Introduction to MATLAB

Anthony J. O'Connor
School of Science, Griffith University, Brisbane, Australia

1. What is MATLAB ?

MATLAB started as an interactive program for doing matrix calculations and has now grown to a high level mathematical language that can solve integrals and differential equations numerically and plot a wide variety of two and three dimensional graphs. In this subject you will mostly use it interactively and also create MATLAB scripts that carry out a sequence of commands. MATLAB also contains a programming language that is rather like Pascal.

The first version of Matlab was produced in the mid 1970s as a teaching tool. The vastly expanded Matlab is now used for mathematical calculations and simulation in companies and government labs ranging from aerospace, car design, signal analysis through to instrument control & financial analysis. Other similar programs are Mathematica and Maple. Mathematica is somewhat better at symbolic calculations but is slower at large numerical calculations.

Recent versions of Matlab also include much of the Maple symbolic calculation program.

In this lab you will cover the following basic things:

- using Matlab as a numerical calculator
- entering row vectors and column vectors
- entering matrices
- forming matrix and vector products
- doing matrix products, sums etc
- using Matlab to solve linear equations
- Matlab functions that operate on arrays
- Plotting basic graphs using Matlab.

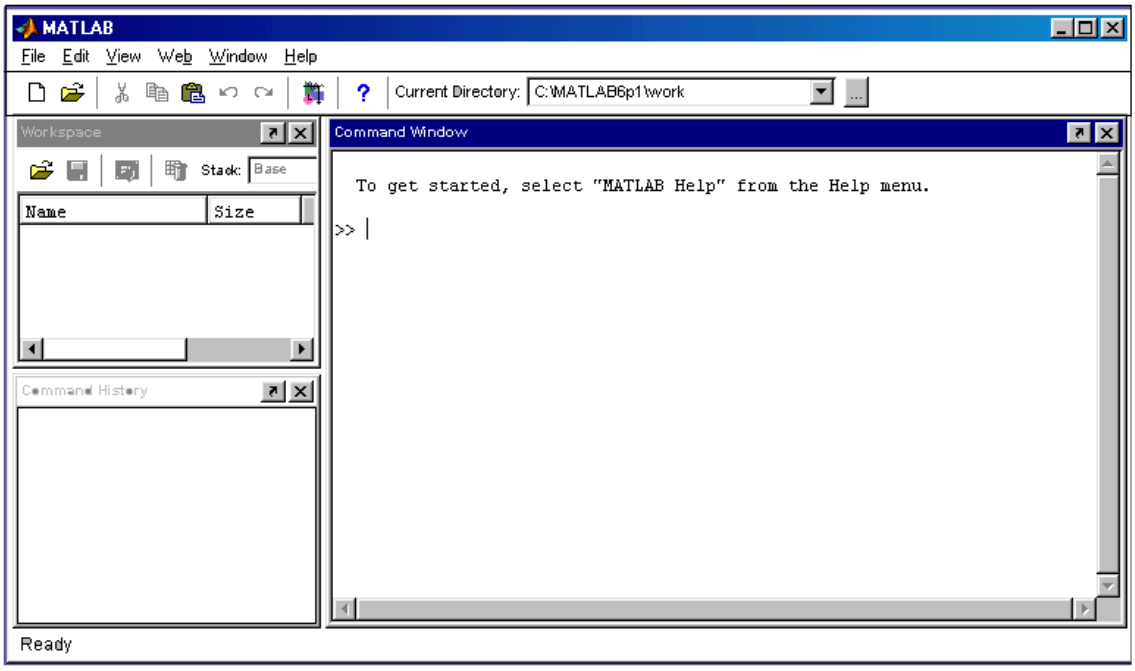
1 How to get started

When the computer has started go through the following steps in the different menus

- Look for the Network Applications folder and double click on that
- Within this you will see a little icon for Matlab61 – double click on that

Within about 30 seconds Matlab will open (you may get a little message box about ICA applications – just click OK on this)

You should now see a screen like the picture below



This is the Matlab screen – it broken into 3 parts

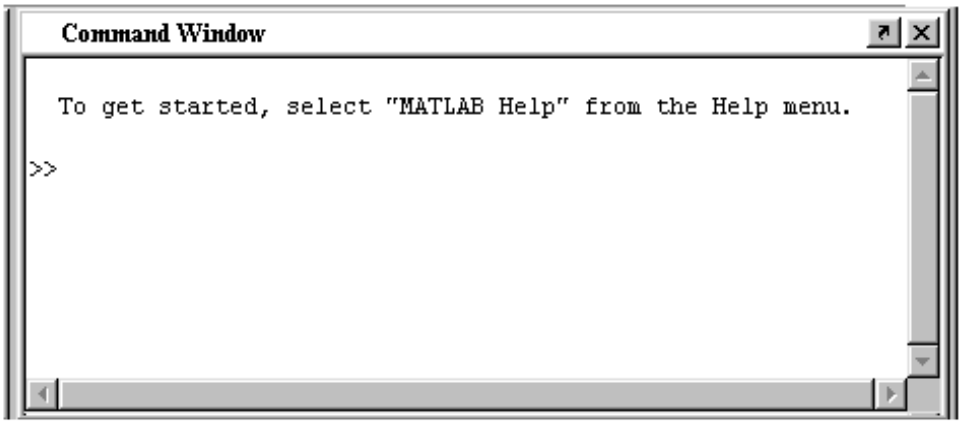
On the right you have the **Command Window** – this is where you type commands and usually the answers (or error messages) appear here too

On the top left you have the **Workspace** window – if you define new quantities (called variables) there names should be listed here.

On the bottom left you have Command History window – this is where past commands are remembered. If you want to re-run a previous command or to edit it you can drag it from this window to the command window to re-run it.

2.1 The Matlab prompt is >> .

Look at the **Command Window** and you will see the cursor flickerring after the >> prompt. This means that Matlab is waiting for further instructions.



2.2 Seeing the Matlab demo.

Just type the command

>> **demo** **(and then press Enter or Return key)**

and run the simple slideshow under the **Matrix manipulation** option.

This briefly shows you some of the things that Matlab can do - don't worry if you do not know what everything means here.

2.3 Simple arithmetic with Matlab

The basic arithmetic operators are +, -, * (for multiplication), / (for divide) & ^ for powers. Where necessary brackets should be used to make the order in which things are evaluated completely clear.

Enter the following commands to learn the basic way that formulas can be evaluated in Matlab (press enter after you have typed in the command. Record the answer beside the command.

>> **3 + 5 -7** .....

The result of this calculation is stored in the temporary variable called **ans**.
It is changed when you do another calculation.

If you want to save the answer from a calculation you can give it a name e.g.

>> **t = 3 + 5 - 7** .....

This line creates a variable t to save the answer in. If you want to find out what t is use

>> **t** .....

Equally you can use t in a formula

>> **2*t + 2**.....

2.4 The order of operations and putting brackets in can matter

In the following calculation does Matlab do the * or the ^ first

>> **3^2*4** **…….**

To find this out think about the two possible options

- One way to do the calculation is that powers are done before multiplication. This $3^2 * 4 = 9 * 4 = 36$

- The other option is to do the multiplication first is $3^{2*4} = 3^8 = 6561$

In this case you should have obtained the answer 36 – this is because Matlab does 3^2 first and then multiplies the answer by 4 – generally powers are done first, then multiplication and division and finally additions and subtractions.

One way to force Matlab (or any software) to do calculations in a defined order is to use brackets.

Compare the following commands and make sure that you understand what is happening and why you get the answer

>> **3 – 4/4 –2** **……..**

>> **(3-4)/(4-2)** .....

>> **(3-4)/4 –2** **……..**

The rules for evaluating expressions involving numbers are

- things inside brackets are done first

- within a bracket or expression generally operations closest to a value are done first

2.5 Extended arithmetic (IEEE)

You need to understand the following section to interpret the output from some calculations – particularly when we make accidental errors.

What does Matlab produce if you ask it to find:

>> **1/0** .....

>> **-1/0** .....

>> **0/0** .....

>> **1/Inf** .....

The symbols **Inf** (meaning infinity) and **NaN (not a well defined number)** are part of a special standard for computer arithmetic. This is the IEEE international standard for extended arithmetic.

Typically you get **Inf** from 1/0 and **NaN** from 0/0 type calculations.

While Inf and NaN are usually signs of bad problem setup Matlab will try to manipulate these results consistently

For example extended arithmetic will ensure that 1/(1/x) always equals x.

Test this out by asking Matlab to calculate

>> **1/(1/0)** **……**

Matlab can also handle c**omplex numbers** – they are entered as 1 + i or -1+3*i etc
The symbol **i** is best kept as a reserved symbol (Matlab also interprets **j** as a complex number to fit in with usage in electrical engineering)

2.6 Matlab works to 15 significant figures but usually only shows 5 figures

Type **pi** and you will normally get 3.1416

You can change to the **long format** in which all 15 figures are displayed

>> **pi** **……**

>.> format long

>> pi **.....**

An easier and more flexible format is scientific notation with 5 significant figures

>> **format short e**

>> **pi** .....

Mostly we work in the short format.

To go back to short format

>> format short ……..

Very large or small numbers use exponential format - e.g. 6.00e+23 = $6*10^{23}$

2.7 The consequences of finite accuracy on a computer

Matlab only has limited accuracy (although it is more than enough for most uses)

Computers are normally set up to store numbers in a fixed amount of memory
Matlab uses 64 bits and this lets it store numbers as large as $2*10^{308}$ or as small as $2*10^{-308}$ But no matter which number you work with Matlab can only store it to 15 significant figures

For example:

Matlab could handle 1.234 567 890 234 56 (fifteen figures)

But a 20 digit number like 1. 234 567 890 234 567 890 12 is truncated to 15 figures.

This is an example of a **round-off**. Other cases of round-off occur if you add or multiply 15 figure numbers and then only store the first 15 significant figures of the answer. **Round-off cannot be avoided in computer calculations.**

This type of unavoidable 'error' can affect the answers you get

What is the exact value of sin(π) ……

What does Matlab gives for

>> **sin(pi)** **……**

Matlab uses a high order polynomial to approximate sin(x). This has very slight roundoff errors that give a small non-zero answer instead of exactly zero

Interpreting very small answers

Sometimes a answer that really ought to be exactly zero might appear as a small number like 10^{-15}.

2.7 Variable names:

Matlab distinguishes between lower case and capital letters - so **A and a are different objects for Matlab.** The normal convention to use lower case names except for global variables. Only the first 19 characters in a variable name are important.

3. Matlab has many built in functions

Among them are:

Trig functions sin, cos, tan, sec, cosec and cotan
(sin(x) etc assume that x is in radians)

Are the following what you expect ?

>> **sin(pi/2)** .....

>> **cos(pi/2)** .....

Remember that pi is a built-in constant

Inverse trig functions asin, acos, atan
(these are abbreviations for arcsin, arccos and arctan - the answers are returned in radians so that asin(1) = pi/2)

Exponential functionexp

Logarithm functions log is log to base e, while
log10 and log2 are logs to bases 10 or 2

Hyperbolic functionscosh, sinh, and tanh

Matlab also has many other more sophisticated functions for solving linear equations, getting eigenvalues of matrices, solving differential equations or calculating integrals numerically.

4. Vectors and matrices in Matlab

Matlab is most used to work with matrices and vectors. Vectors are either row vectors or column vectors and it is usually important to be clear as to what kind of vector you mean.

Row vectors have one row and several columns like $\begin{bmatrix} 4 & 7 & 10 \end{bmatrix}$

Column vectors have several rows in one column like $\begin{bmatrix} 4 \\ 7 \\ 10 \end{bmatrix}$

The main use of matrices and vectors is in solving sets of linear equations:

The equations

$$\begin{aligned} 2x + 6y + 3z &= 5 \\ 3x + 8y + z &= 7 \\ x + 5y + 3z &= 2 \end{aligned}$$

can be re-written in matrix/vector form as

$$\begin{bmatrix} 2 & 6 & 3 \\ 3 & 8 & 1 \\ 1 & 5 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 2 \end{bmatrix}$$

Here the vectors are column vectors . The matrix contains the coefficients from the equations.

4.1 Entering row vectors

To enter the row vector $v = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$ into Matlab use the command

>> **v = [1, 2, 3, 4]**

The square brackets are essential in Matlab when defining an array.
To check this see what Matlab has stored as **v**

>> **v ……**

You can leave out the commas and simply put in a few spaces between different entries.

>> **w = [1.2 -6.7 8.91]** .....

4.2 Entering column vectors

Column vectors are like the ones that appear on the RHS of a set - e.g. $\begin{bmatrix} 3 \\ 4 \\ 1.2 \end{bmatrix}$

This is a matrix with just one column. The example here has 3 rows.

To create this column vector in Matlab you must mark the end of a row with a semi-colon ;

>> **a = [3; 4; 1.2] …….**

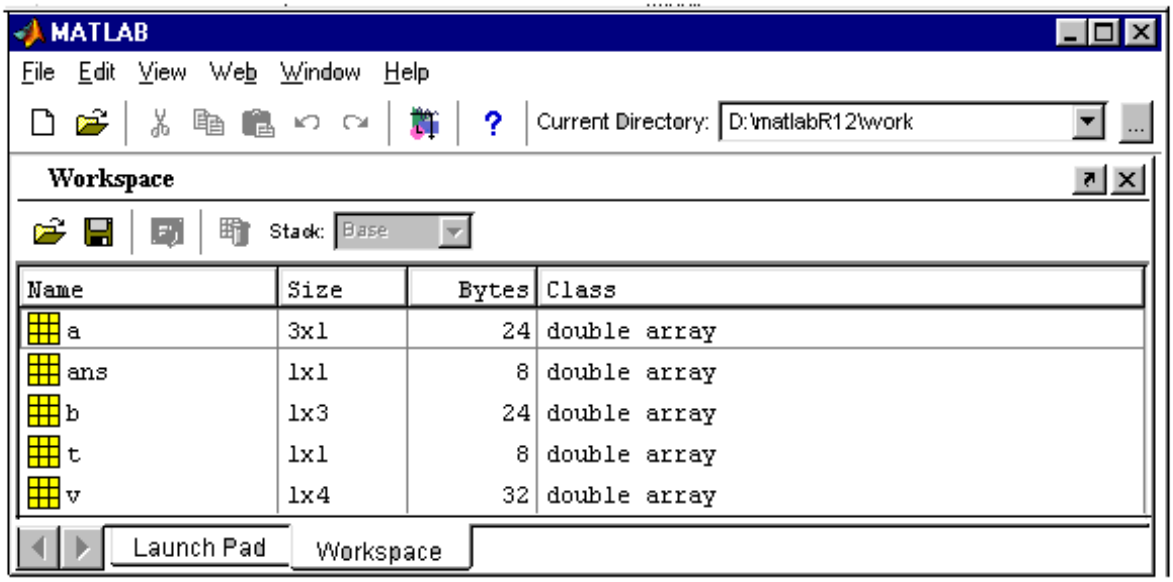
The semi-colons are essential here.

If you leave out the semicolons what will you get ?

>> **b = [3 4 1.2] ……**

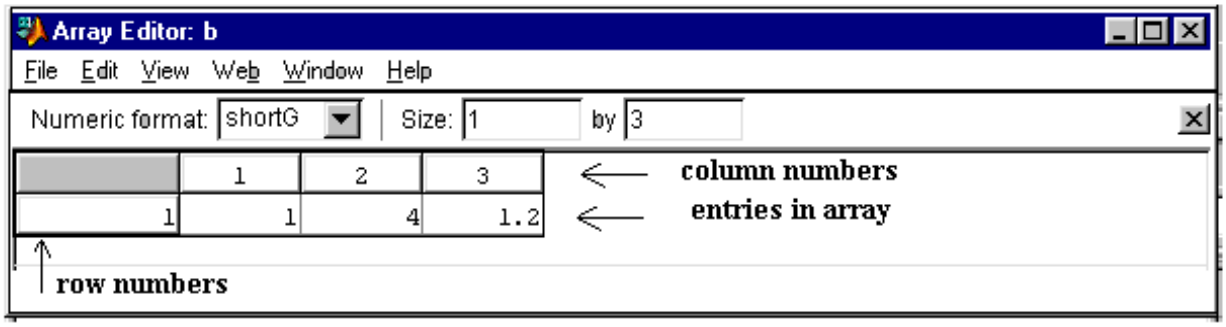
4.3 What variables have you defined already ?

Now look across at the **Workspace** window and you should see something like the picture below _ you may need to expand this window a bit to see everything



This gives the name of the variable – its size as an array (1 x 1 is a single number). You can also see the content of an array by double clicking on the yellow box beside its name.

For example double clicking on b above gives



When you open the array editor you can see the entries in the array - for large arrays you can scroll up/down and left/right

If you want to keep in the command window you can just do **whos** and it prints a table like the window above.

4.4 Creating matrices

To define a matrix we use the symbol **;** to indicate the end of row.

Try the following to create the matrix p = $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

>> **p = [1 2; 3 4 ; 5 6]** .....

What output does this produce – how many rows & columns does it define:

Repeat the definition of p but make the deliberate mistake

>> **p = [1 2 3; 3 4; 5 6]**

This is an example of a Matlab error message.
Do you understand the error here ?

Check that this command has not altered the value of p - how ? .....

4.5 Selecting numbers, rows and columns for a matrix

The matrix p that was just defined was $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

Try the commands and comment on what they select:

>> **p(1, 2)**.....

>> **p(2,1) …….**

>> **p(2, :)** .....

>> **p(: , 2)** .....

Now use the information that you can get from these examples to write down the commands that will select

- the number in the 3rd row, 2nd column of p .....
- the number in the 1st row, 2nd column of p .....
- the first row from p .....
- the third row from p .....
- the first column from p .....
- the third column from p .....

5 Defining some common matrices and vectors

5.1 Vectors and matrices of zeros

To create a row vector containing 4 zeros check that the following command works

>> **r = zeros (1, 4)**

What does the following command do

>> **r = zeros(4, 1)**

Based on these two examples how would you create a matrix of zeros with 3 rows and 4 columns

>>

5.1 Basic matrix operations

The symbols for matrix addition, subtraction, multiplication and powers are +, -, * and ^

If a is a square matrix then **a^2** means a*a - the matrix product of a with itself.

Define the following matrices and see what happens with the matrix commands
(see the last page to revise the definition of matrix multiplication)

Write Matlab commands that will create the following matrices

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad b = \begin{bmatrix} 3 & 4 \\ 0 & 1 \end{bmatrix} \qquad c = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

>>

>>

>>

Now try to find (if the answer exists check that it is correct by hand)

>> **b + c** .....

>> **b – 2*c** .....

>> **a + b** .....

Matlab also does matrix multiplication (provided that the matrices can be multiplied consistently). First calculate b*a by hand and then use Matlab to check your answer

$$b*a = \begin{bmatrix} 3 & 4 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad =$$

>> **b*a** .....

Now check that Matlab gives the same results for b^3 and b*b*b (you don't need to do these by hand)

>> **b*b*b** .....

>> **b^3** .....

Does the product a*b make sense ?

>> **a*b**

More complicated algebraic expressions like b + c – (b^2)*c can also be found.

5.2 The dot symbol

The dot symbol is special to Matlab – it is not a standard mathematical notation and should not be confused with the dot product of two vectors.

It changes the meaning of ^ in a very important way.

If a is a square matrix then **a^2** means a*a - the matrix product of a with itself.

But **a.^2** means that we get a new matrix by squaring each element in it separately.

Even if a is not square (so that a*a is not defined) we can still define **a.^2**

We have defined $a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ and $b = \begin{bmatrix} 3 & 4 \\ 0 & 1 \end{bmatrix}$

Predict the values of a.^2 and check that you are right using Matlab

>> **a.^2** .....

Also see what b.^2 is

>> **b.^2** .....

Check that this is different from the matrix b^2

>> **b^2** **……..**

6. The real power of Matlab - solving linear equations in one step

Suppose that you want to solve the equations $\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 1 \\ 3 & 5 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 7 \end{bmatrix}$

You can do this in three steps.

We write this in the symbolic form A.**x** = **b**
[here vectors are written in bold letters]

A is the 3x3 coefficient matrix and b is the 3 row x 1 column vector on the RHS.

Step 1 **Define the matrix A - fill in the definition**

```
>>  A = [ 0  1  2;  1  2  1;   3  5  2]
```

Step 2 **Define column vector b**

```
>>  b =  [ 1; 3; 7]
```

Step 3 **Solve for x using the backslash command**

```
>>  x  =   A\b
```

(notice that we do not have to define a symbolic vector containing the x, y, z symbols.
the variable x is the column vector containing the numerical solutions to the equations)

Matlab returns the value

```
x =   -2.0000
       3.0000
      -1.0000
```

Step 4 **Check that this is really a solution by calculating the difference between the two sides of the equation – this is A*x - b**

```
>>  A*x - b
```

Matlab returns the value

```
ans =  0
       0
       0
```

Be careful to get the order correct and the direction of the slash, b\A will be quite different

Practice the use of the backslash command

Use the \ command to solve the following two equations and check that the Matlab solution is correct.

(i) $$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 1 \\ 3 & 5 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 8 \end{bmatrix} \quad \&$$

x = ...

A*x = ...

7. Creating longer vectors and plotting graphs

One very useful feature of Matlab is its ability to plot graphs of functions quickly.

For example suppose that we want to plot the graphs of $y = x^2$ between 0 and 5.

Step 1: Create a vector of 100 evenly spaced points between 0 & 5

>> **x = linspace(0, 5, 100);**

The semicolon (;) is used to stop Matlab printing all 100 numbers out. If you really want to check that the command worked use Data Editor to see what is in x now.

Step 2 Compute the y values for the curve $y = x^2$

>> **y = x.^2;**

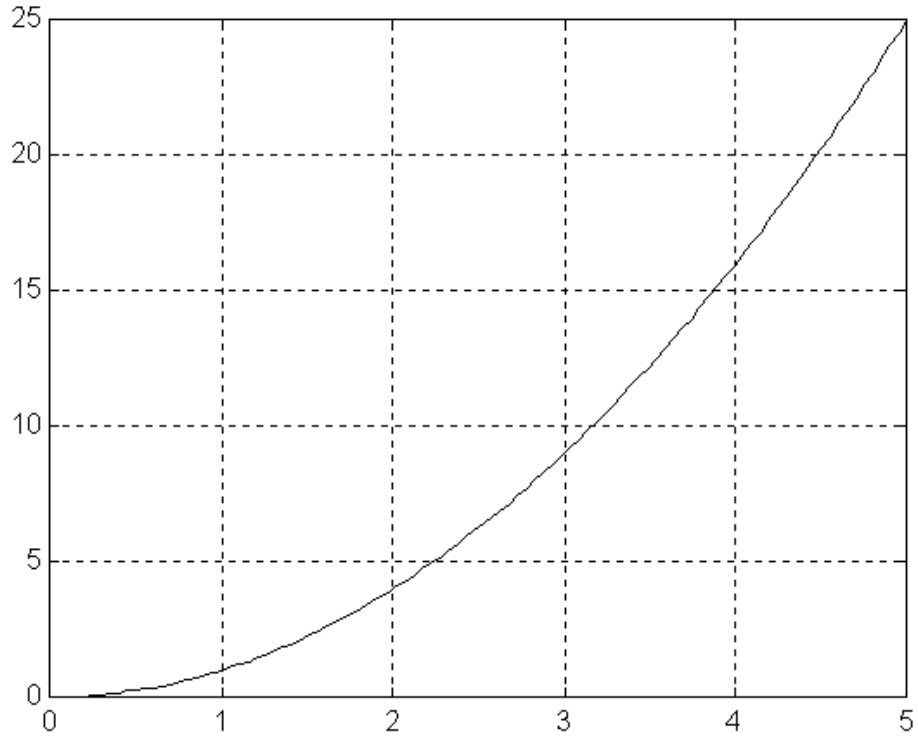
[the dot here is very important - it squares the elements of x one at a time]

Step 3 Plot just the graph of $y = x^2$

>> **plot(x, y); grid; shg**

Here I put three commands on the one line – grid puts dotted lines on the graph to make it easier to interpret. The shg command means 'show graph'

If the graph does not appear automatically go to the Window menu and select figure 1
You should see a graph drawn in black.



7.1 Adding titles and labels to a graph

You can add labels to the x and y axes and also put a title on the graph as shown below. Often the label on the x axis is used as a description of the graph.
You do not have to add labels or titles to graphs but if you are generating a graph to paste in a Word document then labels and/or title are very useful.

The following graph was generated by the commands

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

>>      xlabel(‘ x values’ )
>>      ylabel(‘ y values’ )

>>      title(‘Plot of y = x^2 ' )
```

7.2 Putting several graphs of the one plot

Suppose that I want to compare the graphs of $y = x$, $y = x^2$ and $y = x^3$ over the interval from -2 to +2.

We need to set up the x array (fill this in yourself)

>>

Next we need to calculate the coordinates on these three graphs

>> **y1 = x;**

>> **y2 = x.^2;**

>> **y3 = …….**

An extended version of plot can take more pairs of x, y arrays.
Each x, y pair is added to the graph

>> **plot(x, y1, x, y2, x, y3); grid; shg** [try this yourself]

Add in the grid and the shg command as well

Matlab uses a default colour scheme that usually plots them using colours in the order blue first, then green and then red.

You can control the colour scheme yourself – e.g. the following command makes the graphs blue, black and green

>> **plot(x, y1, 'b', x, y2, 'k', x, y3, 'g'); grid; shg** [try this yourself]

You can change the colours or the style of the three plots yourself by adding a third 'style parameter to any (or all) of the x, y pairs.

This is typical of Matlab – to make plots you have to give the x & y data and then there are optional things that you do the plot.

7.3 You can change the colour and style of the different lines in the graph

Style refers to the way that the curve appears – a solid line, a dotted line, a line made up of circles etc and/or to the colour of the curve.

The complete list of style options is

Various line types, plot symbols and colors may be obtained with PLOT(X,Y,S) where S is a 1, 2 or 3 character string made from the following characters:

The left column gives the colour options, the right column the options for the way the line looks.

y	yellow	.	point
m	magenta	o	circle
c	cyan	x	x-mark
r	red	+	plus
g	green	-	solid
b	blue	*	star
w	white	:	dotted
k	black	-.	dashdot
		--	dashed

Examples

>>	**plot(x, y, 'b')**	plots the graph in blue as a solid line (default)
>>	**plot(x, y, 'bo')**	plots graph as blue circles
>>	**plot(x, y, 'g:')**	graph appears as a dotted green line.
>>	**plot(x, y1,'k', x, y2,':b')**	first curve is a black line and second is blue dots

Warning

When you come to plot a graph on a black & white printer the lighter colours are very faint. A good rule with graphs to use different styles sparingly and just enough to clearly separate different curves.

Exercise

Go back to your plot of y = x, y = x^2 and y = x^3.

>> **plot(x, y1,x, y2, x, y3)**

Modify this command and add others so that you get

- a graph of these three curves with y = x a dotted black line
- y = x^2 a blue dot dash line
- y = x^3 is a green dotted line
- add a grid to this graph
- add a title just below the x axis that says

  'Comparing y = x (solid), y = x^2 (.-) and y = x^3 (..) graphs'

[There is a more to plotting - you can control the size of the graph, the axes and put labels on the axes as well.]

8. Matlab is designed to work easily with vectors

A few Matlab commands hides a lot of background instructions.

For example

>> **x = [1.0, 3.0, 7.5]**

>> **y = sin(x)**

This gives the array [0.8415 0.1411 0.9380] - here 0.8415 = sin(1), 0.1411 = sin(3) and 0.9380 = sin (7.5) - with all values taken in radians.

In most other languages (e.g. Pascal, C, Java) if you wanted to do this calculation you would need to set up a loop to process the three entries individually. In Matlab the loop is done for us so that it is very easy to do many calculations Matlab does not do these things by magic – behind the scenes is a sophisticated 'engine' that collects your commands and interprets them.

In Matlab many functions (e.g. sin, cos, log, exp, sqrt) all accept vector input and give vector output.

Use this and your knowledge of graphics command to write down the commands that will plot the graph of y = cos(x) between x = 0 and x = 10 using 300 sample points

9 Core Matlab syntax and most important built-in functions

This is the information that is most useful for writing Matlab programs. It does not include material on specialized numerical functions or linear algebra functions or graphics. It is supposed a quick reference for the most commonly used commands in programming.

Functions are defined in general form often using names for input variables than numerical examples

9.1 Vectors and arrays

Initializing and creating important vectors

- Empty vector v = []

- Zero vector v = zeros(num_rows, num_colns)
- Array of ones v = ones(num_rows, num_colns)

  (to get a row vector set num_rows = 1; to get a column vector set num_colns = 1)

- Random arrays – useful either for testing functions or for statistical simulation

  Uniform distribution over [0,1] v = rand(num_rows, num_colns)
  Normal distribution, centre 0 and SD = 1 v = randn(num_rows, num_colns)

- Define explicitly – examples

  v = [1, 4, 7] gives a row vector
  v = [1; 4; 7] gives a column vector
  v = [1, 4, 5; 6 7 8] gives a 2 x 3 matrix

- Evenly spaced integer array

  v = start: step: end (begin at start value, increase by step value and finish at or before end value)
  e.g v = 2:2:9 produces v = [2, 4, 6, 8]

  A special case only uses start and end and here step = 1

  e.g. v = 2:5 produces v = [2, 3, 4, 5]

- Evenly spaced real number arrays

  The linspace command generates a fixed number of values beginning at start and finishing at end

  v = linspace(start, end, number_values)

9.3 Accessing array entries

- Change one entry v(p) = a replaces pth entry by a
- Changing a few entries v([p, q]) = [a, b] replaces pth & qth entries

- To change all entries from v(p) to v(q) to constant value

  v(p**:** q) = a

- More involved command – change v(1), v(3), v(5), v(7) to zero

  v(1**:**2**:**7) = 0

- Accessing a whole row u = v(p, **:**) u is the pth row of v
- Accessing a whole column u = v(**:**, p) u is the pth column of v matrix

  In the last two commands the full colon symbol **:** represents all entries in a row or a column.

9.4 Quite a sophisticated example of matrix manipulation

To swap rows 3 and 5 in a matrix

>> a([3, 5], **:**) = a([5, 3], **:**)

The right hand side is the input and the left hand side is the output
This says take the 5th and 3rd row in a as your input data.
They then become the 3rd and 5th row of a.

All the extraction and re-assignment operations are done in memory.

This instruction needs to be thought through really carefully

A less zipped up version is to do it in three steps

>> temp = a(3, **:**) ← make a copy of row 3 temporarily
>> a(3, **:**) = a(5, **:**) ← row 3 in a made the same as row 5
>> a(5, **:**) = temp ← now use the copy of row 3 to set row 5 to correct value

Even here you need to think carefully about the order in which these statements are carried out.

10 Using the find command and logical expressions

The find command lets you find the places in an array where the entries obey a specified condition

An example with v = [0.6, 0.55, 0.3, 0.1, 0.98]

```
>>    index  =  find ( v > 0.5)

>>    index = [1, 2, 5]          ← the 1st, 2nd and 5th entries in v are > 0.5

>>    a = v(index)               ← a = [0.6, 0.55, 0.98] are the desired entries in v
```

The find command involves a logical condition – this typically involves comparision operators as well as not, and & or

10.1 Symbols used in logical operators

Symbol	Represents	Symbol	Represents
>	Greater than	>=	Greater or equal to
<	Less than	<=	Less or equal to
~=	Not equal to	= =	Equal to
Not	~	And	&
Or	| (single vertical line)		

You need to be careful with the = = symbol. This is fine for integer valued arrays but can cause problems for floating point arrays. Many calculations introduce small floating point operations and these can change a value slightly from its correct value. If the totally accurate value is (say) 1.5 it is dangerous to use the test

```
>>    index = find ( v = = 1.5)
```

A better test to look for values that are really close to 1.5 – this allows for build up of round off errors

```
>>    index  = find (  abs(v – 1.5) < 10*eps)
```

Here eps is about 10^{-15} the smallest storable step above 1 and 10*eps allows some flexibility in how far v has strayed from its correct value. Exactly how many eps values we should allow depends on the problem and needs a fairly good knowledge of the process being used.

11 Good programming practice with arrays

If you know how long an array will be it is best to initialize it with the zeros command – this reserves space for the array and then later you fill in the actual entries

```
v = zeros(1,200)

for  p = 1:200
        v(p) = …..
end
```

Of course if the entries in v all followed a simple formula that could be constructed from a dot expression that would be best of all

Here are three ways of doing the same thing:

Problem **Construct a 1000 entry array where** $v(p) = \frac{p}{\sin(p)+2}$

Method 1 Create the entries one by one without reserving space

```
for p = 1:1000
        v(p) = p/(sin(p)+2)
end
```

Method 2 Reserve space for v and then fill in the entries one by one

```
v = zeros(1,1000);
for p = 1:1000
        v(p) = p/(sin(p)+2);
end
```

Method 3 Use the dot construction

```
p = 1:1000        ← create the 1000 integer values using built-in colon
v = p./(sin(p)+1)        ← use builtin vectorized functions
```

Comparing the times to do these

Method 1	1.82 seconds
Method 2	0.16 seconds
Method 3	0.0083 seconds (average of 100 runs of method 3)

12 Some useful built-in Matlab functions for arrays

- sum - adds up entries in 1D array, in a matrix it adds up entries in each column
- max - maximum entry in 1D or for matrix it gets maximum entry in each column
- min - maximum entry in 1D or for matrix it gets maximum entry in each column
- sort – sorts entries in a 1D array into increasing order (from right to left).
- mean - gets average value of entries in array
- stdev – standard deviation of entries in the array

13 Useful functions that work on integers

Mod - mod(a,b) is the remainder when a is divided by b

For example mod(45, 10) = 5 or mod(5,2) = 1 or mod(6,2) = 0

One typical example is:

to test whether a variable is a multiple of 10 and print a message

```
if  mod(n, 10) = = 0
    fprintf('When n is %g ……..\n', n)
end
```

14 Messages and displaying variables

To print a message to the screen use fprintf – you can include values of variables using %g and %e format symbols

```
>>      fprintf(‘ Current value of x is %g and of y is %g \n’, x, y)
```

Do not use this when variables are arrays – the message gets repeated for each entry in the array and is useless.

The sprintf command stores the same output as a string s

```
>>      s = sprintf(‘Current value of x is %g and of y is %g ‘, x, y)
```

To display contents of an array use disp function

```
>>      disp(x)
```

You can also the workspace to inspect the contents of an array

15 Important information on functions

Functions can be written in terms of the following statements

- Input and output (more on these later especially for input and output from files)
- Assignment

  x = y (sets value of x to current value of y) or more involved statements
  The variables on the right of an assignment statement must already be defined

- If statement - either if …. End, if … else …. End
- Control statements - counted for loop and while loop controlled by a logical test

Matlab functions are written as m files –

The name of the file and the name of the function should agree.

```
function  [a, b ] = myfun( x, y)
%  this function does .....
%  input variables
%    x =   .....
%    y =  .....
%   output variables
%     a = ..............
%     b = ..........



return
```

function myfun is stored in the file myfun.m

Most Matlab functions have at least one input and one output but this is not essential.

Input variables can either be numerical values or existing variables
To get all the output variables the function must be called with enough output variables

Each output variable must be given a value before the function returns.

15.1 Examples of calling myfun

>> [c, d] = myfun(5, 3) ← a and b are where output values are stored

>> myfun(5, 3) ← ans = the value for a but b value lost

>> x = 6; y = 7; [c, d] = myfun(x, y) ← existing variables used as input values

We do not need to call the input variables and the output variables the same names as they are given in the way the function is written.

15.2 What happens to the internal variables used in a function

While a function is being calculated it has its own workspace but once the function returns all these internal variables are wiped. This can make debugging a function more difficult – we need to use the step by step debugger.

How are variables passed to a function

If we have the myfun function and call it as

>> [c, d] = myfun(x, y)

The variables x and y exist in the workspace and have to be passed to myfun

There are two ways to do this:

- **Pass by value** – we make a copy of x and y and give this copy to myfun to use.

  This is the right thing to do if myfun alters the values of x and y during its operation
  To pass by value we have to use extra memory but we do not endanger the real values of x and y in the Matlab workspace.

- **Pass by reference** – we actually let the function work with values of x and y.

  This is the best choice if myfun never alters x and y and only reads values from the x and y arrays – not having to make a separate copy of x and y is more efficient.

There is another related topic for functions – this is **persistent variables**
A persistent variable within a function is not wiped out after the function returns.- it keeps its value on exit until the next time the function is called.

16 Functions can include sub-functions

Functions can involve quite a few processes and it is often useful to split the workings of a function into several smaller parts. These are called ‘sub-functions’ and can be defined in the same file. However sub-functions are only available within the m file where they are defined

```
function  [a, b] = myfun( x,y)


        % call  subfun 1
        z = subfun1(x)

       % call subfunction 2
       [u, v] = subfun2(x,y)

return

%-------------------------------------
%  subfunction 1 defined here
function a = subfun1(b)

    ….

return

%-------------------------------------
%  subfunction 2 defined here
function [c, d] = subfun2( e, f)

  ……..

return
```

17 Inline functions

This is a new way to define functions that you might only want to use for a short time. They are only available during the current Matlab session unlike m files which are kept as permanent files. They are useful for creating test examples.

Here are three examples of defining an inline function

- g = inline('t^2')

  This defines the function $g(t) = t^2$ - Matlab automatically detects the variable is called t

- g = inline('x^2 –y^2', 'x', 'y')

  This defines the function $g(x, y) = x^2 - y^2$ which has two variables – we also specify that the variables are called x and y and everything is enclosed in single quotes

  We would call g as

  ```
  >>    g(1, 2)
  ```

  This would give the answer –3

- g = inline('sin(a*t+b)','t','a','b')

  A more complicated example $g(t, a, b) = \sin(at + b)$ with three variables t, a and b
  Here the order of the variables that we give to g is specified as t first, then a and b

  ```
  >>    g(1,2,4)
  ```

means that t = 1, a = 2 and b = 4

18 Functions that call other functions – the feval function

A example of such a function might be to estimate the integral $\int_a^b f(x).dx$

The definition of this function might be

```
function est_integral = integrate( f, a, b)



return
```

Here f is the name of a file where the function to be integrated is defined
This function would have a definition like:

```
function res =  f(x)


return
```

Within integrate we have to use the **feval** command. This is just another way to evaluate a function when we know the name of the file where the function is defined.

For example the function sin(x) is defined in the file sin.m

The following commands are the same

```
>>      y = sin(c)

>>      y = feval( 'sin', c)
```

Side comment – functions, procedures and subroutines

Matlab functions are more general than functions in Pascal, Basic or Fortran that usually only give one output. A Matlab function is equivalent to a procedure in Pascal or a +subroutine in Basic or Fortran.

19 Functions that work on strings

Strings are just lists of characters – e.g. s = 'fgh' or s = '12fgh'
Strings are actually stored as arrays of characters and so we can use the length function and refer to individual characters or substrings using array notation.

For example if s = 'abcdefgh'

length(s) = 7
s(3) refers to the 3rd entry in s and gives 'c'
s(3:5) refers to the characters from 3rd to 5th position and gives 'cde'

Some useful built-in Matlab functions for strings

To compare two strings to see if they are exactly the same – strcmp

Example >> a = strcmp('abc', 'abd')

They are not the same so the answer is a = 0 (0 stands for false here). If they were the same a would be set to 1

To search for a substring with a longer string

Example 1 to see if the substring 'cde' occurs within 'abcdefgh'

```
>>  s1 = 'abcdefgh';  s2  = 'cde'
>>  a = findstr(s1, s2)
```

The value of a is 3 – meaning that s2 does occur in s1 and the start of s2 is at the 3rd character in s1

Example 2 Where the substring occurs several times in s1

If the substring occurs several times in s1 the answer from findstr is all the places where s2 starts

```
>>  s1 = 'abcdefghabcdefgh';  s2 = 'cde'
>>  a = findstr(s1, s2)
```

The answer for a is [3, 11]

20 Timing operations

There are two straightforward ways to time an calculation in Matlab.

Timing is best done in a coded function – otherwise most of the time will be spent while you type in commands and will not really tell you the actual CPU time (CPU = central processing unit = time spent on data manipulation and calculations)

- Simplest on command line

  tic; commands; toc

  The commands are best written on one line. Tic starts the clock and toc finishes it
  The 'elapsed time' is printed out in seconds (to nearest 0.01 of a second

- Timing within an m file – either a function or a script file

  Use the cputime

```
% at start of timing store current cpu time
t0 = cputime

…. then do the calculations

t1 = cputime;                    ← get final cputime at end

% print message

sprintf('Total time for calculation was  %g \n ', t1-t0)
```

  Some calculations are done so quickly (in less than 0.01 second) that this method would tell us that zero time was taken. If you really want to track down the time for such calculations you should repeat the calculations 100 times or so

The code for this would be

```
% at start of timing store current cpu time
t0 = cputime

for p = 1:100
….  then do the calculations
end

t1 = cputime;                    ← get final cputime at end

% print message

sprintf('Total time per calculation was  %g \n ', (t1-t0)/100 )
```