running.

R2012b onwards. Make sure that the `EDITOR` tab is selected in the top line of the *'Editor'* window. Next click on `Save`, followed by `Save As...`.

R2012a or earlier. To do this click on `File` on the top line of the *'Editor'* window, followed by `Save As...`.

A new window should appear with a default `File name` of something like `untitled.m` or `Untitled.m` or `UntitledN.m` for some integer `N`; this is rather uninformative. To change the file name click in the box next to `File name` and change the entry to, say, `listsquares.m`. Then click `Save` to accept this name. If you look in the *'Current Folder'* pane a file `listsquares.m` should now have appeared.

Having saved your code, return to the *'Command Window'* and enter

```
>> listsquares
```

The result should be the output

[16] Or cut-and-paste the code from the *'Command Window'*.

```
I = 1, I*I = 1
I = 2, I*I = 4
I = 3, I*I = 9
I = 4, I*I = 16
I = 5, I*I = 25
I = 6, I*I = 36
I = 7, I*I = 49
I = 8, I*I = 64
I = 9, I*I = 81
I = 10, I*I = 100
```

Or, in the case of using `fprintf`,

```
I =  1, I*I =   1
I =  2, I*I =   4
I =  3, I*I =   9
I =  4, I*I =  16
I =  5, I*I =  25
I =  6, I*I =  36
I =  7, I*I =  49
I =  8, I*I =  64
I =  9, I*I =  81
I = 10, I*I = 100
```

If you have made a mistake then you will need to return to the *'Editor'* window and modify your code. (If you have closed the *'Editor'* window either double click on `listsquares.m` in the *'Current Folder'* pane, or enter `edit listsquares` in the *'Command Window'*.) Once you have made your corrections save the code.

If running R2012b onwards. Make sure that the `EDITOR` tab is selected in the top line of the *'Editor'* window, then either click on `Save` followed by `Save`, or click on the `Save` icon (there is no need to use `Save As...` since the file has already been created).

If running R2012a or earlier. Click on `File` on the top line of the *'Editor'* window, followed by `Save` (there is no need to use `Save As...` since the file has already been created).

Suppose now we wish to list the squares from 11 to 20. Rather than typing in the commands again we can edit `listsquares.m` to read:

```
Ilow=11;
Ihigh=20;
for I=Ilow:Ihigh
  disp(['I = ' num2str(I) ', I*I = ' num2str(I*I)])
end
```

Or

```
    Ilow=11;
    Ihigh=20;
    for I=Ilow:Ihigh
      fprintf('I = %2g, I*I = %3g\n',I,I*I)
    end
```

Once you have made the changes save the code as described above. Then in the *'Command Window'* enter

```
    >> listsquares
```

and the result should be the output

```
    I = 11, I*I = 121
    I = 12, I*I = 144
    I = 13, I*I = 169
    I = 14, I*I = 196
    I = 15, I*I = 225
    I = 16, I*I = 256
    I = 17, I*I = 289
    I = 18, I*I = 324
    I = 19, I*I = 361
    I = 20, I*I = 400
```

4.3.2 Functions

If we wish to change `Ilow` and `Ihigh` regularly then there is a better way than repeatedly editing `listsquares.m`, which is to embed the code in a `function`.

How to do this again depends on the version of MATLAB you are running.

If running R2012b onwards. Make sure that the `HOME` tab is selected in the top line of the 'MATLAB' window, then click on `New` followed by `Function`.

If running R2012a or earlier. Click on `File` on the top line of the 'MATLAB' window, followed by `New` and `Function`.[17]

A new *'Editor'* window should open containing something close to:

[17] Or `Function M-File` or `Function M-File`. If the version of MATLAB you are running is 7.6 or lower then read on without opening the editor.

```
function [ output_args ] = Untitled2( input_args )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here


end
```

Functions. The above code is the bare bones of a function. MATLAB functions are like mathematical functions or mappings: they take zero or more input arguments, and produce zero or more output arguments. The name of the above function is that given after the `=` sign on the first line, i.e. it has the name `Untitled2`.[18]

Lines containing %. Anything on a line after a % is interpreted as a comment, and is nearly always ignored by MATLAB.[19] *Inter alia* you can use comment lines to describe what a function and/or program is meant to be doing. Comments can appear anywhere in a line and are used to make the program clearer for reading by humans. Even if you wrote the program yourself, you will still find it easier to understand and debug if you comment it.

`end`. The line at the end of the function consisting of `end` indicates the end of the function. In many cases it is optional.

Using the editor modify the function template to read:[20]

```
function [ Isquares ] = printsquares ( Ilow, Ihigh )
%PRINTSQUARES Function to print the squares of integers
%
% PRINTSQUARES(Ilow,Ihigh) prints the squares from Ilow to Ihigh in
% steps of one, and returns the answers in the (Ihigh-Ilow+1) x 2
% matrix Isquares
%
% Set up a matrix of the correct size to store the results
Isquares = zeros(Ihigh-Ilow+1,2);
%
% Ensure that the matrix indices are all strictly positive
for I = Ilow:Ihigh
  II = I - Ilow + 1;
  Isquares(II,1)=I;
  Isquares(II,2)=I*I;
  disp(['I = ' num2str(Isquares(II,1)) ', I*I = ' num2str(Isquares(II,2))])
end
%
end
```

[18] Depending on what you have done in your MATLAB session it may be called `UntitledN` for some integer `N`.

[19] One of the exceptions to 'nearly always ignored' is the first time that we encounter comment lines. If the second line of a function called `function_name` is a comment line, then that line, and any others immediately following it that are comment lines, are output in response to the command `help function_name`.

[20] The easiest way to do this is to cut-and-paste from `listsquares.m`. Note that if the version of MATLAB you are running is 7.6 or lower, then you will need to open a new blank M-file and type in the code from scratch.

Or replace

```
disp(['I = ' num2str(Isquares(II,1)) ', I*I = ' num2str(Isquares(II,2))])
```

with

```
fprintf('I = %2g, I*I = %3g\n',Isquares(II,1),Isquares(II,2))
```

Compared with `listsquares.m` we have made a number of changes.

The first line. The first line is the function declaration. The function is called `printsquares`, it has two input arguments, namely `Ilow` and `Ihigh`, and it has one output argument, `Isquares`.

The array `Isquares`. The output argument, `Isquares`, is a matrix (or array) into each row of which will be written a number and its square. Thus, in order to hold all results, the matrix needs to be size `(Ihigh-Ilow+1)`×`2`. The line

```
Isquares=zeros(Ihigh-Ilow+1,2);
```

initialises this matrix with zeros by means of setting `Isquares` equal to an array of zeros of size `(Ihigh-Ilow+1)`×`2` (enter `help zeros` and/or `doc zeros` into the *'Command Window'* for information about the `zeros` function).

The comment lines. The comment lines have been modified to provide [bare-bones] help about the function.

Once you have typed in the above code you need to save your code in a file with the same name as the function (i.e. `printsquares`), but with a `.m` appended.

If running R2012b onwards. Make sure that the `EDITOR` tab is selected in the top line of the *'Editor'* window. Next click on `Save`, followed by `Save As...`.

If running R2012a or earlier. To do this click on `File` on the top line of the *'Editor'* window, followed by `Save As...`.

A new window should appear with a default `File name` of `printsquares.m`. Click `Save` to accept this name. If you look in the *'Current Folder'* pane a file `printsquares.m` should now have appeared.

To test your function return to the *'Command Window'* and first enter

```
>> help printsquares
```

The comment lines at the top of your function should be printed out. Next enter

```
>> printsquares(1,10);
```

The result should be the output (or with a few more spaces if you are using `fprintf`)

```
I = 1, I*I = 1
I = 2, I*I = 4
I = 3, I*I = 9
I = 4, I*I = 16
I = 5, I*I = 25
I = 6, I*I = 36
I = 7, I*I = 49
I = 8, I*I = 64
I = 9, I*I = 81
I = 10, I*I = 100
```

If you have made a mistake then, as before, you will need to return to the *'Editor'* window and modify your code. Once you have made your corrections save the code as described earlier on page 17.

The function, as written, can also return the squares in a matrix. To test this enter

```
>> amatrix=printsquares(1,10);
>> amatrix
```

Note that we have used `amatrix` as the name of the matrix in which to store the output from the function `printsquares`; we did not have to use the name `Isquares`. The matrix name `Isquares` is said to be only *locally* defined within the function.

Next, for your choices of `m` and `n` you should check that

```
>> printsquares(m,n);
```

produces the results that you expect.

Finally you should try closing the *'Editor'* window by clicking on the close icon button which is [normally] on the top right of the *'Editor'* window (alternatively, if running R2012a or earlier, click on `File` on the top line of the *'Editor'* window, followed by `Close Editor`). You can always return to editing a file by double clicking on the file name in the *Current Folder* pane.

4.4 Exercises

1. Test your function `printsquares` with `m⩽0` and `m>n`, and then modify your code to work with both these cases.

   Hints.

   `m⩽0`. In this case you need to ensure that the matrix `Isquares` does not have a zero or negative index.

m>n. In this case entering **help sign** and **help abs** might suggest a route forward.

One possible solution to this problem is given in Appendix B on page 63.

2. Write a function, say called printpowers, to display a table listing I, I^2, I^3 and I^4;

Hint: how to edit a file and save it under a new name. If you want to modify an old program to produce a new one you do not need to type it all in again. For instance, suppose that you want to create a printpowers function by modifying your printsquares function. To do this first double click on printsquares.m in the *'Current Folder'* pane. An *'Editor'* window should open up.

If running R2012b onwards. Make sure that the EDITOR tab is selected in the top line of the *'Editor'* window. Next click on Save, followed by Save As....

If running R2012a or earlier. Click on File on the top line of the *'Editor'* window, followed by Save As....

A new window should appear with a default File name of printsquares.m, rather than the printpowers.m desired. Click in the box next to File name and change the entry from printsquares.m to printpowers.m. Then click Save to accept this name. If you look in the *'Current Folder'* pane, a file printpowers.m should now have appeared. Now you are working in the new file. Make your modifications including, say, changing the name of the function from printsquares to printpowers. Once you have done this you will need to save your changes as described earlier on page 17.

5 Help!

One of the advantages of MATLAB is that there is a plethora of ways of getting help. In §1 on page 1 some links to *The MathWorks'* own tutorials have been given, including interactive resources.

For information on debugging code and tracing sources or error, see 'Troubleshooting' §13.

There are several commands in MATLAB to help you get information and find out about your set-up.

`help <function>` *and* `helpwin <function>`. These provide information about `<function>` in the *'Command Window'* and a new window, respectively.

`type <filename>`. This displays the contents of the file `<filename>`. If `<filename>` is a built-in MATLAB function, you will be told.

`which <function>`. This locates functions and files, e.g. `which roots` tells you whether `roots` is a built-in command, a function or doesn't exist.

`lookfor <keyword>`. This performs a keyword search, e.g. `lookfor empty` finds commands that deal with empty arrays and briefly describes them.

`helpbrowser` *and* `doc`. These open a new window to display MATLAB documentation.

`doc <function>`. This tells you about `<function>` in a new window.

`demo <function>`. This command opens the *Demos pane* in the *Help browser*, listing demos for all installed products, e.g. `demo matlab`.

`who`, `whos` *and* `workspace`. `who` and `whos` list the current variables (the latter in long form) in the *Command Window*. `workspace` does the same thing in its own pane/window and provides a GUI (graphical user interface) to manipulate the variables.

`why`. This command was written either by a Monty Python fan, or by someone who had just calculated an answer of `42`.

`path`. Prints the current MATLAB path; this is a list of directories. When you type a `<command>`, MATLAB looks in these directories for the corresponding file. The path may be modified using `addpath`, `rmpath`, `savepath`, or from the `File` menu.

`ver`. Tells you which versions of which toolboxes (libraries) are installed.

6 Vectors and matrices

MATLAB has been designed to work efficiently with matrices, including vectors (i.e. matrices with only one row or one column) and scalars (i.e. a 1×1 matrix).

6.1 Creating matrices

There are a number of ways of entering matrices (or arrays).

1. Matrices can be entered explicitly element by element. For instance try the following commands in the *'Command Window'*:

   ```
   >> clear
   >> rowvec1=[1 2 3 4]
   >> colvec1=[4;3;2;1]
   >> rowvec2=colvec1'
   >> colvec2=rowvec1'
   >> rowvec3=-4:1:0
   >> colvec3=[0:-1:-4]'
   >> A=[1 2 3; 4,5,6; 7 8 9]
   >> A'
   >> B=[1 sqrt(-1); 1+i 4+3i]
   >> C=['upper line';'lower line']
   >> C(1,2), C(2,9)
   ```

   Remarks.

   (i) The command `clear` clears all variables and functions from memory so that you start with a clean slate[21].

   (ii) The command `.'` forms the transpose of a matrix. The command `'` forms the *conjugate* transpose. These of course differ only if the matrix is complex.

   (iii) The construct `a:b:c` works as in `for` loops, i.e. it generates a row vector starting at `a` in increments[22] of `b` until `c` is exceeded (`c` is only included if `c-a` is a multiple of `b`).

   (iv) The elements within a row of a matrix may be separated by commas as well as blanks[23].

   (v) As illustrated by the row vector `B`, MATLAB understands complex numbers. It was for this reason that we used `I` and not `i` in §4.1. If we had [accidentally] redefined `i` it can be reset to the square root of `-1` by the command `clear i`; alternatively `j` may be used as the square root of `-1` (as long as it has not been redefined).

[21] The command `clear var1 var2 var3` will clear just the specified variables, leaving all others alone.
[22] Or decrements if `b` is negative.
[23] Commas are preferred to avoid problems such as `a = [1 3 4 - 7]`. Should this be `[1,3,(4-7)]` or `[1,3,4,-7]`?

(vi) Matrices can consist of strings of characters as well as numbers. As with other matrices, string matrices must have the same number of elements in each row of the matrix.

(vii) You can refer to the element in row `m` and column `n` of a matrix, say `C`, by `C(m,n)`. A matrix, or vector, will only accept positive integers as indices, starting from `1`.

It is possible to create multi-dimensional arrays, as in this example where a $3 \times 4 \times 2$ matrix is generated element by element:

```
>> for i1=1:1:3
     for i2=1:1:4
       for i3=1:1:2
         multi(i1,i2,i3)=i1+i2*i3;
       end
     end
   end
>> multi
```

2. Matrices can be also be formed using one of a number of built-in functions. For instance try the following commands in the '*Command Window*':

```
>> a1=zeros(2)
>> a2=zeros(2,3)
>> b1=ones(2)
>> b2=ones(3,2)
>> c1=eye(2)
>> c2=eye(2,3)
>> c3=eye(3,2)
>> d1=diag(colvec1)
>> d2=diag(rowvec1,1)
>> d3=diag(rowvec2,-2)
>> e1=rand(3)
>> e2=rand(2,3)
>> f=hilb(4)
>> g=magic(5)
```

Remark. Use the `help` command to find out more information about these functions, e.g. `help diag`.

3. Matrices can also be read in from a file. Open up an empty script file as described at the start of §4.3.1 (although we are not going to write a script into it), and enter the following

```
-3  5  5  6
24  3 -5  0
```

Next save this as a file `tempmat.dat` (note the **.dat** replacing the `.m`).[24] Then in the '*Command Window*' execute

[24] The extension `.dat` could be anything, but it's best to avoid `.m` and `.mat`.

```
>> load tempmat.dat
>> tempmat
```

Remark. You have created a 2×4 matrix called `tempmat`.

As well as loading matrices from a file, it is also possible to save matrices to a file. For instance try

```
>> tranmat=tempmat';
>> save tempmat.dat tranmat -ascii -double
```

If you double click on `tempmat.dat` in the *Current Folder* pane you will see that the original 2×4 matrix has been overwritten with its 4×2 transpose.

There are certain restrictions on the matrices than can be written in human-readable, or ASCII, form using the `save` command. Complex matrices and large multi-dimensional matrices (such as `multi`) have to be saved in binary 'MAT-file' form. To illustrate this try the following:

```
>> whos
>> save tempvar tempmat tranmat multi
>> clear
>> whos
>> load tempvar
>> whos
```

Remark. If you wish to tidy up your files you can delete `tempmat.dat` and `tempvar.mat` by right clicking on them, selecting `Delete` and then clicking on `Yes` in the pop-up window.

6.2 Manipulating matrices

We have already seen that we can refer to an element of a matrix by its indices. Rows, columns and sub-matrices can also be selected from a matrix and manipulated. For example:

```
>> D=[ 1 3 5; 2 4 6];
>> E=D(2,:)            % row 2
>> G=D(:,1)            % column 1
>> H=D(:,2:3)          % columns 2 and 3
>> J=D(:,[1,3])        % columns 1 and 3
>> K=[G H]             % the orginal matrix D
>> disp(D-K)           % check
```

We can do arithmetic with matrices, where the normal matrix rules apply. Hence the addition and subtraction operators, `+` and `-`, only make sense when acting on matrices that have the same numbers of rows and columns. For example try

```
>> A=hilb(5)+ones(5)
>> B=hilb(5)-ones(4)
```

The operator `*` performs normal matrix multiplication, so the number of columns of the matrix to the left of `*` must equal the number of rows of the matrix to the right. We have noted previously that `.` before an operator tells MATLAB to carry out the operation elementwise. Therefore, `.*` performs element-by-element multiplication for matrices that have the *same number of rows and columns*. To illustrate this try

```
>> A=hilb(5)
>> B=eye(5)
>> disp(A*B)
>> disp(A.*B)
```

Similarly `^` raises a matrix to a power, while `.^` raises each element to a given power. For illustration try

```
>> A=[1 2 3; 0 1 2; 0 0 1]
>> disp(A^3)
>> disp(A.^3)
>> disp(3.^A)
```

'Division' needs a little more care. `A\B` is the matrix division of `A` into `B`, which is roughly the same as $\texttt{A}^{-1}\texttt{B}$, except it is computed in a different way using Gaussian elimination[25]. `A/B` is the matrix division of `B` into `A`, which is roughly the same as $\texttt{A*B}^{-1}$, except it is computed using `A/B = (B'\A')'`. If the inverse of a matrix is needed then it can be calculated using the `inv` function. To illustrate this try[26]

```
>> A=rand(5)
>> B=rand(5)         % B will not be the same as A: why?
>> inv(A)
>> inv(B)
>> C=A\B-inv(A)*B
>> norm(C)
>> D=A/B-A*inv(B)
>> norm(D)
>> clear I; I=eye(5)
>> norm(A\I-inv(A))
```

Similarly there are element-by-element operators `A.\B` and `A./B`. As an illustration try

[25] Actually there are several different methods employed depending upon the circumstances, and not all of them use Gaussian elimination directly.

[26] Note that whereas `I` was used as a scalar in §4.1, in this example it is defined to be a `5`×`5` matrix. Hence the advisable use of `clear I`.

```
>> A=hilb(3)
>> B=ones(3)
>> I=eye(3)
>> A.\B
>> A./B
>> A.\I
>> A./I
```

6.2.1 Exercise

(i) Generate a 4x4 matrix with random entries using `rand` (i.e. one with values drawn uniformly from (0,1) - see §10), and add it to its transpose to form a symmetric matrix A.

(ii) Test whether A is non-singular, and if it is then use MATLAB's inbuilt function `eig` to find its eigenvalues and eigenvectors. Test that the eigenvectors are mutually orthogonal. Are they all real-valued? *(You can test orthogonality using a single operation on the matrix of eigenvectors, or by taking scalar products between individual ones using* `dot`*, or by multiplying them directly. Try all three methods.)*

(iii) Generate a random unit vector $\mathbf{v}$ of length 4. Apply A to $\mathbf{v}$, normalise the result (using `norm`) to obtain a new vector, and repeat this process, applying A iteratively until convergence is reached (you'll need to decide on a criterion for this) or the number of iterations exceeds some limit which you choose. How does your final result $\mathbf{v}_*$ compare with the eigenvectors from the previous part?

(iv) If the eigenvectors are distinct you should find that $\mathbf{v}_*$ corresponds to the one with largest eigenvalue say $\mathbf{e}_1$. Can you modify your initial vector $\mathbf{v}$ so that the process will converge to a different eigenvector? *Hint: Subtract the component of* $\mathbf{v}$ *in the direction* $\mathbf{e}_1$*.)*

Remarks.

(a) Care must be taken to multiply vectors in the intended order: `a*a'` is not the same as `a'*a`.

(b) If `a` and `b` are complex-valued then `dot(a,b)` is their complex scalar product, so that, whether `a` is a real- or a complex-valued column vector, `dot(a,a) = a'*a = ||a||`^{2}.

(c) You may find the IA course *Vectors and Matrices* helpful here.

7 A few more functions

We have already encountered a number of functions, e.g. `disp`, `fprintf`, `zeros`, `ones`, `eye`, `diag`, `rand`, `hilb`, `magic`, `inv` and `norm`. MATLAB has hundreds of other predefined functions to perform many mathematical, graphical and other operations (a partial list in given in Appendix C). Further, many more functions are provided in the optional 'toolboxes' (enter `ver` to see which toolboxes are available to you), and even more functions are freely available on the web. Below we list a few more mathematical functions that [primarily] act on scalars, vectors and matrices.

7.1 Scalar functions

The following functions essentially operate on scalars, but operate element-by-element on matrices:

Function	Description	Function	Description
`cos`	cos (angle in radians)	`acos`	inverse cos
`sin`	sin (angle in radians)	`asin`	inverse sin
`tan`	tan (angle in radians)	`atan`	inverse tan
`exp`	exponential	`atan2`	2-argument form of inverse tan
`cosh`	hyperbolic cos	`acosh`	inverse hyperbolic cos
`sinh`	hyperbolic sin	`asinh`	inverse hyperbolic sin
`tanh`	hyperbolic tan	`atanh`	inverse hyperbolic tan
`log`	natural log	`rem`	remainder after integer division
`log10`	base 10 log	`fix`	round towards 0
`abs`	absolute value	`floor`	round down to the nearest integer
`sign`	sign (either -1 or +1)	`ceil`	round up to the nearest integer
`sqrt`	square root	`round`	round to the nearest integer

7.2 Vector functions

There are other MATLAB functions that operate on row or column vectors; most also act on matrices column-by-column to produce a row vector containing the results of each column. A few of these functions are

Function	Description
`all`	true if all elements of a vector are nonzero
`any`	true if any element of a vector is a nonzero number or is logical 1
`max`	largest element
`mean`	mean value
`median`	median value
`min`	smallest element
`prod`	product of elements
`std`	standard deviation
`sum`	sum of elements

It follows that the largest entry in a matrix `A` is given by `max(max(A))` rather than `max(A)`. Alternatively, `max(A(:))` can be used since `A(:)` reshapes matrix `A` as a single-column vector.

7.2.1 Exercise

Using the function `max` find the largest entry of the matrix `multi` defined on page 25.

7.3 Matrix functions

There are yet other MATLAB functions that perform common operations on matrices; the table below lists a small subset.

Function	Description
`cond`	Condition number with respect to inversion
`det`	Matrix determinant
`diag`	Create or extract diagonals
`eig`	Eigenvalues and eigenvectors
`expm`	Matrix exponential
`full`	Convert a sparse matrix to a full matrix
`inv`	Matrix inverse
`lu`	LU matrix factorization
`norm`	Vector and matrix norms
`null`	Null space
`poly`	Characteristic polynomial
`rank`	Rank of matrix
`rref`	Reduced row echelon form
`size`	Size of matrix
`sparse`	Convert a matrix to sparse form
`svd`	Singular value decomposition
`trace`	Sum of diagonal elements
`tril`	Lower triangular part of matrix
`triu`	Upper triangular part of matrix

7.4 Random number generation

MATLAB has several algorithms for random number generation. Usually the defaults will suffice, but look at `rng` for more details; see also §10.

Function	Description
`rand`	Uniformly distributed random numbers
`randi`	Uniformly distributed random integers
`randn`	Normally distributed random numbers
`randperm`	Random permutation
`rng`	Control the random number generator used by `rand`, `randi` & `randn`
`RandStream`	Create a random number stream

8 Program flow control

The `for` construct is one of several control structures available for programming in MATLAB. In this section we will describe some of the other control structures.

8.1 The `if-else` control structure

The if–else control structure allows you to execute different sets of instructions depending on whether a test condition, or expression, is *true* or *false*. The general form of the statement is

```
if <expression>
  <statements>
elseif <expression>
  <statements>
else
  <statements>
end
```

The `else` and `elseif` parts are optional, and multiple `elseif` clauses are allowed.

As an example, consider the following function that uses *real arithmetic*[27] to solve a quadratic equation, and gives real or complex solutions as appropriate.

[27] As noted earlier MATLAB supports complex arithmetic, but as an example we choose to restrict ourselves to real arithmetic.

```
function solvequad( a, b, c )
%SOLVEQUAD Solves a quadratic equation using real arithmetic
%
%   SOLVEQUAD(a,b,c) solves the quadratic equation a*x*x+b*x+c=0
%
if abs(a*c) <= eps*b*b
  disp('a*c is too small compared with b*b')
  return
end
%
disc = b*b-4*a*c;
if disc >= 0
  disc = sqrt(disc)/(2*a);
  r = -b/(2*a) + disc;
  fprintf('Solution 1 is %14.4g\n',r)
  r = -b/(2*a) - disc;
  fprintf('Solution 2 is %14.4g\n',r)
else
  re = -b/(2*a);
  im = sqrt(-disc)/(2*a);
  fprintf('Solution 1 is %14.4g + %14.4g i\n',re,im)
  fprintf('Solution 2 is %14.4g - %14.4g i\n',re,im)
end
end
```

The first `if` *control structure.* `abs(a*c)` calculates the absolute value of `a*c` (i.e. |`a*c`|), while `eps` is a MATLAB positive constant that is set to the spacing between two successive floating point numbers as approximated in the computer. If `abs(a*c)` is too small compared with `b*b` then rounding errors can be important.

The first `if` control structure tests whether `abs(a*c)` is less than or equal to `eps*b*b`. If this test is true then an informative message is printed out using `disp`, and control is '`return`ed' to the program which called the function (i.e. execution of the function is terminated).

The second `if` *control structure.* Immediately before this control structure the variable `disc` is set equal to the discriminant `b*b-4*a*c`. The if–else control structure then tests the value of `disc` to decide whether to calculate real or complex solutions.

`disc` *greater than or equal to zero.* If `disc` is greater than or equal to zero, then the lines

```
disc = sqrt(disc)/(2*a);
r = -b/(2*a) + disc;
fprintf('Solution 1 is %14.4g\n',r)
r = -b/(2*a) - disc;
fprintf('Solution 2 is %14.4g\n',r)
```

are executed. The expression `sqrt(disc)` computes the square root of `disc` (see §7 for more details on functions). Then following some more arithmetic, the real roots are written to the display. The format string '%14.4g' in `fprintf` tells the computer to output the real variable in a field fourteen characters (or positions) wide with four decimal places of accuracy.

`disc` *less than zero.* If `disc` is negative then the lines

```
re = -b/(2*a);
im = sqrt(-disc)/(2*a);
fprintf('Solution 1 is %14.4g + %14.4g i\n',re,im)
fprintf('Solution 2 is %14.4g - %14.4g i\n',re,im)
```

are executed instead, and the complex roots are written to the display.

Logical expressions. The logical expressions used in conditional statements are often constructed using the following relational operators:

Operator	**Operation**
`==`	equal to ($=$)
`~=`	not equal to ($\neq$)
`>`	greater than ($>$)
`<`	less than ($<$)
`>=`	greater than or equal to ($\geqslant$)
`<=`	less than or equal to ($\leqslant$)

Conditional statements can also be constructed with the following logical operators (see also `help relop` and page 37):

Operator	**Operation**
`&&`	Short-circuit logical AND
`||`	Short-circuit logical OR
`&`	Element-wise logical AND
`|`	Element-wise logical OR
`~`	Logical NOT
`xor`	Logical EXCLUSIVE OR
`bitand`	Bitwise AND. See also `bitor`, `bitxor` and `bitcmp`
`any`	True if any element of vector is nonzero
`all`	True if all elements of vector are nonzero

Further help. Further details of the if–else statement can be found by entering

```
>> help if
```

or

```
>> doc if
```

8.1.1 Exercises

(i) Type the above function into a file `solvequad.m`. Run the function for various choices of `a`, `b` and `c`, e.g. `solvequad(1,1,1)`.

(ii) Modify the above code to return the two solutions as output arguments. Compare your answer with the output of the MATLAB command `roots`, which you can find out more about using `help roots`.

Hints.

(a) The MATLAB command `roots` accepts a vector argument of the form `[a b c]`; e.g. try

```
>> a=1; b=1; c=1;
>> solvequad(a,b,c)
>> roots([a b c])
```

(b) To learn more about vectors, matrices and arrays use MATLAB's extended documentation. To do this click on '`Help`' on the top line of the 'MATLAB' window, then click on '`Product Help`'. In the new window that opens there is a search box towards the top left-hand corner. Enter '`creating and concatenating matrices`' into this search box, and then press '`return`' to start the search. Documentation will appear in the main pane.

(iii) Check if the two roots are the same and, if so, only output one of them. Note that because the computer only stores real numbers approximately (with a relative error of `eps`), you will find that checking if a real number is equal to zero will not work — you will need to use a test that checks if a real number is 'close' to zero.

8.2 The `while` control structure

A `while` loop repeats statements an indefinite number of times while an expression is true. The general form of a `while` statement is:

```
while <expression>
  <statements>
end
```

- The `break` statement can be used to terminate the loop prematurely, i.e. to force a break to the statement immediately following the `end`.
- The `continue` statement can be used to pass control to the next iteration of a `for` or `while` loop in which it appears, skipping any remaining statements in the body of the `for` or `while` loop.

A simple example which writes out the factors of a natural number is the program:

```
function printfactors( intnum )
I = int32(intnum);
J = 1;
while J <= I
  k=I/J;
  if k*J == I
    fprintf('%4d\n',J)
  end
  J=J+1;
end
fprintf(' are factors of %d\n',I)
end
```

The `int32` *function.* The statement

```
I = int32(intnum);
```

rounds the [floating point] number `intnum` to the nearest *signed 32-bit integer*. The value of a signed 32-bit integer can be in the range -2^{31} to $(2^{31} - 1)$.[28]

The arithmetic operations $+$, $-$, $*$, ^ and / are defined for 32-bit integers, where the divide operator / rounds to the nearest integer. Arithmetic operations involving both floating-point numbers and 32-bit integers result in 32-bit integers. Hence the statement

```
k=I/J;
```

results in a 32-bit integer (since `I` is a 32-bit integer).

The `while` *loop.* The loop starts with the word `while`. Then comes the loop test condition `J <= I`, which is *true* if `J` is less than or equal to `I` and *false* otherwise. Then comes a set of [indented] statements, namely

```
k=I/J;
if k*J == I
  fprintf('%4d\n',J)
end
J=J+1;
```

If the loop test condition is *true* these statements are executed. The the loop test condition is then re-evaluated and if *true* the statements are executed again. This sequence of 'test and re-execution' is repeated until the loop test condition becomes *false*; then the loop ends and execution continues with the first statement after the loop.

A single equals sign (=) means 'set equal to'. The statement `J=J+1` means 'set the value of `J` to its old value plus `1`'; it does not mean that `J` is equal to `J+1`.

The factor test. The statement

[28] Within MATLAB you can find the range using `intmin('int32')` and `intmax('int32')`. Values outside the range -2^{31} to $(2^{31} - 1)$ are said to 'saturate on overflow', namely they are mapped to -2^{31} or $(2^{31} - 1)$.

```
if k*J == I
```

tests whether `k` multiplied by `J` is equal to `I`. This is *true* if `J` is a factor of `I`, in which case `J` is printed.

8.2.1 Exercise

Change the `printsquares` function in §4.3 to use a `while` loop in place of the `for` loop. There are advantages and disadvantages in using each of these control structures (e.g. `for` loops are particularly useful when manipulating array and string variables). Try to think of an example that would be simpler to write using a `while` loop than a `for` loop.

8.3 The `switch-case` control

The if–else pair is often the best way of controlling the possible branching in the execution of a program. Sometimes though `switch-case` control structure, which switches among several cases based on an expression, contributes to neater-looking code. The general form of the `switch` statement is:

```
switch <switch_expression>
  case <case_expresion>,
    <statements>
  case {<case_expression1>, <case_expression2>,...}
    <statements>
  ...
  otherwise,
    <statements>
end
```

The statements that follow the first `case` in which the `switch_expression` matches with the `case_expression` are executed. When the `case_expression` is a succession of alternative expressions (within braces and separated by commas), as in the second case above, the statements that follow the `case` are executed if the `switch_expression` matches any of the alternatives. If none of the `case_expressions` match with the `switch_expression` then the `otherwise` clause is executed (if present). Only one `case` is ever executed and execution resumes with the statement after the `end` statement.[29]

A simple example, which prints out a different message according to an input choice, is the program:

[29] Unlike C, the `switch` statement does not 'fall through' from one `case` to the next, and hence `breaks` are unnecessary.

```
function switchcase
%SWITCHCASE A simple example function using the switch statement
%
option=0;
while option ~= 4
  disp('1: Euler''s Method');
  disp('2: Leap Frog Method');
  disp('3: Runge Kutta Method');
  disp('4: Quit');
  option=[];
  while ~isa(option,'numeric') || isempty(option)
    option=input('Please enter your choice:  ');
  end
  switch(option)
    case 1,
      fprintf('Attempting Euler\n\n')
      % Code to solve ODE using Euler
    case 2,
      fprintf('Attempting Leap frog\n\n')
      % Code to solve ODE using LF
    case 3,
      fprintf('Attempting Runge Kutta\n\n')
      % Code to solve ODE using RK
    case 4,
      fprintf('Exiting function\n\n')
      return
    otherwise,
      fprintf('Please choose an integer between 1 and 4\n\n')
  end
end
end
```

In addition to demonstrating the use of `switch`, this function also illustrates a number of other points.

The outer `while` *loop.* The outer `while` loop continues execution until the variable, `option`, is found to be equal to '4'.

The empty matrix and `isempty`. `[]` is the empty matrix. A variable can be tested to see if it is equal to the empty matrix using the function `isempty`, which returns *true* if the argument is the empty matrix. For more information use `help isempty`.

The function `isa`. The function `isa(object,'classname')` returns true if `object` is an instance of `'classname'`. In the example program `isa` is used to test whether or not the variable `option` is an integer or a floating point number. The *not* operator `~` in front of the call to `isa` negates the test. For more information use `help isa`.

The logical OR operator `||`. The operator `||` is used between two expressions as in `A || B`. The expression `A || B` is true if `A` and/or `B` is true; whereas the expression `A && B` is true only if both `A` and `B` are true. Note that we use the short-circuit operator, `||`, rather than the simple `|`. The reason is that it only requires the second evaluation if the first one has not already decided the outcome. For more information use `help relop`.

The function `input`*.* The function `input('string')` outputs a character string and then waits for input from the keyboard. The input can be any MATLAB expression, which is evaluated, using the variables in the current workspace. If the user presses the return key without entering anything, `input` returns an empty matrix. For more information use `help input`.

`''` *and* `\n`*.* To output a `'` in a display string we have to enter `''`. As before, to output a newline in a display string we use `\n`. For more information use `help fprintf`.

8.3.1 Exercise

Type in and run the code. Try and break it, e.g. in response to the prompt try `1.2`, `text`, `'a string'`, `int32(3)`, etc.

9 Elementary graph plotting

MATLAB can be used to produce a wide variety of plots and curves, including 2D plots, 3D plots and 3D surface plots. The basics are covered below, and there is also an elementary MATLAB graphics tutorial at

http://www.mathworks.co.uk/help/matlab/learn_matlab/plots.html.

9.1 The `plot` command

The `plot` command creates a linear plot; if x and y are vectors of the same length, the command `plot(x,y)` opens a graphics window and draws a plot of the elements of x on the abscissa versus the elements of y on the ordinate. For instance

```
>> x=-pi:pi/10:pi
>> y=sin(x)
>> plot(x,y)
```

plots the sine function between $-\pi$ and π. Here `pi` is π, `x` is a row vector that consists of real values between $-\pi$ and π in steps of $\pi/10$, while `sin(x)` calculates the sine of [all] the elements of `x`. A smoother graph can be obtained by reducing the step, or mesh, size between points on the abscissa (sse Figure 3):

```
>> x=-pi:pi/1000:pi; y=sin(x); plot(x,y)
```

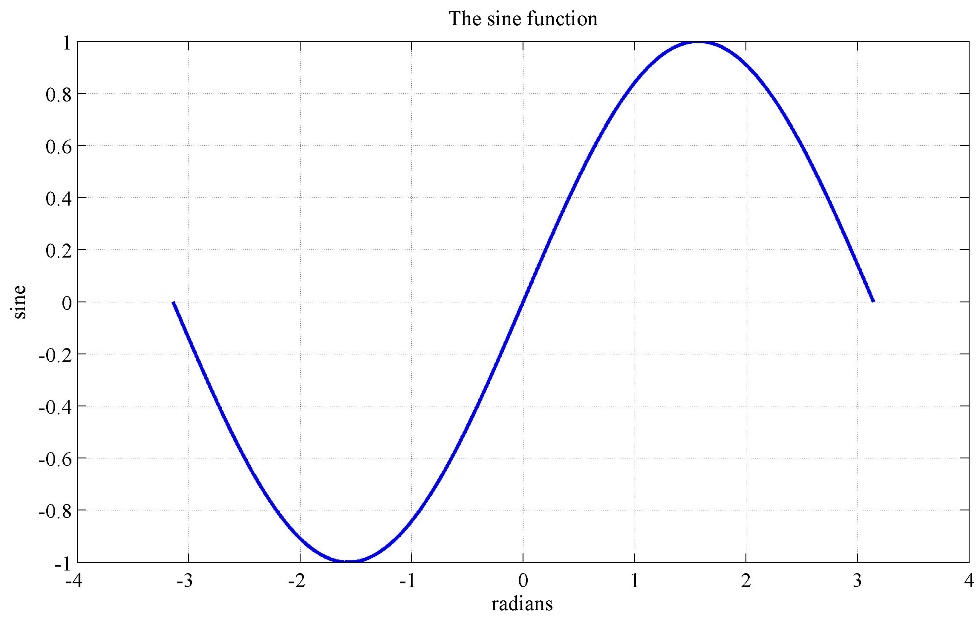

Figure 3: A smooth plot of $y = sin(x)$. The step size is $\pi/1000$

However, it is not always possible to plot a smooth graph if you choose the wrong function (see Figure 4):

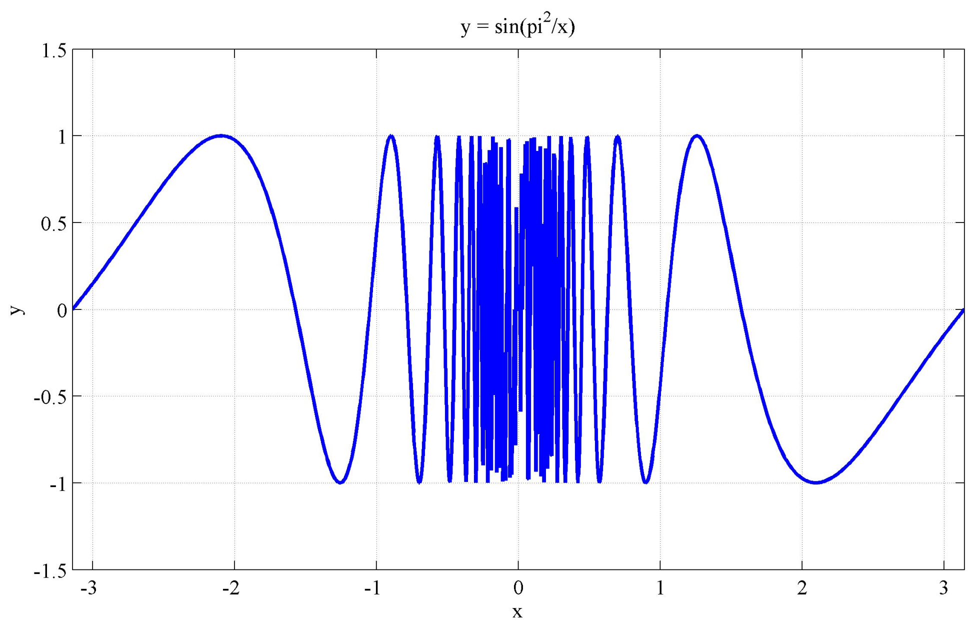

Figure 4: The function $y = sin(\pi^2/x)$. The step size is $2\pi/1000$

```
>> x=linspace(-pi, pi, 1000); y=sin(pi*pi./x); plot(x,y)
>> axis([-pi pi -1.5 1.5])
```

Remarks.

- The `linspace` command creates a vector of 1000 elements equally spaced between $-\pi$ and π inclusive.[30]
- Note the use of the element-by-element *right array divide*, i.e. '`./`' rather than *right matrix divide*, i.e. '`/`'.[31]
- Once you have a graph it's easy to add titles and other decoration. You can either use the menus on the graphics window (e.g. click on `Insert`, or enter the following commands into the *'Command Window'*:

```
>> title('The sine function')
>> xlabel('radians')
>> ylabel('sine')
>> grid on
```

- You can plot more than one function on a graph. However, the following does not work since the first graph is erased.

```
>> w=cos(x); plot(x,w)
```

[30] Why choose an even number of elements between $-\pi$ and π for this particular function?

[31] `A./B` denotes element-by-element division. `A` and `B` must have the same dimensions unless one is a scalar. A scalar can be divided with anything.

It is first necessary to enter '`hold on`' to hold the current plot and all axis properties so that subsequent graphing commands add to the existing graph. Try

```
>> x=linspace(-pi, pi, 2001); y=sin(x); w=cos(x);
>> plot(x,y)
>> hold on
>> plot(x,w)
>> xlabel('radians'), title('The sine and cosine functions')
```

- The trouble now is that it is difficult to tell the graphs apart since they are in the same colour and line style. One way forward is to plot the two lines at once, since then the lines are automatically drawn in different colours. It is then possible to distinguish between the graphs using the `legend` command (see Figure 5):

```
>> hold off
>> plot(x,y,x,w)
>> xlabel('radians'), title('The sine and cosine functions')
>> legend('sine','cosine');
```

Note. It was necessary to issue a `hold off` instruction so that the figure could be redrawn.

- The `plot` command has many options to control colour, markers and line style: see `help plot` and/or `doc plot`. Hence to plot the sine graph in a green dashed line and the cosine graph in a red dash-dot line one can use

```
>> plot(x,y,'g--',x,w,'r-.')
>> xlabel('radians'), title('The sine and cosine functions')
>> legend('sine','cosine');
```

- It is possible to add or remove a grid using `grid on` or `grid off` respectively.
- By means of the menu options on the top line of the *Figure window* it is possible to change many features of graphs. Alternatively you can use the `axis` command to change many axis features: range, aspect ratio, etc.
- Within a function it is not possible to use the menu options. Instead, we use MATLAB's *handle graphics* features to manipulate the properties of the figures, axes, lines etc.

```
>> h = plot(x,y,x,w);
>> set(h(1),'LineStyle','--','Color','g')
>> set(h(2),'LineStyle','-.','Color','r')
```

9.1.1 Exercise

Plot the sine and cosine functions with a large number of points (e.g. 2001). Then try

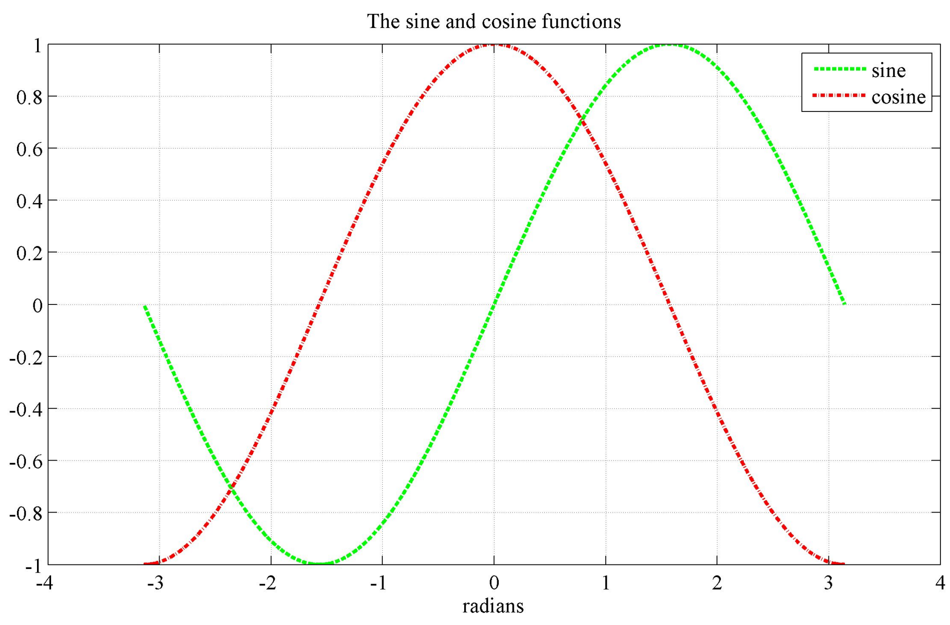

Figure 5: Plots of $y = sin(x)$ and of $w = cos(x)$, demonstrating linestyles, colours and legends.

```
>> hold on;
>> v=tan(x);
>> plot(x,v)
>> title('The sine, cosine and tangent functions')
>> xlabel('radians'); legend('sine','cosine', 'tan');
```

What has happened is that the ordinate has been automatically rescaled. Using `help axis`, or otherwise, choose axes to obtain a sensible plot (e.g. see Figure 6).

9.2 Other 2D graphs

There are many other 2D graphical tools available in MATLAB, e.g.

Function	**Description**	**Function**	**Description**
`loglog`	Log-log scale plot	`semilogx`	Semi-log scale plot
`semilogy`	Semi-log scale plot	`polar`	Polar coordinate plot
`fplot`	Plot function	`barh`	Horizontal bar graph
`bar`	Bar graph	`comet`	Comet-like trajectory
`compass`	Compass plot	`ezplot`	Easy to use function plotter
`errorbar`	Error bar plot	`ezpolar`	Easy to use polar coordinate plotter
`feather`	Feather plot	`fill`	Filled 2-D polygons
`area`	Filled area plot	`stem`	Discrete sequence or 'stem' plot
`hist`	Histogram	`rose`	Angle histogram plot
`pareto`	Pareto chart	`pie`	Pie chart
`scatter`	Scatter plot	`plotmatrix`	Scatter plot matrix
`stairs`	Stairstep plot		

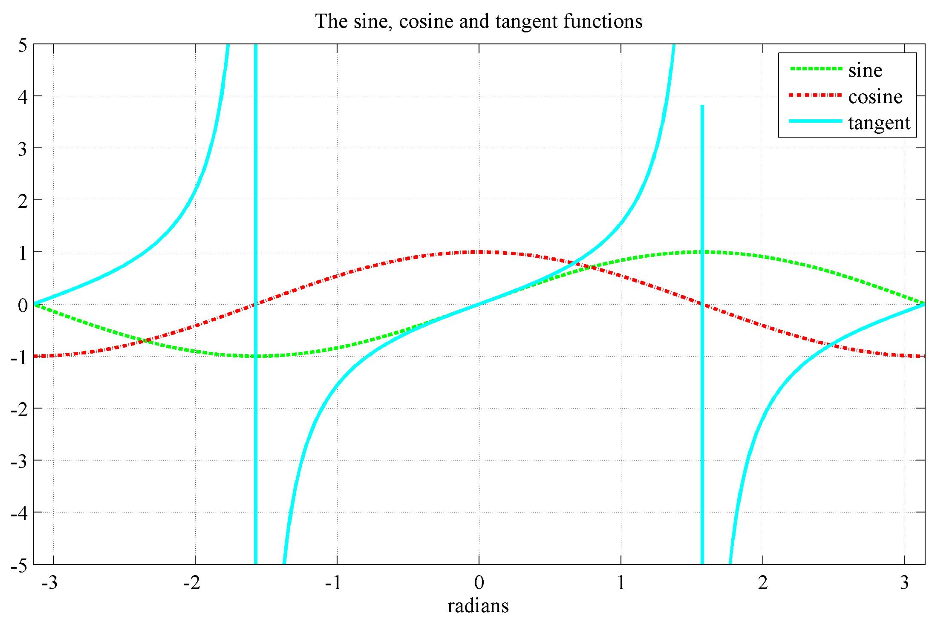

Figure 6: Plots of $y = sin(x)$, $w = cos(x)$ and $v = tan(x)$.

To find out more about the above functions use `help graph2d` or `help specgraph` (or alternatively `doc graph2d` or `doc specgraph`).

As an example, try the command `fplot`, which plots the graph of a function. For instance the sine function can alternatively be plotted using

```
>> clf
>> fplot(@sin,[-pi pi])
```

Remarks.

(i) `clf` has been used to clear the current figure.

(ii) `@sin` is a function handle to the function `sin`. See `help function_handle` to learn more.

(iii) The command `fplot(@(x)sin(x),[-pi pi])` is an equivalent alternative.

(iv) `fplot` has a number of options (see `help fplot`). Like `plot` it is possible to plot more than one graph at a time, e.g. try

```
>> fplot(@(x)[sin(x),cos(x),tan(x)], 2*pi*[-1 1 -1 1])
```

9.2.1 Exercise

Work out what the following commands plot:

```
>> clf, clear
>> f = @(x,n)abs(exp(-1i*x*(0:n-1))*ones(n,1));
>> fplot(@(x)f(x,10),[0 2*pi])
```

Remarks.

(i) Here, in addition to clearing the current figure, we have also cleared all variables and functions from memory.

(ii) Note the use of `i` as the square root of `-1`: see `help i`.

(iii) `f` is an *anonymous function*: see `doc function_handle`.

9.3 Multiple figures and plots

It is possible to have more than one *Figure window* open at any one time by using the `figure` command (see `help figure`). Suppose figure 1 is the current figure, then the command figure(2) (or simply figure) will open a second figure (if necessary) and make it the current figure. The command figure(1) will then expose figure 1 and make it again the current figure. The command `gcf` will return the number of the current figure. Try the following

```
>> figure(1); fplot(@sin,[-pi pi]); hold on
>> figure(2); fplot(@cos,[-pi pi])
>> figure(1); fplot(@cos,[-pi pi])
```

Remark. Figure `n` can be cleared or closed with `clf(n)` or `close(n)`, respectively.

It is also possible to have many graphs in each figure by using the `subplot` command to create axes in tiled positions (see `help subplot`). So for example if you want 3 rows each containing 2 graphs, and you want to plot $sin(x)$, $cos(x)$, $sin^2(x)$, $cos^2(x)$, $sin^3(x)$ and $cos^3(x)$ in the six graphs, you enter (see Figure 7):

```
>> subplot(3,2,1); fplot(@sin,[-pi pi],'b-')
>> subplot(3,2,2); fplot(@cos,[-pi pi],'g-.')
>> subplot(3,2,3); fplot(@(x)sin(x)^2,[-pi pi],'r--')
>> subplot(3,2,4); fplot(@(x)cos(x)^2,[-pi pi],'c-')
>> subplot(3,2,5); fplot(@(x)sin(x)^3,[-pi pi],'m-.')
>> subplot(3,2,6); fplot(@(x)cos(x)^3,[-pi pi],'k--')
```

When it comes to writing up your projects, it is often clearer to include more than one graph on a page. If appropriate you are advised to do so.[32]

[32] Submitting reams of output with only one graph per page may not be the best way to endear yourself to the person marking your projects.

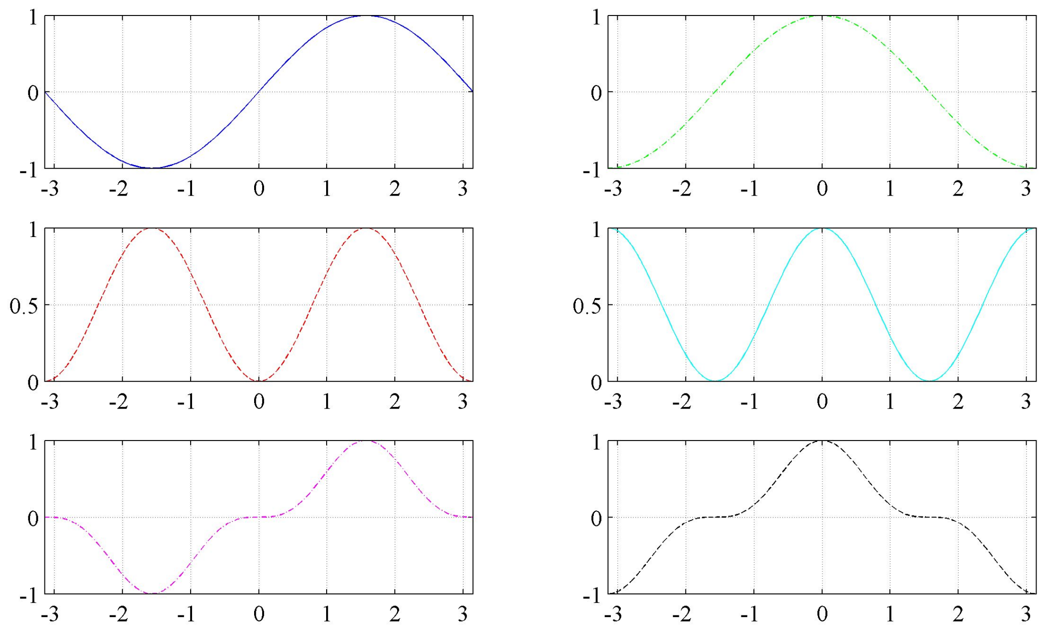

Figure 7: Various sines and cosines.

9.4 Saving your figures

It is possible to save your figures to a file (or to send them to a printer) using the `print` command. The format in which the figure is saved is specified by an option:

Option	**Format**
`-dpdf`	Portable document format
`-dps`	PostScript for black and white printers
`-dpsc`	PostScript for color printers
`-dpsc2`	Level 2 PostScript for color printers
`-deps`	Encapsulated PostScript
`-depsc2`	Level 2 Encapsulated Color PostScript
`-depsc`	Encapsulated Color PostScript
`-dpng`	Portable Network Graphic
`-djpeg<nn>`	JPEG image, quality level of nn; quality level defaults to 75 if nn is omitted.

For later inclusion into a LaTeX document[33] `print -depsc2 <filename>` (colour encapsulated postscript) or `print -dpdf <filename>` (portable document format) might be best. For Word documents, try `print -depsc2 -tiff <filename>`, which adds a TIFF preview.

To print in landscape mode, issue the command `orient landscape` prior to printing.

[33] Also consider using the *psfrag* package which provides for arbitrary replacement of PostScript strings with LaTeX code. This is an excellent way of annotating figures within a LaTeX document.

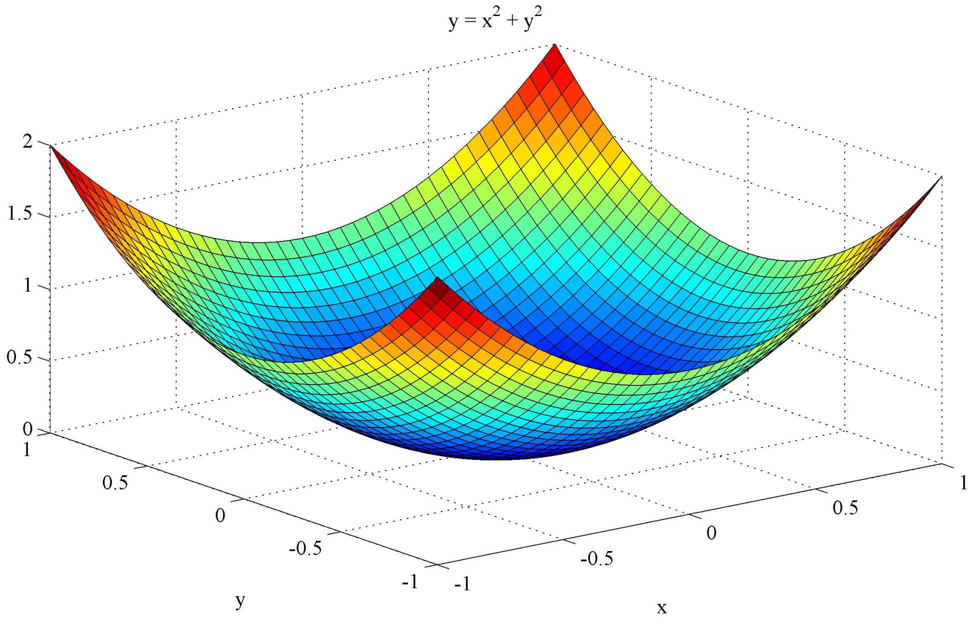

Figure 8: Surface plot of $z = x^2 + y^2$.

9.4.1 Exercise

Save one your graphs in pdf format in a file and open it using an external viewer (e.g. acroread). Then issue the command `orient tall`, save the graph again, and view it again (use `help orient` or `doc orient` to discover the reason for the change).

9.5 3D graphs

There are many 3D graphical tools available in MATLAB. You can find out more using `help graph3d` and `help specgraph` (or `doc graph3d` and `doc specgraph`).

As an example (see Figure 8), the following code draws the surface

$$z = x^2 + y^2 \,,$$

over the region $-1 \leqslant x \leqslant 1$, $-1 \leqslant y \leqslant 1$:

```
>> n=41
>> x=linspace(-1, 1, n);
>> y=linspace(-1, 1, n);
>> [X,Y]=meshgrid(x,y);
>> Z=X.^2+Y.^2;
>> surf(X,Y,Z)
```

Remarks.

(i) The larger the value of `n` the smoother the surface.[34]

[34] With earlier versions of MATLAB on the MCS there was a 'bug' with the result that before setting `n`$\geqslant$`16` it was necessary to issue the command `opengl neverselect`.

(ii) The `meshgrid` command creates two 2D matrices, each with as many elements as there are grid-points. The `X` matrix has the x-coordinates and the `Y` matrix the y-coordinates. The `Z` matrix contains the 'height' of the [single-valued] function defined over the rectangular grid.

(iii) In calculating `Z` note the use of `.^`, to square each element of the matrices `X` and `Y`, rather than the use of `^` (which would have squared the matrices not the elements).

(iv) The colouring of the surface is proportional to surface height. A key to the colours can be added with

```
>> colorbar
```

The type of shading can be modified with

```
>> shading interp
```

(v) The command `mesh(X,Y,Z)` could have been used to draw a wireframe mesh instead of a surface.

10 Random number generation

The simplest way to obtain a pseudo-random number uniformly distributed in $(0, 1)$ in MATLAB is with the command `rand`. Calling this with 2 arguments, `rand(m,n)`, returns an $m \times n$ matrix of uniformly distributed pseudo-random numbers in $(0, 1)$. These are random in the sense that there should be no measurable correlations between successive numbers, but are pseudo-random in the sense that a deterministic algorithm generates these numbers.

Introducing random numbers into any code adds an interesting wrinkle when trying to debug that code. Imagine wanting to check that the code produces the same results before and after making some small change. However, simply calling `rand` introduces an unreproducible element to the code which precludes this kind of check. We can fix this by using the deterministic nature of pseudo-random numbers. We use a *random number stream* initiated with a user-given *seed*. That is, we assign a variable which we pass to `rand` which defines an initial condition for the pseudo-random number generator and keeps track of how many numbers have been generated.

```
>> seed = 1234;  % determines starting point in series of random numbers
>> rns =  RandStream('mt19937ar', 'seed', seed);
>> randomarray = rand(rns,1,10); % array of 10 random numbers
```

Remarks.

(i) You can change `seed` when you want to use a different string of pseudo-random numbers.

(ii) Using `doc RandStream` you can read more in the MATLAB help pages about the generation algorithms; `mt19937ar` seems like a reasonable default.

(iii) You can pass variables of type `RandStream` to functions you write in order to ensure the whole code uses a single random number stream.

(iv) You can define separate random number streams for use in different parts of the code.

Random numbers from many other probability distributions can also be obtained. In particular the command `randn(m,n)` returns an $m \times n$ matrix of values drawn from the (standard) normal distribution with mean zero and variance 1, while `normrnd` generates normally distributed values with any chosen mean and standard deviation[35]. More generally the command `random` can be used for many of the most frequently-needed probability distributions. For details of their use see the corresponding `help` files.

[35] The `normrnd` and `random` functions are part of the Statistics Toolbox, which is an optional extra for MATLAB. If the Statistics Toolbox is not available on your installation (it is at DAMTP) instead use `r = randn(m,n) .* sigma + mu;`

11 Symbolic manipulation

MATLAB's powerful Symbolic Toolbox allows you to define and work easily with symbolic variables. For example you can define a function of x, say, and get MATLAB to differentiate or integrate, find its Taylor expansion to some chosen order, and so on. This can be useful (for example) to provide an extra check that a calculation is correct. *Note: If a project allows you to use symbolic manipulation to carry out key calculations, rather than evaluating by hand, you will normally be told this explicitly.*

For example, to define variables x and y use `syms`

```
>> syms x y
```

Now try defining a function:

```
>> f = exp(-x^2-y^2)

f =

1/exp(x^2 + y^2)
```

By defining `f` in this way it also becomes a symbolic object, that is it has class 'sym'. Try some simple operations on `f`:

```
>> fx      = diff(f);         % differentiate with respect to x
>> fint    = int(fx);         % integrate with respect to x
>> fseries = taylor(f,y);     % Taylor expansion (by default to 5th order)
```

The behaviour of `diff` depends upon the class of its argument. Here it represents symbolic differentiation, but when applied (for example) to a vector V of N numerical values it returns a vector of length $N-1$ of the differences between successive values $\{V(i) - V(i-1)\}$.

By default these commands are carried out with respect to x, but we can tell MATLAB explicitly which variable to use as we have done with `taylor`. Similarly the Taylor expansion can be modified to give as many terms as we choose, e.g. `taylor(f,7)`.

What about symbolic simplification? If two calculations give the same function, MATLAB will not necessarily recognise automatically that they are the same:

```
 >> h= exp(x+y) - exp(x)*exp(y)

h =

exp(x + y) - exp(x)*exp(y)
```

but the command `simplify` will search for what MATLAB regards as simplifications, and in this case gives

```
>> simplify(h)

ans =

0
```

We can even tell MATLAB to work harder to simplify an expression by using more steps in the process. For example `simplify(f,80)` will use 80 in place of the default 50 steps. In many cases only one or two steps will be needed.

For a further glimpse into the extensive capabilities of the Symbolic Toolbox and the functions provided type `help symbolic`.

12 Sets and set operations

MATLAB can also perform elementary operations on sets, including pairwise unions and intersections. Used in this way MATLAB will, for example, treat a numerical array as the set of numbers (ignoring repeats), or a character string as the set of letters it contains. Thus from

```
>> vec1=[1 3 2 1 1 3 3 2];
>> vec2=[3 4 5];
```

we get

```
>> u = union(vec1,vec2)

u = 1 2 3 4 5

>> v = intersect(vec1,vec2)

v = 3
```

and so on. The resulting objects are of the same type as the arguments, and are sorted. Notice that this provides a simple way to sort and remove repeated values, but an even simpler command is `unique`:

```
>> u = unique (vec1)

u = 1 2 3
```

This can also be used to remove repeated rows of a matrix:

```
>> A = [1 2 1; 3 4 5; 1 2 1]

A =
     1     2     1
     3     4     5
     1     2     1
>> B = unique (A)

B =
     1     2     1
     3     4     5
```

13 Troubleshooting

13.1 Errors and debugging

As you write programs, you will inevitably introduce bugs by accident which prevent your program from behaving as you intend or prevent it from working at all. These bugs may be spelling mistakes, incorrectly sized matrix variables, errors in the logic of your program or a host of other mistakes which make the processing of the program differ from your intentions. When you come to run your program (either once it is complete, or for larger programs as you are writing them) you will need to debug the program. Being able to debug software quickly is an important skill which you will develop with experience.

A run-time error will cause the execution to stop. When you come to run your program, you may find that the nature and line number of a bug is picked up by MATLAB and reported as an error message at the command line. For example, calling the following function

```
function [b]=timestwo(a)
  b=2*as
  end
```

would give an error

```
?? Undefined function or variable ’as’.
Error in ==> timestwo at 3
b=2*as;
```

In other cases, your program will simply fail to do what you expect. By investigating what your program is actually doing (as opposed to what you intended it to do), you will be able to identify and remedy the bugs. A general approach to debugging a program is to check that the instructions (particularly any loops or conditional statements) are happening in the order you intend and that during the course of the program, the variables are taking the values that you expect. Tracking this through the course of the program should enable you to identify the bug and correct it.

13.1.1 Debugging from the Editor

Debugging can be done by adding output commands to your program (such as printing the value of a variable at each iteration of a loop) to give you more information about what the program is doing. MATLAB also includes a number of tools to help with debugging. Within the editor GUI, you can add ‘breakpoints’ at a line within your program by clicking on the hyphen in the *left-hand* margin. The breakpoint will be indicated by a red button on the corresponding line.

When the computer comes to process this line in the program, the process will pause and allow you to interrogate the state of the program: you can use the command line to examine or change the value of any variables within the program. Hovering the mouse over the variables within the editor GUI will also give you some information about the value of the variable. To make the program continue, use the continue command from the **Debug** drop-down menu within the editor GUI. A very useful alternative is to use the **Step** command to process one line of your program at a time. Once the program reaches a breakpoint, this option can be selected from the **Debug** menu.

When you have edited a MATLAB file within the editor GUI you may notice several hyphens along the *right-hand* margin, denoting warnings (for example when MATLAB notices you have assigned a value which is never used) or errors. Allowing the mouse to hover over these will show them in detail and in some cases will suggest the correction.

13.1.2 Command line tools

`dbstop`: A similar series of tools can be enacted from the command line by using the `dbstop` commands. This is one of several debugging commands. `dbstop` sets breakpoints in a similar way to the use of breakpoints in **Debug** mode within the GUI.

For example. to make the program stop on the line prior to a previously-observed error and to open the file in the MATLAB Editor, use the command

```
>> dbstop if error
```

Try-catch commands: Another useful and versatile debugging tool is the **try-catch** block, which has the form

```
try
  <program statements>
catch
  <error-handling statements (if required)>
end
<program statements>
```

Normally, a run-time error in a block of code will cause an error message to be output and execution will stop. If the block of code has no errors then the `try-catch-end` commands will have no effect, and the code in the `catch` section will be ignored. However, if an error is encountered then, instead of simply failing, execution will pass to the `catch` block. Even if no code is provided within the `catch` block, execution will simply continue until the program completes or until the next error.

If `catch` is replaced by `catch ME`, say, then `ME` will be of type MException and will contain all the information about the error message as a character string.

The range of uses are best understood by experimenting, and following guidance given in the help files (type `doc try` to open the help browser). One such use might be when a function is called repeatedly, and we do not want execution to be terminated by an error occurring occasionally. Using these commands we can let MATLAB quietly ignore these cases, or alternatively we can pause execution to examine the data leading to the error.

To find out more about how to use the debugging features of MATLAB, as ever, search the online help or use the help command from the command line

```
>> help debug
```

There are many other helpful references on debugging available; for instance:

(a) MATLAB's main guide to debugging at

http://www.mathworks.com/help/techdoc/matlab_prog/f10-60570.html.

(b) A general article by Terence Parr that gives a good flavour of the joys of debugging at

http://parrt.cs.usfca.edu/doc/debugging.html.

(c) A discussion from the MATLAB user community

http://blogs.mathworks.com/loren/2010/03/19/debugging-approaches/;

indeed the MATLAB user community is often a useful resource.

13.2 Timing

It is very often helpful or necessary to find out how much of the total computation time is being spent on specific sections of code. This can help you check, say, whether a change to your algorithm improves efficiency, or how computation time changes as a function of a given parameter.

A simple way is to do this is to measure actual time that has elapsed using `tic` and `toc` around the chosen section of code, for example:

```
>> tic                              % This starts the stopwatch
>> <program statements>
>> toc                              % This stops it
Elapsed time is 0.000505205 seconds
```

Of course, this is much more useful and accurate in an `.m` file than typing directly into the command window, as the clock will be running while you are typing[36]. Note that the measurement may be affected significantly by other jobs running at the same time.

Shorter programs sometimes run too quickly to get useful information directly from these commands. In that case a simple trick is to run the block of code repeatedly in a loop (inside `tic` and `toc`) and take an average.

The `cputime` function measures total CPU time, and in principle can be used in place of these commands, although MATLAB currently recommends using `tic` and `toc` exclusively.

[36] Putting all commands, including the `tic` and `toc` on a single command line avoids counting the time taken to type the commands: `tic;s = std(randn(100,10000));toc`

MATLAB also includes the **Profiler** utility which gives a more comprehensive view of your program and of where time is being spent. This can be launched by selecting it from the Desktop drop-down menu in the MATLAB GUI, or typing `profile viewer` in a command window.

14 This and that

14.1 Programming style

As has already been mentioned, when writing your programs it's good (some would say essential) to include informative comments, and it helps if they are readable (e.g. by sensible use of indentation).

What constitutes the best programming style is debatable, and to some extent is a theology. However, before you start to program in earnest it is probably sensible to read at least one article. As of February 2018 the article in Wikipedia discusses *Elements of Good Style*, and *Code Appearance* grouped under heading *Indentation*, *Vertical Alignment*, *Spaces*, and *Tabs*, and is not bad, although not specific to MATLAB: see

http://en.wikipedia.org/wiki/Programming_style

We mention two of these issues.

14.1.1 Indentation

Indenting can be very helpful in identifying blocks of commands and logical flow. If you choose to use the built-in MATLAB editor it will take care of some of these issues automatically. MATLAB's editor will automatically add the spaces as shown below so that the inner and outer loops are more easily distinguishable.

```
n = 5;
m = 10;
for i  = 1:n                % Outer loop starts
    a  = sin(float(i))
    a2 = a^2
    for j  = 1:m            % Inner loop starts
        b  = cos(float(j))
        c  = a*b
        c2 = c*c
    end                     % Inner loop ends
    d = d+c2
    a = float(j)*d
end                         % Outer loop ends
```

14.1.2 Vertical alignment

It is often helpful to align similar elements vertically, to make it more obvious when typographical errors are introduced. For example when calling a function repeatedly this can make it easier to check that parameters have been given correctly. Similarly aligning a string of arithmetic assignments at the 'equals' sign may be helpful (as in the above example). As another example consider

```
vdiff=funct1(a,b,c,random,RK4,gauss,d)
vaccel=funct1(aa2,bb2,rr2,zz2,euler,dd2)
```

Here the function `funct1` has been called by `vdiff` and `vaccel` with a different number of parameters, in error. This would have been easier to spot if we had written this as

```
vdiff  = funct1(a,   b,   c,   random, RK4,   gauss, d)
vaccel = funct1(aa2, bb2, rr2, zz2,    euler, dd2)
```

14.2 Some terminology

`Inf`. The IEEE (Institute of Electrical & Electronics Engineers) representation of positive infinity.

`NaN`. The IEEE representation of 'Not-a-Number', used when an operation has an undefined result (e.g. 0/0). All logical operations involving `NaN` return false, except for $\neq$. Nothing appears if you try to plot points with the value `NaN`.

Note: It is important to notice that MATLAB will not normally alert you to an error if your calculations result in these quantities. For example, if you accidentally divide by zero at an early stage in a calculation, it will assign `Inf` to your variable, and the execution will continue, simply using this value for the variable throughout subsequent calculations.

15 Sample project: Fibonacci numbers

This sample mini-project is intended to illustrate some of the points, both mathematical and computational, that occur in the Part IB Computational Projects.

15.1 Definitions

The *Fibonacci numbers* u_n are defined by the recurrence relation

$$u_n = u_{n-1} + u_{n-2} \tag{1}$$

and the initial conditions $u_0 = 0$ and $u_1 = 1$. The *Lucas numbers* v_n satisfy the same recurrence, but with initial conditions $v_0 = 2$ and $v_1 = 1$.

15.2 Recursion versus iteration

A function in a computing language is *recursive* if the body of the function contains a further call or calls to that same function.

Question 1 Write a program which reads in a number n and computes the nth Fibonacci number recursively. You should find that even for moderate values of n it begins to take an unreasonable amount of time to complete the calculation. What can you say about the time required to compute in this way? Explain the underlying cause. What about computing the nth Lucas number?

You will probably conclude that *iteration* is a more satisfactory method for computing Fibonacci numbers: that is, computing all the u_i for $i \leqslant n$ successively.

Question 2 Write a program which reads in a number n and computes the nth Fibonacci number and Lucas numbers iteratively. Tabulate u_n and v_n for n up to 25. What can you say about the time required to compute u_n this way? Compare the time and space required to compute Fibonacci numbers by the two methods.

15.3 The size of Fibonacci numbers

Question 3 Use your program from the previous question to experiment with the rate of growth of the Fibonacci numbers. Test the hypothesis that the rate of growth is like n^β for some β versus β^n for some β. You should find that at least one of these hypotheses seems highly plausible: which? From your data, estimate the correct value of the parameter β.

Question 4 Solve the recurrence relation directly to find expression for u_n and v_n. Compare these expressions with your previous results.

16 Acknowledgments

This guide was based on an earlier version for C by C.D. Warner & A.N. Ross. Extracts have been gleaned from documentation available from the Department of Engineering, University of Cambridge, and in particular from that written by Tim Love.[37] Many people have contributed and made suggestions, but particular thanks is due to Tristram Scott, Charlie Hogg, Elie Bassouls and Matthew Perry.

[37] See `http://www.eng.cam.ac.uk/help/tpl/programs/matlab.html`.

A Using Windows: Basics

This section provides some additional information for users new to Windows and complements Section §2. It again assumes that you are sitting in front of one of the Desktop Services (DS) computers, such as those in the CATAM room, GL.04, in the basement of Pavilion G at the CMS.

A.1 Logging in

If the monitor screen is dark, check that the monitor is switched on and move the mouse around to switch off any active *screensaver*. If the computer is turned off (check whether the green power light is lit) press the round power switch. If the computer still does not work, check that the 'wall' power supplies for the computer and monitor (round the back of the desk) are switched on.

After you switch on, a screen will appear asking you to select either MCS Linux or MCS Windows. If Windows is not already highlighted then you should highlight this using the arrow keys (and then select it again if necessary because of a bug in the system). Note that only the computers towards the back of GL.04 default to MCS Windows; those at the front boot to MCS Linux unless you highlight Windows.

If your machine boots into Linux, or Linux is already running, then you can change to Windows by restarting the machine by first selecting *Shut Down*, and then *Restart*, from the menu obtained by clicking on the circular icon at the top right of the screen. Once the machine reboots follow the instructions in the previous paragraph. There is a poster summarising these instructions on the noticeboard at the back of GL.04.

Wait until Windows has started and a window appears saying `Ctrl+Alt+Delete to Login or Shutdown`. While holding down the **Control** and **Alt** keys, press the **Delete** key. This will bring up a window which gives information relating to the *Computer Misuse Act 1990*, the *Regulation of Investigatory Powers Act 2000* and the *Lawful Business Practice Regulations 2000* (read this information at least once, since you are expected to both *be aware of it and abide by it*).[38] Click the **OK** button to close the window.

[38] For the academic year 2017/18 this reads:

> "This computer may be used only by persons authorised to do so by the University of Cambridge. Use by others will be in contravention of the Computer Misuse Act 1990.
>
> As required by the Regulation of Investigatory Powers Act 2000 and the Lawful Business Practice Regulations 2000, the Computing Service draws the attention of all users of the Cambridge University Data Network (CUDN) to the fact that their communications may be intercepted as permitted by legislation. In particular, but without limitation, users should be aware that their communications may be intercepted, as provided for by the Lawful Business Practice Regulations 2000, in order to investigate or detect unauthorized use of the CUDN or networks to which the CUDN is directly or indirectly connected.
>
> See 'Rules and conditions relating to network use' (`http://www.cam.ac.uk/cs/network/rules/`) for documents which specify the authorized use of the CUDN and JANET."

Next a window with boxes for your user name and for your password will come up. Type in your user name and password in the correct boxes and click **OK**. Note you need your password for the MCS (which may, or may not, be different to your Hermes password and your Raven password). The computer will then log you onto the DS. You will see a window appear with some start-up messages but this will disappear when Windows has finished loading. A `Message of the Day` window will also appear. To close this window click the red **X** button that is in the top right-hand corner of the frame.

A.2 Windows basics

The Windows **desktop** contains several windows and icons. Each program (or **application**) running on the computer displays output in one or more windows. A window can be hidden from view by **minimising** or **iconifying** it by left-clicking on the _ (underscore) button that is in the top right-hand corner of the frame. This can stop the screen getting too cluttered.

Windows are controlled using the mouse or keyboard. These notes concentrate on mouse techniques which are easier for the beginner, but a few **keyboard shortcuts** are mentioned. Windows are, in general, controlled with the **left-hand** mouse button using one of the following operations:

- **click:** press and release the left button without moving the mouse;
- **double click:** click the left button twice quickly;
- **drag:** hold down the left button and move the mouse.

Because in earlier versions of Windows, some people found double clicking difficult, in Windows it is possible to set an option that permits a single click to do the same job; if this option is set on your computer, whenever this guide asks you to double click, a single click will have the required effect. Click, i.e. left-click, on a window to make it **active**. The active window appears in front of other windows and responds to characters typed on the keyboard. The active window also has a dark blue rather than a light blue **title bar**. A single click is also used to **select** an item prior to carrying out some action on it. A selection is marked by highlighting or a dotted rectangle around the perimeter of buttons

A double click **chooses** an item. You choose an item to carry out an action, for example to start a program running from an icon on the desktop.

Dragging is used to move or re-size windows[39], select blocks of text for copying or deleting, etc.

A.3 The start menu and task bar

At the bottom of your screen you will typically see a blue bar with a green **start** button on the far left, and perhaps some other buttons to the right of it. This is the **task bar**.

[39] By dragging the title bar or a corner of a window respectively.

Clicking the green **start** button on the left brings up **start menu**. The start menu is used to start up other windows applications such as MATLAB software or Microsoft Word. After clicking on **start**, click on **All Programs**, you will see a **menu** appear. In this list click on **Teaching Packages**, then click on **Catam**. A small menu of Catam related programs will be displayed, click on **MATLAB** to start. The **Catam News** icon gives useful information about the CATAM projects and the computer facilities and you should check it from time to time. The **Maths Computational Projects** icon takes you to the CATAM home page, from which you can access the electronic version of the project booklets and other useful information.

The task bar may also contain several other buttons, one for each window on the screen. If you click on a button the associated window will become the active window. In the far right of the task bar is a clock.

A.4 Window elements

Most windows have several elements which are used to control both the appearance of the window and its associated program.

- The **title bar** identifies the window. Drag the title bar around to move the window. The **active** window (the one receiving input, normally the foremost window), is shown with a bright blue title bar. **Click** anywhere on a window to make it active. Drag the **window border** to change its size.
- The **minimise button**, a blue button showing the _ character, hides the window from view. It still appears on the task bar. Clicking the associated button on the task bar will display the window again and make it the active window.
- The **maximise button**, a blue button showing a square, enlarges the window to its maximum size. For many applications the window fills the entire screen. The maximise button changes its appearance to show two overlapping rectangles. Click the button again to restore the window to its original size.
- The **close button**, a red button showing a **X** character, closes the window and terminates the program if there are no other windows associated with it.
- A **scroll bar** (horizontal or vertical) is displayed when information such as a text document is too large to fit entirely in the window. The position of the **scroll box** within the scroll bar indicates which part of a document is currently displayed in the window.

  Drag the scroll box to move rapidly through a document, click in the scroll bar above or below the **scroll box** to move up or down a page in the document, and click the **scroll arrows** to move forwards or backwards through a document one line at a time.

  Alternatively, you can use the the arrow keys and the page up/down keys to navigate through a document.
- The window **menu bar** contains a list of menus of commands used to control the application. Select a menu by clicking its name, then choose a menu option by clicking or dragging. Most applications have a **File** menu which is used to save and load data files, print, or close the application. There is also usually a **Help** menu which gives information about the program.

A.5 Files and folders

My Computer and **Windows Explorer** are used to organise **files** and **folders** on the computer disks. Windows Explorer is usually found by clicking on **start**, then on **All Programs** then on **Accessories** then on the **Windows Explorer** icon. Information (MATLAB programs, data, results etc.) is stored in files which are given names consisting of two parts: a filename and (optionally) an extension. A filename must not contain any of the characters `\`, `/`, `:`, `*`, `?`, `"`, `< or >` and you are advised not to use any of `'`, `(`, `)` or `space`[40]. A filename extension consists of a full stop, followed by one to three letters or digits. The extension denotes the type of information stored in the file; for instance '**.m**' indicates that a file contains MATLAB script code, while '**.exe**' indicates an executable program. It is not unusual for Windows to have been set up so that the extensions are not shown on the screen when using Windows Explorer.

A.6 Logging out

When you have finished working you need to log off from the network to ensure that no-one else can alter or delete your files. First exit from any programs you have been running. To log out click on **start** then on the **Log Off** icon and finally on the the **Log Off** button. After a few seconds you will see your programs being closed down and the window saying `Press Ctrl+Alt+Delete to Login or Shutdown` appear again. You can now leave the computer.

Please note that it is against Faculty rules to leave yourself logged into a Desktop Services PC while not at the console. If you leave the console for other than a few seconds (e.g. to collect printer output), please log out. This is for the convenience of others, and also for your safety (e.g. you do not want to come back to discover that someone has deleted all your files).

[40] Although sometimes it is very tempting to use `space`.

B A generalised printsquares code

There is more than one way to skin this cat, but one answer is[41]

```
function [ Isquares ] = printsquares ( Ilow, Ihigh )
%PRINTSQUARES Function to print the squares of numbers
%
%   PRINTSQUARES(Ilow,Ihigh) prints the squares from Ilow to Ihigh in
%   steps of one, and returns the answers in the (Ihigh-Ilow+1) x 2
%   matrix Isquares
%
Ilow=round(Ilow); Ihigh=round(Ihigh);
%
% Find how many squares to print
range=abs(Ihigh-Ilow)+1;
%
% Set up a matrix of the correct size to store the results
Isquares=zeros(range,2);
%
% Ensure that the matrix indices are all strictly positive
for I=1:range
  Isquares(I,1)=Ilow+sign(Ihigh-Ilow)*(I-1);
  Isquares(I,2)=Isquares(I,1)*Isquares(I,1);
  fprintf('I = %2g, I*I = %3g\n',Isquares(I,1),Isquares(I,2))
end
%
end
```

Note that we have forced `Ilow` and `Ihigh` to be integers by using `round` (for more information enter `help round`).

[41] This code, or an approximation to it, can be downloaded from
http://www.maths.cam.ac.uk/undergrad/catam/MATLAB/manual/MATLAB/printsquares.m.

C Index of functions in this booklet

NOTE: This is not intended as a complete function reference for MATLAB. It *is* intended as a handy *aide-memoire* should you struggle to recall the name of a function that you may have learned while working through this guide. Further help about the command `command-name` can be obtained from within the MATLAB Command Window, by using the command

```
>> help command-name
```

.

Function purpose	MATLAB command name
General use	
Control loops	for, do, while
Escape a loop	break
Control structure	if, else, switch, case
Declare function	function
Output data/messages	fprintf
Display variable	disp
Suppress output	add semicolon (;) to end of line
Saving/Loading data	save,load
Basic mathematics	
Basic arithmetic	+, -, *, /, ˆ
Componentwise arithmetic	+, -, .*, ./, .ˆ
sin(h),cos(h),tan(h)	sin(h),cos(h),tan(h)
Inverse sin(h),cos(h),tan(h),	asin(h),acos(h),atan(h)
Natural log, log base 10	log,log10
Absolute value	abs
Sign	sign
Square root	sqrt
Rounding to integers	fix,floor,ceil,round
Round to nearest integer	int32
Find remainders	rem
$=, \neq, <, >, \leq, \geq$	==, ~=, <, >, <=, >=
Linear algebra	
Matrix of zeros	zeros
Matrix of ones	ones
Identity matrix	eye
Diagonal matrix	diag
Random matrix	rand
Multidimensional array	multi
Determinant	det
Trace	trace
Eigenvalues/eigenvectors	eig
Characteristic polynomial	poly
Null space	null
Inverse	inv
Matrix “division”	/
Exponential of matrix	expm

Function purpose	MATLAB command name
Graph plotting	
Plot graphs (2D)	plot
Log-log/Log-linear plot	loglog,semilogx,semilogy
Plot function	fplot
Scatter plot	scatter
Surface plot (3D)	surf
Create x, y-input for surf	meshgrid
Change appearance of surface plot	shading,colorbar,mesh...
Title,axis labels	title,xlabel,ylabel
Plot again on same figure	hold on
Overwrite current figure	hold off
Add legend	legend
Change line style/colour	(options in plot command)
General help	graph2d
Change to figure	figure
Make subplots	subplot
Clear figure	clf
Save figures	print