Programming Camp: MATLAB

Brian Higgins and Ciaran Rogers

August 2021

1 Why MATLAB?

MATLAB is one of the most widely used programming languages in the Economics profession. Some of the main reasons for this are:

1. Easy to use scripting language for linear algebra
2. Built-in numerical computation
   - Matrix decompositions
     It is particularly well suited for solving systems in matrix form. As you will see later, the more you write your code in matrices rather than vectors/elements, the faster the computation will be.
   - Numerical integration
     It can effectively approximate integrals of complex functions.
   - Optimization, Dynamic Programming etc.
     The predominant use of this programme within some fields (e.g. Macroeconomics/Finance) is to solve models that requires different agents to solve their own optimization problems. MATLAB provides a tractable way of doing this that can be written in such a way that can be understandable/replicable.
3. Plotting in tractable and appealing way

   One can use the programme to directly construct useful figures/tables that can be applied to project write-ups.

2 MATLAB Interface

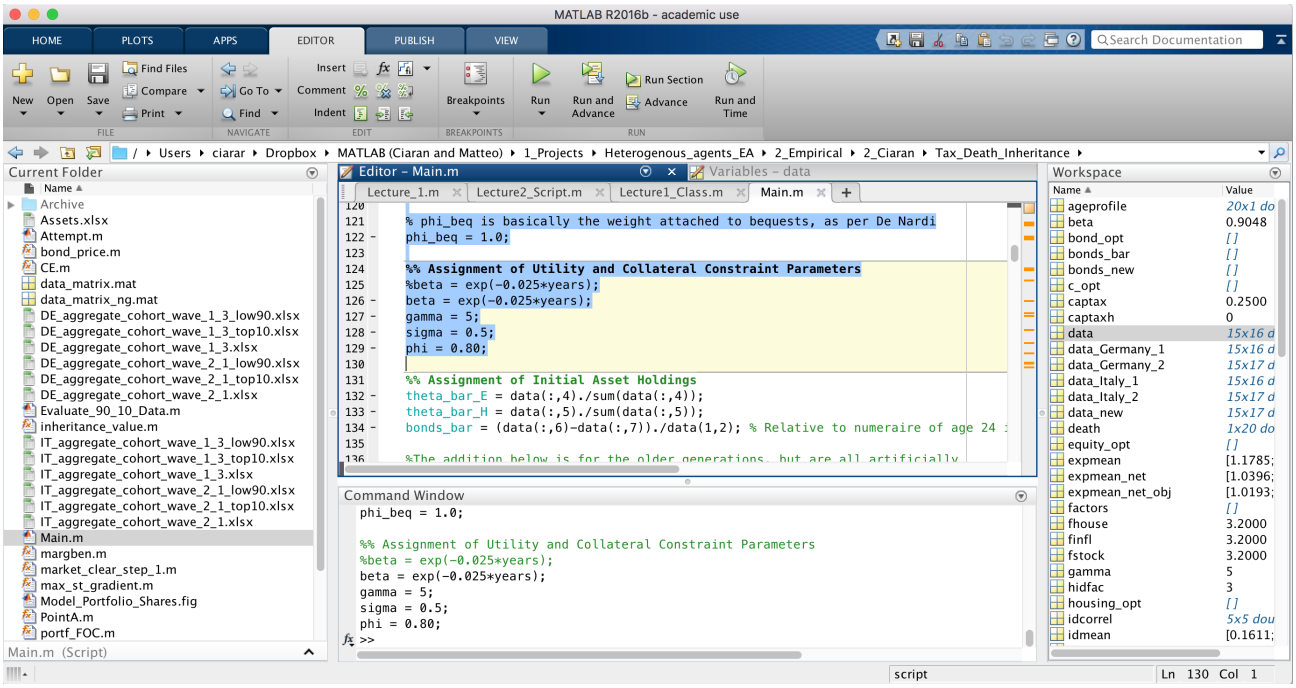

Figure 1: MATLAB Interface

We now go through the various elements of the Interface to help you navigate through what you see.

1. **Command Window**
   Located in the lower center, this displays the lines of code that have been executed. An option for you is to write lines directly in the Command Window to run. You can also have the Command Window display results of your code.

2. **Current Folder**
   Located on the left-hand side, it indicates what folder you are working from. By default, any MATLAB functions / excel files / figures that you may try and use when writing and running your code must come from this folder. However, as we will show later, to access files from another folder, you just need to include an "addpath" line at the beginning of your code.

3. **Command History** - you can't see it here, but you can access this by pressing ↑ while in the Command Window, displaying all the previous lines you have run.

4. **Workspace** - Located on the right-hand side, it contains all the variables/functions/structures that you have created from the code that you have run. These can then be subsequently called on and used in subsequent lines of the code.

5. **Variables and Script** - Location on the upper center, the script is the equivalent of the "Do" file in Stata. The orange lines on the right-hand side indicate points in the code where Matlab believes a mistake may be made / conventions are not upheld. A red bar identifies a problem in the code that will not let it run (e.g. a for loop has not been ended). Generally, very useful to keep tabs with them.

3 Getting Help

IF YOU DON'T KNOW, GOOGLE IT!

As simple as this may sounds, this works 99% of the time. Any problem that you may have is likely to have been experienced by another MATLAB user. The key advantage of MATLAB is that it is a paid service, meaning that help that is provided by the MATLAB providers is very well structured and easy to understand. There also exists online forums that supplement this information. From personal experience, this is a key advantage of MATLAB over other languages.

To look for help directly from the MATLAB interface, simply write one of the two commands in the command window:

» help <command>
» doc <command>

You can also use the following:

» lookfor <search>

But, in general, it is just better to search on Google.

4 Basic Syntax

As a starting point, it is useful to write a few basic commands in the Command Window to get a handle on the language.

1. **Run a Line of Code**: To run a line of code, just write it in the Command Window and press enter. For example:

   $$>> 4 + 5$$

   This will display the answer "9" in the Window, as well as storing the answer as "ans" in the Workspace. If the answer of an equation is not named, the default is to define it as "ans".

2. **Semicolon**: A semicolon stops the output that would be otherwise produced in the command window by the line.

   $$>> 4 + 5;$$

3. **% for comments**: If it is written once in the middle of a line, then everything after the % is not executed.

   $$>> 4 * 5\% - 6$$

4. **Assignment**: This assigns a particular name to the output, creating a new variable of that name. You will see the new variable appear on the RHS of the interface.

   $$>> y = 4/5$$

5. **Matrix Assignment**: You can create matrices by hand, where each row is separated by a semi-colon, and each element of a row separated by a comma.

$$>> y = [1, 2; 3, 4]$$

6. **Single Element Assignment**: For any n-dimensional matrix, you can assign a value to any given element. For example, for a $2 * 2$ matrix, you assign calling on the row and then column number.

$$>> y(2, 2) = 4/10$$

7. **Transpose**

$$>> y = y'$$

8. **Logical**: This is a command that returns a value 1 if True or 0 if false. Within the brackets is the statement to check, where the LHS is a vector/matrix, and the RHS is a scalar.

$$>> z = (y == 0)$$

9. **Colon Operator**: This creates a row vector, where between the colons is the start point, the spacing between points, and the end point

$$>> x = -\text{pi} : 0.1 : \text{pi};$$

10. **Linspace Command**: An alternative method to the Colon operator, and more often used, is to use the linspace function, whose arguments are the start point, the end point and the number of elements within the vector.

$$>> z = \text{linspace}(-\text{pi}, \text{pi}, 100)$$

11. **Call Functions**: You can call on functions that are either in-built (sin, cos, log, exp etc.) or created yourself (as shown later).

$$>> y = \sin(x)$$

12. **Using Logical Statements**: You can use logical statements to find elements with certain characteristics, and assign them a common value

$$>> y(y > 0.5) = 0.5$$

13. **Matrix Multiplication**: Two matrices can either be multiplied in the usual way, or element-by-element. The latter requires using a "." before $*$

$$>> x * x$$
$$>> x. * x$$

Of course, make sure that dimensions allow for the matrix multiplication, otherwise you will get an error (you will be surprised how often it is the reason your code doesn't run!).

14. **Strings**: This creates an effective vector that can have letters as elements. They can be treated like matrices in terms of re-ordering (e.g. transpose)

$$>> a = \text{['two' 'words']}$$
$$a(2)$$

15. **Size**: This produces the size of a matrix for each of its dimensions. If you are looking for the size of a particular dimension, you specify in the second argument which nth dimension.

$$>> \text{size}(x)$$
$$>> \text{size}(x, 2)$$

16. **Zeros**: A useful command is to create a matrix of zeros. This is often a way to initialize a matrix of a specific size that you will subsequently fill with desired values. You can use the "size" function as an argument to make a matrix of the same size.

$$>> \text{zeros}(5, 3)$$
$$>> \text{zeros}(\text{size}(x))$$

17. **Stopping Code Running**: If the code is running for too long / tried to run a script that MATLAB could not handle, you may want to stop the code running, using (for Mac, may be different for Windows):

$$\text{ctrl} + \text{c}$$

18. **Clear Command/Variables/Figures**: This can be done, consecutively, using:

$$\text{clc; clear all; close all}$$

5 Using and Running the Script

Above the command tab on the MATLAB interface is the script section. This is where you can write code without having to run each line one at a time, and can also use it to save written code.

There are several ways to run the code in a script.

- "Run" Tab - This is found on the "Editor" tab at the top of the Interface. This runs the whole script all at once.
- "Run Section" Tab - Also on the "Editor" tab, this runs the desired "Section" of the script. Sections within a script are defined as regions within two lines that each contain "%%" at the beginning. To run a given section, click a line in the section so that it is highlighted, and then select "Run Section".
- Cmd+Enter - Alternatively, highlight all the lines you want to run, and then press Cmd+Enter (for Macs).

6 Basic Plotting

Getting used to generating figures on MATLAB yields substantial benefits for when you want to present your work. Indeed, with enough practice, the figures can become things of beauty!

In order to create the pop-up upon which the figure will be drawn, you always need to start with the following command:

>> figure;

A first step is to merely plot one vector against another, using the following command:

>> plot(x,y);

Then, to simply add a title and label axes, you write the following:

» title('myplot'); xlabel('x'); ylabel('y');

Now, if instead you want a scatter plot, run the following command:

» scatter(x,y);

However, the potential problem here is that any new plotting on the figure will remove whatever was there and will replace it with the new plot. If you, for example, want to retain the plot and the scatter, write "hold on" before running scatter.

» hold on; scatter(x,y);

7 Saving and Loading

This is done using the following commands:

```
save('filename', 'var1', 'var2', 'var3');
load('filename');
```

The example above is for 3 variables. 'filename' is the name you want to assign to the data matrix, whereas 'vars' is actually the data matrix you want to save. When you re-load the matrix, it will retain the names of the variables, not the name assigned to the filename.

8 Adding Excel Data

Suppose we start with a given excel file

Dataset.xlsx

This needs to either be in the folder that is opened on the LHS of the interface, or is in a folder whose path has been added by the function addpath using:

addpath(<pathname>)

To input the data into the MATLAB variable list, I run the following command:

[data textdata] = xlsread('Dataset.xlsx')

The function "xlsread" extracts the file. "data" takes out all the cells that are numbers, whereas "textdata" takes the cells that are words. Most commonly, the first row is in "textdata" i.e. the variable names, while the rest is data.

9 Example: TripAdvisor Dataset

See Section 1 of Problem Set.

10 MATLAB Functions

Throughout your first year, you will be writing many of these: functions! They are essentially a way of you writing code so that, in other scripts, you can call on it like a function f(x). To construct one, you open a new script, and write a code that exhibits the following format:

(BEGINNING)

function < outputvars >=< function-name > (< inputvars >)

(lines that define how the function maps from inputvars to outputvars)

end

(END)

You then save the script under the function name. For example, suppose we want to construct a function $y = min(x, 2)$. The script will be the following:

```
function y = minimumfunc(x)
y = min(x,2)
end
```

This is then saved as minimumfunc.m. Then, in another script within the same folder, writing minimumfunc(x) will produce min(x,2).

11 Control Flow

Again and again, you will be using "for" and "if" loops throughout your code. "For" loops essentially loop over an indicator, repeatedly doing whatever is described within the "for" loop. In general, "for" loops exhibit the following form:

```
for i = 1 : 10
      statements
end
```

For example, consider the following "for" loop:

```
x = 0;
for i = 1:10
      x = x + 2*i;
end
```

So, it runs the loop 10 times, where at each successive loop, i increases by 1, and 2*i is added to x. "If" loops, on the other hand, perform an exercise within the loop ONLY when something is true. In general, they exhibit the following form:

```
if "Conditioning Statement"
      statements
else
      statements
end
```

For example, consider the following "if" loop:

```
x = 2;
y = 3;

if x==y
    z = 'True';
else
    z = 'False';
end
```

Finally, we also have "while" statements. Here it means that, as long as the conditioning statement is true, it will continue to do the loop. They exhibit the following general structure:

```
while "Conditioning Statement"
    statements
end
```

An example is the following:

```
d = 1;
x = 0;
while d>0.001
    x = x + 0.001;
    d = (1-x);
end
```

This "while" loop is a margin of error for getting x as close to 1 as possible.

12 Combining Functions and Loops

1. **Example 1: Fibonacci Numbers**

   See Section 2 of Problem Set.

2. **Example 2: Pareto Distributions**

   See Section 3 of Problem Set

13 Function Handles and Expected Values

This is a convenient alternative to writing an alternative function .m file. This is particularly useful if the function is simple. In general, they exhibit the following form:

funcname = @$(x)(f(x))$

Examples

- Example 1: addone = @$(x)(x+1)$
- Example 2: addition = @$(x,y)(x+y)$
- Example 3: multiply = @$(x,y)(x*y)$