MATHEMATICAL METHODS FOR ENGINEERS 1 (MATH 1063)

CALCULUS 1 (MATH 1054)

MATLAB Practicals

Kuva Jacobs and Yuliya Tugai

School of Information Technology & Mathematical Sciences

University of South Australia

February 2008, February 2013 (revised)

Table of Contents

Practical 1: Introduction to MATLAB …………………………………………….. 1

Practical 2: Variables, Script M-Files and Graphing ……………………….….. 7

Practical 3: Arrays ..……………………………………………………………… 13

Practical 4: `For`- and `While`-Loops, `If`-Statements ………………………….. 19

Practical 5: Function and Script M-Files …………………………………………… 27

Practical 6: Polynomial Functions and Symbolic Toolbox ……………………….. 35

Practical 7: Symbolic Toolbox (Cont'd) ……………………………………..... 41

Appendix A : MATLAB Layout Overview …………………………………………… 47

Appendix B : Summary of Tables ………..………………………………………. 51

Practical 1: Introduction to MATLAB

Aim of this Practical

1. Get a general understanding of the purpose of MATLAB.
2. Understand the MATLAB Workspace
    - a) start up MATLAB
    - b) type commands in main window
    - c) change current directory
3. Use MATLAB as a calculator
    - a) perform some arithmetic calculations
    - b) understand the importance of operators, and functions
    - c) use MATLAB's help files
    - d) use functions like $\sin x$, $\cos x$, or $|x|$ to solve problems

Working Through This Practical

- Go through this practical at your own pace. It explains the environment in more detail than the following practicals.
- Do each exercise or activity in the shaded areas.
- Ask your practical supervisor if you have any questions or need advice.
- MATLAB code will always be denoted by the `courier` font.
- If you don't finish during the scheduled practical time, it is your responsibility to finish it before your practical next week.
- An arrow > at the start of the line in an exercise indicates an activity for you to complete.

What is MATLAB?

MATLAB is a powerful mathematical software package used by both engineers and mathematicians in the workplace. The name stands for MATrix LABoratory. Originally designed for manipulating matrices, MATLAB is great for many different mathematical tasks.

Why do I have to learn MATLAB?

Some good reasons for learning MATLAB include

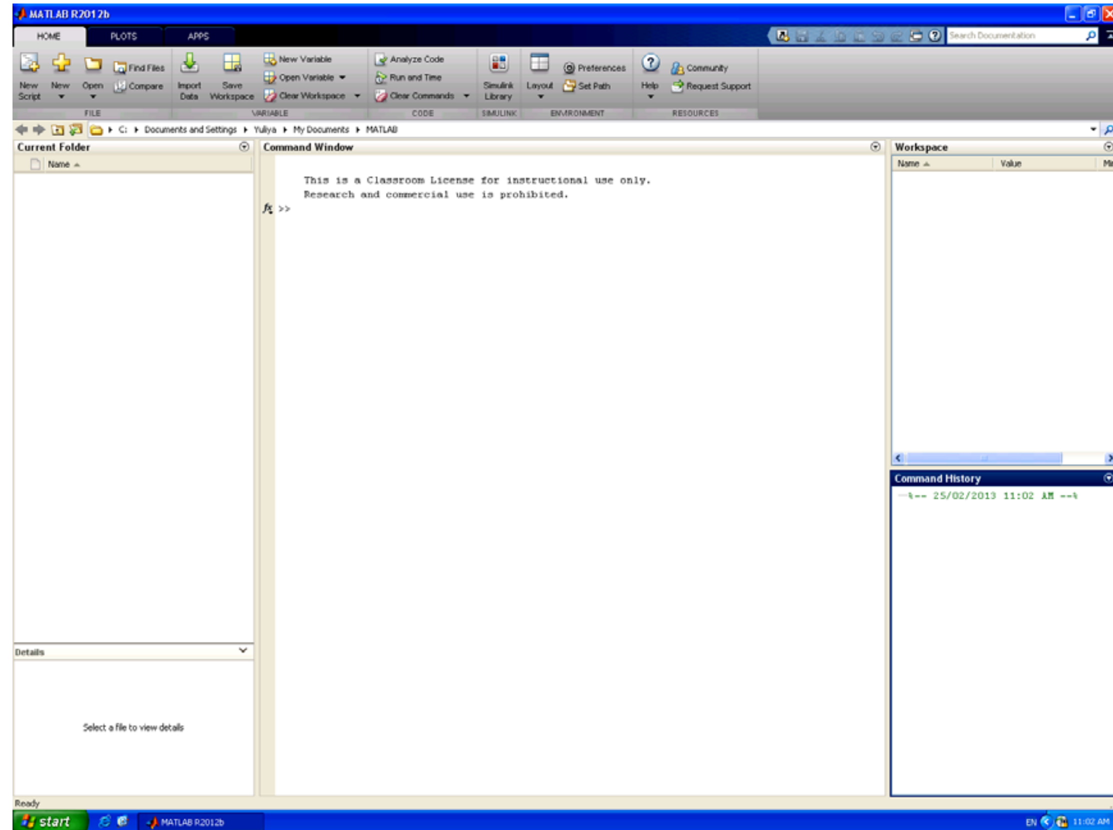
- It will allow you to check your assignment/homework calculations
- You will gain a better insight into mathematical functions through complex graphing that can't easily be done by hand
- You can quickly find solutions to problems that take a long time by hand (like finding the determinant of a 5x5 matrix)
- Later, you will be able to find approximate numerical solutions to problems that can't be solved by hand (like complicated integrals)

Getting Started in MATLAB

***Availability**:* MATLAB is available in the following PC Labs at Mawson Lakes : A2-23, GP1-06, F1-13, F1-15, F1-17, P1-12, P1-13, P1-15, MLAV, MLAV03.
It is not available in the barns.
For further availability check http://w3.unisa.edu.au/ists/students/pools/default.asp under *software and hardware search.* You will need to type MATLAB in Software search box. You can also check the current PC availability under *find a computer available for use.*

Activity 1 Opening MATLAB

>Locate the folder on your desktop with format *"<room name> Applications",* for example *"GP1-06 Applications"*, and open it to find the MATLAB folder and icon. Double click on the icon to launch MATLAB. You should now see the MATLAB desktop layout as shown below.



***The MATLAB Desktop Layout* (please refer to Appendix A for details)**

The MATLAB layout is divided into 4 windows (white area):

a) *Command Window* (centre), where you will type in all commands after the double arrow ">>"

b) *Command History* (bottom right), showing a history of commands in the order you typed them.

c) *Workspace* (top right), which will show your current variables. We will come back to this when we introduce variables.

d) *Current Folder* (left) has a toolbar with your current directory shown. All your work will be saved in this directory.

At the top of the screen in the grey coloured area you will see tabs *Home, Plots* and *Apps*. When you start MATLAB it shows all menu items (modes) in *Home* tab grouped by their function. At the bottom of grey coloured area you will see names of the functional groups for a particular tab.

Activity 2 Understanding the Desktop Layout

> Change your MATLAB C*urrent Folder* to your "My Documents" folder or to a folder on your USB drive. The folder you choose will be where your MATLAB documents are saved.

> To understand better the purpose of the three windows, type in the simple command **`3+5`** in the *Command Window* and press **enter**.
> What happened in the *Command, Command History* and *Workspace Windows*?

MATLAB as a Calculator

MATLAB can be used as a powerful calculator. The following exercises demonstrate the order in which MATLAB evaluates expressions.

Exercise: Calculations in MATLAB

The following commands illustrate the order of priority when evaluating expressions.
> Before typing each command into your *Command Window*, think about what you expect the answer to be.
> If the output differs from your expected answer, figure out why.
> Type in the following commands, pressing E**nter** after each one.

```
5*2+3
3+5*2
```

> What if instead, we want to add 3 to 5 and then multiply the result by 2? We can use brackets to change the order of evaluation as MATLAB will evaluate the contents of each bracket pair first.
> Type in the following line of code:

```
(3+5)*2
```

Coding tip: Many students have troubles matching bracket pairs. Experienced MATLAB *users reduce these errors by*
1. typing both brackets ( ),
2. then using the back arrow key ← to move back and fill in the bracket contents (3+5).

> The following code is not valid. Type it in, press Enter, then read the error message and change the code so that it works (note the vertical line pointing to the first error).

```
2(3-5)
```

Coding Tip: The up-arrow key ↑ on the keyboard will recall a previous line for you to edit.

> Type in the following lines to figure out what the hat ^ key represents.

```
3^2
3^3
```

> Figure out the answers you would expect for the following two lines of code and then type them in to verify your answer.

```
9-3^2
81/3^2
```

As we have just seen, the simplest mathematical operators are represented by the following symbols.

Symbol	Operator	Order of Priority	
^	Raise to a power	1	evaluated first
*	Multiplication	2	evaluated after any powers
/	Division	2	
+	Addition	3	evaluated after any powers, multiplication and division.
-	Subtraction	3	

Order of Evaluation

Operator Order of Priority

- The power operator has the highest order of priority.
- Multiplication and division both have equal 2nd order of priority.
- Addition and subtraction both have equal 3rd order of priority.

'Left-to-right' rule

Expressions are evaluated from left to right: the leftmost operation is performed first and the rightmost operation is performed last. *However, note that the operator order of priority supersedes the above left-to-right rule.*

Brackets

Round brackets, ( ), can be used to change the order of operations: any expression enclosed in brackets will be calculated first

Exercise: Exploring the Effect of Brackets

> Type the following arithmetic expressions in the *Command Window*.
> Mentally calculate the answers before pressing the **Enter** key.

```
(8+6)/(10-2^3)
8+6/(10-2^3)
8+6/10-2^3
```

Coding Tip: The up-arrow ↑ on the keyboard will recall a previous line for you to edit.

Exercise: Minimising Required Brackets

The following expressions both have a value of 12.
> Type in the equivalent MATLAB expressions to obtain the correct answer using as few pairs of brackets as possible.

$$\frac{17}{2/3+3/4} \quad \textit{(needs 1 pair of brackets)}$$

$$5+\frac{9^{3/2}+1}{2(5-3)} \quad \textit{(needs 3 or 4 pairs of brackets)}$$

Elementary Mathematical Functions

MATLAB has a number of built-in functions, such as square root, sine, cosine, exponential, etc, functions. The following table gives a list of some commonly used functions. Later you will also write your own functions.

`round(x)`	Rounds x to nearest integer
`floor(x)`	Rounds x down to nearest integer
`ceil(x)`	Rounds x up to nearest integer
`rem(y,x)`	Remainder after dividing y by x (*eg remainder of 17/3 is 2)*
`sign(x)`	Returns -1 if $x<0$; returns 0 if $x=0$; returns 1 if $x>0$
`rand or rand(1)`	Generates a random number between 0 and 1
`exp(x)`	Exponential function, e^x
`log(x)`	Natural logarithm function, $y=\ln x$ (where $e^y=x$)
`sqrt(x)`	Square root function, $\sqrt{x}$
`abs(x)`	Absolute value function, $\lvert x\rvert$
`sin(x)`	Sine function, $\sin x$
`cos(x)`	Cosine function, $\cos x$
`tan(x)`	Tangent function, $\tan x$

Exercise: Using Mathematical Functions

> Let x = 1.2. Verify that the following expressions have the values shown, by typing in the correct MATLAB commands.

Mathematical Expression	Answer	MATLAB command/Tips
$\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}$	0.1942	`1/sqrt(2*pi)*exp(-0.5*1.2^2)`
$\frac{\sqrt[3]{5+\cos 4x}}{\lvert\sin 3x\rvert}$	3.8866	`(5+cos(4*1.2))^(1/3)/abs(sin(3*1.2))`
$\sin^2(\pi x)$	0.3455	*Sets of brackets required: 1*
$\frac{e^{\sin x}}{\sqrt{x^2+1}}$	1.6259	*Sets of brackets required: 3*
$x\arctan x-\frac{1}{2}\ln(1+x^2)$	0.6053	*Note: you will need to look up the arctangent function in the* MATLAB *help files (click on* Help *at the top menu). Sets of brackets required: 3*

Exercise: Kiwi Change

You go to the supermarket only to find you left your wallet at home. After hunting on the floor of your rather messy car, you find 95c and decide to buy some kiwi fruit, which are 35c each.

> Use the `floor` and `rem` functions to find out the answers to the following questions:

How many kiwi fruit can you buy?

How much change do you have left?

MATLAB/Coding Terms you should become familiar with
Command, Command Window, Command History, Workspace, Current Directory/Folder, Evaluate, Line of Code, Operator, Operation, Order of Evaluation, Order of Priority, Elementary Mathematical Functions, Expression
To Hand Up:
Solutions for Kiwi Exercise

Practical 2: Variables, Script M-files and Graphing

Aims of Practical

1. Understand the purpose of variables and how to create variables.
2. Write a script M-File (a list of MATLAB commands, saved in a file) with an emphasis on using appropriate comments.
3. Learn how to plot a graph.
    a) Plot a graph of a straight line with fixed start and end points.
    b) Plot a graph of a straight line with variable start and end points.
    c) Plot a graph with two lines.
    d) Use key plotting options including setting the title and axis.

Naming of Numerical Values in MATLAB

Why would we want to name numerical values? Here is one example: Consider the polynomial function $y = x^2 + 4x + 3$ to be calculated for $x = 3$. We could do so by typing in the following:

```
3^2 + 4*3 + 3
```

but what if we also want to know the answer when $x = 7$ and $x = 5$? Wouldn't it be easier to create a name for 3 called x as follows:

```
x = 3
```

and then evaluate our polynomial function by typing in:

```
x^2 + 4x + 3
```

`x` is what we call a **variable**.

Exercise: Using Variables

> Type in the above lines to set the value of `x` equal to 3 and then evaluate the value of the polynomial `y`.

Warning: The above code contains an error that is commonly made by learners.

> What is the error?

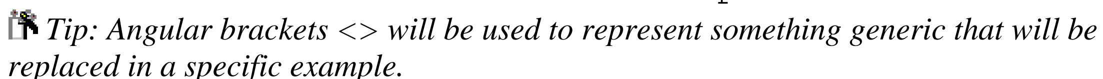
> Change the value of `x` to evaluate the polynomial for `x` = 5 and 7. You will need to re-run the command to solve the polynomial each time you update `x`.

Coding Reminder: The up-arrow on the keyboard will recall a previous line for you to edit.

Variables

Variables must be assigned values in the following format *before* they can be used in an expression.

```
<variable name> = <valid expression>
```


Tip: Angular brackets <> will be used to represent something generic that will be replaced in a specific example.

What this statement means is that the **left hand side** of an *equals sign* is always a **variable name,** say, `x`**.** The **right hand side** is an **expression** representing a *new value* to be assigned to that variable.

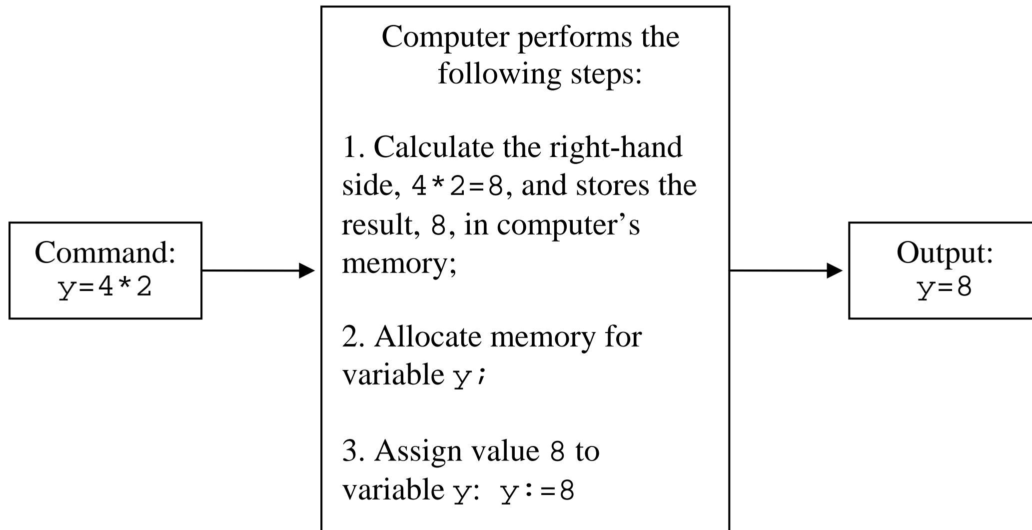
Exercise:

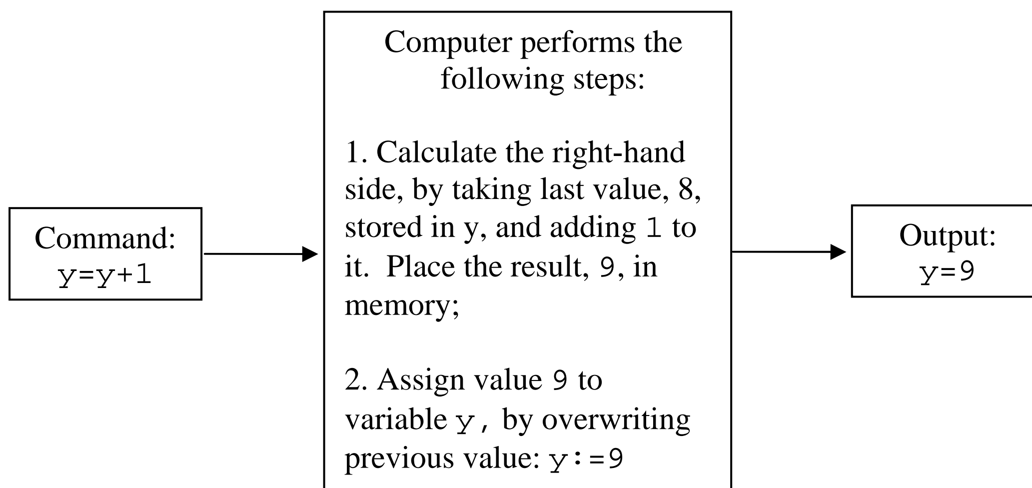
> Type in the following examples exactly as written to see what is a `<valid expression>` and `<variable name>`.

> Check what happens in your *Workspace Window* when you create and update your variables

`x=1+4`	*Valid? Y/N*	*Reason (if invalid)*________________________
`x=1+2+…+10`	*Valid? Y/N*	*Reason (if invalid)*________________________
`x+1=7`	*Valid? Y/N*	*Reason (if invalid)*________________________
`x=2(6)`	*Valid? Y/N*	*Reason (if invalid)*________________________
`bigX=x+1`	*Valid? Y/N*	*Reason (if invalid)*________________________
`x=newX+1`	*Valid? Y/N*	*Reason (if invalid)*________________________
`x=x+1`	*Valid? Y/N*	*Reason (if invalid)*________________________

The last example shows an important concept: a variable can refer to itself to be given a new value. How would the last example work in the computer?





Naming Variables

It is a good programming practice to give your variables meaningful names, so that you and other people can read your code easily.

Exercise

A love-struck engineer specialising in planning the foundations of a building wrote the following code to calculate the area of a rectangle.

> Determine what each variable represents and then rename the variables to write the new code in MATLAB with meaningful variable names.

```
dinner = 4
nice = 7
roses = dinner*nice
```

Some important **rules for variable names** are:

1. They are **case sensitive**: `nice`, `NICE`,`NiCe` and `NicE` are all different.
2. They can have a **length of up to 31 characters** `Pachycephalosaurus` is a valid variable name (although it is quite frustrating to type).
3. They **must start with a letter**, followed by any combination of letters, numbers and underscores. For example, `week1`, `week1_prac`, `MATLAB_prac_1_is_fun`,`x` are all valid variable names.
4. There are some **reserved words** that **can't be used** as a variable name, i.e. `help, pi, sin, pretty, etc.`
5. `clear` statements are used to delete the values of previous variables. `clear all` deletes all values, whereas `clear <variable name>` just deletes the variable you nominate. It is a good programming practice to start your code with command `clear all`.

My preferred naming convention is to capitalise the start of each new word, as in `thisIsALongVariable` however some people prefer to separate words with underscores, as in `this_is_a_long_variable`.

Exercise

> Use MATLAB to evaluate the following functions at the given values of x. Type in the command `clear x` beforehand.

$f = \sqrt{3x-6}, \quad x = 5$ *(answer is 3)*

$g = \frac{x}{|x|}, x = 3, x = -3$ *(answers are 1,-1)*

$h = 2x^2 - x, \; x = 2$ *(answer is 6)*

Comments in MATLAB

When we write a code, it is helpful to add comments to describe or explain the purpose of one or more lines of code. Conveniently, the symbol `%` lets us add text that will be read only by users, not by MATLAB. All characters, letters and words after a percent (`%`) sign on a given line are ignored, and not executed.

Exercise

> Type in the following line (including the comment):

```
t = 9 % sets time variable equal to 9
```

> Then add a comment to the next line, which finds the acceleration at time t.

```
a = 15*sqrt(t) % <add your comment here>
```

Comments as good coding practice

Soon, the code you will be able to write will become much more complicated. You will need to add comments to make it easier for both you and others to read.

Script M-files

A script M-file allows you to place MATLAB commands in a simple text file and then tell MATLAB to open the file and execute all commands precisely as if you had typed them at the *Command Window* prompt.
Script M-files must have file names ending with extension '.m' and must contain only valid MATLAB code.

Exercise

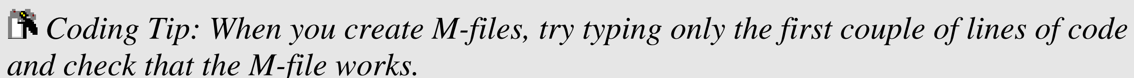
> Open a new script M-file by clicking on *New Script* while in *Home* tab. A new window will be opened. It has three tabs: *Editor*, *Publish* and *View*. To create, save, run, and edit your M-file you need to stay in the *Editor* tab.
> From now on every M-file should start with comment headers that include your name and student ID as well as a description of the purpose of the code as follows:

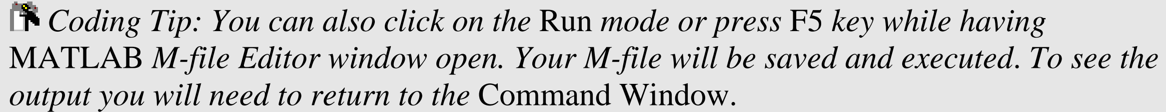
```
% Practical 2 Graphing exercise
% <Add your name, student ID and today's date here>
```

> Add in the commands:

```
clear all
x=2^3
```

> Save your M-file, for example, as prac2e1.m in your preferred directory and make sure that the *Current Folder* in MATLAB is set to the same location.
> Enter command `prac2e1` (or whatever name you saved your M-file as) in the *Command Window*. It should execute your M-file code. **Please, note absence of the file extension .m**


Coding Tip: When you create M-files, try typing only the first couple of lines of code and check that the M-file works.



Coding Tip: You can also click on the Run *mode or press* F5 *key while having* MATLAB *M-file Editor window open. Your M-file will be saved and executed. To see the output you will need to return to the* Command Window.

Plotting Graphs

There are several ways to plot graphs in MATLAB. We will start by using the `plot` command to create a simple straight line graph by providing two sets of coordinates ([x_1 x_2], [y_1 y_2]) defining the start and end points, (x_1, y_1) and (x_2, y_2), respectively.

Exercise

> Create and run a script M-file called plotMe.m for the following command. Remember to add your comment headers at the top.

```
plot([1 3], [2 5])
```

Note: *The graph will appear in a separate Figure Window named Figure 1 that is often hidden behind the main* MATLAB *window. Maximise it on your taskbar if it does not pop up.*

> What does the graph look like? Verify which points of 1, 2, 3, 5 are *x*-coordinates or *y*-coordinates.

Exercise

> Modify your plotMe script M-file to plot the functions $y = 3x + 2$ and $z = 4x$ over the region of $1 \le x \le 3$ by following these steps.

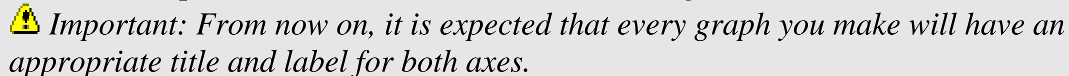
1. Define the endpoints of x as the two variables x_1 & x_2.
2. Define the endpoints of y, as the two variables y_1 & y_2, in terms of x_1 & x_2.
3. Change the plot command to:
   ```
   plot([x1 x2], [y1 y2])
   ```
4. Run your file to check that your updated graph is correct.
5. Define the endpoints of z, as the two variables z_1 & z_2, in terms of x_1 & x_2.
6. Change the plot command so that both lines are plotted on one graph as follows:
   ```
   plot([x1 x2], [y1 y2], [x1 x2], [z1 z2])
   ```
7. Check that your updated graph is correct.

Table of the most useful commands for plotting graphs.	
`hold on`	Allows plotting several graphs on the same figure. All commands after this one apply to a current figure, until the `hold off` command is used.
`plot([x1 x2], [y1 y2])`	plots the straight line between points (x_1, y_1) and (x_2, y_2)
`axis([minX maxX minY maxY])`	specifies axis limits
`legend('y1=equation1', 'y2=equation2')`	sets the labels for the legend *(must be after plot command)*
`xlabel('label for x axis')`	sets the label for the x - axis
`ylabel('label for y axis')`	sets the label for the y - axis
`title('graph title')`	sets the label for the title
`hold off`	Following commands no longer apply to the current figure.

Exercise:

> Modify your script M-file, plotMe to specify an appropriate axis, legend, labels and title. Remember to use the `hold on` and `hold off` commands.

> Use the help files for the `plot` command to change the colour of the lines.


Important: From now on, it is expected that every graph you make will have an appropriate title and label for both axes.

MATLAB/Coding Terms you should become familiar with

Variable, Script M-file, Comment, Valid.

To Hand Up

Your *commented* plotMe.m script M-file with the accompanying *appropriately labelled* graph.

Practical 3: Arrays

Aim of this Practical

1. Learn how to create an array, a list of ordered numbers.
   a) understand the advantages of the different ways of creating arrays including the standard format and the `linspace` command.
   b) access specific numbers in arrays using their position.
   c) use array commands to find quantities such as the sum of the numbers in the array.
2. Learn how to plot a graph using arrays and the `plot` command.
3. One-dimensional array operations.
4. Use the cell environment to execute several lines of code in an M-file.

Arrays

An array in MATLAB is a list of objects, e.g. numbers or characters that can easily be stored and accessed. Each object or element in the array has a fixed position.

Arrays are useful in MATLAB as they allow an operation to be performed on more than one number using a single command. Arrays are also important for plotting.

Exercise: Creating Arrays

> Try these different ways of creating arrays.

1) Type in the elements of the array *one-by-one* at the prompt within square brackets. Elements should be separated by commas or spaces.

```
x1=[2, 5, 13, -1, 0.111, pi]
```

advantage*: easy for small set of numbers,* ***disadvantage****: tedious and time consuming for lots of numbers.*

2) Create an array of integers with increment 1 (by default) from smallest to largest:

```
smallest= -3;
largest= 24;
x2= [smallest:largest]
```

The following is also valid:

```
x2= [-3:24]
```

advantage*: fast, especially for many numbers,* ***disadvantage****: increment of 1 only.*

3) > You can also specify the increment or the step size, h. Check the following (note you must define smallest and largest first).

```
h= 0.06
x3= [smallest:h:largest]
```

advantage: *allows different increments,* ***disadvantage:*** *one must calculate number of elements.*

4) > You can specify the number of points instead of the step size using the `linspace` command.

```
points= 20;
x4= linspace(smallest,largest,points)
```

Useful Coding Commands

Ctrl+c (read: control c): *command to stop execution of the code at any time. You will need to press at the same time both 'Ctrl' and 'c' keys on the keyboard*
; (called semi-colon): *suppresses the appearance of an unnecessary output when typed at the end of the command line*

Exercise: Stopping Calculations

> Type in the following command and see what happens.

```
[1:0.0001:1000]
```

Press Ctrl + c *to stop the calculations!*

>Type in the same line again, but this time add a semi-colon at the end.

```
[1:0.0001:1000];
```

Array Subscripts

Subscripts, or indices, are used to refer to elements at a specified location in an array. In MATLAB, `A(i)` is the ith element in the array `A` and *i* is the index (or position). You can refer to one or more elements and use these to create new scalar variables or *sub-arrays*.

Exercise: Accessing Array Elements

> Type in the following code and decide which commands create scalars and which arrays.
> Check the output and determine the process by which the elements have been selected:

```
A= linspace(5,100,20)
A(1)
A(7)
A(3:6)
```

> Now create two new sub-arrays of `A` as follows:

```
a4 = A(1:2:13)
a5 = A(8:-3:2)
```

>Recall the elements in the sub-arrays as follows:

```
a4(2)
a5(1:3)
```

Array Functions

Function	Description
`sum(Z)`	adds all elements of array `Z`
`prod(Z)`	multiplies all elements of array `Z`
`length(Z)`	returns the length of array `Z`
`[Zmax, i] = max(Z)`	returns largest element `Zmax of Z` and its position, `i`
`[Zmin, i] = min(Z)`	returns smallest element `Zmin` of `Z` and its position, `i`
`sort(Z)`	sorts elements of array `Z` in ascending order
`find(Z>3)`	finds the **indices** of elements of array `Z` larger than 3

Exercise: Using the Array Functions

>Create a script M-file called arrayFunctions.m to run the following code, starting by entering comment headers with your name, the date and the purpose of the M-file.

1) Create an array of 23 random numbers between 0 and 200 called `randy` using the following command:

```
randy = 200*rand(1,23)
```

2) Find the average of the numbers in `randy` using the `sum` and `length` commands.
3) Find the largest element and the smallest element in `randy` and swap them.
4) Sort `randy` into ascending order. Do you know how to sort it in descending order?
5) Find the indices of elements in `randy` which are larger than 100.

Plotting Graphs with Arrays

Last week we introduced some of the basic plotting functionality, including plotting a straight line and modifying the graph axes, title etc. This week we will look at using arrays to plot functions.

How does the `plot` *command work? Actually, it draws a straight line segment between each set of coordinates that you give it. If the line you are trying to draw is a curve, it will appear smooth by plotting many tiny lines.*

Exercise: Graphing with Arrays

> Create a script M-file called plotArray.m for the following code in order to plot a sine curve between 0 and 2π . Don't forget to add comment headers.

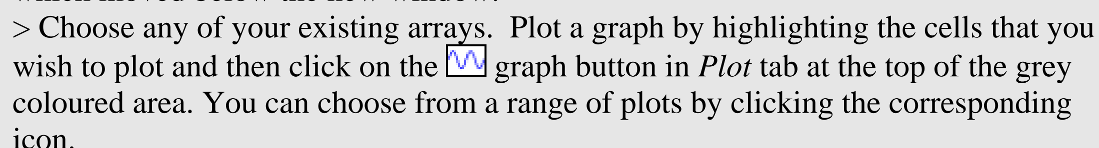
```
clear all
points = 3
x = linspace(0,2*pi,points)
y = sin(x)
plot(x,y)
```

> What is wrong with this graph? Change the variable called `points` to 5 and save before re-running the code, then increase it to 10, 20, 40, 80, 200. Examine each new graph and stop when you think that it is acceptable.

The MATLAB Variable Editor

Switch to 'VARIABLE' tab to edit your variables/arrays manually. It allows you to edit your arrays and variables in a spreadsheet environment.

> Press Open icon to choose a variable or an array. Otherwise simply double click on the name of the variable or an array you wish to edit in your *Workspace Window*. A new window named 'Variables - <name of the variable>' will be opened. Change any value and see what happens in the *Workspace Window* and in the *Command Window* which moved below the new window.

> Choose any of your existing arrays. Plot a graph by highlighting the cells that you wish to plot and then click on the  graph button in *Plot* tab at the top of the grey coloured area. You can choose from a range of plots by clicking the corresponding icon.

> To switch off Variable Editor close the window named 'Variables - <name of the variable>'.

One-dimensional Array Operations

MATLAB has a specific **array or 'element-by-element' multiplication, •*,** designed for one-dimensional arrays, that is also called 'Hadamard product' in mathematics, and uses a dot, **•,** before multiply sign, *, eg `V1•*V2`

$$\begin{bmatrix}3\\2\\4\end{bmatrix} \bullet * \begin{bmatrix}5\\1\\3\end{bmatrix} = \begin{bmatrix}3*5\\2*1\\4*3\end{bmatrix} = \begin{bmatrix}15\\2\\12\end{bmatrix} \quad \text{or} \quad [1\ 2\ 3] \bullet * [0\ 1\ 1] = [0\ 2\ 3]$$

Exercise: Multiplication with Arrays

> First define `A=linspace(0,5,11)` and `B=linspace(1,6,11), c=4,` and `d=-1`
> Find out what happens if you run the following equations:

```
A*c+d
A*c+d*B
```

> What happens if you multiply?

```
A*B
```

Note: MATLAB sends you an error message that 'Matrix dimensions should agree'. It means that you forgot to use a dot before the multiplication sign. Matrix multiplication MATLAB referred to will be taught in further courses.

> Try it again with a **dot** to perform array multiplication:

A**.***B.

.

Exercise: Multiplication of Arrays and Scalars

> Type in the following commands to find out which one works.

```
A*5     A.*5     5*A
A/5     5/A      5./A
A^2     2^A      2.^A
```

Table of Array Multiplication Expressions Involving a Dot Before Operator

`A.*B`	multiplies corresponding elements of `A` and `B`
`A./B`	divides corresponding elements of `A` and `B`
`5./A`	divides 5 by each element of `A`
`A.^B`	calculates `A(i)^B(i)` for every corresponding element in `A` and `B`
`2.^A`	finds `2^A(i)` for every element of `A`
`A.^3`	cubes each element in `A`

Debugging exercise

The following exercise use *code sections*, also known as *code cells* or *cell mode* in the M-file environment to break code into smaller sections. A *code section* contains lines of code that you want to evaluate as a group in a MATLAB script, beginning with two comment characters (%%). You can run each section and find the errors independently. Such a procedure is called *debugging*.

> Use the *cell mode* environment by following these steps (*making sure you update the comment headers first*):

1. Open new M-file in the *Editor* tab.
   You may like to dock the M-file *Editor* in the *Command Window* area by clicking *Dock* option in *Editor* window' *Actions* button. In this case you can see your code in the *Editor* window and its execution results below in the *Command window*.
3. Type the code.

Cells or sections are marked by the double percent sign- %% and each cell has been given a title. You will notice that when you click on a cell, the text is highlighted in yellow.

4. Click on the first cell and execute it by pressing the *Run Section* button.
5. Correct and re-run the first cell.
6. Go through each of the following cells and debug it. You can choose a cell either by clicking on *Advance* button or simply by clicking the cursor within the cell you want to debug
7. After debugging this exercise, go to *Publish* tab. Press on *Publish* button to publish your script M-file as an HTML Document.

   You will probably find you have some unnecessary output.

8. Close the output window. Add in semi-colons at the end of lines with unwanted output in your M-file before republishing.

Notes on publishing:

Comment lines starting with the double percent %% will be formatted as headers.
Additional comment lines will be treated as text.
MATLAB commands will be formatted as code.
Command Window and graphical output will also be included.
For further information, type in the *Search for documentation* bar 'Publish MATLAB code'.

>**Exercise:**

```
%% Practical 3 debugging exercise
% Add your name and student ID here
% Remember you can suppress output by adding the ;

%% Part 1
% Create an array that contains the cubic roots
% of all elements of array A
clear all
A = [0 1 8 27 64]
```

```
cubic_root = A^(1/3) % **** There is a dot missing here
%-------------------------------------------------

%% Part 2
% Plot & sum an array containing the series 1/n^2
% for integers from 1 to 100
clear all
x=[1:100]
y=1/x^2 % **** There are two dots missing here
plot(x,y)
sum(y) % The answer should be 1.6350
%-------------------------------------------------

%% Part 3
% Conjecture that the limit of sin(x)/x = 1 as x -> 0
% by evaluating sin(x)/x for x = 0.1, 0.01,., 0.00000001
% Find the positions where it is 1 for the given precision
clear all
format long % uses 16 digits for each value
x=0.1^[1:8] % **** One dot missing here
y=sin(x)/x  % **** One dot missing here
error = 1-x(8)
c=find(y>=error) %finds indices of elements of array that are bigger
                 %than defined value of error
plot(x,y)
%-------------------------------------------------

%% Part 4
% Find out why this code is not working
% Update the value of maxX to make it work properly
clear all
maxX = 9 % change this value
X=[1:2:maxX]
Y=[10:-1:1]
X+Y
%-------------------------------------------------
%% Part 5
% Find the sum of the series (n+1)*n^(1/3)
% for n = 1 to 100
% Your answer should be 2.047524258847828e+004
clear all
n_plus_1 = [2:101]
n_cubic_root = [1:100]^(1/3) % **** One dot is missing here
% note that the above line is a shorter version of part 1
sum_of_series = sum(n_plus_1*n_cubic_root) % **** add dot for the
%array multiplication
```

MATLAB/Coding Terms you should be familiar with

Array, Index, Dot notation, Cell mode, Sections.

To Hand up

Your *commented* plotArray.m script M-file with the accompanying *appropriately labelled* graph.

Practical 4: `For`- and `While`- Loops, `If`-statements

Aims of Practical

1. Get an overview of command sequence controls- `for`, `while`, `if-else`
2. Create a `for`- loop to repeatedly execute statements a fixed number of times.
4. Create a `while`- loop to execute commands as long as a certain condition is met.
3. Use relational operators:
    a) greater than, less than, equal to, etc.
    b) Boolean operators: *and, or, not.*
4. Use `if-else` construction to change the order of execution.
5. Understand the purpose of count variables.
6. Learn how to indent in order to improve readability.
7. Implement a small problem that requires these commands.

Command Sequence Controls

MATLAB offers features to allow you to control the sequencing of commands by setting conditions. The following table shows the main types

`for` - loop	Executes a set of commands repeatedly by incrementing a variable by a given step size until the set maximum is reached.
Simple example: For each hour from 1pm to 12pm, print the statement “it is <hour> o’clock”.	
`while` - loop	Executes a set of commands if a condition after `while` is true.
Simple example: You are asked to count during one minute. In other words, while chronometer hand have not done a whole circle, keep counting. In this case you do not know what will be your last number.	
`if-else` statements	Executes a set of commands once, if a given condition is met, else- executes a set of commands, if the given condition is not met.
Simple example: **If** the hour of the day is 6pm, print the statement “it is dinner-time, it’s <hour> o’clock” **Else** the hour of the day is not 6pm, print the statement “it is not dinner-time, it’s <hour> o’clock”.	

`for`**- loop syntax**

```
for <variable> = <start>:<step>:<finish>
     <commands>…..
     ……………..
end
```

`while`**- loop syntax**

```
while <condition>
     <commands>…..%executed if condition is true
     ……………..
end
```

Example: `for`-loop versus `while` - loop

```
for i = 1:1:10
     i=i+1   %***********
end
```

Note*: the command* ‘`for i=1:1:10`’ *does not create an array.*
Note*: the command* ‘`for i=1:1:10`’ is equivalent to ‘`for i=1:10`’.

Now, let’s do the same by using `while`- loop

```
i=1; %we need to have a starting point to be able
       to check the condition at the  first time
while i<12 % to check is whether i<12 or not
    i=i+1  %this command is the same as in for-loop
           % we should get the next value of i
end
```

This way is a bit longer, so if you definitely know how many times you will need to do some commands, use `for` –loop. As you will see below, there are some situations, when using `while`-loop is the only way to solve the problem.

Example: (from a past MATLAB test)
> Calculate the sum S of elements $a_i = \sqrt{2i-1}$, $i = 1, 2, \ldots$, until the sum will exceed 20. Type in the following code and examine the output.

```
S=0;  % Initial assignment for sum to be able to
       % check condition
i=1;   % first value for i is 1
while S<20 %condition to check
     S=S+sqrt(2*i-1); % recurrent formula for S
     i=i+1; % next i
end
number_of_loops=i-1 % do you know why i-1?
S % shows the final value
```

>Exercise: Understanding Looping
Type the above `for`- loop into MATLAB. How many times will the starred line of the previous example be executed? What are starting point, increment (step) and ending point of this loop?
Start:________step:_________end:_________total number of loops:_________

Relational Operators

Symbol	Example	Meaning of Example
`<`	`a<6`	true if `a` is less than `6`
`<=`	`a<=6`	true if `a` is less than or equal to `6`
`>`	`a>6`	true if `a` is greater than `6`
`>=`	`a>=6`	true if `a` is greater than or equal to `6`
`==`	`a==6`	true if `a` is equal to `6` *(note double equals sign)*
`~=`	`a~=6`	true if `a` is not equal to `6`

Boolean operators

Symbol	Definition	Example	Meaning of Example
`&`	And	`(a>2) & (a<6)`	`a` is greater than `2` *and* less than `6`
`|`	Or	`(a<2) | (a>6)`	`a` is less than `2` *or* greater than `6`
`~`	Not	`~(a==6)`	`a` is *not* equal to `6`

Simple `if`- statements

```
if <expression>
    <commands>…%executed only when
               %expression is true
end
```

Example:

```
% This code assigns the value of x to negativeValue
% only if x is negative
x = 4;
if x<0
     negativeValue = x
end
```

Exercise: Using Relational Operators

> *What is wrong with the following code? Type it in MATLAB to check the error message.*

```
% This code assigns the value of x to zeroValue only
% if x is zero
x=4
if x=0
     zeroValue = x
end
```

> What happens when `x` equals to zero or `x` is not zero?

Complex `if-elseif-else-end` statements:

```
if <expression 1>
     <command> % evaluated if expression 1 is true
elseif <expression 2>
     <command> % evaluated if expression 2 is true
elseif expression3
     <command> % evaluated if expression 3 is true
….
else
     <command> % evaluated if none of expressions are
true
end
```

Example:

```
% If x is negative, assign value of x to negativeValue
% If x is positive, assign value of x to positiveValue
% If x is zero, assign value of x to zeroValue

x=4
if x<0
     negativeValue = x
elseif x>0
     positiveValue = x
else
     zeroValue = x
end
```

Exercise: Using `if`- statements

> Verify that the above code is correct by checking with a positive, negative and zero value of `x`. It will be faster to run the code from a script M-file.
> Why we don't need to check if `x` is actually zero when assigning it to `zeroValue`?

Counter Variable

Often, loops involve a count or a counter variable that is updated each time the code within a certain condition is executed.

Exercise :

>Type in the following code and verify the result.

Tip: Clear the Command Window by going to Edit>Clear Command Window.

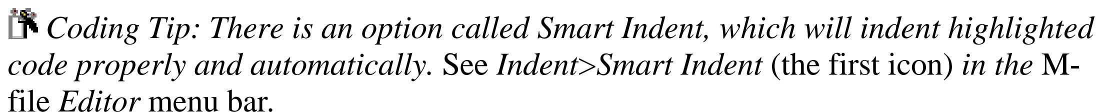
```
% determine the number of elements in allNumbers bigger
% than 100

count = 0                      % initialize count
allNumbers = 200*rand(1,10);   % array of random numbers
for index = 1:1:10
     if allNumbers(index)>100
          count = count+1;     % update count
     end
end
count                          % print count
```

Indenting

Note that the above code has indents under each of `for` and `if` statement

Indenting makes code more readable, as you can see the logical structure which becomes especially important when you have several `for/if` statements in one M-file. While working on this practical, make sure that you follow this formatting convention.

>Reconsider the above example without indenting. Which is easier to read?

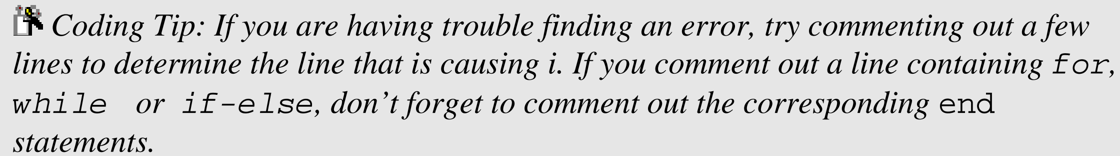

Coding Tip: There is an option called Smart Indent, which will indent highlighted code properly and automatically. See *Indent>Smart Indent* (the first icon) *in the* M-file *Editor* menu bar.

Example:

```
% determine the number of elements in allNumbers bigger
% than 100

count = 0;
for index = 1:1:10
    if allNumbers(index)>100
        count = count+1;
    end
end
count
```

Debugging Exercise

> Complete the exercises in the script M-file for this week. Run each cell separately and follow the instructions, making sure that you also add in the required code marked by starred lines.


Coding Tip: If you are having trouble finding an error, try commenting out a few lines to determine the line that is causing i. If you comment out a line containing `for`, `while` *or* `if-else`*, don't forget to comment out the corresponding* `end` *statements.*

Exercises:

> Fill in the correspondent commands doing what required within %…****

```
%% Practical 4
% Enter your name, student ID and the date here
%----------------------------------------------------------

%% Exercise 1
% Use a for- loop to print out the square of integers from 1 up to
%maxValue.

maxValue = 10;
for num = % ** increment num from 1 to maxValue **
    num % this line simply prints num
    square = num^2 % this line prints the square
end
```

> Use array operations from Practical 3 to solve the problem in Exercise 1.

```
%% Exercise 2
% Use a while- loop to print out the square of integers from 1 up to
% maxValue.

maxValue = 10;
num=1;
while % ** num is not bigger than maxValue **
    num % this line simply prints num
    square = num^2 % this line prints the square
    num=num+1;
end
%----------------------------------------------------------

%% Exercise 3
% Sam gets paid compound interest at a rate defined at 5% per annum.
% Calculate his resulting investment each year and after 10 years.
% Change the rate on the 8th year to 5.75%

rate = 0.05;
investment = 5000;
for year = % ** increment the year from 1 to 10 **
    if % ** year is 8 **
        rate = 0.0575
    end
    year % this line simply prints the year
    investment = (1+rate)* investment % compounding function
end
%----------------------------------------------------------
```

```
%% Exercise 4
% create a graph that draws a straight line from the point (0,0) to
% every other
% point of the set (1,0), (1,1), (1,2), (1,3), (1,4).

hold on %allows us to plot multiple lines on a graph
for y =  %** increment y **
    plot([0,1],[0,y]); % plot x coordinate and y coordinate arrays
end
hold off
%------------------------------------------------------------

%% Exercise 5A
% Use an if-else statement to check if an integer is either a prime %
or a square number.
% Make sure you change the value of the integer.
% Check the help file to find a function that checks if a number is
% prime.
% note that this function works on both arrays and scalars.

integer = 4;
if % ** check if integer is a prime **
    Prime = integer % => If condition is met
elseif % ** check if the number is square **
    Square = integer % => Else condition is met
end
%-------------------------------------------------------------------

%% Exercise 5B
% Add some lines of code in the correct place above to check if the
% number is divisible by 6 and assign it to variable FactorOf6.

%% Exercise 5C
% Modify the above code as follows:
% Use a for-loop to check if each integer from 1 to 100 is a prime, %
square or divisible by 6.
% Collect the integers that meet the above conditions to the
% correspondent arrays Prime, Square or FactorOf6.
% Create count variables pCount, sCount and fCount which are updated
% each time you find a new Prime, Square or FactorOf6 element and use
% these counters to add new elements to your arrays.
% Print Prime, Square & FactorOf6 after your for- loop is finished.
```

MATLAB/Coding Terms you should become familiar with
Valid, Relational and Boolean Operators, `for`-loops, `while`-loops, `if-else` statements, Iteration.
To Hand Up
Your published Word document from the Practical 4 debugging exercise.

Practical 5: Function and Script M-files

Aims of Practical

1. Understand the purpose of function M-files including:
   a) the importance of functions through standard MATLAB functions
   b) the purpose of input and output variables
   c) the difference between function and script M-files
   d) creating and calling a function M-file
2. Be able to create a function M-file by following the steps provided including:
   a) write a function declaration statement
   b) use `return` statements to stop execution.
3. Be able to use commands that operate on functions, such as `fzero` and `fplot`.
4. Create function files from some provided code.

Script M-file

We discussed M-files as early as in Practical 2.

Generally speaking, script M-file is a list of commands (programming code) saved in one document.

Some advantages:

-Easy to use: you can run all the commands in your file by hitting one key.

- No need to re-type or re-run lengthy code.

Disadvantages:

- Values from the script M-file can't be easily assigned to variables in the *Command Window*. You must open, edit and save the M-file to edit values.

Exercise: Creating a Script M-file

The following script M-file finds the value of the function $y = \sin x + x^3$ at $x = 3$.

```
% exercise1script.m
x = 3
y = sin(x) + x^3
```

> Run this M-file by typing the following in the *Command Window*:

```
exercise1script
```

> Then update the M-file to find the value of $f(x)$ for $x = 4$, 5, 6.

Quite often we need to be able to calculate the value of a function $y = f(x)$ for any value of x. Obviously, it is not practical to change value of x each time. For this purpose MATLAB has a special type of M-file, called M-function.

Some Properties of Function M-files
A MATLAB file of a special format that contains code with optional inputs and outputs is called function M-file.
Some advantages:

- Functions can be called from inside of other script and function M-files.
- Inputs allow variable values to be modified when calling the function (eg from the *Command Window*).
- Outputs can be assigned to other variables at the *Command Window* or within a separate M-file.

Disadvantages:
- A slight disadvantage with a function M-file is that you must follow the prescribed format, or else it won't run correctly. Once you get the hang of that, you will see they are often very useful.

The following function M-file finds the value of the function $f(x) = \sin x + x^3$ for any value of x. Type it in and save as exercise1func.m in your *Current Directory*

```
% <insert your name and the date here>
% exercise1func.m
% input: x
% output: p, solved in terms of x

 function p = exercise1func(x) %Note special format!
          p = sin(x) + x^3
```

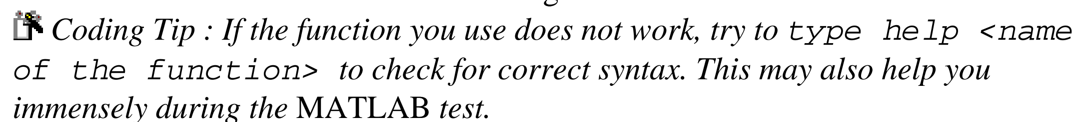
> Call this M-file from the *Command Window* using the following command and then update it to find the value of $p(x)$ for x=3, 4, 5, 6.

```
     exercise1func(3)    % returns value for x = 3
```

> Consider the case, when the variable x is an array [3 4 5 6]. Modify your function M-file so, that it will be able to work with arrays.

Constructing Function M-files
Function M-files allow you to define, construct and store your own functions.
Why would I want to do that? You may want to create a function that plots a graph with certain fixed parameters, or create a complex equation that you will call often, or even, you may want to create a modified version of an existing function, such as a random number generator that creates numbers with values between 100 and 200 instead of 0 and 1.

First, let's recall some of the in-built MATLAB functions you have already used. If you type command `help <name of the function>`, such as `help abs`, in the *Command Window*, MATLAB will print description and correct syntax of this function. These functions have a few things in common and a few differences.



Coding Tip : If the function you use does not work, try to type `help <name of the function>` *to check for correct syntax. This may also help you immensely during the* MATLAB *test.*

Function:	`y = abs(x)`
Description:	Assigns the value `x` to `y` if `x` is non-negative *or* `−x` if `x` is negative.
Input:	1 number: `x`
Output:	1 number: either `x` or `−x`

Function:	`y=rem(a,b)`
Description:	Assigns the remainder of `a/b` to `y`
Input:	2 numbers: `a` is the numerator, `b` is the denominator
Output:	1 number: remainder of `a/b`

Function:	`plot(xArray,yArray)`
Description:	Plots a graph, given an array of `x`-coordinates, `xArray`, and an array of `y`-coordinates, `yArray`.
Input:	2 arrays of equal length: `xArray`, `yArray`. *Note: there can be many additional optional inputs.*
Output:	A graph.

Exercise: Determining the Inputs & Outputs

> For the following function (which you may recall from the first practical), use MATLAB to help you write the description, inputs and outputs.

Function:	`round(a)`
Description:	____
Inputs:	____
Outputs:	____

Exercise: Creating a Customised Random Number Generator

> Create a function M-file called `myRand`.m that outputs a random number between the inputted values of `minRand` and `maxRand` by adding code in the starred lines.

The MATLAB `rand` function returns a random value between 0 and 1. You will need to use this function as well as calculating the scale and offset values.

Example: If you want to find a number between 3 and 10, your scale is 7 and your offset is 3.

```
% <insert your name and the date here>
% myRand.m
% inputs: minRand, maxRand
% outputs: y, a random number with value between
% minRand & maxRand

function y = myRand(minRand, maxRand)
% **calculate the scale**
% **calculate the offset**
% **calculate y, using your scale, offset and
%MATLAB’s rand function**
```

Note: Save your function in the *Current Directory*, otherwise you will need to switch to the directory you saved your function in or type in a full path.

>Type in the following lines to call this function from your *Command Window* and verify that it works.

```
myRand(1,10)
myRand(100,100+1)
myRand(3,pi)
myRand(20)
myRand(20,1)
```

>How do you think your function should handle the last two entries? You may need to edit your function to accommodate inputs of this form or generate message about the error.

Steps for creating function M-files
1. The **function name** and its **M-file name** must be identical (except that the M-file must end with **.m**).
2. The first executable statement must be a **function declaration** of the form
`function <outputVariables> = <functionName>(<inputVariables>)`

Exercise

One of the following function declarations has been taken from the MATLAB `rem.m` function M-file. Determine which one must be correct declaration:

```
function rem = remainder(x,y)
function out = rem(x,y)
out = function rem(x,y)
function out(x,y) = rem(x,y)
```

Exercise: Writing Function Declarations

Write down the `<outputVariables>`, `<functionName>` and `<InputVariables>` for the two function M-files from the previous exercises and verify they match the function declaration statement.

Function M-file 1: $p(x) = \sin x + x^3$

Function Name: ____________________________

Output Variables: ____________________________

Input Variables: ____________________________

Function M-file 2: Creating a Customised Random Number Generator

Function Name: ____________________________

Output Variables: ____________________________

Input Variables: ____________________________

Exercise: Creating a Function M-file

> Follow steps below to create a simple function M-file that computes and outputs the nth power of 2, 2^n , where *n* is a number specified each time the function M-file is run.

1. Create a blank M-file and save it as twoN.m

2. What should go at the top of every M-file? Add in header comments. This time, make sure you include the function description, inputs and outputs as well as your name and the date.

3. Type the **function declaration** into your file called twoN.m

```
function <outputVariables> = <functionName>(<inputVariables>)
```

a) replace <function Name> with twoN

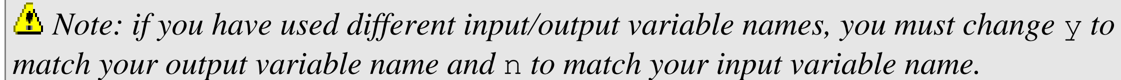
b) decide upon an appropriate input variable name. In this case you may call it simply n.

c) decide upon an output variable name. The name y will be used in this case. *(In other examples you could use* fn*,* f_n *or another name of your choice as an output variable name.)*

Important: Every output variable must be assigned a value within your code.

4. Now, you need to write your code. This function is pretty simple, so the code should only contain the following line:

```
y=2^n
```



Note: if you have used different input/output variable names, you must change y *to match your output variable name and* n *to match your input variable name.*

5. OK, now you're ready to save and test your function M-file. After saving, make sure that your *Current Directory* matches the one you saved your M-file in.

6. Run the following lines in the *Command Window* to verify your function M-file works.

```
twoTo8 = twoN(8)
newNumber = twoN(5)
squareOfTwo = twoN(2)
twoN(9)
rootOfPower = twoN(5)^(1/2)
twoN %Why this does not work?
```

Exercise: Writing your Own Function M-file

Create a function M-file called quadRoots to find the roots of quadratic polynomials of the form $y = ax^2 + bx + c$

Its inputs will be the coefficients *a*, *b* and *c*.

Its outputs will be the two roots, r_1 and r_2 calculated by the formula:

$$r_1,\ r_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Test quadRoots in the *Command Window* with the following three polynomials:

$y = x^2 + 3x + 2$ (ans: $r_1 = -1$, $r_2 = -2$)

$y = x^2 + 6x + 10$ (ans: $r_1 = -3 + i$, $r_2 = -3 - i$)

$y = x^2 + 6x + 13$ (ans: $r_1 = -3 + 2i$, $r_2 = -3 - 2i$)

Return statements

`Return` statements can be used to break the flow of execution of a function.

Example:

```
% indexOf: a function that finds the first index of a given number
% in an array. If the number is not in the array, the function
% outputs the index as -1.
% Inputs: a number, an array
% Outputs: an array index or -1 (means there is no such number in
% your input)

function position = indexOf(value, inputArray)
position = -1; %    sets index to -1, number not yet found in array
for i = 1:length(inputArray)
    i   % this line shows how many times the for loop runs
    if value == inputArray(i)
        position = i;
            return;     % return stops the loop if the number is
                        % found
    end
end
```

Exercise: Using Return Statements

> Put the indexOf.m M-file in your *Current Directory*.

> Test it with the following commands in your *Command Window*:

```
idx = indexOf(8, [1, -1, 1, 4])
idx = indexOf(8, [1, 1, 8, 1, 8])
i=2
xArray = [1, 2, 5, 7]
idx = indexOf(i, xArray)
```

> Disable the `return` statement by adding a comment sign, %. Re-test the function to see how many times the `for`- loop runs without it.

Coding Tip: Make sure you ***save*** *each time you edit your function.*

Functions operating on functions
Suppose a function `y = demoFun(x)` has been defined in a function M-file `demoFun.m`

`fplot(@demoFun,[a b])`	Plots the function for `a ≤ x ≤ b` without setting up arrays
`fzero(@demoFun,[a b])`	Finds one value of `x` for `demoFun(x) = 0` provided that the signs of `demoFun(a)` and `demoFun(b)` are opposite.
`fzero(@demoFun,c)`	Finds one value of `x` for `demoFun(x) = 0` by starting a search at `x = c`
`fminbnd(@demoFun,a,b)`	Finds the coordinates of a minimum point of `demoFun(x)` at the interval `a ≤ x ≤ b`
`quadl(@demoFun,a,b)`	Finds an accurate value for $\int_a^b y(x)dx$ Note: `quadl` *uses arrays, so therefore you must set your function up treating x as an array and so using the dot notation for operations.*

Exercise: Using Functions

1. Create a **function M-file** called `myCubic.m` whose output is the value of the function $y = x^3 + 2x^2 - 5x - 8$

   Input: `x`
   Output: `y`

> Verify that this function works by checking that `myCubic(-5)=-58` and `myCubic(5)=142`

2. Create a **script M-file** called `cubicExercise.m` that contains the following five cells with code that:

   a) plots `myCubic(x)` between the values of [-5, 5],
   b) finds a local minimum of `myCubic(x)`, located between 0 and 5,
   c) finds the all three roots of `myCubic(x)` using appropriate intervals `[a,b]`
   d) finds the value of the definite integral of `myCubic(x)` between -5 and 5.
   Hint: *You will need to use the array dot notation so you can use the* `quadl` *function to calculate the integral of y.*

3. Make sure cubicExercise.m is marked up using *cell* formatting and publish it.

MATLAB/Coding Terms you should become familiar with
Input, Function call, Function Declaration, Function M-file, Return Statement.

To Hand Up:
Your published document from the cubicExercise.m script M-file exercise.

Practical 6: Polynomial Functions and Symbolic Toolbox

Aims of Practical
1. Use coefficient arrays for representing polynomials in MATLAB.
2. Be able to add and subtract polynomials.
3. Have a general understanding of the polynomial commands available in MATLAB.
4. Use these commands in some short exercises.
5. Understand the purpose of the Symbolic Toolbox
6. Solve symbolic equations
   a) solve a single equation
   b) solve simultaneous equations

Polynomials in MATLAB
In MATLAB, a polynomial is represented by an array of its coefficients of the powers. The MATLAB polynomial functions allow us to perform some useful commands such as addition, multiplication and finding the roots of polynomials.

Example:
$p(x) = 3x^5 - 4x^4 + 7x^2 - 9x + 3 = 3x^5 - 4x^4 + 0x^3 + 7x^2 - 9x^1 + 3x^0$ would be entered as

```
p = [3   -4   0    7   -9   3],
```

so that `p(1)=3` is a coefficient for x^5 (term with the highest power), `p(2) =-4` is for x^4,.., `p(6)=3` – for x^0 (the lowest power term).

Exercise: Creating Coefficient Arrays
> Write the polynomial $p(x)$ corresponding to array of coefficients `q`

`q = [4 5 -6]` $p(x) =$

> Type in the coefficient array `s` corresponding to polynomial $s(x)$

$s(x) = -4x^4 + 3$ `s=[ ]`

Adding and subtracting polynomials
The coefficient arrays of the polynomials can be added and subtracted.

Exercise: Adding Polynomials with Coefficient Arrays
> Add the following polynomial coefficient arrays:

```
p = [3 -4 0 7 -9 3]
q = [4 5 -6]
p + q
```

> *What is the problem here? Re-define* `q` *as follows and check whether the addition works as you would expect.*

```
q = [0 0 0 4 5 -6] %Now terms with corresponding
                    powers will be added
p + q
```

Note: To add or subtract two polynomials one should adjust their coefficient arrays putting zeros for absent powers, so that coefficients of terms with the same power will be on the same positions within both coefficient arrays.

Table of some MATLAB polynomial commands.

`roots([c1 c2 c3])`	finds the roots of a polynomial given its array of coefficients, `[c1 c2 c3]`.
`poly([r1 r2])`	constructs a coefficient array, given the roots, `[r1 r2]`, of the desired polynomial.
`polyval(p,x)`	finds the value of the polynomial with coefficient array `p` at any given number `x`.
`conv(p,q)`	multiply two polynomials, $p(x)$ and $q(x)$, with coefficient arrays, `p` and `q`
`[Q,R]=deconv(p,q)`	finds the quotient and remainder of polynomial division, $p(x)/q(x)$

Exercise:

>Define two polynomials $p(x)$ and $q(x)$ as coefficient arrays `p` and `q` in MATLAB

```
p=[ 1 2 3], q=[2 1]
```

> use `conv` function to multiply them

```
conv(p,q)
```

>Now, **by hand,** convert the arrays `p` and `q` back to polynomials and multiply them. Compare your result with the one you received from MATLAB. Is MATLAB result trustworthy?

Exercise: Finding Polynomial Roots

> Find the roots of two polynomials, $p(x)$ and $q(x)$, with coefficient arrays `p` and `q`

```
p = [3 -4 0 7 -9 3]
q = [4 5 -6]
```

Exercise

> Multiply $p(x)$ by $q(x)$ using the `conv` function and corresponding coefficient arrays.

> Use `deconv` function to find the remainder and quotient of $p(x)/q(x)$

Exercise: Re-creating Polynomials from Given Roots

> Find a polynomial $s(x)$ with roots $r_1 = 5$, $r_2 = -5$, $r_3 = 4$, and $r_4 = -4$.

Exercise: A Polynomial Plot Function

> Create a function M-file called `plotPoly.m` that uses an array of polynomial coefficients to plot the polynomial over an interval specified by the user.

```
% Input: coefficient array, endpoints array.
% Output: polynomial graph.
% **create x array using linspace**
% **create y array using polyval**
% **plot the graph using the plot command**
```

> Test your function for `p` and `q` from the previous exercise at the interval [-5, 5].

Exercise: Fitting a Polynomial Curve

> Check the help files under polynomial functions for an appropriate function that fits a polynomial to a set of given data points.

> Create a script M-file called `fitPoly.m` with the following two cells

1) Fit the points (1,7), (2,-1) and (3,3).
2) Plot the resulting polynomial with your plotting M-file.

The Symbolic Toolbox

The past practicals have introduced MATLAB as a powerful programmable graphics calculator. This week we will start showing you how, using MATLAB Symbolic Toolbox, you can, not only solve equations, but also perform algebraic manipulations, such as differential and integral calculus like finding the derivative $f'(x)$ analytically.

Exercise

> Type in the following commands.

```
clear all
sin(x)
```

>What error message is displayed?

The reason for this error message is that you have not first given x a value. However, if we make x a **symbolic variable** by using the `syms x` command, then we will be able to create and manipulate formulae without giving x a value first.

Exercise: Creating Symbolic Variables

> Type in the following and see if you still get the same error message.

```
syms x
sin(x)
```

The `syms` command is actually a shortcut for the `sym` command. The longer way of writing is:

```
sym('x')
```

Exercise: Creating Symbolic Variables

> Type in the following and decide which variables are symbolic.

```
syms x
y = x + 1.4*sqrt(x+3) + 5.49
pretty(y)
```

>What does the `pretty` command do here?

Solving Equations

The Symbolic Toolbox can be used to find solutions for single or simultaneous equations using the `solve` command.

Solving a Single Equation

The first task is to solve a single equation $f(x)=0$ for the variable x, with the result stored in the variable `a`. This can be done with command `solve`.

```
a = solve(f,x)
```

Exercise: Using `solve` for a Single Equation

> Go through and type in the code for the following steps to solve the single equation.

$$\frac{10}{x^3+1}=4-2x$$

> *Watch out! There is an error that you will need to correct!!*

1. Decide what $f(x)$ is (you will need to rearrange the equation)	In this case $f(x)=\frac{10}{x^3+1}-4+2x$
2. Create a symbolic variable x	`syms x`
3. Create a symbolic variable f by assigning a symbolic expression to it	`f=10/(x^3+1)-4+2x`
4. Solve $f(x)=0$	`u = solve(f,x)`

Exercise: Another Example of Solving Symbolic Equation

> Solve the general quadratic equation $ax^2+bx+c=0$ for any x by making the three coefficients a, b, c and x symbolic variables.

```
syms a b c x
```

> Instead of creating a new variable `f` for function, use the following command to solve directly:

```
soln = solve(a*x^2+b*x+c)
```

The above command doesn't specify the variable to solve for. If it is not specified, the solve command will:

1) Solve for x by default or
2) Solve for the last alphabetical variable if there is no variable called x or
3) Solve for the only symbolic variable if there is only one.

Exercise

> Solve the equation again, replacing *a* with *u*, *b* with *v*, *c* with *w* and *x* with *s*.

```
syms u v w s
...
```

> *What is the answer this time?*

Solving Simultaneous Equations

The `solve` command not only allows us to find solutions of single equations, but also simultaneous equations.

The command for solving more than one equation is of the form

```
[a1,a2,...,an] = solve(f1,f2,...,fn,v1,v2,...,vn)
```

This command can solve *n* equations for *n* unknowns.

Exercise: Using `solve` for Simultaneous Equations

> Use MATLAB to find the solution of the following simultaneous equations:

$$\begin{cases} 2y + 3x = 4 \\ 5y - 2x = 5 \end{cases}$$

> This time you will need to create two functions, `f1` and `f2` to solve `f1=0` **and** `f2=0` using the following command:

```
[x,y] = solve(f1,f2)
```

MATLAB/Coding Terms you should become familiar with

Coefficient array, Symbolic Toolbox, Symbolic Variable.

To Hand up

The published document from your fitPoly.m script M-file together with the code from your plotPoly.m function M-file.

Practical 7: Symbolic Toolbox (Continued)

Aims of Practical

1. Be able to graph symbolic equations using `ezplot`
2. Understand the advantages/disadvantages of the three types of graphing in MATLAB - `plot, fplot, ezplot`
3. Realise how numbers are represented with the Symbolic Toolbox.
4. Manipulate equations with algebraic operations.
5. Use the variable substitution & expression evaluation commands.
6. Perform symbolic differentiation and integration.

Graphing using the Symbolic Toolbox: `ezplot`

The Symbolic Toolbox command `ezplot` is very convenient for plotting. You will need to specify the symbolic function and the endpoints (optional). By default, the endpoints are -2π and 2π. Hence, you will need to determine endpoints only if they are different from the default ones.

Note : It is possible to plot several functions on the same graph using `ezplot` and `hold on, hold off` commands. But the graphs will have the same line properties (colour, thickness, etc) and thus sometimes are hard to distinguish.

Exercise: Using the `ezplot` **command**

> Type the following code and examine output plots.

```
syms x
ezplot(sin(x))
ezplot(sin(x),[-pi pi])
hold on
ezplot(sin(x))
ezplot(3*x+2)
hold off
```

Table for comparison of three graph plotting functions

	`plot (p.13)`	`fplot (p.32)`	`ezplot (p.43)`
Required Inputs	-array of x- coordinates -array of y- coordinates	-function handle -endpoints	-symbolic function -endpoints
Advantages	does not need a function formula full control of graph parameters	function is stored in M-file for easy access (via handle) and modification	can easily create a graph using only two lines of code
Disadvantages	requires more code to generate	must create a function M-file if it is not MATLAB in-build function	function is not stored in easily accessible M-file, graphs have the same colour

Example:

Create a graph of the function $y = \begin{cases} x, & x \le 0 \\ 2x, & 0 < x < 1 \\ 3x, & x \ge 1 \end{cases}$ over the interval [-2,2].

Which plot command would you use?

Answer:

The answer depends on your overall task.

If you will need this function for further calculations, then use `fplot` command. To use `fplot`, you will need to create a function M-file, specifying the three parts of the function separately by using `if` statements.

If you want just to plot this function, then use `ezplot` with `hold on` and `hold off` commands to plot it ‘piece-by-piece’, specifying endpoints.

```
clear all
syms x
hold on
ezplot(x,[-2, 0])
ezplot(2*x,[0, 1])
ezplot(3*x,[1, 2])
axis([-2 2 -2 6]) %> why do you need to do this?
title('Piece-wise function')%what changes will happen if
                            %you delete this command?
hold off
```

Note: If you plot several functions using `ezplot,` then the title of the figure will be of the last function plotted.

The hardest option for this sort of function is to use the `plot` command. Otherwise, all the above options are relatively same.

Example 2:
Quickly check that the equation $speed = 20\ln(t+1)$ accurately reflects the speed of a car for *t* between 0 and 10 seconds.

Hint: use `ezplot`.

Exercise:
In the table below you are given the results of a student's weekly assignments to plot.

>Which graphing technique would you use in this case?

Week	1	2	3	4	5	6	7	8	9	10	11	12
Grade	88	97	100	49	76	33	100	88	87	63	10	82

Finding the value of an equation at a point using commands `subs,eval.`

Exercise: Evaluating Equations
Consider the function $f(x) = (x^3 + 2)\sec x$
> Define this function f in the Symbolic Toolbox, by first defining *x* as a symbolic variable.
> Find $f(0.157)$ by evaluating this function at $x = 0.157$ using the `subs` command as follows:

```
y = subs(f,x,0.157)
```

> Find $f(0.123)$ by evaluating this function at $x = 0.123$ using the `eval` command as follows:

```
x = 0.123
y = eval(f)
```

As you can see the difference in these two examples is only in the number of commands you need to use to evaluate the function at a certain *x*.

Algebraic transformation of a function using `subs` **command**

Exercise: Variables' substitution
> Type the following commands, where function `f` is the same as in the previous example.

```
syms t
y2=subs(f,x,t)    %substitutes x with t
y3=subs(f,x,t-1)  %substitutes x with t-1
y3=simple(y3)     %to simplify the expression for y3
```

Exercise:
> Define function $f(x) = x^4 - 5x^2 + 4$ in the Symbolic Toolbox. Make sure that you typed `syms x` before the function' definition. Substitute *x* with $\sqrt{t}$ using `subs` command. Do not forget to type `syms t` before substitution.

Numbers and the Symbolic Toolbox
Let's use the following example to illustrate the difference between symbolic and standard numbers representation.

Exercise:
> Use the polynomial `roots` command from last week's practical to find the roots of the polynomial $p(x) = x^2 + 9x + 6$.
> Now, define x as a symbolic variable and define the polynomial p in MATLAB as follows:

```
p = x^2+9*x+6
```

> Use the `solve(p)` command to find the roots using the Symbolic Toolbox.
> What is the difference between the answers for the standard and symbolic methods?

Formats for numbers in the Symbolic Toolbox

By default, the Symbolic Toolbox uses integers or rational fractions such as 1, 1/2. Otherwise, it uses a real number approximation by a fraction of large integers, eg 3333/100000. Command `sym(number)` converts a number to its symbolic form while `double(number)` command reverses symbolic representation to MATLAB standard format. Therefore, `sym` command means 'switch to Symbolic Toolbox', while `double` means 'switch back to MATLAB'

Exercise: Understanding Numerical Formats
> For the following numbers, find out their symbolic and standard formats.

Number	Command Examples	
11/13	`sym(11/13)`	`double(11/13)`
3.85		
pi		
0.33333		
$(4/5)^2$	`sym((4/5)^2)`	`double((4/5)^2)`
$49^{1/2}$		
$59^{1/2}$		
e^1		
sin(1)		

Most common Symbolic Toolbox algebraic operations	
`pretty`	formats the output to look like type-set mathematics
`simplify`	performs algebraic and other function simplifications
`simple`	tries a number of simplification techniques including trigonometric identities; running this command twice can further simplify a complicated expression
`collect`	collects like terms
`factor`	attempts to factor the expression
`expand`	expands all terms
`poly2sym`	converts array of polynomial coefficients into a symbolic expression with x as the variable
`sym2poly`	converts symbolic polynomial into its coefficient array

Exercise: Using Algebraic Operations

Let $p(x) = x^2(x+4)^2 - 8x(x+1)^2 + 13x - 6$

> Define *p* in a script M-file called symPoly.m using the following commands:

```
syms x
p = x^2*(x+4)^2-8*x*(x+1)^2+13*x-6
```

> Now use the algebraic operations to help you create the following five cells.

1) Expand `p`
2) Find the factors of `p`
3) Store the coefficient array of `p` in a variable called `coeff_p`
4) Use the `solve` command to find when `p` = 0
5) Now try the polynomial `roots` command together with `double` to find the roots of $p(x)$ in standard MATLAB format.

Differentiation & Integration

The Symbolic Toolbox features special commands for differentiation and integration. This practical will focus only on the basic `diff` and `int` commands. The default variable for integration or differentiation is *x* (first), or *t* (if there is no *x*) but you can specify another symbol.

Exercise: Finding Derivatives & Integrals

>Open your script M-file symPoly.m with the equation

$p(x) = x^2(x+4)^2 - 8x(x+1)^2 + 13x - 6$.

> Create two new cells using the following commands to find the 1st and 2nd derivatives of $p(x)$ as well as the indefinite and the definite integral from -1 to 1.

```
dp=diff(p,x) %note that you do not need to specify x
             %because it is the default variable
d2p = diff(p,x,2) % How would you find the third
                    derivative of x?
intp = int(p,x)
intpDef=int(p,x,-1,1)
```

Numerical Integration

Sometimes there is no explicit expression because the antiderivative can't be written in terms of familiar functions. Instead MATLAB can use numerical techniques to find the definite integral by using the double command.

Exercise: Limitations of Integration Techniques

> Create a symbolic equation $f(v) = \dfrac{\sin v}{v + e^v}$

> Use the `int` command to find the indefinite integral, and observe the output.

```
intf = int(f,v)
```

> Modify the above command to find the definite integral between v=0 and v=pi. Does the command work?

> Observe what happens if you use the `double` command.

```
defint=double(int(f,v,0,pi))
```

Summary Table of some Symbolic Toolbox functions

`sym('x')` or `syms x`	Creates a symbolic variable x
`sym(1/4)`	Creates symbolic number $1/4$
`double(1/4)`	Converts symbolic number $1/4$ to standard 0.25
`subs(f,x,a)`	Substitutes variable x in $f(x)$ with value a and finds $f(x)$
`x=a` `eval(f)`	Finds value of $f(x)$ at a by assigning a to x first.
`ezplot(f,a,b)`	Plots the symbolic function $f(x)$ over the interval $[a,b]$
`pretty`	Formats output to look like type-set mathematics
`simplify`	Performs algebraic and other function simplifications
`simple`	Tries a number of simplification techniques including trigonometric identities. Running this command twice can further simplify complex expression.
`collect`	Collects like terms
`factor`	Attempts to factor the expression
`expand`	Expands all terms
`poly2sym`	Converts an array of polynomial coefficients into a symbolic expression with x as the variable
`sym2poly`	Converts symbolic polynomial into its coefficient array
`diff(f,x)`	Finds 1st derivative of $f(x)$ with respect to x
`diff(f,x,n)`	Finds nth derivative of $f(x)$ with respect to x
`int(f,x)`	Indefinite integral of $f(x)$
`int(f,x,a,b)`	Definite integral of $f(x)$ with endpoints a , b

MATLAB/Coding Terms you should become familiar with

Symbolic Variables, Symbolic Toolbox.

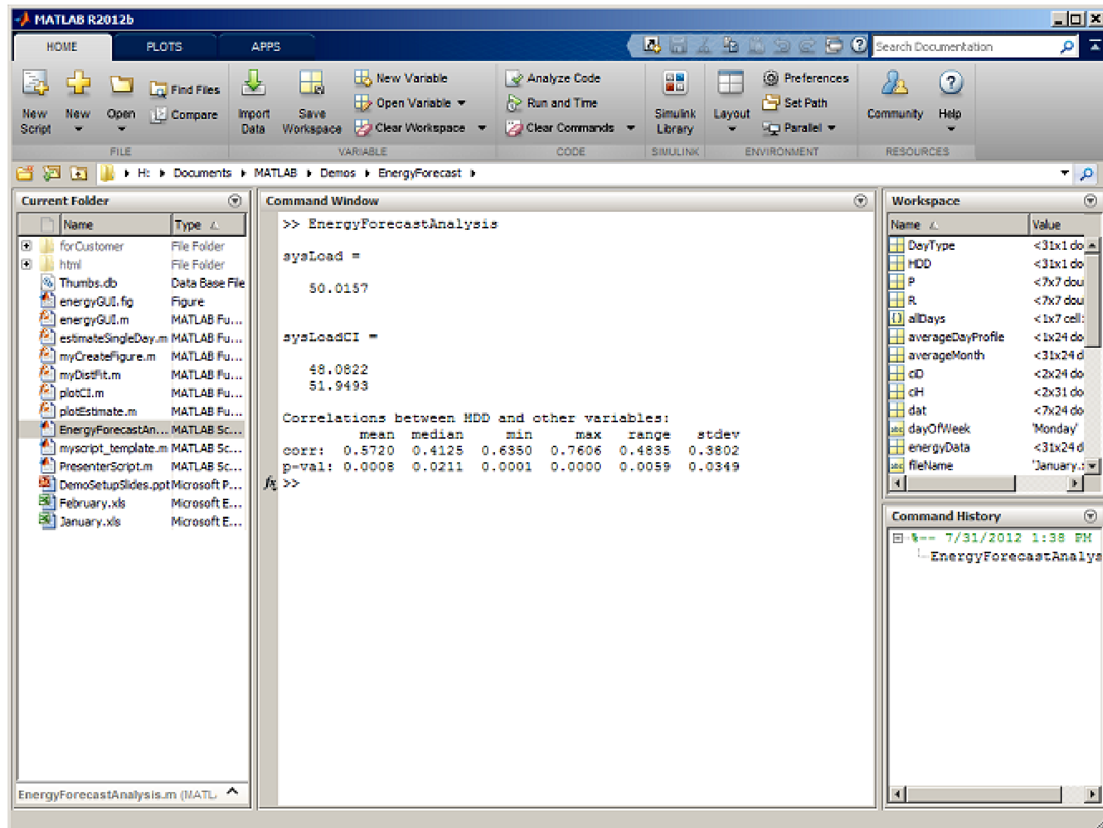
To Hand up

Your published Word document from the symPoly.m script M-file after completing the "Using Algebraic Operations" and "Finding Derivatives and Integrals" exercises.

Appendix A: MATLAB Layout Overview

(adopted from http://blogs.mathworks.com/loren/2012/09/12/the-matlab-r2012b-desktop-part-1-introduction-to-the-toolstrip/#0807a1e1-04e4-41fe-8613-fdc587948079)

In this part we will discuss the appearance and organization of the MATLAB R2012b Desktop layout.



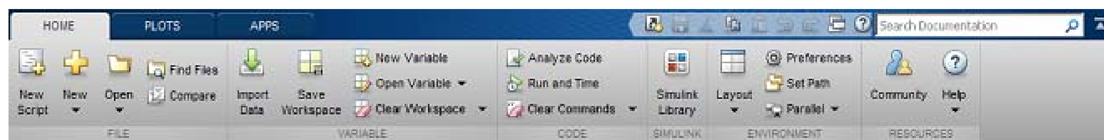
At the top of the screen you can see grey coloured area - the MATLAB Toolstrip. The white area consists of four windows: *Current Folder, Command Window, Workspace and Command History*. We will learn more about them during practicals.

The Toolstrip organizes MATLAB functionality in a series of *tabs*. Tabs are divided into *sections* that contain a series of related *controls*. The controls are buttons, drop-down menus and other user interface elements that you use to do things in MATLAB. For example, the picture above shows the *Home* tab with sections for operations on *Files*, *Variables*, *Code* and so forth. The `File` section has controls to do file related operations including creating scripts (`New Script`), opening files (`Open`), and

comparing two files (`Compare`). The light blue bar in the upper right corner called the *Quick Access Toolbar*.

Global Tabs

When you open the MATLAB R2012b for the first time, you will notice three tabs -- the *Home* tab, the *Plots* tab, and the *Apps* tab. These three tabs are always there no matter what you are doing in MATLAB. For that reason, they are called *global tabs*. The *Home* tab, shown below, is where you go to do general purpose operations like creating new files, importing data, managing your workspace, and setting your Desktop layout.



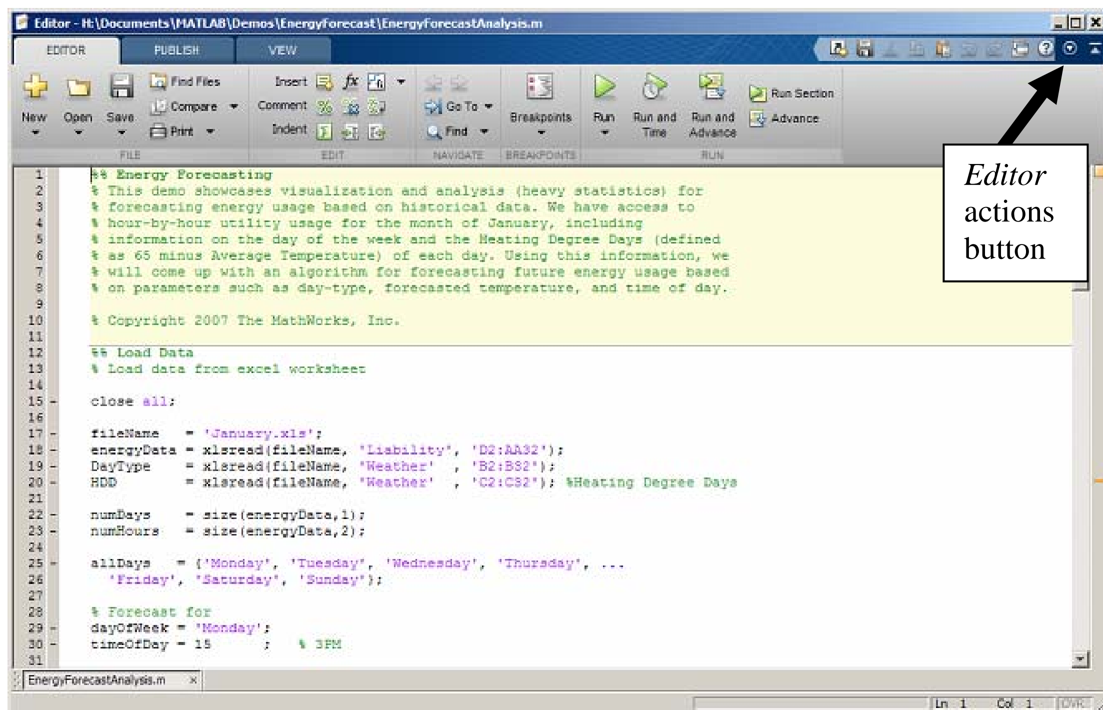
The *Plots* tab, shown below, is where you go to create MATLAB Plots. The *Plots* tab displays a gallery of plots available in MATLAB and any toolboxes that were installed. To create a plot from the gallery, you select the variables in the *Workspace* that you want to plot and then select the type of visualization you want to use for that data. The downward facing arrow on the far right brings down the full extent of the plot gallery with many more choices. The gallery is smart, only showing plots that are appropriate for the data that you've selected.

The last of the global tabs is the *Apps* tab, shown below. It is the place you go to run interactive MATLAB applications. The *Apps* tab presents a gallery of installed applications. The downward facing arrow on the far right brings down the full extent of the applications gallery with many more choices. To start an application you simply click on the correspondent icon.

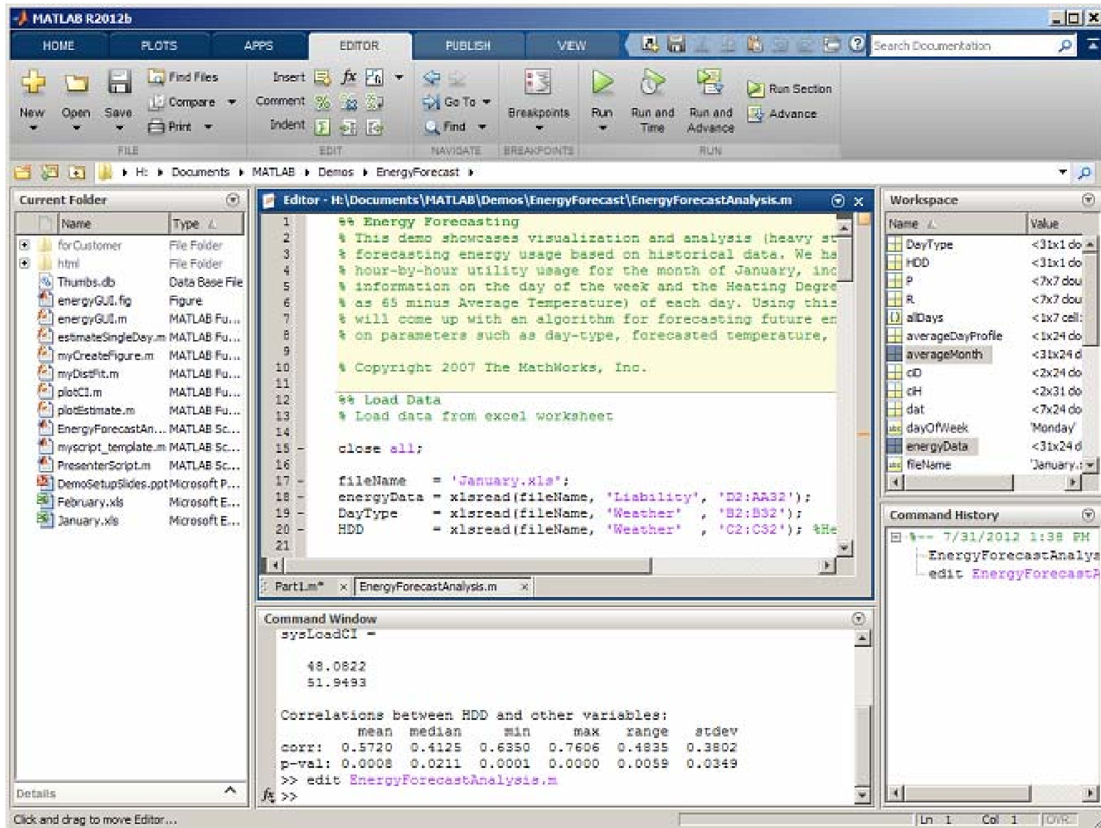
Contextual Tabs

As we've seen, global tabs are always present regardless of what you are doing in MATLAB. In addition to global tabs, the Toolstrip also has *contexual tabs*. Contextual tabs only appear when you're doing certain things in MATLAB. Let's look at the *Editor* as an example. When you edit a file, three new tabs appear -- the Editor tab, *Publish* tab, and *View* tab.

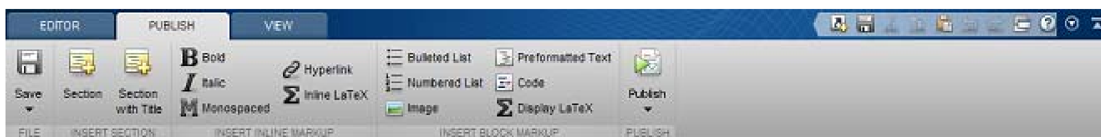
The *Editor* tab, shown above, contains all the functions you need when you're editing your file. All of those great capabilities of the *Editor* are there organized in a way to make them easier to find and use. This window opens over the Desktop layout.



In the upper right corner of the *Editor* window (and any other window) you will find *Actions* button. After clicking on it and choosing Dock option the *Command* window area in the Desktop becomes divided on two parts with the *Editor* window placed at the top and the *Command* window at the bottom. If the *Editor* is docked in the Desktop, those tabs related to the *Editor* appear next to the global tabs as shown below.



Publishing is a very useful feature in MATLAB. The *Publish* tab takes all the formatting controls you need to create beautiful MATLAB documents with publishing and puts them in a single place.



The *View* tab is the last of the *Editor* contextual tabs. It is where you go to control the layout and appearance of files in the *Editor*. You'll also find contextual tabs in the *Variable Editor*. These features of MATLAB will be discussed later in our practicals.

Minimizing the Toolstrip

To Minimize the Toolstrip right-click anywhere in the Toolstrip and select "Minimize Toolstrip" or double-click on any of the tabs. To restore the Toolstrip right-click anywhere on the Toolstrip and select "Restore Toolstrip" or double-click on any of the tabs. When the toolstrip is minimized it looks like this:

Appendix B: Summary of Tables

Practical 1

Mathematical Operators			
Symbol	**Operator**	**Order of Priority**	
`^`	Raise to a power	1	evaluated first
`*`	Multiplication	2	evaluated after any powers
`/`	Division	2	
`+`	Addition	3	evaluated after any powers, multiplication and division.
`-`	Subtraction	3	

Elementary Mathematical Functions	
`round(x)`	Rounds x to nearest integer
`floor(x)`	Rounds x down to nearest integer
`ceil(x)`	Rounds x up to nearest integer
`rem(y,x)`	Remainder after dividing y by x (*eg remainder of 17/3 is 2)*
`sign(x)`	Returns -1 if $x<0$; returns 0 if $x=0$; returns 1 if $x>0$
`rand or rand(1)`	Generates a random number between 0 and 1
`exp(x)`	Exponential function, e^x
`log(x)`	Natural logarithm function, $y=\ln x$ (where $e^y=x$)
`sqrt(x)`	Square root function, $\sqrt{x}$
`abs(x)`	Absolute value function, $\lvert x \rvert$
`sin(x)`	Sine function, $\sin x$
`cos(x)`	Cosine function, $\cos x$
`tan(x)`	Tangent function, $\tan x$

Practical 2

Table of the most useful commands for plotting graphs.	
`hold on`	Allows plotting several graphs on the same figure. All commands after this one apply to a current figure, until the `hold off` command is used.
`plot([x1 x2], [y1 y2])`	plots the graph
`plot([x1 x2], [z1 z2])`	
`axis([minX maxX minY maxY])`	specifies axis limits
`legend('y1=equation1', 'y2=equation2')`	sets the labels for the legend *(must be after plot command)*
`xlabel('label for x axis')`	sets the label for the x - axis
`ylabel('label for y axis')`	sets the label for the y - axis
`title('graph title')`	sets the label for the title
`hold off`	Following commands no longer apply to the current figure.

Practical 3

Array Functions	
Function	**Description**
`sum(Z)`	adds all elements of array `Z`
`prod(Z)`	multiplies all elements of array `Z`
`length(Z)`	returns the length of array `Z`
`[Zmax, i] = max(Z)`	returns largest element `Zmax of Z` and its position, `i`
`[Zmin, i] = min(Z)`	returns smallest element `Zmin` of `Z` and its position, `i`
`sort(Z)`	sorts elements of array `Z` in ascending order
`find(Z>3)`	finds the **indices** of elements of array `Z` larger than 3

Dot Multiplication	
`A.*B`	multiplies corresponding elements of `A` and `B`
`A./B`	divides corresponding elements of `A` and `B`
`5./A`	divides 5 by each element of `A`
`A.^B`	calculates `A(i)^B(i)` for every corresponding element in `A` and `B`
`2.^A`	finds `2^A(i)` for every element of `A`
`A.^3`	cubes each element in `A`

Practical 4

for- and while- Loops and if- Statements Structure

for**- loop syntax**

```
for <variable> = <start>:<step>:<finish>
     <commands>…..
     ……………..
end
```

while**- loop syntax**

```
while <condition>
     <commands>…..%executed if condition is true
     ……………..
end
```

Complex if-elseif-else-end **statements:**

```
if <expression 1>
     <command> % evaluated if expression 1 is true
elseif <expression 2>
     <command> % evaluated if expression 2 is true
elseif expression3
     <command> % evaluated if expression 3 is true
….
else
     <command> % evaluated if none of expressions are 
true
end
```

Relational Operators

Symbol	Example	Meaning of Example
<	a<6	true if a is less than 6
<=	a<=6	true if a is less than or equal to 6
>	a>6	true if a is greater than 6
>=	a>=6	true if a is greater than or equal to 6
==	a==6	true if a is equal to 6 *(note double equals sign)*
~=	a~=6	true if a is not equal to 6

Boolean operators

Symbol	Definition	Example	Meaning of Example
&	And	(a>2) & (a<6)	a is greater than 2 *and* less than 6
|	Or	(a<2) | (a>6)	a is less than 2 *or* greater than 6
~	Not	~(a==6)	a is *not* equal to 6

Practical 5

Functions operating on functions	
`fplot(@demoFun,[a b])`	Plots the function for `a ≤ x ≤ b` without setting up arrays
`fzero(@demoFun,[a b])`	Finds one value of `x` for `demoFun(x) = 0` provided that the signs of `demoFun(a)` and `demoFun(b)` are opposite.
`fzero(@demoFun,c)`	Finds one value of `x` for `demoFun(x) = 0` by starting a search at `x = c`
`fminbnd(@demoFun,a,b)`	Finds the coordinates of a minimum point of `demoFun(x)` at the interval `a ≤ x ≤ b`
`quadl(@demoFun,a,b)`	Finds an accurate value for $\int_a^b y(x)dx$ Note: *quadl uses arrays, so therefore you must set your function up treating x as an array and so using the dot notation for operations.*

Practicals 6 & 7

Polynomial Functions	
`roots([c1 c2 c3])`	finds the roots of a polynomial given its array of coefficients, `[c1 c2 c3]`.
`poly([r1 r2])`	constructs a coefficient array, given the roots, `[r1 r2]`, of the desired polynomial.
`polyval(p,x)`	finds the value of the polynomial with coefficient array `p` at any given number `x`.
`conv(p,q)`	multiply two polynomials, $p(x)$ and $q(x)$, with coefficient arrays, `p` and `q`
`[Q,R]=deconv(p,q)`	finds the quotient and remainder of $p(x)/q(x)$

Symbolic Toolbox Commands	
`sym('x')` or `syms x`	Creates a symbolic variable x
`sym(1/4)`	Creates symbolic number $1/4$
`double(1/4)`	Converts symbolic number $1/4$ to standard 0.25
`subs(f,x,a)`	Substitutes variable x in $f(x)$ with value a and finds $f(x)$
`x=a` `eval(f)`	Finds value of $f(x)$ at a by assigning a to x first.
`ezplot(f,a,b)`	Plots the symbolic function $f(x)$ over the interval $[a,b]$
`pretty`	Formats output to look like type-set mathematics
`simplify`	Performs algebraic and other function simplifications
`simple`	Tries a number of simplification techniques including trigonometric identities. Running this command twice can further simplify complex expression.
`collect`	Collects like terms
`factor`	Attempts to factor the expression
`expand`	Expands all terms
`poly2sym`	Converts an array of polynomial coefficients into a symbolic expression with x as the variable
`sym2poly`	Converts symbolic polynomial into its coefficient array
`diff(f,x)`	Finds 1st derivative of $f(x)$ with respect to x
`diff(f,x,n)`	Finds n^{th} derivative of $f(x)$ with respect to x
`int(f,x)`	Indefinite integral of $f(x)$
`int(f,x,a,b)`	Definite integral of $f(x)$ with endpoints a, b