Macroeconomic Policy: A Short Matlab Tutorial

Mario Alloza*

October 27, 2015

Contents

1 Introduction **1**

2 Basic Commands **2**
2.1 Algebraic Operations 2
2.2 Matrices 2
2.3 Useful Built-in Functions 3

3 Loading and Saving **4**
3.1 Importing Data 4
3.2 Exporting Data 5
3.3 Loading and saving matrices 5

4 Plots **5**

5 Loops **6**
5.1 The "for" Loop 6
5.2 The "while" Loop. 7

6 Functions **7**

7 Debugging, Efficient Computation and Good Practices **8**

8 Learning Matlab **9**

1 Introduction

Matlab is a programming language that allows a wide variety of numerical computations and is particularly powerful when performing matrices manipulations. The user's interface includes the following windows (see Figure 1):

- Commmand Window (highlighted in red in Figure 1): this space has a double function: (i) allows the user to type in any input (commands with instructions) and (ii) shows the output of any requested operation.

- Workspace (highlighted in green): here is where all created objects are stored. They can be accessed by double-click in their icons, or typing `open name_of_the_object` in the Command Window. In both cases an Excel-like spreadsheet (the Variable Editor Window) will pop up listing the values of the object.

- Current Folder (highlighted in yellow): shows the files contained in the current folder. The current path is also shown in the box highlighted in purple in Figure 1; it can be changed it to any other folder.

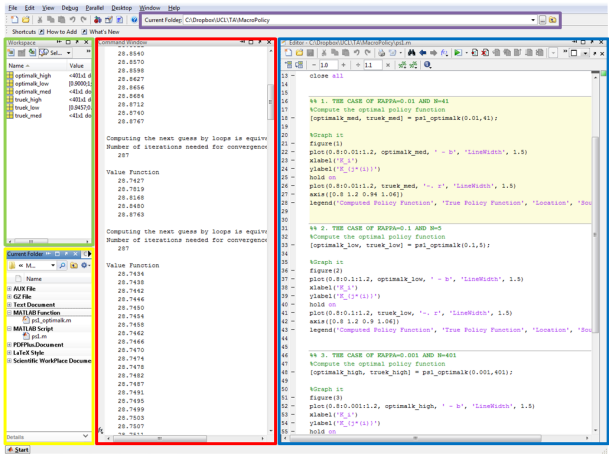
- Command History (not shown): keeps a record of all the commands used.

- Editor (highlighted in blue): in most cases, instead of using the Command Window to type in the instructions, we may prefer to create a file containing a series of commands or a program created by our own. We use the Editor to create this files (with the extension `.m` appended at the end of the filename). These m-files will usually

*PhD candidate, Department of Economics, UCL. E-mail: mario.alloza.10@ucl.ac.uk

be our starting point when creating a new program. The code written in the Editor can be run by pressing the `F5` key or clicking on the green triangle icon.

Figure 1: Matlab User's Interface



2 Basic Commands

2.1 Algebraic Operations

To perform basic operations with scalars, we can type them directly in the Command Window and the output will be shown immediately after our command, as in a calculator; e.g.:

```
5+7
20*3
3/5
ans^2
```

Matlab will store the results of these operations in an object called `ans` which will be overwritten each time we type a command. Alternatively, we can define a name for the result of an operation, e.g.:

```
myresult = 5+7
X = 13-11
Y = 4^X
```

Note that Matlab is case-sensitive, so X $\neq$ x.

2.2 Matrices

Constructing Matrices A matrix can be defined in Matlab by typing one by one its elements separating the columns by a comma (or a space) and jumping to the next row using a semicolon (or by hitting enter). Matrices are always enclosed by square brackets `[]`. Then, we have:

```
X = [3,7;1,4]
Z = [3  7
     1  4]
```

Where `X = Z`, or `X - Z` is the null matrix.

We can transpose a matrix (switching rows by columns) by appending the symbol `'` after a matrix, as in:

```
X'
```

We can create new matrices by merging two or more matrices, or appending vectors or scalars:

```
M = [X Z']
N = [X [3;4] ]
```

Matrices Operations. As in the case with scalar operations, we can perform algebraic operations with matrices:

```
Y = [4 9; 12 0]
X * Y
```

To invert a matrix, we can either use the command `inv()` or, more accurately, the "division" operator. Keep always in mind that matrix multiplication or division are not commutative, i.e. the order does matter.

```
inv(Z)
X * inv(Z)
X / Z
```

We may be interested in computing the Hadamard product (dot product) between two matrices of the same dimensions. We then obtain a new matrix, say

A, with elements $A_{i,j} = X_{i,j} \cdot Y_{i,j}$, where i indexes the rows, and j indexes the columns.

```
X .* Y
X ./ Y
```

Tip: *Now we can solve system of matrices with the tools we have used so far. Rearrange the system to have the form: $A * B = C$ where A are the coefficients, B the unknowns and C real numbers. We can solve for the unknowns by typing:* `B=inv(A)*C` .

Accessing to Elements of a Matrix. We can access to the elements of a matrix, to edit them, or to create a new variable from them. We select an element of a matrix by writing the number of row and column (in brackets) just after the name of the matrix:

```
element_1_2 = X(1,2)
X(1,2) = 654
X(end,2) = 7
```

We can access a group of elements by using the symbol `:`. Hence, `Q(1:4,3)` selects the elements from the 1st to the 4th row in the 3rd column of matrix `Q`. If we just write `Q(:,3)`, all the elements of the 3rd row will be selected. In our example:

```
X(1:2,1) = 3
X(:,2) = Y(:,2)
```

Some Special Matrices. Here we list some commands regarding interesting matrices such as the identity matrix, or matrices containing always the same element:

```
eye(3)       % a 3x3 diagonal matrix
X*eye(2)     % returns matrix X
5 * eye(3)   % a 3x3 diagonal matrix with 5s
zeros(3)     % a 3x3 matrix of zeros
ones(4)      % a 4x4 matrix of ones
diag(X)      % vector with the diagonal elements
```

Tip: *a vector of ones can be easily added to an existing matrix of regressors when we want to estimate a model with constant: if we have* `T` *observations in a matrix* `X`*, we can write* `X = [ones(T,1) X]`.

2.3 Useful Built-in Functions

A list with some convenient commands:

Housekeeping. `clear` cleans all the objects defined during the work-session. `clc` cleans all the commands (and their output) typed in the Command Window. `close all` closes all the opened windows showing figures.

Vectors. We can create vectors of consecutive numbers by using the symbol `:`. If we write `n:m`, Matlab will produce a row vector containing a list of numbers $n, n+1, n+2, \ldots, m-2, m-1, m$. We can create a column vector by transposing the row vector. Additionally, we can create a list of numbers $n, n+k, n+2k, \ldots, m-2k, m-k, m$ with the command `n:k:m`:

```
1:6
(1:6)'
0:2:10
```

Similarly, the command `linespace(x1,x2,n)` will create a row vector of `n` equally separated numbers from `x1` to `x2`.

```
linspace(1,10,10)'
```

Tip: *any of the above commands, can be used to incorporate an extra column (or row) to a matrix of regressors containing a linear trend. How could we add a quadratic trend to a matrix?*

Elementary Functions. Commands as `sqrt()`, `std()`, `exp()` or `log()` compute the square root, the standard deviation, the exponential or the logarithm of the argument inside the brackets:

```
sqrt(144)
std(1:6)
exp(1)
log(exp(1))
```

Note that there exist lots of built-in function in Matlab as `mean`, `min`, `max`, `sum`, `round`, ... Use the `help` command to see how they work.

Time-Series Functions. When dealing with time-series data, latest versions of Matlab include commands such as `lagmatrix(Y, n)`, which creates a new matrix as `Y`, but lagged `n` periods. Note that when estimating a time-series model, you may want to include a number of lags, then you can substitute `n` by a sequence of lags as `1:n`. The command `diff(Y,n)` will compute the n-th difference of matrix `Y`. Examples:

```
R = [4 7; 5 8; 9 3; 12 0; 1 5];
lagmatrix(R,1:3)
diff(R,1)
```

Random Number Generators. Sometimes, we may be interested in generating random numbers. This functionality is implemented in Matlab by using the commands `rand(m,n)` or `randn(m,n)` to generate $m \times n$ matrices of random numbers following a uniform or a $\mathcal{N}(0,1)$ distribution, respectively.

```
rand
rand(10,2)
randn(5)
```

Tip: *What if we want to draw numbers from a $\mathcal{N}(\mu,\sigma)$ distribution? We could implement this (assuming, for example $\mu = 5$ and $\sigma = 2$ using* `data=randn(10000,2)*2 + 5;`. *To check it, write* `mean(data)` *and* `std(data)`.

Changing the Shape of Matrices. If we are interested in replicating the same matrix a number of times, we can make use of the Kronecker product function built in Matlab. Recall that $A \otimes B$ multiplies each element of matrix A by the whole matrix B, therefore, if we substitute A by a matrix of ones of order n, we would be replicating matrix B n times. The same can be achieved by using the command `repmat`:

```
X = [3,7;1,4]
kron(ones(2,2),X)
repmat(X,2,2)
```

We can also vectorise a matrix in Matlab. Hence, a $n \times m$ matrix would be converted into a $nm \times 1$ vector, by stacking all the columns of the matrix. Inversely, the command `reshape(X,n,m)` can change the shape of a matrix `X` to a new one with `n` rows and `m` columns.

```
vecX = X(:)
reshape(vecX,2,2)
```

We can determine the size of a matrix by means of the command `size()`. This particular instruction, as many other functions implemented in Matlab, delivers two different objects as output: an object containing the number of rows, and an object containing the number of columns. Therefore, we have to define the output (what is at the left hand side of the = symbol) accordingly:

```
[rows columns] = size(X)
```

Tip: *the command* `length(X)` *can alternatively be used to compute the maximum size of matrix* `X` *(i.e. in a $n \times m$ matrix, it returns n if $n > m$ or m otherwise).*

3 Loading and Saving

3.1 Importing Data

Matlab can load data in excel format (`.xls` or `.xlsx` extensions) by using the command `xlsread`. We will have to specify first the name of the file (including its extension) and the sheet where the data is place (if there is only one sheet, you can skip this step). Both arguments should be enclosed in `''`:

```
mydata = xlsread('GDPdata.xls','dataMatlab');
TIME = mydata(:,1);
GDP = mydata(:,2);
```

It is important to note that our Current Folder (see section 1) must be the one that contains the file that we are trying to load.

When we want to import data in other format rather than a spreadsheet, or when we are using a Mac (we cannot use the above command with computers that do not use Microsoft Windows as operative system) we can use the command `importdata`:

```
data_imp = importdata('GDPdata.csv');
mydata = data_imp.data;
TIME = mydata(:,1);
GDP = mydata(:,2);
```

In this example we have used a comma-separated (`.csv`) file, but we could have alternatively used other types of text files. Notice that, when we use the command `importdata`, the object that is created (data_imp in the above example) is what Matlab calls a "structure", which includes both data and text. The second line in the above code extracts just the data component of this structure.

3.2 Exporting Data

The easiest way to export data is to open the Variable Editor Window (see the description of the Workspace in section 1) and then copy the values and paste them into Excel (or elsewhere). This can be done in a more elegant way by using the commands `xlswrite` or `export`

3.3 Loading and saving matrices

The commands `load` and `save` allow us to create or open a file containing all or some of the matrices in our Workspace. The resulting file will be readable only by Matlab. This could be a potential solution when using very large matrices that use up the computer's memory; in that case, by dividing a big matrix in smaller ones that will be saved and loaded sequentially, we would be able to perform certain operations in a faster way.

4 Plots

Matlab is a particularly suitable tool to plot data. The basic command for plotting a vector is `plot(Y)`. However, we can also type `plot(X,Y)` to plot data in vector `Y` on vector `X`:

```
X = (1:100)';
Y1 = sin(X)
Y2 = cos(X)+2
plot(X,Y1)
```

When running a code/program that plots multiple pictures, keep in mind that by default Matlab will make the plots in the same window, overwriting the previous figure. To make things clear, you can either use the command `close all` to close the previous file or use the command `figure` before the code regarding the plot to ask Matlab to draw the new picture on a separate window. These windows can be numbered: `figure(1)`, `figure(2)` ...

```
figure
plot(X,[Y1 Y2])
```

Matlab implements a wide range of plot types, a feature which may be useful depending on the nature of the data we want to plot:

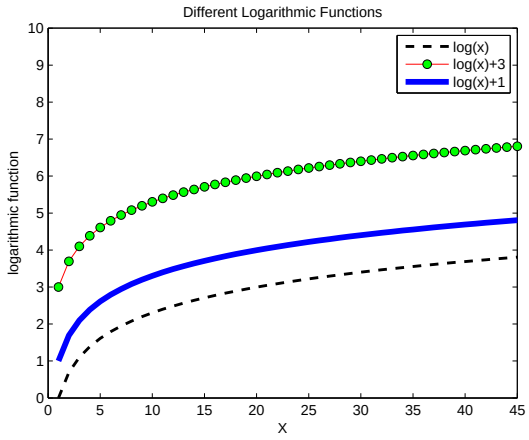
```
close all
a= randn(1000,1);
hist(a)
```

You can manipulate the options of a plot: font, size, colours, etc. Here we show an example of plot using some of the available features, which can be appreciated in Figure 2

```
figure
plot(log(1:50),'--k','LineWidth',2)
hold on;
plot(log(1:50)+3,'-ro','MarkerEdgeColor','k', ...
        'MarkerFaceColor','g')
plot(log(1:50)+1,'-b','MarkerEdgeColor','k', ...
        'LineWidth',4)
title('Different Logarithmic Functions')
legend('log(x)','log(x)+3','log(x)+1')
xlabel('X')
ylabel('logarithmic function')
axis([0 45 0 10])
```

Saving Plots. You can save any plot that you have computed: in the figure window, click on File and then on Save As. You will be able to store the graph in a wide variety of formats: `pdf`, `eps`, `png`, `jpg`...

Figure 2: An Example of Plotting Options in Matlab



You can also use the format `fig`, the extension for Matlab figures, that will allow you to reopen the saved figure in Matlab and perform operations with it (e.g.: change the appearance of the figure).

5 Loops

When the same instruction is required to be repeated a number of times, we may use a loop to ask Matlab to execute this repeated instruction. The use of loops is very common in programming, and can enormously simplify one's code.

5.1 The "for" Loop

A Simple "for" Loop. The `for` loop repeats the same instruction as many times as we define. The syntax is very simple: the instruction to be repeated is preceded by the command `for index=first_iteration:last_iteration` and is followed by `end`. For example:

```
for n=1:3
   n
end
```

Nested Loops. We can nest two ore more `for` loops to execute different operations. In the following example, we are defining each element of matrix $W_{n,m}$ in row n and column m as $W_{n,m} = n^m$. Therefore, we use two nested `for` loops to "scan" each element in the matrix W and compute the operation (we would have used three loops for a 3-dimensional matrix):

```
W = zeros(5,4)
[rows columns] = size(W)
for m=1:columns
    for n=1:rows
        W(n,m) = n.^m
    end
end
```

Example: generating a Random Walk. Artificial time-series data (or any operation requiring a recursive computation) can be generated using a `for` loop. In this example we generate the following time series: $y_t = y_{t-1} + \varepsilon_t$, where $\varepsilon_t \sim \mathcal{N}(0,1)$ and $y_0 = 0$.

```
y = zeros(100,1);
eps = randn(100,1);
for i=2:100
    y(i,1) = y(i-1,1) + eps(i,1),
end
plot(y)
```

A Simple "for" Loop with Conditional Statements. We often want to compute an instruction conditional to some statement. In this situation, we can use the command `if condition_to_be_checked` followed by the instruction and the command `end`. Additionally, we can add the command `else` to define a new operation to be executed if the condition is not met (Note: we could even add further conditions by using the command `elseif`):

```
P = [rand(20,1) zeros(20,1)];
for i = 1:length(P)
    if P(i,1) ≥ 0.5
        P(i,2) = 100;
    else
        P(i,2) = -2;
    end
end
```

5.2 The "while" Loop.

This loop, will execute the instruction for an undefined number of times until some condition is met. The syntax is very similar to the `for` loop. You should keep in mind to observations when using this kind of loops. First, before the loop finishes (i.e. before writing `end`), we have to write an instruction to tell Matlab that the loop must be kept running before the condition is not met. Second, the `while` loop can be running an infinite number of times if the condition is not met; if this is due to a programming error you must stop Matlab with `Ctrl+C`.

```
draw = rand
while draw <0.9
    display('The draw was below 0.9, try again.')
    draw = rand
end
```

6 Functions

Commands such as `mean()` or `rand()` call built-in functions already incorporated in Matlab. However, we can create our own functions to perform some specific tasks. In this way, we will simplify our code both because we don't have to replicate code (instead of writing the same set of instructions twice, we can just call our function) and because our program will be more readable when having fewer lines in the main file, creating separated files with functions to execute specific tasks.

When writing a function we will use the following syntax:

1. begin the file with `function [output_1, output_2 ...] = function_name(input_1, input_2...)`
2. write all the instructions that making use of `input_1, input_2...` specified by the user will produce `output_1, output_2 ....` Obviously, we must use the same names to define the output variables as we defined them in the previous point.
3. end the function file with the command `end`.

When using our own function, it is important to notice the following:

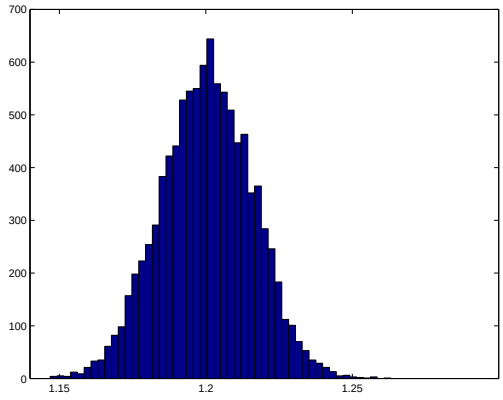
- The `function_name` specified above, must be the same name that we used to save the new `.m` file: if our function is called `ComputeStdErrors`, the new file containing the function will be saved as `ComputeStdErrors.m`.
- We can call the function form our main file by writing `[result_1, result_2 ...] = function_name(argument_1, argument_2...)`. Note that we write `result` and `argument` instead of `output` and `input` to emphasise that the names give to these variables don't need to be the same.

To illustrate the use of functions, the next example creates artificial data and runs an OLS estimation of the classical linear model by means of a function defined by our own function, called `OLSestim`. This example is also meant to summarise some of the highlighted concepts shown in this short tutorial.The main file is:

```
% OLS estimation of artificially generated data

% We assume we know the true DGP
T = 50000; % number of observations
eps = randn(T,1)*7; % errors
X = randn(T,1)*2 + 3; % regressors
X = [ones(T,1) X]; %include a constant
beta_true = [0.7 1.2]'; % true parameter
Y = X*beta_true + eps;

% Run the estimation
[beta_estim,sigma_estim] = OLSestim(X,Y);

display('The true parameters are:');
beta_true
display('The estimated parameters are:');
beta_estim

% Repeat the estimation B times
% for different draws of epsilon
B = 1000;
beta_mat = zeros(B,2);
for i=1:B
  eps = randn(T,1)*7;
  Y = X*beta_true + eps;
 [beta_mat(i,:), sigma_mat(i,:)] = OLSestim(X,Y);
end
```

```
display('Mean estimation of the slope coeff.:')
mean(beta_mat(:,2))
hist(beta_mat(:,2),50)
```

Figure 3: Distribution of the Slope Estimator over Different Samples



The m-file containing the function `OLSestim` is:

```
function [beta_hat,sigma_hat] = OLSestim(XX,YY)

beta_hat = inv(XX'*XX)*XX'*YY;

% Alternatively, a more efficient code would be:
% beta_estimt = X\Y;

residuals = YY - XX*beta_hat;
df = length(XX)-length(beta_hat);
sigma_sq = (residuals'*residuals)/df;
sigma_hat = sqrt(sigma_sq);

end
```

7 Debugging, Efficient Computation and Good Practices

Unfortunately, our own programs will not always run smoothly and some programming mistakes must be detected. Despite Matlab offers some tools to make this task less painful, writing "clean" and fast code, can save us hours (or even days!). Here there are some tips:

- Try to use functions to separate from the main code those tasks that are to be repeated a number of times or that contain specific code which can be easily isolated front the core of our program. Ideally, a Matlab project will involve a bunch of programs/functions which are called from a "master" m-file.

- Be generous writing comments (using the symbol `%`). Despite that is a tedious thing to do when writing code, it will prove to be a very useful practice when you come back to your projects after some time.

- Keep it simple. Avoid complex names for new variables and try to find a method that works for you.

- Keep it smart. There are many ways to deal with the same task, however, more elegant ways tend to make your code more understandable by others and even by your own (e.g. copy and paste several times the some code lines can be substituted by a loop).

- Efficient Computation. Among other features of your computer, Matlab performance is strongly correlated with your computer processor. However, programs can run much faster when we avoid tasks that are particularly slow for Matlab:

  - Matlab loves matrices: when possible, use matrix operations rather than loops. Example: using a loop to execute the same instruction for each element of a matrix may be slower than doing once the instruction for the whole matrix.

  - When running a loop that stores the results in a matrix, it is faster to define this containing only zeros before the loop starts, rather than letting the size of the matrix growing in each iteration.

  - Some commands are faster than others while producing the same results: `Y/X` is faster than `X*inv(Y)`.

  - To check the time that Matlab takes to perform an operation you can use `tic` and `toc` as in `tic; randn(10000,1); toc`.

- Take advantage of Matlab capabilities. You can divide your code in `cells` (see the Matlab menu called "Cells") by writing two comment symbols `%%` for each cell. Matlab allows you to evaluate these cells instead of the whole program, something that may be useful at the early stages of your project.

- When debugging, it is sometimes very convenient to check that Matlab is doing what you think it should be doing. Setting breakpoints (see Matlab menu "Debugging" or click at the right of a line number in for code until a red ball appears) while make Matlab execute all the code until this point and allow you to progress step by step from that point onwards. This is particularly interesting when running a code that involves functions, since when Matlab is executing a function, the objects created by this functions are not stored unless they are defined as output.

- You can also compare that two alternative procedures yield the same results by using conditional statements: true statements as `3==3` will return `1` as an answer, while false statements as `3==5` will return `0` as an answer. If you want to compare two vector of results that seem to have similar results but you cannot check it due to its large dimension, `min(my_result_1==my_result_2)` should be `1`, otherwise, the two methods are not equivalent for some cases.

8 Learning Matlab

Matlab implements a wide variety of functions. Since remembering all the bells and whistles of each one is a hard thing to do, Matlab incorporates a detailed documentation of each command. You can access to this information by writing `help unknown_command`. Additionally, use the "Help" menu to browse and search all the functions implemented to check if Matlab already incorporates what you are looking for. For example, there is no need to program a Choleski decomposition when the command `chol()` already does this.

One of the greatest advantage of Matlab is that is a widely used software. That means that there exist a lot of code already written. Some economists share freely their code what makes learning much easier: visit for example the webpages of Chris Sims, Larry Christiano or Martín Uribe for some examples. Some journals as the Review of Economic Dynamics or some articles of American Economic Review include code replicating the results of the published papers; that's a perfect opportunity to see how a professional code looks like.