Part IB Computing Course
Tutorial Guide to MATLAB

Paul Smith
Department of Engineering
University of Cambridge

19 September 2001

This document provides an introduction to computing using MATLAB. It will teach you how to use MATLAB to perform calculations, plot graphs, and write simple programs. It also describes the computing exercises to be completed in the Michaelmas term. In addition to this handout you will need to review the programming concepts outlined in the *Tutorial Guide to C++ Programming*, and your IA Mathematics notes. This course will also draw upon some of the concepts learnt in the IB Linear Algebra course. An outline of the computing course including details of the objectives, organisation and timetable of the laboratory sessions can be found on pages 4–5.

Contents

1 Introduction **6**
1.1 What is MATLAB? 6
1.2 What MATLAB is not 6
1.3 Who uses MATLAB? 6
1.4 Why did we learn C++ last year, then? 6

2 Simple calculations **7**
2.1 Starting MATLAB 7
2.2 MATLAB as a calculator 7
2.3 Built-in functions 7

3 The MATLAB environment **8**
3.1 Named variables 8
3.2 Numbers and formatting 11
3.3 Number representation and accuracy 11
3.4 Loading and saving data 12
3.5 Repeating previous commands 13
3.6 Getting help 13
3.7 Other useful commands 14

4 Arrays and vectors **14**
4.1 Building vectors 15
4.2 The colon notation 15
4.3 Vector creation functions 16
4.4 Extracting elements from a vector 16
4.5 Vector maths 16

5 Plotting graphs **17**
5.1 Improving the presentation 18
5.2 Multiple graphs 19
5.3 Multiple figures 20
5.4 Manual scaling 21
5.5 Saving and printing figures 21

6 MATLAB programming I: Script files **24**
6.1 Creating and editing a script 24
6.2 Running and debugging scripts 24
6.3 Remembering previous scripts 25

7 Control statements **26**
7.1 `if...else` selection 26
7.2 `switch` selection 27
7.3 `for` loops 28
7.4 `while` loops 28
7.5 Accuracy and precision 29

8 MATLAB programming II: Functions **33**
8.1 Example 1: Sine in degrees . . . 33
8.2 Creating and using functions . . . 34
8.3 Example 2: Unit step . . . 34

9 Matrices and vectors **38**
9.1 Matrix multiplication . . . 38
9.2 The transpose operator . . . 39
9.3 Matrix creation functions . . . 40
9.4 Building composite matrices . . . 41
9.5 Matrices as tables . . . 41
9.6 Extracting bits of matrices . . . 42

10 Basic matrix functions **42**

11 Solving `Ax` = b **46**
11.1 Solution when `A` is invertible . . . 46
11.2 Gaussian elimination and LU factorisation . . . 47
11.3 Matrix division and the slash operator . . . 47
11.4 Singular matrices and `rank` . . . 47
11.5 Ill-conditioning . . . 49
11.6 Over-determined systems: Least squares . . . 50
11.7 Example: Triangulation . . . 50

12 More graphs **51**
12.1 Putting several graphs in one window . . . 51
12.2 3D plots . . . 52
12.3 Changing the viewpoint . . . 53
12.4 Plotting surfaces . . . 53
12.5 Images and Movies . . . 53

13 Eigenvectors and the Singular Value Decomposition **58**
13.1 The `eig` function . . . 58
13.2 The Singular Value Decomposition . . . 58
13.3 Approximating matrices: Changing rank . . . 60
13.4 The `svd` function . . . 60
13.5 Economy SVD . . . 61

14 Complex numbers **68**
14.1 Plotting complex numbers . . . 69
14.2 Finding roots of polynomials . . . 70

15 Further reading **70**

16 Acknowledgements **70**

Part IB Computing Course   Michaelmas Term

A. Aims of Michaelmas Term Computing Course

This guide provides a tutorial introduction to computing using MATLAB. It will teach you how to use MATLAB to perform calculations, plot graphs, and write simple computer programs to calculate numerical solutions to mathematical problems.

B. Objectives

Session 1: Maths using MATLAB

☐ Familiarisation with the MATLAB environment
☐ Using MATLAB as a calculator
☐ Built-in maths functions
☐ Assignment of values to variables
☐ Vectors as data arrays
☐ Plotting graphs

Session 2: MATLAB programming

☐ Repeating instructions using script files
☐ Control structures
☐ User-defined functions
☐ Numerical errors

Session 3: Matrix-vector equations I

☐ Defining and using matrices in MATLAB
☐ Solutions of $\mathtt{A}\mathbf{x} = \mathbf{b}$
☐ Advanced graphics

Session 4: Matrix-vector equations II

☐ Eigenvectors and eigenvalues
☐ Singular Value Decomposition

C. Organisation and Marking

There are four two-hour laboratory sessions scheduled for the Michaelmas term computing course, one every two weeks. Each session consists of a tutorial and computing exercises. The tutorial contains a large number of examples of MATLAB instructions, and you should work through the tutorial at the computer, typing in the examples. Examples that you can type in look like this:

```
>> a=1+1
a =
    2
```

After each few chapters of this tutorial there is a summary and a computing exercise, which you must complete and then get marked by a demonstrator. *You must get each exercise marked before proceeding to the next section.* In total, there are *12 marks* of Standard Credit available for successful completion of the IB Matlab Computing Course. *You must have all six exercises marked by the end of your fourth session; you will not receive marks for any exercises which are completed after this point.* Exercise 7 is optional, but you should complete this if you have time.

D. Timetable

The computing course requires approximately *8 hours* in timetabled laboratory sessions with an additional *2 to 4 hours* for reading the tutorial guide and preparation *before* the laboratory sessions.

Session number	Objectives and exercises	Time required
Session 1	Introductory lecture	30 minutes
	Tutorial sections 1–5	45 minutes
	Computing exercise (1)	45 minutes
Session 2	Tutorial sections 6–7	30 minutes
	Computing exercises (2) and (3)	30–60 minutes
	Tutorial section 8	15 minutes
	Computing exercise (4)	30 minutes
Session 3	Tutorial sections 9–10	20 minutes
	Computing exercise (5)	30 minutes
	Tutorial sections 11–12	20 minutes
	Computing exercise (6)	30–60 minutes
Session 4	Computing exercise (6) (continued)	30–60 minutes
And if time permits	Tutorial section 13	30 minutes
	Computing exercise (7) (optional)	60 minutes
	Tutorial sections 14–15	30 minutes

Although you are expected to work through the tutorial examples during the session, to benefit most from the laboratory session and help from the demonstrators, you should also *read through the relevant tutorial sections before the start of each session.*

1 Introduction

1.1 What is MATLAB?

MATLAB is an interactive software system for numerical computations and graphics; the name is short for 'Matrix Laboratory'. As the name suggests, it is particularly designed for matrix computations: solving simultaneous equations, computing eigenvectors and eigenvalues and so on, but many real-world engineering problems boil down to these form of solutions. In addition, it can display data in a variety of different ways, and it also has its own programming language which allows the system to be extended. Think of it as a very powerful, programmable, graphical calculator. MATLAB makes it easy to solve a wide range of numerical problems, allowing you to spend more time experimenting and thinking about the wider problem.

1.2 What MATLAB is not

MATLAB is designed to solve mathematical problems *numerically*, that is by calculating values in the computer's memory. This means that it can't always give an exact solution to a problem, and it should not be confused with programs such as Mathematica or Maple, which give *symbolic* solutions by doing the algebraic manipulation. This does not make it better or worse—it is used for different tasks. Many real mathematical problems do not have neat symbolic solutions.

1.3 Who uses MATLAB?

MATLAB is the widely used by engineers and scientists, both in industry and academia, for performing numerical computations, and for developing and testing mathematical algorithms. For example, NASA use it to develop spacecraft docking systems; Jaguar Racing use it to display and analyse data transmitted from their Formula 1 cars; Sheffield University use it to develop software to recognise cancerous cells. Many research groups in Cambridge also use MATLAB. It makes it very easy to write mathematical programs *quickly*, or display data.

1.4 Why did we learn C++ last year, then?

C++ is the industry-standard programming language for writing general-purpose software. However, solutions to mathematical problems take time to program using C++, and the language does not naturally support many mathematical concepts, or displaying graphics.[1] MATLAB is specially designed to solve these kind of problems, perform calculations, and display the results. Even people who will ultimately be writing software in C++ sometimes begin by *prototyping* any mathematical parts using MATLAB, as that allows them to test the algorithms quickly.

MATLAB is available on all of the machines in the Engineering Department and you are encouraged to use MATLAB for any problems which need a numerical solution. It can very easily plot graphs of your data, and can quickly be programmed to run simulations, or solve mathematical problems.

[1] Think back to the IA C++ computing course, where the separate Vogle library had to be used to display the graphs.

2 Simple calculations

2.1 Starting MATLAB

Log onto a computer in the CUED teaching system and start the file manager and MATLAB by typing

```
start 1BMatlab
```

After a pause, a logo may will briefly pop up in another window, and the terminal will display the following information:

```
                          < M A T L A B >
              Copyright 1984-2001 The MathWorks, Inc.
                   Version 6.1.0.450 Release 12.1
                            May 18 2001


  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>>
```

and you are now in the MATLAB environment. The `>>` is the MATLAB prompt, asking you to type in a command.[2]

If you want to leave MATLAB at any point, type `quit` at the prompt.

2.2 MATLAB as a calculator

The simplest way to use MATLAB is just to type mathematical commands at the prompt, like a normal calculator. All of the usual arithmetic expressions are recognised. For example, type

```
>> 2+2
```

at the prompt and press return, and you should see

```
ans =
     4
```

The basic arithmetic operators are `+ - * /`, and `^` is used to mean 'to the power of' (e.g. `2^3=8`). Brackets `( )` can also be used. The order *precedence* is the same usual i.e. brackets are evaluated first, then powers, then multiplication and division, and finally addition and subtraction. Try a few examples.

2.3 Built-in functions

As well as the basic operators, MATLAB provides all of the usual mathematical functions, and a selection of these can be seen in Table 1. These functions are invoked as in C++,

[2] The use of the `start` command is only needed for this series of exercises, since it also creates a `1BMatlab` directory and gives you access to some of the data files you will need later. If you want to use MATLAB for any other calculations, you just need to type `matlab` in any terminal window.

with the name of the function and then the function *argument* (or arguments) in ordinary brackets `()`, for example[3]

```
>> exp(1)
ans =
    2.7183
```

Here is a longer expression: to calculate $1.2\sin(40^\circ + \ln(2.4^2))$, type

```
>> 1.2 * sin(40*pi/180 + log(2.4^2))
ans =
    0.7662
```

There are several things to note here:

- An explicit multiplication sign is always needed in equations, for example between the `1.2` and `sin`.
- The trigonometric functions (for example `sin`) work in *radians*. The factor $\frac{\pi}{180}$ can be used to convert degrees to radians. `pi` is an example of a named *variable*, discussed in the next section.
- The function for a natural logarithm is called '`log`', not '`ln`'.

Using these functions, and the usual mathematical constructions, MATLAB can do all of the things that your normal calculator can do.

3 The MATLAB environment

As we can see from the examples so far, MATLAB has an *command-line* interface—commands are typed in one at a time at the prompt, each followed by return. MATLAB is an *interpreted* language, which means that each command is converted to machine code after it has been typed. In contrast, the C++ language which you used last year is a *compiled* language. In a compiled language, the whole program is typed into a text editor, these are all converted into machine code in one go using a *compiler*, and then the whole program is run. These compiled programs run more quickly than an interpreted program, but take more time to put together. It is quicker to try things out with MATLAB, even if the calculation takes a little longer.[4]

3.1 Named variables

In any significant calculation you are going to want to store your answers, or reuse values, just like using the memory on a calculator. MATLAB allows you to define and use named variables. For example, consider the degrees example in the previous section. We can define a variable `deg` to hold the conversion factor, writing

[3]A function's arguments are the values which are passed to the function which it uses to calculate its response. In this example the argument is the value '1', so the exponent function calculates the exponential of 1 (and returns the value of 2.7183).

[4]MATLAB functions (see Section 8) are also compiled, or can be written in C++. Written correctly, MATLAB programs are not any slower than the equivalent program in C++.

`cos`	Cosine of an angle (in radians)
`sin`	Sine of an angle (in radians)
`tan`	Tangent of an angle (in radians)
`exp`	Exponential function (e^x)
`log`	Natural logarithm (NB this is $\log_e$, not $\log_{10}$)
`log10`	Logarithm to base 10
`sinh`	Hyperbolic sine
`cosh`	Hyperbolic cosine
`tanh`	Hyperbolic tangent
`acos`	Inverse cosine
`acosh`	Inverse hyperbolic cosine
`asin`	Inverse sine
`asinh`	Inverse hyperbolic sine
`atan`	Inverse tangent
`atan2`	Two-argument form of inverse tangent
`atanh`	Inverse hyperbolic tangent
`abs`	Absolute value
`sign`	Sign of the number (-1 or $+1$)
`round`	Round to the nearest integer
`floor`	Round down (towards minus infinity)
`ceil`	Round up (towards plus infinity)
`fix`	Round towards zero
`rem`	Remainder after integer division

Table 1: Basic maths functions

```
>> deg = pi/180
deg =
    0.0175
```

Note that the *type* of the variable does not need to be defined, unlike in C++. All variables in MATLAB are floating point numbers.[5] Using this variable, we can rewrite the earlier expression as

```
>> 1.2 * sin(40*deg + log(2.4^2))
ans =
    0.7662
```

which is not only easier to type, but it is easier to read and you are less likely to make a silly mistake when typing the numbers in. Try to define and use variables for all your common numbers or results when you write programs.

You will have already have seen another example of a variable in MATLAB. Every time you type in an expression which is *not* assigned to a variable, such as in the most recent example, MATLAB assigns the answer to a variable called `ans`. This can then be used in exactly the same way:

```
>> new = 3*ans
new =
    2.2985
```

Note also that this is not the answer that would be expected from simply performing 3×0.7662. Although MATLAB *displays* numbers to only a few decimal places (usually four), it stores them internally, and in variables, to a much higher precision, so the answer given is the more accurate one.[6] In all numerical calculations, an appreciation of the rounding errors is very important, and it is essential that you do not introduce any more errors than there already are! This is another important reason for storing numbers in variables rather than typing them in each time.

When defining and using variables, the capitalisation of the name is important: `a` is a different variable from `A`. There are also some variable names which are already defined and used by MATLAB. The variable `ans` has also been mentioned, as has `pi`, and `i` and `j` are also defined as $\sqrt{-1}$ (see Section 14). MATLAB won't stop you redefining these as whatever you like, but you might confuse yourself, and MATLAB, if you do! Likewise, giving variables names like `sin` or `cos` is allowed, but also not to be recommended.

If you want to see the value of a variable at any point, just type its name and press return. If you want to see a list of all the named variables you have created, type

```
>> who

Your variables are:

ans        deg        new
```

[5] Or strings, but those are obvious from the context. However, even strings are stored as a vector of character ID numbers.

[6] The value of `ans` here is actually 0.76617765102969 (to 14 decimal places).

You will occasionally want to remove variables from the workspace, perhaps to save memory, or because you are getting confusing results using that variable and you want to start again. The `clear` command will delete all variables, or specifying

```
clear name
```

will just remove the variable called *name*.

3.2 Numbers and formatting

We have seen that MATLAB usually displays numbers to five significant figures. The `format` command allows you to select the way numbers are displayed. In particular, typing

```
>> format long
```

will set MATLAB to display numbers to fifteen significant figures, which is about the accuracy of MATLAB's calculations. If you type `help format` you can get a full list of the options for this command. With `format long` set, we can view the more accurate value of `deg`:

```
>> deg
deg =
   0.01745329251994
>> format short
```

The second line here returns MATLAB to its normal display accuracy.

MATLAB displays very large or very small numbers using *exponential notation*, for example: $13142.6 = 1.31426 \times 10^4$, which is displayed by MATLAB as `1.31426e+04`. You can also type numbers into MATLAB using this format.

There are also some other kinds of numbers recognised, and calculated, by MATLAB:

Complex numbers (e.g. `3+4i`) Are fully understood by MATLAB, as discussed further in Section 14

Infinity (`Inf`) The result of dividing a number by zero. This is a valid answer to a calculation, and may be assigned to a variable just like any other number

Not a Number (`NaN`) The result of zero divided by zero, and also of some other operations which generate undefined results. Again, this may be treated just like any other number (although the results of any calculations using it are still always `NaN`).

3.3 Number representation and accuracy

Numbers in MATLAB, as in all computers, are stored in binary rather than decimal. In decimal (base 10), 12.25 means

$$12.25 = 1 \times 10^1 + 2 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$$

but in binary (base 2) it would be written as

$$1101.01 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 12.25$$

Using binary is ideal for computers since it is just a series of ones and zeros, which can be represented by whether particular circuits are on or off. A problem with representing numbers in a computer is that each number can only have a finite number circuits, or *bits*, assigned to it, and so can only have a finite number of digits. Consider this example:

```
>> 1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2
ans =
    5.5511e-017
```

The result is very small, but not exactly zero, which is the correct answer. The reason is that 0.2 cannot be *exactly* represented in binary using a finite number of digits (it is 0.0011001100 . . .). This is for exactly the same reasons that that 1/3 cannot be exactly written as a base 10 number. MATLAB (and all other computer programs) represent these numbers with the closest one they can, but repeated uses of these approximations, as seen here, can cause problems. We will return to this problem throughout this tutorial.

3.4 Loading and saving data

When you exit MATLAB, you lose all of the variables that you have created. Fortunately, if you need to quit MATLAB when you are in the middle of doing something, you can save your current session so that you can reload it later. If you type

```
>> save anyname
```

it will save the entire workspace to a file called `anyname.mat` in your current directory (note that MATLAB automatically appends `.mat` to the end of the name you've given it). You can then quit MATLAB, and when you restart it, you can type

```
>> load anyname
```

which will restore your previous workspace, and you can carry on from where you left off.

You can also `load` and `save` specific variables. The basic format is

save *filename var1 var2 ...*

For example, if you wanted to save the `deg` variable, to save calculating it from scratch (which is admittedly not very difficult in this case!), you could type

```
>> save degconv deg
```

This will save the variable into a file called `degconv.mat` You can reload it by typing

```
>> load degconv
```

MATLAB will also load data from text files, which is particularly useful if you want to plot or perform calculations on measurements from some other source. The text file should contain rows of space-separated numbers. One such file is `cricket.txt`, which will be considered in Exercise 6. If you type

```
>> load cricket.txt
```

this will create a new matrix, called `cricket`, which contains this data.

3.5 Repeating previous commands

MATLAB keeps a record of all the commands you have typed during this session, and you can use the cursor keys ↑ and ↓ to review the previous commands (with the most recent first). If you want to repeat one of these commands, just find it using the cursor keys, and press return.

If you are looking for a particular previous command, and you know the first few letters of the line you typed, you can type those letters and then press ↑, and it will find all previous lines starting with those letters. For example, typing `d`↑ will find first the `deg` command you typed earlier, and then if you press ↑ again, it will find the the `deg = pi/180` line.

Once a command has been recalled, you can edit it before running it again. You can use ← and → to move the cursor through the line, and type characters or hit `Del` to change the contents. This is particularly useful if you have just typed a long line and then MATLAB finds an error in it. Using the arrow keys you can recall the line, correct the error, and try it again.

3.6 Getting help

If you are not sure what a particular MATLAB command does, or want to find a particular function, MATLAB contains an integrated help system. The basic form of using help is to type

```
help commandname
```

For example:

```
>> help sqrt

 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also SQRTM.
```

One way in which the help for a command is not very helpful is that the name of the command and the examples are always given in UPPER CASE. Don't copy these exactly—MATLAB commands are almost always in lower case. So the square root function is `sqrt()`, *not* `SQRT()` as the help message seems to indicate.

If you don't know the name of the command you want, there are two ways you can try to find it. A good place to start is by just typing

```
>> help
```

This will give a list of topics for which help is available. Amongst this list, for example, is `matlab/elfun - Elementary math functions`. If you now type

```
>> help elfun
```

you will get a list of many of the maths functions available (of which Table 1 shows only a tiny subset). Take some time browsing around the help system to get an idea of the commands that MATLAB provides.

If you can't find what you're looking for this way, the MATLAB command `lookfor` will search the help database for a particular word or phrase. If you didn't know the name of the square root function, you could type

```
>> lookfor root
```

MATLAB then displays a list of all the functions with the word 'root' in their help message. Be warned that this can take some time. It also pays to be as specific as possible. As you can see if you type this example into MATLAB, there are a lot of functions which involve the word 'root'. If you want to look for a phrase, it needs to be enclosed in single quotes (which is how MATLAB specifies text strings). A better search would be

```
>> lookfor 'square root'
```

3.7 Other useful commands

If you find yourself having typed a command which is taking a long time to execute (or perhaps it is one of your own programs which has a bug which causes it to repeat endlessly), it is very useful to know how to stop it. You can cancel the current command by typing

```
Ctrl-C
```

which should (perhaps after a small pause) return you to the command prompt.

You probably remember semicolons '`;`' from the IA C++ computing course, where they had to come at the end of almost every line. Semicolons are not required in MATLAB, but do serve a useful purpose. If you type the command as we have been doing so far, without a final semicolon, MATLAB always displays the result of the expression. However, if you finish the line with a semicolon, it stops MATLAB displaying the result. This is particularly useful if you don't need to know the result there and then, or the result would otherwise be an enormous list of numbers:

Finally, for a bit of fun, try typing the following command a few times:

```
>> why
```

4 Arrays and vectors

Many mathematical problems work with sequences of numbers. In C++ these were called *arrays*; in MATLAB they are just examples of *vectors*. Vectors are commonly used to represent the three dimensions of a position or a velocity, but a vector is really just a list of numbers, and this is how MATLAB treats them. In fact, vectors are a simple case of a *matrix* (which is just a two-dimensional grid of numbers). A vector is a matrix with only one row, or only one column. We will see later that it is often important to distinguish between *row* vectors $(\cdots)$ and *column* vectors $\begin{pmatrix} \vdots \end{pmatrix}$, but for the moment that won't concern us.

4.1 Building vectors

There are lots of ways of defining vectors and matrices. Usually the easiest thing to do is to type the vector inside *square* brackets `[]`, for example

```
>> a=[1 4 5]
a =
     1     4     5
>> b=[2,1,0]
b =
     2     1     0
>> c=[4;7;10]
c =
     4
     7
    10
```

A list of numbers separated by spaces or commas, inside square brackets, defines a row vector. Numbers separated by semicolons, or carriage returns, define a column vector.

You can also construct a vector from an existing vector, for example

```
>> a=[1 4 5]
a =
     1     4     5
>> d=[a 6]
d =
     1     4     5     6
```

4.2 The colon notation

A useful shortcut for constructing a vector of counting numbers is using the colon symbol ‘`:`’, for example

```
>> e=2:6
e =
     2     3     4     5     6
```

The colon tells MATLAB to create a vector of numbers starting from the first number, and counting up to (and including) the second number. A third number may also be added between the two, making $a : b : c$. The middle number then specifies the increment between elements of the vector.

```
>> e=2:0.3:4
e =
    2.0000    2.3000    2.6000    2.9000    3.2000    3.5000    3.8000
```

Note that if the increment step is such that it can’t exactly reach the end number, as in this case, it generates all of the numbers which do not exceed it. The increment can also be negative, in which case it will count down to the end number.

`zeros(`M, N`)`	Create a matrix where every element is zero. For a row vector of size n, set $M = 1, N = n$
`ones(`M, N`)`	Create a matrix where every element is one. For a row vector of size n, set $M = 1, N = n$
`linspace(`$x1, x2, N$`)`	Create a vector of N elements, evenly spaced between $x1$ and $x2$
`logspace(`$x1, x2, N$`)`	Create a vector of N elements, logarithmically spaced between 10^{x1} and 10^{x2}

Table 2: Vector creation functions

4.3 Vector creation functions

MATLAB also provides a series of functions for creating vectors. These are outlined in Table 2. The first two in this table, `zeros` and `ones` also work for matrices, and the two function arguments, M and N, specify the number of *rows* and *columns* in the matrix respectively. A row vector is a matrix which has one row and as many columns as the size of the vector. Matrices are covered in more detail in Section 9.

4.4 Extracting elements from a vector

Individual elements are referred to by using normal brackets `()`, and they are numbered starting at *one*, not zero as in C++. If we define a vector

```
>> a=[1:2:6 -1 0]
a =
     1     3     5    -1     0
```

then we can get the third element by typing

```
>> a(3)
ans =
     5
```

The colon notation can also be used to specify a range of numbers to get several elements at one time

```
>> a(2:5)
ans =
     3     5    -1     0
>> a(1:2:5)
ans =
     1     5     0
```

4.5 Vector maths

Storing a list of numbers in one vector allows MATLAB to use some of its more powerful features to perform calculations. In C++ if you wanted to do the same operation on a list of numbers, say you wanted to multiply each by 2, you would have to use a `for` loop to step through each element (see the IA C++ Tutorial, pages 29–31). This can also be

done in MATLAB (see Section 7), but it is much better to make use of MATLAB's vector operators.

Multiplying all the numbers in a vector by the same number, is as simple as multiplying the whole vector by number. This example uses the vector `a` defined earlier:

```
>> a * 2
ans =
     2     6    10    -2     0
```

The same is also true for division. You can also add the same number to each element by using the `+` or `-` operators, although this is not a standard mathematical convention.

Multiplying two vectors together in MATLAB follows the rules of matrix multiplication (see Section 9), which doesn't do an element-by-element multiplication. If you want to do this, MATLAB defines the operators `.*` and `./`, for example

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \texttt{.*} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \end{pmatrix}$$

Note the '`.`' in front of each symbol, which means it's a point-by-point operation. In MATLAB:

```
>> b=[1 2 3 4 5];
>> a.*b
ans =
     1     6    15    -4     0
```

MATLAB also defines a `.^` operator, which raises each element of the vector to the specified power. If you add or subtract two vectors, this is also performed on a point-by-point basis.

All of the element-by-element vector commands (`+ - ./ .* .^`) can be used between two vectors, as long as they are the *same size and shape*. Otherwise corresponding elements cannot be found.

Some of MATLAB's functions also know about vectors. For example, to create a list of the value of sine at 60-degree intervals, you can first define the angles you want

```
>> angles=[0:pi/3:2*pi]
angles =
         0    1.0472    2.0944    3.1416    4.1888    5.2360    6.2832
```

And then just feed them all into the sine function:

```
>> y=sin(angles)
y =
         0    0.8660    0.8660    0.0000   -0.8660   -0.8660   -0.0000
```

5 Plotting graphs

MATLAB has very powerful facilities for plotting graphs. The basic command is `plot(x,y)`, where `x` and `y` are the co-ordinates. If given just one pair of numbers it plots a point, but usually `x` and `y` are vectors, and it plots all these points, joining them up with straight lines. The sine curve defined in the previous section can be plotted by typing

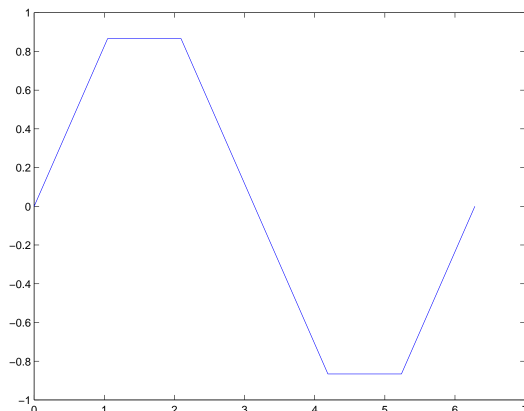


Figure 1: Graph of $y = \sin(x)$, sampled every $60°$.

```
>> plot(angles,y)
```

A new window should open up, displaying the graph, shown in Figure 1. Note that it automatically selects a sensible scale, and plots the axes.

At the moment it does not look particularly like a sine wave, because we have only taken values one every 60 degrees. To plot a more accurate graph, we need to calculate `y` at a higher resolution:

```
>> angles=linspace(0,2*pi,100);
>> y=sin(angles);
>> plot(angles, y);
```

The `linspace` command creates a vector with 100 values evenly spaced between 0 and 2π (the value 100 is picked by trial and error). Try using these commands to re-plot the graph at this higher resolution. Remember that you can use the arrow keys $\uparrow$ and $\downarrow$ to go back and reuse your previous commands.

5.1 Improving the presentation

You can select the colour and the line style for the graph by using a third argument in the `plot` command. For example, to plot the graph instead with red circles, type

```
>> plot(angles, y, 'ro')
```

The last argument is a string which describes the desired styles. Table 3 shows the possible values (also available by typing `help plot` in MATLAB).

To put a title onto the graph, and label the axes, use the commands `title`, `xlabel` and `ylabel`:

```
>> title('Graph of y=sin(x)')
>> xlabel('Angle')
>> ylabel('Value')
```

Strings in MATLAB (such as the names for the axes) are delimited using apostrophes (`'`).

A grid may also be added to the graph, by typing

`y`	yellow	`.`	point	`-`	solid
`m`	magenta	`o`	circle	`:`	dotted
`c`	cyan	`x`	x-mark	`-.`	dashdot
`r`	red	`+`	plus	`--`	dashed
`g`	green	`*`	star		
`b`	blue	`s`	square		
`w`	white	`d`	diamond		
`k`	black	`v`	triangle (down)		
		`^`	triangle (up)		
		`<`	triangle (left)		
		`>`	triangle (right)		
		`p`	pentagram		
		`h`	hexagram		

Table 3: Colours and styles for symbols and lines in the `plot` command (see `help plot`).

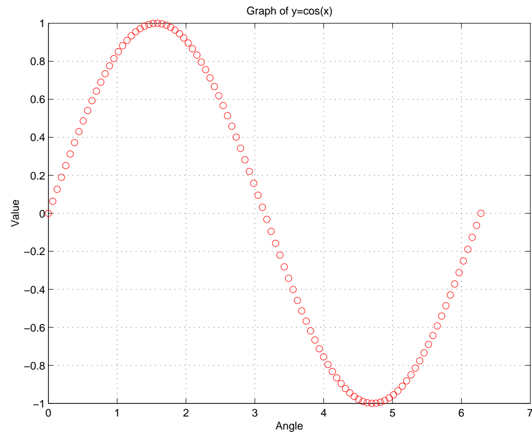


Figure 2: Graph of $y = \sin(x)$, marking each sample point with a red circle.

```
>> grid on
```

Figure 2 shows the result. You can resize the figure or make it a different height and width by dragging the corners of the figure window.

5.2 Multiple graphs

Several graphs can be drawn on the same figure by adding more arguments to the `plot` command, giving the `x` and `y` vectors for each graph. For example, to plot a cosine curve as well as the previous sine curve, you can type

```
>> plot(angles,y,':',angles,cos(angles),'-')
```

where the extra three arguments define the sine curve and its line style. You can add a legend to the plot using the `legend` command:

```
>> legend('Sine', 'Cosine')
```

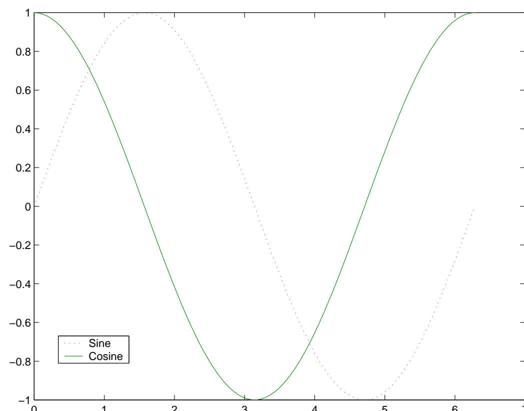


Figure 3: Graphs of $y = \sin(x)$ and $y = \cos(x)$.

where you specify the names for the curves in the order they were plotted. If the legend doesn't appear in a sensible place on the graph, you can pick it up with the mouse and move it elsewhere. You should be able to get a pair of graphs looking like Figure 3.

Thus far, every time you have issued a `plot` command, the existing contents of the figure have been removed. If you want to keep the current graph, and overlay new plots on top of it, you can use the `hold` command. Using this, the two sine and cosine graphs could have been drawn using two separate `plot` commands:

```
>> plot(angles,y,':')
>> hold on
>> plot(angles,cos(angles),'g-')
>> legend('Sine', 'Cosine')
```

Notice that if you do this MATLAB does not automatically mark the graphs with different colours, but it can still work out the legend. If you want to release the lock on the current graphs, the command is (unsurprisingly) `hold off`.

5.3 Multiple figures

You can also have multiple figure windows. If you type

```
>> figure
```

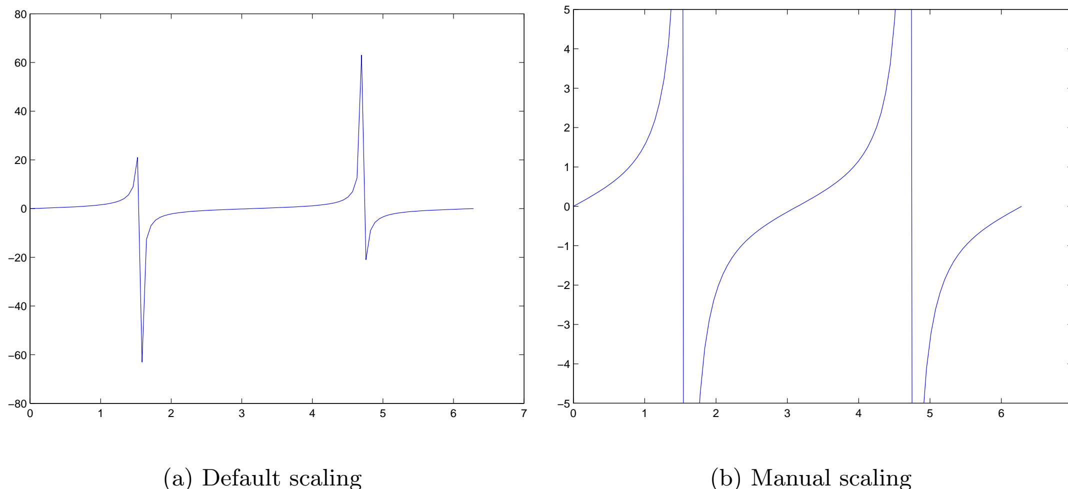
a new window will appear, with the title 'Figure No. 2'. All `plot` commands will now go to this new window. For example, typing

```
>> plot(angles, tan(angles))
```

will plot the tangent function in this new window (see Figure 4(a)).

If you want to go back and plot in the first figure, you can type

```
>> figure(1)
```



(a) Default scaling

(b) Manual scaling

Figure 4: Graph of $y = \tan(x)$ with the default scaling, and using `axis([0 7 -5 5])`

5.4 Manual scaling

The tangent function that has just been plotted doesn't look quite right because the `angles` vector only has 100 elements, and so very few points represent the asymptotes. However, even with more values, and including $\pi/2$, it would never look quite right, since displaying infinity is difficult (MATLAB doesn't even try).

We can hide these problems by zooming in on the part of the graph that we're really interested in. The `axis` command lets us manually select the axes. The `axis` command takes one argument which is a vector defined as $\left(x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}\right)$. So if we type

```
>> figure(2)
>> axis([0 7 -5 5])
```

the graph is rescaled as shown in Figure 4(b). Note the two sets of brackets in the `axis` command—the normal brackets which surround the argument passes to the function, and the square brackets which define the vector, which *is* the argument.

5.5 Saving and printing figures

The `File` menu on the figure window gives you access to commands that let you save your figure, or print it. Figures can be saved in MATLAB's own `.fig` format (so that they can be viewed in MATLAB later) or, the `File`→`Export...` command will let you save it as an image file such as `.jpg` or `.eps`.

The `File`→`Print...` command will send your figure to the printer, and if you are in the DPO then by default this will send it to the main laser printer. Remember that this only prints in black and white! You can also type `print` at the MATLAB command prompt to send the current figure to the printer. However, please don't print things out unless you absolutely have to. You do not need to print out the graphs for your exercises to get them marked—the demonstrators can see them perfectly well on the screen.

Part IB Computing Course — Session 1

Summary

After reading through sections 1 to 5 of the tutorial guide and working through the examples you should be able to:

- ☐ Perform simple calculations by typing expressions in at the MATLAB prompt
- ☐ Assign values to variables and use these in expressions
- ☐ Build vectors and use them in calculations
- ☐ Plot 2D graphs

Exercise 1: π^e vs e^π

A. Exercise

Which is larger, π^e or e^π? Use MATLAB to evaluate both expressions.

In the general case, for what values of x is x^e greater than e^x? Use MATLAB to plot graphs of the two functions, on the same axes, between the values of 0 and 5. Display the two graphs using different line styles and label them fully.

In a separate figure, plot and label the graph of $y = x^e - e^x$, again between x-values of 0 and 5. For what value of x does $x^e = e^x$?

B. Implementation notes

1. **Tutorial sections**
   The tutorial sections immediately preceding this exercise give a number of examples of plotting graphs and performing calculations. Make sure that you have read these sections fully before you start this exercise.

2. **MATLAB constants and functions**
   Both π and e are known to MATLAB. The variable `pi` holds the value of π, and the function `exp(x)` returns the value of e^x. Remember that $e^1 = e$.

3. **Powers of scalars and vectors**
   To raise a single number (a scalar) to a power, you use the `^` operator. If you want to raise each element of a vector to a power, you must use the `.^` operator, with the dot to say 'this is a point-by-point operation'.

4. **Preparing graphs**
   To make a start with your graphs, define the vector of numbers for your x-axis. You can use either the `linspace` function, or the colon notation (see Sections 4.2 and 4.3). Once you have this vector, use it in your arithmetic to create a new vector(s) containing the y values.

C. Evaluation and marking

Show your two figures to a demonstrator for marking. Tell the demonstrator which of π^e or e^π is larger, and at what value the functions x^e and e^x are equal.

6 MATLAB programming I: Script files

If you have a series of commands that you will want to type again and again, you can store them away in a MATLAB *script*. This is a text file which contains the commands, and is the basic form of a MATLAB program. When you run a script in MATLAB, it has the same effect as typing the commands from that file, line by line, into MATLAB. Scripts are also useful when you're not quite sure of the series of commands you want to use, because its easier to edit them in a text file than using the cursor keys to recall and edit previous lines you've tried.

MATLAB scripts are normal text files, but they must have a `.m` extension to the filename (e.g. `run.m`). For this reason, they are sometimes also called *m-files*. The rest of the filename (the word '`run`' in this example) is the command that you type into MATLAB to run the script.

6.1 Creating and editing a script

You can create a script file in any text editor (e.g. `emacs`, `notepad`), and you can start up a text editor from within MATLAB by typing

```
>> edit
```

This will start up the editor in a new window. If you want to edit an existing script, you can include the name of the script. If, for example, you did have a script called `run.m`, typing `edit run`, would open the editor and load up that file for editing.

In the editor, you just type the commands that you want MATLAB to run. For example, start up the editor and then type the following commands into the editor

```
% Script to calculate and plot a rectified sine wave
t = linspace(0, 10, 100);
y = abs(sin(t)); %The abs command makes all negative numbers positive
plot(t,y); title('Rectified Sine Wave');
labelx('t');
```

The percent symbol (`%`) identifies a *comment*, and any text on a line after a `%` is ignored by MATLAB. Comments should be used in your scripts to describe what it does, both for the benefit of other people looking at it, and for yourself a few weeks down the line.

Select `File` → `Save As...` from the editor's menu, and save your file as `rectsin.m`. You have now finished with the editor window, but you might as well leave it open, since you will no doubt need the editor again.

6.2 Running and debugging scripts

To run the script, type the name of the script in the main MATLAB command window. For the script you have just created, type

```
>> rectsin
```

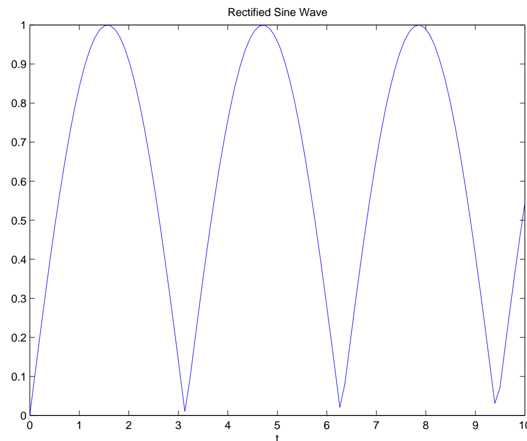


Figure 5: A rectified sine wave, produced by the `rectsin` script

A figure window should appear showing a rectified sine wave. However, if you typed the script into the editor exactly as given above, you should also now see in the MATLAB command window:

```
??? Undefined function or variable 'labelx'.

Error in ==> /amd_tmp/needle-16/users1/pgrad/pas1001/1BMatlab/rectsin.m
On line 6  ==> labelx('t');
```

which shows that there is an error in the script.[7] MATLAB error messages make more sense if they are read from *bottom to top*. This one says that on line 6 of `rectsin.m`, it doesn't know what to do about the command '`labelx`'. The reason is, of course that the command should be '`xlabel`'. Go back to the editor, correct this line, and then *save the file again*. Don't forget to save the file each time you edit it.

Try running the corrected script by typing `rectsin` again, and this time it should correctly label the x-axis, as shown in Figure 5.

6.3 Remembering previous scripts

Scripts are very useful in MATLAB, and if you are not careful you soon find yourself with many scripts to do different jobs and you then start to forget which is which. If you want to know what scripts you have, you can use the `what` command to give a list of all of your scripts. At the moment you will probably only have one:

```
>> what

M-files in the current directory /amd_tmp/needle-16/users1/pgrad/pas1001/1BMatlab

rectsin
```

What is more, the MATLAB help system will automatically recognise your scripts. If you ask for help on the `rectsin` script you get

[7] If you spotted the error when you were typing the script in, and corrected it, well done!

```
>> help rectsin

  Script to calculate and plot a rectified sine wave
```

MATLAB assumes that the first full line(s) of comments in the M-file is a description of the script, and this is what it prints when you ask for help.

7 Control statements

Thus far the programs and expressions that we have seen have contained simple, sequential operations. The use of vectors (and later matrices) enable some more sophisticated computations to be performed using simple expressions, but to proceed further we need some more of the standard programming constructs. MATLAB supports the usual loops and selection facilities. You should be familiar with all of these from the IA C++ computing course.

7.1 `if...else` selection

In programs you quite often want to perform different commands depending on some test. The `if` command is the usual way of allowing this. The general form of the `if` statement in MATLAB is

```
if expression
  statements
elseif expression
  statements
else
  statements
end
```

This is slightly different to the syntax in seen C++: brackets `()` are not needed around the expression (although can be used for clarity), and the block of *`statements`* does not need to be delimited with braces `{}`. Instead, the `end` command is used to mark the end of the `if` statement.

While control statements, such as `if`, are usually seen in MATLAB scripts, they can also be typed in at the command line as in this example:

```
>> a=0; b=2;
>> if a > b
     c=3
   else
     c=4
   end
c =
     4
```

If you are typing them in at the command prompt, MATLAB waits until you have typed the final `end` before evaluating the expression.

<table>
<tr><th>symbol</th><th>meaning</th><th>example</th></tr>
<tr><td><code>==</code></td><td>equal</td><td><code>if x == y</code></td></tr>
<tr><td><code>~=</code></td><td>not equal</td><td><code>if x ~= y</code></td></tr>
<tr><td><code>></code></td><td>greater than</td><td><code>if x > y</code></td></tr>
<tr><td><code>>=</code></td><td>greater than or equal</td><td><code>if x >= y</code></td></tr>
<tr><td><code><</code></td><td>less than</td><td><code>if x < y</code></td></tr>
<tr><td><code><=</code></td><td>less than or equal</td><td><code>if x <= y</code></td></tr>
<tr><td><code>&</code></td><td>AND</td><td><code>if x == 1 & y > 2</code></td></tr>
<tr><td><code>|</code></td><td>OR</td><td><code>if x == 1 | y > 2</code></td></tr>
<tr><td><code>~</code></td><td>NOT</td><td><code>x = ~y</code></td></tr>
</table>

Table 4: Boolean expressions

Many control statements rely on the evaluation of a *logical expression*—some statement that can be either true or false depending on the current values. In MATLAB, logical expressions return numbers: 0 if the expression is false and 1 if it is true:

```
>> 1==2
ans =
     0
>> pi > exp(1) & sqrt(-1) == i
ans =
     1
```

a complete set of relational and logical operators are available, as shown in Table 4. Note that they are not quite the same as in C++.

7.2 `switch` selection

If you find yourself needing multiple `if/elseif` statements to choose between a variety of different options, you may be better off with a `switch` statement. This has the following format:

```
switch x
  case x1 ,
    statements
  case x2 ,
    statements
  otherwise,
    statements
end
```

In a `switch` statement, the value of *x* is compared with each of the listed `case`s, and if it finds one which is equal then the corresponding statements are executed. Note that, unlike C++, a `break` command is not necessary—MATLAB only executes commands until the next `case` command. If no match is found, the `otherwise` statements are executed, if present. Here is an example:

```
>> a=1;
>> switch a
     case 0
       disp('a is zero');
     case 1
       disp('a is one');
     otherwise
       disp('a is not a binary digit');
   end
a is one
```

The `disp` function displays a value or string. In this example it is used to print strings, but it can also be used with variables, e.g. `disp(a)` will print the value of `a` (in the same way as leaving the semicolon off).

7.3 `for` loops

The programming construct you are likely to use the most is the `for` loop, which repeats a section of code a number of times, stepping through a set of values. In MATLAB you should try to use vector arithmetic rather than a `for` loop, if possible, since a `for` loop is about 40 times slower.[8] However, there are times when a `for` loop is unavoidable. The syntax is

```
for variable = vector
    statements
end
```

Usually, *vector* is expressed in the colon notation (see Section 4.2), as in this example, which creates a vector holding the first 5 terms of n factorial:

```
>> for n=1:5
     nf(n) = factorial(n);
   end
>> disp(nf)
     1     2     6    24   120
```

Note the use of the semicolon on the end of the line in the `for` loop. This prevents MATLAB from printing out the current value of `nf(n)` every time round the loop, which would be rather annoying (try it without the semicolon if you like).

7.4 `while` loops

If you don't know exactly how many repetitions you need, and just want to loop until some condition is satisfied, MATLAB provides a `while` loop:

[8]This does not mean that `for` loops are slow, just that MATLAB is highly optimised for matrix/vector calculations. Furthermore, many modern computers include special instructions to speed up matrix computations, since they are fundamental to the 3D graphics used in computer games. For example, the 'MMX' instructions added to the Intel Pentium processor in 1995, and subsequent processors, are 'Matrix Maths eXtensions'.

```
while expression
    statements
end
```

For example,

```
>> x=1;
>> while 1+x > 1
     x = x/2;
  end
>> x
x =
  1.1102e-016
```

7.5 Accuracy and precision

The above `while` loop continues halving `x` until adding `x` to 1 makes no difference i.e. `x` is zero as far as MATLAB is concerned, and it can be seen that this number is only around 10^{-16}. This does *not* mean that MATLAB can't work with numbers smaller than this (the smallest number MATLAB can represent is about 2.2251×10^{-308}).[9] The problem is that the two numbers in the operation are of different orders of magnitude, and MATLAB can't simultaneously maintain its accuracy in both.

Consider this example:

$$a = 13901 = 1.3901 \times 10^4$$
$$b = 0.0012 = 1.2 \times 10^{-3}$$

If we imagine that the numerical accuracy of the computer is 5 significant figures in the mantissa (the part before the $\times 10^k$), then both a and b can be exactly represented. However, if we try to add the two numbers, we get the following:

$$\begin{aligned} a + b = 13901.0012 &= 1.39010012 \times 10^4 \\ &= 1.3901 \times 10^4 \qquad \text{to 5 significant figures} \end{aligned}$$

So, while the two numbers are fine by themselves, because they are of such different magnitudes their sum cannot be exactly represented.

This is exactly what is happening in the case of our `while` loop above. MATLAB (and most computers) are accurate to about fifteen significant figures, so once we try to add 1×10^{-16} to 1, the answer requires a larger number of significant figures than are available, and the answer is truncated, leaving just 1.

There is no general solution to these kind of problems, but you need to be aware that they exist. It is most unusual to need to be concerned about the sixteenth decimal place

[9] The MATLAB variables `realmax` and `realmin` tell you what the minimum and maximum numbers are on any computer (the maximum on the teaching system computers is about 1.7977×10^{308}). In addition, the variable `eps` holds the 'distance from 1.0 to the next largest floating point number'—a measure of the possible error in any number, usually represented by ϵ in theoretical calculations. On the teaching system computers, `eps` is 2.2204×10^{-16}. In our example, when `x` has this value, `1+x` does make a difference, but then this value is halved and no difference can be found.

of an answer, but if you are then you will need to think very carefully about how you go about solving the problem. The answer is to *think* about how you go about formulating your solution, and make sure that, in the solution you select, the numbers you are dealing with are all of of about the same magnitude.

Part IB Computing Course — Session 2A

Summary

After reading through sections 6 and 7 of the tutorial guide and working through the examples you should be able to:

- ☐ Use loops and conditionals to control program execution
- ☐ Create and use scripts to store a series of commands

Exercise 2: Finding π using arctangent with $z = 1$

A. Theory

The search to determine π to a greater and greater accuracy has occupied mathematicians for millennia. In the IA C++ course you considered two methods for calculating π, one by Euler and one by Archimedes. Archimedes's approach, of inscribing and circumscribing circles, was the method of choice from the 3rd century BC until the 17th century AD. In 1671, James Gregory (1638–1675) found the now-standard power series for arctangent, which can be found in the Mathematics Databook:

$$\tan^{-1} z = z - \frac{z^3}{3} + \frac{z^5}{5} - \cdots$$

Three years later, Gottfried Wilhelm Leibniz (1646–1716) independently found and published the same series, noting a special case: that $\tan \frac{\pi}{4} = 1$. In other words, putting $z = 1$ into the arctangent series, gives an approximation for $\frac{\pi}{4}$, and hence for π. The arctangent series is still the way that π is calculated today.

B. Exercise

Write a MATLAB script to calculate the first 100 terms in the arctangent series and plot a graph showing the value of π estimated after each term. How good is the estimate after 100 terms?

Modify the script to continue to calculate more terms, automatically stopping when the estimate of π is accurate to two decimal places after rounding. How many terms does this require?

C. Implementation notes

1. **Try things out in MATLAB first**
   Remember that a MATLAB script is just a series of commands that you could equally well type in at the command line. Try things out in MATLAB first of all, before adding them to the script.

2. **Read error messages from bottom to top**
   If you get errors, when you run your script, remember that these are best read from bottom to top.

3. **Alternating signs and odd numbers**
   You can get a sequence of alternating signs by using $(-1)^n$, which is $+1$ when n is even, and -1 when n is odd. To produce only odd numbers from a counting sequence involving all positive integers n, you can use $(2n-1)$.

4. **MATLAB functions**
   The MATLAB function `abs` can be used to calculate the absolute value of a number (i.e. ignoring the sign), and should be used to find the absolute error. The function `round` can be used to round a number to the nearest integer.

D. Evaluation and marking

Save the MATLAB commands necessary to create and plot the function in a script. Show this working script, and the plot, to a demonstrator for marking.

Exercise 3: Finding π using arctangent with $z = (1/\sqrt{3})$

A. Theory

In 1699, Abraham Sharp (1651-1742) broke the previous record for the accuracy of π, finding 72 digits (which may not seem many, but is more than the native accuracy of MATLAB). He also used the arctangent series but realised that if, instead of $z = 1$, he used the identity $\tan\frac{\pi}{6} = \frac{1}{\sqrt{3}}$, he would get much quicker convergence.

B. Exercise

Why do you think the series is likely to converge faster using $z = (1/\sqrt{3})$?

Modify your script to estimate π using this new series. Plot the convergence of this series on the same graph as the previous one. How many iterations of this new series are required to find π accurate to two decimal places.

What happens to the convergence of this series in MATLAB for a large number of terms (more than 30)? Why do we see this behaviour?

8 MATLAB programming II: Functions

Scripts in MATLAB let you write simple programs, but more powerful than scripts are user-defined *functions*. These let you define your own MATLAB commands which you can then use either from the command line, or in other functions or scripts.

You will have come across functions in the C++ computing course, where they are used for exactly the same purposes as in MATLAB. However, MATLAB functions are somewhat simpler: variables are always passed by *value*, never by reference. MATLAB functions can, however, return more than one value.[10] Basically, functions in MATLAB are passed numbers, perform some calculations, and give you back some other numbers.

A function is defined in an text file, just like a script, except that the first line of the file has the following form:

```
function [output1,output2,...] = name(input1,input2,...)
```

Each function is stored in a different m-file, which *must have the same name as the function.* For example, a function called `sind()` must be defined in a file called `sind.m`. Each function can accept a range of arguments, and return a number of different values.

Whenever you find yourself using the same set of expressions again and again, this is a sign that they should be wrapped up in a function. As a function, they are easier to use, make the code more readable, and can be used by other people in other situations.

8.1 Example 1: Sine in degrees

MATLAB uses radians for all of its angle calculations, but most of us are happier working in degrees. When doing MATLAB calculations you could just always convert your angle `d` to radians using `sin(d/180*pi)`, or even using the variable `deg` as defined in Section 3.1, writing `sin(d*deg)`. But it would be simpler and more readable if you could just type `sind(d)` ('sine in degrees'), and we can create a function to do this. Such a function would be defined using the following lines:

```
function s = sind(x)
%SIND(X)   Calculates sine(x) in degrees
s = sin(x*pi/180);
```

This may seem trivial, but many functions *are* trivial and it doesn't make them any less useful. We'll look at this function line-by-line:

Line 1 Tells MATLAB that this file defines a function, rather than a script. It says that the function is called `sind`, and that it takes one argument, called `x`. The result of the function is to be known, internally, as `s`. Whatever `s` is set to in this function is what the user will get when they use the `sind` function.

Line 2 Is a comment line. As with scripts, the first set of comments in the file should describe the function. This line is the one printed when the user types `help sind`.

[10]When variables are passed by value, they are *read only*—the values can be read and used, but cannot be altered. When variables are passed by reference (in C++), their values can be altered to pass information back from the function. This is required in C++ because usually only one value can be returned from a function; in MATLAB many values can be returned, so passing by reference is not required.

It is usual to use a similar format to that which is used by MATLAB's built-in functions.

Line 3 Does the actual work in this function. It takes the input `x` and saves the result of the calculation in `s`, which was defined in the first line as the name of the result of the function.

End of the function Functions in MATLAB do not need to end with `return` (although you can use the `return` command to make MATLAB jump out of a function in the middle). Because each function is in a separate m-file, once it reaches the end of the file, MATLAB knows that it is the end of the function. The value that `s` has at the end of this function is the value that is returned.

8.2 Creating and using functions

Create the above function by opening the editor (type `edit` if it's not still open) and typing the lines of the function as given above. Save the file as `sind.m`, since the text file must have the same name as the function, with the addition of `.m`.

You can now use the function in the same way as any of the built-in MATLAB functions. Try typing

```
>> help sind

 SIND(X)   Calculates sine(x) in degrees
```

which shows that the function has been recognised by MATLAB and it has found the help line included in the function definition. Now we can try some numbers

```
>> sind(0)
ans =
     0
>> sind(45)
ans =
    0.7071
>> t = sind([30 60 90])
t =
    0.5000    0.8660    1.0000
```

This last example shows that it also automatically works with vectors. If you call the `sind` function with a vector, it means that the `x` parameter inside the function will be a vector, and in this case the `sin` function knows how to work with vectors, so can give the correct response.

8.3 Example 2: Unit step

Here is a more sophisticated function which generates a unit step, defined as

$$y = \begin{cases} 0 & \text{if } t < t_0 \\ 1 & \text{otherwise} \end{cases}$$

This function will take two parameters: the times for which values are to be generated, and `t0`, the time of the step. The complete function is given below:

```
function y = ustep(t, t0)
%USTEP(t, t0) unit step at t0
%   A unit step is defined as
%         0 for t <  t0
%         1 for t >= t0
[m,n] = size(t);
% Check that this is a vector, not a matrix i.e. (1 x n) or (m x 1)
if m ~= 1 & n ~=1
   error('T must be a vector');
end
y = zeros(m, n); %Initialise output array
for k = 1:length(t)
  if t(k) >= t0
    y(k) = 1;  %Otherwise, leave it at zero, which is correct
  end
end
```

Again, we shall look at this function definition line-by-line:

Line 1 The first line says that this is a function called `ustep`, and that the user must supply two arguments, known internally as `t` and `t0`. The result of the function is one variable, called `y`.

Lines 2-5 Are the description of the function. This time the help message contains several lines.

Line 6 The first argument to the `ystep` function, `t`, will usually be a vector, rather than a scalar, which contains the time values for which the function should be evaluated. This line uses the `size` function, which returns *two* values: the number of rows and then the number of columns of the vector (or matrix). This gives an example of how functions in MATLAB can return two things—in a vector, of course. These values are used to create an output vector of the same size, and to check that the input *is* a vector.

Lines 7-10 Check that the input `t` is valid i.e. that it is not a matrix. This checks that it has either one row or one column (using the result of the `size` function). The `error` function prints out a message and aborts the function if there is a problem.

Line 11 As the associated comment says, this line creates the array to hold the output values. It is initialised to be the same size and shape as the input `t`, and for each element to be zero.

Line 12 For each time value in `t`, we want to create a value for `y`. We therefore use a `for` loop to step through each value. The `length` function tells us how many elements there are in the vector `t`.

Lines 13–15 According to our definition, if $t < t_0$, then the step function has the value zero. Our output vector `y` already contains zeros, so we can ignore this case. In the

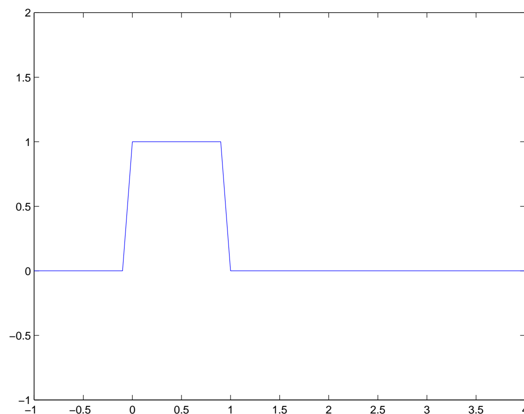


Figure 6: A unit, one second pulse created from two unit steps, using the function `ustep`.

other case, when $t \geq t_0$, the output should be 1. We test for this case and set `y`, our output variable, accordingly.

As in C++, all variables created inside a function (`m`, `n` and `k` in this case) are *local* to the function. They only exist for the duration of the function, and do not overwrite any variables of the same name elsewhere in MATLAB. The only variables which are passed back are the return values defined in the first `function` line: `y` in this case.

Type this function into the editor, and save it as `ustep.m`. We can now use this function to create signal. For example, to create a unit pulse of duration one second, starting at $t = 0$, we can first define a time scale:

```
>> t=-1:0.1:4;
```

and then use `ustep` twice to create the pulse:

```
>> v = ustep(t, 0) - ustep(t, 1);
>> plot(t, v)
>> axis([-1 4 -1 2])
```

This should display the pulse, as shown in Figure 6. If we then type

```
>> who

Your variables are:

a          b          n          t          x
ans        c          nf         v          y
```

we can confirm that the variables `m` and `n` defined in the `ustep` function only lasted as long as the function did, and are not part of the main workspace. Also, the `y` variable still has the values defined earlier by the `rectsin` script, rather than the values defined for the variable `y` in the `ustep` function. Variables defined and used inside functions are completely separate from the main workspace.

Summary

After reading through sections 8 of the tutorial guide and working through the examples you should be able to:

☐ Define and use your own functions

Exercise 4: Building signals from functions

A. Definition

The integral of the unit step function is the *unit ramp* function, defined by

$$y = \begin{cases} 0 & \text{if } t < t_o \\ t - t_0 & \text{otherwise} \end{cases}$$

and shown in Figure 7(a)

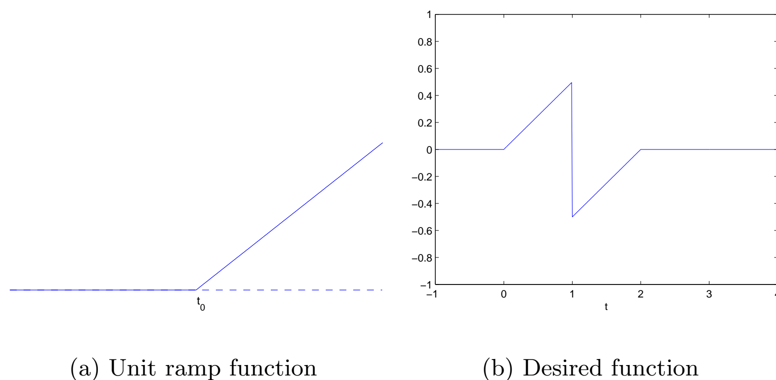
B. Exercise

Modify the `ustep` function (Section 8.3) to create a new MATLAB function, `uramp`, which generates the unit ramp function.

Using `ustep` and `uramp`, what MATLAB expressions are needed to generate the signal shown in Figure 7(b)? Produce your own version of this figure, noting carefully the scale.

C. Evaluation and marking

Save the MATLAB commands necessary to create and plot the function in a script. Show this working script, and the plot, to a demonstrator for marking.



(a) Unit ramp function (b) Desired function

Figure 7: A unit ramp function, and the desired output function

9 Matrices and vectors

Vectors are special cases of *matrices*. A matrix is a rectangular array of numbers, the size of which is usually described as $m \times n$, meaning that it has m rows and n columns. For example, here is a 2×3 matrix:

$$A = \begin{bmatrix} 5 & 7 & 9 \\ -1 & 3 & -2 \end{bmatrix}$$

To enter this matrix into MATLAB you use the same syntax as for vectors, typing it in row by row:

```
>> A=[5 7 9
     -1 3 -2]
A =
     5     7     9
    -1     3    -2
>>
```

alternatively, you can use semicolons to mark the end of rows, as in this example:

```
>> B=[2 0; 0 -1; 1 0]
B =
     2     0
     0    -1
     1     0
```

You can also use the colon notation, as before:

```
>> C = [1:3; 8:-2:4]
C =
     1     2     3
     8     6     4
```

A final alternative is to build up the matrix row-by-row (this is particularly good for building up tables of results in a `for` loop):

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

9.1 Matrix multiplication

With vectors and matrices, the `*` symbol represents matrix multiplication, as in these examples (using the matrices defined above)

```
>> A*B
ans =
    19    -7
    -4    -3
>> B*C
ans =
     2     4     6
    -8    -6    -4
     1     2     3
>> A*C
??? Error using ==> *
Inner matrix dimensions must agree.
```

You might like to work out these examples by hand to remind yourself what is going on. Note that you cannot do `A*C`, because the two matrices are incompatible shapes.[11]

When just dealing with vectors, there is little need to distinguish between row and column vectors. When multiplying vectors, however, it will only work one way round. A row vector is a $1 \times n$ matrix, but this can't post-multiply a $m \times n$ matrix:

```
>> x=[1 0 3]
x =
     1     0     3
>> A*x
??? Error using ==> *
Inner matrix dimensions must agree.
```

9.2 The transpose operator

Transposing a vector changes it from a row to a column vector and vice versa. The transpose of a matrix interchanges the rows with the columns. Mathematically, the transpose of `A` is represented as $\mathtt{A}^T$. In MATLAB an apostrophe performs this operation:

```
>> A
A =
     5     7     9
    -1     3    -2
>> A'
ans =
     5    -1
     7     3
     9    -2
```

[11]In general, in matrix multiplication, the matrix sizes are

$$(l \times m)*(m \times n) \rightarrow (l \times n)$$

When we try to do `A*C`, we are attempting to do $(2 \times 3)*(2 \times 3)$, which does not agree with the definition above. The middle pair of numbers are not the same, which explains the wording of the error message.

```
>> A*x'
ans =
    32
    -7
```

In this last example the `x'` command changes our row vector into a column vector, where it can be pre-multiplied by the matrix `A`.

9.3 Matrix creation functions

MATLAB provides some functions to help you build special matrices. We have already met `ones` and `zeros`, which create matrices of a given size filled with 1 or 0.

A very important matrix is the *identity matrix*. This is the matrix that, when multiplying any other matrix or vector, does not change anything. This is usually called `I` in formulæ, so the MATLAB function is called `eye`. This only takes one parameter, since an identity matrix must be square:

```
>> I = eye(4)
I =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

we can check that this leaves any vector or matrix unchanged:

```
>> I * [5; 8; 2; 0]
ans =
     5
     8
     2
     0
```

The identity matrix is a special case of a *diagonal matrix*, which is zero apart from the diagonal entries:

$$D = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

You could construct this explicitly, but the MATLAB provides the `diag` function which takes a vector and puts it along the diagonal of a matrix:

```
>> diag([-1 7 2])
ans =
    -1     0     0
     0     7     0
     0     0     2
```

The `diag` function is quite sophisticated, since if the function is called for a matrix, rather than a vector, it tells you the diagonal elements of that matrix. For the matrix `A` defined earlier:

```
>> diag(A)
ans =
     5
     3
```

Notice that the matrix does not have to be square for the diagonal elements to be defined, and for non-square matrices it still begins at the top left corner, stopping when it runs out of rows or columns.

9.4 Building composite matrices

It is often useful to be able to build matrices from smaller components, and this can easily be done using the basic matrix creation syntax:

```
>> comp = [eye(3) B;
            A    zeros(2,2)]
comp =
     1     0     0     2     0
     0     1     0     0    -1
     0     0     1     1     0
     5     7     9     0     0
    -1     3    -2     0     0
```

You just have to be careful that each sub-matrix is the right size and shape, so that the final composite matrix is rectangular. Of course, MATLAB will tell you if any of them have the wrong number of row or columns.

9.5 Matrices as tables

Matrices can also be used to simply tabulate data, and can provide a more natural way of storing data:

```
>> t=0:0.2:1;
>> freq=[sin(t)' sin(2*t)', sin(3*t)']
freq =
         0         0         0
    0.1987    0.3894    0.5646
    0.3894    0.7174    0.9320
    0.5646    0.9320    0.9738
    0.7174    0.9996    0.6755
    0.8415    0.9093    0.1411
```

Here the nth column of the matrix contains the (sampled) data for $\sin(nt)$. The alternative would be to store each series in its own vector, each with a different name. You would then need to know what the name of each vector was if you wanted to go on and use the data. Storing it in a matrix makes the data easier to access.

9.6 Extracting bits of matrices

Numbers may be extracted from a matrix using the same syntax as for vectors, using the () brackets. For a matrix, you specify the row co-ordinate and then the column co-ordinate (note that, in Cartesian terms, this is y and then x). Here are some examples:

```
>> J = [
     1     2     3     4
     5     6     7     8
    11    13    18    10];
>> J(1,1)
ans =
     1
>> J(2,3)
ans =
     7
>> J(1:2, 4)   %Rows 1-2, column 4
ans =
     4
     8
>> J(3,:)      %Row 3, all columns
ans =
    11    13    18    10
```

The : operator can be used to specify a range of elements, or if used just by itself then it refers to the entire row or column.

These forms of expressions can also be used on the left-hand side of an expression to write elements into a matrix:

```
>> J(3, 2:3) = [-1 0]
J =
     1     2     3     4
     5     6     7     8
    11    -1     0    10
```

10 Basic matrix functions

MATLAB allows all of the usual arithmetic to be performed on matrices. Matrix multiplication has already been discussed, and the other common operation is to add or subtract two matrices of the same size and shape. This can easily be performed using the + and - operators. As with vectors, MATLAB also defines the .* and ./ operators, which allow the corresponding elements of two matching matrices to be multiplied or divided. All the elements of a matrix may be raised to the same power using the .^ operator.

Unsurprisingly, MATLAB also provides a large number of functions for dealing with matrices, and some of these will be covered later in this tutorial. Try typing

```
help matfun
```

for a taster. For the moment we'll consider some of the fundamental matrix functions.

The `size` function will tell you the dimensions of a matrix. This is a function which returns a vector, specifying the number of rows, and then columns:

```
>> size(J)
ans =
     3     4
```

The inverse of a matrix is the matrix which, when multiplied by the original matrix, gives the identity ($\mathtt{AA}^{-1} = \mathtt{A}^{-1}\mathtt{A} = \mathtt{I}$). It 'undoes' the effect of the original matrix. It is only defined for square matrices, and in MATLAB can be found using the `inv` function:

```
>> A = [
     3     0     4
     0     1    -1
     2     1    -3];
>> inv(A)
ans =
    0.1429   -0.2857    0.2857
    0.1429    1.2143   -0.2143
    0.1429    0.2143   -0.2143
>> A*inv(A)  %Check the answer
ans =
    1.0000    0.0000   -0.0000
         0    1.0000         0
         0    0.0000    1.0000
```

Again, note the few numerical errors which have crept in, which stops MATLAB from recognising some of the elements as exactly one or zero.

The *determinant* of a matrix is a very useful quantity to calculate. In particular, a zero determinant implies that a matrix does not have an inverse. The `det` function calculates the determinant:

```
>> det(A)
ans =
   -14
```

Summary

After reading through sections 9 and 10 of the tutorial guide and working through the examples you should be able to:

- ☐ Define matrices in MATLAB
- ☐ Use matrices and vectors in simple calculations
- ☐ Perform standard operations on matrices

Exercise 5: Euler angles

A. Theory

A rotation matrix in two dimensions is easy to define. There is only one axis of rotation, which is normal to the 2D plane, and a rotation of a positive angle θ (in a right-handed co-ordinate system) is given by the matrix

$$\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}$$

Defining rotations in three dimensions is more difficult. One method of defining rotations is known as *Euler angles*, which are often used in mechanics.[12]

The method of Euler angles, specifies three angles. These are commonly called azimuth, elevation and roll (or in aeronautical terms yaw, pitch and roll). We shall label these three angles as ψ, θ and ϕ, and our object exists in a normal right-handed co-ordinate set, x, y and z.

The rotations must be applied in a specific order. A point (x, y, z) is first rotated from its original location by an angle ψ about the x axis (roll):

$$\begin{pmatrix}x'\\ y'\\ z'\end{pmatrix} = \begin{bmatrix}1 & 0 & 0\\ 0 & \cos\psi & -\sin\psi\\ 0 & \sin\psi & \cos\psi\end{bmatrix}\begin{pmatrix}x\\ y\\ z\end{pmatrix}$$

Then it is rotated by an angle θ about the original y axis (elevation):

$$\begin{pmatrix}x''\\ y''\\ z''\end{pmatrix} = \begin{bmatrix}\cos\theta & 0 & -\sin\theta\\ 0 & 1 & 0\\ \sin\theta & 0 & \cos\theta\end{bmatrix}\begin{pmatrix}x'\\ y'\\ z'\end{pmatrix}$$

The final rotation, ϕ is then about the original z axis (azimuth), defining the new location x''', y''' and z''' as

$$\begin{pmatrix}x'''\\ y'''\\ z'''\end{pmatrix} = \begin{bmatrix}\cos\phi & -\sin\phi & 0\\ \sin\phi & \cos\phi & 0\\ 0 & 0 & 1\end{bmatrix}\begin{pmatrix}x''\\ y''\\ z''\end{pmatrix}$$

[12] Named after the same Leonhard Euler (1707–1783) responsible for the method of finding π that was considered in the IA C++ computing course.

B. Computing exercise

Write a MATLAB that function which takes three Euler angles and produces a single 3D rotation matrix, transforming points (x, y, z) to (x''', y''', z'''), as defined above.

Use this function to find the rotation matrix `R` which rotates a co-ordinate system by the Euler angles $(90°, 20°, 15°)$. Verify that this is a valid rotation matrix i.e. that it is orthogonal, and has a determinant of $+1$.

The point $\mathbf{p}$ is to be mapped to a new location $\mathbf{p'}$ using your matrix `R`. If $\mathbf{p} = (2, 3, 0)^T$, what is $\mathbf{p'}$? What matrix reverses this transformation?

Find the matrix `S` which represents the rotation given by $(-90°, -20°, -15°)$. To where does your transformed point $\mathbf{p'}$ map under this new rotation? Why is `S` is not the reverse of `R`?

C. Implementation notes

1. **Degrees and radians**
   Remember that MATLAB works in radians, not degrees. You might like to make use of the `sind` function you defined earlier, and also to define a `cosd` function.

D. Evaluation and marking

Write a MATLAB script which uses your Euler angle function to perform all the calculations necessary for the exercise. Show this working script, and your Euler angle function to a demonstrator for marking.

11 Solving $\mathtt{A}\mathbf{x} = \mathbf{b}$

One of the most important use of matrices is for representing and solving simultaneous equations. A system of linear equations is

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots &= \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

where the a_{ij} and b_i are known, and we are looking for a set of values x_i that simultaneously satisfy all the equations. This can be written in matrix-vector form as

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

or

$$\mathtt{A}\mathbf{x} = \mathbf{b}$$

In this representation, `A` is the matrix of coefficients, $\mathbf{b}$ are the constants, and $\mathbf{x}$ is the vector of parameters that we want to find.

Since MATLAB is designed to process matrices and vectors, it is particularly appropriate to use it to solve these forms of problems.

11.1 Solution when `A` is invertible

The simplest set of linear equations to solve occur when you have both n equations and n unknowns. In this case the matrix `A` will be square and can often be inverted. Consider this simple example:

$$\begin{aligned} x + y &= 3 \\ 2x - 3y &= 5 \end{aligned}$$

Solving this in MATLAB is a case of turning the equations into matrix-vector form and then using the inverse of `A` to find the solution:

```
>> A=[1  1
      2 -3];
>> b=[3 5]';
>> inv(A)*b
ans =
    2.8000
    0.2000
>> A*ans    %Just to check
ans =
    3.0000
    5.0000
```

So the solution is $x = 2.8$, $y = 0.2$.

11.2 Gaussian elimination and LU factorisation

Calculating the inverse of a matrix is an inefficient method of solving these problems, even if MATLAB can still invert large matrices a lot more quickly than you could do by hand. From the IB Linear Algebra course, you should be familiar with Gaussian elimination, and LU factorisation (which is just Gaussian elimination in matrix form).[13] These provide a more efficient way of solving $\mathtt{A}\mathbf{x} = \mathbf{b}$, and MATLAB makes it very easy to use Gaussian elimination by defining this as the meaning of *matrix division* for invertible matrices.

11.3 Matrix division and the slash operator

In a normal algebraic equation, $ax = b$, if you want to find the value of x you would just calculate $x = b/a$. In a matrix-vector equation $\mathtt{A}\mathbf{x} = \mathbf{b}$, however, division is not defined and the solution is given by $\mathbf{x} = \mathtt{A}^{-1}\mathbf{b}$. As a shortcut for this, MATLAB defines a special operator \ (note that this is a backslash, not the divide symbol), and can be thought of as 'matrix division'. Using this, the solution to our earlier equation may be calculated as:

```
>> A\b
ans =
    2.8000
    0.2000
```

Note that this is *not a standard notation* and in written mathematical expressions you should still always write $\mathtt{A}^{-1}\mathbf{b}$.

You should recall that matrix multiplication is not commutative i.e. $\mathtt{AB} \neq \mathtt{BA}$. This means that, while the solution to $\mathtt{AX} = \mathtt{B}$ is given by $\mathtt{X} = \mathtt{A}^{-1}\mathtt{B}$, the solution to $\mathtt{XA} = \mathtt{B}$ is $\mathtt{X} = \mathtt{BA}^{-1}$, and these two expressions are different. As a result, MATLAB also defines the / (forward slash) operator which performs this other variety of matrix division. The former case, however, is more likely, and it is usually the backslash that you will need to use. The two slash operators are summarised in Table 5.

operator	to solve	MATLAB command	mathematical equivalent	name
\	$\mathtt{AX} = \mathtt{B}$	`A\B`	$\mathtt{A}^{-1}\mathtt{B}$	left division (backslash)
/	$\mathtt{XA} = \mathtt{B}$	`B/A`	$\mathtt{BA}^{-1}$	right division (forward slash)

Table 5: Summary of MATLAB's slash operators. These use Gaussian elimination if the matrix is invertible, and finds the least squares solution otherwise.

11.4 Singular matrices and `rank`

The slash operator tries to do its best whatever matrix and vector it is given, even if a solution is not possible, or is undetermined. Consider this example:

$$\begin{aligned} u + v + w &= 2 \\ 2u + 3w &= 5 \\ 3u + v + 4w &= 6 \end{aligned}$$

[13] The `lu` function in MATLAB will perform the LU factorisation of a matrix for you (see the help message for that function).

This is again an equation of the form $\mathtt{A}\mathbf{x} = \mathbf{b}$. If you try to solve this in MATLAB using the slash operator, you get

```
>> A=[1 1 1
      2 0 3
      3 1 4];
>> b=[2 5 6]';
>> x=A\b
x =
   1.0e+015 *
    -5.4043
     1.8014
     3.6029
>> A*x   %Should be the same as b
ans =
     2
     6
     6
```

So the solution given is not a solution to the equation! You should perhaps have been suspicious, anyway, from the size of the numbers in `x`. What has gone wrong?

In this case the matrix `A` is *singular*. This means that it has a zero determinant, and so is non-invertible. We can check this by typing

```
>> det(A)
ans =
     0
```

When the matrix is singular it means that there is not a unique solution to the equations. But we have three equations and three unknowns, so why is there not a unique solution? The problem is that there are *not* three unique equations. The `rank` function in MATLAB estimates the rank of a matrix—the number linearly of independent rows or columns in the matrix:

```
>> rank(A)
ans =
     2
```

So there are only two independent equations in this case. Looking at the original equations more carefully, we can see that the first two add to give $3u + v + 4w = 7$, which is clearly inconsistent with the third equation. In other words, there is no solution to this equation. What has happened in this case is that, thanks to rounding errors in the Gaussian elimination, MATLAB has found an incorrect solution.

The moral of this section is that the slash operator in MATLAB is very powerful and useful, but you should not use it indiscriminately. You should check your matrices to ensure that they are not singular, or near-singular.[14]

[14]Some singular matrices can have an infinity of solutions, rather than no solution. In this case, the slash operator gives just one of the possible values. This again is something that you need to be aware of, and watch out for.

11.5 Ill-conditioning

Matrices which are near-singular are an example of *ill-conditioned* matrices. A problem is ill-conditioned if small changes in the data produce large changes in the results. Consider this system:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1.01 \end{bmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 2.01 \end{pmatrix}$$

for which MATLAB displays the correct answer:

```
>> M=[1 1; 1 1.01]; b=[2; 2.01];
>> x=M\b
x =
    1.0000
    1.0000
```

Let us now change one of the elements of M by a small amount, and see what happens:

```
>> M(1,2) = 1.005;
>> M\b
ans =
   -0.0100
    2.0000
```

Changing this one element by 0.5% has decreased `X(1)` by approximately 101%, and increased `X(2)` by 100%!

The sensitivity of a matrix is estimated using the *condition number*, the precise definition of which is beyond the scope of this tutorial. However, the larger the condition number, the more sensitive the solution is. In MATLAB, the condition number can be found by the `cond` function:

```
>> cond(M)
ans =
  402.0075
```

A rule of thumb is that if you write the condition number in exponential notation $a \times 10^k$, then you last k significant figures of the result should be ignored. MATLAB works to about fifteen significant figures, so the number of significant figures that you should believe is $(15 - k)$. In this example, the condition number is 4×10^2, so all of the the last 2 decimal places of the solution should be dropped i.e. the result is perfectly valid.[15] For the singular matrix `A` from earlier, which gave the spurious result, `cond` gives a condition number of 2.176×10^{16}. In other words, all of the result should be ignored![16]

[15] This assumes that the values entered into the original matrix `M` and vector **b** were accurate to fifteen significant figures. If those values were known to a lesser accuracy, then you lose k significant figures from *that* accuracy.

[16] The condition number for this singular matrix again shows one of the problems of numerical computations. All singular matrices should have a condition number of infinity.

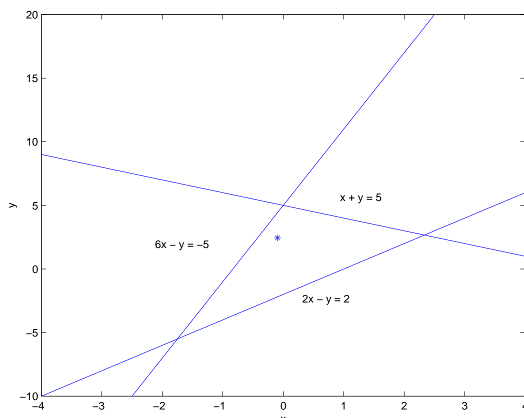


Figure 8: Three lines and the least squares estimate of the common intersection point

11.6 Over-determined systems: Least squares

A common experimental scenario is when you wish to fit a model to a large number of data points. Each data point can be expressed in terms of the model's equation, giving a set of simultaneous equations which are *over-determined* i.e. the number of independent equations, m, is larger than the number of variables, n. In these cases it is not possible to find an *exact* solution, but the desired solution is the 'best fit'. This is usually defined as the model parameters which minimise the squared error to each data point.

For a set of equations which can be written as $\mathtt{A}\mathbf{x} = \mathbf{b}$, the least-squares solution is given by the *pseudoinverse* (see your IB Linear Algebra notes):

$$\mathbf{x} = \left(\mathtt{A}^T\mathtt{A}\right)^{-1}\mathtt{A}^T\mathbf{b}$$

In MATLAB you can again use the \ operator. For invertible matrices it is defined to solve the equation using Gaussian elimination, but for over-determined systems it finds the least squares solution. The pseudoinverse can also be calculated in MATLAB using the `pinv` function, as demonstrated in the following example.

11.7 Example: Triangulation

A hand-held radio transmitter is detected from three different base stations, each of which provide a reading of the direction to the transmitter. The vector from each base station is plotted on a map, and they should all meet at the location of the transmitter. The three lines on the map can be described by the equations

$$\begin{aligned} 2x - y &= 2 \\ x + y &= 5 \\ 6x - y &= -5 \end{aligned}$$

and these are plotted in Figure 8. However, because of various errors, there is not one common point at which they meet.

The least squares solution to the intersection point can be calculated in MATLAB in a number of different ways once we have defined our matrix equation:

```
>> A=[2 -1; 1  1; 6 -1];
>> b = [2 5 -5]';
>> x = inv(A'*A)*A'*b
x =
   -0.0946
    2.4459
>> x=pinv(A)*b
x =
   -0.0946
    2.4459
>> x = A\b
x =
   -0.0946
    2.4459
```

All of these, of course, give the same solution. This least squares solution is marked on Figure 8 with a '*'.

12 More graphs

MATLAB provides more sophisticated graphing capabilities than simply plotting 2D Cartesian graphs. It can also produce histograms, 3D surfaces, contour plots and polar plots, to name a few. Details of all of these can be found in the `help` system (under the subjects `graph2d` and `graph3d`), and only a few examples will be considered here.

12.1 Putting several graphs in one window

If you have several graphs sharing a similar theme, you can plot them on separate graphs within the same graphics window. The `subplot` command splits the graphics window into an array of smaller windows. The general format is

`subplot(`*`rows`*`,`*`columns`*`,`*`select`*`)`

The *`select`* argument specifies the current graph in the array. These are numbered from the top left, working along the rows first. The example below creates two graphs, one on top of the other, as shown in Figure 9.

```
>> x = linspace(-10, 10);
>> subplot(2,1,1)  % Specify two rows, one column, and select
>>                 % the top one as the current graph
>> plot(x, sin(x));
>> subplot(2,1,2);
>> plot(x, sin(x)./x));
```

The standard axes labelling and title commands can also be used in each `subplot`.

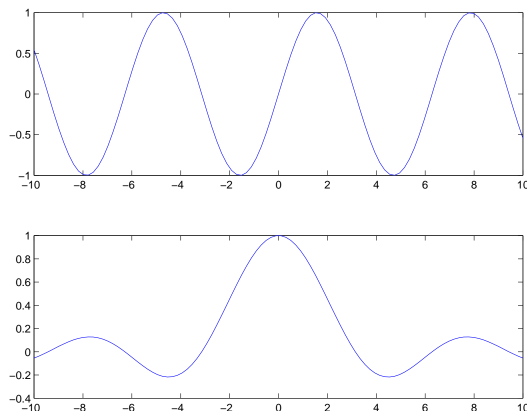


Figure 9: Example of the `subplot` command, showing a sine and sinc curve one above the other.

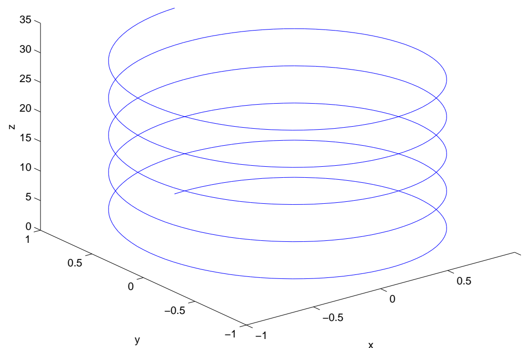


Figure 10: Example of the `plot3` command, showing a helix.

12.2 3D plots

MATLAB provides a wide range of methods for visualising 3D data. The simplest form of 3D graph is `plot3`, which is the 3D equivalent of the `plot` command. Used for plotting points or drawing lines, this simply takes a list of x, y and z values. The following example plots a helix, using a parametric representation.[17]

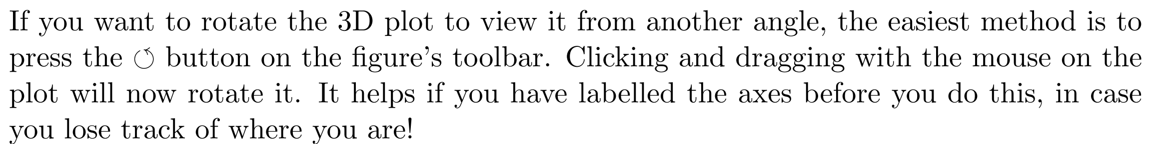
```
>> t = 0:pi/50:10*pi;
>> x = sin(t);  y = cos(t); z = t;
>> plot3(x, y, z);
```

The graph is shown in Figure 10. The `xlabel` and `ylabel` commands work as before, and now the `zlabel` command can also be used. When plotting lines or points in 3D you can use all of the styles listed in Table 3.

[17] A line is only a one dimensional shape—you can define a point along its length, but it has no width or height. Lines can therefore be described in terms of only one parameter: the distance along the line. The x, y (and in 3D z) co-ordinates of points on the line can all be written as functions of this distance along the line: 'if I am this far along the line, I must be at this point.'

12.3 Changing the viewpoint

If you want to rotate the 3D plot to view it from another angle, the easiest method is to press the ⟳ button on the figure's toolbar. Clicking and dragging with the mouse on the plot will now rotate it. It helps if you have labelled the axes before you do this, in case you lose track of where you are!



The MATLAB command `view` can be used to select a particular viewpoint. Look it up on the help system for more information.

12.4 Plotting surfaces

Another common 3D graphing requirement is to plot a surface, defined as a function of two variables $f(x, y)$, and these can be plotted in a number of different ways in MATLAB. First, though, in order to plot these functions we need a grid of points at which the function is to be evaluated. Such a grid can be created using the `meshgrid` function:

```
>> x = 2:0.2:4;  % Define the x- and y- coordinates
>> y = 1:0.2:3;  %  of the grid lines
>> [X,Y] = meshgrid(x, y);   %Make the grid
```

The matrices `X` and `Y` then contain the x and y coordinates of the sample points. The function can then be evaluated at each of these points. For example, to plot

$$f(x, y) = (x - 3)^2 - (y - 2)^2$$

over the grid calculated earlier, you would type

```
>> Z=(X-3).^2 - (Y-2).^2;
>> surf(X,Y,Z)
```

`surf` is only one of the possible ways of visualising a surface. Figure 11 shows, in the top left, this surface plotted using the `surf` command, and also the results of some of the other possible 3D plotting commands.

12.5 Images and Movies

An image is simply a matrix of numbers, where each number represents the colour or intensity of that pixel in the image. It should be no surprise, therefore, that MATLAB can also perform image processing. There are several example images supplied with MATLAB, and the following commands load and display one of them:

```
>> load gatlin
>> colormap(gray(64))  % Tell MATLAB to expect a greyscale image
>> image(X)
```

The file `gatlin.mat` contains the matrix `X`. If you look at the elements of this matrix, you will see that it simply contains numbers between 0 and 63, which represent the brightness of each pixel (0 is black and 63 is white). The `image` command displays this matrix as an image; the `colormap` command tells MATLAB to interpret each number as a shade of grey, and what the range of the numbers is.[18]

[18] Try `colormap(jet)` for an example of an alternative interpretation of the numbers (this is the default colormap). The colormap is also used to determine the colours for the height in the 3D plotting commands.

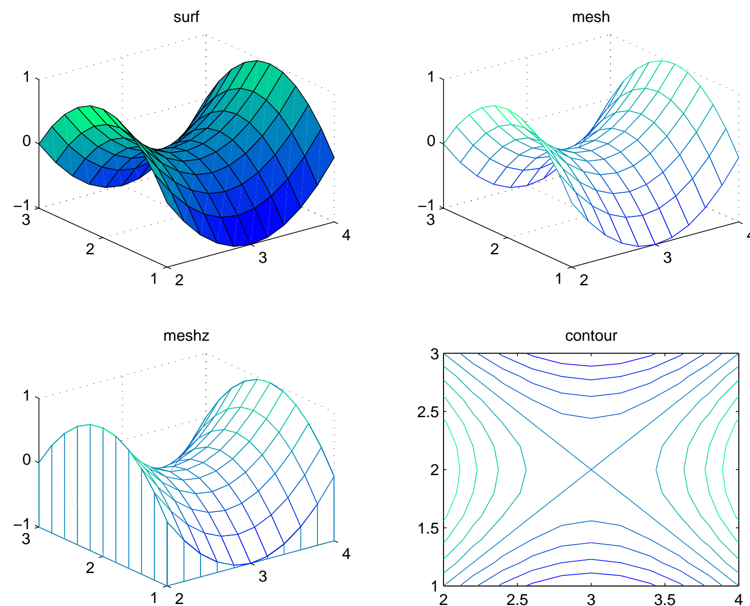


Figure 11: Examples of the same saddle-shaped surface visualised using different MATLAB commands. (The 4 graphs are plotted in the same figure using the command `subplot(2,2,i)`.)

MATLAB can also create and display movies. These are stored in a special format, but you can turn the current figure into a movie frame by using

```
M(frame_no) = getframe;
```

or movie frames can also be created directly from an image matrix:

```
M(frame_no) = im2frame(im, colormap_name);
```

Once you have built up a movie, it can be played using the `movie` command:

```
movie(M)
```

The optional Exercise 7 in this tutorial will consider images and movies.

Part IB Computing Course Session 3B

Summary

After reading through Sections 11 and 12 of the tutorial guide and working through the examples you should be able to:

- ☐ Write linear simultaneous equations in matrix-vector form
- ☐ Use the slash operator in MATLAB to solve `Ax` $= \mathbf{b}$
- ☐ Understand what the slash operator does in different scenarios
- ☐ Appreciate the problems of ill-conditioned matrices
- ☐ Produce more advanced figures, including displaying three-dimensional data

Exercise 6: Leg Before Wicket?

A. Description

The Leg Before Wicket (LBW) rule in the game of cricket is the cause of many contentious decisions, but the problem is a very simple one: if the batsman had not got in the way, would the ball have hit the stumps? This is an extrapolation problem. We can track the ball's position from the point it bounces, until it hits the batsman, and the problem is then just one of fitting a curve to the observed trajectory and extrapolating this to see if it hits the stumps.

A tracking system measures the ball's position at a rate of 150Hz, and for each sample stores the x, y and z co-ordinate of the ball. Measurements are in metres, according to the axes defined in Figure 12, centred on the bottom of the middle stump. The ball's position is accurate to about 2cm.

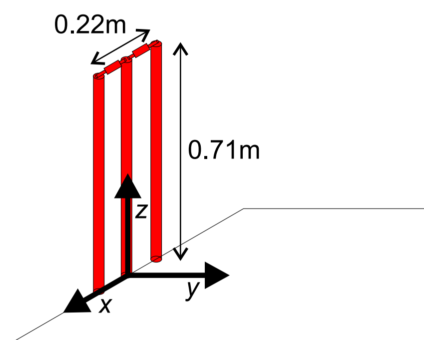


Figure 12: Definition of axes, and dimensions of the stumps

B. Exercise

The file `cricket.txt` contains data from a disputed decision, to be analysed by this system. The file contains 4 columns of data, in the format (*time x y z*). Load this file into MATLAB and produce a three dimensional plot of the position data, marking each sample with a blob.

Assuming a linear horizontal trajectory (i.e. that the ball's velocity is constant in x and y), and that its height describes a parabola, express each of the three co-ordinates as a function of time. Use least squares to estimate the motion parameters and so fit a curve to the sample points. *You should not assume you know the value of the acceleration due to gravity, g, and you should instead include this as one of your unknown parameters.* What value for g is given by this model? Add your estimated curve to the 3D plot, extrapolating it past the stumps.

Fit another curve by least squares, this time assuming that $g = 9.81$, and add this to your plot in a different colour. Do you notice a significant difference? Using this model of the ball's trajectory, what was the velocity of the ball when $t = 0$? Finally, calculate the ball's position as it passes the stumps. Would the ball have hit the stumps?

C. Implementation notes

1. **Solving equations of motion by least squares**

   For each sample point you can write three equations in terms of time: one for x, one for y, and one for z. The first two are linear, in the form

   $$x = x_0 + u_x t$$

   And similarly for y. Each of these equations has two parameters, the speeds u_x and u_y, and the initial positions x_0 and y_0. The z motion will have an additional quadratic term involving g:

   $$z = z_0 + u_z t + \frac{1}{2} g t^2$$

   If g is unknown (as in the first part of this exercise), then this has three unknown parameters, giving seven parameters in total. The three equations may be written in matrix-vector form as

   $$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \left[\quad \mathtt{A} \quad \right] \begin{pmatrix} u_x \\ u_y \\ u_z \\ x_0 \\ y_0 \\ z_0 \\ g \end{pmatrix}$$

   where the elements in `A` are given by a straightforward comparison of coefficients with the three motion equations.

   Each sample point gives different values of x, y and z, and so three different equations. Each of these equations gives different values for the matrix `A`, but the parameters u_x etc. *stay exactly the same* (these are our initial conditions, and define the

fitted trajectory). We can therefore stack all of our equations on top of each other to create some huge matrices and vectors:

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ \hline x_2 \\ y_2 \\ z_2 \\ \hline \vdots \\ \hline x_n \\ y_n \\ z_n \end{pmatrix} = \begin{bmatrix} \\ \mathtt{A_1} \\ \\ \hline \\ \mathtt{A_2} \\ \\ \hline \vdots \\ \hline \\ \mathtt{A_n} \\ \\ \end{bmatrix} \begin{pmatrix} u_x \\ u_y \\ u_z \\ x_0 \\ y_0 \\ z_0 \\ g \end{pmatrix}$$

This then an over-determined equation of the form $\mathtt{A}\mathbf{x} = \mathbf{b}$, and can be solved in MATLAB with the slash operator or the pseudoinverse.

In the second part, when g is known, there are only six unknown parameters, and so your parameter vector will be

$$\begin{pmatrix} u_x & u_y & u_z & x_0 & y_0 & z_0 \end{pmatrix}^T$$

You will therefore need a different matrix `A` and vector **b**.

2. **Building the matrix `A`**
   Each data point from the text file contributes three equations, and so three lines to both `A` and **b**. The best way to build this matrix and vector is with a `for` loop which steps through the rows of the raw data and, for each row, fills in three rows of `A` and **b**. A row i from the text file ($i = 1 \ldots n$) provides information for rows $3(i-1)+1$, $3(i-1)+2$ and $3(i-1)+3$ of this matrix and vector.

3. **The position of the stumps**
   The dimensions of the stumps are given in Figure 12. In addition, the script `stumps.m` will draw a representation of the stumps in a 3D figure window.

D. Evaluation and marking

Write a MATLAB script that contains all the commands necessary for this exercise, so that it displays the required numbers and plots the graphs. Use an `if` statement to print out a message saying whether the ball would have hit the stumps. Show this working script to a demonstrator for marking.

Once you have finished this exercise and had it marked, you have qualified for your full twelve marks. This exercise must be marked by the end of your fourth session. You should now read through the rest of this tutorial and, if you have time left, should attempt Exercise 7.

13 Eigenvectors and the Singular Value Decomposition

After the solution of $\mathtt{A}\mathbf{x} = \mathbf{b}$, the other important matrix equation is $\mathtt{A}\mathbf{x} = \lambda\mathbf{x}$, the solutions to which are the *eigenvectors* and *eigenvalues* of the matrix A. Equations of this form crop up often in engineering applications, and MATLAB again provides the tools to solve them.

13.1 The eig function

The eig function in MATLAB calculates eigenvectors and eigenvalues. If used by itself, it just provides a vector containing the eigenvalues, as in this example:

```
>> A=[1 3 -2
      3 5  1
     -2 1  4];
>> eig(A)
ans =
    6.6076
    4.9045
   -1.5120
```

In order to obtain the eigenvectors, you need to provide two variables for the answer:

```
>> [V,D]=eig(A)
V =
    0.4804    0.3153    0.8184
    0.8764   -0.2071   -0.4347
   -0.0324   -0.9261    0.3758
D =
    6.6076         0         0
         0    4.9045         0
         0         0   -1.5120
```

The columns of the matrix V are the eigenvectors, and the eigenvalues are this time returned in a diagonal matrix. The eigenvectors and eigenvalues are returned in this format because this is then consistent with the *diagonal form* of the matrix. You should recall that, if U is the matrix of eigenvectors and Λ the diagonal matrix of eigenvalues, then $\mathtt{A} = \mathtt{U}\Lambda\mathtt{U}^{-1}$. We can easily check this using the matrices returned by eig:

```
>> V*D*inv(V)
ans =
    1.0000    3.0000   -2.0000
    3.0000    5.0000    1.0000
   -2.0000    1.0000    4.0000
```

which is the original matrix A.

13.2 The Singular Value Decomposition

Eigenvectors and eigenvalues can only be found for a square matrix, and thus the decomposition into $\mathtt{U}\Lambda\mathtt{U}^{-1}$ is only possible for these matrices. The Singular Value Decomposition

(SVD) is a very useful technique, which provides a similar decomposition for any matrix. The specifics are covered in the IB Linear Algebra course, but an outline of the outcome of the decomposition is given here.

The SVD takes an $m \times n$ matrix $\mathtt{A}$ and factors it into $\mathtt{A} = \mathtt{Q}_1 \Sigma \mathtt{Q}_2^T$, where

- $\mathtt{Q}_1$ is an $m \times m$ orthogonal matrix whose columns are the eigenvectors of $\mathtt{AA}^T$
- $\mathtt{Q}_2$ is an $n \times n$ orthogonal matrix whose columns are the eigenvectors of $\mathtt{A}^T\mathtt{A}$
- Σ is an $m \times n$ diagonal matrix whose elements are the square roots of the eigenvalues of $\mathtt{AA}^T$ and $\mathtt{A}^T\mathtt{A}$ (both have the same eigenvalues, but different eigenvectors)

This is better shown graphically:

$$\overset{\mathtt{A}}{\begin{bmatrix} & & & & \cdot \\ & & & & m \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix}} = \overset{\mathtt{Q}_1}{\begin{bmatrix} & & \cdot \\ & & m \\ \cdot & m & \cdot \end{bmatrix}} \overset{\Sigma}{\begin{bmatrix} & & & & \cdot \\ & & & & m \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix}} \overset{\mathtt{Q}_2^T}{\begin{bmatrix} & & & & \cdot \\ & & & & \cdot \\ & & & & n \\ & & & & \cdot \\ \cdot & \cdot & n & \cdot & \cdot \end{bmatrix}}$$

The most important part of this decomposition to consider is the matrix Σ, the diagonal matrix, whose elements are called the *singular values*, σ_i:

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & & & \\ & \sigma_2 & & & & & \\ & & \ddots & & & & \mathbf{0} \\ & & & \sigma_r & & & \\ & & & & 0 & & \\ & \mathbf{0} & & & & \ddots & \\ & & & & & & 0 \end{bmatrix}$$

Being part of a diagonal matrix, these only multiply particular columns of $\mathtt{Q}_1$ or $\mathtt{Q}_2$.[19] The size of the singular value tells you exactly how much influence the corresponding rows and columns of $\mathtt{Q}_1$ and $\mathtt{Q}_2^T$ have over the original matrix. Clearly, if a singular value is tiny, very little of the corresponding rows and columns get added into the matrix $\mathtt{A}$ when it is reconstituted. It is quite common for a matrix to have some zero eigenvalues, as shown above. These zero elements on the diagonal indicate correspond to rows or columns that give *no* extra information. This indicates, of course, that not all of the matrix is useful—that it is *rank deficient*. Or, in terms of simultaneous equations, that not all the equations are independent.

The SVD is a very useful tool for analysing a matrix, and the solutions to many problems can simply be read off from the decomposed matrices. For example, the nullspace of the matrix (the solution to $\mathtt{A}\mathbf{x} = \mathbf{0}$) is simply the columns of $\mathtt{Q}_2$ which correspond to the zero singular values. The SVD is also highly numerically stable—matrices which are slightly ill-conditioned stand more chance of working with SVD than with anything else.

[19] The singular values Σ of course multiply the *rows* of $\mathtt{Q}_2^T$, and hence the columns of the un-transposed version, $\mathtt{Q}_2$.

13.3 Approximating matrices: Changing rank

Another important use of the SVD is the insight it gives into how much of a matrix is important, and how to simplify a matrix with minimal disruption. The singular values are usually arranged in order of size, with the first, σ_1, being the largest and most significant. The corresponding columns of Q_1 and Q_2 are therefore also arranged in importance. What this means is that, while we can find the exact value of A by multiplying $Q_1 \Sigma Q_2^T$, if we removed (for example) the last columns of Q_1 and Q_2, and the final singular value, we would be removing the least important data. If we then multiplied these simpler matrices, we would only get an approximation to A, but one which still contained all but the most insignificant information. Exercise 7 will consider this in more detail.

13.4 The svd function

The SVD is performed in MATLAB using the `svd` function. As with the `eig` function, by itself it only returns the singular values, but if given three matrices for the answer then it will fill them in with the full decomposition:

```
>> A =[ 1  3 -2  3
        3  5  1  5
       -2  1  4  2];
>> svd(A)
ans =
    8.9310
    5.0412
    1.6801
>> [U,S,V] = svd(A,0)
U =
    0.4673   -0.3864   -0.7952
    0.8621   -0.0003    0.5068
    0.1961    0.9223   -0.3329
S =
    8.9310         0         0         0
         0    5.0412         0         0
         0         0    1.6801         0
V =
    0.2980   -0.4428    0.8280   -0.1719
    0.6616   -0.0473   -0.1097    0.7403
    0.0797    0.8851    0.4556    0.0529
    0.6835    0.1356   -0.3079   -0.6478
>>
>> U*S*V'  %Check the answer
ans =
    1.0000    3.0000   -2.0000    3.0000
    3.0000    5.0000    1.0000    5.0000
   -2.0000    1.0000    4.0000    2.0000
```

Note that MATLAB automatically orders the singular values in decreasing order.

13.5 Economy SVD

If a $m \times n$ matrix `A` is overdetermined (i.e. $m > n$), then the matrix of singular values, `Σ`, will have at least one row of zeros:

```
>> A=[1 2
      3 4
      5 6];
>> [U,S,V] = svd(A)
U =
    0.2298    0.8835    0.4082
    0.5247    0.2408   -0.8165
    0.8196   -0.4019    0.4082
S =
    9.5255         0
         0    0.5143
         0         0
V =
    0.6196   -0.7849
    0.7849    0.6196
```

If we then consider the multiplication `A = U*S*V'`, we can see that the last column of `U` serves no useful purpose—it multiplies the zeros in `S`, and so might as well not be there. Therefore this last column may be safely ignored, and the same is true of the last row of `S`. What we are then left with is matrices of the following form:

$$
\overset{\texttt{A}}{\begin{bmatrix} & \cdot \\ & \cdot \\ & m \\ & \cdot \\ \cdot \;\; n & \cdot \end{bmatrix}} = \overset{\texttt{Q}_1}{\begin{bmatrix} & \cdot \\ & \cdot \\ & m \\ & \cdot \\ \cdot \;\; n & \cdot \end{bmatrix}} \overset{\Sigma}{\begin{bmatrix} & \cdot \\ & n \\ \cdot \;\; n & \cdot \end{bmatrix}} \overset{\texttt{Q}_2^T}{\begin{bmatrix} & \cdot \\ & n \\ \cdot \;\; n & \cdot \end{bmatrix}}
$$

MATLAB can be asked to perform this 'economy' SVD by adding '`,0`' to the function argument (NB this is a zero, not the letter 'O'):

```
>> [U,S,V] = svd(A,0)
U =
    0.2298    0.8835
    0.5247    0.2408
    0.8196   -0.4019
S =
    9.5255         0
         0    0.5143
V =
    0.6196   -0.7849
    0.7849    0.6196
```

This is highly recommended for matrices of this shape, since it takes far less memory and time to process.

Part IB Computing Course   Session 4

Summary

After reading through Sections 13 of the tutorial guide and working through the examples you should be able to:

- ☐ Calculate eigenvectors and eigenvalues using MATLAB
- ☐ Have an understanding of what the Singular Value Decomposition (SVD) tells you about a matrix
- ☐ Be able to perform the SVD in MATLAB

Exercise 7: Video textures (optional)

A. Theory

A video frame contains an enormous amount of data—for example a small video frame of 100×100 pixels still contains 10,000 elements of information. However, in a typical video of an object, one needs nowhere near this many pieces of information to talk about what's going on. Figure 13 shows part of a video of a fountain. If we wanted to describe this fountain to someone else, we wouldn't describe every pixel. We would say 'it's a small fountain', and could perhaps describe each frame by saying how strong the fountain was at that instant.

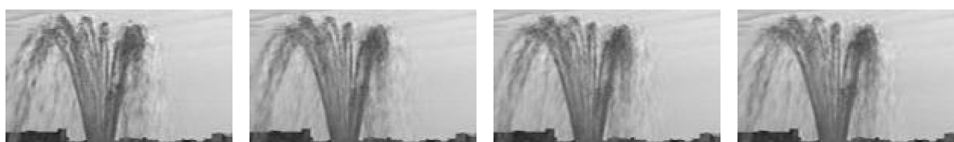


Figure 13: The first four frames of the 'fountain' video sequence

What has been described in vague terms is two different *state* vectors for this video sequence. The current state of the image can be described as a list of numbers, which represent the pixel intensities. But the current state of the fountain could be described in terms of how strong the flow is, and exactly which way the wind is blowing, and other such descriptions. Such a description, in terms of fountain characteristics, is a much more compact representation. What is needed, however, is the ability to translate from one state representation to the other, so that given a fountain state vector (i.e. a description of what the fountain is doing), we can create an image state vector (i.e. what the fountain looks like). This can easily be written in matrix-vector terms:

$$\mathbf{y}(t) = \mathtt{C}\mathbf{x}(t)$$

where $\mathbf{y}(t)$ is the vector describing the image pixels at time t, $\mathbf{x}(t)$ is the (much smaller) description of the current fountain state. The matrix C is the mapping which knows how

to convert the state (e.g. 'a fountain that's a bit stronger than usual at the moment') into the image vector.

What we want to find is a single matrix C that will do this job—a matrix which knows everything about fountains that's needed for the sequence. In this way, rather than storing every video frame, we can instead just store this matrix and then each frame can be described by it's short fountain state vectors.

So, how do we find C or, indeed, the best fountain state vector $\mathbf{x}(t)$? The first thing to do is to to represent each image as vector. Rather than having a matrix of numbers, we just want to turn it into a long list, perhaps produced by scanning each row in turn. This can be done in MATLAB using the `:` operator, as explained in the implementation notes. This turns each image into a vector $\mathbf{y}(t)$. Our fountain state vector is already a vector, $\mathbf{y}(t)$.

We want to analyse all the frames together, so let's start by stacking them column by column to make a matrix:

$$\begin{aligned} \mathtt{Y} &= \begin{bmatrix}\mathbf{y}(1) & \mathbf{y}(2) & \cdots & \mathbf{y}(\tau)\end{bmatrix} \\ \mathtt{X} &= \begin{bmatrix}\mathbf{x}(1) & \mathbf{x}(2) & \cdots & \mathbf{x}(\tau)\end{bmatrix} \end{aligned}$$

Because these are just all the state vectors stacked together, and the same translation C works for all of them, we can then write[20]

$$\mathtt{Y} = \mathtt{CX}$$

What is therefore needed is some sensible way of dividing our known image data matrix, Y, into two matrices: one common translation matrix C, and the matrix of fountain states. There are many possible ways, but it turns out that the optimum approach is to use the SVD.[21]

Performing SVD on Y gives:

$$\mathtt{Y} = \mathtt{USV}^T$$

and we will choose to simply assign C and X from this decomposition:

$$\begin{aligned} \mathtt{C} &= \mathtt{U} \\ \mathtt{X} &= \mathtt{SV}^T \end{aligned}$$

In graphical terms, this gives (using the economy SVD since the number of image pixels, N is greater than the number of frames, τ):

$$\overset{\mathtt{Y}}{\begin{bmatrix} & & & & \cdot\\ & & & & \cdot\\ & & & & \cdot\\ & & & & N\\ & & & & \cdot\\ & & & & \cdot\\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}} = \overset{\mathtt{U}}{\begin{bmatrix} & & & & \cdot\\ & & & & \cdot\\ & & & & \cdot\\ & & & & N\\ & & & & \cdot\\ & & & & \cdot\\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}} \overset{\mathtt{S}}{\begin{bmatrix} & & & & \cdot\\ & & & & \cdot\\ & & & & \tau\\ & & & & \cdot\\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}} \overset{\mathtt{V}^T}{\begin{bmatrix} & & & & \cdot\\ & & & & \cdot\\ & & & & \tau\\ & & & & \cdot\\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}}$$

[20] This is a useful general trick if you want to perform the same operation on a number of vectors.

[21] This should be an obvious choice in any case, since the SVD is the most numerically stable of the common decompositions, and so is the answer in most problems.

Thus far this is not a great saving, but now we can use the approximating power of the SVD to remove some dimensions (see Section 13.3). We have τ singular values, but let us choose to keep only the n best. This equates to deleting the last $\tau - n$ columns of U, the last $\tau - n$ rows and columns of S, and the last $\tau - n$ columns of V (or rows of $\mathtt{V}^T$). This leaves us with

$$\overset{\tilde{\mathtt{Y}}}{\begin{bmatrix} & & & & \cdot \\ & & & & \cdot \\ & & & & \cdot \\ & & & & N \\ & & & & \cdot \\ & & & & \cdot \\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}} = \overset{\tilde{\mathtt{U}}}{\begin{bmatrix} & & \cdot \\ & & \cdot \\ & & \cdot \\ & & N \\ & & \cdot \\ & & \cdot \\ \cdot & n & \cdot \end{bmatrix}} \overset{\tilde{\mathtt{S}}}{\begin{bmatrix} & & \cdot \\ & & n \\ \cdot & n & \cdot \end{bmatrix}} \overset{\tilde{\mathtt{V}^T}}{\begin{bmatrix} & & & & \cdot \\ & & & & n \\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}}$$

and hence, writing $\tilde{\mathtt{C}} = \tilde{\mathtt{U}}$ and $\tilde{\mathtt{X}} = \tilde{\mathtt{S}}\tilde{\mathtt{V}^T}$:

$$\overset{\tilde{\mathtt{Y}}}{\begin{bmatrix} & & & & \cdot \\ & & & & \cdot \\ & & & & \cdot \\ & & & & N \\ & & & & \cdot \\ & & & & \cdot \\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}} = \overset{\tilde{\mathtt{C}}}{\begin{bmatrix} & & \cdot \\ & & \cdot \\ & & \cdot \\ & & N \\ & & \cdot \\ & & \cdot \\ \cdot & n & \cdot \end{bmatrix}} \overset{\tilde{\mathtt{X}}}{\begin{bmatrix} & & & & \cdot \\ & & & & n \\ \cdot & \cdot & \tau & \cdot & \cdot \end{bmatrix}}$$

Which is what we were after. There is a different column of $\tilde{\mathtt{X}}$ for each time instant, and each of these vectors is short—only n elements long. Each of these is translated through a matrix $\tilde{\mathtt{C}}$ to give the final video sequence, where the image vectors are N numbers long.

B. Exercise

Load the MATLAB file `fountain.mat`. This contains the 'fountain' sequence as a three dimensional array (see the implementation notes). Use the `size` command to see the dimensions of the image, and the number of frames. Write a function, `playmovie`, which turns this 3D array into a MATLAB movie and plays it. Use this to see what the original video looks like.

Write three further MATLAB functions:

- `encodevideo`: Converts the frames of the video into column vectors, and forms the matrix Y. Perform SVD on this matrix to give U, S and V.
- `quantisevideo`: Takes the three matrices, and a specified number of dimensions n, and removes the necessary rows and columns, returning the three modified matrices $\tilde{\mathtt{U}}$, $\tilde{\mathtt{S}}$ and $\tilde{\mathtt{V}}$.
- `rebuildvideo`: Takes the three matrices, multiplies them back together to give the reconstructed matrix $\tilde{\mathtt{Y}}$. Then turns each of the columns of $\tilde{\mathtt{Y}}$ back into a matrix and builds a three-dimensional array holding the sequence.

First use the `encodevideo` and `rebuildvideo` functions pull apart and remake the video. Use `playmovie` on this new array to check that the reconstructed movie plays the same as the original. Now try using `quantisevideo` between these two calls, setting $n = 1$ to see what the video looks like with the very barest description of the fountain's state. Try some other values of n to find out what kinds of motions you can get with different numbers of significant values.

Use trial and error to find the smallest value of n which gives a RMS pixel error, over the whole sequence, of less than 5. With this value of n, how many numbers are needed to describe the matrices $\tilde{\mathtt{C}}$ and $\tilde{\mathtt{X}}$. How does this compare with the number of numbers in the original sequence? What is the compression ratio?

C. Implementation notes

1. **Three dimensional arrays**
   The video file you will use is stored in a *three dimensional* array, `[x, y, time]`. Three dimensional arrays can be accessed exactly analogously to normal 2D matrices: if the array is called `T`, the first image in the sequence is `T(:,:,1)` i.e. all rows and all columns of the data at the first time instant.

2. **Making movies**
   To make a MATLAB movie, you need to write a `for` loop which steps through each time `t` step in the sequence, extracts the image, and calls

   ```
   M(t) = im2frame(image, gray(256))
   ```

   for each. Here, `M` is the movie you are building up, and `t` specifies the frame number. *`image`* is matrix which holds the image you want to use as that movie frame. the The call to `gray(256)` makes a colormap with values from 0–255, which is the range of the intensities in the video sequence.

3. **Turning matrices into vectors, and vice-versa**
   You will need to be able to turn an image matrix into a vector of numbers to make the matrix `Y`. You can do this by typing

   ```
   t = A(:)
   ```

   The `(:)` tells MATLAB to turn the matrix `A` into a vector called `t`. Numbers are read into the vector column-by-column.
   To turn a vector back into a matrix, you need to have a matrix *of the right size and shape* into which the numbers will go. Otherwise, with just a vector, MATLAB doesn't know how many rows and columns you want. Assuming that the matrix `B` is the right size and shape,

   ```
   B(:) = t
   ```

   will put the vector `t` into `B`. You will have to pass the size (number of rows and columns) to your `rebuildvideo` function to allow it to reshape the vectors into the correctly-sized images.

4. **Computation time**

   Calculating the SVD of a matrix of this size, or performing other calculations, does take a good few seconds, so do not be surprised when you have to wait a little while. Please ensure, however, that you use the 'economy' SVD as described in Section 13.5, as the full version takes a lot longer and also has a habit of consuming more memory than is available on these machines!

5. **Calculating RMS error**

   To calculate the RMS difference between two matrices, you can first subtract the two matrices and square all the values (using the `.^` operator). You can calculate the sum of all the elements in a matrix `A` by using `sum(sum(A))`. The first call to `sum` totals up each column to give a vector of partial sums. The next call then adds up all the elements in that vector. After this, you just need to divide by the number of elements, and find the square root.

D. Evaluation

Demonstrate your final encoded sequence to a demonstrator.

Additional: More video textures

A. What is the point of video textures?

The compression ratios possible using video texture techniques are far from stunning. In contrast, the MPEG-2 compression used in DVDs uses a block-based JPEG-style compression, with motion prediction, and can reach compression ratios of 40:1. However, compression is not the main advantage of video textures. What this approach enables us to do is to synthesise novel videos of the object.

At the moment in computer-generated videos, and computer games, moving scenery (such as waves, or the fountain considered earlier) are poorly generated. They are produced either by playing a short recorded sequence again and again, or by using very simple (and artificial-looking) model, which is again repeated *ad infinitum*. The problem of generating textures randomly is twofold: creating an image of a fountain at random involves knowing a lot about 'fountain-ness', and also that it must also be consistent with what's happened before—the fountain can't suddenly jump from being very small to being very large. The video texture approach knows about 'fountain-ness' in the matrix `C`, and it is much easier to randomly generate the small **x** vectors than a whole image. In fact, it is easy and fast enough to do this in real time.

B. Synthesising video textures

Synthesising a new fountain video is fairly simple given the video texture decomposition—it is just a case of creating a plausible new sequence of **x** vectors. There is not time in this course to also go over the theory to do this, but a function `randommovie` has been

written, which performs this synthesises. Type `help randommovie` for information, or you can have a look at the function in the editor if you are really interested to see how it works. Try generating a new movie for your sequence.

C. More sequences

Two more sequence are provided for on which you can test your video texture functions: `toilet.mat` and `steam.mat`. Try your functions on these ones as well.

14 Complex numbers

Apart from matrix and vector computations, MATLAB also supports many other mathematical and engineering concepts, and among these are complex numbers. Complex numbers can be typed into MATLAB exactly as written, for example

```
>> z1=4-3i
z1 =
   4.0000 - 3.0000i
```

Alternatively, both `i` and `j` are initialised by MATLAB to be $\sqrt{-1}$ and so (if you've not redefined them) you can also type

```
>> z2 = 1 + 3*j
z2 =
   1.0000 + 3.0000i
```

You can perform all the usual arithmetic operations on the complex numbers, exactly as for real numbers:

```
>> z2-z1
ans =
  -3.0000 + 6.0000i
>> z1+z2
ans =
     5
>> z2/z1
ans =
  -0.2000 + 0.6000i
>> z1^2
ans =
   7.0000 -24.0000i
```

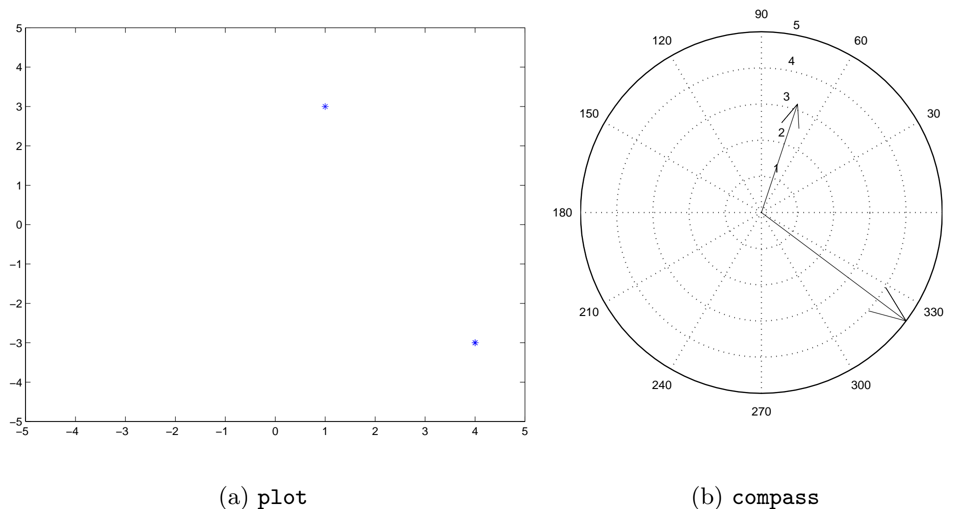
MATLAB also provides some functions to perform the other standard complex number operations, such as the complex conjugate, or the modulus and phase of the number, and these are shown in Table 6. Other functions which have mathematical definitions for complex numbers, such as $\sin(x)$ or e^x, also work with complex numbers:

```
>> sin(z2)
ans =
   8.4716 + 5.4127i
>> exp(j*pi)    %Should be 1
ans =
  -1.0000 + 0.0000i
>> exp(j*2)
ans =
  -0.4161 + 0.9093i
>> cos(2) + j*sin(2)  %Should be the same as e^2i
ans =
  -0.4161 + 0.9093i
```

Matrices and vectors can, of course, also contain complex numbers.

function	meaning	definition ($z = a + bi$)
`imag`	Imaginary part	a
`real`	Real part	b
`abs`	Absolute value (modulus)	$r = \|z\|$
`conj`	Complex conjugate	$\overline{z} = a - bi$
`angle`	Phase angle (argument)	$\theta = \tan^{-1}\left(\frac{b}{a}\right)$

Table 6: Complex number functions



(a) `plot`

(b) `compass`

Figure 14: Graphical representations of the complex numbers $z_1 = 4 - 3i$ and $z_2 = 1 + 3i$ using the `plot` and `compass` commands.

14.1 Plotting complex numbers

The `plot` command in MATLAB also understands complex numbers, so you can use this to produce an Argand diagram (where the x-axis represents the real component and the y-axis the imaginary):

```
>> plot(z1,'*', z2,'*')
>> axis([-5 5 -5 5])
```

These commands produce the plot shown in Figure 14(a). This is not immediately clear since it is difficult to tell where the origin is. You could also mark the origin, or draw lines from the origin to the numbers, but an alternative is to plot them using the `compass` function, as shown in Figure 14(b):

```
>> compass(z1)
>> hold on
>> compass(z2)
```

(The `compass` function can also plot ordinary 2D vectors, by giving an x and y displacement.)

14.2 Finding roots of polynomials

MATLAB provides a function, `roots`, which can be used to find the roots of polynomial equations. The equation is entered into MATLAB, surprise, surprise, using a vector for the array of coefficients. For example, the polynomial equation

$$x^5 + 2x^4 - 5x^3 + x + 3 = 0$$

would be represented as

```
>> c = [1 2 -5 0 1 3];
```

The `roots` function is then called using this vector:

```
>> roots(c)
ans =
  -3.4473
   1.1730 + 0.3902i
   1.1730 - 0.3902i
  -0.4494 + 0.6062i
  -0.4494 - 0.6062i
```

15 Further reading

This tutorial guide has introduced the basic elements of using MATLAB. There are further guides available from CUED, which are available in the racks at the west end of the DPO:

- *MATLAB by Example*, G. Chand, T. Love, 1998
- *Using Matlab at CUED*, T. Love, 2000
- *Getting started with Matlab*, J.M. Maciejowski and T. Love, 2000

There are also many textbooks about MATLAB, and the following book is particularly recommended:

- *Matlab 5 for engineers*, A. Biran and M. Breiner, Addison-Wesley, 2000

The video texture work used for Exercise 7 is described in more detail in

- Dynamic Textures, S. Soatto, G. Doretto and Y.N. Wu, In *Proceedings 8th International Conference on Computer Vision*, pages 436–446, Vancouver, July 2001
- `http://vision.ucla.edu/projects/dynamic-textures.html`

16 Acknowledgements

A brand new course like this is always a step into the unknown, and I hope there aren't too many teething problems. My thanks go to Roberto Cipolla and Richard Prager for sharing their advice and experience, and to Maurice Ringer for drawing the cricket stumps. And to all of those who tested early versions of this for me.

The cricket example is based on David Mansergh's 2000–2001 4th year project. The video texture sequences are borrowed from Martin Szummer at MIT's AI lab. Inspiration was also drawn from various other MATLAB tutorials and text books.