MPHY0020 Computing in Medicine - MATLAB Exercises 1

Exercise 1.1 – Writing a script and setting the path

In this first exercise, you will run MATLAB for the first time, write your first scripts, and configure MATLAB to ensure subsequent sessions are a little bit easier. Please follow the steps below:

1. Launch MATLAB.
2. Create a new script using the 'New Script' button in HOME tab or pressing Ctrl+n / CMD+n.
3. In this script, enter some simple expressions, such as

```
disp('This is my first script')
x = pi
y = sin(x)
```

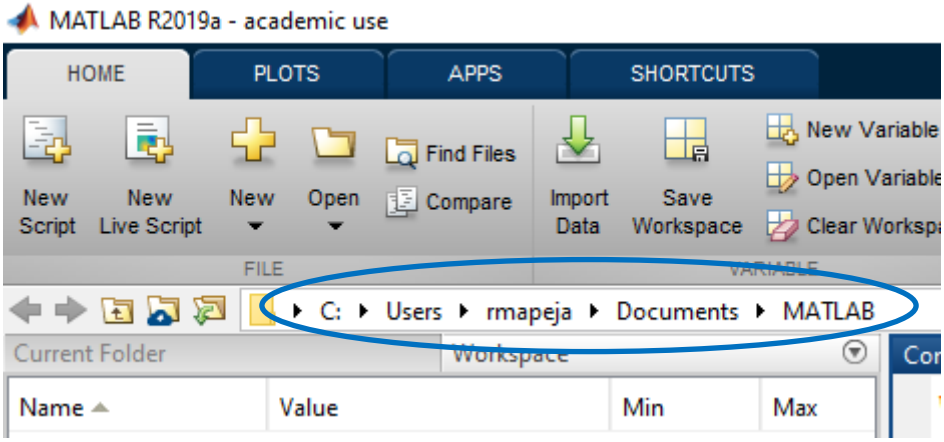
4. Save the script using the 'Save' button in the EDITOR tab of the MATLAB editor
   a. Create a new folder called *my_matlab_code* in a location that is convenient for you.
   b. Save the script into this folder giving it a name like `myFirstScript.m`

Now that you have written your very first script, it is time to execute it. As you have seen in the lectures, MATLAB scripts can be called from the command window by typing the script name.

5. Try running the script by typing `myFirstScript` into the command line, and press <ENTER>. Most likely you will see an error message saying:

```
??? Undefined function or variable 'myFirstScript'.
```

   This message means that MATLAB cannot locate your script, as it is not in the current directory or the MATLAB path.
6. Now check the MATLAB current folder. This is displayed in the top left corner of the MATLAB desktop (see image below). Most likely your current folder will not be the folder you saved your first script in – therefore MATLAB does not know where to find it! (Note that your current folder will differ from that in the image.)



7. To teach MATLAB where to look for your script(s), use the current folder browser (found in the top left of the main app, see image below) to change the current folder to your *my_matlab_code* folder.

8. Try running the script again typing `myFirstScript` into the command line. You should now get the following output:

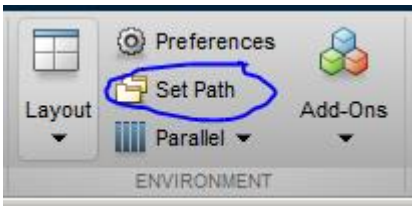
```
This is my first script
x =
    3.1416

y =
  1.2246e-016
```

Alternatively, you can also run a script by clicking the "Run" button (green play button) at the top of the Editor window - check this also works.

The steps above enable you to execute your first script from the command window. However, MATLAB does not remember your current directory by default, and you would have to take the previous steps each time you start MATLAB. Fortunately, there is another mechanism to tell MATLAB where to find scripts and functions: the `path`.

9. In order for MATLAB to remember your path, you can permanently add your *my_matlab_code* folder to the MATLAB path.
    a. Click on the "Set Path" button in the HOME tab.
    b. Click "Add Folder…"
    c. Select *my_matlab_code* folder and then click 'Select Folder'
    d. Now save the MATLAB path (check that your folder *my_matlab_code* is indeed listed!) using the Save button. This will try and save your current MATLAB path in a file called *pathdef.m* that is located in *matlab_install_dir/toolbox/local* (where *matlab_install_dir* is the folder that MATLAB was installed to).



NOTE: If you are working on UCL machines in the cluster rooms or Birkbeck, the file *pathdef.m* will be reset on every login. To make your path persistent across different sessions and different UCL computers, you need to save the pathdef.m file to you home directory. To do this, you will need to use the `savepath` command, i.e.

```
savepath home_dir\pathdef.m
```

where *home_dir* is the path to your home directory.

10. Check that MATLAB can now find *myFirstScript* in the path:
    a. Make sure that the MATLAB current folder is NOT your *my_matlab_code* folder
    b. Run your `myFirstScript` script by typing the script name in the command window. This should now run successfully.

11. Finally, check that MATLAB remembers your path when it restarts:
    a. Exit MATLAB. If you are using a UCL cluster computer, please logout and login again.
    b. Restart MATLAB
    c. Check that the MATLAB current folder is NOT your *my_matlab_code* folder
    d. Run you *myFirstScript* script and verify it runs successfully.

Exercise 1.2 – for loops

As you have seen in the lectures, MATLAB can carry out repeat operations using loop structures. One such structure is the `for:end` construct. Consider the following `for` loop that computes, for each value between 1 and 10, the square of that number, and displays the results as it goes.

```
% repeat the following code with i taking each value
% between 1 and 10 (in steps of 1)
for i=1:1:10
    disp([i,i*i]) % display the number and its square
end % end the for-loop (very important!)
```

Now modify this loop to display:

1. The square of only all the *even* numbers between 2 and 10;
2. The square of each number in descending order from 10 to 1;
3. The cube (i^3) of each number between 1 and 20.

Save the code for all of these loops in a single script.

Exercise 1.3 – Writing a script with user input

Here you will write a script that has MATLAB ask for user input during execution.

1. Create a new script.
2. Use the `input` function, which you have seen in the lectures, to ask the user to input a positive integer, and save this value as a variable `maxn`.
3. Use the following command to write the result to the command window as a formatted string:
   ```
   fprintf('Your number was %d.\n',maxn);
   ```
   Add this line to the end of your code, run it, and verify the output is as expected.
   Look at the MATLAB help for the `fprintf` function, and try and understand how it works, and especially what "`%d`" and "`\n`" do.

NOTE: only use the MATLAB function `input` in scripts, and never in functions, as this severely limits a function's versatility!

Exercise 1.4 – Maths: using matrices to rotate a vector

MATLAB was originally developed to simplify the numerical implementation of linear algebra, and as such is excellent in handling matrices and vectors in easy and efficient ways. Here you will apply this functionality to rotate a vector. More information on vector rotations using matrices can be found here; https://en.wikipedia.org/wiki/Rotation_matrix.

Consider the code below.

```
% define a 2D column vector:
v_orig = [1 ; 0]
% Define angle to correspond to a 60-degree rotation:
angle = pi/3
% Define a 2x2 rotation matrix:
M = [cos(angle) -sin(angle) ; sin(angle) cos(angle)]
% rotate the original vector by the 2x2 matrix
v_rot = M*v_orig
```

This code rotates a 2D vector over a known `angle` by means of a matrix multiplication.

Now modify the code above to rotate the vector $\begin{bmatrix} 2 \\ -2 \end{bmatrix}$ by -45°. It may help to sketch out the original and rotated vectors on paper to check the answers are as you expect. Save your code in a new script.

Exercise 1.5 – Functions: using Matrices to rotate a vector

Now that you have written the code to perform vector rotations using matrix multiplications, you might want to reuse this code in the future. Functions are an efficient way of reusing code – they package up a discrete task that you may want to repeat many times. Functions have zero or more input variables, which are processed by the code within the function, and the function finally returns zero or more output variables that contain the results of the operations.

Using the code from exercise 1.5 as a guide, write a function which takes two inputs (a column vector `v_orig` and an `angle` (in degrees)), rotates the vector by the specified angle, and returns the result in an output variable `v_rot`. The first line of the function (the "function declaration") should look like this:

```
function v_rot = rotateVector(v_orig, angle)
```

Once you have written your function, confirm that it yields the correct results for the vectors and angles used in exercise 1.5. Like a script, you can call a function from the command window; however, in the case of functions you have to include the input arguments as well as any outputs the function generates! Typically, you call a function using the format specified by the function declaration – in this case, simply type

```
>> v_rot = rotateVector([2 ; -2], -45)
```

Alternatively, you can call the function using

```
>> v_rot = rotateVector(input_vector, rot_angle)
```

Where `input_vector` and `rot_angle` are variables that you defined within the command window.

Exercise 1.6 – Factors, prime numbers and perfect numbers

The factors of a number are those integer numbers that exactly divide the number into an integer. For example, the factors of 28 are {1, 2, 4, 7, 14, 28}. In principle factors can be both positive and negative, but in this task we will limit ourselves to only the positive factors.

1. Write a function that takes a positive number `n` as input argument, and returns a vector containing all the positive factors of `n`. Note that the numbers 1 and `n` are factors of the number `n`, and should hence always be included.
    a. One way to do this is to loop over all integers between 1 and `n` and test whether they divide `n` without remainder (look up the `mod()` or `rem()` functions in MATLAB). If they do, they are a factor of `n`.
    b. Can you think of easy ways to make this faster? (Note: do not use the built-in `factor` function – this does not provide the correct functionality!)
2. Prime numbers are interesting numbers that have many uses in cryptography and numerical analysis. Prime numbers only have two factors: itself and the number 1.
    a. Write another function that tests if a number is a prime number. This function should take one positive number as input and return a Boolean (logical) variable as output.
    b. Your implementation should call the factorise function you wrote above.
3. Perfect numbers are numbers whose positive factors (excluding the number itself) add up to that number. For example, 1+2+4+7+14=28, which makes the number 28 a perfect number.

a. Write another function that tests if a number is a perfect number. This function should take one positive number as input and return a Boolean variable as output.
b. Your implementation should call the factorise function you wrote above.
c. How many perfect numbers can you find with your code?
(Note: computing the 5^{th} perfect number using the method in these instructions might take well over an hour – it is probably wise to end your search after the first 4 or 5. And the 6^{th} perfect number will take about 300 times longer to compute still!)