

MATLAB Training Course

Tuesday 19th January 2010

Course Leaders:

Dr Phil Robbins p.t.robbsins@bham.ac.uk x 47601
Professor Colin Thomas c.r.thomas@bham.ac.uk x 45355
School of Chemical Engineering

C&IT Contact:

Aslam Ghumra a.k.ghumra@bham.ac.uk x45877

Attendees:

Riadh Lebib	School of Psychology
Lisa Bishop	School of Biosciences
Johanna Gietl	School of Geography, Earth and Environmental Sciences
Zeshu Shao	School of Psychology
Andrew Southam	School of Biosciences
Anja Woiciechowsky	Division of Cancer Studies
Catherine Jex	School of Geography, Earth and Environmental Sciences
Samuel Coward	School of Biosciences
Jonathan Hunt	IT Services
Ulf Sommer	School of Biosciences

Contents:

	page
Learning Outcomes	2
Prerequisites	2
Syllabus and Schedule	2
Exercises	4
Example Solutions to Exercises	14
Appendix 1: Brief Guide to MATLAB	21
Appendix 2: Indexing and Loops	28

Learning outcomes:

At the end of the Course, the students will be able to

- (i) use MATLAB in the Command Window for simple tasks;
- (ii) use the Workspace and Command History Windows effectively;
- (iii) access Help;
- (iv) write script m-files for more complicated tasks;
- (v) handle vectors and matrices in MATLAB;
- (vi) handle polynomials in MATLAB;
- (vii) use loop structures, whilst understanding the advantages of vectorisation effectively;
- (viii) write and use function m-files and user-defined functions, and use function functions; and
- (ix) solve simple systems of ode's in MATLAB.

Pre-requisites:

Some knowledge of programming.

Understanding of vectors and matrices, polynomials and mathematical functions in general.

Syllabus and schedule:

Morning Session (9:30 am to 12:30 pm):

Part 1: General (15 min)

Students will practise using the Command Window to execute simple programming tasks, taking advantage of the Workspace and Command History Windows as necessary, and using the Help files. Built-in functions will be introduced and the students will practice their use on scalars.

Part 2: Vectors and matrices (15 min)

Students will create vectors and matrices, and solve simple sets of linear equations in the Command Window. The difference between array and matrix multiplication will be considered. The way MATLAB handles polynomials will be described, with relevant common functions discussed. The students will attempt simple programming exercises based on these functions.

Part 3: Script or m-files (45 min)

Students will learn to use the MATLAB Editor to create and run m-files. Looping structures in MATLAB and the advantages of vectorisation over loop structures in many instances will be described, using built-in functions. The students will write simple m-files to solve problems using these concepts and those of the previous section.

Break 10 min

Part 4: Plotting (20 min)

Students will learn to use MATLAB's 2D plotting capabilities, following a brief function m-files and user-defined functions demonstration. There will be a brief introduction to 3D plotting.

Part 5: Programming tasks (1 h 15 min)

Students will have time to develop their MATLAB skills by undertaking more programming tasks.

Afternoon Session (2 to 4:30 pm):

Part 6: Function m-files (30 min)

Students will learn to use function m-files, write user-defined functions, and function functions, using them to solve appropriate problems.

Part 7: Ordinary differential equations (50 min)

The students will learn how to use MATLAB to solve simple systems of ODE's, with mention of the various built in solvers. Students will write function m-files, write user-defined functions, and function functions, using them to solve appropriate problems.

Break 10 min

Part 8: Consolidation exercises (1 h)

Books:

There are many books on MATLAB, some fairly specific e.g. for Engineers. For first year Engineers we recommend:

Moore, H. (2007) MATLAB for Engineers. Pearson Prentice Hall.

Many people like:

Gilat, A. (2008) MATLAB: An Introduction with Applications. John Wiley and Sons

Examples are taken from these and other books e.g.

Etter, D. (1996) An Introduction to MATLAB for Engineers and Scientists Prentice Hall

Hahn, B.D. and Valentine, D.T. (2009) Essential MATLAB for Engineers and Scientists. Butterworth-Heinemann

Gustafsson, F. and Bergman, N. (2003) MATLAB for Engineers Explained. Springer-Verlag.

Check the MathWorks website www.mathworks.com for information on books, of a general nature and by specific discipline) and other useful stuff. Amazon lists a lot of popular MATLAB books too.

Exercises:

Note: MATLAB commands are shown in **bold** below, so they are easy to see, but you can just type them normally.

What you cannot complete in the time allotted you should attempt in the time set aside for “Programming tasks” and “Consolidation exercises”.

There are example solutions to the exercises given later. They may not always be best programming practice! Please send improvements to c.r.thomas@bham.ac.uk.

Part 1: General (15 min)

Demonstration (CRT): based on Appendix 1.

1. Mathematical operators in MATLAB are fairly obvious including $^$ for “raise to power”. The usual rules of operator precedence are followed. Try some simple mathematical expressions. Also try some of MATLAB's built-in functions at the Command Window e.g.

```
>> sqrt(4)
>> log(2)
>> log10(10)
```

Note that MATAB uses log for natural logs (\log_e).

If you go to Help - Product Help in the MATAB menus you can find all the functions in alphabetical order and by category. There is lots of help available. Help also includes video tutorials and demos. We strongly recommend that you find some time, now or in the next few weeks, to look at these.

2. Note that a semicolon (;) after a command suppresses the output, which is often useful if you are writing a programme or your variable is very large.

Try

```
>> A = magic(3)
```

and

```
>> B = magic(4);
```

Even though B is not printed in the Command Window, it is available in memory (the workspace) as can be seen in the Workspace Window.

3. You can clear the Command Window with the command

```
>> clc
```

This does not affect the variables stored in the workspace. Try it!

4. If you want to save your variables to disk for later use, use the **save** command e.g.

```
>> save mydata
```

which creates a file called mydata.mat that you can load later. (Of course you can use any legal filename instead of mydata.) If you use the command

```
>> save mydata A -ascii
```

you can get a text file readable by Word and Excel.

On the other hand, if you want to clear your functions and variables from memory

```
>> clear
```

is what you need. Try it before you type too much, and have to start again!

5. By now you should have made some mistakes. If not, make one or more on purpose. Notice how MATLAB gives useful information about errors of syntax, including its best guess of where the mistake lies. In correcting mistakes in commands, the up and down arrow keys $\uparrow \downarrow$ are useful for stepping through previous commands. You can also use the Command History Window to drag a previous command to the Command Window for editing, or you can double click on a command to run it again. Try out these operations; they are incredibly useful.

Note that MATLAB does not recalculate previous commands e.g. if you change the value of a variable. Thus (*display condensed to save space*):

```
>> a =5;
>> x = a^2
x =
    25
>> a =7;
>> x
x =
    25
>> x = a^2
x =
    49
```

6. MATLAB can handle strings (bits of text)

```
>> greeting = 'Hello, world.'
```

String arrays are created using the single quote delimiter '. A useful function is **disp**, which allows you to display text on the screen without the variable name being repeated. For example

```
>> disp('Hello, world').
```

Try your own messages.

7. A paper cup shaped as a frustum of a cone is designed to have a volume of 250 mL. The radius of the top of the cup is 1.25 times the radius of the base. Determine the top and bottom radii if the cup is 50 mm high.

Note:

$$V = \frac{1}{3}\pi h(R_1^2 + R_2^2 + R_1 R_2)$$

where V is the volume, h is the cup height, R_2 is the radius at the top, and R_1 is the radius at the bottom.

(Ans.: 44.2 mm, 35.4 mm)

Adapted from Gilat

Useful function: **sqrt**

8. Zeller's Congruence (http://en.wikipedia.org/wiki/Zeller's_congruence) can be used to compute the day of the week, given the date. For the Gregorian calendar, Zeller's congruence is

$$h = \left(q + \left\lfloor \frac{26(m+1)}{10} \right\rfloor + y + \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{c}{4} \right\rfloor - 2c \right) \bmod 7$$

where h is the day of the week (0 = Saturday, 1 = Sunday, 2 = Monday, ...), q is the day of the month, m is the month (3 = March, 4 = April, 5 = May, ..., 14 = February), y the year of the century (e.g. 95 for 1995), and c is the century *number* (e.g. in 1995 the century would be 19, even though it was the 20th century.) The square brackets indicate taking the integer part of the number enclosed.

Warning: January and February are counted as months 13 and 14 of the previous year, so, for example, January 2000 would be the 13th month of 1999!

Test this on today's date.

Useful functions: **floor**, **mod**

Part 2: Vectors and matrices (15 min)

MATLAB stands for MATrix LABoratory. It is built around matrices and their algebra. Demonstration (CRT): based on Appendix 1.

9. Create row vectors $a = [1 \ 3 \ 5 \ 7 \ 9]$ and $b = [2 \ 1.5 \ 1 \ 0.5 \ 0]$ using the colon operator.

Create the matrix $A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 1 \\ 1 & 1 & 0 \end{pmatrix}$.

10. Try

```
>> a*b
```

Why doesn't this work?

The transpose operator in MATLAB is a single quote '.

Try

```
>> b' and then
```

```
>> a*b' (which actually gives the dot or scalar product of the vectors).
```

11. Matrix and array multiplication. Compare $A*A$, A^2 , $A.*A$ and $A.^2$.

The fullstop here signifies an element by element operation, rather than matrix multiplication. This is often what you want in implementing scientific formulae.

12. Create some matrices using **ones**, **zeros**, **eye**, **rand**, **randn** to see how they work.

Check the meaning and syntax in the Help files. In each case use either 1 or 2 arguments. Note how MATLAB interprets the meaning.

Note that **eye(size(A))** is very useful for creating an identity matrix the same size as A, where A is a given matrix.

13. Try **sum**, **sort**, and **max** on A. Note these (and others) work on matrix columns. How would you find the maximum value in a matrix?

14. Construct a square matrix of integers of size 7x7. Try something like

```
>> A = round(10*rand(7))
```

Other possibilities instead of round are **floor**, **ceil** and **fix**.

What is the rank of A, and the value of its determinant? Try

```
>> rank(A)
```

```
>> det(A)
```

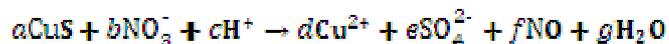
If the rank is not 7, make a new matrix A of rank 7 for exercise 15.

15. Create a column vector b of 7 rows. Try something like:

```
>> b = [0:-1:-6]'
```

Imagine your A matrix from exercise 14 contains the coefficients for 7 variables in 7 equations and b is the matrix of constants, so that $Ax = b$. Evaluate x using matrix division (“left division”), and by finding the inverse of A with **inv(A)**, and then finding **inv(A)*b**. The answers may look identical, but they are not exactly because of truncation errors during the calculations.

16. The reaction of copper sulphate and nitric acid in aqueous solution is described by the following chemical equation:



where a to g are the numbers of molecules participating in the reaction and are unknown. Balancing for each element gives:

$$a = d$$

$$a = e$$

$$b = f$$

$$3b = 4e + f + g$$

$$c = 2g$$

$$-b + c = 2d - 2e$$

There are 7 unknowns a to g and 6 equations. However, a to g must be positive integers, so a solution can be found. This is done by setting a = 1, solving the resulting matrix equation and seeing if the solution is positive integers. If not, a is set to 2, and the solution is repeated, and so on.

Find a to g with the lowest possible value of a. (*Hint: arrange the equations in matrix form, where the elements of the matrix are the appropriate coefficients of the unknowns. The Array Editor is useful for creating this sort of array.*)

(Ans.: 3, 8, 8, 3, 3, 8, 4)

Adapted from Gilat.

Useful functions: **zeros**

17. Vectorisation

MATLAB expressions and functions will often accept vectors and matrices as input arguments. For example

```
>> times = 0:0.1:1;  
>> temp = 5*times + 273.16  
temp =
```

Columns 1 through 8

```
273.1600 273.6600 274.1600 274.6600 275.1600 275.6600 276.1600 276.6600
```

Columns 9 through 11

```
277.1600 277.6600 278.1600
```

Using vectors in this way reduces the need for looping structures (although these are available, see later) and can make code much faster.

Repeat Exercise 7 but use a vector of h values from 50 to 100 mm in steps of 10 mm. Once you have defined the vector, you should be able to find the commands to calculate the radii in your Command History. Try the command to calculate R1; you should see an error message. Find out why this appears and correct the command so it works as you wish. (Hint: consider Exercise 11.)

18. Polynomials

In MATLAB, polynomials are represented by a row vector of the coefficients e.g. a polynomial $f = 2x^3 - 3x^2 + x - 1$ is specified by the coefficient vector

```
>> a = [2 -3 1 -1]
```

If $x = [1 \ -2]$, then **polyval**(a, x) = [-1 -31].

roots(a) gives the roots of the polynomial represented by the coefficient vector a.

poly(r) gives the coefficients of the polynomial with roots r.

There is a lot more to polynomials than this!

Find the roots of the polynomial $f = 3x^3 - 2x^2 + x - 1$.

(Ans.: 1.3982, 0.0509 + 0.5958i, 0.0509 - 0.5958i. Note that 2 roots are complex. i in MATLAB is used for $\sqrt{-1}$.)

Useful function: **roots**

Find the value of this polynomial at $x = 2, 5, 13$ and 14. Use vectorisation.

(Ans.: 5, 179, 3899, 4913)

Useful function: **polyval**

Part 3: Script or m-files (45 min)

Many MATLAB operations and functions can be used at the command line (in the Command Window). Commands can also be stored in text files to be executed later from the command line. Such a file has an extension .m and is therefore called an m-file.

MATLAB has a built-in editor.

A “script” m-file is equivalent executing a sequence of commands at the command line. You can store m. files for later use.

Demonstration (CRT). Also see Appendix 1.

#: anything on a line after # sign is a comment and will be ignored. You should comment your code extensively so it is easily understood and maintained.

19. Write an m-file that will convert degrees Fahrenheit to degrees Centigrade and produce a table like that below (on screen).

Note: $^{\circ}\text{C} = (^{\circ}\text{F} - 32) \times 5/9$

deg F	degrees C									
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	-17.78	-17.22	-16.67	-16.11	-15.56	-15.00	-14.44	-13.89	-13.33	-12.78
10	-12.22	-11.67	-11.11	-10.56	-10.00	-9.44	-8.89	-8.33	-7.78	-7.22
20	-6.67	-6.11	-5.56	-5.00	-4.44	-3.89	-3.33	-2.78	-2.22	-1.67
30	-1.11	-0.56	0.00	0.56	1.11	1.67	2.22	2.78	3.33	3.89
40	4.44	5.00	5.56	6.11	6.67	7.22	7.78	8.33	8.89	9.44
50	10.00	10.56	11.11	11.67	12.22	12.78	13.33	13.89	14.44	15.00
60	15.56	16.11	16.67	17.22	17.78	18.33	18.89	19.44	20.00	20.56
70	21.11	21.67	22.22	22.78	23.33	23.89	24.44	25.00	25.56	26.11
80	26.67	27.22	27.78	28.33	28.89	29.44	30.00	30.56	31.11	31.67
90	32.22	32.78	33.33	33.89	34.44	35.00	35.56	36.11	36.67	37.22
100	37.78	38.33	38.89	39.44	40.00	40.56	41.11	41.67	42.22	42.78
110	43.33	43.89	44.44	45.00	45.56	46.11	46.67	47.22	47.78	48.33
120	48.89	49.44	50.00	50.56	51.11	51.67	52.22	52.78	53.33	53.89
130	54.44	55.00	55.56	56.11	56.67	57.22	57.78	58.33	58.89	59.44
140	60.00	60.56	61.11	61.67	62.22	62.78	63.33	63.89	64.44	65.00
150	65.56	66.11	66.67	67.22	67.78	68.33	68.89	69.44	70.00	70.56
160	71.11	71.67	72.22	72.78	73.33	73.89	74.44	75.00	75.56	76.11
170	76.67	77.22	77.78	78.33	78.89	79.44	80.00	80.56	81.11	81.67
180	82.22	82.78	83.33	83.89	84.44	85.00	85.56	86.11	86.67	87.22
190	87.78	88.33	88.89	89.44	90.00	90.56	91.11	91.67	92.22	92.78
200	93.33	93.89	94.44	95.00	95.56	96.11	96.67	97.22	97.78	98.33
210	98.89	99.44	100.00	100.56	101.11	101.67	102.22	102.78	103.33	103.89

Useful functions: **reshape**, **blanks**, **disp**, **fprintf**

20. Loops:

Try out some simple flow control structures. **if**, **for**, **while** are most useful. See the Help files and/or Program Flow Control in Appendix 1 and Appendix 2.

Write scripts to do the following. Create an array of numbers [1 2 3 ... 99 100]

Create an array of numbers [2 3 4 ... 100 101]

Create an array of numbers [1 2 3 ... 48 49 50 102 104 ... 196 198 200]

Create an array containing the squares of 1 to 10

Given the formula $y = x^2 + 2x + 1$, calculate y for $x = 1\ 2\ 3\ \dots\ 8\ 9\ 10$

21. If you are feeling adventurous, write a program called `hi_lo.m`, which asks the user to discover a random integer between 0 and 100 chosen by the computer. Only 7 guesses are allowed. Use a **for** loop to execute the code for inputting and checking guesses up to 7 times (**break** will allow you to exit if a guess is successful). Use **if .. elseif .. else** for checking hi, lo and correct.

Useful functions: **input**, **int2str** (note also **num2str**, **double2str**, **str2double**)

Part 4: Plotting (20 min)

MATLAB has excellent plotting capabilities, which give you a very good control over your output.

Demonstration (CRT). Also see Appendix 1.

22. Plotting a single 2D curve in MATLAB:

If `xdata` and `ydata` contain corresponding pairs of data values, **plot**(`xdata`, `ydata`) will plot the `y` values against the `x` values. Try plotting some sine or cosine curves. For example, create a vector of `x` values from 0 to 2π , calculate a corresponding vector of `y` values, where `y = sin(x)`. Then try

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

Note that you could also do this by

```
>> plot(x,sin(x))
```

but you don't get the vector `y` to use later and the code is much harder to read.

Check out the options in Help. You should try to specify different styles and colours of lines and markers. Messing on is recommended!

The command **axis** allows you to specify the axis ranges. Try it!

By default, `plot` deletes existing lines and resets all axes when a new plot command is made. To add lines to an existing plot, use **hold on**. Don't forget to turn **hold off**!

You can annotate graphs with **title**, **xlabel**, **ylabel**, **text**, **plotedit**. There is access to plot editing tools from a button in the Figure Window.

Look at the plotting Demos (via Help) to get some ideas of what MATLAB can help you do.

23. Look in the Help files under 3-D Visualization to see the large number of possibilities here. Try a script file with these commands

```
x = -3:0.25:3;  
y = -3:0.25:3;  
[X Y] = meshgrid(x,y);  
Z = X.^2 + Y.^2;  
mesh(X, Y, Z)
```

and then try **meshc**, **contour** and **surf** instead of **mesh**.

Part 5: Programming tasks (1 h 20 min)

Complete all the Exercises you were not able to finish earlier. It might also be worth working through Appendix 1 to become more familiar with the syntax. Here are some additional exercises if you need them:

24. Use MATLAB to show that the sum of the infinite series $4 \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)}$ converges to π . Do this by using a loop to increase n in powers of 2 until the difference between the sum of the series and MATLAB's value of π differ by less than $1e-6$. Plot the differences as a function of n .

Adapted from Gilat.

Useful functions: **abs**, **semilogx**, **xlabel**, **ylabel**, **title**

25. If n is an integer > 1 then a sequence can be formed where the next number is $3n + 1$ if n is odd and $n/2$ if n is even (the hailstorm series). It is an unproven conjecture that whatever is the value of n , the sequence will always reach 1. Write a program to compute the sequence given a user input of n . If n is not greater than 1, an error message should be generated. Otherwise print the values to screen and show a plot of how the series reaches 1 (if it does).

Adapted from Gustafsson and Bergman.

Useful functions: **ceil**, **floor**, **rem**, **disp**, **plot**

Part 6: Function m-files (30 min)

A function m-file is a script file that takes arguments (inputs and outputs). MATLAB provides a lot of such files, and you can write your own.

Demonstration (CRT)

26. **humps(x)** is a function – it takes in values of x , and outputs values of y . Look at how this function file is constructed by entering

```
>> type humps
```

at the command line. Note in particular the structure of the first line:

```
function [out1,out2] = humps(x)
```

The word function identifies this as a function. Its name is humps, and it should be in an m-file called by the same name i.e. humps.m. The input argument is x , and the output arguments are out1 and out2 in a vector. The function must calculate or set values for all the output arguments, usually based on the value of the input argument.

The comment lines that follow the first line are what appear if a user types

>>help humps
at the command line:

```
%HUMPS A function used by QUADDEMO, ZERO DEMO and F PLOT DEMO.  
% Y = HUMPS(X) is a function with strong maxima near x = .3  
% and x = .9.  
%  
% [X,Y] = HUMPS(X) also returns X. With no input arguments,  
% HUMPS uses X = 0:.05:1.  
%  
% Example:  
% plot(humps)  
%  
% See QUADDEMO, ZERO DEMO and F PLOT DEMO.
```

This help describes what the function does, some points of syntax and often and example.

Try plotting **humps** as the help suggests. (The default $x = 0$ to 1 is a good range). Use the editor to make your own version of humps (“myhumps”?) that has at least one root in the range of x you are using. Store myhumps for later (Exercise 27). Plot myhumps on top of humps.

27. Function functions:

There are some functions in MATLAB which work on other functions. Try the function functions **fminbnd** and **quad** on myhumps (Exercise 26).

```
>> help funfun
```

gives general help, but you can use (for example)

```
>> help fzero
```

and so on, as usual for specific help.

You usually have to pass a function to another function as a “handle” or as a string in quotes e.g.

```
>> fminbnd(@humps,0,1)
```

```
ans =
```

```
0.6370
```

28. User m-files:

Start the editor, and write a function m-file to implement the algorithm for determining a square root, which says that if y is a guess of the square root of x , then a better guess is the mean of y and x/y is a better guess. The syntax should be $y = \text{mysqrt}(x, y0, \text{tol})$ where the iterations terminate when 2 successive guesses are closer than tol .

Note that all variables inside a function are “local”. This means that the bits of the program outside the function don’t have access to the values of any variables other than the arguments of the function (input, output variables). This means that you cannot accidentally override the value of a variable in the Workspace. However, it does mean

you should pass in all the variables you need as input arguments. If there are a lot, consider a “structure”. See the Help files for more information.

Part 7: Ordinary differential equations (50 min)

Materials to be supplied separately.

Part 8: Consolidation exercises (1 h)

Complete the exercises you have not managed to finish and ask for more if you need them.

Example Solutions to Exercises:

Exercise 7:

```
>> V = 250*1E-6; % m^3
>> h = 0.05 ; % m
>> R1 = sqrt(3*V/pi/h/(1+1.25^2+1.25)) % m
R1 =
    0.0354
>> R2 = R1*1.25 % m
R2 =
    0.0442
```

(Ans.: 44.2 mm, 35.4 mm)

Exercise 8:

```
>> q = 19;
>> m = 13;
>> y = 9;
>> c = 20;
>> h = (q+floor(2.6*(m+1))+y+floor(y/4)+floor(c/4)-2*c);
>> h = mod(h,7)
h =
     3
```

(Ans: Tuesday)

Exercise 10: for matrix multiplication, the inner dimensions must agree.

Exercise 13: for matrix A,

```
>> max(max(A))
```

will give the maximum element.

Exercise 16:

```
>> A = zeros(7);
A then produced in the Array Editor by changing individual value giving:
>> A
A =
     1     0     0    -1     0     0     0
     1     0     0     0    -1     0     0
     0     1     0     0     0    -1     0
     0     3     0     0    -4    -1    -1
     0     0     1     0     0     0    -2
     0    -1     1    -2     2     0     0
     1     0     0     0     0     0     0
```

Condensed:

```
>> b = [0 0 0 0 0 1]';           >> a_to_g = A\b
% Notice transpose here.

b =                                a_to_g =
  0                                1.0000
  0                                2.6667
  0                                2.6667
  0                                1.0000
  0                                1.0000
  0                                2.6667
  1                                1.3333
```

```
>> b(end) = 2;                   >> a_to_g = A\b

b =                                a_to_g =
  0                                2.0000
  0                                5.3333
  0                                5.3333
  0                                2.0000
  0                                2.0000
  0                                5.3333
  2                                2.6667
```

```
>> b(end) = 3;                   >> a_to_g = A\b

a_to_g =
  3.0000
  8.0000
  8.0000
  3.0000
  3.0000
  8.0000
  4.0000
```

All integers as required.

Note: could have inferred the solution by looking at the first set of values for *a* to *g*. However, it would also be possible to write code to give the desired solution.

Exercise 17:

```
>> h = (50:10:100)/1000;
>> R1 = sqrt(3*V/pi/h/(1+1.25^2+1.25))
??? Error using → mrdivide
Matrix dimensions must agree.
>> R1 = sqrt(3*V/pi./h/(1+1.25^2+1.25))
R1 =
```

```

    0.0354  0.0323  0.0299  0.0280  0.0264  0.0250
>> R2 = R1*1.25
R2 =
    0.0442  0.0404  0.0374  0.0350  0.0330  0.0313

```

Exercise 18:

```

>> a = [2 -3 1 -1];
>> x = [2 5 13 14];
>> polyval(a,x)
ans =
     5     179    3899    4913

```

Exercise 19:

```

% This could really do with comments. As you work out what is going on here,
% why not comment the code yourself?
clc
deg_F = 0:219;
deg_C = (deg_F - 32)*5/9;
C_table = reshape(deg_C, 10, 22)';
ext_C_table = [(0:21)'*10 C_table];
header_1 = [blanks(32) 'degrees C'];
header_2 = ['deg F ' ' +0' blanks(5) '+1' blanks(5) '+2' ...
    blanks(5) '+3' blanks(5) '+4' blanks(5) '+5' blanks(5) '+6' ...
    blanks(5) '+7' blanks(5) '+8' blanks(5) '+9'];
disp(header_1)
disp(header_2)
fprintf('%-6.0f %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f\n',
ext_C_table')

```

Exercise 20:

The “short answers” in the table below are deliberately as compact as possible – please ask if they’re not clear. Some of them use “vectorisation” which is a powerful way in MATLAB to speed up calculations and make the code shorter. Instead of dealing with values of a variable one at a time (say in a loop), you can often feed a vector or matrix of values into a function, and get out a corresponding vector or matrix of answers.

Q	Long answer	Short answer
1	for n = 1:100 x(n) = n; end disp(x)	x = 1:100;
2	for n = 1:100 x(n) = n+1; end disp(x)	x = 2:101; or x = [1:100] + 1;

3	<pre> for n = 1:100 if n <= 50, x(n) = n; else x(n) = n*2; end end </pre>	<pre> x = [1:50 102:2:200]; </pre>
4	<pre> for n=1:10 x(n) = n^2; end </pre>	<pre> x = [1:10].^2; </pre> <p>Note the “.^2” means “square each element individually”. (1:10)^2 tries do a matrix squaring on it which fails as it’s the wrong shape.</p>
5	<pre> for n = 1:10 x(n) = n^2 + 2*n + 1; end </pre>	<pre> x = 1:10; x = x.^2 + 2*x + 1; </pre> <p>Note “x =” means “make the new value of x equal to whatever the answer calculated on the right is”</p>

Exercise 21:

```

%HI_LO plays the game with this name.
% When prompted enter your guess, the computer
% will tell you if your guess is above or below
% the random number calculated by the computer.

```

```

x = fix(100*rand);
n = 7;
test = 1;
for k = 1:7
    numb = int2str(n);
    disp([ 'You have right to ' numb ' guesses' ])
    disp([ 'A guess is a number between 0 and 100'])
    guess = input('Enter your guess ')
    if guess < x
        disp('Low')
    elseif guess > x
        disp('High')
    else
        disp('You won')
        test = 0;
        break
    end
    n = n - 1;
end
if test > 0

```

```
disp('You lost')
end
```

Exercise 24:

```
%SUMMING_TO_PI sums a series supposed to converge to pi for increasingly
% large numbers of terms and calculates the difference between the sum and
% MATLAB's value of pi, plotting the successive estimates.
```

```
clc
clear
```

```
difference_OK = false; % Initialise a flag for success.
number_terms = 2; % number of terms in the partial sum.
index = 0; % A counter for plotting.
while ~difference_OK
    index = index + 1; % Increment index on every loop.
    n_values = 0:number_terms;
    elements_of_sum = (-1).^n_values./(2*n_values+1);
    % Array calculation here.

    % Store values of the pi estimates in a vector.
    pi_estimate(index) = 4*sum(elements_of_sum);
    difference = pi-pi_estimate(index);
    if abs(difference) < 1e-6
        difference_OK = true; % pi and estimate are close enough.
    end
    % Store the number of terms in a vector for plotting.
    number_of_terms(index) = number_terms;
    % Double the number of terms in the partial sum
    % and try again.
    number_terms = 2*number_terms;
end
semilogx(number_of_terms,pi_estimate)
xlabel('Number of terms')
ylabel('{\pi} estimate')
% Note use of LaTeX here.
title('Summing a series for {\pi}.')
```

Exercise 25:

```
%HAILSTORM1
% Takes n (integer>1) and produces a sequence of numbers. If a number in
% the sequence is odd, the next number is 3*number+1, if even, number/2.
% Unproven conjecture that the sequence always reaches 1.

% CRThomas (2009)
```

```

clc
n = input('What is the value of your integer? ');
if n/ceil(n)<1 % Check whole number (NB will be type float)
    disp('Input should be a whole number ')
    disp(['but processing floor(input) anyway i.e. ' num2str(floor(n)) char(10)])
    n = floor(n);
    if n < 1
        n = 1;
    end
end
end

k = 1;
a(k) = n;

while n > 1
    k = k+1;
    if rem(n,2)==0
        n = n/2;
    else
        n = 3*n+1;
    end
    a(k) = n;
end

plot(a)
disp(a)

```

Exercise 28:

```

function y = mysqrt(x,y0,tol)
%MYSQRT uses an iterative algorithm to find the square root of a scalar x,
% starting with initial guess y0, to a tolerance between successive answers
% in the iterations of tol.

```

% Should be modified to trap error messages and to fail gracefully.

% CR Thomas (2010)

% Check there are sufficient input arguments or set defaults:

```

if nargin == 1||2 % 1 or 2 input arguments need tol.

```

```

    tol = 1e-6; % default tolerance.

```

```

end

```

```

if nargin == 1 % also need y0.

```

```

    y0 = 1; % default initial value.

```

```

end

```

```

if nargin == 0

```

```

    disp('Need at least one input argument')

```

```
    return % terminates execution of function.
end

% This function does not accept a vector or matrix for x. Check x is a
% scalar
if ~isscalar(x)
    disp('This function only accepts scalar input')
    return % terminates execution of function.
end

estimate = y0;
current_value = 0;
while abs(estimate - current_value)>tol
    current_value = estimate;
    estimate = (current_value + x/current_value)/2;
end
y = current_value;
```


Generating Matrices

e.g. `A = zeros(4, 3)`; generates a matrix with 4 rows and 3 columns of zeros.

`A = ones(3, 2)`; generates a matrix with 3 rows and 2 columns of ones.

`A = eye(3)`; generates a square identity matrix of order 3.

You can set elements of matrices individually e.g. `A(3,2) = 2`; sets the element on the third row, in the second column to 2.

User Input and Display

e.g. `s = input('Enter values for s in brackets: ');` displays the message in quotes on the screen and accepts values for s enclosed in brackets e.g a response of `[1 4 2 6]` gives `s = [1 4 2 6]`.

A variable can be displayed without its name using `disp` e.g. `disp(s)` shows “1 4 2 6” on the screen. See `fprintf` for more control over the output.

Plots

e.g. `plot(x, y)` produces a graph of points of y against x, where x and y are vectors. See the help files for how to make your plots beautiful.

Array and Matrix Operations

Array operations are performed element by element

e.g. (if A and B are matrices of the same size) `A.*B` multiplies each element of A by the corresponding element in B.

On the other hand `A*B` performs the matrix multiplication of A and B.

Similarly, `B = A.^2` squares every element of A, whereas `B = A^2` means `B = A*A`.

Matrix division: if `AX = B`, then X can be found by “left division”. `X = A\B`.

(MATLAB uses Gaussian elimination for this process.)

`det(A)` gives the determinant of A.

`inv(A)` gives the inverse of A.

If you see a message like:

??? Error using ==> /

Matrix dimensions must agree.

you may be using matrix algebra when you don't mean to.

MATLAB Functions

MATLAB has thousands of functions. Some useful ones are:

abs(x)	absolute value of x	sin(x)	sine of x
sqrt(x)	square root of x	cos(x)	cosine of x
round(x)	rounds to the nearest integer	tan(x)	tangent of x
exp(x)	computes e^x	sinh(x)	hyperbolic sine of x
log(x)	$\log_e(x)$	cosh(x)	hyperbolic cosine of x
log10(x)	$\log_{10}(x)$	tanh(x)	hyperbolic tangent of x

Note x does not have to be a scalar; it can be a vector or other matrix. The output is then the same size and shape as x.

Look up these and other functions in the help files. Be very careful with log() which means the same as LN() in Excel.

Polynomials

polyval(a, x) evaluates a polynomial with coefficients given in the vector a for the values in x e.g. a polynomial $f = 3x^3 - x^2 - 1$ is specified by the coefficient vector $a = [3 \ -1 \ 0 \ -1]$. If $x = [2 \ 1]$, then $\text{polyval}(a, x) = [19 \ 1]$.

Coefficient matrices can be added, subtracted and multiplied by a scalar.

To multiply two polynomials, their coefficient vectors are “convoluted”. Division is by deconvolution. If f is as above with coefficient matrix $a = [3 \ -1 \ 0 \ -1]$ and $g = x^3 - x + 3$ with coefficients given by $b = [1 \ 0 \ -1 \ 3]$, then the coefficients of the product of g and f are given by $\text{conv}(a,b) = [3 \ -1 \ -3 \ 9 \ -3 \ 1 \ -3]$.

roots(a) gives the roots of the polynomial represented by the coefficient vector a.

poly(r) gives the coefficients of the polynomial with roots r.

Logical Operators

If = is for assignments, what is used for “equals”? This is one of the logical operators.

<	less than	~=	not equal
<=	less than or equal	&	and
>	greater than		or
>=	greater than or equal	~	not
==	equal		

These are particularly useful in branch and loop decisions. See below.

Symbolic Maths

To define a symbolic object, use sym or syms e.g.

sym('x') or

sym x y t n

Define a symbolic function e.g. $f = 3*x + 2*y + t^2$.

$g1 = \text{subs}(f, x, 2)$ substitutes 2 for x in f to produce g1. NB f is not affected by subs unless you write $f = \text{subs}(f, x, 2)$.

Differentiate:

If $g = \cos(3*x)$

$g2 = \text{diff}(g)$ gives $g2 = -3*\sin(3*x)$.

$g3 = \text{diff}(f, t)$ differentiates f with respect to t.

Limits:

$\text{limit}((1 + x/n)^n, n, \text{inf})$ finds the limit of $(1 + x/n)^n$ as $n \rightarrow \infty$, giving $\exp(x)$.

Integrate:

$\text{int}(x^n)$ gives $x^{(n+1)}/(n+1)$ as expected; an indefinite integral.

$\text{int}(\sin(3*x), 0, \text{pi}/2)$ gives the value of the definite integral $\int_0^{\pi/2} \sin(3x) dx$ i.e. $\frac{1}{3}$

Plotting:

If have a function such as $f = 3*x^2 + 2*x + 2$,

$\text{ezplot}(f)$ will plot the function for the range $-\pi \leq x \leq \pi$. You can change the range and many other details of the plot.

This would also work for $f = 3*t^2 + 2*t + 2$.

The function `findsym` can be used to find out the “free variable” that is used by MATLAB to differentiate or integrate with respect to, unless you tell it otherwise.

Other stuff:

Look at `collect`, `expand`, `simplify`, `simple` and `pretty` and for ways to simplify and tidy up expressions.

Use `double` to get a double precision value from a symbolic number.

e.g.

$\text{sym}(1/2) + \text{sym}(3/5)$ gives 11/10 (symbolic).

$\text{double}(\text{sym}(1/2) + \text{sym}(3/5))$ gives 1.1000 (double precision).

`vpa` (variable precision arithmetic) can be used to get more digits e.g.

$\text{pi} = \text{sym}('pi')$; makes pi symbolic object.

$\text{vpa}(\text{pi}, 20)$ gives the value of π to 20 digits i.e. 3.1415926535897932385

Solving equations:

$\text{solve}(f)$ tries to find values of the symbolic variable in f for which f is zero.

For example

```
syms a b c x
f = a*x^2 + b*x + c;
solve(f)
gives
ans =
```

$$\frac{1}{2}a^{1/2}(-b+(b^2-4ac)^{1/2})$$
$$\frac{1}{2}a^{1/2}(-b-(b^2-4ac)^{1/2})$$

If $f \neq 0$, use quotes e.g. `g3 = solve('cos(2*x) + sin(x) = 1')`¹ gives

```
g3 =
    pi
     0
 1/6*pi
 5/6*pi
```

You can solve simultaneous equations e.g

```
[x y] = solve('x + y = 3','x - y = 1')
```

giving

```
x =
```

```
2
```

```
y =
```

```
1
```

Symbolic solutions to differential equations:

e.g. `dsolve('Dx = 1 + t')` gives

```
ans =
```

$$\frac{1}{2}t^2+t+C1$$

Note (a) D stands for the first derivative; (b) the integration is assumed to be with respect to t, but you can change this, and (c) the constant of integration C1.

You can specify an initial condition e.g. `dsolve('Dy = 1 + t', 'y(0) = 1')` gives

```
ans =
```

$$\frac{1}{2}t^2+t+1$$

You can also try to solve higher order equations this way.

Command Line and m-Files

Many MATLAB operations and functions can be used at the command line (in the Command Window). Commands can also be stored in text files to be executed later from the command line. Such a file has an extension .m and is therefore called an m-file. MATLAB has a built-in editor.

A “script” m-file is equivalent executing a sequence of commands at the command line.

¹ From the MATLAB Help files

A “function” m-file specifies input and output variable values; all other variables are “local”, having meaning only inside the function e.g. imagine a function called isprime. This might be in an m-file called isprime.m. This could contain the lines:

```
function p = isprime(n)
% p = isprime(n)
% p == 1 if n is prime; 0 otherwise
p = 1; % initialize p
for k = 2:sqrt(n)
    if rem(n, k) == 0
        p = 0;
    end
end
```

Anything on a line after a % is a “comment” and is ignored.

You could use this in a short segment of code like this, in order to test if a number is a prime:

```
N = input('Input a positive integer: ');
if isprime(N)
    disp([num2str(N), ' is a prime number.'])
else
    disp([num2str(N), ' is not a prime number.'])
end
```

num2str(N) turns a number into string for display.

Program Flow Control

One advantage of m-files is the possibility of program flow control, so the program can make decisions on the basis of variable values.

```
if
e.g. if <logical condition>
    <statements for first case>
    elseif <logical condition>
    <statements for second case >
    else
    <otherwise>
end
```

If the first logical condition is true, the first group of statements is executed, and then the program jumps to end. If it is not true, the second logical condition is tested, and if true, the second group of statements is executed, and the program jumps to end. Otherwise the third set of statements will be executed.

It is not necessary to have elseif and/or else to use an if statement.

If you have many cases to test, try switch.

for and while

If you want something to repeat, use for or while

```
e.g. for n = 1:1000;
      if isprime (n)
      disp(num2str(n))
      end
    end
```

All the statements are executed 1000 times.

Note that it is often possible to replace for loops with a vector approach, which is much faster.

If you only want to loop until some condition is reached, use while e.g.

```
n = 1;
while n <= 500
  disp(num2str(n))
  n = n^2+ 1;
end
```

Appendix 2: Indexing and Loops:

RM Ward/CR Thomas

We mostly use MATLAB script files and functions to do repetitive and/or complex tasks. (If it was a one-off and easy we'd do it by hand.) This typically involves using groups of numbers which we work through one after the other.

We normally store groups of numbers – measurements, calculated results, vectors, matrices etc. – in arrays. You have already looked at arrays in MATLAB, and their use as matrices.

As an example, we might have the first 7 Fibonacci numbers² in an array or matrix of one row (sometimes called a row vector): `fib = [1 1 2 3 5 8 13]`. Perhaps we want to calculate the 8th and 9th terms. However, before we can look at ways to do this, we need to know about "indexing".

If you have an array or matrix, you can refer to its elements using "indexing". For example, if

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

then in MATLAB you can find the value of the element in the 2nd row, and 3rd column

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & \textcircled{1} \\ 0 & 1 & 1 \end{bmatrix}$$

← 2nd row
↑
3rd column

by the command:

```
>> A(2,3)
```

The indices are row number first, then column number, starting in the top left position. You can use the colon operator to refer to a whole row or column e.g.

```
>> A(:,3)
```

means all the rows of column 3, and returns the column vector $\begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$, whilst

```
>> A(2,:)
```

means row 2, all the columns, and returns the row vector `[4 0 1]`.

Note that

```
>> my_variable = A(2,3)
```

² Fibonacci numbers are formed by adding the two previous numbers in the series that begins 1,1 and continues 2,3,5,8,13,21,

would set the value of my_variable to 1, should you want to use or change the value in some way, without changing A. Then, if you executed the command

```
>>my_variable + 1
```

at the Command Line, and the value of my_variable would become 2, whilst the value of A(2,3) would remain 1.

You can use indexing to set the value of an element, or even a group of elements in a matrix or vector e.g.

```
>> A(2,3) = 2 changes A to give A =  $\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 2 \\ 0 & 1 & 1 \end{bmatrix}$ , and  
>> A(:,3) = [0 0 0]'
```

note the transpose here to make the row vector into a column vector

```
A =  $\begin{bmatrix} 1 & 3 & 0 \\ 4 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ .
```

gives

Finally, you can also use a variable as an index, usually in a loop. For example, if you ran a script file with the following code in it:

```
m = 3;  
n = m - 1;  
my_variable = A(m, n)
```

then the value of my_variable would be set to 1.

Returning to the variable fib defined earlier

```
fib = [1 1 2 3 5 8 13]
```

we can see that the value of fib(1) is 1, whilst fib(7) is 13.

You should now be able to see how we could find fib(8) and fib(9) i.e.

```
>>fib(8) = fib(7) + fib(6)  
>>fib(9) = fib(8) + fib(7)
```

If we do this MATLAB just increases the length of (the vector) fib for us by adding the new term on its end.

This is fine, but long winded if we want to do it for the next 100 values. Instead we'd typically use variables – e.g. n – as our subscripts (pointers to position in the array) rather than numbers. So

```
>>n = 8;  
>>fib(n) = fib(n-1) + fib(n-2)  
>>n = 9;  
>>fib(n) = fib(n-1) + fib(n-2)
```

The neat thing about this is that the line "fib(n) = fib(n-1) + fib(n-2)" is the same both times – only n changes – so we can easily repeat it using a for loop in a script file e.g.

```
for n = 8:9
```

```
fib(n) = fib(n-1) + fib(n-2)
end
```

In fact we could generate more or less as many elements as we like e.g. to get 100 values:

```
fib(1) = 1;
fib(2) = 1;
these lines "initialise" the process i.e. they give the values we need to get started
for n = 3:100
    fib(n) = fib(n-1) + fib(n-2)
end
```

Note that MATLAB is not C++, and in MATLAB it is considered bad practice to use *i* as an index or "loop counter"; *i* is usually reserved to mean $\sqrt{-1}$ for when you are dealing with complex numbers. If you are only going to use real numbers, you could redefine *i* as a variable, but in any case it may be best to use a meaningful name such as *index* or *fib_index*. Although such a name is a pain to write, it makes the code much easier to read and understand.

Another commonly used bit of code in MATLAB is the 'conditional' statement, assuming you already have a value for *n* and want to test if it matches a condition such as "greater than" (>), "is equal to" (==), or "is less than" (<).

```
if n > 100,
    disp('n is greater than 100')
end
or
if n > 100,
    disp('n is greater than 100')
else
    disp('n is less than or equal to 100')
end
```

We can combine these ideas e.g.

```
for num = 1:100
    if num <= 50
        my_var(num) = num;
    else
        my_var(num) = 3*num;
    end
end
```

Finally, if you are using a loop to add terms one at a time to an array like *fib*, don't forget to use those pesky semicolons. Otherwise, MATLAB will write new values of the variable (to the Command Window) every time the program goes around the loop. Messy! Let the loop finish, and if you want to see the result, put a line like

```
my_var
```

into your code, which just prints the value of *my_var* to the screen, or use *disp*.