Lab 1 - Part 1: Introduction to MATLAB

Lab Overview

This lab is split into two sections, the first of which we will work through today. It's purpose is to introduce you to some of the functionality of MATLAB. This is the computing environment that we will be using for all EOSC 442 labs, so it is critical that you understand how to perform the basic operations you will see today.

Learning Goals

After this lab you should be able to:

- Recognize and use some of the major MATLAB windows
- Perform basic arithmetic in MATLAB
- Create, index into, and manipulate arrays
- Use logical indexing for choosing subsets out of a dataset
- Create plots in MATLAB
- Use scripts to write MATLAB code

To Hand In

Lab 1 worksheet with associated figures

0. Before You Begin

Log in to your computer using your EOSC username and password. Create a new folder in the Z: drive named *EOSC442* and within that another folder called *Lab1*. Avoid using spaces in file and folder names.

Open the Matlab program. Your first task is to ensure that Matlab is currently working in the directory *Lab1* that you just created. In the toolbar near the very top you will see a small field labelled *Current Folder* followed by the path of a folder. If this path is not `Z:\EOSC442\Lab1` use the dropdown menu to navigate there now. Alternatively, you can enter the following command at the command window prompt:

```
>> cd Z:\EOSC442\Lab1
```

Note, the command `cd` stands for '**c**hange **d**irectory'.

1. Introduction

Matlab (**Mat**rix **Lab**oratory) is a computing environment frequently used by scientists and engineers for computer modelling and for data manipulation and visualization. The first time that you open Matlab, it will display the home screen with multiple smaller windows embedded inside of it. You will likely see the title bars '*Command Window*', '*Workspace*', '*Current Folder*', etc...
***Command Window*:** As the name suggests, this is where individual commands may be entered. You enter commands at the prompt symbol:

```
>>
```

Next to the *Editor,* which we will learn about later, this is the most important window in Matlab.

***Workspace*:** This is the place where you can interactively look at the variables Matlab currently has stored in memory. It should be empty when you start Matlab and will begin to fill as you create variables.

2. Getting started: Integers, Floats, Arrays, Strings

Matlab defines classes of variables. Roughly speaking, these are integers, floating point numbers (decimals), arrays, and strings. There are others that we may encounter in the future.

Integers, Floats, and Strings

You can think of integers and 'floats' simply as numbers. Try some simple arithmetic operations at the command prompt, for example:

```
>> 5*4
>> 3.2+7.5
```

If you want to keep the result, you can assign it to a variable, which you can then use in calculations. For example, try the following

```
>> x=5;
>> y=4;
>> z = 2*(x+y)^2
```

The semicolon after the first two commands suppresses the output. The white spaces in the third command are ignored, but I have added them for readability. All three variables `x, y,` and `z` are now stored in memory. In order to see which variables you have saved in memory, enter

```
>> who
```

at the command prompt or have a look at your *Workspace* window. Variables are saved until they are overwritten, explicitly cleared, or Matlab is closed.

If you would like to see the value stored in a variable, simply type its name at the prompt. For example, to see `x`, enter:

```
>> x
```

Similarly to integers or floats, you can create character strings and store them as variables:

```
>> my_name = 'Your name here'
```

In Matlab, anything entered in single quotes is a string. "`Your name here`" is the string and it is stored in the variable `my_name`. You can now use `my_name` anytime you see fit, such as when labelling a plot (we'll get to that). Do not use spaces or special characters in variable

names! An ‘underscore’ is the only exception to this rule.

Arrays

You will be working with these regularly. An array is a set of individual numbers combined into a single object that you may work with. An array can be a column vector, a row vector, or a matrix, and each number within the array is called an ‘element of the array’. Try all the following:

To create a row vector:

```
>> A=[1 3 5 7]
```

A column vector:

```
>> B=[2; 4; 6; 8]
```

A 2x2 matrix

```
>> C=[1 2; 3 4]
```

Notice what the semicolon does in these cases. Arrays are always denoted by square brackets. You can also create new arrays from old ones:

```
>> v1 = [11 12 13 14]
>> v2 = [15 16 17 18]
>> vector1 = [v1 v2]
>> vector2 = [v1; v2]
```

You can transpose a vector or matrix using the ′ operator. Try entering `B'` and `C'` at the prompt. What happens?

As it’s name suggests, Matlab is particularly convenient for matrix operations. Notice what happens when you enter

```
>> A * B
```

This is a matrix multiplication which is (obviously) different than the regular scalar multiplication you are used to. Notice that `A*B` is not equal to `B*A`. Try it. In this course we won’t be using matrix operations much. Instead we will make frequent use of the ‘element-by-element’ operators. These are `.*` `./` and `.^` The `+` and `-` operators perform as you may expect.

Try the following operations

```
>> A .* B'
>> 2*A + B'
>> A .^ 2
```

Notice, in the first two examples, you have to first transpose `B`. Why is this? Try the same operations without first transposing `B` and note the error you get.

A very useful shorthand notation that we will make use of regularly involves the ':' symbol. To create a vector with consecutively increasing values from 1 to 10, enter

```
>> 1:10
```

Similarly, you can create vectors using increments other than 1 over any range that you specify:

```
>> -10:2:10
```

And, of course, you can assign these vectors to variable names if you choose to do so. Making use of a number of the concepts you've seen, you should have no problems understanding this

```
>> E = [400:-50:-400]'
```

In the first two of the above three examples, I neglected to use square brackets around the array. I could have put them in, but they are implied since I am defining a series of numbers. However in the last example I needed to use them explicitly since I also wanted to transpose the vector.

3. Indexing into Arrays

Data that you download or collect will usually be stored in arrays. However, typically you will want to access only parts of a data set. That is, often you will want to access only parts of an array that you have created. This is called indexing.

Regular indexing

The most important thing to remember when indexing is the format *(row, column)*. Thus if you want to access the bottom-left element of the matrix `C` that you defined earlier, you need to access row 2, column 1. In Matlab, this is entered

```
>> C(2,1)
```

You can store the output of this command, which is the integer `3`, as a new variable if you choose:

```
>> my_new_variable = C(2,1)
```

If you are indexing into a vector instead of an array, it is even easier to do since there is only one dimension. To access the second element of `A` enter `>> A(2)` and to access elements two and three, use

```
>> A(2:3)
```

where, like before, the colon indicates a range, in this case "elements two to three". To access all elements except the first in a long array, you could use

```
>> F = E(2:end)
```

where `end` is a keyword that simply means "the last element".

Lastly, if you want to access (say) the first row of a larger matrix, you can do so using the following syntax

```
>> C(1,:)
```

In this context, the colon is shorthand for "all columns". If instead you wanted to access only the first column, the corresponding command is `C(:,1)`.

Logical indexing

To illustrate how logical indexing works, do the following. Let's start by creating a 20 element column vector filled with random numbers between 0 and 100. We'll call this vector `R`.

```
>> R = 100*rand(1, 20)
```

Here `rand` is a function that produces random numbers between 0 and 1. The input arguments to the function are `1` and `20` giving the size of the random number matrix (1x20) that will be created. For more information on `rand()` you can enter

```
>> help rand
```

at the Command prompt. This works for any function that you want to learn about.

Logical indexing works on the principle of selecting data that obeys a logical condition that you specify. For example, say we want to find all points in `R` that are greater than or equal to 50. To do so simply enter

```
>> R >= 50
```

Notice that this returns a vector of zeroes and ones. For reasons that will become clear, let's call this new vector `mask`

```
>> mask  =  R >= 50;
```

Remember that white spaces are ignored; again, I just added them for readability. In order to see what `mask` actually is, look at it side by side with `R`:

```
>> [R mask]
```

This command doesn't actually *do* anything - it just outputs the two vectors side by side to the screen so you can see them. Look closely at the variable `mask` that you just created; it is a *logical array* telling you whether the element in a given position in the vector `R` satisfies the logical condition you specified (1 = true) or does not satisfy it (0 = false). You can now use this logical array to access all the elements that satisfy the condition you specified, or equivalently, to mask the elements that don't satisfy the condition:

```
>> R_new = R(mask)
```

The new vector `R_new` now contains all the values of `R` greater than 50, preserving the order. The possible logical statements relating any two numbers are

`a>b`	greater than
`a>=b`	greater than or equal to
`a<b`	less than
`a<=b`	less than or equal to
`a==b`	is equal to
`a~=b`	is not equal to

4. Plotting

One of the great advantages of Matlab is its ability to easily make and manipulate scientific figures. If you have two vectors `x` and `y` that you wish to plot against each other, the easiest way to plot them is to simply call `plot(x,y)`. Try this example

```
>> x = -10:0.01:10;
>> y = sin(x);
>> plot(x,y)
```

Notice that `x` and `y` must be the same length (have the same number of elements). If you now want to adjust the limits on your axes, you can do so by calling the function `axis([xmin xmax ymin ymax])` where the array specifies the limits. For example

```
>> axis([-10.5 10.5 -1.1 1.1])
```

In order to add labels to the axes, use the following commands

```
>> xlabel('You can put any string here')
>> ylabel('This is a sine wave')
```

You can also try playing around with some of the interactive features in the plotting window. Once you are happy with your plot, you will need to export it to an image file. To do this use the `print` command like this

```
>> print -djpeg 'my_first_plot.jpg'
```

Or if you prefer, you can save your image to a PDF file using

```
>> print -dpdf 'my_first_plot.pdf'
```

Matlab will save the image into the current working directory (folder) which, if you followed Section 0, should be `Z:\EOSC442\Lab1`

We will see some more sophisticated plotting calls in the next section.

5. The Editor: Matlab Scripts

So far we have entered individual commands into the Command Window, asking Matlab to execute those commands one by one. However, any work you do this way will be lost when you close Matlab. Therefore when working on a project or an assignment, you will write your commands in a Matlab script. A script is simply a text file (saved with the file-extension *.m*) consisting of commands which Matlab reads and executes in order.

In the top left of the Matlab window click on the 'New Script' button, enter CTRL+N on your keyboard, or enter

```
>> edit
```

in the Command Window. These will all create a new script. Save this script by clicking on `File > Save As` in the menu bar. Make sure you are saving to the correct directory (folder) and then give it a filename such as `lab1.m`

Now, you can begin to write your first script. Let's imitate a real data set by taking a quadratic function and adding some random noise. Plot this using red open circles. Then we'll plot a pure quadratic function over top of this with a blue line.

```
%%%%%% CLEAR THE WORKSPACE %%%%%%%%
clear

%%%%%% CREATE VARIABLES %%%%%%
x = -10:0.1:10 ;
y1 = x.^2 + 10*rand(1,length(x)) - 5 ;
y2 = x.^2 ;

%%%%%% INITIALIZE THE FIGURE %%%%%%
figure(1);
clf; hold on; grid on;
set(gca, 'fontsize', 15);

%%%%%% PLOT BOTH DATA SETS %%%%%%
plot(x, y1, 'ro', 'linewidth', 1.5)
plot(x, y2, 'linewidth', 1.5)

%%%%%% LABEL THE AXES %%%%%%
xlabel('Place correct x-label here')
ylabel('Place correct y-label here')

%%%%%% EXPORT FIGURE TO AN IMAGE FILE %%%%%%
print -dpdf 'your_figure_name.pdf'
```

Once you have written this script in your editor window, you can run the entire script by entering its name (without the '.m' file extension) at the Command prompt:

```
>> lab1
```

Again, this only works if you are running Matlab in the correct directory! The shortcut for running a script is to press F5 while in the editor window. If you want to run only part of a script (very useful for debugging), you can highlight the section you want to run and pass it to the Command window by pressing F9.

There are a few things here that we haven't looked at before. Any line starting with `%` is a *comment* and will be ignored by Matlab. The call `clear` at the very top clears the workspace. That is, it removes all variables currently in memory. The call `length(x)` simply returns the length of the vector `x`.

Notice that before plotting the data, we *initialize* the figure so that it has the properties we would like. The call `figure(1)` opens "Figure 1"; `clf` clears the figure in case something is already there from a previous plotting command. The call `hold on` allows you to call plot more than once for the same figure. Without this the figure would be cleared each time you call plot. And I'm sure you can deduce what the call `grid on` accomplishes.