If you haven't completed it already, please start with Exercise 0.1

Exercise 0.1 – Explore the interactive workspace

Work through the following short simple exercises to give you a taster of some of the MATLAB commands and to get you used to the interactive workspace. This exercise will help you when you come to do the examples on which this course is assessed.

Start with some basic commands.

1. MATLAB can do arithmetic like a standard calculator. Notice the precedence of the operators. In the Command Window do the following:

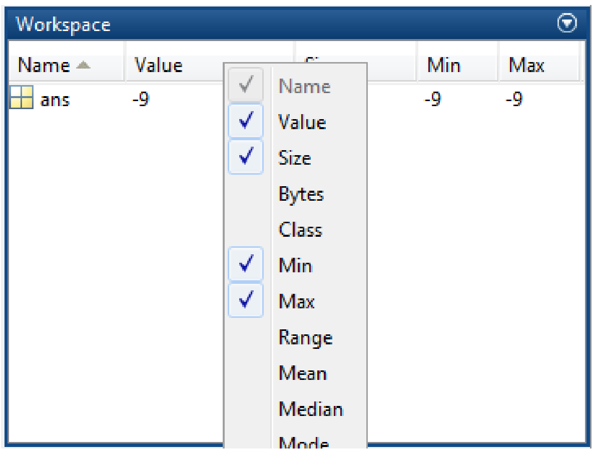
   a. `5+12`

   b. `5*12`

   c. `5*sin(30*pi/180 )`

   d. `3^2 - 6*3`

   Notice that the result is assigned to the variable name *ans*.
   Look in the Workspace. Right-clicking on the column headings allows you to select which properties of the variable you want displayed.



   Some are not relevant for a single value but they provide a quick way to view certain properties, especially when looking at large arrays of data.

2. Clear the Command Window and Workspace using `clc` and `clear`. Once you have a variable defined, it can be used in further equations, try:

   a. `5+12`

   b. `ans + 5`

   c. `ans + 1`

   Note that `ans` takes on the new value.
   Remember that adding a semi-colon after the command prevents the outcome displaying in the Command Window.

3. Now undertake some calculations using variables

a) Clear your workspace.

b) Create some new variables for distance travelled and time taken. To check if your variable name is valid use the isvarname command followed by your variable name. It will return a 1 if valid. Note: this will not tell you if the variable name you have chosen is also a function name. Use `which` *`variable_name`* to check if your variable name is used, if it returns a file path, then a function or variable already exists. For example, you may find distance is already a function name however distance_travelled is not.

c) Set distance_travelled to 323 and time to 38 (we'll assume in metres and seconds respectively).

d) Now type the formula to work out the speed.

e) Once the calculation is done, see that all three variables are now in the workspace.

f) Add 5m/s to the speed to set a new value for speed.

g) Now calculate the new distance that would be covered in the same time.

4. Saving the Workspace and loading a previous one.

a) Set the current folder to folder you wish to use in My Documents, ideally you should create a folder called MATLAB.

b) You'll see whatever is in your chosen directory in the Current Folder window.

c) You should by now have 3 variables listed in your workspace, in the Command Window type `save`. You should notice a file named matlab.mat has appeared in the Current Folder. You can use `save filename.mat` to change from the default matlab.mat filename. We will look at this and related file functions in more detail in lab 3.

d) This file contains all the data from your Workspace in a MATLAB specific format. If you wish to share the data, it is more convenient to save the data in text format, to do this use:

   a. `save filename.txt -ascii`

   where filename is the name of the file you want to save. If you are saving more than one variable, you should use the .mat format.

e) Use `clear` to empty your workspace.

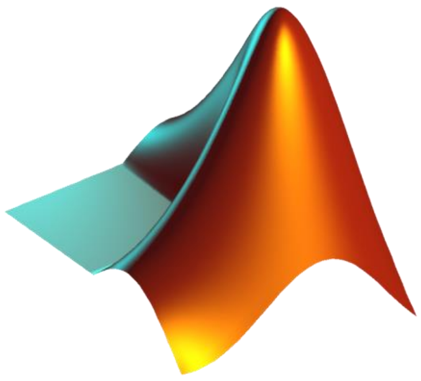
f) Now type `load matlab.mat`. Your variables will have reloaded into Workspace.

g) Try it saving as a .txt file and see what happens when you load the data back using `load matlab.txt -ascii`

We'll cover more about file I/O in Lab 3.

End of exercise 0.1

Exercise 1.1 – Working with matrices in MATLAB



Right – let's try out some matrix stuff, since that is one of the things MATLAB is really good at. This task will introduce the methods and some of the functions used to manipulate matrices in MATLAB. You may like to refer to your MATLAB_quick_guide.pdf for help (found in the Support section of the VLE). Most steps in the task will have a reference to the MATLAB Quick Guide e.g. (QG 2.2) means the action described can be found in section 2.2 of the quick guide.
NOTE: If example code is given in the practical handouts, it often needs to be changed to suit the task you are undertaking. If text in the example code is red, it means you need to edit that text.

1. Let's start with assembling a simple 3x3 matrix. Assign this to a variable (e.g. myarray = ….)

   $$\begin{bmatrix} 2 & 9 & 4 \\ 7 & 7 & 3 \\ 6 & 1 & 8 \end{bmatrix}$$

   If the middle number were a 5 this would be known as a magic square, that is all the diagonals, rows and columns add up to the same number. The first thing is to enter the array. Remember when inputting a matrix, spaces or commas separate columns and semi-colons or the return key separate rows. (QG 1.4)

2. Now time for some manipulations.
   a) Use the size() function to assign the size of the matrix to a variable called s.(QG 3)

   b) Assign the middle number in the matrix to be a variable called middle. Remember this can be obtained using a single or double index. For a double index your expression will look something like this:
   ```
   desired_value = myarray(i,j)
   ```
   where i and j are the row and column index respectively. (QG 2.2)

   c) Assign the values in the middle row to be a vector middle_row. Hint: Use the colon operator to get all the values in either a row or column. (QG 2.2)

   d) Assign the values in the middle column to be a vector middle_column. (QG 2.2)

   e) Now for a challenge! See if you can change the middle number of the matrix to the number 5. (QG 2.4) Your expression is going to look something like:
   ```
   myarray(row number, column number) = some_new_value
   ```

   f) Now you have done that, use MATLAB to give you the sum of the rows. Type `help sum` and see if you can follow the explanation. See if you can find the sum of the columns too. (QG 3)

   g) Now remove the final row of the matrix, to do this use: `myarray(end,:)=[ ]`

3. We can now try some matrix multiplication. There are quite a few notes on this in the MATLAB users guide found in the resources section of the VLE although we will not be covering all the concepts this week. Let's start with a simple 2x2 matrix so that you can verify what is going on by hand. Define the following matrix in MATLAB (give it a sensible name – let's call it some_matrix):

$$\begin{bmatrix} 5 & 4 \\ 3 & 2 \end{bmatrix}$$

a) On a piece of paper try multiplying the matrix by itself and see what answer you get. This equation explains how to multiply an *m*x*p* by a *p*x*n* matrix to get a *m*x*n* matrix:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

for all *i* = 1,2,...,*m* and *j* = 1,2,..., *n* . This algorithm indicates that each element in the *i*th row of **A** is multiplied by the corresponding element in the *j*th column of **B**. The sum of the products is the element *cij*

b) Now in MATLAB try the following calculations:

```
first_matrix = some_matrix * some_matrix
second_matrix = some_matrix .* some_matrix
third_matrix = some_matrix .^2
fourth_matrix = some_matrix ^2
```

(notice that in two examples we are using the dot operator before the multiplication sign)
So what is the difference here?
It's very important to recognise if you are working on *element by element* operations, or *whole matrix* operations.

4. Now a short challenge that will set you in good stead for this week's self study. Programmatically do the following:

a) Create the following matrix:

$$x = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

b) Index the value in the 3rd row and 2nd column of *x* and assign it to a new variable *y*. (QG 2.2)

c) Replace the value in the 2nd row, 1st column of matrix *x* with *y*. (QG 2.4)

d) Index the middle column of matrix *x* and assign it to a variable *z*. (QG 2.2)

e) Transpose *z*. (QG 2.4)

f) Add row vector *z* to the bottom of x as an additional row to give the new matrix x. (QG 2.4)

g) Find the maximum value of z and assign a new variable *max_z*. (QG 3)

h) Multiply *x* by *max_z* to give the new matrix *x*.

i) Find the sum of all the values in the new *x* and assign it to the variable *answer*. (QG 3)

j) Ask a demonstrator to check your value for *answer*.

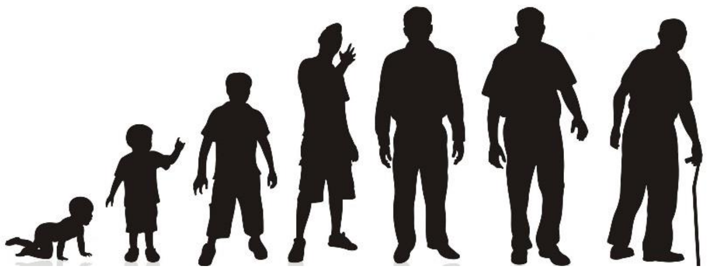
End of exercise 1.1

Exercise 1.2 – Working with m-files

This will introduce you to using m-files for developing programs in MATLAB and writing script, you'll look at an existing m-file and see how you can run the file and allow the user to interact with the program. You'll then write your own basic program in Editor and run it.

M-files are a way of collecting commands together so that you have a program you can rerun.
There are a few ways to launch a m-file, one is from within MATLAB, by selecting File -> New -> Script (or ctrlN as a keyboard shortcut). A second is using the edit command from the Command Window, e.g. edit myfirst.m, if the file does not exist, you will be asked if you want to create a new one. Edit alone will launch the editor window.

How Old Are You?



Looking at an existing m-file.
1.Logged onto the VLE go to the MECH1010 pages and from MATLAB: Labs and Self-study -> MATLAB Labs -> Lab 1 download howold.m into your working directory. The file howold.m will appear in the list of files in the current directory.

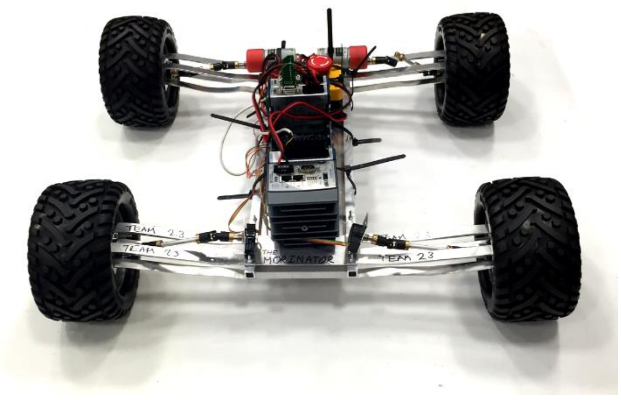
2. Type howold into the command window to run the m-file. Run it a few times trying different dates to check for errors (it is possible to get a result that doesn't make sense).

3. If you double click on the m-file in the Current Directory window, the editor will open with the script for howold.m. Notice the comments at the top of the file. Now return to Command Window and type `help howold`. What do you notice about what is displayed?

4. Look over the m-file, see if you can work out the code with the help of the comments? We'll cover the while loops and if/else conditional statements in next week's lab.

5. What other functions can you spot? You can right click over the function and select *Help on selection* or type help followed by the function in the Command Window.

Bouncing Buggies



Writing your own m-file.
You are now going to create your own basic m-file. Its function will be to load some vertical acceleration data from the 2nd Year Daring dash competition and determine which buggy generated the highest vertical acceleration in the chassis (i.e. had the worst suspension).

1. Logged onto the VLE go to the MECH1010 pages and from Labs and Self-study -> MATLAB Labs -> Lab 1 download **acceleration_data.mat** into your working directory. This contains the

vertical acceleration data for 16 buggies. Each column represents a different buggy and each row a different sample from the accelerometer.

2. Now open the editor window by clicking **New Script** in the MATLAB toolbar. The editor window will open.

3. At the start of any program you need to type a first comment that includes the name of the .m and a brief description of what the program will do. Your program will be called analyse_acceleration.m so your first comments should be `% analyse_acceleration.m` followed by the description. The "%" is used to identify the line as a comment and subsequent text will be covered green.

4. It is good practice at the start of any program to clear both the command window and the workspace. In your script on new lines following the comments add `clc` to clear the command window and `clear` to clear the workspace. We are now ready to start with our program. (QG 0)

5. The first thing we need to do is load in our acceleration data. To do this we use the load function so you need to type `load` followed by the filename containing the data, in this case `acceleration_data.mat`. You should be aware of the .mat file as a MATLAB specific file type that contains workspace variable data. (QG 11.1)

   Add a semi-colon to the end of the line to stop MATLAB from displaying all the data in the Command Window. Now try saving the code as analyse_acceleration.m using the save button in the toolbar to your working directory. Now press the Run button in the editor window.
   Look at the Workspace window, you should now see a variable *acceleration_data* with a size of 2766x16.

6. We now want to find the maximum absolute acceleration. To do this we can use the `abs` function. The input argument for abs can be our *acceleration_data*. Use the following line of code in your program:

```
abs_acceleration_data = abs(acceleration_data);
```

   This will create a new variable *abs_acceleration_data* that will contain the absolute values of the matrix *acceleration_data*. (QG 3)

7. We now wish to find the highest value. To do this we will use the `max` function. Type `help max` in the Command Window for help on how the function works. When max is applied to a matrix, it will return a row vector containing the maximum value in each column. We can then use the max function again to find the highest acceleration in that row vector. (QG 3)

   Max can also be used to find the index of where the maximum occurred. To do this we need to add an addition output argument when we call the function. Additional input arguments are extra variables we pass to functions when they are called and are added in the parentheses after the function name and are separated by a comma. Remember that MATLAB does not use zero indexing.

   Additional output arguments return extra outputs from a function when called and are added to the left of the equals sign and are surrounded by square brackets and separated by a comma.

   For example if x is a row vector `y = max(x)` will return just the maximum value of x whereas `[y,z] = max(x)` will return the maximum value of x as variable `y` and the index where it occurred as a variable `z`. We'll cover input and output arguments in more detail later in the course.

So in our code we want to first get the row vector of the peak acceleration for each buggy. To do this we only need one output argument so we can use:

```
buggy_max_acc = max(abs_acceleration_data);
```

8. Now we want to find the buggy with the highest acceleration so we can use the max function again but with two output arguments. To do this we use the square brackets to the left of the equal sign and have two arguments max_acc and buggy_index. Program this line yourself, it should search the row vector buggy_max_acc and return the highest acceleration and the index or buggy number. (QG 3)

9. We've now determined the peak acceleration (`max_acc`) and which buggy it was measured on (`buggy_index`). We will use the disp function to present this information to the user. In this case we want to display the text where n is buggy_index and x is max_acc. We use the disp function with a string array as the input argument, which is we will enclose the string in square brackets. Strings should be enclosed in apostrophes to define them as strings (they appear purple in the editor) and we cannot include a numeric variable in a string array. In order to include the numeric we need to convert it to a string, to do this we use the num2str function. Here's an example:

```
>> x = 5;
>> disp(['The value of x is ' num2str(x) ' which is nice. ']);
```

Will display in the Command window "The value of x is 5 which is nice." Place your own `disp` function in the code to display "Buggy *n* had the highest vertical acceleration of *x* m/s^2". (QG 12.2)

10. Once you have added this last line of code, save your file and run the code. You should see the line of text stating the buggy number and peak vertical acceleration.

You can now add comments to your code to explain what each line is doing using the '%' before each comment. Try typing `help analyse_acceleration` in the command window.

Bonus buggy exercise:
Now a challenge: Can you use what has been covered in this session to determine the RMS (Root Mean Square) acceleration of buggy 4. Include this at the end of your program and use the display function to show the RMS value of buggy 4 in the Command Window. You'll need to get a column vector for the 4th column in the acceleration_data matrix. RMS means root mean square. It is the square root of the mean of all the values squared. A quick internet search will give you more information.

End of exercise 1.2

If you finish, why not start looking at self study 1 which can be found on the VLE.