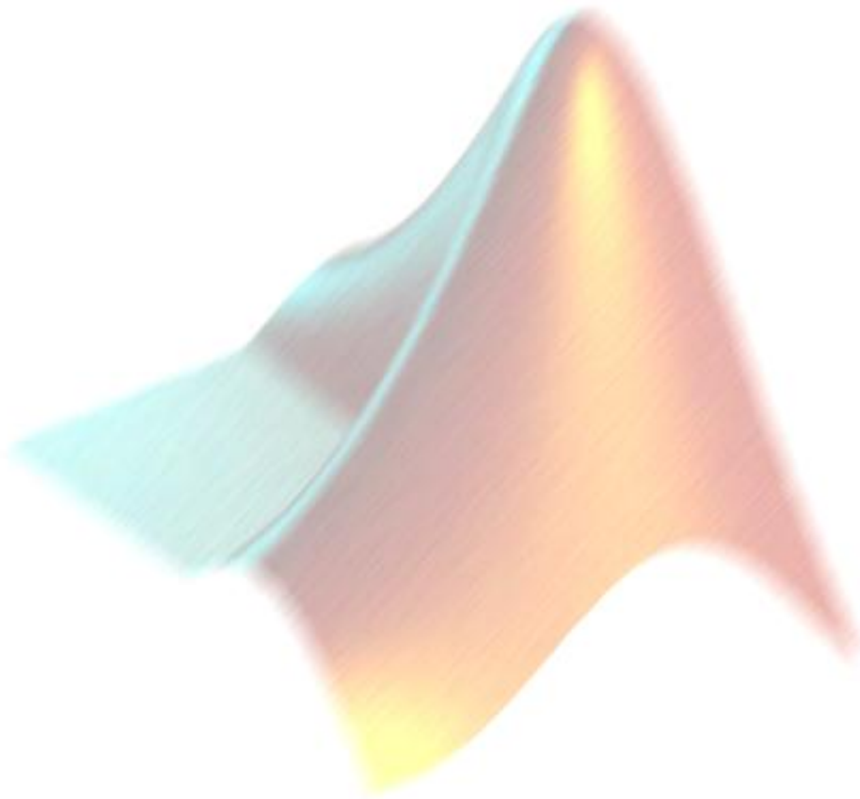
THE UNIVERSITY OF AUCKLAND
FACULTY OF ENGINEERING

AP Induction Week Course

Introduction to Engineering Computation

Laboratory Manual

MATLAB programming



Department of Engineering Science

MATLAB Laboratory Manual Contents

Laboratory 1: An introduction to MATLAB 2
Laboratory 2: Debugging, Functions and Problem Solving 22
Laboratory 3: Logical Operators, Conditional Statements and Loops 32
Laboratory 4: Graphics and Image Processing 46
Laboratory 5: Strings and Files 52
Laboratory 6: Linear equations and differential equations 57

Laboratory 1: An introduction to MATLAB

GETTING STARTED

Your Tutor will show you how to login to the Network. Most of the applications you will use reside on File Servers rather than on the local hard drive of your computer. You have also been allocated an ec (Electronic Campus) home directory (H drive), in which you can save your work.

If your computer is switched off (you can tell this by the absence of a light on the front of the computer) you should press the Power on the computer's front panel.

Do not switch the monitor on or off at any time!

If the light is already on, you can simply restart the login sequence by holding down the Ctrl and the Alt keys and pressing the Delete key.

GENERAL LAB DIRECTIONS

You should work your way through the tutorials reading everything. Interaction with the computer takes place throughout the lab. The sections are not stand-alone so you must work through them in order. There are a number of assignment tasks throughout the Lab that will be checked by your tutor when you have completed the **whole lab**. Any MATLAB commands for you to type are shown as they would appear in MATLAB. Other MATLAB terms in a body of text will be bolded to make it easy to see. Please feel free to help your fellow students understand the concepts and never hesitate to ask questions of the tutors.

Some labs include optional exercises at the end. These exercises are aimed at students who are able to complete the other lab tasks quickly and who wish to learn/practice more MATLAB. Completion of optional tasks is not required to get signed off for the lab. The material presented in optional exercises will often increase your understanding of MATLAB so if you have a chance give these exercises a go.

Be sure to look out for

TASK There is a lab task to complete.

TIP There is a handy tip to help you with MATLAB.

Star There is a description of good programming practice. If you follow these tried and true conventions you will be a programming star!

Stop There is a "roadblock to understanding". Make sure you ask a tutor for clarification if you don't understand what MATLAB is doing.

INTRODUCTION TO MATLAB

MATLAB (**MAT**rix **LAB**oratory) is an interactive software system for numerical computations and graphics. As the name suggests, MATLAB is especially designed for matrix computations: solving systems of linear equations, performing matrix transformations, factoring matrices, and so forth. In addition, it has a variety of graphical capabilities, and can be extended through programs written in its own programming language.

MATLAB is supported on Unix, Macintosh, and Windows environments. A student version of MATLAB for Macintosh or Windows may be purchased from the Science Student Resource Centre, which you can install on your own computer.

MATLAB Advantages

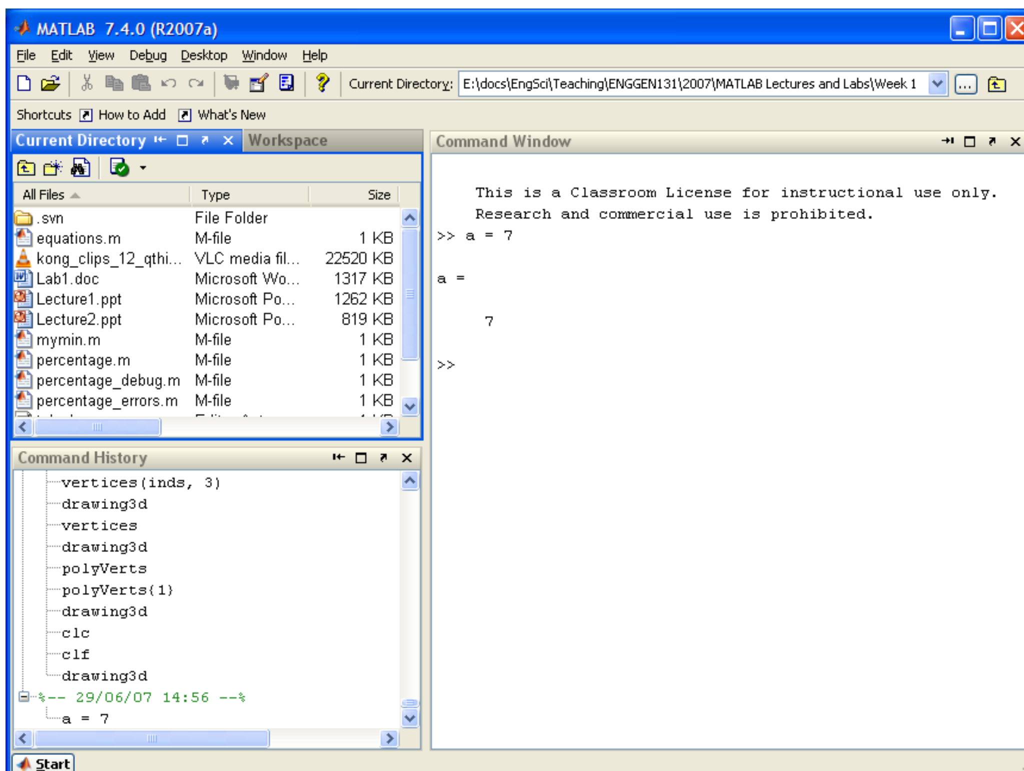
- ✓ It simplifies the analysis of mathematical models
- ✓ It frees you from coding in lower-level languages (saves a lot of time - with some computational speed penalties)
- ✓ Provides an extensible programming/visualization environment
- ✓ Provides professional looking graphs

MATLAB Disadvantages

- × It is an interpreted (i.e., not pre-compiled) language, so it can be slow.

GETTING STARTED

Double click on the MATLAB icon. The MATLAB window should come up on your screen. It looks like this:

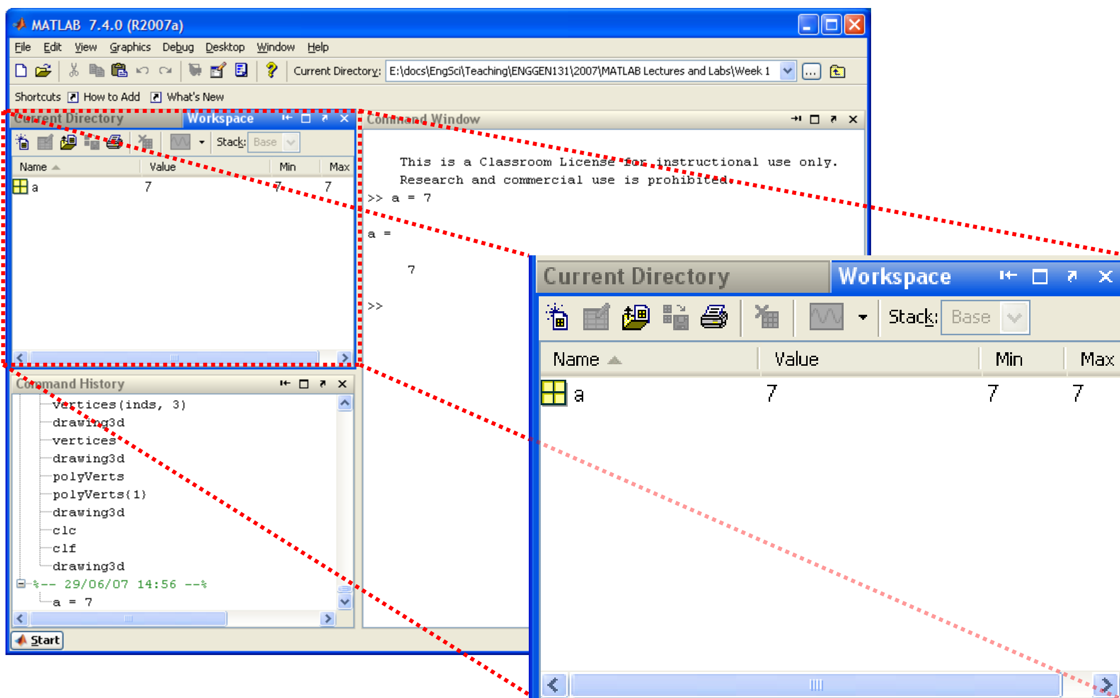


Finding Your Way Around the Windows

The main window on the right is called the *Command* Window. This is the window in which you interact with MATLAB. Once the MATLAB prompt **>>** is displayed, all MATLAB commands are executed from this window. In the figure, you can see that we execute the command:

```
>> a = 7
```

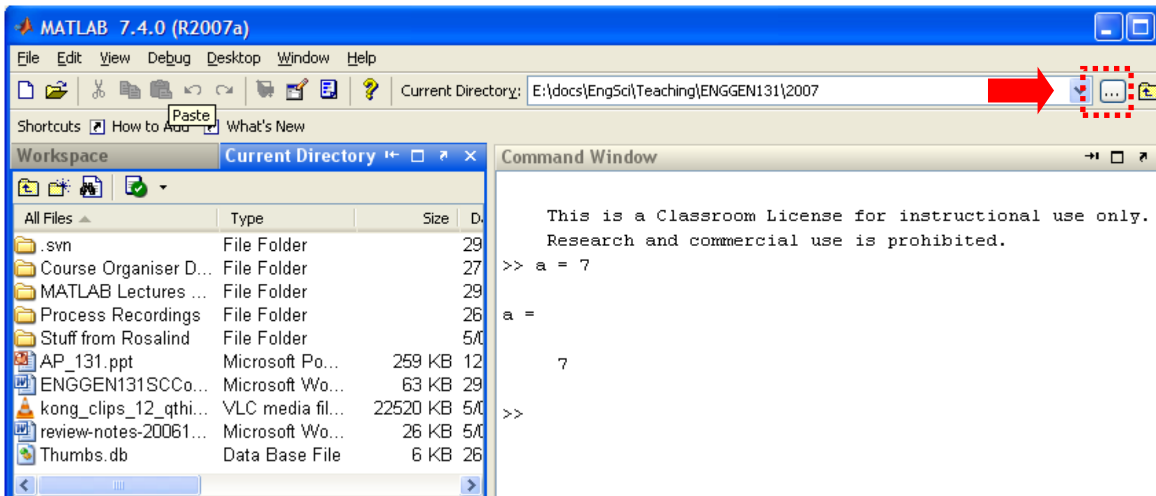
In the top left corner you can view the *Workspace* window and the *Current Directory* window. Swap from one to the other by clicking on the appropriate tab. Note that when you first start up MATLAB, the workspace window is empty. However, as you execute commands in the *Command* window, the *Workspace* window will show all variables that you create in your current MATLAB session. In this example, the workspace contains the variable **a**.

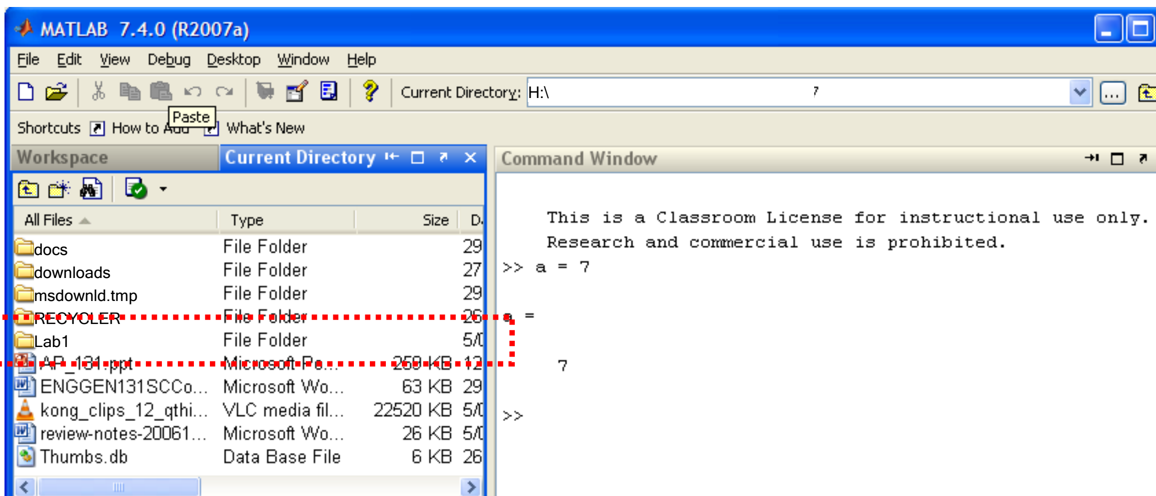


In the bottom left corner you can see the *Command History* window, which simply gives a chronological list of all MATLAB commands that you used.

During the MATLAB sessions you will create files to store programs or workspaces. Therefore create an appropriate folder to store the lab files.

Set the current directory to be your H drive by clicking on the *Browse* button next to the *Current Directory* path. Go to *My Computer* (at the top), click on the + and select *H*.





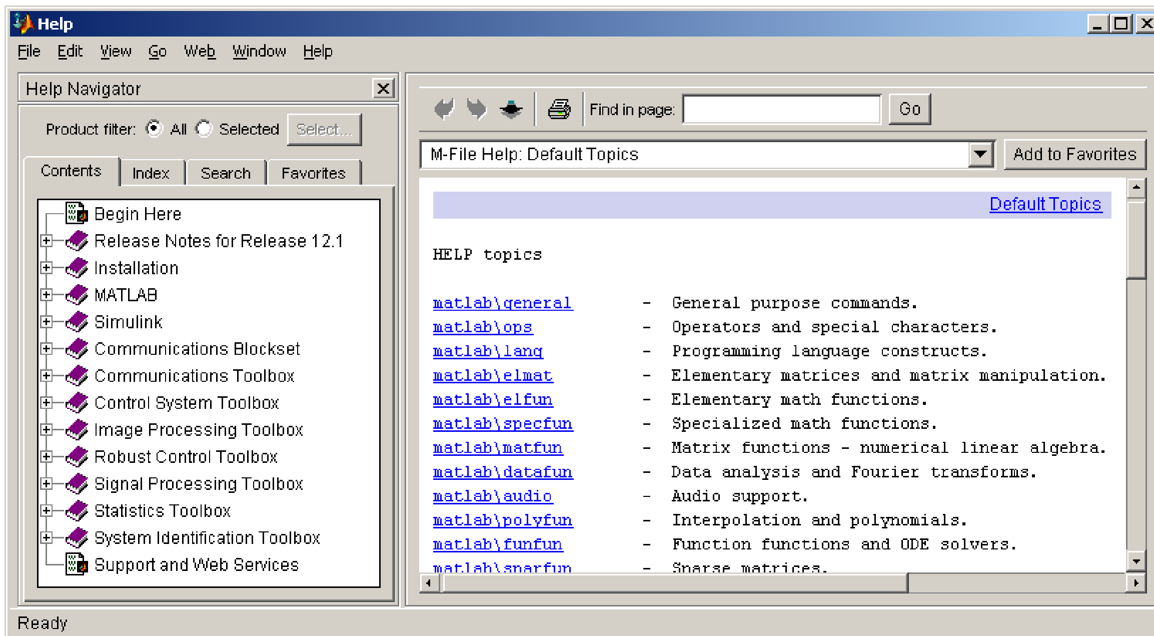
Find the *New folder* button on the menu and name the new folder as **Lab1**. Now double click the *Lab1* folder you just created so that MATLAB will automatically save files in this folder.

Note that the current directory (or folder - <u>folder and directory mean the same thing</u>) is also displayed in the top right corner next to the main menu.

You will get to know the various buttons and menus on the MATLAB window in the next few labs. We will not discuss all of them. Therefore, play around and familiarise yourself!

THE MATLAB HELP SYSTEM

MATLAB's help system provides information to assist you while using MATLAB. Select *Help* → *MATLAB Help* to open the help system.



You can then browse the commands via the *Contents* window, look through the *Index* of commands or *Search* for keywords or phrases within the documentation. This is the most comprehensive documentation for MATLAB and is the best place to find out what you need. You can also start the MATLAB *Help* using the **`helpwin`** command.

A faster way to find out information about commands is via the **`help`** command. If you just type **`help`** and hit return, you will be presented with a menu of help options. If you type **`help`** **<command>**, MATLAB will provide help for that particular command. For example, **`help hist`** will call up information on the **`hist`** command, which produces histograms. Note that the **`help`** command only displays a short summary of how to use the command you are interested in. For more detailed help documentation, use the **`doc`** command. For example, **`doc hist`** will display the full documentation of the **`hist`** command, including examples of how to use it and sample output.

Another very useful MATLAB command is the **`lookfor`** command. This will also search for commands related to a keyword. Find out which MATLAB commands are available for plotting using the command **`lookfor plot`** in the command window. It may take a bit of time to get all commands on the screen. You can stop the search at any time by typing *ctrl-c* (the *ctrl* and the *c* key simultaneously).

A list of the most basic MATLAB commands and functions is supplied in the appendix of your Course Manual.

WORKING AT THE COMMAND LINE

Our initial use of MATLAB was as a calculator. Refer to chapter 1 to see what the mathematical operators are and to remind yourself what BEDMAS stands for. Let's try some of the basic operations. Try typing the following:

```
>> 5 + 4
ans =
     9
>> 5 - 4
ans =
     1
>> 5 * 4
ans =
     20
>> 5 / 4
ans =
     1.2500
>> 5^4
ans =
     625
>> 34^16
ans =
     3.1891e+024
```

The last `ans` (the result of the last calculation) is a quantity expressed in scientific notation. MATLAB uses scientific notation for very large numbers and very small numbers. MATLAB has a special way of representing this:

$$34^{16} = 3.1891 \times 10^{24}$$

```
>> 34^16
ans =
        3.1891e+024
```

Note that as you type commands they are added to the command history window, giving you a record of everything you have typed for this MATLAB session.

The first two tasks require you to type some simple MATLAB commands at the command prompt. When you have finished all the lab tasks you will need to show a tutor these commands to prove you have done the tasks.

You may like to handwrite the answers for these first two tasks in your lab book to show your tutor later. Alternatively you can simply scroll back through the command history to show your tutor the commands you typed. Note that if you close down MATLAB your command history is erased.

TASK 1 Using MATLAB Help

Below is a table containing some expressions you could use a calculator to compute. To familiarise yourself with entering expressions into MATLAB, complete the table. Use MATLAB *Help* to look up any functions you don't know, e.g., `>> lookfor log` or `>> help log`

Remember that $\ln(x)$ is the natural logarithm and that the function e^x is called the exponential function. Also, notice that the last expression produces a complex number (which MATLAB represents using the special variable `i` for the imaginary part).

Expression	MATLAB Expression	MATLAB Result
$\frac{1}{\sqrt{2\pi}}$	`1 / sqrt(2 * pi)`	
$5\times10^9+3\times10^8$	`5e9 + 3e8`	
$\log_{10}72$		`1.8573`
$\cos\pi$		`-1`
e^1		`2.7183`
$\ln(\sin(\pi^2))$		`-0.8433 + 3.1416i`

TIP Getting tasks signed off

Remember that you must complete **ALL** the lab tasks before getting your tutor to sign you off. Continue with the remainder of the lab before asking your tutor to check your tasks.

For all but the first two tasks you will be creating files. Remember to open **ALL** the files you have written **before** asking your tutor to check you off.

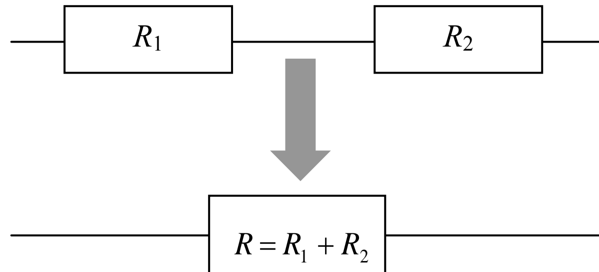
When you are being checked off you can expect that your tutors will ask you some questions about the tasks, to check your understanding of the lab material. As you work through the labs make sure you understand what you are doing and please ask questions if you are confused.

Note that your tutor may refuse to sign you off if it is obvious you do not understand the material. In that case you may need to reread some of the course notes and repeat parts of the lab until you have learnt the material.

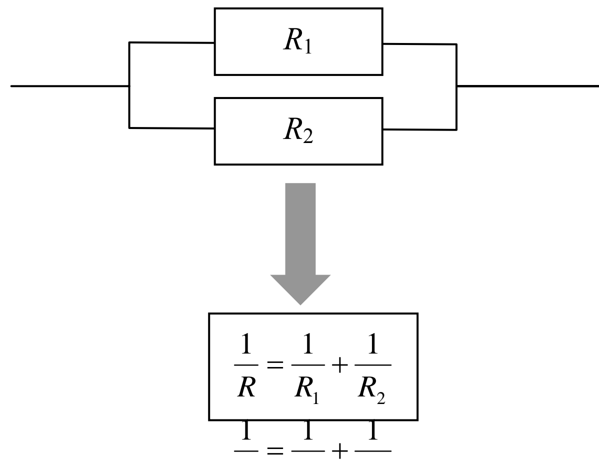
Now that you have become used to the MATLAB Command Line, let's use it to do some basic electrical engineering calculations.

TASK 2 Basic Electrical Circuits

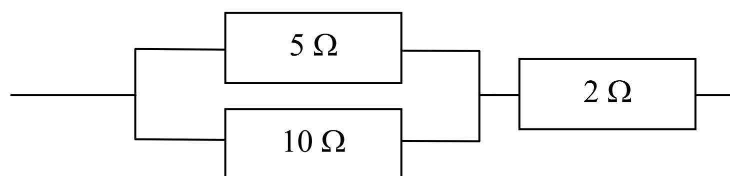
If 2 resistors (resistance R_1 and R_2 measured in ohms) are placed in series in a circuit, they may be represented as a single resistor (resistance R, also in ohms) where



If 2 resistors (resistance R_1 and R_2) are placed in parallel in a circuit, they may be represented as a single resistor (resistance R) where



Now, use MATLAB to calculate the resistance of the following configuration of resistors



You may like to write the commands you use below:

MATLAB VARIABLES

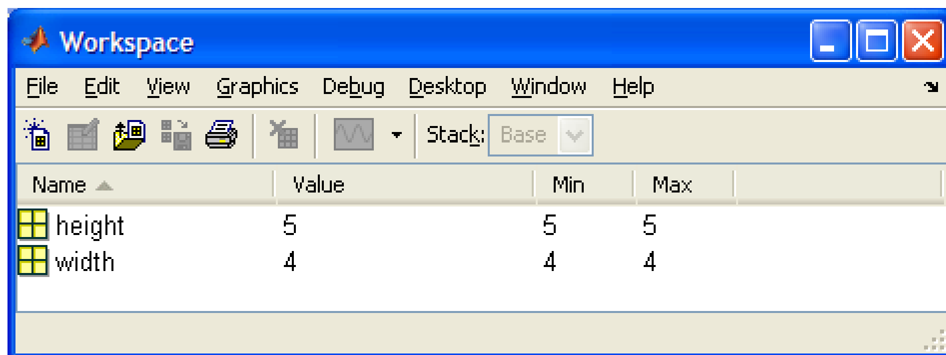
As we discussed, MATLAB stands for 'MATrix LABoratory'. This title is appropriate because the structure for the storage of all data in MATLAB is a matrix. We talk more about this in upcoming lectures. Right now we will only work with scalar variables. MATLAB stores these as matrix variables with 1 row and 1 column. These variables show in your workspace is shown as 1x1 matrices.

Assigning Variables

To create a (scalar) variable in MATLAB simply enter a valid variable name at the command line and assign it a value using **=**. Once a variable has been assigned a value it is added to the MATLAB Workspace and will be displayed in the *Workspace* window. For example, after entering:

```
>> height = 5
height =
     5
>> width = 4
width =
     4
```

the *Workspace* window will contain both **height** and **width** as scalars:



We should *NOT* use **length** as a variable name as this is a built-in MATLAB function.

TIP *Rules on Variable Names*

There are a number of rules as to the variable names you can use:

1. must start with a letter, followed by letters, digits, or underscores. **X12**, **rate_const**, **Flow_rate** are all acceptable variable names but not **vector-A** (since **-** is a reserved character);
2. are case sensitive, e.g., **FLOW**, **flow**, **Flow**, **F1Ow** are all different variables;
3. must not be longer than 31 characters;
4. must not be a reserved word (i.e., special names which are part of MATLAB language);
5. must not contain punctuation characters;

Be careful not to confuse the number **1** with the letter **l**, or the number **0** with the letter **O**.

Star *Naming Variables*

You should choose variable names that indicate quantities they represent, e.g., **`width`**, **`cableLength`**, **`water_flow`**. You can use multiple words in the same variable name either by capitalizing the first letter of each word after the first one, e.g., **`cableLength`**, or by putting underscores between words, e.g., **`water_flow`**.

At the rear of the course manual is a style guide which indicates the naming conventions we recommend you follow for this course. Not all the code in this course manual follows these conventions as we wish to expose you to a range of conventions. This is important since you may need to read or use code written using different styles. You may also prefer using a different convention.

If we do (accidentally) use **`length`** (or another built-in function name) as a variable name we must remove it from the MATLAB Workspace to restore the built-in function.

TIP *Some MATLAB Workspace Functions*

You can see the variables in your workspace by looking at the *Workspace* window or by using the **`whos`** command:

```
>> whos
  Name        Size            Bytes  Class     Attributes

  area        1x1                 8  double
  height      1x1                 8  double
  width       1x1                 8  double
```

If you want to remove a variable from the MATLAB you can use the **`clear`** command, e.g.,

```
>> clear width
```

removes width from the workspace. If you want to get rid of all the variables in your workspace just type:

```
>> clear
```

Calculations with Variables

Once you have added variables to the MATLAB Workspace, you can use them within calculations (just like you have used the special variable **`pi`** in Task 1). For example, to calculate the area of a 5 x 4 rectangle you could enter:

```
>> height = 5
>> width = 4
>> area = height * width
```

ERROR MESSAGES

If your command is invalid MATLAB gives you explanatory error messages (luckily!). Read them carefully. Normally MATLAB points to the exact position of where things went wrong.

Enter:

```
>> height* = 5
```

The result will be:

```
??? height* = 5
            |
Error: The expression to the left of the equals sign is not a
valid target for an assignment.
```

The command has failed. MATLAB attempts to explain how/why the command failed.

Stop ***Invalid Expression***

Make sure you know why the command failed in MATLAB. Ask a tutor if you are unsure.

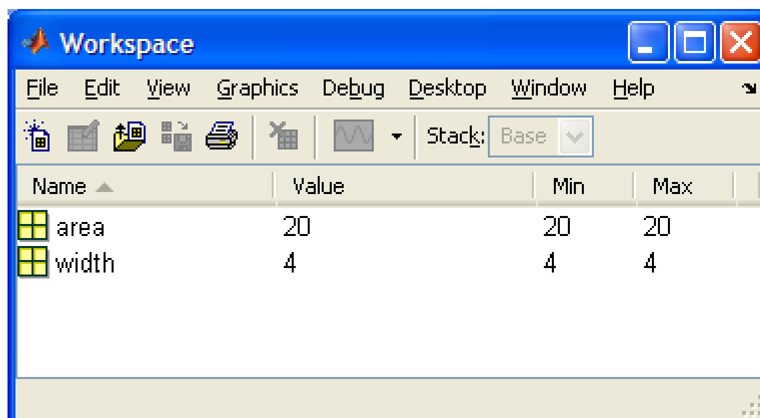
FINDING OLD COMMANDS

Suppose you want to repeat or edit an earlier command. If you dislike typing it all back in, then there is good news for you: MATLAB lets you search through your previous commands using the *up-arrow* and *down-arrow* keys. This is a very convenient facility, which can save a considerable amount of time.

Example. Clear the height of the rectangle in the earlier example:

```
>> clear height
```

and look at the *Workspace* window:



Now you want it back! Press the *up-arrow* key until the earlier command:

```
>> height = 5
```

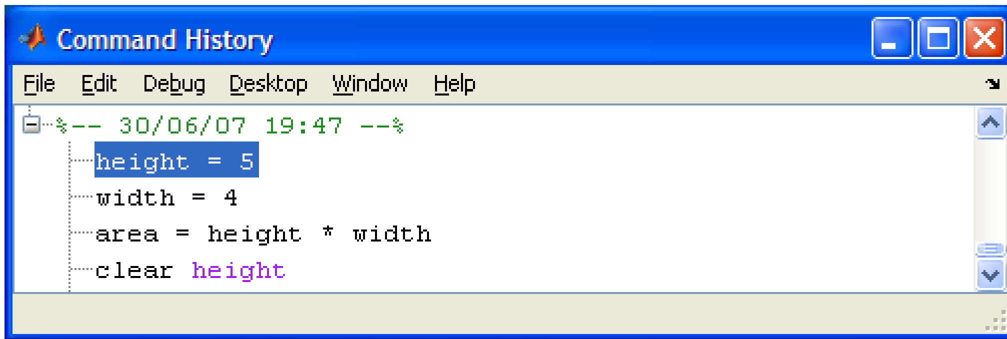
appears in the *Command* Window (the *down-arrow* key can be used if you go back too far). Now press enter. Now **`height`** is back to what it was originally.

You can speed up the scroll if you remember the first letters of the command you are interested in. For example, if you quickly want to jump back to the command:

```
>> height = 5
```

just type **h** and then press the *up-arrow* key. MATLAB will display the most recent command starting with **h**. Alternatively, you can use *Copy* and *Paste* under *Edit*.

You can also use the *Command History* window to jump to a previous command. Go to the *Command History* window and double-click on the line **height = 5**:



The command:

```
>> height = 5
```

will appear in the *Command Window* and be carried out immediately.

SUPPRESSION OF OUTPUT

Up till now you have always seen the results of each command on the screen. This is often not required or even desirable; the output might clutter up the screen, or be long and take a long time to print. MATLAB suppresses the output of a command if you finish the command with a semi-colon - `;`.

Enter:

```
>> height = 5
```

You will notice that MATLAB prints out the result of this command after you press enter:

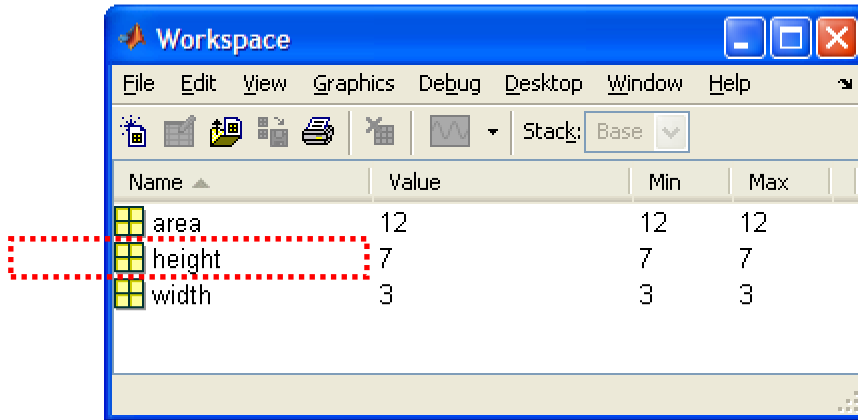
```
height =

      5
```

Now enter:

```
>> height = 7;
```

MATLAB will not print the result of the command. With scalar variables this doesn't make much difference - but it can make a great deal of difference when dealing with large lists of numbers.

If you look at **height** in your *Workspace* window you will see its value has changed to 7:



SAVING YOUR WORK - SCRIPT FILES

So far all your work has been at the command line. This is useful for quick calculations. For calculations that are more complex, or that you are likely to perform often it is much more useful to save your work in script files. We will use script files from here onwards in the course.

To create a new script file (also called an M-file) choose *File→New→M-File* from the MATLAB menu bar. This will open a blank script editor window. You can enter commands into this window using the same syntax as you would at the MATLAB command line. These commands can be saved, repeated and edited.

Let's start by creating a script file to calculate the area of a rectangle (one of our earlier examples). In the script file window enter:

```
height = 5
width = 4
area = height * width
```

Now save your script file by choosing *File→Save* from the menu in the script file editor. Give your script file a name, such as **area_calc.m**. The **.m** extension is used to denote MATLAB script files. The file will be saved in the working directory you set earlier in the lab.

Stop ***Invalid Filenames***

Filenames cannot include spaces. They also cannot start with a number, include punctuation characters (other than the underscore) or use a reserved MATLAB command. If you use an invalid filename you will get some unusual errors when you try to run your script. Check your filename now to see if it is valid.

Now go back to the MATLAB command line (you can click on the MATLAB icon in the bar at the bottom of the screen or use *Alt-Tab* to get there). First, **`clear`** the workspace. Now, at the command prompt enter:

```
>> area_calc
```

You will see the **`height`**, **`width`** and **`area`** variables appear with their values.

TIP *Long Lines in Script Files*

If you have a very long command in your script file, it can be difficult to see the whole command at once, e.g.,

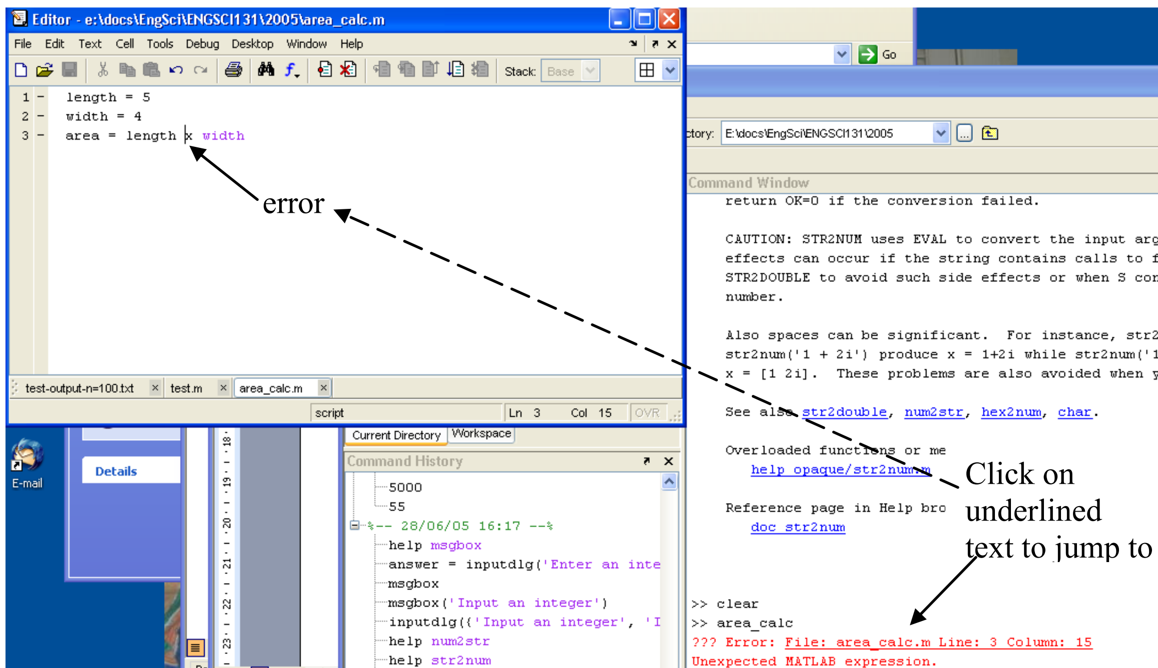
```
21
22 -  = 1:length(mintension),
23 -  sp(['Attempt ', num2str(i), ': Length = ', num2str(length(i)), 'm']);
24 -   isSafe,
```

You can split the command into multiple lines, but you must add three dots, i.e., **`...`**, to make sure MATLAB knows it is all one command, e,.g.,

```
22 -  for i = 1:length(mintension),
23 -      disp(['Attempt ', num2str(i), ': Length = ', ...
24            num2str(length(i)), 'm']);
```

Errors in Script Files

If you make a mistake in a script file MATLAB will let you know with an error message. It will also let you jump to the (likely) source of the error by clicking on the error text.



Error message

Change **area_calc.m** so the area calculation has an error:

```
height = 5
width = 4
area = height x width
```

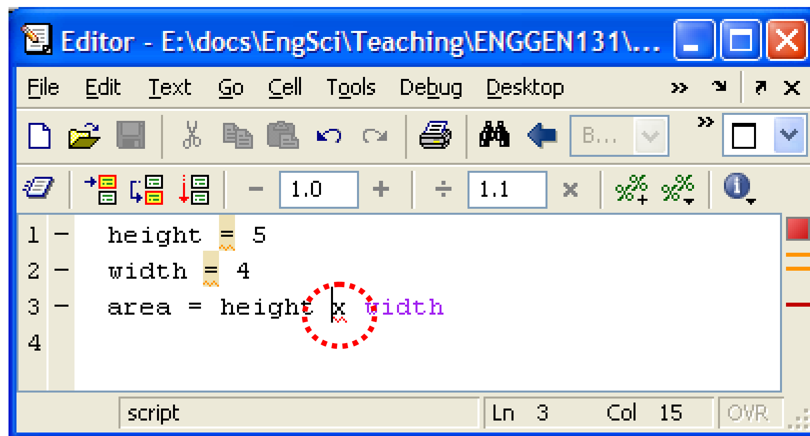
and run the script file again:

```
>> area_calc
```

You should get an error:

```
??? Error: File: area_calc.m Line: 3 Column: 15
Unexpected MATLAB expression.
```

By clicking on the underlined text you will jump to `height x width` (and the error):



Stop Script File Error

Make sure you know why MATLAB gave an error. Ask a tutor if you are unsure.

COMMENTING

One of the most important facets of programming is documenting your program, or *commenting*. Comments lines begin with the `%` character and turn green in the MATLAB editor. Comments allow you to document your program to make it clear what steps you are taking. Throughout this course you are expected to have useful comments throughout your code. The expectation is that another person could rewrite (most of) your program starting only with the comments.

TIP Comments as Help

The first *block* of comments in a script file are shown if the command **help filename** is entered in the *Command Window*. For more information search for "Help Text" in the MATLAB *Help*.

Star *Good Commenting*

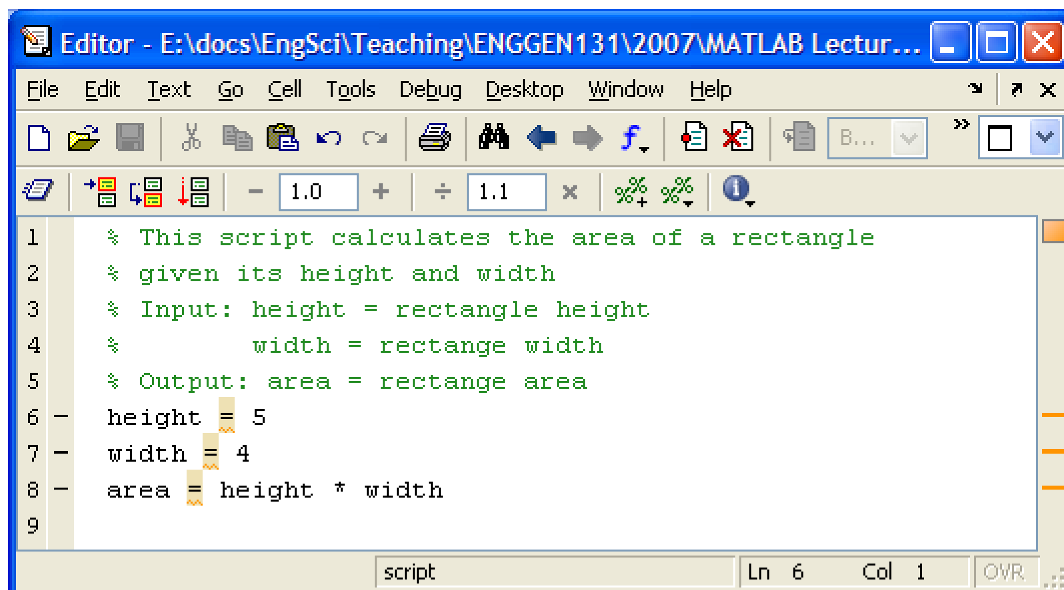
Your initial commenting block should have a brief description of the script file functionality and also clearly define the input and output for the script file. For example:

area_calc.m

```
% This script calculates the area of a rectangle
% given its height and width
% Input: height = rectangle height
%        width = rectangle width
% Output: area = rectangle area

height = 5
width  = 4
area = height * width
```

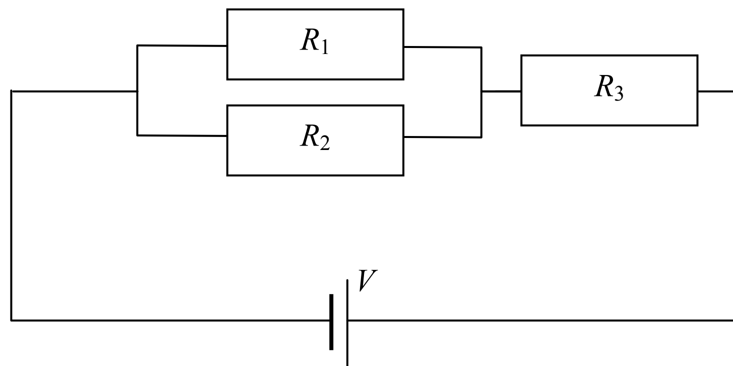
```
>> help area_calc
  This script calculates the area of a rectangle
  given its height and width
  Input: height = rectangle height
         width = rectangle width
  Output: area = rectangle area
```

You should also have (at least) one line of commenting for any command in your script file that is non-trivial. For example:



TASK 3

Consider the following electrical circuit:



The current through the circuit (in amps) is determined by the voltage through the circuit (in volts) divided by the resistance (in ohms) through the circuit.

Write a script file to calculate the current through the circuit for any combination of R_1, R_2, R_3 and V. Test your script file using:

$R_1 = 5\ \Omega$, $R_2 = 10\ \Omega$, $R_3 = 2\ \Omega$ and $V = 6$ V (the current is 1.125 A).

Be sure to include good commenting in your script file.

BASIC USER INTERACTION

Basic User Input

Script files are more useful if they can interact with users. One simple way to do this is via the **`input`** command. You can use it to prompt the user for a value to assign to a variable. For example, rather than editing **`area_calc.m`** each time the values for **`height`** and **`width`** change you could use

```
height = input('Enter the height of the rectangle: ')
width  = input('Enter the width of the rectangle: ')
```

Try this out by opening your **`area_calc.m`** script file and saving it as **`area_calc_input.m`**. Now replace

```
height = 5
width = 4
```

with the input commands given above.

Now enter **area_calc_input** at the MATLAB command prompt

```
>> area_calc_input
Enter the height of the rectangle: 3
height =
     3
Enter the width of the rectangle: 6
width =
     6

area =
    18
```

You can use `;`'s to suppress the output, e.g.,

```
height = input('Enter the height of the rectangle: ');
width  = input('Enter the width of the rectangle: ');
```

so now only the result of the area calculation is displayed

```
>> area_calc_input
Enter the height of the rectangle: 4
Enter the width of the rectangle: 3

area =

    12
```

Basic Formatted Output

MATLAB's basic output is ok for inspecting the value of variables, but it is not very informative. You can control your output more using the `disp` command. Once you have calculated **area** you can suppress MATLAB's output and then write its value along with some extra information

```
area = height * width;
disp('The area of the rectangle is: ')
disp(area)
```

Using **disp** this way and entering 5 and 4 for height and width gives the following output

```
>> area_calc_input
Enter the height of the rectangle: 5
Enter the width of the rectangle: 4
The area of the rectangle is:
    20
```

If you want to put the value of **area** on the same line as the text you can use the basic formatting function **num2str** and *concatenate* the text and the formatted **area** together.

```
disp(['The area of the rectangle is: ' num2str(area)])
```

We'll learn more about why this works later in the course.

TASK 4

Modify your script file from Task 3 to get user input for R_1, R_2, R_3 and V.

Display the total resistance of the circuit and current through the circuit using **`disp`**. Your output should look something like this:

```
The total resistance for the circuit (in ohms) is:
    5.3333
The total current through the circuit (in amps) is:
    1.1250
```

Test your script file using:

$R_1 = 5\ \Omega$, $R_2 = 10\ \Omega$, $R_3 = 2\ \Omega$ and $V = 6$ V (the current is 1.125 A)

and:

$R_1 = 8\ \Omega$, $R_2 = 4\ \Omega$, $R_3 = 5\ \Omega$ and $V = 9$ V (the current is 1.1739 A).

Laboratory 2: Debugging, Functions and Problem Solving

DEBUGGING

In lectures you saw how MATLAB helps you find bugs in your code. Using the Matlab debugger (called *M-Lint),* you can identify errors while you edit code. There is also the *Debug* menu for stepping through your programs to identify and fix errors.

TASK 2 *Debugging the Final Percentage Example*

Recall the script file for calculating your 131 final percentage. This script file **FinalPercentage.m** is available for download from cecil:

http://www.auckland.ac.nz/cecil

When downloading any script files from cecil make sure you check that the filename is correct and then place the file founinto an appropriate directiory.

and is shown below:

```
% This script file calculates your 131 final percentage
% from your coursework percentage and your exam percentage
% Inputs: C = coursework percentage
%         E = exam percentage
% Output: F = final percentage
clear;

% Get coursework percentage C
C = input('Enter coursework percentage > ');

% Get exam percentage E

% Calculate C + 10
C + 10 = Cinc;

% Calculate E + 10
Einc = E + 10

% Calculate (C + E) / 2
Avg = (C + E / 2;

% Set final percentage F to be minimum of C + 10, E + 10, (C + E)
/ 2
F = min([Cinc, Einc, Avg])
```

Open this script in the MATLAB *Editor*, fix any bugs and use it to calculate your final percentage if you achieved 90% for your coursework and 83% for your exam.

FUNCTIONS

Consider the task of writing a function to convert a metric measurement (in metres) to an imperial measurement in feet and inches. Generally it is much easier to work with metric measurements but sometimes people want results given to them in imperial format. A conversion function could be used by any program which needs to display output in imperial format.

Recall The 5 steps for problem solving:

1. State the problem clearly
2. Describe the input and output information
3. Work the problem by hand (or with a calculator) for a simple set of data
4. Develop a solution and convert it to a computer program
5. Test the solution with a variety of data

We will work through our five steps to develop a function that converts metric measurements to imperial measurements.

Step 1: State the problem clearly

Write a function to convert a metric measurement (in metres) to an imperial measurement (in feet and inches).

Step 2: Describe the input and output information

Fill in the missing labels on the I/O diagram

Step 3: Work the problem by hand for a simple set of data

We will convert 2 metres to the imperial equivalent. We know that there are 12 inches to a foot and that one inch is the equivalent of 2.54 cms.

2 metres is 200 cm. Now calculate the number of inches:

$$200/2.54 = 78.7402$$

Now figure out how many feet we have by dividing the total number of inches by 12:

$$78.7402/12 = 6.5617$$

We have 6 feet. To calculate the number of inches remaining we will need to subtract off the number of inches in 6ft from the total number of inches:

$$78.7402 - 6 \times 12 = 6.7402$$

So 2 metres is 6 feet 6.7402 inches.

Stop ***Make sure you understand the hand worked example***

Before continuing make sure you have understood how we converted 2 metres to 6 feet 6.7402 inches. You may find it helpful to work through another example by hand. Try converting 1.5 metres to feet and inches to get the answer.

Step 4: Develop a solution and convert it to a computer program

Our pseudocode is as follows:

INPUTS: m

- Calculate the total number of cms
- Convert number of cms to the total number of inches
- Calculate the total number of feet, ignoring the value after the decimal point
- Find the remaining number of inches

OUTPUTS: ft and in

TIP ***Pseudocode can be used as comments***

Often it can be helpful to begin your code by writing comments that describe what you want to do. You can then fill in the required code. In many cases you can simply use your pseudocode as comments.

Download the MetricToImperial.m file from cecil.

```
function [ft,in] = MetricToImperial(m)
% This function converts a metric measurement (in metres)
% to an imperial measurement (in feet and inches)
% Input:  m = measurement in metres
% Outputs:     ft = number of feet
%              in = number of inches

% Calculate the total number of cms
cm = m * 100;

% Calculate the total number of inches
totalInches = cm/2.54;

% Calculate the total number of feet, ignoring the value
% after the decimal point
% the floor command rounds down to the nearest whole number
ft = floor(totalInches/12);

% Find the remaining number of inches
in = totalInches - 12*ft;

return
```

Check that you have saved the downloaded file as MetricToImperial.m

Star *Naming functions*

You should choose functions names that give an indication of what the function does. You can use multiple words in the same variable name either by capitalizing the first letter of each word, e.g., **`MetricToImperial`**, or by putting underscores between words, e.g., **`metric_to_imperial`**.

At the rear of the course manual is a style guide which indicates the naming conventions we recommend you follow for this course. For user defined functions we recommend the convention of capitalizing the first letter of each word. This helps to distinguish them from MATLAB built-in functions which use lowercase only.

Step 5: Test the solution with a variety of data

Try running the MetricToImperial function and testing it with a number of values. In particular try using an input value of 2 metres and confirm the results match that of our hand worked example.

IMPORTANT: remember that the MetricToImperial function returns TWO values. If you want to store both values in variables, you must assign BOTH outputs to a variable, by typing the following (or similar) at the Matlab command window.

```
[feet,inches] = MetricToImperial(1)
```

Star *Testing your code*

It is VERY important that you test that your code works, using a number of values. People often make the mistake of testing their code with only one value and then assume it works because it gives the answer expected in that one case. It is easy to write code that works for one value but gives the wrong result in other cases.

Good programmers always test their code on a range of values to ensure it is working as expected.

TASK 3 *Imperial to metric function*

Use the five steps for problem solving to write a function to convert an imperial measurement (in feet and inches) to a metric measurement (in metres).

Remember to comment your function.

You should fill out the template below when completing this task.

Step 1: State the problem clearly

Give a one sentence description of the problem.

Step 2: Describe the input and output information

Either write down the inputs and outputs OR draw an I/O diagram.

Step 3: Work the problem by hand for a simple set of data

You may like to work through the problem by hand for several different values, to give you a range of values you can use to test that your function works as expected.

Step 4: Develop a solution and convert it to a computer program

Either write pseudocode OR draw flowchart below. Then write your code.

Stop ***Make sure you have written a function and not a script file***

The task asked you to write a function. Have you used the keyword function?

Stop ***Make sure your function name is valid***

Remember that your function name must match your filename exactly. Your function name cannot include spaces. It also cannot start with a number, include punctuation characters (other than the underscore) or use a reserved MATLAB command. If you use an invalid filename you will get some unusual errors when you try to run your script. Check your filename now to see if it is valid and that it matches your function name.

Step 5: Test the solution with a variety of data

For step 5 write a script file that tests your function with a variety of data, including the problem you worked by hand in step 3.

PROBLEM SOLVING

The following problem is taken from page 128, "Introduction to MATLAB 7 for Engineers", William J. Palm III):

The potential energy stored in a spring is $kx^2/2$ where k is the spring constant and x is the compression in the spring. The force required to compress the spring is kx. The following table gives the data for five springs:

Spring	1	2	3	4	5
Force (N)	11	7	8	10	9
Spring constant k (N/m)	1000	800	900	1200	700

We wish to be able to find the compression and potential energy for any given spring.

TASK 4 **Compression and potential energy for springs**

Use the five steps for problem solving to write a function that will find the compression and potential energy for any given spring.

Remember to comment your function.

You should fill out the template below when completing this task.

Step 1: State the problem clearly

Give a one sentence description of the problem.

Step 2: Describe the input and output information

Either write down the inputs and outputs OR draw an I/O diagram.

Step 3: Work the problem by hand for a simple set of data

You may like to work through the problem by hand for several different values, to give you a range of values you can use to test that your function works as expected.

Step 4: Develop a solution and convert it to a computer program

Either write pseudocode OR draw flowchart below. Then write your code.

Stop *When testing your function recall that it returns two outputs*

You will need to assign both your outputs to variables when testing your function, otherwise you will only see one output.

Step 5: Test the solution with a variety of data

For step 5 write a script file that tests your function with a variety of data, including the problem you worked by hand in step 3.

Laboratory 3: Logical Operators, Conditional Statements and Loops

RELATIONAL AND LOGICAL OPERATORS

In lectures we introduced relational operators and logical operators. These can be used in expressions to compare values. These operators are:

Relational Operators

==	equal to
~=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Logical Operators

~	not
&	and
|	or

Enter the following commands into MATLAB:

```
>> a = 1;
>> b = 2;
>> (a == 1)
>> (a == 2)
>> (b == 2)
>> (a == b)
```

Remember that 0 indicates false and any non-zero value (usually 1) indicates true. Do you get the results you expect?

Now try a few more:

```
>> (a == 1) & (b == 2)
>> (a == 2) & (b == 2)
>> (a == 2) | (b == 2)
>> a & b
>> ~a
>> c = 0;
>> ~c
>> (~b) | (~c)
```

Do these statements give the results you expect?

Stop *Output from Expressions*

If you're not sure about the output from the statements above, make sure you ask a tutor to clarify things.

CONDITIONAL STATEMENTS

Relational and logical operators can be used to test conditions in **`if`** statements. The syntax of the simplest form of an **`if`** statement is:

```
if  condition
        Do something
end;
```

The syntax of an **`if`** statement can be expanded so that if the condition is not true something else is done instead:

```
if  condition
        Do something
else
        Do something else
end;
```

The **`if`** statement can be further expanded by the inclusion of one or more **`elseif`** sections:

```
if  condition
        Do something
elseif  another condition
        Do a different thing
elseif  yet another condition
        Do a different thing
else
        Do something else
end;
```

Let's use an **`if`** statement to write out the correct value of a number. Save the following if statement in the script file **`testnum.m`**

```
if (number == 1)
     disp('The number is 1');
end;
```

Now try the following commands:

```
>> number = 1;
>> testnum
>> number = 3;
>> testnum
```

Does MATLAB do what you expect? Notice that we are only displayed a message if the number is one. Let us extend our if statement so that we get a useful message regardless of what the number is. Edit **testnum.m** to read as follows

```
if (number == 1)
     disp('The number is 1');
else
     disp('The number is not 1');
end;
```

Now try the following commands:

```
>> number = 1;
>> testnum
>> number = 3;
>> testnum
>> number = 2;
>> testnum
```

Does MATLAB do what you expect? Try stepping through **testnum.m** for each of the different values for **number**.

Finally we will extend our if statement so that it also tells us if our number is 2. Edit **testnum.m** to read as follows:

```
if (number == 1)
     disp('The number is 1');
elseif (number == 2)
     disp('The number is 2');
else
     disp('The number is not 1 or 2');
end;
```

Now try the following commands:

```
>> number = 1;
>> testnum
>> number = 3;
>> testnum
>> number = 2;
>> testnum
```

Does MATLAB do what you expect? Try stepping through **testnum.m** for each of the different values for **number**.

TIP Be careful to use == when checking for equality

To check if number was equal to 1 we used TWO equals signs. A very common programming error is to accidently use one equals sign when you wanted to check for equality. In MATLAB this will result in an error.

Star Indenting

If you properly indent your script files it makes them much easier to read and debug. If you enclose commands within a statement (conditional, loop) you should indent the commands (usually by 2-4 spaces or by using the tab). Commands at the same level should have the same indentation. Correct indentation often helps you identify if you have left **`end`** out of your conditional statements or loops.

TIP Smart Indenting

Indenting is very important to get right. MATLAB helps by giving the *Smart Indent* option in the *Text* menu (the shortcut is *ctrl-i*) . You should use this in your script files to ensure the indenting is correct.

TIP Be careful of precedence when using logical operators

To check if a number is equal to 7 or 11 you might be tempted to use:

```
number == 7 | 11
```

This will NOT do what you might expect. This is because without brackets MATLAB will first evaluate `number == 7`, giving true or false and it will then or this with the value 11. As 11 is always interpreted as true the conditional will always evaluate to true.

The following will not work either:

```
number == (7 | 11)
```

MATLAB will first evaluate `7 | 11`, (interpreting both values as true and giving a result of 1). It will then check to see if `number` is equal to 1. This conditional will only ever be true if `number` is equal to 1. If `number` was 7 it would return false.

The way to achieve what we want is to use:

```
(number == 7) | (number == 11)
```

In this case the brackets could be omitted but it is a good idea to include them to clarify what is being done and to avoid any problems with precedence.

TASK 1 Types of quadratic roots

You will be familiar with the following formula for finding the roots of a quadratic.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The term under the square root is called the discriminant and can be used to determine the type of roots the quadratic has.

If the discriminant is positive then the polynomial has two real roots;
If the discriminant is zero that the polynomial has one repeated root;
Otherwise the discriminant is negative and the polynomial has two complex roots.

Write a function that will tell a user the number of real roots a quadratic equation has. Use the template given below.

Step 1: State the problem clearly

Give a one sentence description of the problem.

Step 2: Describe the input and output information

Either write down the inputs and outputs OR draw an I/O diagram.

Step 3: Work the problem by hand for a simple set of data

You may like to work through the problem by hand for several different values, to give you a range of values you can use to test that your function works as expected.

Step 4: Develop a solution and convert it to a computer program

Draw a **flowchart** below. You may need to refer to the flowchart appendix. Then write your code.

Step 5: Test the solution with a variety of data

TASK 2 Testing of quadratic roots function

For step 5 write a script file that tests your function with a variety of data, including the problem you worked by hand in step 3. Your script file should ask the user to enter the coefficients a, b and c, call your function and then display an appropriate message based on the result.

```
>> TestQuadraticRootsFunction
Finding the number of roots for ax^2 + bx + c = 0
Enter the coefficient a:
  1
Enter the coefficient b:
  3
Enter the coefficient c:
  2
The equation has 2 real roots
```

LOOPS

While Loops

While statements repeatedly execute a piece of code while a given condition is true. The syntax of a **`while`** statement is:

```
while condition
        do something
end
```

For example, we can use MATLAB's **`isprime`** function to write out the first 10 prime numbers:

```
>> count = 0;
>> i = 1;
>> while count < 10
     if isprime(i)
       disp(i);
       count = count + 1;
     end;
     i = i + 1;
   end;

     2
     3
     5
     7
    11
    13
    17
    19
    23
    29
```

TASK 3 *Who wants to be a millionaire?*

One of the course lecturers has (hypothetically) decided to retire when they have a million dollars in the bank. Assume this lecturer is currently 30 years old, has $20,000 in the bank and can earn a guaranteed rate of return of 6% per annum on the money they have in the bank. Every year the lecturer is able to add an extra $10,000 to their savings (over and above the investment returns on the money they have already saved).

Use a while loop to predict when the lecturer can retire. What criteria should your while loop test to determine whether to exit the loop? Every year the following things happen:
- The amount of money the lecturer had saved increases by a factor of (1 + rate_of_return/100) since they accrue interest.
- The lecturer adds an additional $10,000 to their savings, i.e. after interest is added at the last step.
- The lecturer gets one year older.

When your while loop has finished your script file should write a message telling the lecturer what age they will be when they are predicted to be a millionaire, and how much money they have saved at the point. Note that the solution will only be accurate to the nearest year (i.e. age 67, as opposed to age 67.28).

Sample output:
```
>> task3
Savings goal achieved at age 62 with balance of $1,037,965.51
```

Notes:
When entering value of variables in MATLAB do not use $ signs, commas, or % signs, e.g.
```
savings = 20000
```
not
```
savings = $20,000
```

and
```
rate_of_return = 6
```
not
```
rate_of_return = 6%
```

The financial toolbox is not installed in the lab version of MATLAB so it will not print numbers as currency automatically. If you would like your output to be tidy you can use the currency_string function (provided on the class web site) to print out a numerical value as currency. For example

```
>> a = 8972354.45565;
>> disp(currency_string(a))
$8,972,354.46
```

TASK 4 An uncertain world

Life is not always as predictable as in task 3. Rate of return on investments vary and occasionally people may have extra money (e.g. from an inheritance) they want to contribute to their savings. Modify your script file from task 3 to account for this variability.

1. Allow the investment rate of return to vary between -3% and 12%. Each year MATLAB should pick a rate of return for that year based on a uniform distribution (i.e. all values between -3% and 12% are equally likely). This can be achieved defining the rate of return as:

```
rate_of_return = -3 + (12 - -3)*rand
```

Note that the `rand` command produces a random number uniformly distributed between zero and one.

2. The lecturer fortunately receives occasional $10,000 inheritances from elderly relatives. The probability of receiving an inheritance in any year is 10% (i.e. 0.1). Any money from an inheritance is added to the lecturer's savings that year. Incorporate this extra detail into your script file by checking whether a **rand**om number generated each year is less than 0.1. **If** that is true add an additional $10,000 to the amount saved that year.

Try running this script file several times. You should notice that there is now some variability in when it predicts the lecturer is able to retire.

For Loops

For statements repetitively execute a piece of code a given number of times. The syntax of a **`for`** statement is:

```
for counter = start:step:stop
        do something
end
```

This loop works by setting **`counter = start`**, *doing something*, adding **`step`** to **`counter`**, *doing something*, adding **`step`** to **`counter`**, etc until **`counter`** reaches (or *passes*) **`stop`**. Remember that if the step is not specified it is assumed to be 1.

The following commands write out the numbers from 1 to 10:

```
>> for i = 1:10
     disp(i)
   end
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
```

The following commands write out the square of the numbers from 1 to 10:

```
>> for i = 1:10
     disp(i^2)
   end
     1
     4
     9
    16
    25
    36
    49
    64
    81
   100
```

TIP *For loop counter names*

It is very common for programmers to use the letter i or j as a counter variable name. This is a good habit to get into. If, however, you are working with complex numbers it is a good idea to use a different counter name to avoid confusion with the special MATLAB variables used to represent the square root of negative 1.

Any valid variable name can be used for your array counter

If you want to write out the odd numbers between 1 and 10 (in increasing order) you would use these commands:

```
>> for i = 1:2:10
     disp(i)
   end
     1
     3
     5
     7
     9
```

Notice that even though `10` is the stop value, it is not written out. This is because the loop stops after the stop value has been passed, even if the counter variable doesn't take this value.

If you want to write out the numbers from 10 to 1 you would use the following commands:

```
>> for i = 10:-1:1,
     disp(i)
   end
    10
     9
     8
     7
     6
     5
     4
     3
     2
     1
```

Stop *For Loops*

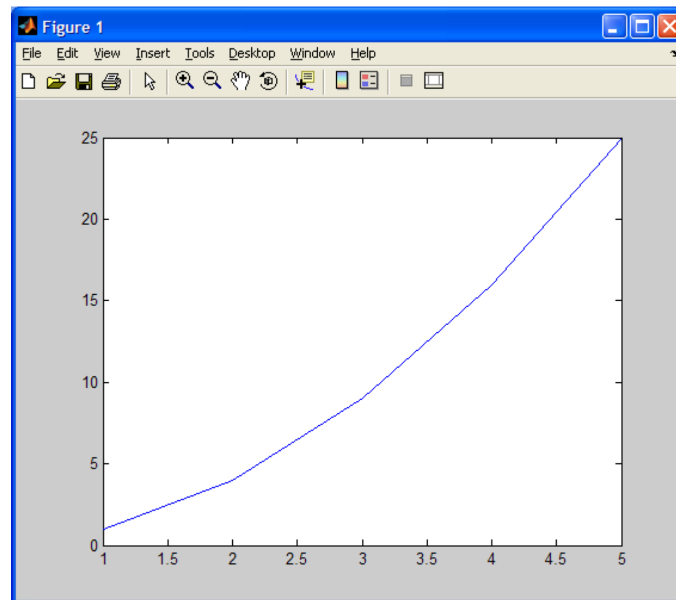
How could you write out the odd numbers between 1 and 10 in decreasing order?
If you're not sure, ask a tutor to clarify things.

For Loops and 1D Arrays

A for loop is one way to populate a 1D array (which is essentially just a list of values). 1D arrays are often used as data structures for plotting. For example, if I wanted to draw $y = x^2$ between 1 and 5, I could use the following code:

```
>> for i = 1:5,
     x(i) = i;
     y(i) = x(i)^2;
   end;
>> plot(x, y)
```

Note that **`plot(x, y)`** draws x vs y. Enter this code into MATLAB to get the following plot:



Examining **`x`** and **`y`** shows how the for loop works:

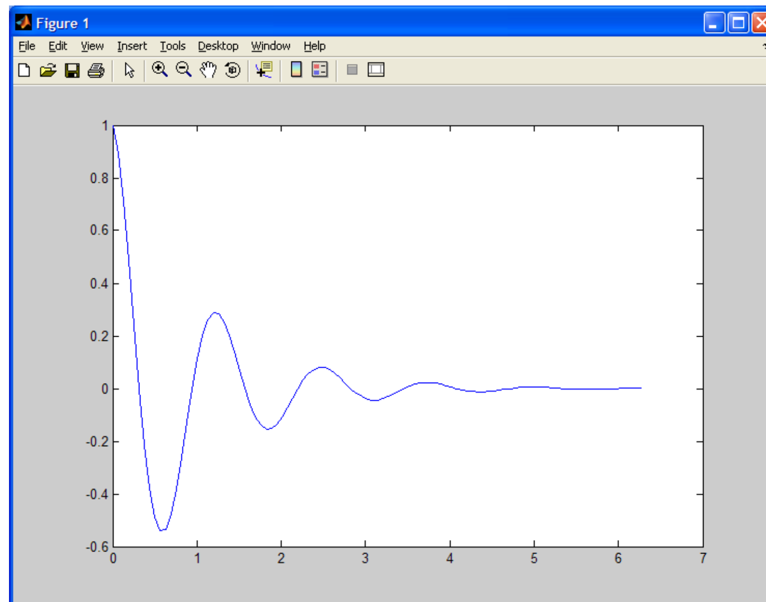
```
>> x, y
x =
     1     2     3     4     5
y =
     1     4     9    16    25
```

If I want to create a "high definition" plot of a more complex function, e.g.,

$$y = e^{-x}\cos(5x), 0 \le x \le 2\pi$$

one way to do it would be to use a for loop to move through a list of x values and calculate the corresponding y value (there are other ways which we'll see in Chapter 4).

```
>> x = linspace(0, 2 * pi, 100);
>> for i = 1:length(x),
     y(i) = exp(-x(i)) * cos(5 * x(i));
   end;
>> plot(x,y)
```



TASK 5 Monte Carlo simulation

When dealing with processes that have uncertainty involved in them (like task 4) we are often interested in the most likely outcome. One way to determine this outcome is via Monte Carlo simulation. This process runs the process a large number of times and stores the outcome (in this case the age at which the lecturer is a millionaire). The list of outcomes can be then be studied statistically.

Modify your script file from task 4 to use a **for** loop to simulate 10,000 calculations of the lecturer's retirement plans. Each time a calculation is made store the age at which the lecturer is a millionaire in a 1D array called retirement_age (which will ultimately have 10,000 entries). This modification only requires that three extra lines are added to your script file from task 4.

To find the most likely age at which the lecturer is a millionaire use the **hist** command to draw a histogram of the ages, i.e.:

```
hist(retirement_age)
```

Laboratory 4: Graphics and Image Processing

PLOTTING BASICS

Remember that all plots should include a title and labels (with units if appropriate). You can easily add a title and label your axes using the appropriate functions:

```
t = linspace(0,10,100);
plot(t,t.^2);
title('Graph of rocket path');
xlabel('time (seconds)');
ylabel('distance (metres)');
```

TIP *Extra Figure Windows*

If you want to save the figure you have created and start work on a new figure just use the **`figure`** command. This creates a new figure window for drawing on. You can choose to draw on existing figure window *Figure n* by using the command:

```
>> figure (n)
```

For more information look up **`figure`** in the MATLAB *Help* i.e.,

```
>> doc figure
```

PLOTTING MULTIPLE DATA SETS

When plotting multiple data on the same set of axes you should use different colours/line styles and include a legend.

TASK 1 *Plotting more than one set of data on the same figure*

Write a SINGLE script file that plots the following three sound waves, which are all functions of time.

$f(t) = e^{0.2t}$

$g(t) = \sin(t)$

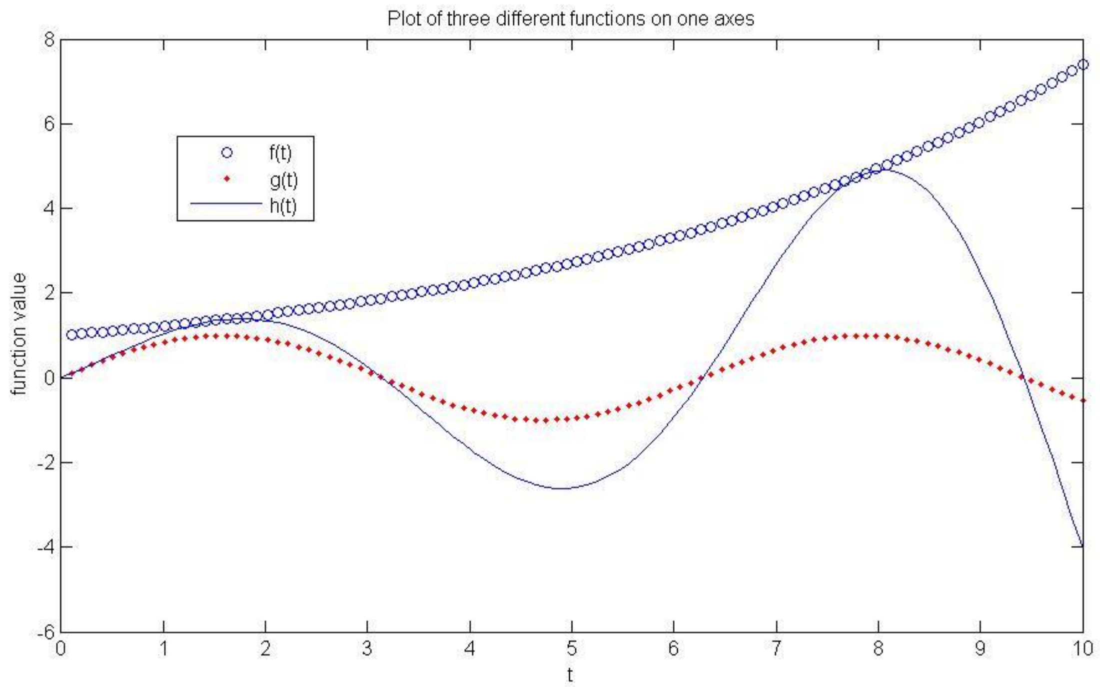
$h(t) = e^{0.2t}\sin(t)$

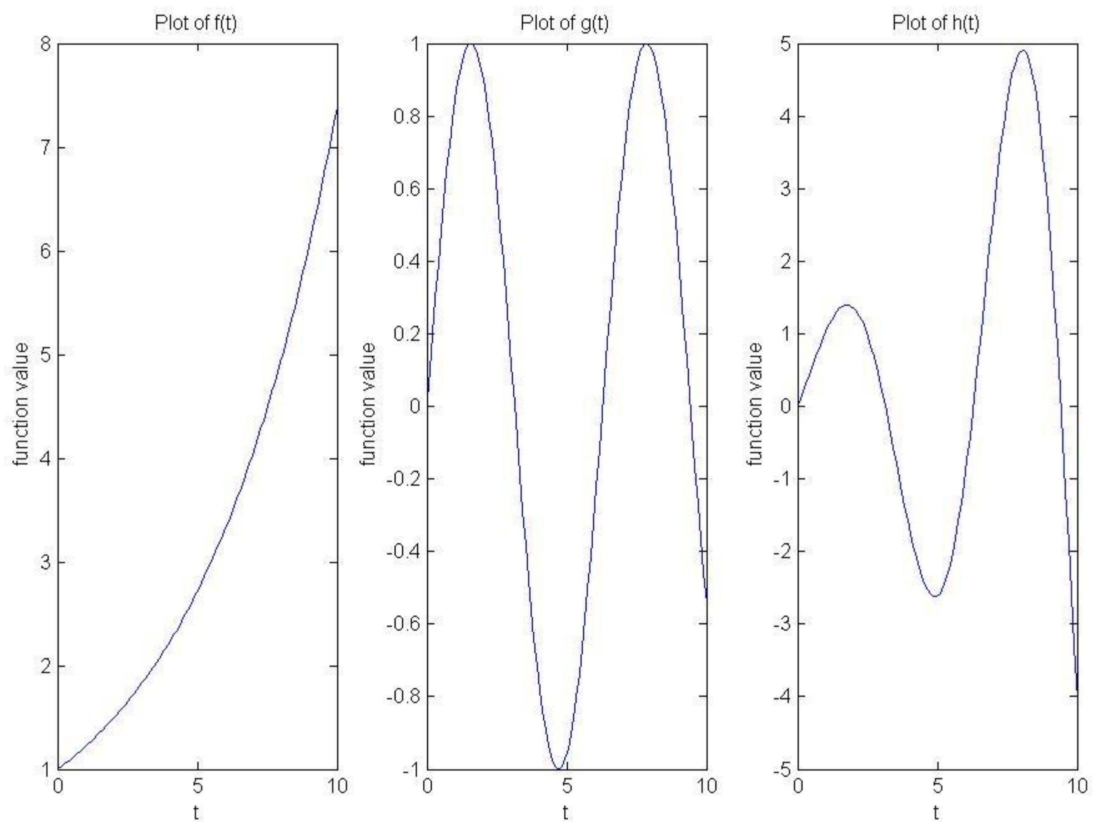
You should plot these functions for a 10 second period, over the domain $0 \le t \le 10$.

Your script should produce TWO figures, as shown overleaf.

Figure 1: A plot of all three functions on the same set of axes.

Figure 2: A plot of all three functions on the same figure but with each function on a different set of axes, side by side (you will need to use the subplot function)





Stop ***Make sure you wrote a single script file***

Remember to follow instructions. If we request a single script file, we want a single file, not two.

DRAWING SURFACES

The `meshgrid` function

Use the MATLAB *Help* to find out more about **`meshgrid`**. What is the output of the following commands?

```
>> x = [0.1 0.2 0.3];
>> y = [1.5 2.0 2.5 3.0];
>>[X, Y] = meshgrid(x, y)
```

Why is this useful for drawing surfaces?

Stop *Meshgrid*

If you're not sure how **`meshgrid`** works or why it is important make sure you ask a tutor to clarify things.

TASK 2 *Drawing Surfaces*

Write a SINGLE script file that uses two different methods to draw the surface

$$f(x, y) = x^2 y(y-1)(y+1)$$

over the domain . $-1 \leq x \leq 1$ and $-2 \leq y \leq 2$

Method 1

1. Divide the interval for each dimension into 10 points.
2. Use a nested for loop to create the surface. Draw the surface.

Method 2

Use **`meshgrid`** and the dot operator to create the surface. Draw the surface in a **separate figure**.

Make sure that your surfaces look the same. Try increasing the number of points to 100 in each direction and see what happens to your surfaces.

IMAGE PROCESSING

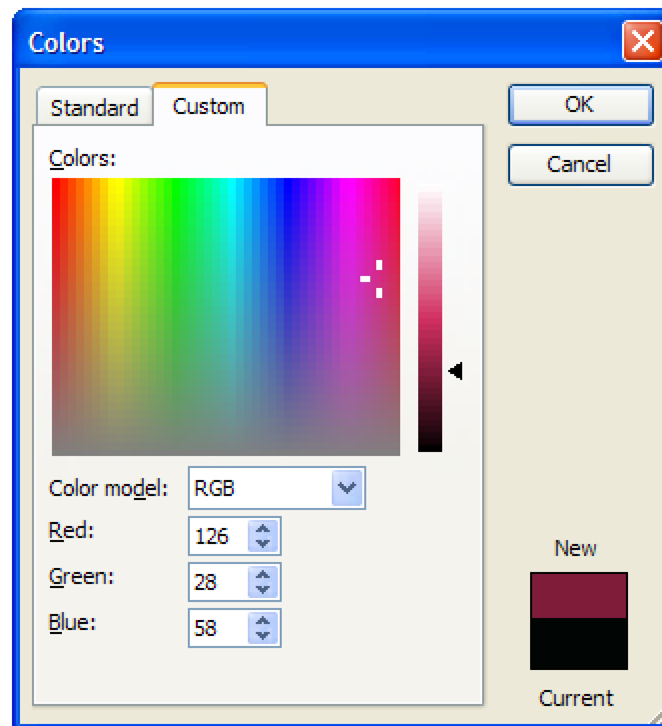
MATLAB allows you to display many different types of image files using the `imread` and `image` functions. For example, you can read the supplied `logo.jpg` file using the command

```
>> logo_RGB = imread('logo.jpg', 'JPG');
```

The variable `logo_RGB` is a 3-dimensional variable of *unsigned 8-bit integers* (integers that take values between 0 and 255). The first two dimensions represent the location of pixels in the image and the third dimension identifies whethere we are dealing with the amount or red, blue or green. MATLAB represents images using the RGB colouring for the pixels.

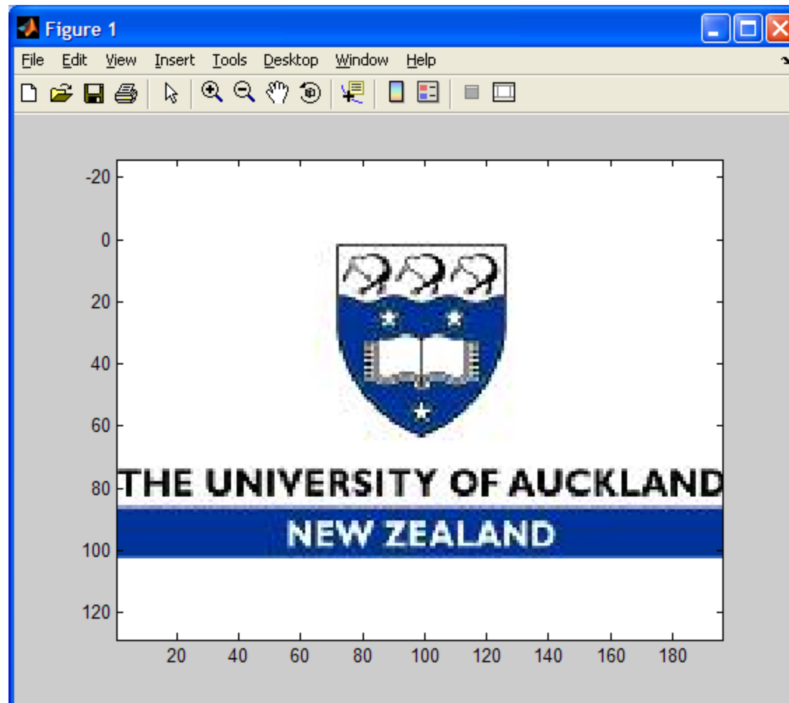
```
>> size(logo_RGB)

ans =
   102   196     3
```

Thus, pixel (i, j) has red content `logo_RGB(i, j, 1)`, green content `logo_RGB(i, j, 2)` and blue content `logo_RGB(i, j, 3)`. As an example of RGB colours, consider the following *Colors Dialog Box* for choosing colours in Microsoft Word:



To display an image represented by an $m \times n \times 3$ matrix of unsigned 8-bit integers you simply input the matrix into the `image` function.

```
>> image(logo_RGB)
>> axis equal
```



TASK 3 Write a function to test if a pixel is blue

Your friend told you that a pixel with RGB value (r, g, b) is considered to be blue if

$$r < 128 \text{ and } g < 128 \text{ and } b \geq 128$$

Write a function called PixelIsBlue that recognizes if a pixel is blue by examining its red, green and blue values.

Your function should take in three inputs (the amount of red, green and blue) and return a single output (true or false, depending on whether the pixel is blue or not).

Test your function. Your tests should include trying the following calls:

```
PixelIsBlue(10,20,200)
```

(should return 1, ie true)

```
PixelIsBlue(120,180,0)
```

(should return 0, ie false)

TASK 4 Create a Grayscale Logo

For this task you need to create a grayscale logo for your black-and-white documents. Since the logo is currently black, white and blue you only need to change the blue pixels into gray pixels.

The RGB values for gray are (153, 153, 153).

Write a script file that changes the logo image into a grayscale logo image. Make sure to read and display the logo before changing it to grayscale. You will then need to examine each pixel in turn and replace it with a gray one if it is blue. You shoud use the PixelIsBlue function that you wrote in Task 3 to determine if pixels are blue or not. Also be sure to display the final grayscale logo in a separate figure.

Your script file should not work completely? Why not? Explain why not to a tutor.

Laboratory 5: Strings and Files

STRINGS

Recall that there are a number of useful functions for comparing strings.

strcmp	Compare two strings
strncmp	Compare the first n positions of two strings
strcmpi	Compare two strings ignoring case
strfind	Search for occurrences of a shorter string in a longer string

Try typing the following and check that you understand the result of each string comparison

```
str1 = 'banana'
str2 = 'baNANA'

strcmp(str1,str2)
strcmp(str1,lower(str2))

strncmp(str1,str2,2)
strcmpi(str1,str2)

strcmp(str1,'bandana')
strncmp(str1,'bandana',3)
strncmp('banana','bandana',4)
```

Stop ***Check your understanding***

Make sure you know why the above commands evaluated to either true or false. Ask a tutor if you are unsure.

The `strfind` function is useful for searching a string for occurrences of a shorter string and returning the location(s) of that pattern. The `strfind` function returns an array listing the location(s) of that pattern. If the pattern is NOT found then the array will be empty.

To easily check if a pattern is found we can simply check if the length of the array containing the locations is 0 or not.

```
str1 = 'banana'

strfind(str1,'ana')
length(strfind(str1,'ana'))

strfind(str1,'skin')
length(strfind(str1,'skin'))
```

CELL ARRAYS

Sometimes we wish to work with a special kind of array where each element of the array is a string. These arrays have a special name in MATLAB, they are called Cell arrays. They are created and indexed with **curly braces**:

```
helloWorld = { 'hello', 'world' }
length(helloWorld)
disp( helloWorld{1} )
upper( helloWorld{2} )
```

When importing a file into MATLAB it is quite common for a cell array to be created.

TIP ***Remember to use curly braces with cell arrays***

If you use square brackets to try and create a cell array you will not create a cell array, instead you will just end up with a concatenation of the individual strings.

If you use round brackets to try and index a cell array you will get a 1x1 cell array back rather than a string. This can be very confusing if you do not spot your error.

MATLAB string functions will also work on cell arrays but the results can be rather confusing. If working with a cell array it is often a good idea to use a for loop to work through your cell array, allowing you to work on one string at a time.

Try saving the following code to a script file and running it

```
myMessage = { 'Remember', 'to', 'use', 'curly', 'braces' }
for i = 1:length(myMessage)
     str = myMessage{i};
     len = length(str);
     disp ( ['length of ' str, ' is ', num2str(len)] )
end
```

STRINGS AND USER INPUT/OUTPUT

By default the input command expects users to enter a numerical value. If you want to interpret the entered value as a string the input command needs to be passed the letter s as a second argument:

name = input('Enter your name:', 's')

The sprintf command is very useful for creating a nicely formatted string of characters which can then be displayed with the disp command. It supports a wider range of kinds of outputs.

Here is a reminder of some of the more common outputs.

specifier	output	example
s	string	hello
c	character	c
d or i	decimal integer	-23
e	scientific notation	1.2345e+10
f	decimal floating point	23.1234
g	The shorter of e or f	

The `sprintf` command takes a format string which usually includes one or more % signs followed by optional control flags and a specifier. Other arguments passed into the `sprintf` command are inserted in order in place of the specifiers, using the format specified format.

Try typing in the following commands, leaving off the semi-colon so that you can see what string is created:

```
x = 10
sprintf('The value of x is %d and x cubed is %e',x,x^3)

name = input('Enter your name:','s');
sprintf('Hello %s, how are you?',name)

sprintf('The value of pi is: %f',pi)
sprintf('Pi with scientific notation is: %e',pi)
```

Inserting a decimal and a number between the % character and the specifier allows you to specify how many decimal places to use

```
sprintf('Pi with 2dp is: %.2f',pi)
sprintf('Pi with 8dp and scientific notation is : %.8e',pi)
```

Inserting just a number between the % character and the specifier allows you to specify the minimum width to reserve for displaying the value. This is handy when wanting to format output in columns.

Try typing the following and running it:

```
for i=0:10:100
     string = sprintf('The sqrt of %3d is %7.4f',i,sqrt(i));
     disp(string);
end
```

TASK 1 Processing strings

Open the results.mat file. Examine the variables in the workspace. You will see one array called mark and one cell array called grade. Use the array editor to see what sort of values they contain.

Write a script that outputs a summary of the results for the class.
You should only loop through the marks array ONCE to do this

An example of the expected output format is below:

```
Class average: 65.2
Passing grades    Failing grades
A:      200       D:      30
B:      260       DNS:    10
C:      100
Total:  560       Total:  40
```

Note how the columns line up. You will need to use `sprintf` commands to achieve this.

You may like to test your code using the smaller fridayLabResults.mat file.

FILE I/O

MATLAB provides an import wizard which can be used to import data files. Try using the file import command to import the data contained in the results.csv file (go to File > Import Data)

Once the data is imported calculate the average mark for this class.

Sometimes the import wizard fails and we need to write code which will read in data from a file. Also if you wish to process many files at a time it is much more convenient to be able to use commands to read the contents of a file.

Instead of using the import wizard we will read in the contents of the results file using file commands. See the chapter on file I/O for a reminder on how to do this.

Remember that you must open the file for reading using fopen before reading from it. You should close your file using fclose when finished.

TASK 2 Reading from a file

Write a script that prompts the user to enter the name of a results file to read. Your script should open the file and then read the data. You can assume that the file will contain a column of real numbers (representing a mark out of 100).

The script should then calculate the following summary statistics:
The average mark for the course
The number of pass grades and the number of fail grades

TASK 3 Writing to a file

Extend your script from task 3 so that the summary statistics are written to a file.

An example of the contents of the summary file is as follows:

```
Summary statistics for file results.txt
Class average: 65.2
Total Passing: 560     Total Failing: 40
```

Laboratory 6: Linear equations and differential equations

LINEAR EQUATIONS

Systems of linear equations appear in many different branches of engineering. It is easy to solve a system of linear equations using MATLAB and only requires writing a few lines of code once the system has been written in matrix form.

Consider the problem of finding the intersection of the following two lines:

$$y = \frac{1}{2}x + 1$$
$$y = -x + 4$$

Replacing x with x_1 and y with x_2 we can rearrange these two lines to get the following system

$$-x_1 + 2x_2 = 2$$
$$x_1 + x_2 = 4$$

This system of linear equations can be written in matrix form as:

$$\begin{bmatrix} -1 & 2 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

which is of the general form: $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} = \begin{bmatrix} -1 & 2 \\ 1 & 1 \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$

In MM1 you will have learnt how to solve matrix equations of the form $\mathbf{Ax} = \mathbf{b}$.

Recall that if **A** has an inverse you can multiply both sides of the equation by the inverse of **A** to get:

$$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$$
$$\mathbf{Ix} = \mathbf{A}^{-1}\mathbf{b}$$ That is, *the solution is the inverse of A multiplied by b*.
$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

In MATLAB it is easy to find the inverse of square matrices and to perform matrix multiplication. To find the solution we can simply type the following:

```
A = [-1 2;1 1]
b = [2;4]
x = inv(A) * b
```

Check the accuracy of your solution by typing `A*x` and verifying that we get `b` (or close to it)

Once the matrix **A** and the column vector **b** have been assigned values it only took one line of code to solve the system. MATLAB also supplies the left division method, which is equivalent to Gaussian elimination and **generally gives a more accurate solution**. Try typing the following:

```
x = A\b
```

Verify that you get similar solution values.

Check the accuracy of your solution by typing `A*x` and verifying that we get `b` (or close to it)

Both of those methods ONLY work if **A** has an inverse. This can be checked before hand. Recall that a matrix has an inverse if, and only if, the determinant of the matrix is nonzero. To find the determinant of a matrix we can use the `det` command. Find the determinant of our matrix **A** by typing:

```
det(A)
```

Verify that it is nonzero. Recall that if your determinant is zero then the system of equations has no solution. This also means that the `inv` command would generate an error as the inverse does not exist.

TASK 1 Solving a system of linear equations

Find the intersection of the following three planes using MATLAB.

$$\begin{aligned} x + y + 2z &= 9 \\ 2x + 4y - 3z &= 1 \\ 3x + 6y - 5z &= 0 \end{aligned}$$

SOLVING ODES

Ordinary differential equations arise in many branches of engineering. It is relatively easy to find numerical solutions for ODEs using MATLAB if they can be written in the following form:

$$\frac{dy}{dt} = f(y,t)$$

i.e. the derivative can be written as some function of the dependent variable and independent variable. In many cases the derivative will only depend on one of the variables but MATLAB can handle functions which depend on both.

Consider the following differential equation.

$$\frac{dy}{dt} = \cos \omega t \qquad \text{with initial condition } y(0) = 0$$

This can easily be solved by direct integration to give $y = \frac{1}{\omega} \sin \omega t$

Investigate the graph of this solution for time 0 to 1 seconds and an angular frequency of 2π by using the following script (available from cecil):

```
% calculate the analytic solution of the ODE
% dy/dt = cos(omega * t);

% omega is the angular frequency
omega = 2*pi;

% our time range is from 0 to 1
t = linspace(0,1,100);

yAnalytic = 1 / omega * sin(omega * t);

plot(t,yAnalytic)
```

Now we will solve the same equation in MATLAB and compare it against the numerical solution. To calculate a numerical solution the solver needs three things:

- A formula for the derivative (in the form of a function)
- A time interval (start time and finish time stored in an array)
- An initial value at the start time (initial condition)

First we need to write a MATLAB function that calculates the derivative for any given values of *t* and *y*. This function will then be called many times by our solver to find our numerical solution. We can choose any valid function name for our derivative but as always it is a good idea to give the function a meaningful name.

Note that the solvers require the derivative function to take **both** the independent and dependent variables as inputs (even if one or the other may not be used). The independent variable must be the

first input and the dependent the second. The output for the function must be the derivative for the given inputs.

We can now write this function as follows:

```
function [dydt] = SinusoidDerivative(t,y)
% calculate the derivative dy/dt for the equation
% dy/dt = cos(omega * t)
% inputs: t, the independent variable representing time
%         y, the dependent variable representing displacement
% output: dydt, the derivative of y with respect to t

% omega is the angular frequency
omega = 2 * pi;
dydt = cos(omega * t);

return
```

Download this function from cecil and save it as SinusoidDerivative.m.

Try testing the function with a few values of y and t and verify that it gives you the correct derivative value for the following inputs:

$t = 0, \quad y = 0$

$t = 0.25, \quad y = 1$

$t = 0.5, \quad y = 0$

$t = 1, \quad y = 1$

Now we are ready to write a script file to solve our ODE:

```
% calculate the numerical solution of the ODE
% dydt = cos(omega * t);

% set up an array containing the start time and finish time
% we will calculate solution values for this time range
% we only need to specify TWO values (the start and finish)
timeInterval = [0 1];

% our initial condition is y(0)=0
yinit = 0;

% solve our ODE, the solver expects three arguments in the
% the following order
% - the name of the derivative function (in quotes)
% - the time interval to solve for (a two element row vector)
% - the value at the start time
[t,y] = ode45('SinusoidDerivative', timeInterval, yinit);
plot(t,y)
```

Donwload this file from cecil and run it. Compare your plot with the analytical solution.

A ROCKET-PROPELLED SLED

(Adapted from an example on page 535 of "Introduction to MATLAB 7 for Engineers", William J. Palm III.)

Newton's law gives the equation of motion for a rocket-propelled sled as:

$$m\frac{dv}{dt} = -cv$$

where m is the sled mass (in kg) and c is the air resistance coefficient (N s/m). We know the initial velocity $v(0)$ of the sled and want to find $v(t)$.

Finding $v(t)$ Analytically

Using our MM1 knowledge we know we can solve this ODE using *Separation of Variables*.

Solve $m\frac{dv}{dt} = -cv$ given $v(0)$ is known.

Step 0 : Separate the variables to different sides

$$m\frac{dv}{v} = -c\ dt$$

Step 1: Integrate

$$\int m\frac{dv}{v} = \int -c\ dt$$

$$m\ln v = -ct + k$$

Note that k is a constant.

Step 2: Make v the subject

$$m\ln v = -ct + k$$

$$\ln v = \frac{-ct + k}{m}$$

$$v = e^{\frac{-ct+k}{m}}$$

Step 3: Rework the constant

$$v = Ae^{\frac{-c}{m}t}$$

$$\text{where } A = e^{\frac{k}{m}}$$

Step 4: Evaluate the constant

$$v(0) = Ae^{\frac{-c}{m}0}$$

$$A = v(0)$$

Step 5: Answer the question

$$v(t) = v(0)e^{\frac{-c}{m}t}$$

TASK 2 Plot the Analytic Solution

Given a mass of 1000kg, an initial velocity of 5 m/s and air resistance coefficient of 500 N s/m, plot the rocket-propelled sled's velocity over the time interval $0 \le t \le 10$.

TASK 3 Plot the Numerical Solution

Now we want to solve the ODE numerically.

Write a function that returns the derivative $\frac{dv}{dt}$.
Write a script file that uses **`ode45`** to numerically solve the ODE and then produces a plot of velocity vs time.

Use the MATLAB *Help* if needed.