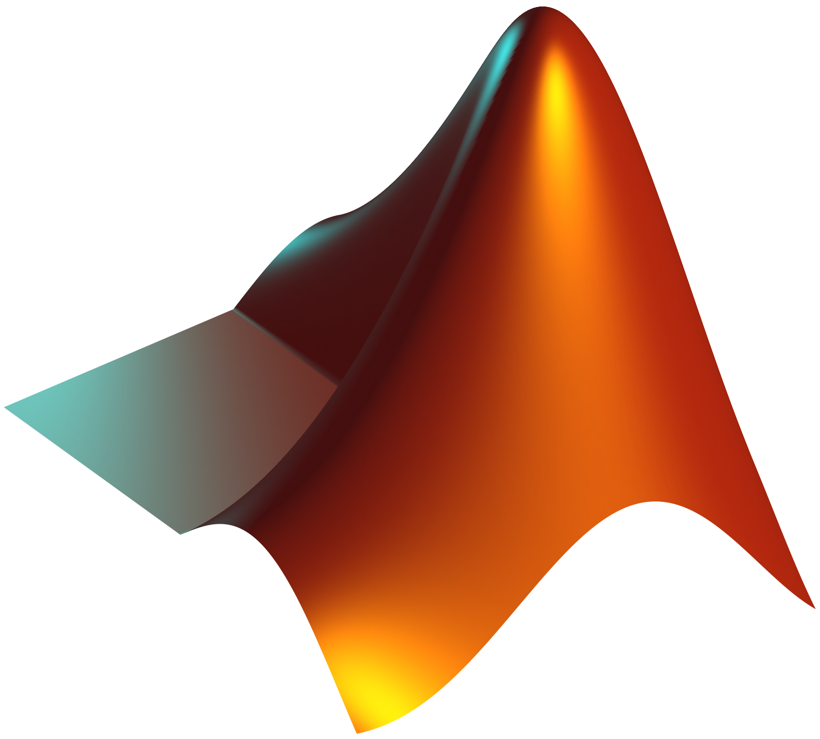
ENGGEN 131

Introduction to Engineering Computation and Software Development

LAB 0

An Introduction to MATLAB



The Department of Engineering Science
The University of Auckland

Getting Started

If the lab computer is switched off, you should press the power button ⏻ the computer's front panel. Similarly, if the monitor is switched off, press the power button ⏻ on the monitor's front panel.

If the computer's power light is already on, you can log in by holding down the keys Ctrl + Alt + Del. If someone has already logged into the computer, log off the current account and sign in with your account.

If you are unfamiliar with the login process for the university computers, your teaching assistant can show you.

Most of the applications you will use on campus reside on file servers rather than on the computer's local hard drive. This file storage approach means that you can access your files on any computer found around the university. The university has allocated you an electronic campus home directory H: which you can use. The directory H: is where you can save your work on the university network; for example, you could create a folder in H: 〉 131_Lab_1.

Stop: Check your Save Location!

Do not save your documents in the directory C: of the university computers, as this location is not a part of the university network.

Note that the folders Desktop, Documents and Downloads located in the directory H: are okay but not recommended.

Tip: Make a Backup

Keeping a file backup with a physical USB drive or in the cloud with One Drive, (the university provides free) is highly recommended. Alternatively, you can visit: `https://homeweb.auckland.ac.nz` to access your H: from home. For more about the university's file saving and sharing, visit the library website.

If you want to use your personal computer, you will need the basic MATLAB installation. MATLAB is supported on Unix, Mac, and Windows environments. A student version of MATLAB for Mac or Windows is free with your university email, which you can install on your personal computer. Alternatively, an online version, MATLAB Online can also be used.

Introduction to MATLAB

MATLAB (**MAT**rix **LAB**oratory) is an interactive software system for numerical computations and graphics. MATLAB (as the name suggests) is specially designed for matrix computations such as:

- Solving systems of linear equations.
- Performing matrix transformations.
- Factoring matrices and so forth.

In addition, MATLAB has a variety of graphical capabilities and functionality extended through programs written in its own programming language.

Advantages:

✔ MATLAB simplifies the analysis of mathematical models.
✔ MATLAB frees you from coding in lower-level languages (saves a lot of time - with some computational speed penalties).
✔ MATLAB provides an extensible programming/visualisation environment.
✔ MATLAB provides professional-looking graphs.

Disadvantages:

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

The MATLAB Integrated Development Environment

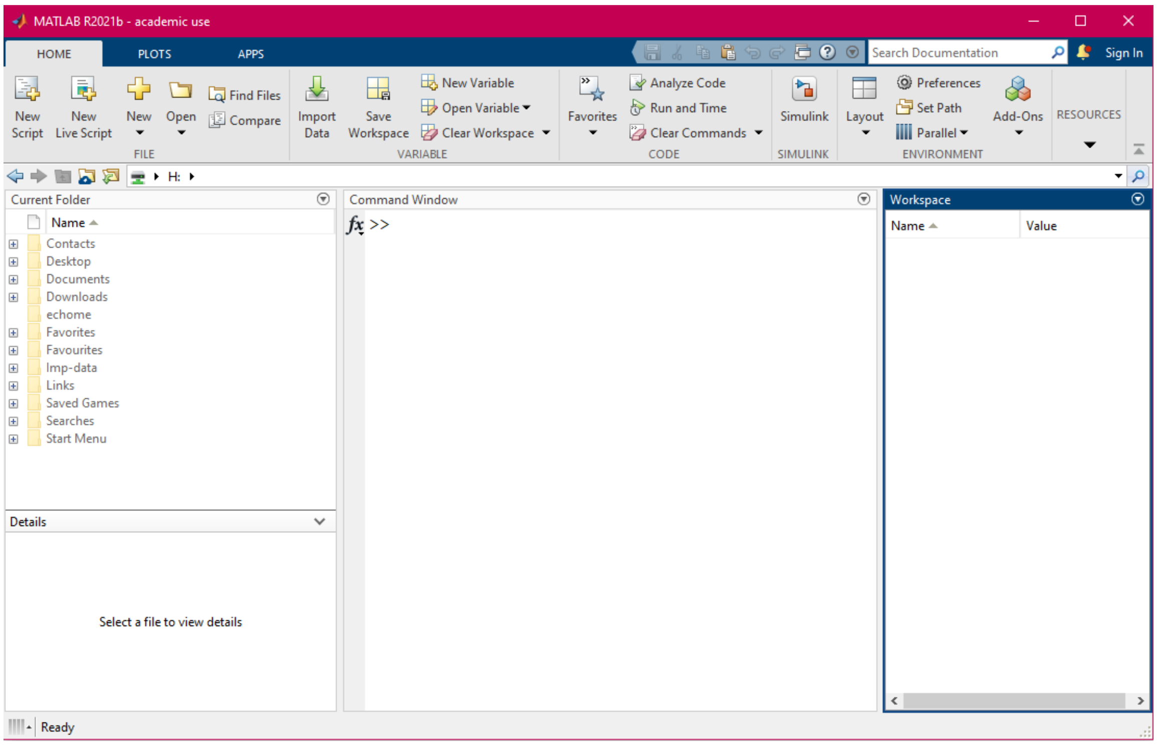
Find MATLAB's Integrated Development Environment (MATLAB IDE). It will be called "MATLAB" + "R" + year + ('a' or 'b'), for example, "MATLAB R2021a". MATLAB may already be on the desktop or toolbar. Otherwise you may have to search for it in the search bar.

Double click on the MATLAB icon to open the program. The MATLAB IDE should appear on your screen and look something like Figure 0.1.

The main window of Figure 0.1 is called the *Command Window.* It is the window in which you can interact directly with MATLAB. Once the MATLAB prompt >> is displayed, the user can execute all MATLAB commands from this *Command Line.* In the command window shown in Figure 0.2, we execute the command:

```
>> a = 7
```

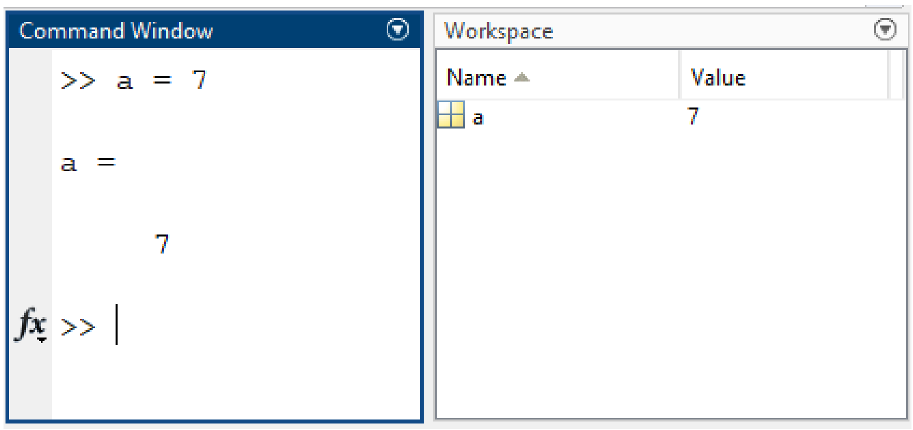
The *Workspace* window is in the right of Figure 0.1, or more closely in Figure 0.2. Note that the workspace window is empty when you first start up MATLAB. However, as you execute commands in the Command Window, the Workspace window will show all variables you create in your current MATLAB session. In this example, the workspace contains the



```
>>
```

Figure 0.1: Default MATLAB IDE View.

variable `a`.



```
>> a = 7

a =

     7

>> 
```

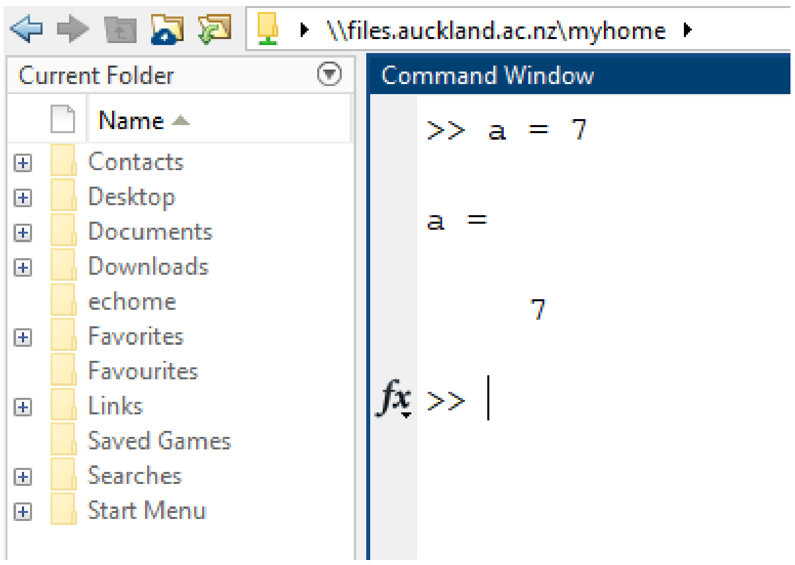
Name	Value
a	7

Figure 0.2: MATLAB Command and Workspace Windows.

If you enable the *Command History* window through Home › Layout › Command History › Docked, it will be visible in the bottom right of Figure 0.1. This window gives a chronological list of all MATLAB commands that you used.

The *Current Folder* window is in the top left corner of Figure 0.1, or more closely in Figure 0.3. This window shows the current folder you are currently working in. File paths will generally be relative to this location.

You will create files to store programs or workspaces during the MATLAB course. Therefore, you must create an appropriate folder to store the lab files.



```
>> a = 7

a =

     7

>>
```

Figure 0.3: MATLAB Current Folder Window.

You can the change directory to be your H: directory (or any other directory) by clicking on the button next to the *Current Directory Path*.

You will know the various buttons and menus on the MATLAB IDE during the course. We will not discuss all of them here. Therefore, play around and familiarise yourself!

Tip: Managing MATLAB Windows

Feel free to adjust the locations and sizes of the MATLAB windows to suit your computer screen or preference by clicking and dragging the window headers and borders.

If you want to reset the view to the **default**, you can find the option in the main application window: Home › Layout › Default.

If you want to change the font of the text, you can find the option in the preferences: Home › Preferences › MATLAB › Fonts

Working at the Command Line

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

```
>> 5 + 4
ans =
         9

>> 5 - 4
```

```
ans =
      1

>> 5 * 4
ans =
      20

>> 5 / 4
ans =
      1.2500

>> 5 ^ 4
ans =
      625

>> 34 ^ 16
ans =
      3.1897e+024
```

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

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

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.

MATLAB Variables

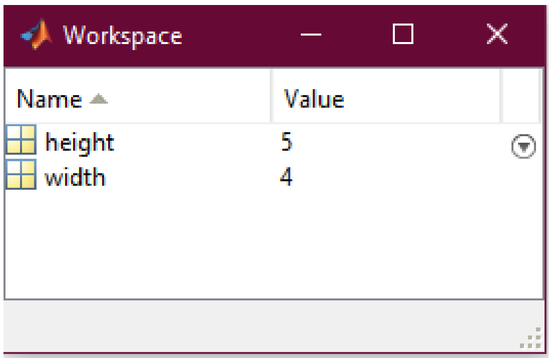
As we discussed, MATLAB stands for **MAT**rix **LAB**oratory. This title is appropriate because the structure for storing all data in MATLAB is a matrix. We will talk more about this in upcoming lectures. Right now, we will only work with scalar variables. MATLAB stores these as matrix variables with 1 row and 1 column as 1 x 1 matrices.

Assigning Variables

To create a (scalar) variable in MATLAB, enter a valid variable name at the command line and assign it a value using =. Once a variable is assigned a value, it is added to the MATLAB Workspace and 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 like in Figure 0.4.



Name	Value
height	5
width	4

Figure 0.4: Example MATLAB Workspace

We should **not** use length as a variable name as this is a built-in MATLAB function. If we do (accidentally) use length (or another built-in function name) as a variable name, we must remove it from the workspace to restore the built-in function.

Tip: Rules on Variable Names

There are some rules as to the variable names you can use:

1. Must start with a letter, followed by letters, digits, or underscores. `X12`, `rate_const` and `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`, and `FlOw` 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, for example, `width`, `cableLength` and `water_flow`. You can use multiple words in the same variable name either by capitalising the first letter of each word after the first one, like `cableLength`, or by putting underscores between words like `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 various conventions.

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, for example:

```
>> 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 workspace, you can use them within calculations. For example, to calculate the area of a 5 x 4 rectangle, you could enter:

```
>> height = 5
height =
       5
>> width = 4
width =
       4
>> area = height * width
area =
       20
```

Error Messages

If your command is invalid, MATLAB gives you explanatory error messages (luckily). Please 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:  Incorrect use of '='operator.  To assign a value
to a variable, use '='.  To compare values for equality,
use '=='.
```

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

Stop: Incorrect use of '='Operator

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

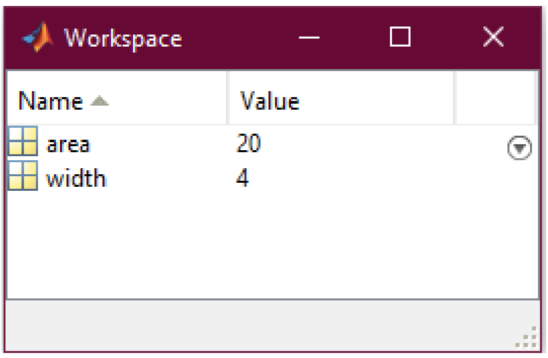
Finding Old Commands

Suppose you want to repeat or edit an earlier command. If you dislike typing it all back in, then there is good news: MATLAB lets you search through your previous commands using the keys ↑ and ↓. This feature is very convenient and can save you considerable time.

Example: clear the height of the rectangle in the earlier example:

```
>> clear height
```

and now look at the Workspace window in Figure 0.5.



Workspace

Name	Value
area	20
width	4

Figure 0.5: Updated Example MATLAB Workspace.

Now you want it back! Make sure the text cursor is in the Command Window, then press the key ↑ until the earlier command:

```
>> height = 5
```

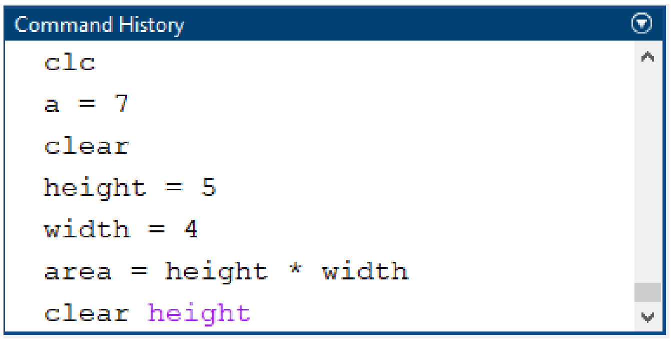
appears in the Command Window (the key ↓ can be used if you go back too far). Now press hit the return key ↲. Now `height` is back to what it was initially.

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 ↑ key. MATLAB will display the most recent command starting with `h`. Alternatively, you can Copy and Paste.

You can also use the Command History window to jump to a previous command. Go to the Command History window, similar to Figure 0.6 and **double-click** on the line `height = 5`.



```
clc
a = 7
clear
height = 5
width = 4
area = height * width
clear height
```

Figure 0.6: Example MATLAB Command History Window.

The command:

```
>> height = 5
```

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

Suppression of Output

Until now, you have always seen the results of each command on the screen. This output is often not required or 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 does not make much difference - but it can make a great deal of difference when dealing with large lists of numbers.

Saving your work - Script Files

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

To create a new script file (also called an m-file), choose Home > New Script or Home > New > Script in MATLAB. 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 x width
```

Now save your script file by choosing Editor > Save, Editor > Save As or the keys Ctrl + S . Give your script file a name, such as `area_calc.m`. The file extension .m denotes MATLAB files. MATLAB will save the file in the working directory you set earlier in the lab.

Stop: Invalid Filenames

To avoid invalid filenames (and the associated errors), make sure filenames:

- Do not include spaces.
- Do not start with a number or include punctuation characters (other than the underscore).
- Avoid using reserved MATLAB commands as filenames.
  - If you are unsure whether a filename is a reserved MATLAB command, try typing the word help followed by the filename you wish to use. If it is reserved, you'll see an explanation of what that command or function does.

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 the keys Ctrl + 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 simultaneously.

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.

Errors in Script Files

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

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
Invalid expression.  Check for missing multiplication
operator, missing or unbalanced delimiters, or other
syntax error.  To construct matrices, use brackets
instead of parentheses.
```

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

Stop: Invalid Expression

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

Commenting

One of the most important facets of programming is documenting your program (i.e., 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 Documentation.

Tip: Comment Keyboard Shortcuts

The Editor menu bar has buttons to comment a line or multiple lines. As well as buttons to uncomment.

Select the line(s) and press the buttons.

Alternatively, the keyboard board shortcuts can also be used.

Star: Good Commenting

Your initial commenting block should briefly describe the script file functionality and clearly define the input and output for the script file. You should also indicate the author of the script. For example, here are the contents of the file `area_calc.m`:

```
% This script calculates the area of a rectangle
% of height 5m and width 4m
% Author: Bilbo Baggins

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

The initial commenting block becomes part of the help system, so you can find information about a script by typing the word help followed by the script name:

```
>> help area_calc
This script calculates the area of a rectangle
of height 5m and width 4m
Author:  Bilbo Baggains
```

You should also have (at least) one line of commenting for any command in your script file that is non-trivial. The above example is so simple it probably does not merit further commenting, but as you progress to harder tasks, you will want to add comments to the body of your function. For example:

```
% This script calculates the area of a rectangle
% of height 5m and width 4m
% Author: Bilbo Baggins

height = 5
width  = 4

% A comment could go here, on a line of its own
area = height * width % Or at the end of the line
```

✍ Task 0.1: Creating a Simple Script

Write a script called `FivePlusOne.m` that adds 1 to a value of 5. Your script should assign the number 5 to variable `x`, with variable `y` being the addition of `1` to `x`.

Submit the completed script `FivePlusOne.m` to MATLAB *Grader*. Use this task to familiarise yourself with MATLAB *Grader*.

✍ Task 0.2: Creating a Simple Function

Create a simple function in MATLAB called `PlusTwo` that pluses two to the input and returns the result.

Input:

A number `x`

Output:

A number that has two added to it

For example:
Calling `PlusTwo(3)` will return a value of `5`.
Calling `PlusTwo(10)` will return an array containing the values `12`.

Note: you can create a function (from a template) in MATLAB using Home › New › Function or Editor › New › Function.

Use the command line to test whether your function is working as intended.

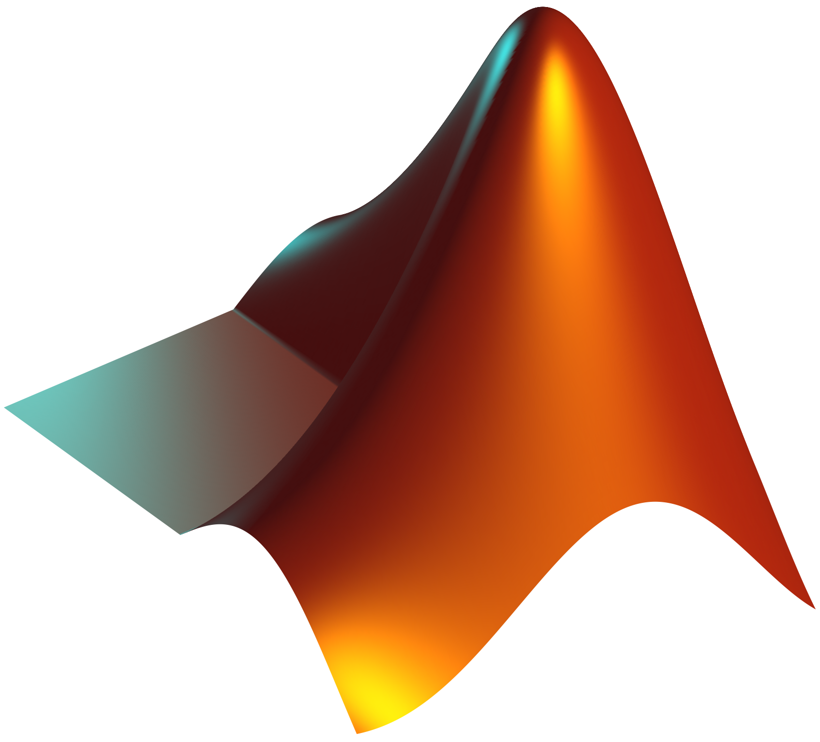
Submit the completed function `PlusTwo` to MATLAB *Grader*. Use this task to familiarise yourself with MATLAB *Grader*.

ENGGEN 131

Introduction to Engineering Computation and Software Development

LAB 1

Console, Scripts and Functions



The Department of Engineering Science

The University of Auckland

General Lab Directions

It would be best if you worked your way through the lab manual, **reading everything**. Interaction with the computer takes place throughout the lab. The sections and lab tasks are not stand-alone, so you must work through them in order. All lab tasks need to be **submitted online**, but you can get **feedback** from the teaching assistants when attending your lab in person.

Any MATLAB commands for you to type appear as they would in MATLAB. Other MATLAB terms in a body of text will be in code font to make it easier to see. Please feel free to help your fellow students understand the concepts (but you should do your own tasks) and never hesitate to ask questions of the teaching assistants.

Some labs include optional study exercises. While completing these optional study tasks is not required to get marks for the lab, you should try them when time permits. The material presented in the optional study exercises will increase your understanding of MATLAB. We **highly** recommend them to help you prepare for the final exam, so give these exercises a go when you have a chance.

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. Following these tried and true conventions will make you a programming star!
🛑	Stop	There is a "roadblock to understanding". Make sure you ask a teaching assistant for clarification if you don't understand what MATLAB is doing.
🕮	Study	There are extra exercises that will increase your understanding of MATLAB, these are not assessed tasks. But, they will prepare you for the tests and exams.

Class Forum

We are using Piazza as a question-and-answer forum for the class. This forum is the best place for you to ask questions and discuss the course content. You will have been sent an email with a link to use to activate your Piazza account.

Activate your Piazza account if you have not already done so.

Go to the class forum, either by using the link on Canvas or by going to:
`https://piazza.com`.

The MATLAB Help System

MATLAB's ***Help System*** provides information to assist you while using MATLAB. Going to Home » Help » Documentation will open the Help System. You can then browse the commands via Contents » Category » Matlab » ..., look through the index of commands via Contents » Category » Matlab » Functions or Search for keywords or phrases within the documentation. This place has the most comprehensive documentation for MATLAB and is the best place to find out what you need (the most common search engine results will include this documentation). You can also open the MATLAB Help System using the `helpwin` command.

When working at the command line, a faster way to find information about commands is via the `help` command.

If you type:

```
>> help
```

and press the return key ↵, the console will present a text menu of help options. If you type `help` followed by the name of a command, MATLAB will provide help for that particular command. For example:

```
>> help sin
```

will call up information on the `sin` command, which calculates the sine of an angle in radians.

Note that the `help` command only displays a summary of how to use the command you are interested in. For more detailed help documentation, use the `doc` command. For example:

```
>> doc sin
```

will display the complete documentation of the `sin` command, including examples of how to use it and sample output.

Another handy MATLAB command is the `lookfor` command, which will search for commands related to a keyword. Find out which MATLAB commands are available for plotting using the command `lookfor log` in the command window. It may take some time to get all commands on the screen (and the most useful ones are likely to be first on the list). You can stop the search anytime by pressing the keys Ctrl+C in the Command Window

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

Tip: Quick Help

To quickly open documentation, press the key `F1`. To quickly open documentation for a specific command, highlight or put your text cursor on the command and press the key `F1`.

Alternatively, open a search engine in your browser and search "MATLAB" + command, for example, "MATLAB sin". More often than not, the first result is from the MATLAB documentation.

Task 1.1: 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 supplied script `MatlabExpressions.m` to calculate the values of the given expressions. When the lab is complete, upload this script to MATLAB *Grader* for your work to be marked.

Use the MATLAB Help system to look up functions you do not know, for example:

```
>> lookfor log, or
>> help log
```

Try out any functions you have found by using them at the command line before adding an appropriate line to the script file.

Remember that $\ln(x)$ is the natural logarithm and that the function e^x is called the exponential function. We will use the standard MATLAB trig functions that assume inputs in **radians** (rather than degrees).

Note that the last expression produces a complex number (which MATLAB represents using the special variable `i` for the imaginary part).

Variable Name	Expression to assign to variable
`surd`	$\frac{1}{\sqrt{2\pi}}$
`power`	$5 \times 10^9 + 3 \times 10^8$
`logbase10of72`	$\log_{10}(72)$
`cosinePi`	$\cos(\pi)$
`eulerValue`	e^1
`complex`	$\ln\left(\sin(\pi^2)\right)$

Submit the completed script `MatlabExpressions.m` to MATLAB *Grader*.

Electrical Circuit Theory

(Required background for Task: 1.2 and Task: 1.3)

If we place two resistors R_1 and R_2 (Ω) in series in a circuit, they can be represented as a single resistor R (Ω), like in Figure 1.1.

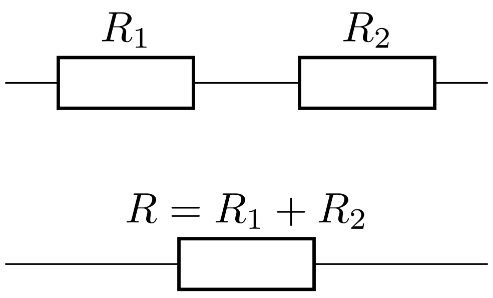


Figure 1.1: Resistors in Series.

If we place two resistors R_1 and R_2 (Ω) in parallel in a circuit, they can be represented as a single resistor R (Ω), like in Figure 1.2.

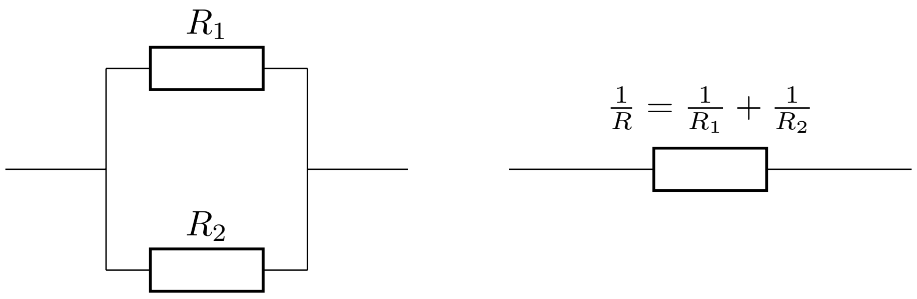


Figure 1.2: Resistors in Parallel.

Try using the Command Line to calculate the resistance of the configuration of resistors in Figure 1.3 (you might also need to do some work with pen and paper).

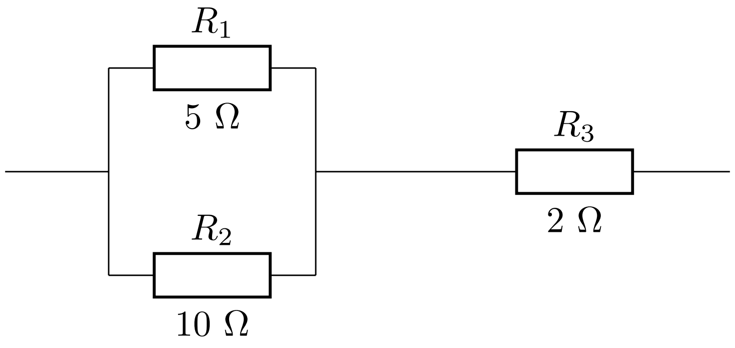


Figure 1.3: Connected Resistors.

Now consider the configuration of resistors in the closed-circuit in Figure 1.4

Through Ohm's Law ($I = V/R$): The current through the circuit (amps) is determined by the voltage (volts) through the circuit divided by the resistance (ohms) through the circuit.

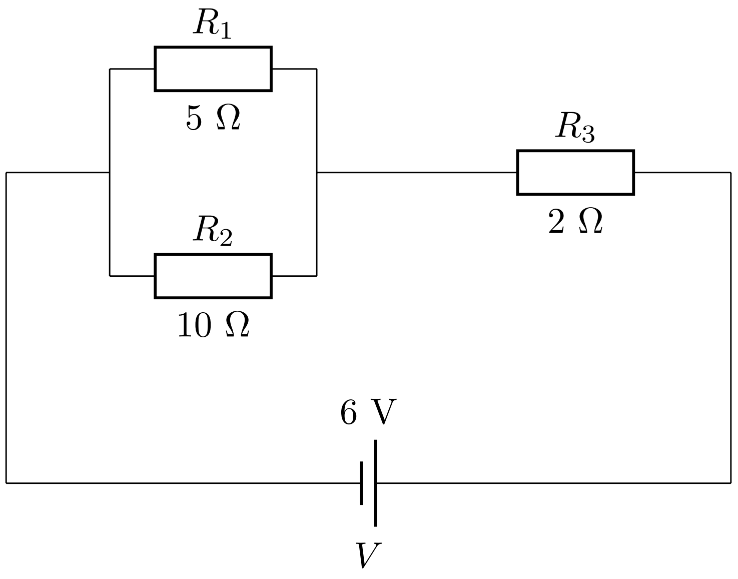


Figure 1.4: Resistors in a Closed Circuit.

Now use the Command Line to calculate the current through the circuit above.

✍ Task 1.2: Basic Scripts

We wish to write a script file to calculate the total resistance for the configuration of resistors and the current through the circuit in Figure 1.4.

However, when the values of the individual resistors or voltage change, the current will also change.

For a more general case, consider the circuit in Figure 1.5.

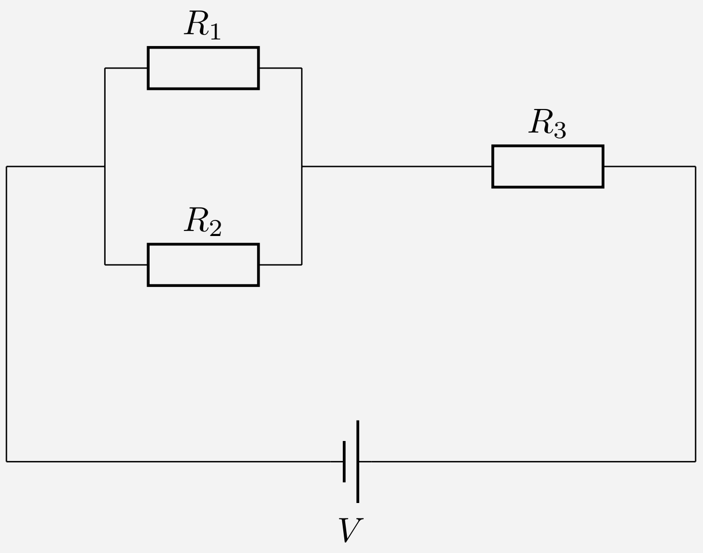


Figure 1.5: A General a Closed Circuit.

Write a script called `CalcCircuitCurrent.m` to calculate the total resistance for the first configuration of resistors and the current I through the circuit. For the resistors R_1, R_2, R_3 and V, create MATLAB variables named `R1`, `R2`, `R3` and `V` to hold variable resistances and voltage values. Use the variables you have created in your calculations to obtain the resistance for the configuration of resistors shown in Figure 1.4 and store it in a variable `RTotal`. Then calculate the total current through the circuit and store it in a variable `I`. Finally, display both the value of `RTotal` and `I`.

Make sure you have commented your script file.

Submit the completed script `CalcCircuitCurrent.m` to MATLAB *Grader*.

Stop: Check your Variable Names Carefully!

Did you use the exact variable names given in the task? If you did not, you would lose marks when you upload this script to MATLAB *Grader*, as it will check that the values of those variables exist and are correct.

Stop: Commenting is Vital!

Did you remember to comment your script file? Do you have a header comment? Do they describe what your file does **and** include the author (i.e., your name/UPI)? Do you have comments describing any non-trivial commands or functions?

Basic User Interaction

Basic User Input

Script files are more useful if they can interact with users. One straightforward 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 script file `area_calc.m` 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.m` 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:

```
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 okay 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 will learn more about why this works later in the course.

Tip: Get help when you need it!

If you do not understand something, try reading through your notes or the recommended text. If you still do not understand, then get some help. There are many possible sources of help:

During the lab, you can:

- Ask a teaching assistant.
- Ask a friend.
- Post a question on the forum.

Outside of lab times, you can:

- Ask a friend.
- Post a question on the forum.
- Visit the lecturers during their office hours.
- Pop into the Part 1 Assistance Centre.

Please **do not** leave it to the last minute to get help on assignments and projects. One very good reason to start assignments and projects early is that if you run into problems, you will have much more time to seek help.

✍ Task 1.3: More Interactive Scripts

In Task: 1.2, to calculate the circuit's total resistance and current, the individual resistor R_1, R_2, R_3 and voltage V need to be known and written into the script beforehand. We would need to edit the script if we wanted to use different resistances. Instead of changing the value for the resistors R_1, R_2, R_3 or voltage V before running the script, we can ask the user to enter these values during the execution of the script by using the `input` command.

You might have noticed that the calculation of the total resistance is a little complicated. We can separate this calculation into reusable code by creating a function called `TotalCircuitResistance`. This function will take as inputs the three resistances and then calculate the total resistance for that part of the circuit. Moving code into a function makes it much easier to reuse and allows us to write more readable and easier-to-understand code. It does this by simplifying the problem into smaller, more manageable pieces. You will be writing a lot of functions in this course.

We have written this particular function for you, and it is a part of the supplied files. To call this function using the script, we can use a line like:

```
    RTotal = TotalCircuitResistance(R1, R2, R3);
```

(Note for this to work, you will need the function saved to the **same directory** as any code that calls it.)

Write a script called `CalcCircuitCurrentInteractive.m` that will ask for a resistor value for each resistor variable R_1, R_2, R_3 and voltage value for voltage V using the function `input`. Then, calculate the total resistance (using the supplied function `TotalCircuitResistance`) and current. Finally, display both the value of total resistance and current.

Make sure you have commented your script file.

Submit the completed script `CalcCircuitCurrentInteractive.m` to *Canvas*.

✍ Task 1.4: Command Line, Scripts And Functions

The questions below highlight some benefits and differences between coding using the MATLAB command line, script files and functions. Think about the **best** choice between command line, script file and function for the following questions. Note that in each scenario, there are many good choices. However, *Canvas* does not fully support answers like that, so choose the single **best** answer. Then, submit your answers to the *Canvas quiz*.

1. What should you use if you need to do "quick and dirty" calculations (i.e., where you will only need to perform the calculation once)?

2. What should you use if you need to do a difficult calculation that requires multiple steps that all depend on each other?

3. What should you use if you need to perform a sequence of calculations that will be repeated many times but with slightly different values going into the calculation?

4. What should you use if you need to test one or two lines of code?

5. What should you use if you need to test many lines of code?

Study: Exploring MATLAB Capability

You may wish to explore some of the other capabilities of MATLAB that we will not cover in class. This material is not examinable.

You can see more examples of MATLAB's capabilities by typing `help demos` at the Command Line. You can run any of them by entering the name of the demo at the command line. Some of the demos we recommend looking at are `truss`, `travel`, `ballode`, `graf2d`.

If you are looking for something less serious like Figure 1.6, see `xpbombs`, this is an implementation of Minesweeper, which for many years was one of the only games that came as standard with the Windows operating system.

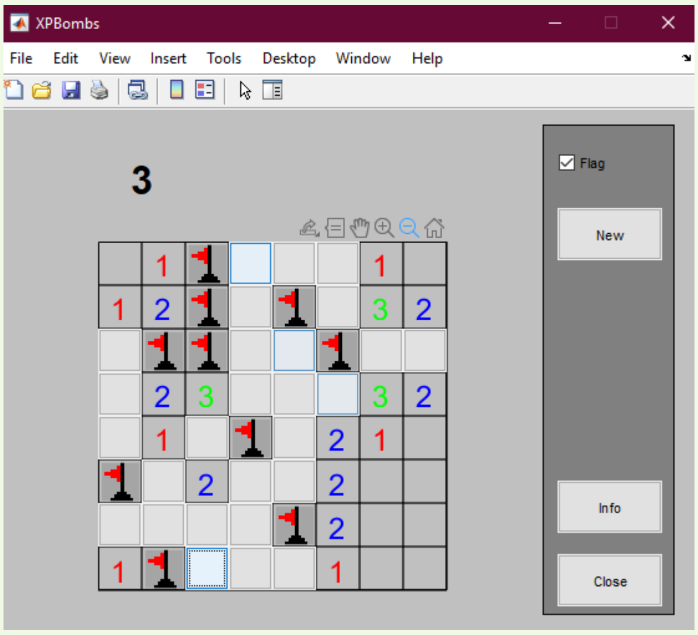


Figure 1.6: xpbombs, a MATLAB implementation of minesweeper.

✍ Task Submission Summary

☐ Task: 1.1 **Using MATLAB Help**. [2 Marks]

- Modify the supplied script `MatlabExpressions.m`.
- Submit to MATLAB *Grader*.

☐ Task: 1.2 **Basic Scripts**. [2 Marks]

- Create and complete the script `CalcCircuitCurrent.m`.
- Submit to MATLAB *Grader*.

☐ Task: 1.3 **More Interactive Scripts**. [4 Marks]

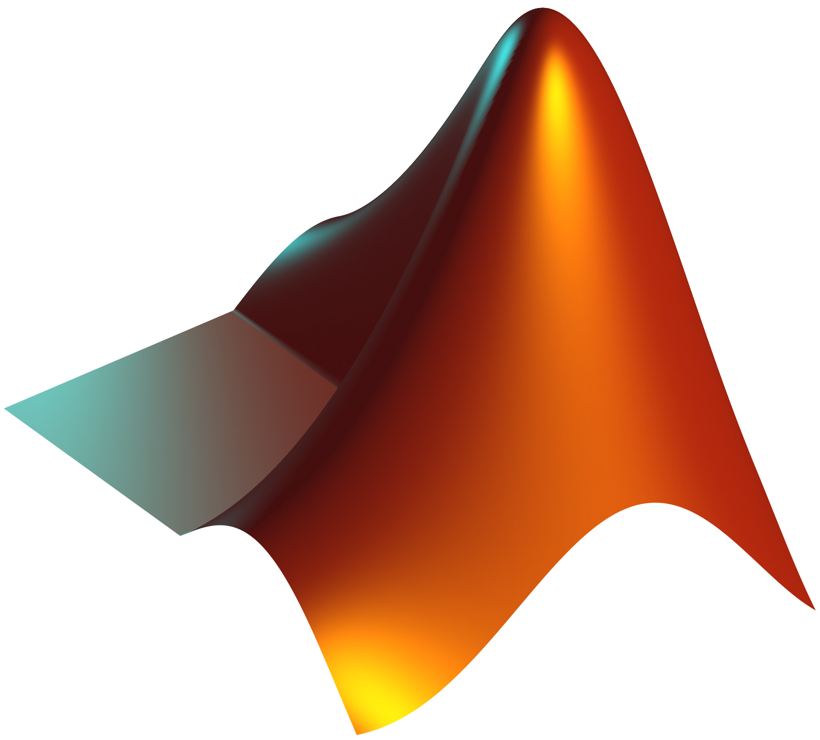
- Create and complete the script `CalcCircuitCurrentInteractive.m`.
- Upload your completed and commented script file to *Canvas*.

☐ Task: 1.4 **Command Line**, **Script Files And Functions**. [2 Marks]

- Read and think about the questions.
- Submit your answers to the *Canvas Quiz*.

ENGGEN 131
Introduction to Engineering Computation and Software Development

LAB 2
Functions, Problem Solving and Debugging



The Department of Engineering Science
The University of Auckland

Functions

✍ Task 2.1: Simple Functions

Create a simple function in MATLAB called `Cube` that cubes the function input.

Input:
  A number, or an array of numbers
Output:
  The input number cubed, or an array of cubed numbers

For example:
Calling `Cube(3)` will return a value of `27`.
Calling `Cube([1 2 3])` will return an array containing the values `[1 8 27]`.

We provide you with a simple test script `TestCube.m`, that calls the function `Cube` and displays the output of the function `Cube` and the expected output. Use the test script `TestCube.m` to test whether your function is working as intended.

Submit the completed function `Cube` to MATLAB *Grader*.

🛑 Stop: A Function, not a Script File!

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

Metric/Imperial Unit Conversion

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. Any program which needs to display output in imperial format could use a conversion function like this.

Recall The five steps for problem-solving:

1. State the problem clearly.
2. Describe the input and output information.
3. Work out the problem by hand (or with a calculator) for a simple data set.
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 out the problem by hand for a simple data set.

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 cm.

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

$$200 \div 2.54 = 73.7402$$

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

$$73.7402 \div 12 = 6.5617$$

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:

$$73.7402 - 6 \times 12 = 6.7402$$

So 2 metres is 6 feet 6.7402 inches.

Stop: Understand the Hand Worked Example!

Before continuing, ensure you understand 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 the 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 use your pseudocode as comments.

Download the file `MetricToImperial.m` from *Canvas*.

```
function [ft,in] = MetricToImperial(m)
% MetricToImperial 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
%
% Author: USA

% 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;

end
```

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

✵ Star: Naming Functions

You should choose function names that indicate what the function does. You can use multiple words in the same variable name either by capitalising 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 capitalising the first letter of each word. This convention helps distinguish them from MATLAB built-in functions that only use lowercase.

Step 5: **Test the solution with a variety of data.**

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

Important: remember that the `MetricToImperial` function returns **two** values. 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(2)
```

✵ Star: Testing your Code

You **must** test that your code works using several different 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. Writing code that works for one value but gives the wrong result in other cases is an easy pit to fall into.

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

✍ Task 2.2: Multi-Input Functions

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

You should work through the template below when completing this task.

We provide you with a simple test script `TestImperialToMetric.m` that calls the function `ImperialToMetric` and displays to the console both the output and the expected output of the function `ImperialToMetric`.

Calculate the remaining expected outputs by hand and use the test script `TestImperialToMetric.m` to test whether your function is working as intended.

Make sure you have commented your function.

Submit the completed function `ImperialToMetric` to MATLAB *Grader*.

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 out the problem by hand for a simple data set.

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 a flowchart below. Then write your code.

Stop: A Function, not a Script File!

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

Stop: Invalid Filenames

To avoid invalid filenames (and the associated errors), make sure filenames:

- Do not include spaces.
- Do not start with a number or include punctuation characters (other than the underscore).
- Avoid using reserved MATLAB commands as filenames.
  - If you are unsure about whether a filename is a reserved MATLAB command, try typing the word help followed by the filename you wish to use. If it is reserved you'll see an explanation of what that command or function does.

Check your filename now to see if it is valid.

Step 5: Test the solution with a variety of data.

You should use the provided test script to test that your function working. You can test more thoroughly by calling it from the command line with different inputs or by adding more tests to the test script.

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 $\frac{kx^2}{2}$ where k is the spring constant and x is the compression in the spring. The force required to compress the spring is kx. Table 2.1 gives the data for five springs.

Table 2.1: Spring Data

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

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

✍ Task 2.3: Multi-Input & Multi-Output Functions

Use the five steps for problem solving to write a function called `CalcSpringCompAndEnergy` that when given the force and the spring constant (in that order). It will return the compression and potential energy for that spring (in that order). This means your function will have **two** inputs and return **two** outputs.

You should work through the template below when completing this task.

Make sure it will still work for array inputs.

You are not provided with a test script. See Task: 2.4 in regards to testing your function (Task: 2.4 requires you to write a test script for this function)

Make sure you have commented your function.

Submit the completed function `CalcSpringCompAndEnergy` to MATLAB *Grader* **after** you have tested it.

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 out 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 a flowchart below. Then write your code.

Stop: Two Function Outputs!

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

Step 5: Test the solution with a variety of data.

✷ Star: Writing Test Scripts First

Some programmers follow the practice of writing their test scripts first **before** they write the important code (that will be tested with their test script). This can be a very good approach, as once your tests pass, you have some confidence that you have written your code correctly.

✍ Task 2.4: Creating Test Scripts

Create a test script called `TestCalcSpringCompAndEnergy.m` that tests the function `CalcSpringCompAndEnergy` from Task: 2.3.

You need to use at least three sets of input data, for example, from Table 2.1, the input values and corresponding expected output values (possibly hand calculated like in Step 3). You should display the values returned by your function **and** the expected values for the input data set.

You may use the test scripts provided for Task: 2.1 and Task: 2.2 to help you create a simple test script for this task.

Complete the test script `TestCalcSpringCompAndEnergy.m`, then use the MATLAB *Publish* feature to publish the code, comments and outputs into a single file. Submit the `.html` or `.pdf` file to *Canvas*.

Note: You can find a MATLAB *Publish* guide on *Canvas*.

STOP Stop: Check your TestScript Calls your Function!

It will not test your function very well if it does not call it!

Debugging

Debugging is a vital skill and one we will practice in each lab. Two main categories of things that can go wrong (which you may experience when completing the Lab Tasks):

- Syntax errors, which prevent it from running at all.
- Bugs, which means that although it might run, it does not produce the desired results.

Syntax errors are a little easier to track down, as MATLAB will helpfully point out lines that generate syntax errors (including an informative message that will help you understand what caused the problem once you get used to interpreting the sometimes cryptic sounding language). When dealing with syntax errors, ensure you read the error message before trying to fix the line, as it will often give you a vital clue about the error.

In lectures, you saw how MATLAB helps you find bugs in your code. Using the MATLAB *Debugger* (M-Lint), you can identify errors while editing your code. There is also the menu Debug for stepping through your programs to identify and fix errors. Stepping through a section of code line by line can be an invaluable tool when trying to spot the problem. Remember, you can view the variables in the workspace while doing this to track whether they match what you expect them to be.

✍ Task 2.5: Debugging Functions

A group of wealthy University of Auckland Engineering graduates decide to make New Zealand great again. They want to build an enormous square-based pyramid of solid gold floating on Waitematā Harbour (Auckland Harbour). Although they advertised the pyramid to be made of solid gold, it was too expensive and secretly decided to scam the public by building it with compressed trash and painting the surface with gold paint.

They have several different pyramid base length and height dimension configurations to choose from. They have created a function `PyramidSAandVol` to calculate the square-based pyramid's surface area and volume so that they can make an informed decision on how much paint and trash they need. However, their function `PyramidSAandVol` has many bugs and does not work as expected.

To calculate the volume of a pyramid with a base length of 6 m and height of 9 m, they **should** be able to use the call:

```
>> [SA, Vol] = PyramidSAandVol(6, 9);
```

and end up with a surface area of 149.842 m^2 as their first output and a volume of 108 m^3 as their second output. It should also work with array inputs to test multiple combinations at once.

Open the function `PyramidSAandVol` (and script `TestPyramidSAandVol.m`) in the MATLAB Editor, fix any bugs in the function and use it to calculate the surface area and volume of square-based pyramids.

Extend the script `TestPyramidSAandVol.m` to thoroughly test the function.

Submit the debugged function `PyramidSAandVol` to MATLAB *Grader*.

Study: Approximating the Normal Distribution

The standard normal distribution is described by the function:

$$s(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$$

The probability of randomly selecting a value within the range $-\alpha$ to α is then given by:

$$p = \int_{-\alpha}^{\alpha} s(x)\ dx = 2 \int_{0}^{\alpha} s(x)\ dx$$

Unfortunately, you cannot calculate this integral analytically.

Write a MATLAB function which takes a single input value x and returns the value $s(x)$. Write your function so that it will work on 1D arrays.

Write a second MATLAB function which takes a single input value α and returns the probability p of obtaining a value in the range $-\alpha$ to α.

Your function should use the trapezium method to integrate $s(x)$ numerically.

Recall that the trapezium method approximates the area under a curve by summing up a number of thin trapeziums, as illustrated in Figure 2.1.

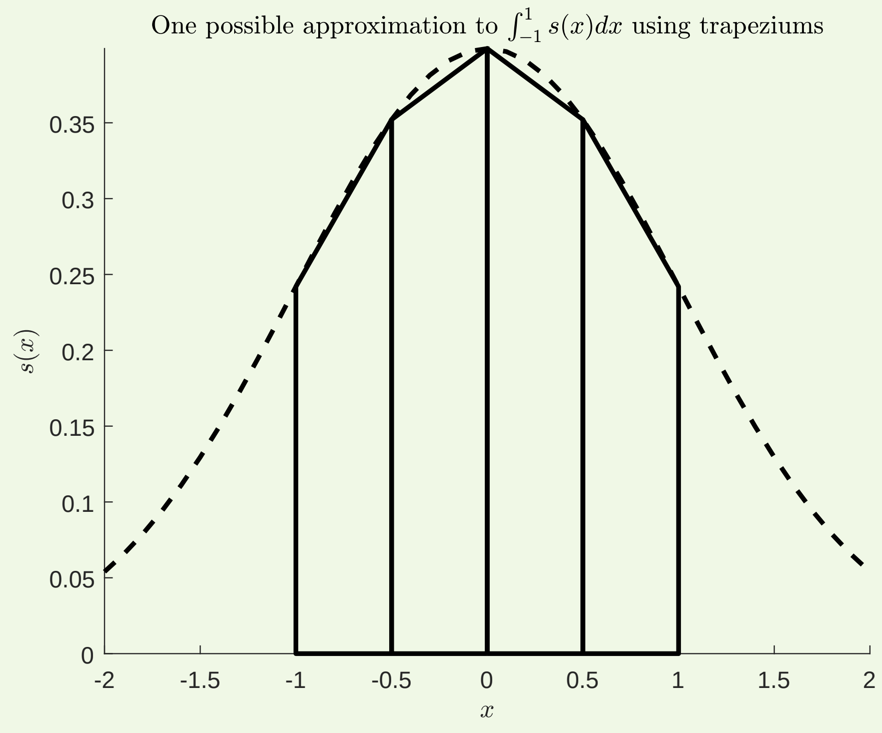


Figure 2.1: Trapizium Method Example

Pseudocode for the second function is provided:

```
create an array of 100 x values from 0 to alpha
width = x(2) - x(1)
Heights = s(x)
LeftHeights = Heights array with the last element removed
RightHeights = Heights array with the first element removed
Areas = (LeftHeights + RightHeights)/2 * width
Total Area = sum the Areas
p = 2 times the total area
```

✍ Task Submission Summary

☐ Task: 2.1 **Simple Functions**. [2 Marks]

- Submit the completed function `Cube` to MATLAB *Grader*.

☐ Task: 2.2 **Multi-Input Functions**. [2 Marks]

- Submit the completed function `MetricToImperial` to MATLAB *Grader*.

☐ Task: 2.3 **Multi-Input & Multi-Output Functions**. [2 Marks]

- Submit the completed function `CalcSpringCompAndEnergy` to MATLAB *Grader*.

☐ Task: 2.4 **Creating Test Scripts**. [2 Marks]

- Use MATLAB *Publish* to publish the completed script `TestCalcSpringCompAndEnergy.m` to a single file.
- Submit the published `.html` or `.pdf` file to *Canvas*.

☐ Task: 2.5 **Debugging Functions**. [2 Marks]

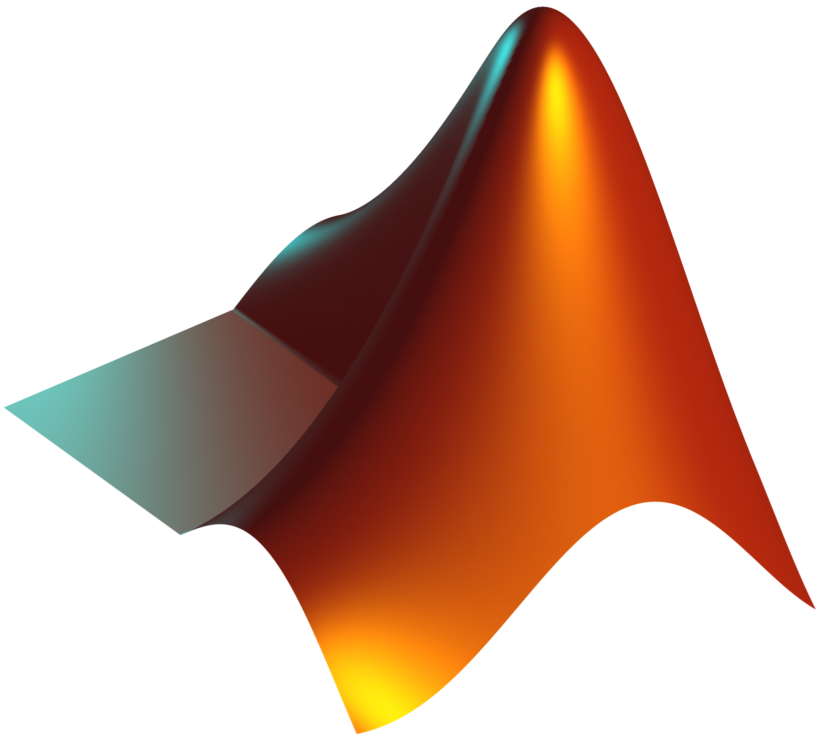
- Submit the debugged function `PyramidSAandVol` to MATLAB *Grader*.

ENGGEN 131

Introduction to Engineering Computation and Software Development

LAB 3

Logical Operators, Conditional Statements and Loops



The Department of Engineering Science

The University of Auckland

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 are unsure about the output from the statements above, ask a teaching assistant 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
```

You can expand the syntax of an `if` statement so that if the condition is not true, do something else instead:

```
if Condition
    Do something
else
    Do something else
end
```

You can further expand the `if` statement with 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 to get a useful message regardless of the number value. 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` using the debugger 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` using the debugger for each of the different values for `number`.

Tip: Be careful to use == when checking for equality

We used **two** equality signs == to check if a number was equal to `1`. A very common programming error is accidentally using one equality sign = when you want to check for equality. In MATLAB, this will result in an error.

Star: Indenting

Properly indenting your script files makes them much easier to read and debug. If you enclose commands within a statement (conditional, loop), you should indent the commands (usually with four spaces or using the tab key Tab). Commands at the same level should have the same indentation. Correct indentation often helps you identify if you have left an `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 right-click menu (the shortcut is Ctrl+I), which will indent any selected text. You can use this in your script files to ensure correct indenting. If you wish to use smart indent on your entire file, the shortcut for selecting all text is Ctrl+A.

Remember: use Ctrl+A followed by Ctrl+I to ensure indenting is correct.

Tip: Be Careful of Logical Operator Precedence

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

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

which will **not** do what you might expect. Without brackets, MATLAB will first evaluate `number == 7`, giving `true` or `false`, and then | (or) this with the value `11`. As `11` is always interpreted as `true`, the conditional will always evaluate as `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, we could omit the brackets, but it is a good idea to include them to clarify what is being done and to avoid any problems with precedence.

✍ Task 3.1: Conditionals

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, then the polynomial has one repeated root.
- Otherwise, the discriminant is negative, and the polynomial has two complex roots (i.e., no real roots).

Knowing how many real roots a quadratic equation has can have important consequences. For example, you obtain a quadratic characteristic equation when solving 2nd order linear homogeneous ODES with constant coefficients. Determining how many real roots your characteristic equation has tells you important information about the nature of the solution to the ODE.

Write a function called `NumRealRootsQuadratic` that takes as inputs the coefficients a, b and c and returns the number of real roots a quadratic equation has. An example of calling your function:

```
    >> n = NumRealRootsQuadratic(1,3,2)
```

After running this function, you would expect `n` to have assigned the number of real roots for the quadratic $x^2 + 3x + 2$. In this case, `n` is assigned the value `2`.

We provide you with a more advanced test script `TestNumRealRootsQuadratic.m` that calls the function `NumRealRootsQuadratic` and displays whether a test passes or fails.

Use the template given on the following few pages to develop your function.

Submit the completed function `NumRealRootsQuadratic` to MATLAB *Grader*.

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 out the problem by hand for a simple data set.

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. Try to have at least three input values, one for each possible output value.

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

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

Stop: Input and Disp within Functions

Generally, values are passed into functions when called, either from the command line or within an `.m` file. When called with inputs, the function will return the desired result as an output. Functions should not ask users to enter a value (usually done in scripts), so your function should not ask you to enter inputs using the `input` command Instead, you should pass values into your function when you call it. If you use the `input` command within a function to assign a value to an input, you will probably be overwriting whatever value was passed into the function when it was called.

In general, functions should also not display anything unless the function 'output' **is** the display.

Step 5: Test the solution with a variety of data.

Use the provided test script as a starting point for testing your function. Include more tests that cover a variety of data, including the problem you worked by hand in Step 3.

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 a script called `FirstTenPrimes` to print 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
```

Running `FirstTenPrimes` would produce the following output:

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

Tip: Break Out of an Infinite Loop

When you begin working with loops, it is likely at some point that you will accidentally create an infinite loop. You can stop the code from running by pressing the Ctrl + C keys (you may need to be active in the MATLAB *Command Window*).

Alternatively, you may have to end the MATLAB process using the *Task Manager*.

Task 3.2: Mission Impossible?

Some activities carry with them a relatively high risk of death. For example, around 4% of climbers who attempt to summit Everest die. The physicist Richard Feynman estimated that the chances of failure of a space shuttle mission with loss of vehicle and life were roughly 1%. Tragically two shuttles were lost.

Flights can be determined to be successful or not. To do this, we will compare a randomly generated value (between 0 and 100) and see if it is in the range 0 to 1 (a value in this range **inclusive** will be deemed a failed mission corresponding to the 1% chance of losing a shuttle).

Write a function called `IsMissionSuccessful`, whose input is a single number (between 0 and 100) and returns `true` if the mission is successful; otherwise, return `false`.

We do not provide you with a test script, but you should write one yourself (possibly based on Task: 3.1 or those from Lab 2). You can also test your function by passing in a randomly generated value.

Hint: You can generate a random number between 0 and 100 using the function `rand` (see `help rand` for more information). The function `rand` generates a random value between 0 and 1, and we can multiply this value by 100 to get a value in the range 0 to 100:

```
    100 * rand
```

Try calling your function `IsMissionSuccessful` with a randomly generated value between 0 and 100, for example:

```
    v = 100 * rand
    result = IsMissionSuccessful(v)
```

Does it behave as expected?

Submit the completed function `IsMissionSuccessful` to MATLAB *Grader*.

✍ Task 3.3: While Loops

As a follow-on from Task: 3.2, this raises the question of how many shuttle missions could NASA have expected to launch before losing a shuttle.

Write a function called `NumberOfSuccessfulMissions`, whose input is a 1D array of random numbers (between 0 and 100) and returns a number indicating the number of successful missions before a failure.

Use a `while` loop to scan the inputted random numbers and the function `IsMissionSuccessful` from Task: 3.2 to count how many successful missions occur before a failure.

Calling:

```
n = NumberOfSuccessfulMissions([ 4.4 3.3 2.2 1.1 0.4 5.5]);
```

will return a value of `4`, as there were 4 successful missions before failure (indicated by the 0.4).

Note that it is possible for all missions to succeed, for example, calling:

```
n = NumberOfSuccessfulMissions([12.3 45.2 17.8 91.2]);
```

will return a value of `4`, as all missions were successful (none failed since no values were below 1).

Hint: You can generate a 1D array of n random values between 0 and 1 by using the `rand` function as follows:

```
    v = rand(1, n);
```

If you would like to test your code with a list of random values between 0 and 100 that MATLAB can generate, try using something like the following:

```
    v = 100 * rand(1, 5)); % 5 values between 0 and 100
    n = NumberOfSuccessfulMissions(v);
```

Hint: You will need to think very carefully about what happens if you reach the end of the array without detecting a failed mission, as you don't want to write code that tries to look at values beyond array's length. When using loops to move through arrays, it is very common to get the following error:

```
    Index exceeds the number of array elements
```

This error results from accessing an array element that does not exist, for example, asking for the value in position 11 of an array with only ten values.

We do not provide you with a test script, but you should write one yourself (possibly based on Task: 3.1, Task: 3.2, or those from Lab 2).

Submit the completed function `NumberOfSuccessfulMissions` to MATLAB *Grader.*

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`, 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 `i` or `j` as a counter variable name. You can use any valid variable name for your array counter. However, a descriptive counter is best.

If you are working with complex numbers, using a different counter name (i.e., not `i` or `j`) is a good idea to avoid confusion with the special MATLAB variables used to represent the square root of negative 1.

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. The loop stops after the stop value has been passed, even if the counter variable does not take this value.

If you want to write out the numbers from 10 to 1, 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 are not sure, ask a teaching assistant to clarify things.

For Loops and 1D Arrays

A `for` loop is one way to populate a 1D array (i.e., a list of values). We often use 1D arrays as data structures for plotting. For example, if we wanted to draw $y = x^2$ between 1 and 5, we 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 a plot like Figure 3.1.

Examining `x` and `y` in the command line shows how the `for` loop works:

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

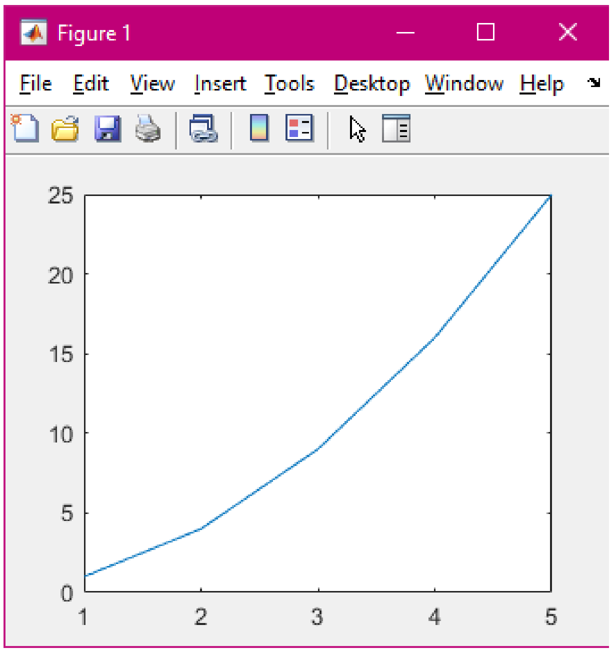


Figure 3.1: MATLAB plot of $y = x^2$.

If we want to create a "high definition" plot of a more complex function:

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

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, e.g. you could use element by element operations).

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

The resulting plot is shown in Figure 3.2.

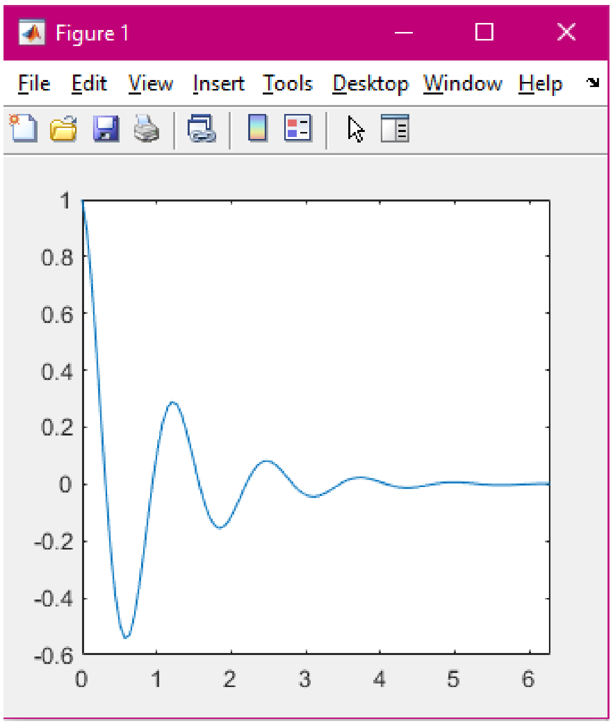


Figure 3.2: MATLAB plot of $y = e^{-x}\cos(5x)$.

✍ Task 3.4: For Loops

When dealing with uncertain processes (like Task: 3.3), we are often interested in the most likely outcome. Running the code once only produces a single value, but how do we know whether this value was likely or not? For example, if someone told us we ran 100 successful space shuttle missions, were we lucky or unlucky to have made it to that many missions?

One way to determine this outcome is via Monte Carlo simulation. We simulate the process many times (by calling the function `NumberOfSuccessfulMissions` multiple times with a different 1 by 1000 array of random numbers every simulation) and store all the outcomes (in this case, the number of successful missions). We can then study the list of outcomes statistically.

Create a script that asks the user how many simulations they would like to run and then uses a `for` loop to perform the required number of simulations. For each simulation, store the number of successful missions in a 1D array.

To investigate the distribution of the number of successful missions, we can use the function `histogram` to draw a histogram of the number of missions:

```
histogram(successful_missions)
```

You may use the provided function `SuccessfulMissionHistogram` to plot the successful missions (stored in the 1D array) or write your own code.

For example, a histogram for 100 simulations may look like Figure 3.3.

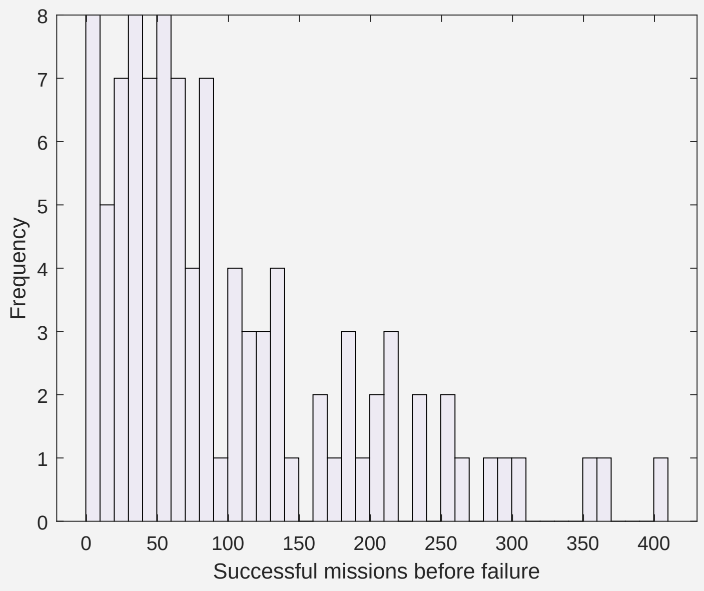


Figure 3.3: Example Shuttle Monte Carlo Simulation Histogram.

Figure 3.3 may differ from what you may obtain because of the random numbers generated. Try running your script several times, entering a different number of simulations each time to get a feel for how many simulations are required to get an accurate picture of the distribution.

Find a suitable number of simulations (you may need to hard code the number of simulations). Then, use the MATLAB *Publish* feature to publish the code, comments and outputs into a single file. Submit the `.html` or `.pdf` file to *Canvas*.

Note: You can find a MATLAB *Publish* guide on *Canvas*.

✍ Task 3.5: Debugging Conditionals

The Auckland Harbour Bridge has eight lanes for traffic, with four northbound and four southbound lanes. During peak traffic in the morning and afternoon, the number of lanes in each direction changes to accommodate the increased traffic flow. In addition, accidents in lanes can also halt some traffic flow over the bridge.

However, Auckland traffic over the bridge is still terrible, and after an extensive investigation, the investigation found that the software deciding the traffic lane assignment is faulty. The people in charge have tasked you with fixing the faulty software and have isolated the faults to the function `Traffic_Lane_Assignment`, which has many of bugs and does not work as expected. All function comments in the header and body provide details on how the function is supposed to work.

The script `Traffic_Lane_Assignment_Test.m` calls the function

`Traffic_Lane_Assignment` and displays the traffic lane directions.

Open both files in the MATLAB *Editor*, and read through the code and the comments to better understand the code's intention and current function. Then fix any bugs in **only** the function.

Conditional bugs can be quite tricky to spot. Be on the lookout for examples where a single equals sign is used to check equality (rather than the required two), and make sure you think carefully about precedence (see the tip earlier in the lab).

Remember, you can use the debugger to step through line by line (and you can step into a function) to see how each line of code changes the variables in the workspace.

Submit the debugged function `Traffic_Lane_Assignment` to MATLAB *Grader*.

Study: Stress, Strain, and Shear

For the next America's Cup, Team New Zealand are trialling a carbon fibre tiller on a boat. Under the most extreme conditions likely to be experienced, this component will be subjected to a 2D strain given by the strain matrix:

$$\mathbf{e} = \begin{pmatrix} e_{xx} & e_{xy} \\ e_{xy} & e_{yy} \end{pmatrix} = \begin{pmatrix} 0.230 & -0.009 \\ -0.009 & -0.080 \end{pmatrix}$$

This matrix represents the normal and shear strains experienced by the component in two dimensions. Applying these strains to the component generates stresses in the material. We are tasking you with finding the angle of orientation of fibres in the , which minimises the maximum shear stress experienced within the component.

To do this, you must first download `GetMaxShearStress.m` and `strainData.mat` from *Canvas*. Use the command:

```
    >> load strainData.mat
```

to load the variables `e` and `C` into the workspace. `e` is the strain matrix given above, and `C` (a "compliance matrix") contains information on the material's mechanical properties.

Using the function `GetMaxShearStress.m` (where we have provided some comments), write a script that uses a `for` loop to determine the angle of orientation, which minimises the maximum shear stress in the component for the given applied strain. Your function should calculate an array of maximum shear stress values for 1000 angles starting at 0 and going through to $\pi/2$. It should then determine which angle minimises the maximum shear stress and display a message:

```
The fibre angle that minimises the shear stress is _ radians
```

where the value replaces _.

Hint: You may find the `min` function helpful. Use MATLAB Help to investigate how to use it.

Study: Finding Roots

The following algorithm, given in pseudocode, is used to find a root of a mathematical function $f(x)$ (i.e., the input value for a function that gives zero).

```
Get initial guess, x, from user
Get tolerance from user
fx = f(x)
h = 0.000001
while |fx| > tolerance
fx = f(x)
   dfdx = (f(x+h)-f(x-h))/2h
   x = x - fx / dfdx
end
display value for root, x
```

Write a MATLAB script that implements the above algorithm to find a root of the `sin` function when given an initial guess and tolerance.

A sample session of your script running is shown below:

```
Please enter the initial guess:  3
Please enter the tolerance:  0.001
Root of sin(x) is 3.142
```

Important: Make sure you have commented your script file.
Hint: The absolute value function in MATLAB is called `abs`.

✍ Task Submission Summary

☐ Task: 3.1 **Conditionals**. [2 Marks]

- Submit the completed function `NumRealRootsQuadratic` to MATLAB *Grader*.

☐ Task: 3.2 **Mission Impossible?** [2 Marks]

- Submit the completed function `IsMissionSuccessful` to MATLAB *Grader*.

☐ Task: 3.3 **While Loops**. [2 Marks]

- Submit the completed function `NumberOfSuccessfulMissions` to MATLAB *Grader*.

☐ Task: 3.4 **For Loops**. [2 Marks]

- Use MATLAB *Publish* to publish the completed script to a single file.
- Submit the published `.html` or `.pdf` file to *Canvas*.

☐ Task: 3.5 **Debugging Conditionals**. [2 Marks]

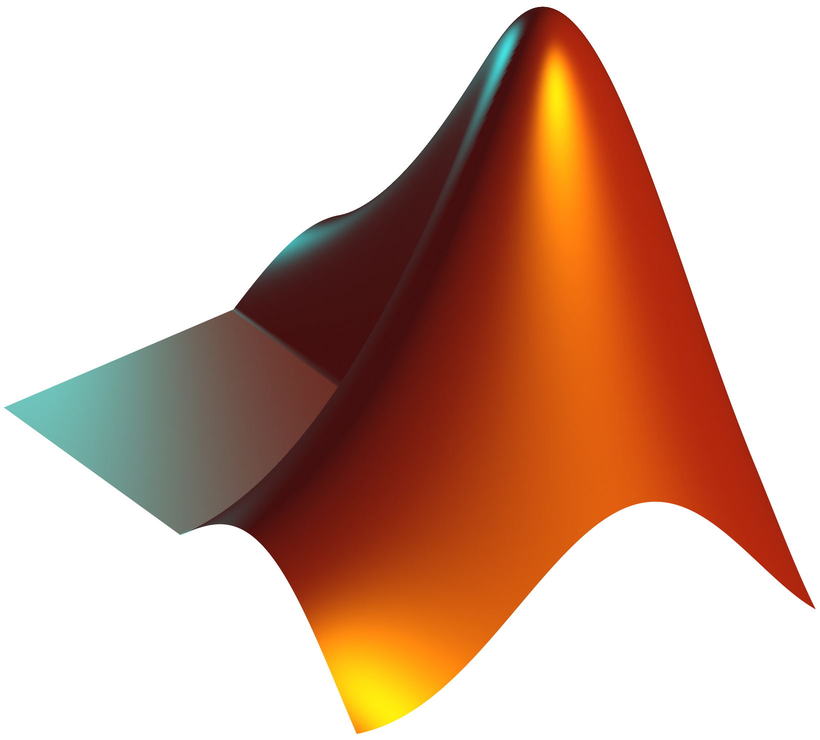
- Submit the debugged function `Traffic_Lane_Assignment` to MATLAB *Grader*.

ENGGEN 131

Introduction to Engineering Computation and Software Development

LAB 4

Graphics and Image Processing



The Department of Engineering Science

The University of Auckland

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 created and start working on a new figure, use the `figure` command.

This command creates a new figure window for drawing on. You can choose to draw on the existing figure window *Figure n* by using the command:

```
figure(n)
```

For more information, look up `figure` using MATLAB Help.

Plotting Multiple Data Sets

When plotting multiple data on the same axes you should use different colours/line styles and include a legend. Refer to your lecture notes to ensure you know how to modify line colours and styles.

Signal Processing – Background for Task: 4.1, Task: 4.2 and Task: 4.3

Electrocardiography (ECG) is a technique to record the heart's electrical activity and determine the rate and regularity of a test subject's heartbeat.

The data is measured using electrodes attached to the skin's surface, which detect electrical impulses generated by the heart and display them as a waveform.

Often, the signal provided by these electrodes suffers from electrical noise. Signal processing techniques can reduce this noise and provide data that suffers from less uncertainty. In Task: 4.1, you must create a function to smooth the signal once, reducing the signal's noise. Then in Task: 4.2, you must write a function that will smooth the signal multiple times. Finally, in Task: 4.3, we plot the original unprocessed signal and smoothed versions.

Three-Point Smoothing

One way to smooth a signal is to use a three-point smoothing algorithm. The basic idea is that you can create a smoother version of a signal by creating a new array, where each element (i.e., point) in the new array is found by taking a weighted average of the corresponding element and the two elements on either side of it.

With a bit of thought, it should be evident that we can not smooth the first and last elements in the array using this idea, as they only have an element on one side of them (so their values will remain the same).

We can smooth all other elements by finding a weighted average of the corresponding element from the original array (along with the elements on either side of it).

The weighted average of three consecutive elements (i.e., points) e_1, e_2 and e_3 is found using the formula

$$\frac{e_1 + 2 * e_2 + e_3}{4}.$$

For example, if we have three values, 4, 1, 12 then the weighted average of these values will be

$$\frac{4 + 2 * 1 + 12}{4} = \frac{18}{4} = 4.5\,.$$

We have provided you with a function called `ThreePointSmooth` that will take as an input a 1D array containing three points in a row and calculate this weighted average for you.

Now consider a small array `[4 1 12 3 8]`, smoothing it once using this idea. The first and last values stay the same so that we can fill them in as `[4 ? ? ? 8]`.

Element 2 in the smoothed array will be the weighted average of elements 1, 2 and 3 from the original array (i.e., the weighted average of 4, 1 and 12)

$$\frac{4 + 2 * 1 + 12}{4} = \frac{18}{4} = 4.5\,.$$

Now, this gives us `[4 4.5 ? ? 8]`.

Element 3 in the smoothed array will be the weighted average of elements 2, 3 and 4 from the original array (i.e., the weighted average of 1, 12 and 3)

$$\frac{1 + 2 * 11 + 3}{4} = \frac{28}{4} = 7\,.$$

Now, this gives us `[4 4.5 7 ? 8]`.

Element 4 in the smoothed array will be the weighted average of elements 3, 4 and 5 from the original array (i.e., the weighted average of 12, 3 and 8)

$$\frac{12 + 2 * 3 + 8}{4} = \frac{26}{4} = 6.5\,.$$

Finally, this gives us `[4 4.5 7 6.5 8]`.

We have now completed **one** pass of three-point smoothing on the array.

Now we can repeat the process on our smoothed array `[4 4.5 7 6.5 8]` to get a slightly smoother signal, which would be the second pass. We can then smooth that signal, which would be the third pass. Repeated passes should produce smoother and smoother versions of the signal.

Try smoothing a signal for several passes by completing Table 4.2.

Table 4.2: Signal Smoothing Example.

Num Passes	Signal				
0	4	1	12	3	8
1	4	4.5	7	6.5	8
2					
3					
4					
5					

✍ Task 4.1: Signal Smoothing

Write a function called `SmoothSignalPass` that applies the function `ThreePointSmooth` to smooth the signal **once** (i.e., it performs one pass of the three-point smooth algorithm). Your function should return the smoothed signal.

Input:
  An array containing the unsmoothed signal.
Output:
  An array containing the smoothed signal.

Your function should call the function `ThreePointSmooth` multiple times. Think about how you will loop over all the elements in the unsmoothed signal.

Be careful with the endpoints of your input array; remember, they get treated a little differently.

Ensure you read the documentation in the header of the supplied function `ThreePointSmooth` to understand how this function works and the required input.

You should create a test script to test the function `SmoothSignalPass`, possibly with some of the data from the table you have completed.

Here is an example call:

```
s = SmoothSignalPass([4 1 12 3 8]);
```

that should return the array `[4 4.5 7 6.5 8]`.

Submit the completed function `SmoothSignalPass` to MATLAB *Grader*.

✍ Task 4.2: Signal Smoothing n Times

Write a function called `SmoothSignal` that will smooth a signal `n` times (i.e., it does `n` passes) and returns the resulting smoothed signal.

Inputs:
  An array containing the unsmoothed signal.
  The number of times to smooth the signal `n`.
Output:
  An array containing the smoothed signal.

Your function should call your `SmoothSignalPass` function `n` times.

Remember that from the second pass onward, you should be working on smoothing the values found from the previous pass (not the original unsmoothed signal).

You should create a test script to test the function `SmoothSignal`, possibly with data from the table you have completed.

Submit the completed function `SmoothSignal` to MATLAB *Grader*.

✍ Task 4.3: Plotting Signals

For this task, write a **single** script file which produces **two separate** figures:

- **Figure 1**: A single set of axes showing the unprocessed signal (in green) and a processed signal (in dotted red) obtained after 100 passes of the three-point smoothing algorithm on the original signal.
- **Figure 2**: Three sets of axes showing the unprocessed signal, the signal after one pass of the three-point smoothing algorithm, and the signal after 100 passes of the three-point smoothing algorithm. Use solid black lines for each of these plots.

All figures also require appropriate axis labels, ranges, titles and legend.

We have provided you with a signal in the file `signal.mat`, with the file in your working directory, use the command:

```
load signal.mat
```

in your script file to load two variables, `y` and `t`, into the workspace. `y` is an array of values giving the measurements taken by the electrodes (i.e., the signal), and `t` is an array of the corresponding relative time at which they were taken.

The figures you should expect to obtain are presented in Figure 4.1 as **Figure 1** and Figure 4.2 as **Figure 2**.

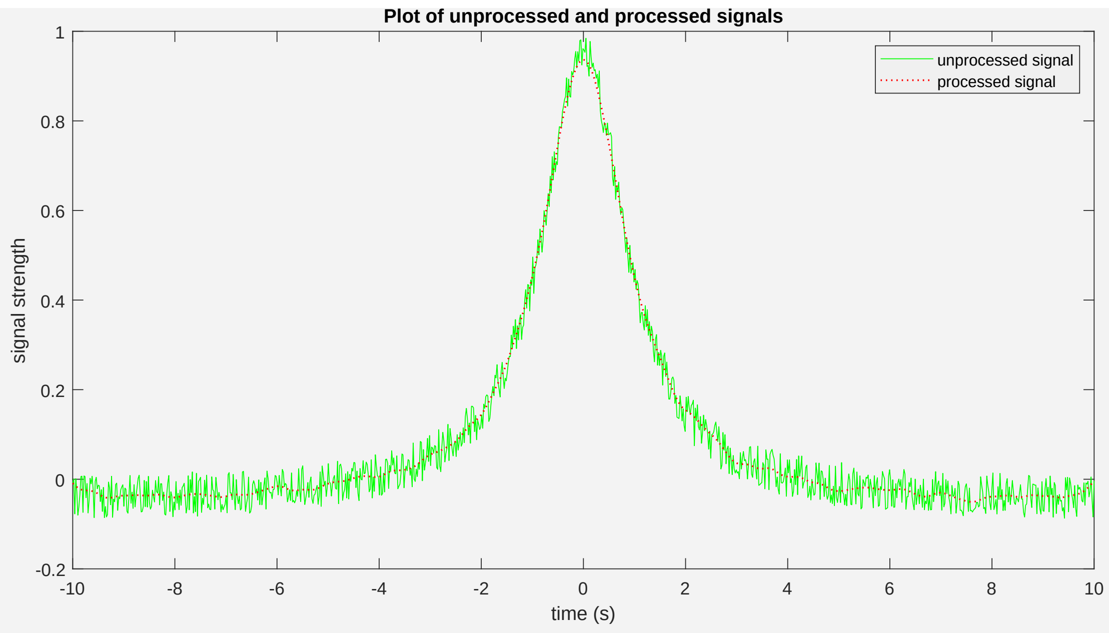


Figure 4.1: Expected plot for **Figure 1**.

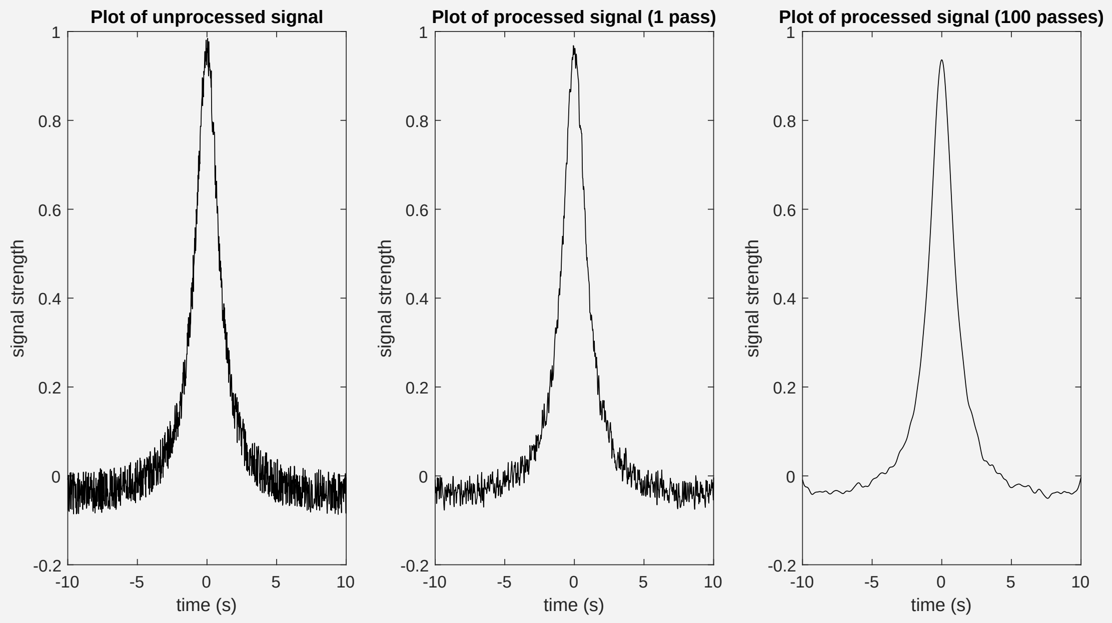


Figure 4.2: Expected plot for **Figure 2**.

Complete the script, then use the MATLAB *Publish* feature to publish the code, comments and outputs into a single file. Submit the `.html` or `.pdf` file to *Canvas*.

Note: You can find a MATLAB *Publish* guide on *Canvas*.

Tip: Hold On

The command `hold on`, allows you to retain the current plot when adding new plots instead of replacing the old plot. That is, it will allow you to draw multiple plots on the same figure.

For example:

```
figure
hold on
plot(x, y_1)
plot(x, y_2)
```

will plot both `y_1` and `y_2` in the same figure.

Alternatively, using `plot(x, y_1, x, y_2)` will also work but will not be very clear when you require many different plots on the same figure.

Stop: One Single Script, Two Figures

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

Image Processing

MATLAB allows you to display many different image files using the functions `imread` and `image`. For example, you can read the supplied `xray.jpg` file (available for download from *Canvas*) using the command:

```
>> xray_RGB = imread('xray.jpg', 'JPG');
```

The variable `xray_RGB` is a 3-dimensional variable of unsigned 8-bit integers. Unsigned 8-bit integers can not store negative numbers, fractions or values larger than 255. Attempting to assign values that cannot be represented as unsigned 8-bit integers to an array of unsigned 8-bit integers can produce unexpected results. Depending on what you want to do with the array's contents, you may first need to create an array of standard numerical values (i.e., doubles) using the function `double`. If required, you could always convert back to unsigned 8-bit images using the function `uint8`.

The first two dimensions represent the location of pixels in the image, and the third dimension identifies whether we are dealing with the amount of red, blue or green. MATLAB represents images using the RGB colouring for the pixels.

```
>> size(xray_RGB)
```

```
ans =
    1024    745    3
```

Thus, pixel `(i, j)` has:

- Red content `xray_RGB(i, j, 1)`.
- Green content `xray_RGB(i, j, 2)`.
- Blue content `xray_RGB(i, j, 3)`.

For example, consider the *Colors* window shown in Figure 4.3 for choosing various RGB colours in Microsoft Office.

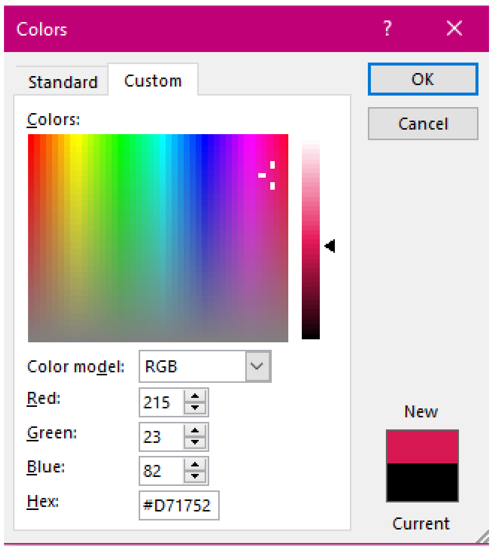

Color model: RGB
Red: 215
Green: 23
Blue: 82
Hex: #D71752

Figure 4.3: Microsoft Office Colors Window.

To display an image represented by an *m* by *n* by 3 matrix of unsigned 8-bit integers, you input the matrix into the function `image`. To ensure each pixel is square, set the axis to be equal.

```
>> image(xray_RGB)
>> axis equal
```

If you have read in the supplied image, try typing the above commands to see what the image looks like.

Note that the x-ray image is used under the licensing terms CC BY 2.5 and can be found here.

Stop: uint8

Remember that to draw an image, you must ensure the array passed into the function `image` contains data stored as unsigned 8-bit integers.

You can create an array of unsigned 8-bit integers from a standard array of numerical values by using the function `uint8`.

✍ Task 4.4: What Colour Is The Sky?

Airport security staff use x-ray imaging to scan luggage and detect dangerous objects such as weapons. Metal objects are inorganic and typically show up as blue on airport x-rays. So being able to detect blue pixels can be crucial.

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 `IsPixelBlue` that recognises if a pixel is blue by examining its red, green and blue values.

Input:
Red intensity.
Green intensity.
Blue intensity.
Output:
A Boolean value indicating whether the pixel is blue or not.

You should create a test script to test the function `IsPixelBlue`. Your tests should include trying the following calls:

`IsPixelBlue(10, 20, 200)`	should return `true`.
`IsPixelBlue(120, 180, 0)`	should return `false`.

Submit the completed function `IsPixelBlue` to MATLAB *Grader*.

✍ Task 4.5: Image Filtering

For this task, you need to create a filtered image to aid airport security staff, replacing all pixels **except** the blue pixels with the colour white.

The RGB values for white are $(255,\ 255,\ 255)$.

Write a function called `FilterImage` that filters the x-ray image to make the inorganic material more noticeable.

Input:
A 3D image array of `uint8` values.
Output:

A filtered 3D image array of `uint8` values.

It should examine each pixel in turn and replace it with a white one if it is **not** blue. You should use the function `IsPixelBlue` that you wrote in Task: 4.4 to determine if pixels are blue or not.

You should create a test script to test the function `FilterImage`, be sure to read and display the image before filtering it. Also, be sure to display the final filtered image in a separate figure in your test script.

You may use the provided image files: `test.png`, `logo.png` and `xray.png` to test your function.

Does your filter work completely? If not, why not?

Submit the completed function `FilterImage` to MATLAB *Grader*.

Drawing Surfaces

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 are not sure how `meshgrid` works or why it is important, make sure you ask a teaching assistant to clarify things.

✍ Task 4.6: Methods For Surface Plotting

The script `PlotSurfaces.m` uses two methods to draw surfaces over the domain: $-1 \leq x \leq 1$ and $-2 \leq y \leq 2$.

- **Method 1**: Using a nested `for` loop and the dot operator `.` to create the surface
$$f(x,\ y) = \frac{1}{e^{(5x)^2 \cdot (5y)^2}}\,.$$

- **Method 2**: Using `meshgrid` and the dot operator `.` to create the surface

$$f(x,\ y) = y\sin(5x) - x\cos(5y)\,.$$

When the script `PlotSurfaces.m` is run, it should display two plots, one subplot for each method, side by side in two columns on a single figure. However, the script `PlotSurfaces.m` has bugs. Read through the script to better understand the script's intentions and then fix any bugs in the script. Remember, you can use the debugger to step through line by line to see how each line of code changes the variables in the workspace. The expected figure from the script `PlotSurfaces.m` with 10 grid points equally spaced along each axis should look like Figure 4.4.

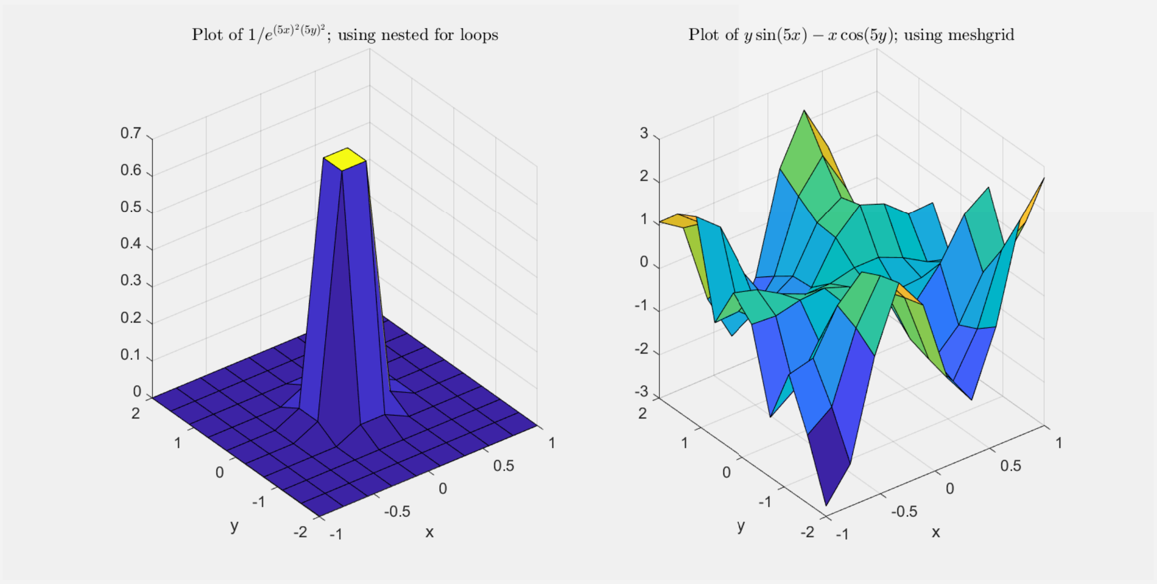


Figure 4.4: Expected figure from `PlotSurfaces.m` for 10 grid points evenly spaced.

Make sure that your surfaces look the same as Figure 4.4 when using 10 grid points. Try increasing the number of points to 100 in each direction and see what happens to your surfaces. Click and drag each surface plot to change the view. Also, you may adjust the plotting options like the edgecolor, colormap, etc., to your liking.

Edit the debugged script with 100 grid points equally spaced in each direction, then use the MATLAB *Publish* feature to publish the code, comments and outputs into a single file. Submit the `.html` or `.pdf` file to *Canvas*.

Note: You can find a MATLAB *Publish* guide on *Canvas*.

Study: Luminance

We can convert a colour RGB image into a grayscale image by replacing the values of red, green and blue for each pixel with a value representing the luminance l for that pixel.

The formula for calculating the luminance l for a given pixel $(r,\ g,\ b)$ is

$$l = 0.3r + 0.59g + 0.11b\,.$$

Setting the r, g and b values all to be l will result in a shade of gray.

Write a function called `luminance` that will calculate the luminance l for a given colour. Your function should take as inputs three values (representing the amount of red, green and blue) and return the luminance l.

Write a second function called `ConvertToGrayscale` which will convert a colour RGB image into a grayscale image by using the function `luminance`. The function `ConvertToGrayscale` should take as input a 3D array describing a colour image and then return a 3D array describing the equivalent grayscale image.

Study: Image -> Ima | ge -> Im | a -> I | m

Write a script file called `halveImage.m` that scales the size of an image down by a factor of 2 and then draws it to the screen.

Your script should work by replacing each 2 by 2 square selection of pixels in the old image with a single pixel in the new image. The RGB values for the new pixel will be the average of the RGB values for the 4 pixels from the 2 by 2 square.

You may assume that your image is an even number of pixels wide and high.

Hint: You will likely need to use the function `double` and the `uint8`.

✍ Task Submission Summary

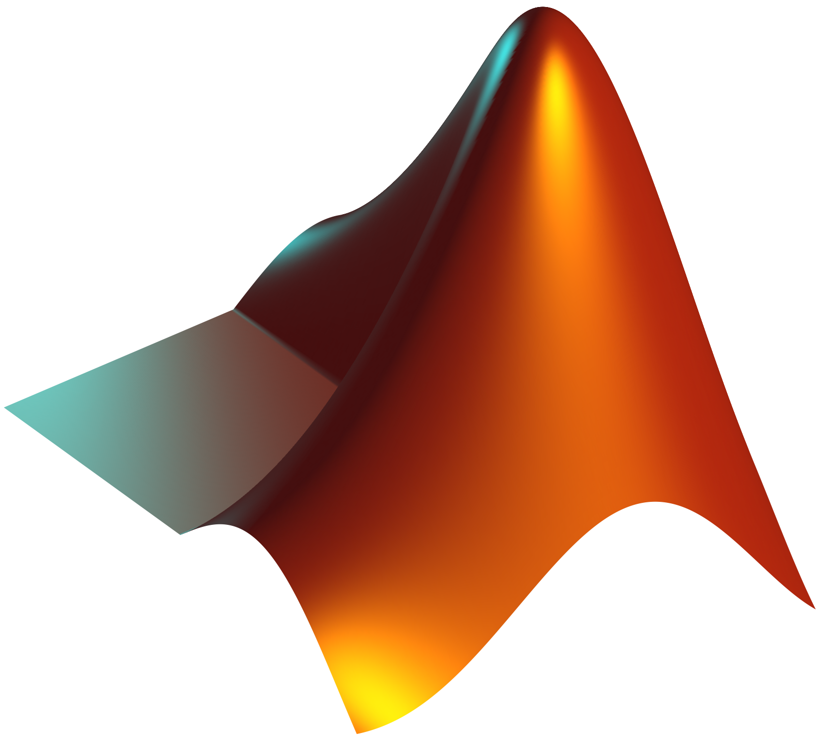
- ☐ Task: 4.1 **Signal Smoothing**. [1 Mark]
  - Submit the completed function `SmoothSignalPass` to MATLAB *Grader*.
- ☐ Task: 4.2 **Signal Smoothing n Times**. [2 Marks]
  - Submit the completed function `SignalPass` to MATLAB *Grader*.
- ☐ Task: 4.3 **Plotting Signals**. [2 Marks]
  - Use MATLAB *Publish* to publish the completed script to a single file.
  - Submit the published `.html` or `.pdf` file to *Canvas*.
- ☐ Task: 4.4 **What Colour Is The Sky?** [1 Mark]
  - Submit the completed function `IsPixelBlue` to MATLAB *Grader*.
- ☐ Task: 4.5 **Image Filtering**. [2 Marks]
  - Submit the completed function `FilterImage` to MATLAB *Grader*.
- ☐ Task: 4.6 **Methods For Surface Plotting** [2 Marks]
  - Use MATLAB *Publish* to publish the debugged script `PlotSurfaces.m` to a single file.
  - Submit the published `.html` or `.pdf` file to *Canvas*.

ENGGEN 131

Introduction to Engineering Computation and Software Development

LAB 5

Strings and Files



The Department of Engineering Science
The University of Auckland

Strings

Recall that there are several useful functions for comparing strings.

`strcmp`	Compare two strings
`strncmp`	Compare the first *n* positions of two strings
`strcmpi`	Compare two strings ignoring the 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 these commands are evaluated as `true` or `false`. Ask a teaching assistant if you are unsure.

The function `strfind` is useful for searching a string for occurrences of a shorter string and returning that pattern's location(s). It returns an array listing the search pattern's location(s). The returned array will be empty if the function does **not** find the pattern.

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'))
```

Strings and User Input/Output

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

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

The function `sprintf` is very useful for creating a nicely formatted string of characters which you can then display with the function `disp`. It supports a wide range 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 function `sprintf` takes a format string which usually includes one or more characters `%` followed by optional control flags and a specifier. Other arguments passed into the function `sprintf` are inserted in order in place of the specifiers, using the specified format.

Try typing in the following functions, leaving off the semi-colon so 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:  %.2f', pi)
>> sprintf('Pi with 8dp and scientific notation:  %.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 feature 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 5.1: Formatting Strings

Often at times, we would prefer string output to look "pretty", which requires the use of special string formatting functions like `sprintf` or `fprintf`.

The function `WeatherPredictionModel` is a "highly complicated" weather model. When given an input date, it will predict the weather conditions for that particular day. However, it does not display any of the output information to the user.

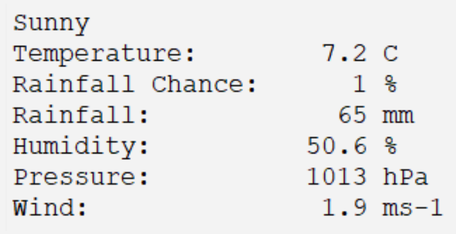
Input:
    Day of the month, as an integer.
    Month of the year, as an integer.
    Year, as an integer.
Output:
    Weather condition as a string (e.g. 'rainy', 'sunny', 'cloudy', etc.).
    Temperature in °C.
    Rainfall probability as a number between 0 and 1.
    Expected rainfall in m.
    Humidity as a number between 0 and 1.
    Pressure in Pa.
    Average wind speed in m s^{-1}.

If you look at a weather app (or equivalent), they generally display the information in a certain way. In particular:

- Temperature is shown as a whole number in °C.
- Rainfall probability as a percentage.
- Rainfall is shown in mm.
- Humidity is shown as a percentage.
- Pressure is shown in hPa.
- Wind speed is shown in km h^{-1}.

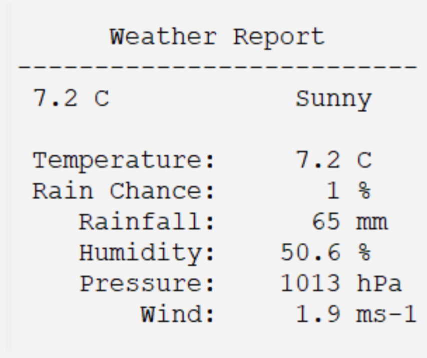
The function `WeatherPredictionModel` does not return these typical formats.

Write a script file that calls the function `WeatherPredictionModel`, and displays "pretty" formatted strings that show the formatted output of the function. Figure 5.1, Figure 5.2 and Figure 5.3 are some possible "pretty" outputs. Notice how in Figure 5.1, Figure 5.2 and Figure 5.3, the columns of text and numbers line up together? These outputs are just an example of what people may consider "pretty".



```
Sunny
Temperature:        7.2 C
Rainfall Chance:      1 %
Rainfall:            65 mm
Humidity:          50.6 %
Pressure:          1013 hPa
Wind:               1.9 ms-1
```

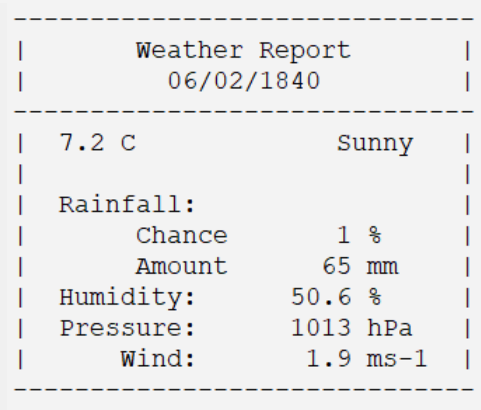
Figure 5.1: An example "pretty" output.



```
      Weather Report
--------------------------
 7.2 C            Sunny

 Temperature:     7.2 C
 Rain Chance:       1 %
    Rainfall:      65 mm
    Humidity:    50.6 %
    Pressure:    1013 hPa
        Wind:     1.9 ms-1
```

Figure 5.2: An example "pretty" output.



```
------------------------------
|       Weather Report       |
|         06/02/1840         |
------------------------------
|  7.2 C             Sunny   |
|                            |
|  Rainfall:                 |
|       Chance       1 %     |
|       Amount      65 mm    |
|  Humidity:      50.6 %     |
|  Pressure:      1013 hPa   |
|      Wind:       1.9 ms-1  |
------------------------------
```

Figure 5.3: An example "pretty" output.

The three examples in Figure 5.1, Figure 5.2 and Figure 5.3 would all be an acceptable submission.

You must display the output with reasonably formatted values (as suggested). However, you may name each output appropriately with whatever embellishments you wish. The important aspect is that the data is presented "pretty" format using the formatting functions like `sprintf` or `fprintf`, and it will work regardless of which input date is used.

You should not be editing the function `WeatherPredictionModel` to change the output format. You should be using the output as given and processing it in your script. Often, you may come across situations where you need to use functions that you can not edit. Treat this task as one of those situations.

Edit the completed script with a unique date, then use the MATLAB *Publish* feature to publish the code, comments and outputs into a single file. Submit the `.html` or `.pdf` file to *Canvas*.

Note: You can find a MATLAB *Publish* guide on *Canvas*.

✍ Task 5.2: Processing Strings

We have provided you with a file `DNAString.mat`, which contains two variables `dnaStringLong` and `dnaStringShort`. These variables are a long string and short string, respectively, of DNA sequences gathered from a newly discovered species.

To analyse the genetic nature of this new species, you must compare its DNA string with a few known DNA strings. These strings are:

Sequence 1a:	`'AGTCACT'`
Sequence 1b:	`'AcgT'`
Sequence 2a:	`'TACTga'`

The underlined letters are case insensitive. For example, an occurrence of `'taCTga'` would be a match for Sequence 2a, but `'tACTgA'` would not. Write a function called `GetDnaMatches` that searches through the DNA strings and finds the number of matches to each sequence.

Input:
A character array containing a DNA string, e.g., `dnaStringLong` or `dnaStringShort`.

Output:
A 1 by 3 array of the number of matches to each Sequence 1a, 1b and 2a respectively.

We have also provided you with the function `DnaOutputFormatter`. It accepts one array (three elements) of values indicating the number of DNA string matches (i.e., the output of the function `GetDnaMatches`).

The output obtained from the function `GetDnaMatches(dnaStringShort)` combined with the function `DnaOutputFormatter` should look something like:

```
        Sequence 1     |     Sequence 2
  a:                 7 | a:                 2
  b:                19 |
---------------------------
```

```
    Total:            26 | Total:            2
```

Submit the completed function `GetDnaMatches` to MATLAB *Grader*.

File Input/Output

MATLAB provides an *Import Wizard*, which you can use to import data files. Now try using the button Home ⟫ Import Data to import the data contained in the file `gasData.txt`.

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 functions to read the contents of a file.

Instead of using the *Import Wizard*, we will read in the contents of the file `gasData.txt` using file functions. See the lecture on File Input/Output for a reminder on how to do this.

Remember to open the file for reading using `fopen` before reading from it. You should also close your file using `fclose` when finished.

✍ Task 5.3: Reading From A File

Measurement data obtained from scientific experiments are often straight exported to files such as `.txt` and `.csv`. These raw measurements can be read into programs like MATLAB and then post-processed to obtain the values we want. We can save these values by then writing to files.

The file named `gasData.txt` contains experimental data on the volume (V) in m^3 and temperature (T) in °C of 2 moles (n) in mol of a particular gas.

Write a function called `ReadGasData` that reads data from a gas data file and returns the volume and temperature data as separate array variables.

Input:
  A filename string of the gas data file.
  The first number of rows `m` of data to read (excluding the header).
Output:
  A 1 by `m` array of gas volume in m^3.
  A 1 by `m` array of gas temperature °C.

For example:

```
    [Vol, Temp] = ReadGasData('gasData.txt', 1000);
```

reads the first 1000 rows of data from the file `gasData.txt`.

Submit the completed function `ReadGasData` to MATLAB *Grader*.

✍ Task 5.4: Writing To A File

Continuing from Task: 5.3, the file `gasData.txt` contains experimental data on the volume (V) in m^3 and temperature (T) in °C of 2 moles (n) in mol of a particular gas. You will need to be convert temperature to K by adding 273.15 to the temperature in °C.

These measurements can be used to calculate the pressure (P) Pa of the gas (which is the data that is required in this situation), using the universal gas constant (R) of $8.314\,\mathrm{m^3\,Pa\,K^{-1}\,mol^{-1}}$ and the relationship:

$$PV = nRT\,.$$

Write a function called `WriteGasData` that writes an array of pressure data to a specific file and returns the same array of pressure data as output.

Input:
 A filename string of the output gas data file.
 A 1 by `m` array of gas volume in m^3.
 A 1 by `m` array of gas temperature °C.
Output:
 A 1 by **m** array of gas pressure in Pa.

For example:

```
Pressure = WriteGasData('pressureData.txt', Vol, Temp);
```

calculates the pressure, then writes the pressure data (with a header line) to the file `pressureData.txt` and returns the pressure data (that could be used later on). Your file should look something like:

```
Pressure (Pa)
576.9813
581.7608
585.1694
581.2526
...
...
...
923.1845
923.1845
927.1167
925.6272
928.9082
```

Submit the completed function `WriteGasData` to MATLAB *Grader*.

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 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 common for a cell array to be created.

Tip: Cell Arrays and Curly Braces

If you use square brackets to try and create a cell array, you will not create a cell array. Instead, you will 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 1 by 1 cell array back rather than a string.

The bracket mix-up can be confusing if you do not spot your error.

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

Try saving running the following code:

```
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
```

✍ Task 5.5: Debugging Cell Arrays

The function `FindPhoneNumber` attempts to find phone numbers associated with a given name (or all matching names).

Open this function in the MATLAB *Editor*, read the header comment to understand what the function should do and then fix any bugs, so it runs correctly.

Remember, you can use the debugger to step through line by line to see how each line of code changes the variables in the workspace.

Test your function by finishing the test script `TestFindPhoneNumber`.

Submit the contents of the debugged function `FindPhoneNumber` to MATLAB *Grader*.

Study: Check File Similarity

You are interested in catching people copying each other's project files. Write a function called `CheckSimilarity` that will take as input two file names and then check to see how similar the first file is to the second file. Your function will do this by counting how many corresponding lines are identical, i.e. if line 2 of `file1.txt` matches line 2 of `file1.txt`, then the count of identical lines is increased. Note that for simplicity, we will only count matches for corresponding lines, so if line 2 from `file1.txt` matches line 3 of `file2.txt`, the count of identical lines is **not** increased. It will also calculate a percentage similarity figure using the formula:

$$\frac{\text{Number of Identical Lines}}{\text{Number of Lines in Shortest File}} * 100$$

Input:
 First filename of the file to compare.
 Second filename of the file to compare.
Output:
 The number of identical lines.
 The percentage similarity between the two files (a value between 0 and 100).

Study: Check Folder Similarity

You are interested in catching people copying each other's project files. Write a script file that will check to see how similar a given file is to all other files in the same directory. You should use the function `CheckSimilarity`, which given two file names, will return the number of identical lines and a similarity score.

You will need the function `GetFileNames`. `GetFileNames` returns a cell array containing a list of all filenames in the current MATLAB directory.

Your script file should ask the user to enter a file name to check, and then it should write out a file called `similarityReport.txt`.

`similarityReport.txt` should contain a summary of how similar each file in the directory is to the entered file. It should use the format shown below:

```
Similarity Check for file:  file1.txt

Filename    Lines the same    %similar
file1.txt               10       100.0
file2.txt                5        50.0
file3.txt                5       100.0
file4.txt                1         5.0
```

✍ Task Submission Summary

☐ Task: 5.1 **Formatting Strings**. [2 Marks]

- Use MATLAB *Publish* to publish the completed script to a single file.
- Submit the published `.html` or `.pdf` file to *Canvas*.

☐ Task: 5.2 **Processing Strings**. [2 Marks]

- Submit the completed function `GetDnaMatches` to MATLAB *Grader*.

☐ Task: 5.3 **Reading From A File**. [2 Marks]

- Submit the completed function `ReadGasData` to MATLAB *Grader*.

☐ Task: 5.4 **Writing To A File**. [2 Marks]

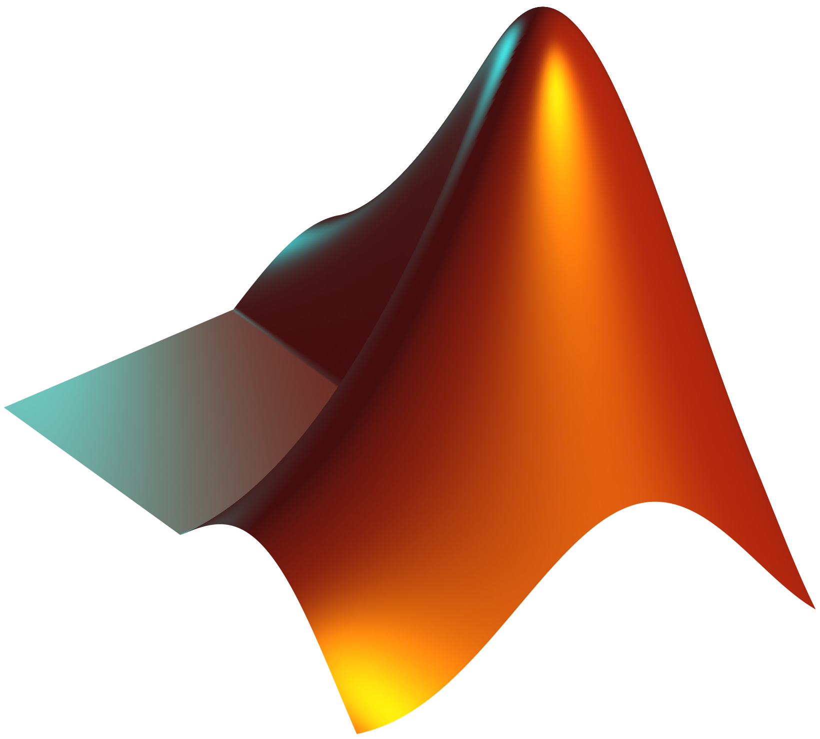
- Submit the completed function `WriteGasData` to MATLAB *Grader*.

☐ Task: 5.5 **Debugging Cell Arrays**. [2 Marks]

- Submit the debugged function `FindPhoneNumber` to MATLAB *Grader*.

ENGGEN 131
Introduction to Engineering Computation and Software Development

LAB 6
Linear Equations and Differential Equations



The Department of Engineering Science
The University of Auckland

Linear Equations

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

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

$$\begin{aligned} y &= \frac{1}{2}x + 1\,, \\ y &= -x + 4\,. \end{aligned}$$

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

$$\begin{aligned} -x_1 + 2 \cdot x_2 &= 2\,, \\ x_1 + x_2 &= 4\,. \end{aligned}$$

The system of linear equations above 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:

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} -1 & 2 \\ 1 & 1 \end{bmatrix}, \\ \mathbf{x} &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \\ \mathbf{b} &= \begin{bmatrix} 2 \\ 4 \end{bmatrix}. \end{aligned}$$

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

Recall that if $\mathbf{A}$ has an inverse, you can multiply both sides of the equation by the inverse of $\mathbf{A}$ to get:

That is, the solution is the inverse of $\mathbf{A}$ multiplied by $\mathbf{b}$ is:

$$\begin{aligned} \mathbf{A}^{-1}\mathbf{Ax} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{Ix} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b}\,. \end{aligned}$$

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

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

Once the matrix **A** and the column vector **b** have been assigned values, solving the system took one line of code. MATLAB also supplies the left division method (character \):

```
x = A\b
```

which is equivalent to Gaussian elimination and generally gives a more accurate solution.

Stop: Have we Solved Ax=b?

Verify that you get similar solution values using the inverse or backslash method.

To check the accuracy of your solution, type `A*x` and verify that we get `b` (or close to it).

Both of these methods **only** work if **A** has an inverse, which can be checked beforehand. 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 function `det`. Find the determinant of our matrix `A` by typing:

```
det(A)
```

Verify that it is nonzero. Recall that if your determinant is zero, the system of equations has no solution. Such a matrix is called a singular matrix.

If your matrix is singular, this also means that the function `inv` would not generate a valid inverse, as the inverse does not exist.

Try the following and see what happens:

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

Is this what you expected? Check the value of `det(A)`. Does `A` have an inverse?

✍ Task 6.1: Solving A System Of Linear Equations

Consider the following electrical circuit in Figure 6.1, where we are interested in determining the current flowing along each branch.

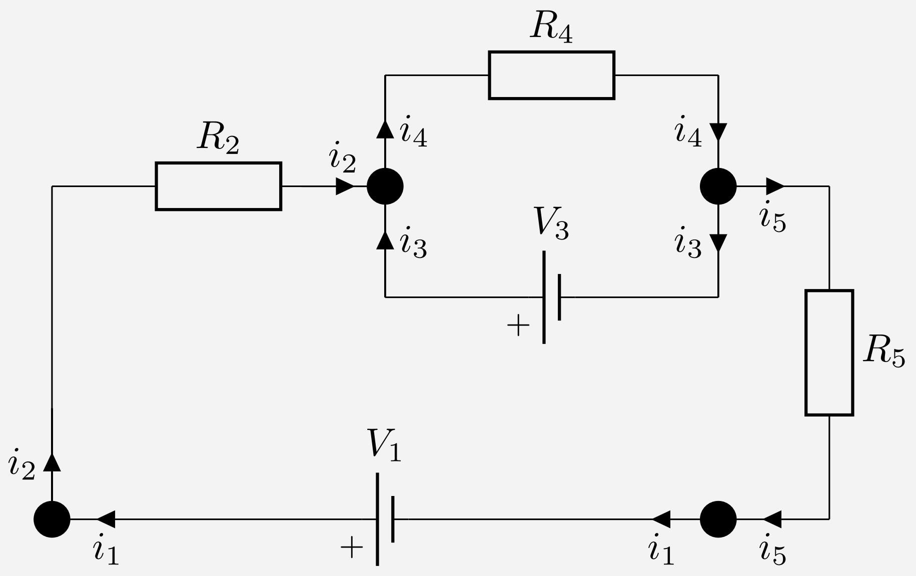


Figure 6.1: Electrical Circuit System.

Applying Kirchoff's current law at each of the nodes (black dots), and using the current directions indicated by the arrows, we get the following system of equations:

$$\begin{aligned} i_1 - i_2 &= 0 \\ i_2 + i_3 - i_4 &= 0 \\ i_4 - i_3 - i_5 &= 0 \\ i_5 - i_1 &= 0 \end{aligned}$$

Applying Kirchoff's voltage law around the three loops in the circuit, we get the equations:

$$\begin{aligned} V_1 - i_2R_2 - i_4R_4 - i_5R_5 &= 0 \\ V_1 - i_2R_2 - V_3 - i_5R_5 &= 0 \\ V_3 - i_4R_4 &= 0 \end{aligned}$$

This gives 7 equations for 5 unknowns (the currents). This is two more equations than we need. From this point on, we **drop the final equation from each group**.

With a little work, we can rewrite the five remaining equations as follows

$$
\begin{array}{rrrrrrrrrrl}
+ & \mathbf{1}\cdot i_1 & - & \mathbf{1}\cdot i_2 & + & \mathbf{0}\cdot i_3 & + & \mathbf{0}\cdot i_4 & + & \mathbf{0}\cdot i_5 & = \mathbf{0} \\
+ & \mathbf{0}\cdot i_1 & + & \mathbf{1}\cdot i_2 & + & \mathbf{1}\cdot i_3 & - & \mathbf{1}\cdot i_4 & + & \mathbf{0}\cdot i_5 & = \mathbf{0} \\
+ & \mathbf{0}\cdot i_1 & + & \mathbf{0}\cdot i_2 & - & \mathbf{1}\cdot i_3 & + & \mathbf{1}\cdot i_4 & - & \mathbf{1}\cdot i_5 & = \mathbf{0} \\
+ & \mathbf{0}\cdot i_1 & + & \mathbf{R_2}\cdot i_2 & + & \mathbf{0}\cdot i_3 & + & \mathbf{R_4}\cdot i_4 & + & \mathbf{R_5}\cdot i_5 & = \mathbf{V_1} \\
+ & \mathbf{0}\cdot i_1 & + & \mathbf{R_2}\cdot i_2 & + & \mathbf{0}\cdot i_3 & + & \mathbf{0}\cdot i_4 & + & \mathbf{R_5}\cdot i_5 & = \mathbf{V_1} - \mathbf{V_3}
\end{array}
$$

Write a function called `ElectricalCircuitCurrent` that determines the current flowing along each branch of the electrical circuit displayed in Figure 6.1.

Input:
An array of circuit parameters `[V1 R2 V3 R4 R5]` in this order.
Output:
An array of currents in each branch `[I1 I2 I3 I4 I5]` in this order.

The function `ElectricalCircuitCurrent` should encode the equations in matrix form, then solve the equations to find the current through each branch.

Test your function by adjusting the provided test script `TestElectricalCircuitCurrent.m`, for example changing the parameters to:

$V_1 = 10\,\mathrm{V}$, $R_2 = 3\,\Omega$, $V_3 = 4\,\mathrm{V}$, $R_4 = 4\,\Omega$, $R_5 = 3\,\Omega$

and

$V_1 = 2.6\,\mathrm{V}$, $R_2 = 2.25\,\Omega$, $V_3 = 7.7\,\mathrm{V}$, $R_4 = 2.25\,\Omega$, $R_5 = 7.3\,\Omega$

Submit the completed function `ElectricalCircuitCurrent` to MATLAB *Grader*.

✍ Task 6.2: Vector Algebra

The following algorithm calculates increasingly accurate approximations to π using vector algebra (where $\times$ is the vector cross product and $|x|$ is the magnitude of x).

```
Set $a$ to $\begin{bmatrix}0\\1\\0\end{bmatrix}$ and $b$ to $\begin{bmatrix}1\\0\\0\end{bmatrix}$
    Set $b$ to $\frac{1}{2}(a+b)$
    Normalise $b$: b = $\frac{b}{|b|}$
    Calculate an approximation of pi using: $\pi = 2^{n+1}|a \times b|$
end
```

Write a script file that asks the user how many iterations n to approximate π using the algorithm described above. The script should print each estimate of π as it is obtained.

The following is an example output of your script with $n = 7$.

```
Value of pi for iteration 1 is 2.82842712
Value of pi for iteration 2 is 3.06146746
Value of pi for iteration 3 is 3.12144515
Value of pi for iteration 4 is 3.13654849
Value of pi for iteration 5 is 3.14033116
Value of pi for iteration 6 is 3.14127725
Value of pi for iteration 7 is 3.14151380
```

Comment out the line with the function `input`. Then edit your script with $n = 10$ and publish the code and output as a `.html` file. Ensure your code, comments and outputs are all included in the same file. Then, submit the `.html` or `.pdf` file to Canvas.

Edit your script with $n = 10$, then use the MATLAB *Publish* feature to publish the code, comments and outputs into a single file. Submit the `.html` or `.pdf` file to *Canvas.*

Note: You can find a MATLAB *Publish* guide on*Canvas.*

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 are written in the following form:

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

that is, the derivative can be written as some function of the dependent 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:

$$\frac{dy}{dt} = \cos(\omega t)\,,$$

with initial condition $y(0) = 0$, it 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 second and an angular frequency ω of 2π by using the following script (available from Canvas):

```
% 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 (i.e., start time and finish time stored in an array).
- An initial value at the start time (i.e, initial condition).

First, we need to write a derivative function that calculates the derivative for any given values of t and y. The solver will call this function many times to find our numerical solution. We can choose any valid function name for our derivative function, but as always, giving the function a meaningful name is a good idea.

Note that the solvers require the derivative function to take **both** the independent and dependent variables as inputs (even if they 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, independent variable representing time
%          y, dependent variable representing ...
   displacement
% output: dydt, derivative of y with respect to t

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

end
```

Download this function from *Canvas* 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     y = 0
t = 0.25  y = 1
t = 0.5   y = 0
t = 1     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 array for time range to solve over
% only need to specify 2 values (start and finish)
tInterval = [0 1];

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

% solve our ODE, the solver expects three arguments
% in the following order:
% - the derivative function name (in quotes)
% - time interval (2 element row vector)
% - start time (value)
[t,y] = ode45('SinusoidDerivative', tInterval, yinit);
plot(t,y)
```

Download this script file from *Canvas* and run it. Compare your plot with the analytical solution.

A Rocket-Propelled Sled

The following problem is adapted from an example on page 535 of "Introduction to MATLAB 7 for Engineer", William J. Palm III. It forms the basis for Task: 6.3, Task: 6.4 and Task: 6.5.

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

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

where m is the sled mass (kg), and c is the air resistance coefficient ($\mathrm{N\,s\,m^{-1}}$). We know the sled's initial velocity $v(0)$ and want to find velocity $v(t)$.

We want to plot the motion of the rocket-propelled sled described above using a few different methods, including an analytic solution (Task: 6.3) and a numerical solution (Task: 6.3 + Task: 6.3).

Our sled has a mass of 1000 kg, an initial velocity of $5\,\mathrm{m\,s^{-1}}$ and an air resistance coefficient of $500\,\mathrm{N\,s\,m^{-1}}$; we want to plot the rocket-propelled sled's velocity over the time interval $0 \leq t \leq 10$ with the different solution methods.

Finding Velocity Analytically

Using our MM1 knowledge, we know we can solve this ODE using the separation of variables.

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

Step 1: Separate the variables to different sides.

$$\frac{m}{v}dv = -cdt$$

Step 2: Integrate.

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

Note that k is constant.
Step 3: Make v the subject.

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

Step 4: Rework the constant.

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

where $A = e^{\frac{k}{m}}$ **Step 5**: Evaluate the constant.

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

Step 6: Answer the question.

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

✍ Task 6.3: Analytic Solution

Create a function called `SledAnalyticSolution` that calculates the analytic solution for the rocket-propelled sled velocity.

Input:

An array of time t.
The initial velocity $v(0)$.

Output:

An array of the calculated velocity v.

Slightly modify the script `PlotSledSolutions.m` to test and view the output of the function `SledAnalyticSolution`.

Submit the completed function `SledAnalyticSolution` to MATLAB *Grader*.

✍ Task 6.4: Velocity Derivative

Write a function called `SledVelocityDerivative` that returns the sled's velocity derivative $\frac{dv}{dt}$. Similar to that of the function `SinusoidDerivative` defined in the example above.

Input:

The time t.

The velocity v.

Output:

The derivative $\frac{dv}{dt}$.

Submit the completed `SledVelocityDerivative` to MATLAB *Grader*.

✍ Task 6.5: ODE45 to Joy

Write a function called `SledNumericalSolutionOde45` that calculates the numerical solution to the sled ODE using MATLAB's function `ode45`.

Input:

An array of the time span (i.e., just the start and end time).

The initial velocity $v(0)$.

Output:

An array of time values t.

An array of the calculated velocity v.

It should call the function `ode45` and the function `SledVelocityDerivative` to solve the sled ODE numerically.

Slightly modify the script `PlotSledSolutions.m` to test and view the output of the function `SledNumericalSolutionOde45`.

Submit the completed function `SledNumericalSolutionOde45` to MATLAB *Grader*.

📖 Study: Max V1

Consider Task: 6.1, in particular the circuit in Figure 6.1. How much we can increase the voltage at V_1 in the positive direction before melting a wire? The wire will melt when more than 5 A of current passes through any of them.

Write a function called `MaxV1` that finds maximum voltage V_1 before the maximum current flowing (in either direction) through any of the wires exceeds 5 A.

Input:
An array of circuit parameters `[V1 R2 V3 R4 R5]`, in this order.
Output:
The maximum voltage V_1.

It should use an appropriate type of loop with a step size of 0.1 V that increases the voltage V_1 in the positive direction until the maximum current flowing through any of the wires exceeds 5 A. Remember, you need to find the maximum voltage **before** the wire melts. Recall that both voltage and current can also be negative, indicating that the directions initially assumed are incorrect and the actual direction is the opposite of the indicated directions.

Your function should use the function `ElectricalCircuitCurrent` from Task: 6.1. You can assume that the first solution will not result in any current already exceeding 5 A.

For example, the expected maximum voltage V_1 for the following parameters is is 33.9 V.

$V_1 = 10\,\text{V}$, $R_2 = 3\,\Omega$, $V_3 = 4\,\text{V}$, $R_4 = 4\,\Omega$, $R_5 = 3\,\Omega$

📖 Study: Using Euler's Method (Debugging)

Another way we can solve ODEs of the form:

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

is by using Euler's method. Recall that in Euler's method, subsequent y values are given by:

$$y_{n+1} = y_n + hf(y_n, t_n)$$

where h is the step size.

If you need more information about Euler's method, see your MM1 notes or Wikipedia.

We have provided you with a function `SledNumericalSolutionEuler` that should solve the ODE problem from Task: 6.3, Task: 6.4 and Task: 6.5 but by using Euler's method and the function `feval`. However, the function `SledNumericalSolutionEuler` has bugs.

Note the function `SledNumericalSolutionEuler` references the function `SledVelocityDerivative` from Task 6.4.

Open the function `SledNumericalSolutionEuler` in the MATLAB *Editor*, read the comments to understand what the function should do and then fix any bugs so it runs correctly.

Remember, you can use the debugger to step through line by line to see how each line of code changes the variables in the workspace.

Slightly modify the script `PlotSledSolutions.m` to test and view the output of the function `SledNumericalSolutionEuler`.

Study: To the Moon

The rocket-propelled sled is now sent off a ramp located on a cliff with an elevation h above sea level. A few seconds after the rocket-propelled sled leaves the ramp, the rocket catches on fire. The fire causes malfunctions in the onboard systems. Before hitting the ground and exploding, the onboard computers had continuously sent updates on the sled's flight path elevation (relative to the cliff top elevation) as a mathematical function $f(x)$, which is a function of distance x from the ramp.

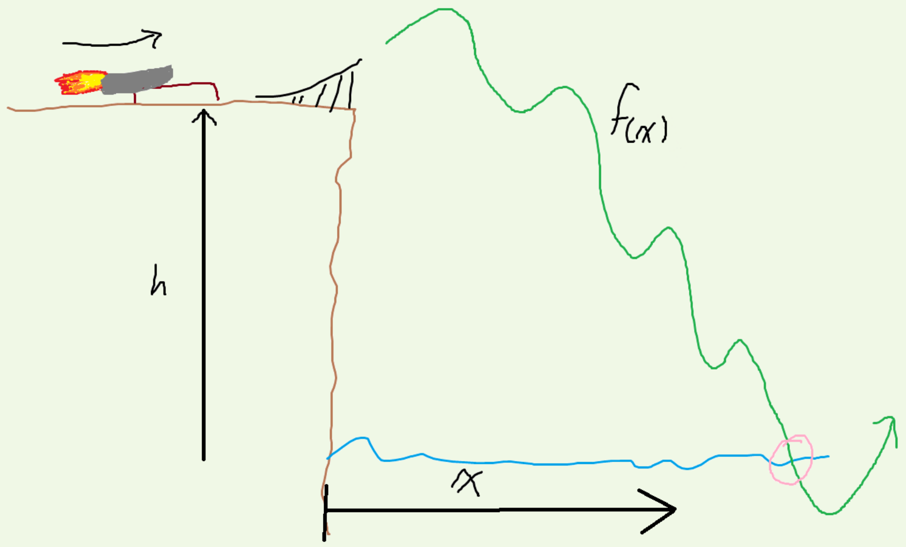


Figure 6.2: Rocket-Propelled Sled Path

To determine where the crash site location is, we must determine when the flight path elevation is first $-h$. That is, we want to find the first value of x such that:

$$\begin{aligned} -h &= f(x) \\ 0 &= f(x) + h \end{aligned}$$

We have provided you with a function `SledCrashSiteLocation` that should be able to determine the crash site location x using the function `fzero`. However, the function `SledCrashSiteLocation` has bugs.

Open the function `SledCrashSiteLocation` in the MATLAB *Editor*, read the comments to understand what the function should do and then fix any bugs so that it runs correctly.

Remember, you can use the debugger to step through line by line to see how each line of code changes the variables in the workspace.

Use the script `TestSledCrashSiteLocation.m` to test and view the output of the function `SledCrashSiteLocation`.

Debug the function `SledCrashSiteLocation` so that it correctly calculates the crash site.

✍ Task Submission Summary

☐ Task: 6.1 **Solving A System Of Linear Equations**. [2 Marks]

- Submit the completed function `ElectricalCircuitCurrent` to MATLAB *Grader*.

☐ Task: 6.2 **Vector Algebra**. [2 Marks]

- Use MATLAB *Publish* to publish the completed script to a single file.
- Submit the published `.html` or `.pdf` file to *Canvas*.

☐ Task: 6.3 **Analytic Solution**. [2 Marks]

- Submit the completed function `SledAnalyticSolution` to MATLAB *Grader*.

☐ Task: 6.4 **Velocity Derivative**. [2 Marks]

- Submit the completed function `SledVelocityDerivative` to MATLAB *Grader*.

☐ Task: 6.5 **ODE45 to Joy**. [2 Marks]

- Submit the debugged function `SledNumericalSolutionOde45` to MATLAB *Grader*.