Introduction to Matlab

Lesson 01 — Matlab Syntax

Rafael Serrano-Quintero

Department of Economics
University of Barcelona

Preliminaries

Preliminaries

The Course

- Introduction to Matlab Programming

- Check the syllabus

Me

- Rafa Serrano-Quintero (rafael.serrano@ub.edu)

- Office hours: Any time. Send me an email and we can arrange a meeting.

You

- A quick roundtable of names, interests, and coding background.

What will we cover?

1. Matlab preliminaries.
   - First interactions. Script vs Command Window.
   - Creating Variables. Basic Operations. Arrays and Matrices.
   - Control Flow. Plots. Functions.
2. Importing and manipulating data. Polynomial fit and evaluation. Nonlinear least squares.
3. Basics of root finding, numerical differentiation and integration.
4. Basics of numerical optimization.

Syllabus Highlights

What you have to do

1. Two problem sets
2. In class solutions
3. Final exam (take-home)

Syllabus Highlights — Materials

- Stormy Attaway (2019). *MATLAB: A Practical Introduction to Programming and Problem Solving*. 5th. Butterworth-Heinemann
- Peter H. Gruber — Script Solving Economics and Finance Problems with MATLAB
- QuantEcon Lectures. These are written for Python and Julia but many ideas port to Matlab easily.
- QuantEcon Cheatsheet — for Matlab, Python, and Julia.

What Matters

- Not where you end up relative to your classmates but where you end up relative to your current self!
- Most of you never have programmed, and that's **fine!**
- What is programming? Problem solving!
- Programming in one diagram:

Why do we need programming?

Exercise 1

Suppose a firm faces a linear demand curve such as $P = a - bQ$ and the firm's cost function is given by $C(Q) = cQ + d$ where P is price, Q quantity, and $a, b, c,$ and d are positive real constants. Find the optimal quantity to produce.

Why do we need programming?

Exercise 1

Suppose a firm faces a linear demand curve such as $P = a - bQ$ and the firm's cost function is given by $C(Q) = cQ + d$ where P is price, Q quantity, and a, b, c, and d are positive real constants. Find the optimal quantity to produce.

Solution 1

Deriving the profit function $\Pi = PQ - C(Q)$ *with respect to* Q*, we find*

$$\frac{\partial \Pi}{\partial Q} = -2bQ + (a - c) = 0 \Rightarrow Q = \frac{a - c}{2b}$$

Why do we need programming?

Exercise 2

Suppose now the demand curve is $P = 10 - Q^{1/2} - Q^{1/5}$ and $C(Q) = cQ + d$.

Solution 2

Now the profit function is $\Pi = 10Q - Q^{3/2} - Q^{6/5} - cQ - d$ *and the FOC is*

$$\frac{3}{2}Q^{1/2} + \frac{6}{5}Q^{1/5} + c - 10 = 0$$

How do we solve this "by hand"? Say $10 - c = 1 \Rightarrow c = 9$*. Then, we can define* $G(Q) \equiv \frac{3}{2}Q^{1/2} + \frac{6}{5}Q^{1/5} - 1$ *and find* $G(Q) = 0$*.*

- $G(0) = -1$

- $G(1) = 1.7 > 0$

We know there exists some $q \in (0, 1)$ *such that* $G(q) = 0$*. But we cannot say much more analytically!*

Why do we need programming?

In Matlab, we could find the solution with two lines of code:

```
gq = @(q) (3/2) .* q.^(1/2) + (6/5) .* q.^(1/5) - 1;
qopt = fzero(gq, 0.5);
qopt =
      0.0509155768653817147884055316353 62
```

Why do we need programming?

- Econometrics/Empirical analysis.
- Applied general equilibrium.
- Complement for theory. Substitute when looking at very complex models.
- We will use Matlab but many things translate to other languages (e.g. Python, Julia, R…).
- Try to learn **concepts!**

First Time Opening Matlab

First Time Opening Matlab

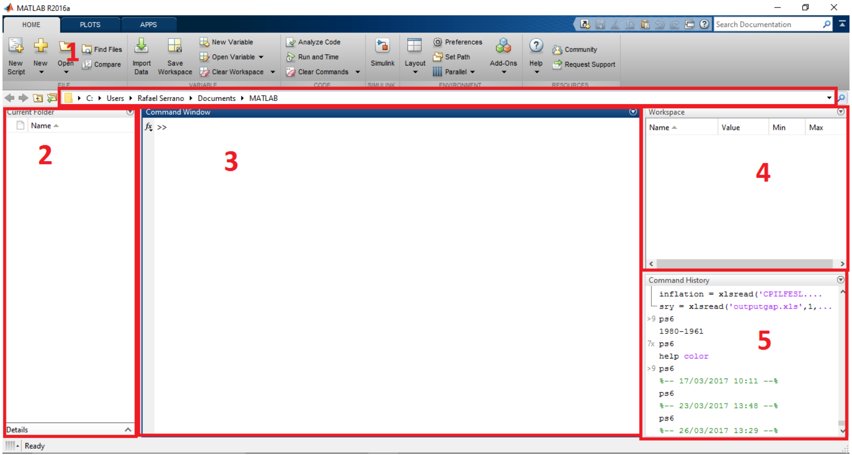

Figure 1: Matlab's Interface

Working Directories

- For Matlab to find files and know where its working, it needs a **working directory**.

- A directory is a folder that has relationships to other folders.

```
matlab-class
 ├── slides
 ├── codes
 └── problem-sets
      ├── ps1
      └── ps2
```

- Say we want to run codes in the `codes` subfolder. Then, we must **point to where** the folder is.

Working Directories

```
matlab-class
├── slides
├── codes
└── problem-sets
    ├── ps1
    └── ps2
```

- In this example `matlab-class` is called the **parent directory**.
- But `problem-sets` is the parent directory of `ps1` and `ps2`.
- The **working directory** refers to the directory on your computer that Matlab assumes is the starting place for all paths that you construct or try to access.

Paths

```
matlab-class
├── slides
├── codes
└── problem-sets
    ├── ps1
    └── ps2
```

- Suppose `matlab-class` is in `C:\Users\Rafa\Documents\matlab-class\`.
- That is called the **absolute path**.
- The absolute path of `slides` is `C:\Users\Rafa\Documents\matlab-class\slides\`.
- The **relative path** is a path relative to the working directory.
- The relative path of `ps1` would be `./problem-sets/ps1/`.
- Matlab understands relative paths.

The Two Most Useful Commands

- If you make some mistake and need Matlab to stop, `Ctrl+C` on the command window.
- When you do not know how a command works or what it does, use `help`.

Scripts vs Command Window

Scripts vs Commands

- The *command window* allows us to evaluate commands we type.
- *Scripts* are *recipes* that can be saved. Useful for:
  - Reproducing a set of codes exactly in the same order
  - Automating tasks
  - Correcting mistakes in long tasks
- A script is evaluated sequentially line by line.

Scripts vs Commands

- To start a new script:
  - `Home -> New Script`
  - Type `edit` in the command window
- Write comments with the symbol `%`
- Everything after a `%` will not be processed by Matlab
- A block of comments is defined within `%{ %}`

Script Example

```
clear all
close all
clc

% This is a comment

%{
    This is a block of comments. Everything within the two
        symbols is not processed by Matlab. Useful to define
        headers or helps for user defined functions.
%}
```

Creating Variables

Creating Variables

- We can **assign** values to a variable.
- Matlab has several types of objects (arrays, struct arrays, cell arrays...)
- To name a variable the first character **needs to be a letter**.
- Matlab is case sensitive $x \neq X$

Creating Variables

Forbidden names:

- i and j indicate complex numbers.
- `pi` is assigned to π.
- `ans` is assigned to the last value that has not been assigned to anything.
- `Inf` or `-Inf` are $\pm\infty$.
- `NaN` represents *"Not a Number"* (typically missing data).
- `eps` is the *machine epsilon* (we will comment a bit on this below).

Creating Variables

```
clear
clc

%==========================================================
                % === Creating Variables === %
%==========================================================

% Assignment and basic operations
x = 5
x*2
x-7
x+7

% Creating variable from another
y = x^2;    % Semicolon ; suppresses output
disp(y)
```

Creating Variables

```
x = 3;
y = 4;
x = y;
y = 2;
```

- What is the value of `x`?

Creating Variables

```
x = 3;
y = 4;
x = y;
y = 2;
```

- What is the value of x?
- The = operator is an **assignment operator**.
- In Matlab, $x = 4$ **not** 2, so it will not be equal to y anymore.
- In some languages, this operator might tie x and y together. Always check!

Matlab as a Calculator

- To perform basic arithmetic operations Matlab uses five symbols.

- These operators are defined for both *matrices and scalars*.

Table 1: Basic Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

Matlab as a Calculator

Exercise 3

Use Matlab as a calculator and try to solve the following operations using the functions needed. Solve them for $x = 0$ and $x = \frac{\pi}{4}$

$$\frac{\left(\ln\left(1 + x^2\right)\right)^2 - \sqrt{1 + \sqrt[3]{x^2}}}{1 + \sin^2 x} \; ; \; \ln\left|\frac{x - \pi}{x + \pi}\right| + \sqrt{\frac{e^x}{1 + xe^x}}$$

Hint: Check commands `log`, `sqrt`, `sin`, `abs`, `pi`

Matlab as a Calculator

Solving for $x = 0$. Check commands:

- log, sqrt, sin, abs, pi

```
% === Ex. 1: Solve Complex Operations === %
x = 0;

op1 = ((log(1+x^2))^2-sqrt(1+x^(2/3)))/(1+sin(x)^2);
op2 = log(abs((x-pi)/(x+pi)))+sqrt(exp(x)/(1+x*exp(x)));
```

Arrays

Arrays — Vectors and Matrices

- Matrices and arrays are the fundamental representation of data in Matlab.

- An array is just a systematic arrangement of objects. A vector is a one-dimensional array, while a matrix is a two-dimensional array.

- **Row vectors** (the default in Matlab) are created as

```
%=========================================================
                    % === Vectors === %
%=========================================================

rowV = [1 2 3 4 5];        % Spaces separate elements
rowV2 = [6,7,8,9,10];      % Commas work as spaces
```

Arrays — Vectors and Matrices

- **Column vectors** are created with the semicolon operator ;

```
colV = [1; 2; 3; 4; 5];
```

- Sequences as row vectors

```
% A sequence from 0 to 10 in steps of 2
seq1 = 0:2:10;
% 1000 equally spaced elements in [0,10]
seq2 = linspace(0,10,1000);
```

- **Check your workspace!**

Arrays — Vectors and Matrices

- **Matrices** can be created element by element or by *concatenating vectors*.

- A semicolon ; after a number indicates a new row.

```
A = [1 2 3; 4 5 6; 7 8 9];
B = [4 5 6; 7 9 2; 1 5 32];
ConcatenatedMatrix = [rowV;rowV2];
```

- Produces:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \; B = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 9 & 2 \\ 1 & 5 & 32 \end{bmatrix} \; ConcatenatedMatrix = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \end{bmatrix}$$

Arrays — Vectors and Matrices

Array Indexing

- In matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ the element $a_{2,3} = 6$ is the element in row 2 and column 3.

- In Matlab we can access element $a_{j,k}$ as `A(j,k)` to access 6 in matrix A before:

```
A(2,3)     % Element in row 2 and column 3 of matrix A
```

- For vectors, we can index by the position only.

```
colV(1)    % First element in colV vector
```

Arrays — Vectors and Matrices

Array Indexing

- For matrices in general, we can access whole columns or rows.

```
A(2,:)          % Index the 2nd row and all columns
```

- In general, we can index matrices as

Table 2: Basic Indexing

Index	Result
`A(i,j)`	$a_{i,j}$
`A(i,:)`	Row i
`A(:,j)`	Column j

Arrays — Vectors and Matrices

Exercise 4

Create the vectors $v_1 = [1, 2, 3]$ and $v_2 = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$. Note that to transpose a matrix/vector you can use the apostrophe (').

1. Concatenate v_1 and v_2 to create the matrix $M_1 = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$
2. Concatenate v_1 and v_2 to create the matrix $M_2 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$
3. Extract element in position $(2, 2)$ of M_1 and M_2.

Arrays — Operations

- We can use the same operators described in Table 1 **BUT** mind the laws of matrix algebra.

- Matrix products `A*B`

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \; B = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 9 & 2 \\ 1 & 5 & 32 \end{bmatrix} \; AB = \begin{bmatrix} 21 & 38 & 106 \\ 57 & 95 & 226 \\ 93 & 152 & 346 \end{bmatrix}$$

- To transpose a matrix A:

```
% To transpose a matrix use transpose() or '
A'
transpose(A)
```

Arrays — Operations

Element-wise Operations

- Element-wise product of A and B is computed by $a_{i,j} \times b_{i,j}$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \; B = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 9 & 2 \\ 1 & 5 & 32 \end{bmatrix} A \odot B = \begin{bmatrix} 4 & 10 & 18 \\ 28 & 45 & 12 \\ 7 & 40 & 288 \end{bmatrix}$$

- In Matlab, we use

```
% Element-wise multiplication of A and B
A.*B

% Be careful with element wise multiplication. Check what
    would happen if:
rowV.*rowV2'
```

Arrays — Matrix Operations

Table 3: Matrix Arithmetic Operations

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`.*`	Element-wise product
`\`	Left division $(A \setminus B = A^{-1}B)$. Equivalent to `mldivide()`
`/`	Right division $(A/B = AB^{-1})$. Equivalent to `mrdivide()`
`./`	Element-wise division
`^`	Exponentiation
`.^`	Element-wise exponentiation
`inv()`	Inverse of matrix

Exercise 5

Solve the following system of linear equations using `inv()` and `\`.

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - z &= 0 \end{aligned}$$

Arrays — Matrix Operations

To solve systems of equations, it is recommended to use \ instead of `inv()`. If you're interested you can check this or try this example.

```
% === Ex.2 Solve the system === %
b = [1; - 2; 0];
A = [3 2 -1; 2 -2 4; -1 0.5 -1];
x = inv(A)*b;    % Slower and inaccurate
x_oth = A\b;     % This method is preferred
x_oth2 = mldivide(A,b); % Same as the previous one
disp([x,x_oth,x_oth2])
```

Relational and Logical Operators and Loops

Relational and Logical Operators

Relational Operators

- Check if $a \neq b$

- Check wether $a \lesseqgtr b$

Logical Operators

- Check wether one or more conditions are satisfied

- Access elements that are **NOT** equal to a.

Relational and Logical Operators

- Relational and logical operators return either *true* or *false* values.
- In Matlab, *true* is coded with 1 and *false* with 0.
- But **these are not numbers!!**
- The result is a *logical array*.

Relational and Logical Operators

```
A > 5
A == 5
A <= 5
```

Table 4: Relational Operators

Operator	Meaning
==	Exactly equal to
~=	NOT equal to
<	Lower than
<=	Lower or equal than
>	Lower than
>=	Lower or equal than

Relational and Logical Operators

```
A > 5 | A < 9
~(A > 3 & A < 6)
```

Table 5: Logical Operators

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td><code>&</code></td><td>Element-wise AND</td></tr>
<tr><td><code>&&</code></td><td>AND for scalars</td></tr>
<tr><td><code>|</code></td><td>Element-wise OR</td></tr>
<tr><td><code>||</code></td><td>OR for scalars</td></tr>
<tr><td><code>~</code></td><td>NOT</td></tr>
<tr><td><code>any()</code></td><td>True if any element of a vector is true</td></tr>
<tr><td><code>all()</code></td><td>True if all elements of a vector are true</td></tr>
</table>

If-Else Statements

- Typically, relational and logical operators are used as conditions.
- **If** something happens, do something. **Else** do another thing.
- If-Else statements start with `if` and are closed with `end`. General syntax:

```
b = 3;
if b < 0
    disp('b is negative')
else
    disp('b is non-negative')
end
```

If-ElseIf-Else Statements

- If we want to include two possible conditions, we use `elseif`

```
b = 4;
if mod(b,2) == 0 % Check if even
    disp('b is even')
elseif mod(b,5) == 0 % Check if divisible by 5
    disp('b is divisible by 5')
else
    disp('b is not even nor divisible by 5')
end
```

For and While Loops

- Sometimes we need to repeat the same operation several times.
- When we know how many times exactly, we should use `for` loops.
- If we do not know how many times, but we know a criterion, then we should use `while`

For and While Loops

Suppose we want to simulate an $AR(1)$ process for 100 periods such as

$$y_{t+1} = \rho y_t + \varepsilon_t$$

where $\rho = 0.85, y_0 = 0$, and $\varepsilon_t \underset{iid}{\sim} \mathcal{N}(0,1)$

```
T = 100;
rho = 0.85;
y = zeros(100,1);
for t=2:T
    y(t,1) = rho*y(t-1,1) + randn;
end
```

Check command `randn` **!!**

For and While Loops

- Here `t` is the **iterator variable**.
- The **range of the loop** is given by the expression `2:T`
- So we know `t` is going to take values `2,3,4,5,6...`
- Why do we do `y = zeros(100,1)` outside of the loop?

```
T = 100;
rho = 0.85;
y = zeros(100,1);
for t=2:T
    y(t,1) = rho*y(t-1,1) + randn;
end
```

For and While Loops

Preallocation:

- We want to store the values for y.
- And we know the size of this vector.
- We could start with `y[1]` and expand it with each iteration.
- A better method is to **preallocate** y.

```
T = 100;
rho = 0.85;
y = zeros(100,1);
for t=2:T
    y(t,1) = rho*y(t-1,1) +
        randn;
end
```

For and While Loops

- Suppose we want to add one to a number for as long as it remains below a threshold. We can use `while` loops!

- We need a **condition** to be satisfied.

- Similar to an `if` statement. If it is true, evaluate. Otherwise stop.

- **Caution:** the condition must become false, otherwise we get an infinite loop.

```
a = 0;
while a < 25
    a = a + 1;
end
```

An Example with Grades

Exercise 6

Suppose we have a list of grades of students. Create a loop that goes through all the notes and checks whether that student has passed the subject or not. To get the list, generate a random list of 200 grades uniformly distributed between 0 and 10 (check `rand` command), and to assign if a student has passed or not, generate a vector called `passed` that equals one if the student has obtained a grade larger or equal than 5, and 0 otherwise.

An Example with Grades

```
% === Ex. 5 List of students === %
nstudents = 200;
grades = 10.*rand(nstudents,1);
pass = ones(nstudents,1);

for student = 1:nstudents
    if grades(student,1) >= 5
        pass(student,1) = 1;
    else
        pass(student,1) = 0;
    end
end
```

An Example with Grades — A More Efficient Approach

The previous example was perfectly correct, but we could improve performance by *vectorizing* the operations.

```
pass_vect = zeros(nstudents,1);
pass_vect(grades >= 5) = 1;

% To test they are equal
isequal(pass,pass_vect)
```

Vectorizing is **extremely important**. In this simple example, the vectorized function takes $\approx$ 18% of the time it takes for the loop.

Simple Plots

Simple Plots

- Matlab has a powerful command to generate figures `plot`.
- To plot the $AR(1)$ process we simulated before, is as easy as

```
figure
plot(1:T,y)
```

- Command `figure` opens a clear figure window.
- `plot` takes as first argument the $x-$axis vector (the time periods) and the value of y_t as second argument.

A More Involved Example

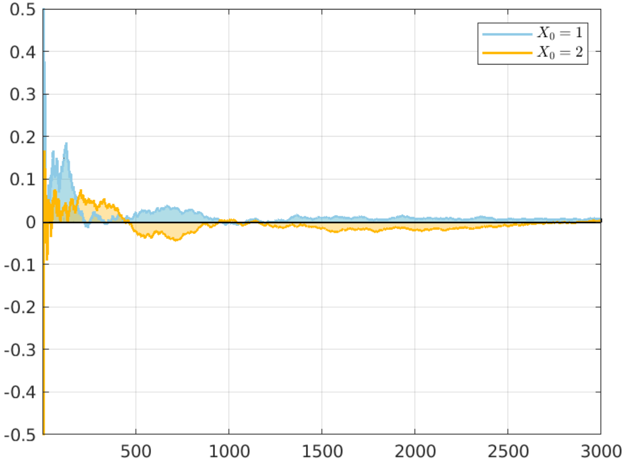

Figure 2: A Model of Unemployment Dynamics

Functions

User Defined Functions

- Commands are functions that take inputs and yield a result.
- In Matlab we can create our own functions just like in the mathematical sense.
- If $f(x) = 3x + 1$ then we can plug any x and the result is $3x + 1$. This function in Matlab would be

```
function [fx] = simple_function(x)
    fx = 3*x + 1;
end
```

User Defined Functions

- The general structure of a function is

```
function [out1,out2,...,outN] = name(in1,in2,...,inN)
    % Document the function. Author, inputs, outputs...
    Operations
    out1 = %operations to get out1;
    ...
    outN = %operations to get outN;
end
```

- Save the function as an `.m` file in the working directory (or add to path).

- The name of the file **must be** the name you assigned.

Exercise 7

Create a function called `my_wave` that gives as output the plot of a sinusoidal wave. The function should take as arguments the parameters that will give the amplitude, the frequency, and the upper and lower bounds in which it will be plotted. By default, plot 1000 points. Plot the sinusoidal wave with amplitude and frequency one for comparison. A sinusoidal wave W with amplitude A and frequency f is computed as

$$W = A\sin(2\pi f x)$$

User Defined Functions

A general function to compute a sine wave

```
function [wave] = my_wave(A,freq,lb,ub)
    points = 1000;
    x = linspace(lb,ub,points);
    wave = A.*sin(2.*pi.*freq.*x);

    figure
    plot(x,wave,'--','LineWidth',1.3)
    hold on
    grid on
    plot(x,sin(x),'-','LineWidth',1.3)
    legend({'$A\sin(\omega x)$','$\sin(x)$'},'Interpreter','latex','Location','best')
    xlabel('$x$','Interpreter','latex')
    title(['Sinusoidal wave with amplitude ', num2str(A), ' and frequency ', num2str(freq)])
end
```

Anonymous Functions

- Anonymous functions are functions *not stored in an `.m` file*.
- These functions work within a script and they are stored in the workspace as `function_handle` objects.
- To create an anonymous function that multiplies a number by two:

```
% Create the function
bytwo = @(x) 2*x;
% Call it
a = bytwo(10);
```

Anonymous Functions

- Note that these functions can contain only **one** computable statement.
- Typically:
  - Used for functions in the mathematical sense.
  - Simple.
  - Can take multiple inputs.

Anonymous Functions

Exercise 8

Write the Rosenbrock function as an anonymous function and compute the value of $f(1,1)$, $f(0,0)$, $f(1,0)$, and $f(0,1)$. The Rosenbrock function is given by:

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$