Modeling Mathematics ENG4xxx

MatLab: A quick guide

example-driven

FLORIMOND GUENIAT

BIRMINGHAM CITY University

Chapter 1

Quick tour `MatLab`

The objective of these tutorial are to illustrate the ENG 4XXX courses as well at to help you to learn quickly how to numerically solve problems.

It will hence provide to the reader the first concepts, not only behind `MatLab`, but as well on how to code and how to solve problems.

The emphasis here is learning by doing. Therefore, the reader should not to read these documents without a computer close-by.

Legal stuff

`MatLab` is a registered trademark of MathWorks, Inc.

I About `MatLab`

I a) What is the use of `MatLab`?

`MatLab` (for MATrix LABoratory) aims at delivering quickly some results on a user-defined problem.

It shines, as expected from the name, when it involves linear algebra, i.e., operations on matrices. `MatLab` makes the manipulaiton of matrices really easy, as it will hopefully been demonstrated through these notes.

`MatLab` has several main advantages compared to language like C/C++ or Fortran are:

- No compilation
  a script can be executed directly.
- The prompt
  results can be analyzed right away.
- Simplicity
  the learning curve is low
- Portability a script will work on any `MatLab`, and on any platform: Linux, Mac or Windows.
- Built-in functions
  - Integration
    solving equations is *relatively* easy
  - Visualization
    plotting the results in one command

- Tool-box
  many tools dedicated to problems already exist
- Data-Analysis
  basic and advanced tools for data-analysis and machine learning are already implemented

The negative points are mostly:

- Sub performance
  No compilation means less efficiency
- Not open source
  The results can not easily be checked
- Price It can be up to 1800£

I b) Equivalent of `MatLab`

Octave and SciLab are almost identical to `MatLab`. A `MatLab` script would work on these two others open-source and free softwares.

Most of the tips can also be applied to python, especially when the packages scipy and numpy (for scientific and engineering computations) are used.

II Hello, World!

Let's print in the console the "Hello, World!".

> **Pro Tip**
>
> When starting to learn a programming language, trying to make a program that print "hello, World!" is usually a good idea.
> It will show the basics of:
>
> - installing the language/program
> - the syntax of the language
> - running the language/program

II a) Starting with `MatLab`

II a) i Launch `MatLab`

Click on the icon, duh! Fig. 1.1 represents the icon.

II a) ii Organization of the typical `MatLab` window

When clicking on the icon, `MatLab` will start. The main window opens, see Fig. 1.2. It is separated in a few important sections:

- the command windows
  This is the command prompt
- current folder
  It lists the files
- the Workspace
  It gives details on the objects present in memory

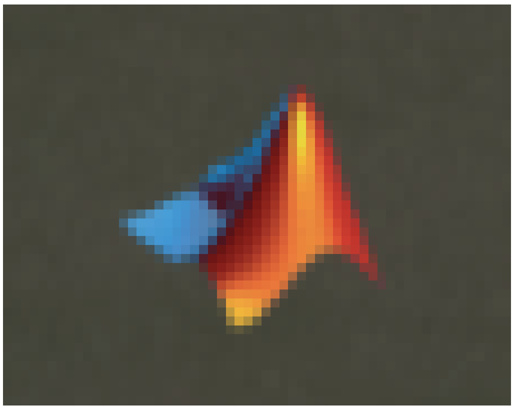


Figure 1.1: The MatLab icon (in super big).

- the editor

  this is where you can write a script

- the ribbon

  It gives access to properties, functions, editor, etc. Similar in spirit to Words and Excel.

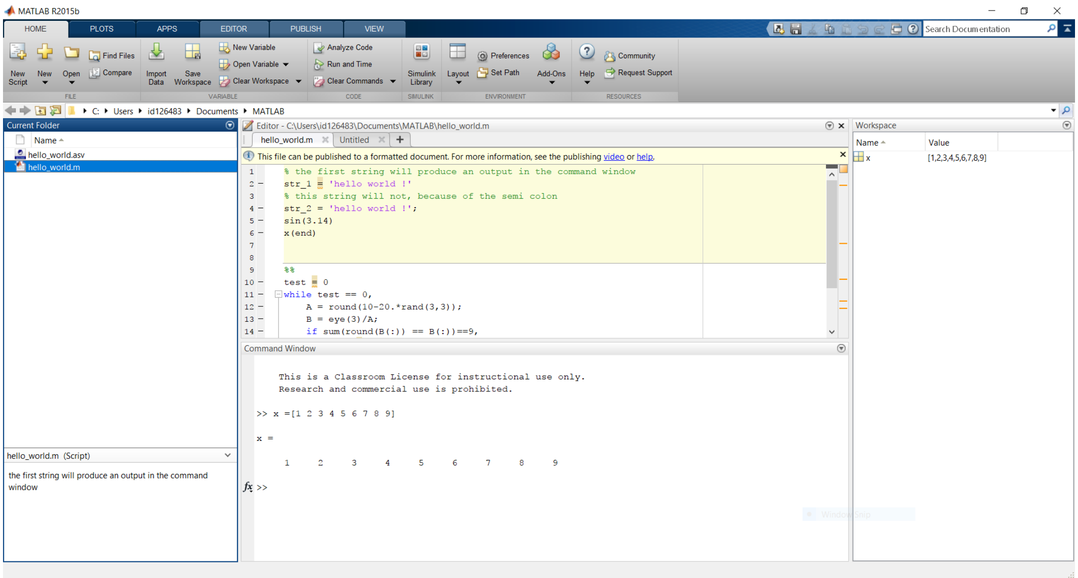


Figure 1.2: MatLab just after being opened.

II b) How to print "hello world"

Click on the Command Window, and type "hello world". You will see:

MatLab printed the 'Hello World', congratulations !

One line was enough to get the desired output. This shows that MatLab is very efficient at getting quickly some results.

In the following, instead of print screen, we will show the results and the commands as:

```
>> 'Hello World'
```

Figure 1.3: How to print 'hello world'.

```
ans =
Hello World
```

'Hello World' has been assigned to a variable named `ans`.

Definition

VARIABLE: A variable is essentially a name that is associated with a value.
Values can be of several types:

- results, such as string, numbers or matrices: `x = 3.`
- functions, for instance `sin` is a built-in function
- complex objects, for instance, a plot

They are usually assigned with the sign "="

Pro Tip

`ans` is short for answer.
With `MatLab`, the results of the command is always stored in the variable `ans`, except if it is assigned to a given variable. Consequently, the command `1+1` will affect the variable `ans`, but `x=1+1` will not, and 2 will be assign to `x`.
`ans` can be re-used in the prompt: `x = ans+1`! However, a good practice is to assign the result to a user-defined variable.

Trying without the quotes leads to:

```
>> Hello World
Undefined function or variable 'Hello'.
```

plus some help.

Hello World is understood by `MatLab` as a function/variable and then an option for this function. `MatLab` hence thinks that Hello is something that already exists ; it is not the case here. As a consequence, an error follows.

The main reason behind that is that the goal here is to print a string.

Definition

STRING: A string is a chain of characters. It is *not* a number.
It has to be between quotes: 'some text' or double quotes: "some text". For example, 'Lorem ipsum dolor sit amet' is a chain or characters.
But it is not that easy. `s = '45'` is the chain of characters '4' and '5'. But `n = 45` is the number 45. `s` and `n` are different.

Pro Tip

How to put a quote in a string ? For instance,
Having `s ='nah, can't do'` is not obvious, as `MatLab` will interprete the quote in "can't" as the end of the string.
To solve the issue, double it ! `s = 'nah, can''t do` will work perfectly.

Let's try now to add a semi colon at the end of the line:

```
>> 'Hello World';
>>
```

Nothing is printed in the command prompt.

Pro Tip

Do not forget the semi-colon ";" at the end of lines !
It is not a big deal when dealing with small matrices and small vectors. But when an image is being manipulated, it means that `MatLab` is manipulated a matrix with dimension around 1000×1000. Forgetting the ";" sign means that `MatLab` will show around a *million* numbers every line of a script!

Definition

COMMAND PROMPT: The command prompt is the >> sign. Command Prompt is a command line interpreter. It is used to execute entered commands. Once enter is hit, `MatLab` will interpret the command, and send back any results.

One of the most important tip to remember: `MatLab` will always print the result of a line if it does not have a ";" at the end of the line.

III `MatLab` as a calculator

`MatLab` can be used as a calculator. The prompt allows to interact directly with variables, quantity, and to do computations with them.

III a) Algebra

After clicking on the prompt, let's try some simple calculations:

```
>> 4+3
ans =
7
>> 4*3
ans =
12
```

It behaves as a calculator would.

`MatLab` respects the BODMAS (Brackets, Order, Division/Multiplication, Addition/Subtraction).

Definition

BODMAS: It stands for Brackets, Order, Division/Multiplication, Addition/Subtraction.

1. start with resolving inside of the brackets:
   $3 \times (4+2) \times 4^2 + 1 = 3 \times 6 \times 4^2 + 1$
2. then resolve orders (powers, roots) :
   $3 \times 6 \times 4^2 + 1 = 3 \times 6 \times 16 + 1.$
3. then resolve division/multiplication :
   $3 \times 6 \times 16 + 1 = 288$
4. then finish with addition/substraction:
   $288 + 1 = 289$

Let's try a few different operations !

```
>> (4+3)*2
ans =
14
>> 4+3*2
ans =
10
```

III b) Details on variables

III b) i `ans`

As seen before, `ans` can be used to store a result, but it will be overwritten every time a command is executed:

```
>> 2+2
ans =
4
>> ans+2
ans =
6
>> ans+2
ans =
8
```

III b) ii Creation and re-assignement

Variable can be easily created and assigned with the sign "=".

```
>> x = 2+2
x =
4
>> x+2
ans =
6
>> x*5
ans =
10
```

III b) iii Naming convention

A variable name can be anything, such as `goodnameforavariable` or `GoodNameForAVariable`, or `good_name_for_a_variable`. However:

- it cannot start with _
- it cannot start with a number
- a few names are protected

Pro Tip

Try to use clever name for variables, it will help to understand the code.
If all the results are named `result_1,result_2,result_3`, it is hard to know what they should contain.
On the contrary, when dealing with the variable `name_city`, it is expected to be a string and having a proper name.
In a similar way, the variable `motor_freq` probably contains a number.
Also, if the piece of code uses the variable `price_pond`, the variable `price_dollar` and the variable `rate_dollar2pound`, them the line `price_pound = rate_dollar2pond * price_dollar` is pretty explicit. If the variables where instead named `x,y,z`, then the line `x = y*z` is much more cryptic.

The choice of a name is important! For instance, `x` is good for an unknown, `s` if its value is expected to be a string, `v` if it is a vector... More complex names can be used, such as `x_problem_1`.

Try to be consistent thorough the piece of code !

A few tips:

- Use different names for different results
- Use a name that is meaningfull (e.g. `str_name` if the variable is assigned with a chain of character that is a name)
- Consequently, avoid unecessary use of index (e.g. `result_1, result_2` etc.)

Pro Tip

Many naming convention exist.
However, the following name convention is fine:

- `UpperCamelCase` for functions: `MyFunction`
- `CAPITALIZED_WITH_UNDERSCORES` for constants `Pi=3.14`
- `lowercase_separated_by_underscores` for other variables `name_of_univ = 'BCU'`.

III b) iv Reassignement

Updating a variable is handy: one might want to change the variable `year` from 2017 to 2018.

A variable can be easily updated, by reassigning a new value to it. It hence uses the sign "=". For instance:

```
>> x = 2+2
x =
4
>> x = x + 5
```

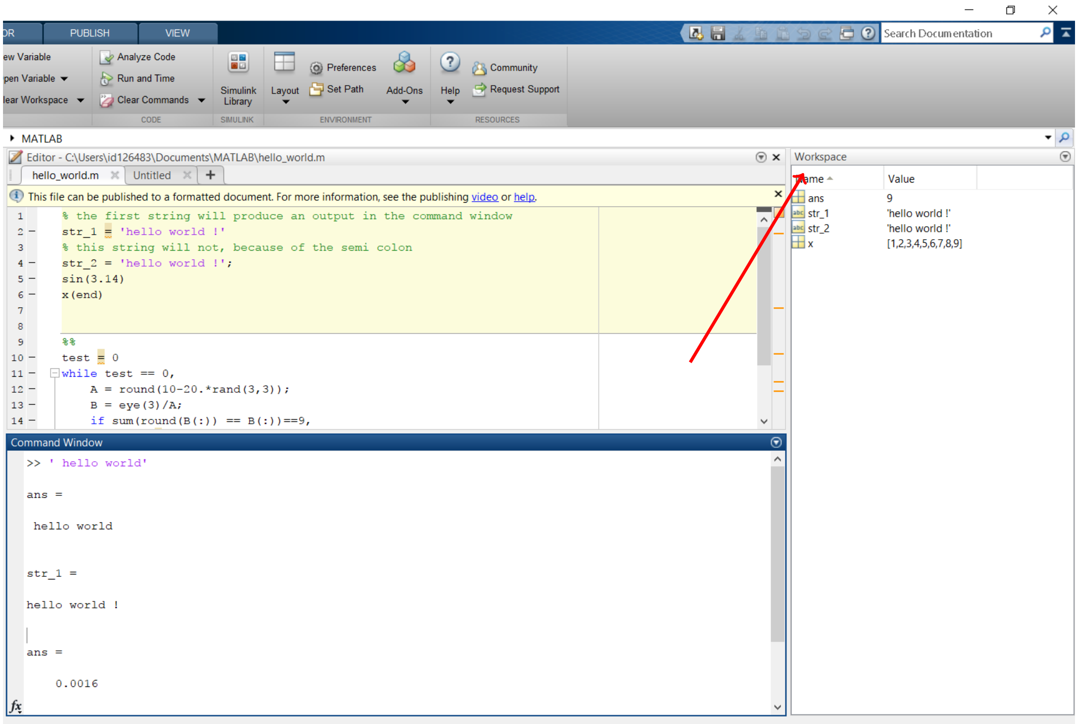


Figure 1.4: The workspace in `MatLab`. Here are the variables `ans`, `str_1`, `str_2` and `x` as well as their contents.

```
x =
9
>> x = 0
x =
0
```

III c) Workspace

When a variable is created, it is available in the *workspace*. It is the area (usually) on the right. It allows to:

- show what variables are currently known to `MatLab`
- know what is present in the memory
- indicate what there is in the variables
- eventually modify the content of a variable

III d) Entering multiple commands per line

It is possible to enter multiple commands per line. Use commas "," or semicolons ";" for that ; the commas will *not* suppress the outputs.

Pro Tip

Try to avoid multiple commands per line.
Most of the time, it makes the code harder to read, especially if there is not a good reason to do so.
Nevertheless, it can make sense write a few commands per line when assigning a variables that are related.

```
>> x = 2 ; y = 3 ; z = 4 ;
>> x = 2 ; y = 3 , z = 4 ;
y =
3
>> x = 2 , y = 3 , z = 4 ,
x =
2
y =
3
z =
5
```

III e) Basic arithmetic

Basic arithmetic operators are pretty classic, and be found in Tab. 1.1:

Table 1.1: Arithmetic operators

operation	command	exemple
addition	+	3+4
soustraction	-	3-4
multiplication	*	3*4
division	/	3/4
power	^	2 ^ 4

III f) Built-in functions

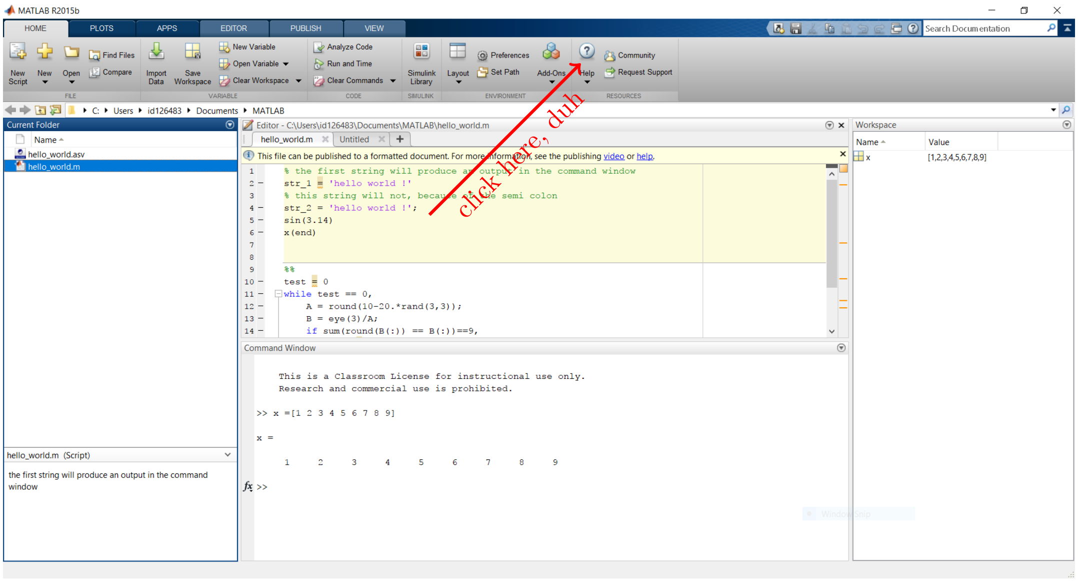
One of the strengths of `MatLab` is that many functions are already available. One can think of common functions like sine or exponential, but `MatLab` also provides more complex functions like **`imread`**, that will import pictures, or **`svd`**, that will do some modal decomposition of a table of data.

III f) i How to find a function or a command ?

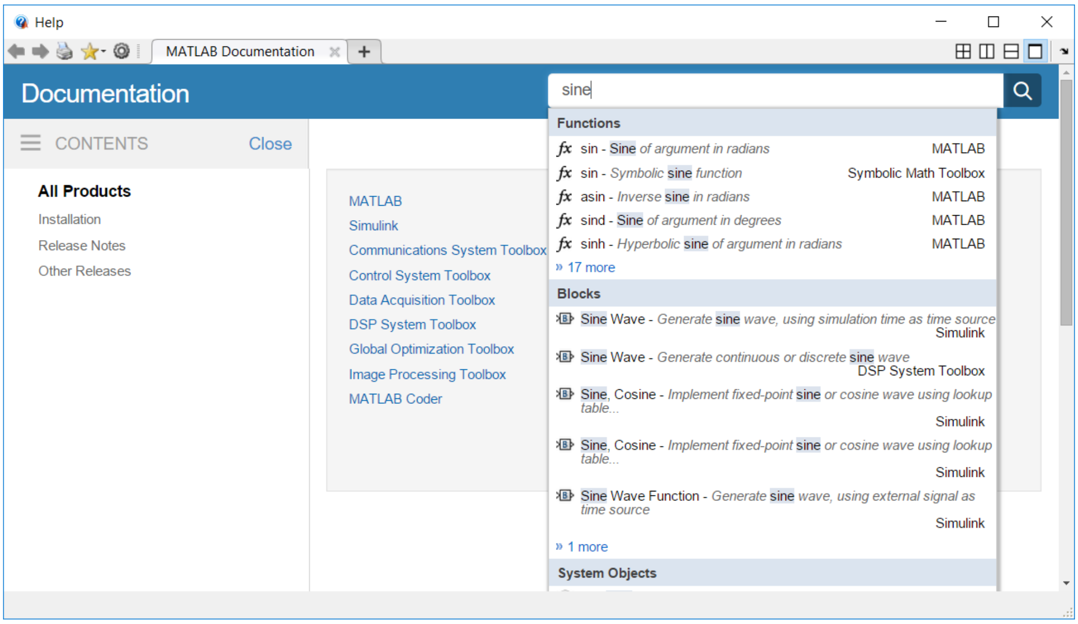
When looking for something, hit the help button. For instance, if one wants to look for the sine function:

Pro Tip

Use the help! It is *very* useful and one will mostly find any function/tool/infos that is needed.
Usually, the help contains a few examples. Do not hesitate to read them carefully, and to try them. They will help understanding how to use the functions and properties of `MatLab`.
There is also a "See Also" section that can be useful when looking for a particular topic.



(a)



(b)

Figure 1.5: Looking for sine in the help. a): where the help is. b): how to use it.

III f) ii Using a function

Calling a function is relatively easy and intuitive. Let's take the sine function as an illustration.

```
>> sin(3.14)
ans =
0.0016
```

Help

Go and look for SIN in the help for details

`MatLab` is asked to evaluate the function `sin` in $3.14 \approx \pi$. For that, the argument just has to be provided between the parenthesis.

Pro Tip

Trigonometric functions in `MatLab` are in radiant. `sin(360)` is hence different from 0 but rather close to 0.96.
In a similar spirit, `log` in `MatLab` is the natural logarithm, and not the log in base 10.

Typical functions are available with somewhat explicit names, see Tab. 1.2. Similarly, many useful constants for the engineer are implemented in `MatLab`, see Tab. 1.3.

Table 1.2: A few function names in `MatLab`. Many others are already implemented in `MatLab`.

Trigonometry	name	Stats	name	Misc.	name
sine	sin	mean	mean	square root	sqrt
cosine	cos	maximum	max	absolute value	abs
exponential	exp	minimum	min	round up	ceil
natural logarithm	log	standard dev.	std	conjugate	conj

Table 1.3: A few useful constant names in `MatLab`.

$\pi \approx 3.14$	`pi`
$i = \sqrt{-1}$	`i`
$i = \sqrt{-1}$	`j`
∞	`Inf`
Not a Number	`NaN`

IV Let's solve a real problem

Problem Julie's car's odometer reading was 35201km when she last filled the fuel tank. Yesterday she checked it again and it read 35403km. Checking the tank, the car used 11 liters of gas to do so. If her car's gas tank holds 35 liters, how long can she drive before running out of gas?

Solution using `MatLab` as a calculator How much has she driven ?

```
>> 35403 - 35201
ans =
202
```

How much distance per liter of gas ?

```
>> 202/11
ans =
18.3636
```

How much gas is left in the tank ?

```
>> 35 - 11
ans =
24
```

Distance she can drive:

```
>> 24 * 18.3636
ans =
440.7264
```

Solution with variables The same code can be scripted, see Chap. 4, Sec. I. It means that we would just have to replace the value in the first three lines to update the whole code.

```
>> odormeter_before = 35201 ;
>> odormeter_after = 35403 ;
>> used_gas = 11 ;
>> distance = odormeter_after - odormeter_before ;
>> distance_per_liter = distance/used_gas ;
>> tank_capacity = 35;
>> estimated_distance = (tank_capacity - used_gas) *
    distance_per_liter
440.7264
```

V What to remember

`MatLab` is basically a powerful calculator.

V a) Key example

Let's summarize in a few examples everything we have seen.

```
>> year = 2018;
>> next_year = year + 1
next_year =
2019
>> angle = pi ;
>> ( 2.*cos(angle) + 1.) ^2
ans =
9.0
```

VI Exercices

VI a) On Variables

1. Create the variables `x,y,z` assigned with 1, 2 and 3.
2. Create the variable `sum_xyz` that is the sum of `x,y` and `z`.
3. Propose a name for a variable that is assigned as a value `'Birmingham'`

4. Propose a name for a variable that is assigned as a value `'BCU'`
5. Try to assign to the variable `year` the value `2017`, and then to `2018`!
6. Try to assign to the variable `girlfriend_name` the value `'Adilah'` (using the sign equal, pun totally intended), and to the variable `ex_girlfriend_name` the value `'Marie'`. Then, reassign to the variable `girlfriend_name` the value `'Kiara'`, and to the variable `ex_girlfriend_name` the value `'Adilah'`.

Chapter 2

Fundamentals on Matrices and Vectors

Linear algebra is the fundations of `MatLab`, and what makes it popular. `MatLab` is *designed* to work with matrices.

In particular, `MatLab` makes the manipulation of matrices and vectors very easy. This section will show how to do:

- operations involving vectors
- operations involving matrices
- operations involving both arrays and matrices

I Basics with matrices and vectors

Definition

MATRIX: The most basic variable in `MatLab` is the matrix.
It is a two-dimensional, rectangularly shaped table. It means it has dimensions, for instance $n \times m$, meaning that it is a table of n rows and m columns. This table will store mostly numbers, but can store other types of data.
Even a number, like 8.1, is store in `MatLab` under a matrix form, as a 1×1 matrix. A vector is a matrix where one of the dimension is 1.

Help

Go and look for MATRICES AND ARRAYS in the help for details

I a) Creation

Follow the procedure:

- start with a bracket [
- write each element of a column, separated with space or commas
- separate rows with a semi-colon
- end with the bracket]

For creating a vector:

```
>> v = [1,2,3]
v =
1 2 3
>> w = [0;1]
w =
0
1
```

`v` is a row vector and `w` is a column vector. For creating a matrix, the following lines are equivalent:

```
>> M =[1,0 ; 0,1];
M =
1 0
0 1
>> m =[1,0 ; 0 1];
M =
1 0
0 1
>> m =[ [1,0] ; [0 1]];
M =
1 0
0 1
```

The last line shows that a matrix is virtually stacked vectors.

I b) Manipulation

You can

- add, subtract, multiply matrices if their sizes are compatible
- access to the ith element of matrix `M` using `M(i)`
- access to the (ith,jth) element of matrix `M` using `M(i,j)`

To add a constant to a vector, a vector to another vector (it works identically for matrices) :

```
>> v = [1 3];
>> v + 4
ans =
5 7
>> v*2.
ans =
2 6
>> w = [2,8];
>> v+w
ans =
3 11
>> w/2.
ans =
1 4
```

To access to an element of a vector (it works identically for matrices) :

```
>> v = [1 3 4 -2.5 8] ;
>> v(1)
ans =
1
```

```
>> v(3)
ans =
4
>> M = [[5,2];[9,-1]] ;
>> M(1)
ans =
5 2
>> M(1,1)
ans =
5
>> M(2,1)
ans =
9
```

To delete or *remove* the ith element, use the operator "[]"

```
>> v = [1 3 4 -2.5 8] ;
>> v(1) = []
v =
3 4 -2.5 8
```

It hence reduces the dimension of the vector. It works similarly for a matrix:

```
>> A = [ [1 3 4] ; [-2.5 8 9]] ;
>> A(:,1) = []
A =
3 4
8 9
```

The first column has been deleted.

II More details on vectors

II a) Creation of a vector

II a) i Row vector

Creating a vector `v` is easy. All the components just have to be put between brackets.

```
>> v = [1,2,3]
v =
1 2 3
```

It is also possible to replace the commas with space:

```
>> v = [1 2 3]
v =
1 2 3
```

It works fine but makes the code harder to read.

Let's create Cartesian vectors in dimension two. $e_x = (1, 0)$ and $e_y = (0, 1)$:

```
>> ex = [1 0]
ex =
1 0
>> ey = [0,1]
ey =
0 1
```

II a) ii Column vector

Creating a column vector is similar to creating a row vector, except that the element are separated with a semi-colon ";".

```
>> v = [1;0;4]
v =
1
0
4
>> w = [0;1]
w =
0
1
```

II a) iii Transpose operator

It is possible to change a column vector to a row vector, and reciprocally, by using the transpose operator "'".

```
>> ex = [1;0]
ex =
1
0
>> ex'
ans =
1 0
>> ey = [0,1]
ey =
0 1
>> ey'
ans =
0
1
```

Definition

TRANSPOSE: The transpose `M'` has the same elements as `M`, but the rows of `M'` are the columns of `M` and the columns of `M'` are the rows of `M`. As a consequence, if `M` is a $n \times m$ matrix, `M'` is an $m \times n$ matrix.
With complex elements, `MatLab` actually send back the conjugate transpose.

II b) Access to the elements of a vector

II b) i Access to one element

The first element of a vector `v` is `v(1)`. The second element is `v(2)`, and so forth.

Accessing the element of a vector is just calling the vector with specifying the desired element:

```
>> x = [1 3 4 -2.5 8]
x =
1 3 4 -2.5 8
>> x(1)
ans =
1
>> x(3)
```

```
ans =
4
>> x(4)
ans =
-2.5
```

The last element of a vector can be called using the argument `end`: One can also call the ith item from the end using `end-i`.

```
>> x = [1 3 4 -2.5 8]
x =
1.000 3.000 4.000 -2.5 8.000
>> x(end)
ans =
8
>> x(end-1)
ans =
-2.5
>> x(end-2)
ans =
4
```

Definition

END: `end` is a very powerful operator in `MatLab`.
It moslty serves two purposes.

- `end` gives access the last element of a matrix or vector. If `v=[1,4,2]`, `v(end)` send back 2.
- That will be detailed later, but `end` also terminates statements that have a scope (for instance, `for, while, switch, try, if` and functions).

II b) ii Access to several elements

The operator ":" gives access to all the elements between the first and the last element (included), in a column vector:

```
>> v = [1 3 4 -2.5 8];
>> v(:)
ans =
1
3
4
-2.5
8
```

Accessing to all the elements between the second and fifth element of `v` is `v(2:5)`:

```
>> v = [1 3 4 -2.5 8 12];
>> v(2:5)
ans =
3
4
-2.5
8
```

To access to *all* the elements between the first and the last element (included), in a column vector, simply use the colon operator:

```
>> x = [1 3 4 -2.5 8];
>> x(:)
ans =
1
3
4
-2.5
8
```

Definition

:-COLON OPERATOR: The colon operator is pivotal in `MatLab`.
It is noticeably useful to

- generate lists: `x= 1:20`
- controlling loops (more details in Chap. 4): `for i=1:20`
- transform a matrix in a column vector `M(:)`
- extract sub-parts of vectors/matrices `M(2:4,1:2:8)`

It will be seen in more details in Sec. IV b) i.

II b) iii Deleting elements

In `MatLab`, using "[]" empty a variable. It is named the *empty vector operator*.

When used on a part of a vector or a matrix, it simply *deletes* this part. It is *gone*. As a consequence, it reduces the dimensions of the matrix.

```
>> v = [1 3 4 -2.5 8] ;
>> v(1) = []
v =
3 4 -2.5 8
```

II b) iv Adding elements (concatenation)

For creating a matrix, we have seen that one can write:

```
>> M =[ [1,0] ; [0 1]];
M =
1 0
0 1
```

It means that vectors can be used to construct a matrix:

```
>> v_1 = [1,0] ; v_2 = [0 1]];
>> M = [v1;v2]
M =
1 0
0 1
```

This operation is called concatenation.

Definition

CONCATENATION: Matrix concatenation is the process of joining one or more matrices to make a new matrix.
The brackets [] operator is also used as the concatenation operator.
It works in a similar fashion of the creation of a matrix:

- `C = [A B]` horizontally concatenates matrices `A` and `B`.
- `C = [A; B]` vertically concatenate matrices `A` and `B`.

It can also be done to extend a vector:

```
>> v_1 = [1,0] ; v_2 = [0 1];
>> v = [v1,v2]
v =
1 0 0 1
```

II c) Basic operations on vectors

II c) i Addition/subtraction

It is easy to add or subtract a given value to *all* the components of a vector, using the signs "+" and "-".

```
>> x = [1 3]
x =
1 3
>> x + 4
ans =
5 7
>> y = x - 2
y =
-1 1
>> z = [5 10 -1 8] + 3
z =
8 13 2 11
```

Vectors can be added, as long as their dimensions correspond:

```
>> ex = [1 0]; ey = [0,1] ;
>> ex + ey
ans =
1 1
>> ex - ey
ans =
1 -1
```

Of course, if their dimensions do not correspond, `MatLab` will send back and error:

```
>> x =[1,0] ; y = [1,0,0];
>> x+y
Matrix dimensions must agree.
```

Pro Tip

`MatLab` considers vectors as 1D matrices.
It is important when dealing with multiplication.
If `v=[1,2,3]`:

- `v*v'` is 14
  it is multiplying a 1×3 matrix with a 3×1 matrix.
- `v'*v` is `[[1,2,3];[2,4,6];[3,6,9]]`
  it is multiplying a 3×1 matrix with a 1×3 matrix.
- `v*v` will not work
  it is multiplying a 1×1 matrix with a 1×3 matrix; dimensions are not compatible.

II c) ii Multiplication

II c) ii 1 Multiplication by a scalar It is easy to multiply or divide by a given value to *all* the components of a vector, using the signs "*" and "/"

```
>> x = [1  3]
x =
1  3
>> x * 4
ans =
4  12
>> y = x / 2.
y =
0.5  1.5
>> z = 3 * [5  10 -1  8]
z =
15  30  -3  24
```

II c) ii 2 dot product It can be done using the function `dot`.

```
>> x =[1,0] ; y = [1,0];
>> dot(x,y)
ans =
1
>> x(1) * y(1) + x(2) * y(2)
ans =
1
>> a =[0,0.5,2] ; b = [2,0,4];
>> dot(a,b)
ans =
8
>> a(1) * b(1) + a(2) * b(2) + a(3) * b(3)
and =
8
```

II c) ii 3 Element-wise multiplication Element-wise multiplication means that each element of a vector is multiply by the corresponding element of the other vector. It is similar to the dot product *except* for the sum. The operator for that is ".*" (it is read "dot product", which is pretty stupid when you think about it!).

```
>> x =[1,0] ; y = [1,0];
>> x.*y
ans =
1 0
>> [x(1) * y(1), x(2) * y(2)]
ans =
1 0
>> a =[0,0.5,2] ; b = [2,0,4];
>> a.*b
ans =
0 0 8
>> [a(1) * b(1), a(2) * b(2), a(3) * b(3)]
and =
0 0 8
```

Pro Tip

In `MatLab`, using "." in front of an operator means that this operator will be applied element-wise (to each element of the vector/matrix).
For instance, `v.^2` means that *all* the elements of `v` will be squared. If `v` is $[2, 4, 3]$, then `v.^2` is $[2^2, 4^2, 3^2]$, or `[4,16,9]`.

[[TODO exercices]]

III More details on matrices

`MatLab` sees vectors a line matrices. Building a matrix is the equivalent of stacking lines. For that, `MatLab` uses the semi-colon sign ";". The following lines are equivalent:

```
>> M =[1,0 ; 0,1];
M =
1 0
0 1
>> m =[1,0 ; 0 1];
M =
1 0
0 1
>> m =[ [1,0] ; [0 1]];
M =
1 0
0 1
```

The last line shows that a matrix is virtually stacked vectors.

III a) Addition/subtraction

It is easy to add or subtract a given value to *all* the components of a matrix, using the signs "+" and "-".

```
>> M = [[1 3];[2,4];
>> M + 4
ans =
5 7
6 8
```

```
>> A = M - 2
A =
3 5
4 6
```

Matrices can be added, as long as their dimensions correspond:

```
>> M = [[1  0];[2 ,3];  A = [[2 , -1];[1 ,1]];
>> M + A
ans =
3 -1
3  4
>> M - A
ans =
1 1
1 2
```

Of course, if their dimensions do not correspond, `MatLab` will send back and error:

```
>> M =[[1 ,0];[1 ,0];[1 ,0]  ; A = [[2 , -1];[1 ,1]];
>> x+y
Matrix dimensions must agree.
```

III b) Multiplication

It is easy to multiply or divide by a given value to *all* the components of a vector, using the signs "*" and "/"

```
>> x = [1  3]
x =
1 3
>> x * 4
ans =
4 12
>> y = x / 2.
y =
0.5  1.5
>> z = 3 * [5  10 -1  8]
z =
15 30 -3 24
```

How `MatLab` deals with multiplication for vectors ? Two options can be proposed.

III b) i dot product

The first option is the dot product between two vectors.

Definition

DOT PRODUCT: The dot product (or inner product) of two vectors is the sum of the multiplication of their components. If $u = (u_i), v = (v_i)$ are n-dimensional vectors, $< u, v >= \sum_{i=1}^{n} u_i \times v_i$.

It can be done using the function `dot`.

```
>> x =[1,0]  ; y = [1 ,0];
>> dot(x,y)
ans =
```

```
1
>> x(1) * y(1) + x(2) * y(2)
ans =
1
>> a =[0,0.5,2] ; b = [2,0,4];
>> dot(a,b)
ans =
8
>> a(1) * b(1) + a(2) * b(2) + a(3) * b(3)
and =
8
```

Help

Go and look for DOT in the help for details

III b) ii Element-wise multiplication

Another option could be the element wise multiplication. It means that each element of a vector is multiply by the corresponding element of the other vector. It is somehow similar to the dot product *except* for the sum. The operator for element-wise multiplication is ".*" (it is read "dot product", which is pretty stupid when you think about it!).

```
>> x =[1,0] ; y = [1,0];
>> x.*y
ans =
1 0
>> [x(1) * y(1), x(2) * y(2)]
ans =
1 0
>> a =[0,0.5,2] ; b = [2,0,4];
>> a.*b
ans =
0 0 8
>> [a(1) * b(1), a(2) * b(2), a(3) * b(3)]
and =
0 0 8
```

[[TODO exercises]]

Try to create the vector $x = (1, 2, 3, 4)$.

III b) iii Accessing and deleting elements

III b) iii 1 Accessing elements It works in a similar way as for vectors, except that the dimensions are separated with a comma. Let's work with `M`:

```
>> M = [ [1,2,3]; [4,5,6]; [7,8,9]]
M =
1 2 3
4 5 6
7 8 9
```

To extract the third column of `M`:

```
>> c = M(:,3)
c =
3
6
9
```

To extract the second row of M:

```
>> l = M(2,:)
l =
4 5 6
```

To extract the first *and* the second row of M:

```
>> A = M([1,2],:)
A =
1 2 3
4 5 6
```

Following this notation, we can exchange the first rows of M:

```
>> A = M([2,1],:)
A =
4 5 6
1 2 3
```

Pro Tip

Using lists (for instance `indices = [1,3,4]`) as arguments of matrices is a very powerful methods for altering a matrix or for creating a new one.

- `M(indices,:)` will select the rows of `M` in the order of the elements of `indices`. They can be copied, if there is repetition in `indices`, as `indices = [1,2,2]` They can be rearranged, if the elements in `indices` are not ranked, as `indices = [3,1,2]`
- `M(:,indices)` will select the rows of `M` in the order of the elements of indices
- `M(indices_row,indices_col)` will extract a sub-matrix of `M`

III b) iii 2 Deleting elements In `MatLab`, using "[]" empty a variable. It is named the *empty vector operator.*

When used on a part of a vector or a matrix, it simply *deletes* this part. It is *gone*. As a consequence, it reduces the dimensions of the matrix.

It hence reduces the dimension of the vector. It works similarly for a matrix:

```
>> A = [ [1 3 4] ; [-2.5 8 9]] ;
>> A(:,1) = []
A =
3 4
8 9
```

The first column has been deleted.

It is possible to delete one element from a vector

```
>> v = [1 3 4 -2.5 8] ;
>> v(1) = []
v =
3 4 -2.5 8
```

But trying to do it for a matrix will lead to an error message.

```
>> A = [ [1 3 4] ; [-2.5 8 9]] ;
>> A(1,1) = []
A null assignment can have only one non-colon index.
```

It is easy to remove a block, a column or a row from a matrix. But a single element can not be removed: there would be a "hole".

IV Useful built-in functions for vectors and matrices

IV a) Dimensions of a matrix

The command `size` send back the dimensions of a matrix.

```
>> A = [ [1 3 4] ; [-2.5 8 9]] ; v = [1 2 3 4];
>> size(A)
ans =
2 3
>> size(A(:))
ans =
6 1
>> size(A(1,:))
ans =
1 3
>> size(v)
ans =
1 4
>> size(v')
ans =
4 1
```

To reuse the size, it can be stored it in variables:

```
>> A = [ [1 3 4] ; [-2.5 8 9]] ;
>> size_a = size(A);
>> size_a(1)
ans =
3
>> [n_x,n_y] = size(A);
>> n_x
ans =
3
>> n_y
ans =
2
```

IV b) Generating a vector

IV b) i The colon operator

One of the most powerful operator in `MatLab` is the colon operator ":". It allows in particular to generate lists.

We have seen already that `v(2:5)` gives the elements of `v` between the 2nd and 5th position. What it does is actually ask `MatLab` to send back elements of `v` in position $2, 3, 4, 5$. These positions are *generated* by the command `2:5`.

```
>> 2:5
ans =
2 3 4 5
```

The command `a:b` hence generates a vector, starting in a, each element being incremented by 1, until it would be greater than b.

```
>> 4:8
ans =
4 5 6 7 8
>> 4.5:8
ans =
 4.5  5.5  6.5  7.5
```

In the second example, the last item is 7.5. 8.5 would be larger than 8 and is hence omitted.

It is possible to force the increment. The command `a:da:b` generates a vector, starting in a, each element being incremented by da, until it would be greater than b.

```
>> 0:.1:1
ans =
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.
```

IV b) ii Knowing well the matrix-at-hand

`MatLab` is mainly good at manipulation matrices. It means that knowing the properties a the matrix-at-hand is fundamental.

`MatLab` hence provides many ad hoc functions to know and control the properties of matrices, see Tab. 2.1.

Table 2.1: A few functions that gives infos on a matrix.

name	description	illustration
`length`	largest dimension	`length([[6,9,8];[7,12,-1]])`
`size`	gives the dimensions	`size([[6,9,8];[7,12,-1]])`
`ndims`	gives the number of dimensions	`ndims([[6,9,8];[7,12,-1]])`
`numel`	number of elements	`numel([[6,9,8];[7,12,-1]])`

IV c) Generating a vector

Table 2.2: A few functions that generate vectors.

name	description	illustration
colon operator :	see Sec IV b) i	0:10:.5
`linspace(a,b,n)`	linearly spaced n-dimensional vector between a and b	`linspace(0,1,11)`
`diag(A)`	diagonal of matrix (A) between a and b	`diag([[1,2];[3,4]])`

Table 2.3: A few functions that generate matrices.

name	description	illustration
`zeros(m,n)`	m, n-dimensional matrix filled with 0	`zeros(2,4)`
`ones(m,n)`	m, n-dimensional matrix filled with 1	`ones(2,4)`
`rand(m,n)`	m, n-dimensional matrix filled with random numbers taken between 0 and 1	`rand(2,4)`
`eye(n)`	n-dimensional identify matrix	`eye(10)`
`diag(v)`	matrix filled with 0 with `v` as diagonal	`diag([1,2,3])`

Table 2.4: A few functions that are useful.

name	description	illustration
`size(M)`	number of elements in M	`size([1,2,3])`
`shape(M)`	number of elements in each direction of M	`shape([2,3,5];[1,2,3]])`

IV d) Generating a matrix

V Arithmetic in MatLab

VI What to remember

VI a) Creating a matrix

Follow the procedure:

- start with a bracket [
- write each element of a column, separated with space or commas
- separate rows with a semi-colon
- end with the bracket]

VI b) Manipulating matrices

One can

- add (with "+"), subtract (with "-"), multiply (with "*") matrices if their sizes are compatible
- access to the ith element of matrix `M` using `M(i)`
- access to the (ith,jth) element of matrix `M` using `M(i,j)`

VI c) Key example

Let's summarize in a few examples everything we have seen.

```
>> M = [[1 ,2];[3 ,4]]
M =
 1  2
 3  4
```

Chapter 3

Plotting data

One of the usefulness of MatLab is being able to easily plot, manipulate and modify data.
With only a few commands, publication-ready figures can be produced.

I Creating the first plot

First one needs data.

```
>> x = [1 2 3 4 5 6 7 8 9 10];
>> y = [1 4 9 16 25 36 49 64 81 100];
```

To plot the data, the main command in `MatLab` is `plot`:

```
>> plot(x,y)
```

Definition

PLOT: The `plot` function allows to plot the graph of a function or to plot data.

- `plot(y)` creates a line plot of the data in y.
- `plot(x,y)` creates a line plot of the data in y versus the corresponding values in x.

`plot` plots solid lines. To see the data points, one must specify a marker symbol, for example, `plot(x,y,'o')`.

Help

Go and look for PLOT in the help for details

Pro Tip

When only one data is provided, `MatLab` assumes that the abscisse are the indexes of the elements of the data.
What will be plotted is hence the coordinates $(1, y(1)), (2, y(2), \ldots, (n, y(n))$.
Compare `plot(x,y)` and `plot(y)`!
When trying to understand some results, never hesitate to plot some data `y` by typing `plot(y)` in the command prompt. Quick and dirty, but useful.

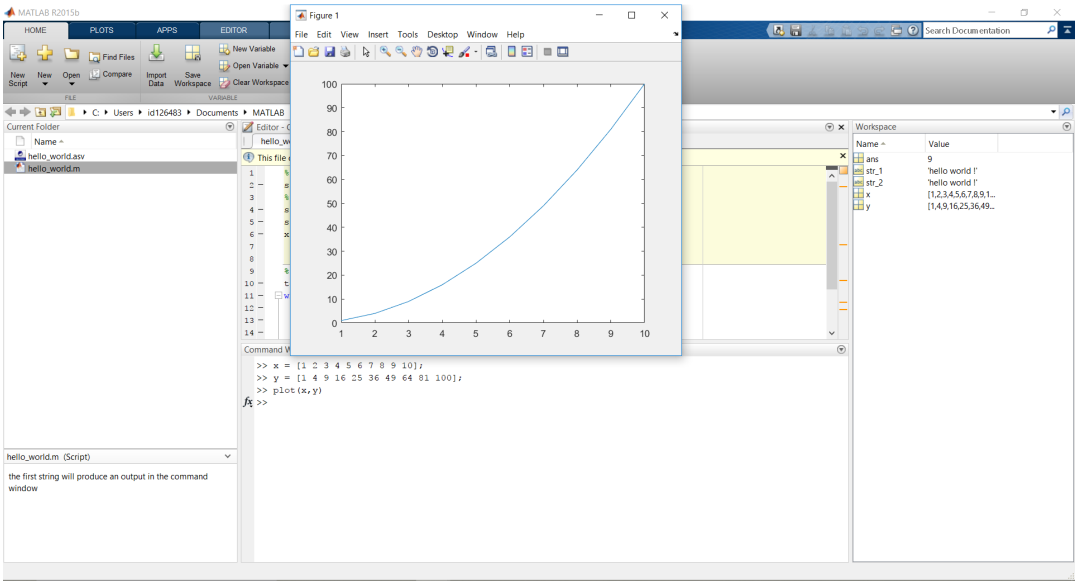


Figure 3.1: Plotting some data.

Pro Tip

The command `figure` can be used for opening a new *empty* figure window. `MatLab` draw any new plot in the last active figure window, erasing the old figure. Creating a new window *before plotting* ensures that the previous plot will not be lost: `figure ; plot(x,y)`

II A beautiful plot

II a) What a figure should be

A good figure has always:

- labels on the x-axis and y-axis
- different colors or patterns for different curves
  If possible, these differences should persist when printing the figure in black and white.

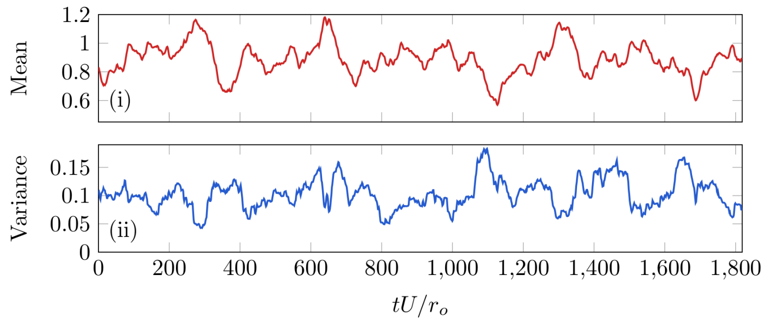
If the figure is in a report :

- instead of a title, the figure can be explained in the text under the figure.
- instead of a legend, the difference between curves can be explained in the text under the figure.

II b) axis

It is *really important* to always have labels for the axis.

```
>> x = [1 2 3 4 5 6 7 8 9 10];
>> y = [1 4 9 16 25 36 49 64 81 100];
>> plot(x,y)
>> xlabel('data along x')
>> ylabel('data along y')
```



(a)

Test	Cell	p-value
PP (F)	1	0.00694
PP (t)	1	0.00622
PP (F)	2	0.00252
PP (t)	2	0.00256
PP (F)	3	0.00386
PP (t)	3	0.00366

(b)

FIG. 11. (a) Windowed (i) mean and (ii) variance computations for $\vec{r}$, with window length $\Delta t_{\text{win.}} = 1000U/r_o$. (b) The results of a Phillips–Perron (PP), or augmented Dickey–Fuller, test for the cell data indicated based upon F and t metrics (see Elder and Kennedy [25]).

Figure 3.2: Illustration from a (soon) published paper. Details are provided in the caption, so titles/legends are not necessary. Labels, on the contrary, are always necessary.

`xlabel` and `ylabel` are the `MatLab` functions that handle the labels.

The argument for these functions are between quotes ””. A label is some text: it means that `MatLab` will need a string.

```
>> x = 0:pi/10:4.*pi;
>> y = cos(x);
>> plot(x,y)
>> str_xlabel = 'x';
>> xlabel(str_xlabel)
>> str_ylabel = 'cos(x)';
>> ylabel(str_ylabel)
```

II b) i Colours and line styles

The default plot color in `MatLab` is blue, and the line default style is plain. To change the style, an extra argument has to be sent to `MatLab`.

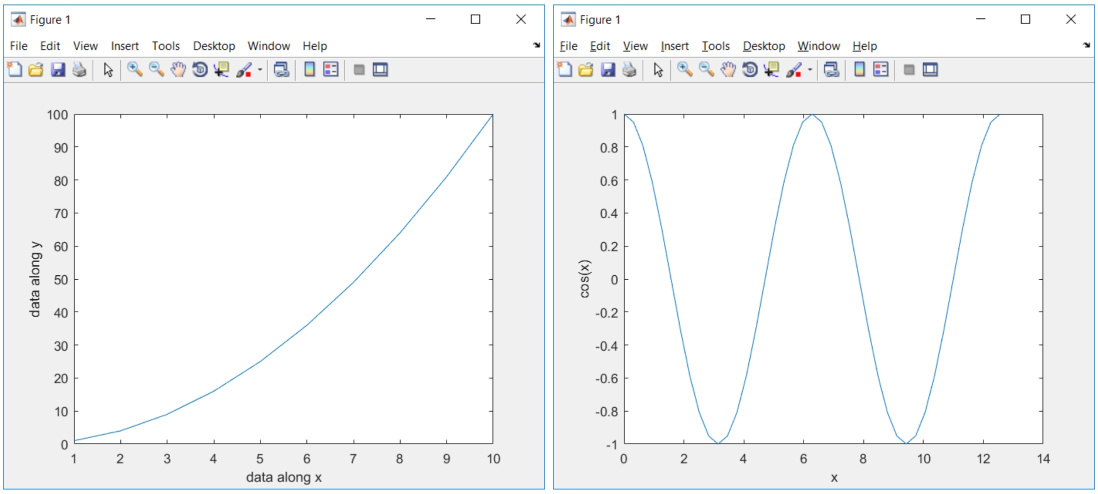
For instance,

- `plot(x,y,'r')` will change the color of the plot to red.
- `plot(x,y,'--')` will change the line style of the plot to dashed.

Note that the argument is between quotes. Arguments are generally passed as a string

Table 3.1: A few colours in `MatLab`.

colours	code	illustration
black	k	`plot(x,y,'k')`
green	g	`plot(x,y,'g')`
red	r	`plot(x,y,'r')`
blue	b	`plot(x,y,'b')`
yellow	y	`plot(x,y,'y')`



(a) (b)

Figure 3.3: Note the labels on these two plots. a): plot of the function $f(x) = x^2$ between 1 and 10. b): plot of cos between 0 and 4π.

Table 3.2: A few line styles in `MatLab`.

style	code	illustration
solid	-	`plot(x,y,'-')`
dashed	–	`plot(x,y,'--')`
dotted	:	`plot(x,y,':')`
dash-dot	-.	`plot(x,y,'-.')`

Table 3.3: A few marker styles in `MatLab`.

marker	code	illustration
plus	+	`plot(x,y,'+')`
cross	x	`plot(x,y,'x')`
circle	o	`plot(x,y,'o')`
triangle up	ˆ	`plot(x,y,'^')`
triangle down	v	`plot(x,y,'v')`
triangle left	<	`plot(x,y,'<')`
triangle right	>	`plot(x,y,'>')`

Pro Tip

A good way of not messing with the style is to follow this order:

1. open the quotes: `'`
2. add the color, e.g. k: `'k`
3. add the style of the line, e.g. `--`: `'k--`
4. add the style of the markers, e.g. x: `'k--x`
5. end the quotes: `'k--x'`

One will then plot with the options:
`plot(x,y,'k--x')`

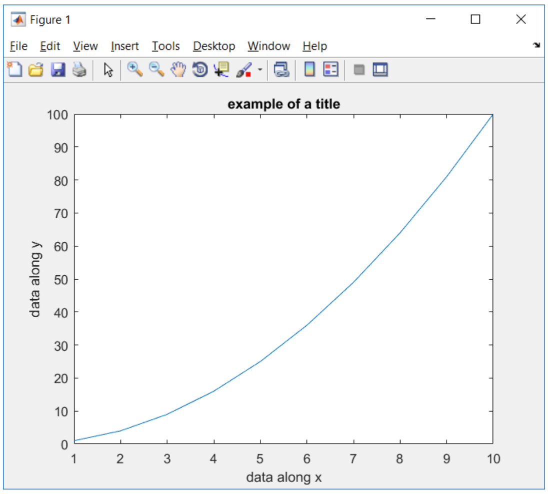


Figure 3.4: Note the title.

II c) Title

Adding the title is simple. In the same spirit of the labels, just pass the title to the function `title`.

```
>> x = [1 2 3 4 5 6 7 8 9 10];
>> y = [1 4 9 16 25 36 49 64 81 100];
>> plot(x,y)
>> xlabel('data along x')
>> ylabel('data along y')
>> title ('example of a title')
```

III Edit a plot in the window

A way to edit the plot properties is to click on the arrow - Edit Plot - on the plot window.

Then, double clicking on the curve will open an extra window where one can play with the properties.

In particular there is two interesting properties:

1. `Line`, where the style of line can be edited (plain, dashed, etc.), the width of the line, and its color

2. `Marker`, where the type of markers can be edited (indication where each data is plotted, for instance with circles), the size of the markers, the color of the inside of the marker as well as the color of their edges

IV Multiple plots

IV a) Multiple plot in one figure

IV a) i In one call

One call to the function `plot` is enough to plot multiple graphs on the same figure. For that, couple of arguments need to be provided to the `plot` function.

```
>> x = 0:0.1:4.*pi
>> y_1 = cos(x); y_2 = cos(x+0.8); y_3 = cos(x+1.6);
```

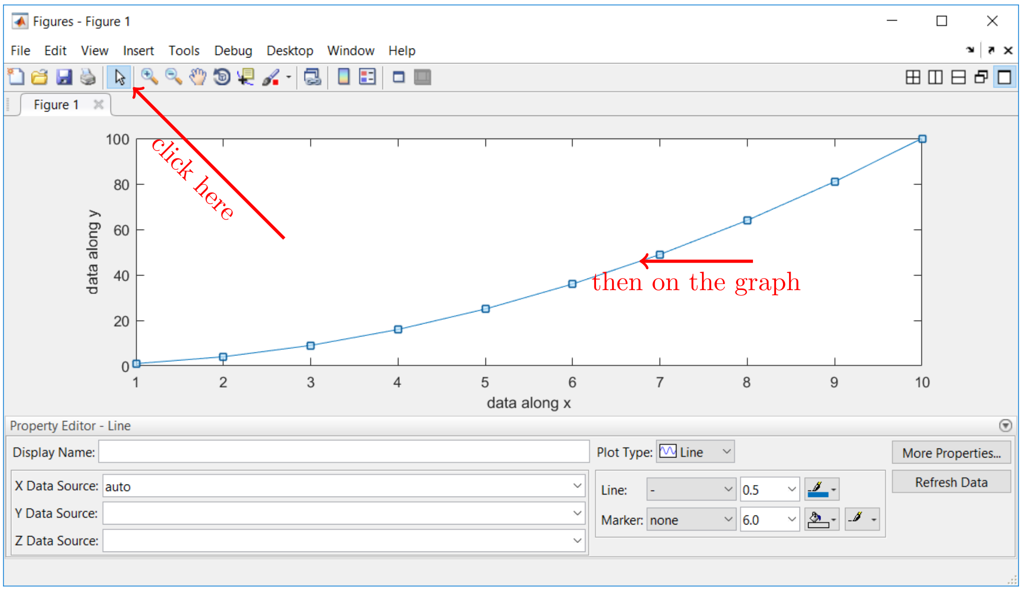


Figure 3.5: Editing a plot from the window is relatively simple, though fastidious. First click on the "arrow" to start the edit mode. Then, clicking on one of the plot objects, for instance, the curve, will allow to start editing it. Note the `Line` and `Marker` properties.

```
>> plot(x,y_1,x,y_2,x,y_3)
>> xlabel('x')
>> ylabel('f(x)')
>> title ('several cos')
```

To add some control on the plot, the style can be provided just after the data:

```
>> plot(x,y_1,'-',x,y_2,':',x,y_3,'--')
>> legend('cos(x)','cos(x+0.8)','cos(x+1.6)')
```

IV a) ii In multiple call

When calling `plot`, the active figure gets erased before `MatLab` draw the new plot.

But calling `hold on` (and later on `hold off`) `MatLab` knows that it should not erase the figure.

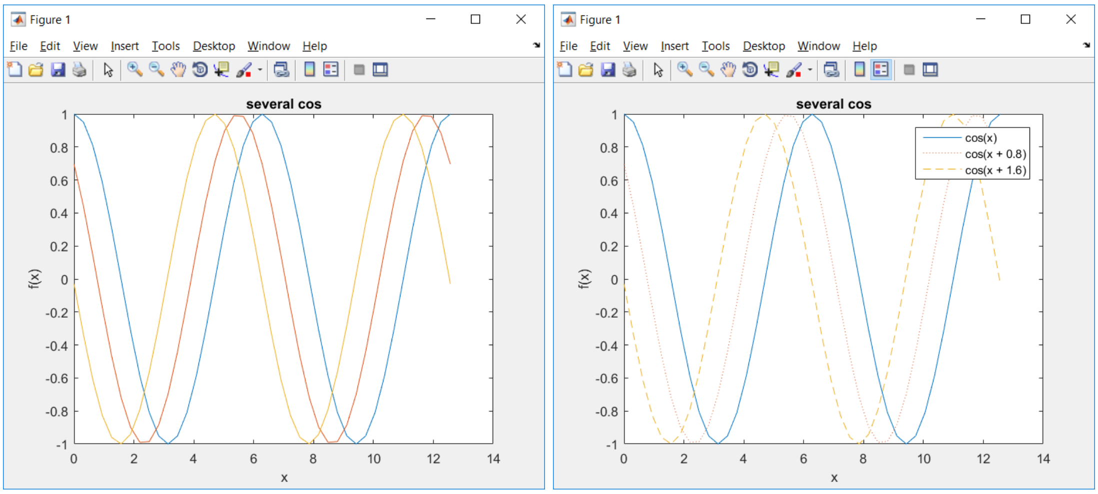
```
>> x = 0:0.1:2.*pi
>> y_1 = cos(x); y_2 = cos(x+0.2); y_3 = cos(x+0.4);
>> plot(x,y_1)
>> hold on
>> plot(x,y_2)
>> plot(x,y_3)
>> hold off
```

`hold off` tell `MatLab` that the figure is no longer protected.

IV b) figure and multiple plots

[[TODO ** image plot **]]

[[TODO this section]]



(a) (b)

Figure 3.6: Several graph on the same figure. a): no style is provided. b): "stylish" curves.

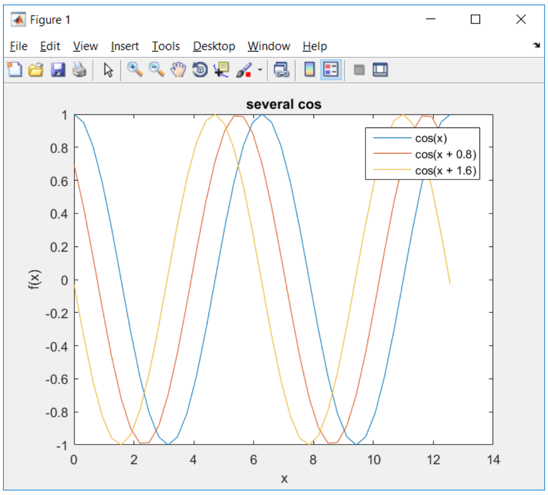


Figure 3.7: Same figure as before, with legends.

IV c) legends

Adding the legend is simple. In the same spirit of the title, just pass the legends, one after the other, to the function `legend`:

```
>> plot(x,y_1,'-',x,y_2,':',x,y_3,'--')
>> legend('cos(x)','cos(x+0.2)','cos(x+0.4)')
```

Chapter 4

Programming

I Script

Scripts are the simplest kind of program file because they have no input or output arguments.

They are useful for automating series of `MatLab` commands, such as computations that have to be performed repeatedly from the command line.

When execution of the script completes, the variables remain in the `MatLab` workspace.

Definition

SCRIPT: A script is basically a text file, with the ".m" extension, for instance, `my_script.m`.
This file contains series of instructions that `MatLab` can and/or will execute.
A script does not take inputs (they hence have to be provided). It does not provide outputs, though the variables will be stored in the workspace.

[[TODO ** image tikz illustration of a script? **]]

I a) Create a script

One can create a new script in the many ways.

I a) i Creating a script with the prompt

The `edit` command allows either to create a script, or to edit an existing script. To create a blank script, named "untitled.m", type:

```
>> edit
```

`MatLab` will then create and open the script. The script is created in the current folder that `MatLab` is using. To change the folder, the "current directory" windows on the left should be used.

To create a script named "my_scipt.m", type:

```
>> edit my_script
```

Note that specifying the extension is unecessary. Also, when the script "my_script.m" exists already, typing `edit my_script` will open the preexisting script.

I a) ii Through `MatLab` interface

Another way is through the IDE. Click on the "EDITOR" tab, then on "+ New", and select script. One can then save it with the desired name.

I a) iii Create a text file

Just create a text file (file with the ".txt" extension) in windows, and change the extension to ".m".

Et voil.

I b) Save a script

Saving a script and running the code can be done using either of these methods:

- Typing the script name on the command line and pressing Enter.
- Clicking the Run button on the Editor tab.
- Clicking on the save icon.

I c) Comments

On of the most important thing is to always try write as much comments as possible in a code.

Definition

COMMENT: A comment is a part of the file that will *not* be executed by `MatLab`. It allows to give explanations on the script. For later reading, or if someone else is reading the code, it will give invaluable information on what the code is actually doing. Any line starting with the percentage symbol "%" will be ignored. Any part between "%{" and "%}" will be ignored as well.

Pro Tip

A good piece of code needs:

- well-named variables
- indentation
- comments

I d) execute the script

There is several way for running a script, from the editor, from the prompt and thanks to convenient shortcuts.

I d) i From the editor

I d) i 1 Run button The easiest is to be in the editor and to click on the triangle button ">".

[[TODO ** image run button **]]

It will run the whole script (and will save it).

I d) i 2 Shortcut There is two extremely handy shortcuts, from the editor:

1. `F5` will run the whole script
2. `ctrl+enter` will run the active cell

The last one is really convenient as well. The active cell is highlighted in light yellow.

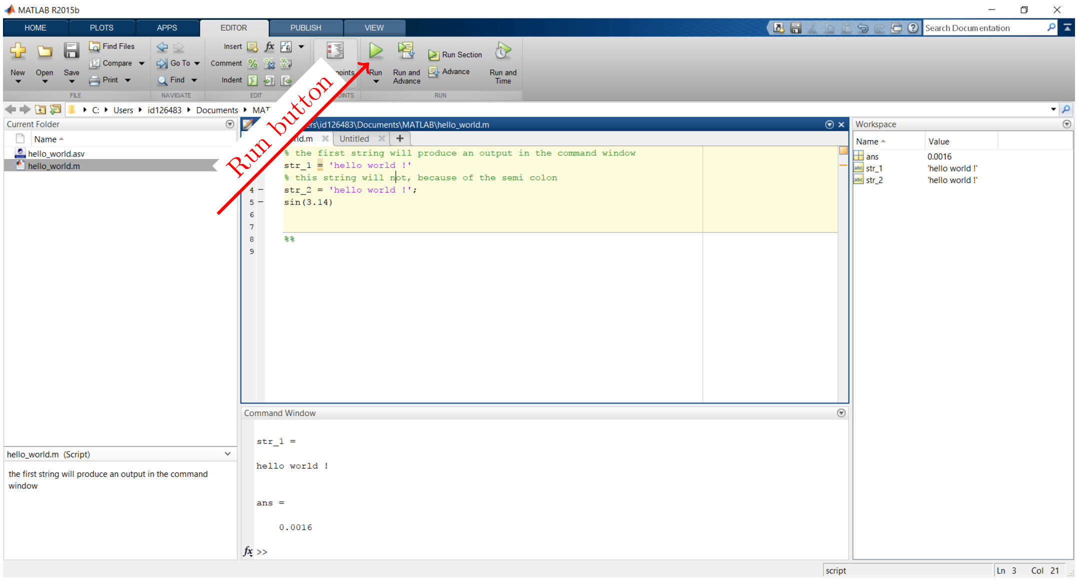


Figure 4.1: the run button in the editor.

I d) ii From the prompt

Another easy way is to type the name of the script:

```
>> my_script
```

Alternatively, the function `run` can be used:

```
>> run my_script
```

I e) reading the script

`MatLab` has a set of colors for helping the reader to understand the code:

- green: comments
- blue: functions (to be verified)
- yellow background: section with focus

II Loops

Loop control statements will repeatedly execute a block of code. It is really convenient, as sometimes, one want to do *almost* the same thing many times. For instance, evaluating a series, such as the Fibonacci sequence, one needs to evaluate repeatedly the same expression $un = u_{n-1} + u_{n-2}$.

There are two types of loops:

- `for` statements loop a specific number of times, and keep track of each iteration with an incrementing index variable.
- `while` statements loop as long as a condition remains true.

Definition

LOOP: A loop will repeat instructions a certain number of times.
It is really useful when something can be done almost exactly similarly from on time to the other. For instance, if one want to compute the distance that Julie (see Sec. IV) can drive if the tank is 20 liters, 25 liters ? or maybe if it is 55 liters ?
One can loop on the variable `tank_capacity` and not have to rewrite the code several times or to execute it several times.

[[TODO ** image tikz illustration of a loop **]]

II a) Loops based on For

`for` loops are based on a finite number of iterations.

Executing this script:

```
for i=1:5,
  i
end
```

will lead in print i at every step of the loop. `i` will hence be 1 then 2 etc. up to 5.

Definition

FOR: A `for` statement will execute a block of code repeatedly.
What is needed for making a `for` loop:

- a list `my_array`
- a variable `var_i`
- some code `do stuff`

The `for` loop is written as:

1. `for var_i=my_array,`
2. `do stuff`
3. `end`

First, `var_i` will, one after the other, be assigned to every value that exists in `my_array`. If `my_array=[1,2,3]`, then `var_i` will be 1, then 2, then 3.

Once `var_i` has been assignated to a new value, `MatLab` will execute the code `do stuff`. If `do stuff` is i^2, then `MatLab` will print 1, then 4, and finally 9.

`end` says to `MatLab` that the part `do stuff` is finished and that he can loop.

`do stuff` can be very complex instructions, and `my_array` can be a very long list, for instance `1:10000`.

A useful example is construction of vectors. Let's for instance construct the Fibonacci sequence up to 100:

```
fibo = ones(100);
for i=3:100,
  fibo(i) = fibo(i-2) + fibo(i-1);
end
```

It is possible to nest loops, i.e., to have a loop in a loop. Let's construct a Vandermonde matrix.

```
v = 1:4;
M = ones(4,4);
for i=1:4,
   for j=1:4,
     M(i,j) = v(i)^j;
   end
end
```

Pro Tip

Indent the loops for readability!
For each loops, indent by using tab or a few spaces.
Without indentation, it is really hard to see when loops start and end.

II b) Loop based on while

II c) Controls on loops

Exiting a loop can be done programmatically by using a break statement, or skip to the next iteration of a loop using a `continue` statement. Details will be given in Sec. III

III Conditions

III a) Is $x > 0.5$?

Pro Tip

Checking is a quantity is equal to an other is *very* important.
For instance, it is pivotal in `if/else` statements. The condition has to be checked. Is it true or false ?
But `MatLab` can not understand *condition* = true. That would assign true to the condition, which is *very* different from checking if the condition is true.
Consequently, `MatLab` uses a special operator for checking if two quantities are equal: the "==" operator.

[[TODO work here]]

III b) `if/else` statement

More than often, computations depend on the situation. Think of a piece-wise function, define on $[0, 1]$:

$$f(x) = \begin{cases} -1\text{if } x < 0.5 \\ 1\text{if } x > 0.5 \end{cases}$$

Evaluating f hence depends if the argument x is larger or lesser than 0.5. The keyword is `if`.

Definition

IF/ELSE: The `if/else` statement executes a block of code if a specified condition is true. If the condition is false, another block of code can be executed.
What is needed for making a `for` loop:

- specified *condition* to be verified
- some code `do stuff` if the *condition* is true
- some other code `do other stuff` if the *condition* is false

The `if/else` statement is written as:

1. `if` *condition* `== True,`
2. `do stuff`
3. `else,`
4. `do other stuff`
5. `end`

If there is no block `do other stuff` then the statement is slightly simpler:

1. `if` *condition* `== True,`
2. `do stuff`
3. `end`

First, `MatLab` will check if the condition is true. For instance, the condition can be x larger than 0.5. The condition then writes `x>0.5`. The first part, corresponding to the evaluation, of the `if/else` statement is then: `if (x>0.5) == true,`. Then, `do stuff` is executed if the *condition* is true. For instance, the variable `fx` can be assigned to 1, with the piece of code `fx = 1;`. Then, if specified and in the eventuality than the *condition* is false, the alternative code can be executed, say that `do other stuff` is to assign -1 to the variable fx: `fx = -1`. Finally, the *end* statement will tell `MatLab` that the `if/else` block is finished.

Understanding if requires to understand

```
x = 0.6
if (x>0.5) == true,
        fx = 1;
else,
        fx = -1;
end
```

Bibliography