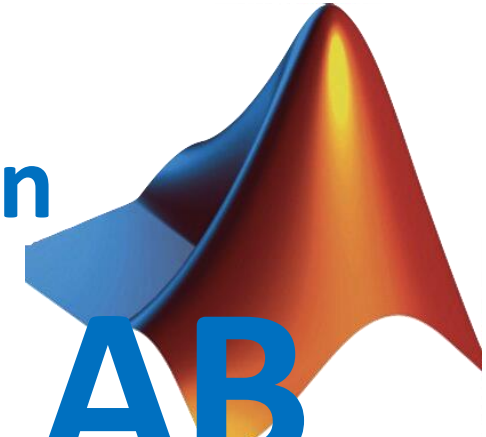
UNIVERSITÀ DI PAVIA

EURELAX
COST ACTION CA15209

Brush up on MATLAB



Giacomo Galuppini

Identification and Control of Dynamic Systems Lab, **University of Pavia**, Italy.

Stelar s.r.l. , Mede, Italy.

STELAR

IDentIFY

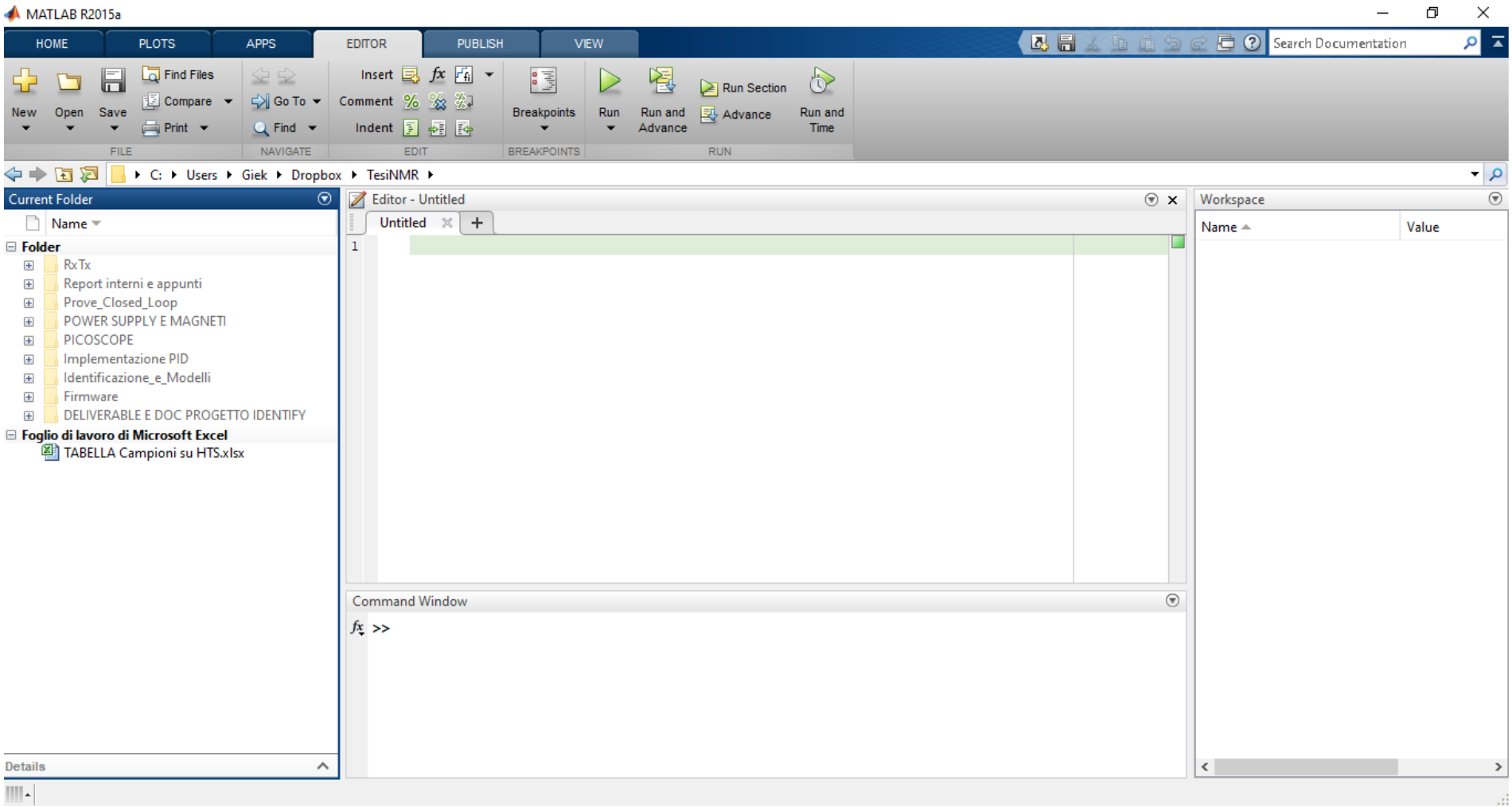
MATLAB: What is it?

MATLAB (from **MAT**rix **LAB**oratory) is a high level numerical computing environment and programming language.

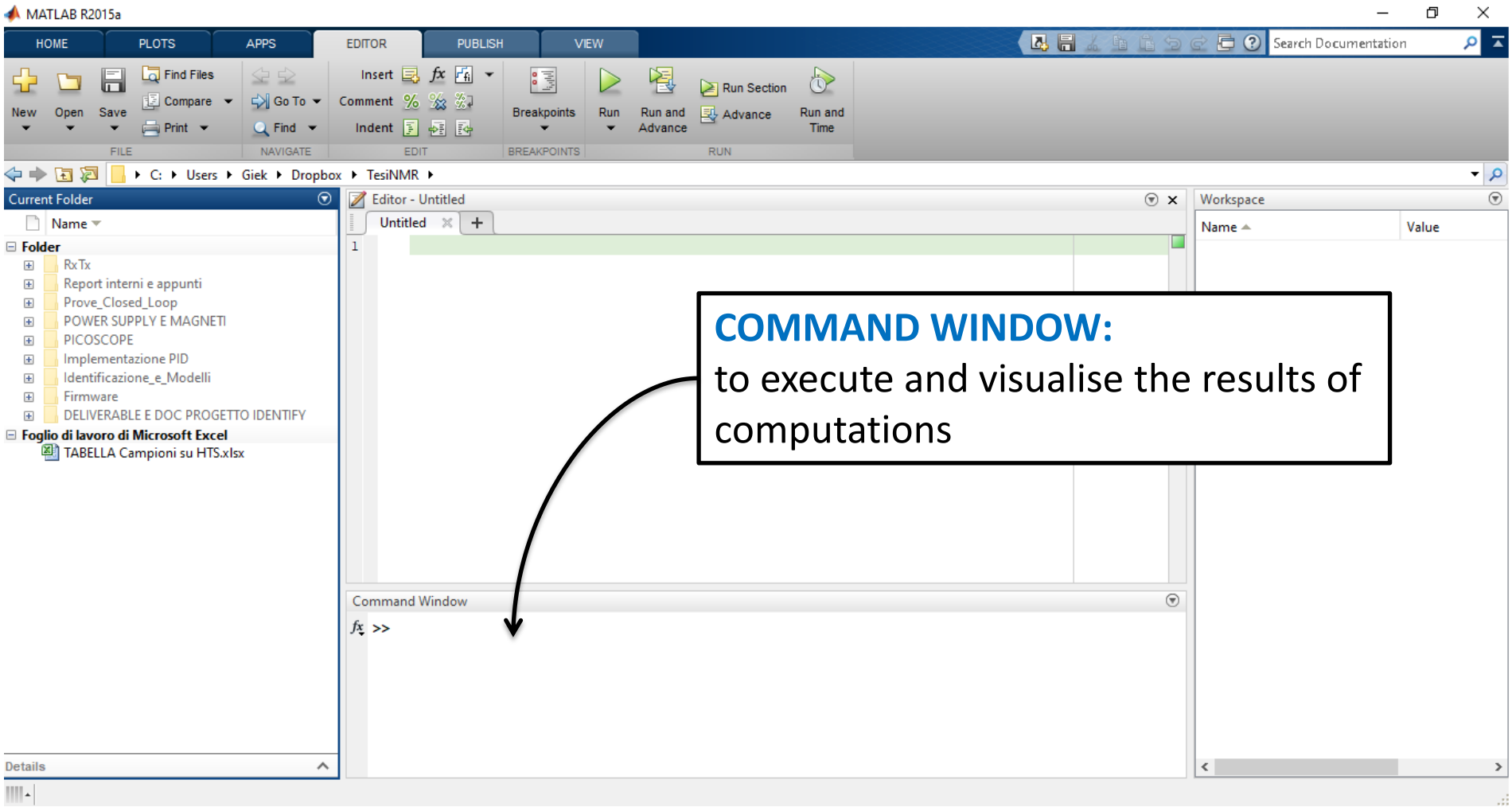
MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages.

MATLAB is intended primarily for numerical computing and includes several **Toolboxes**, that is additional functions to solve specific classes of problems.

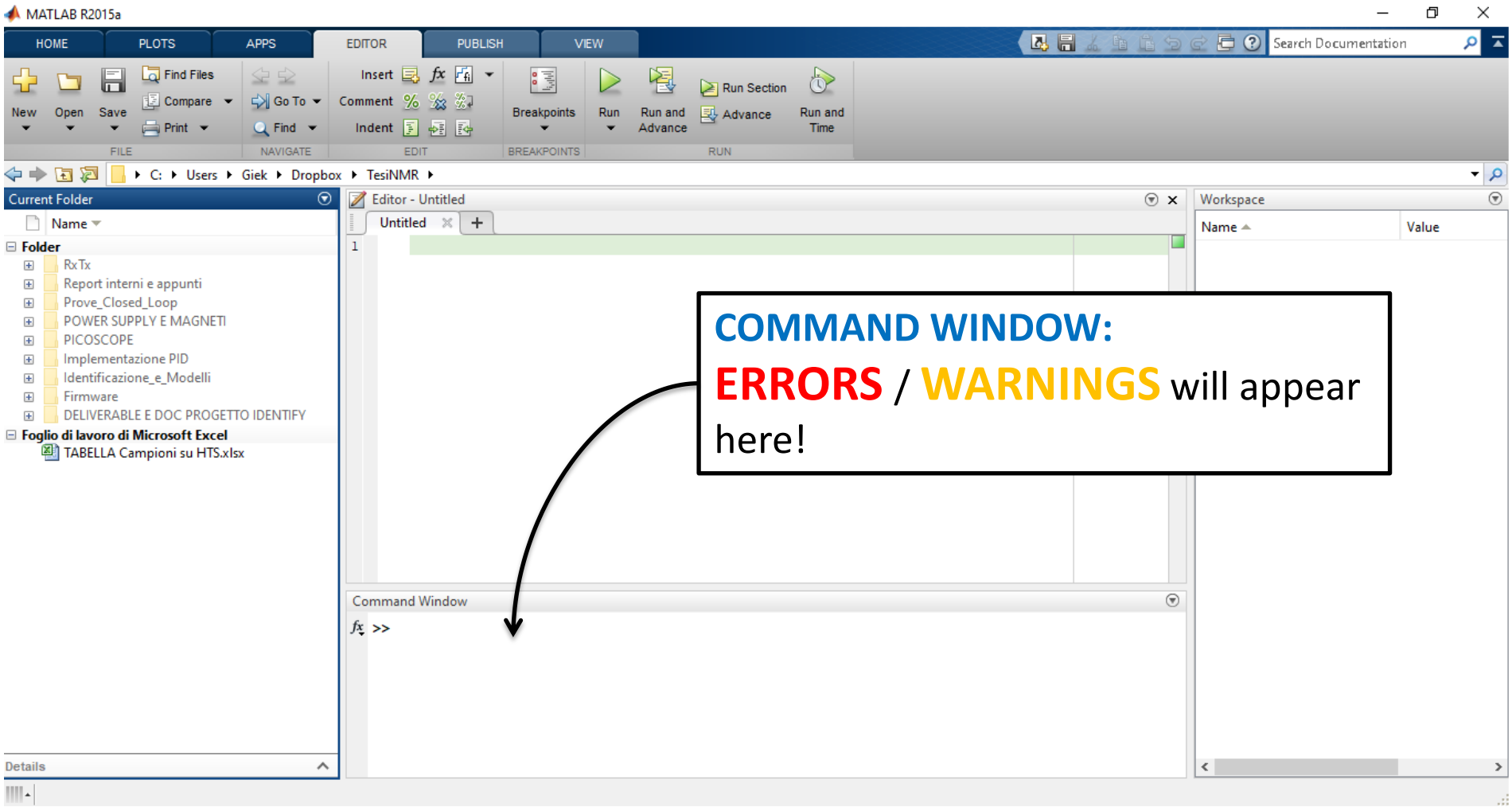
Work environment



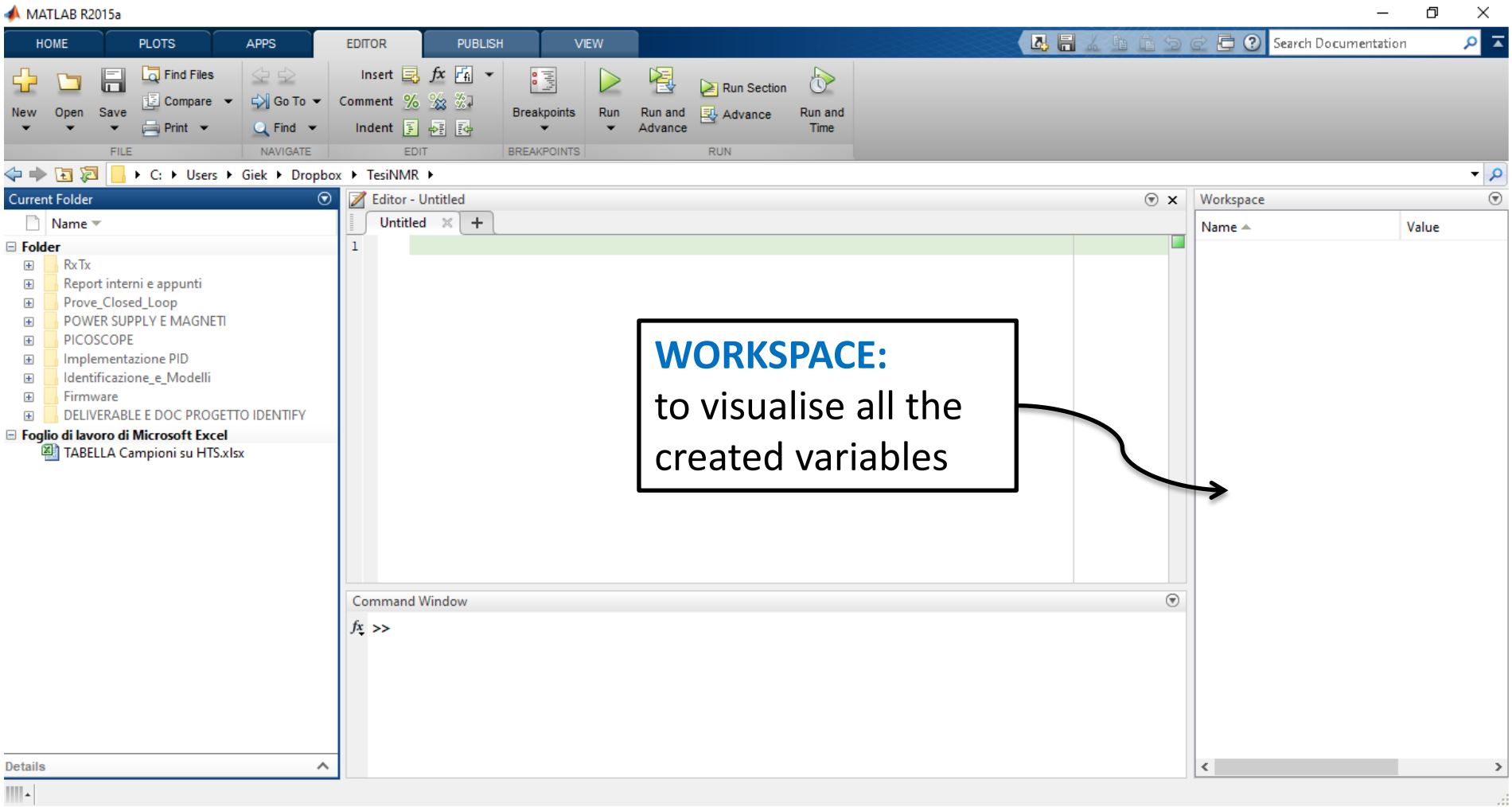
Work environment



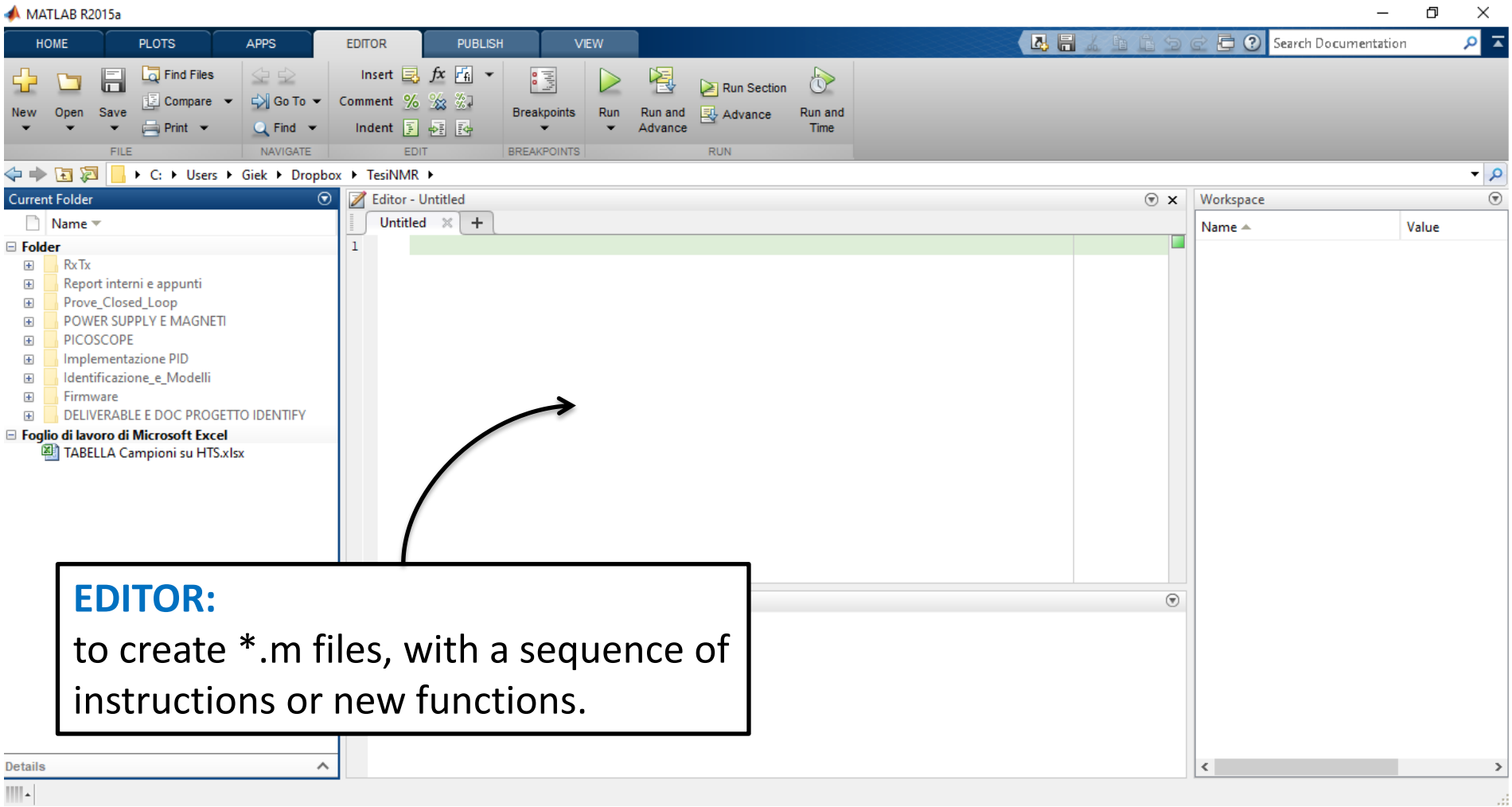
Work environment



Work environment

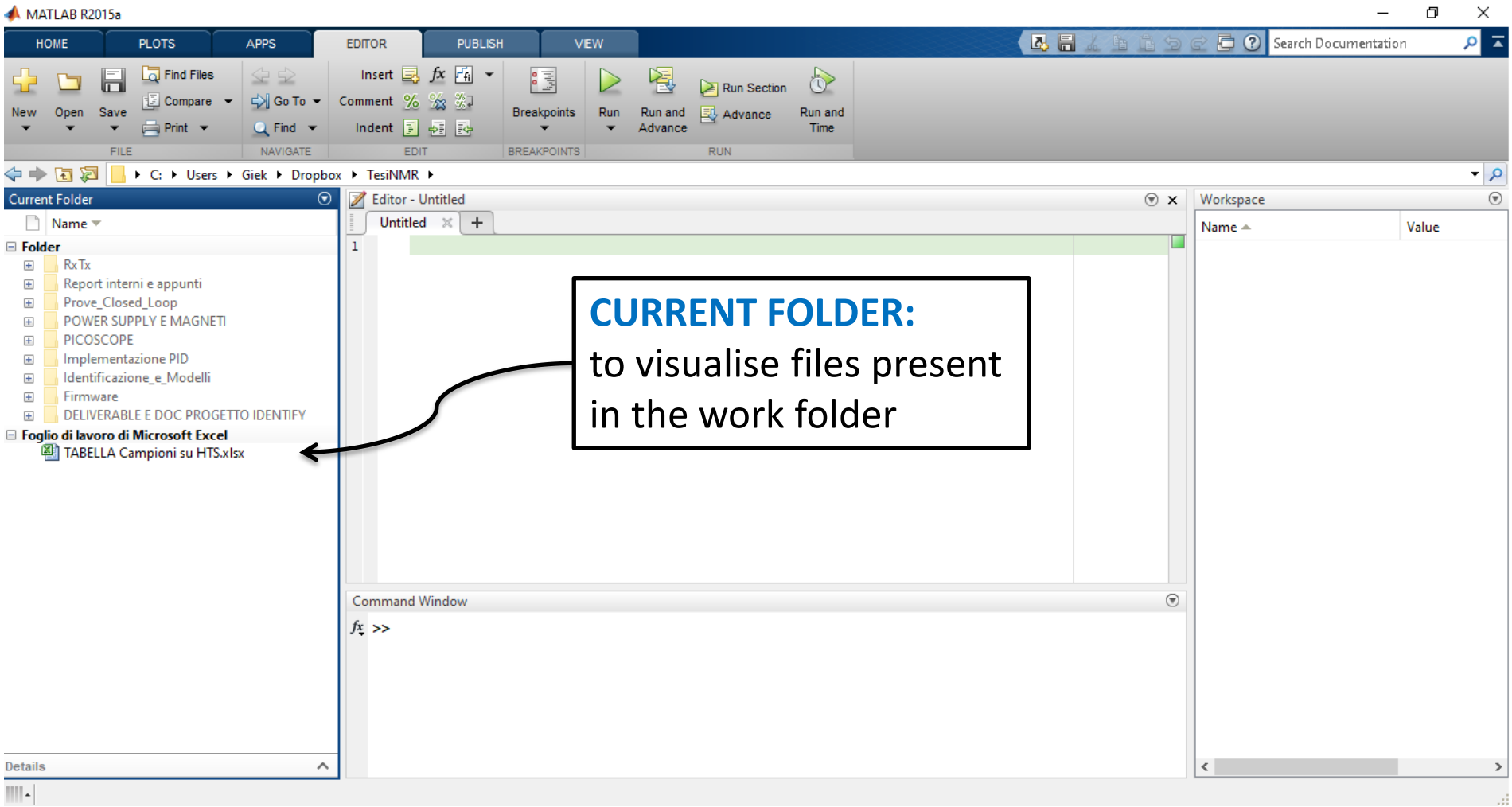


Work environment

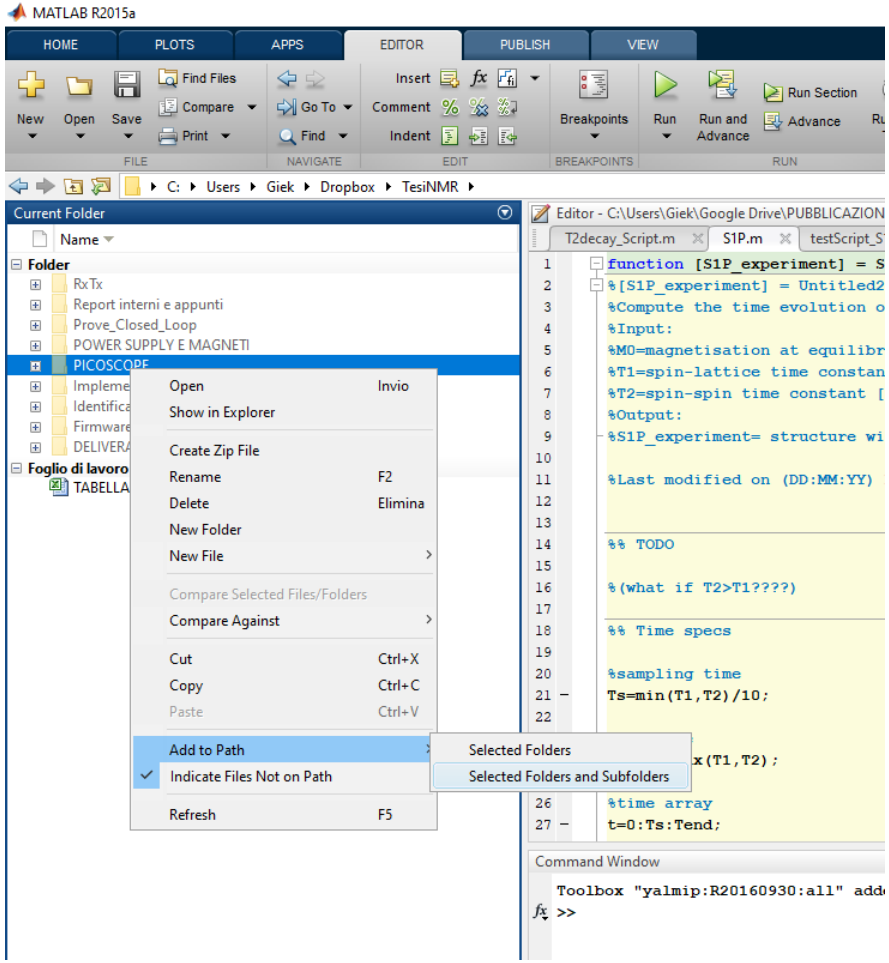


EDITOR:
to create *.m files, with a sequence of instructions or new functions.

Work environment



Matlab path



The **MATLAB path** is the **set of folders MATLAB has axcess to**.

From the «Current Folder» window: Right click on your folder -> Add to Path -> Selected Folder [and Subfolders]

Or use commands:

```
addpath('path')
```

or

```
addpath(genpath('path'))
```

(MOST) Useful commands

`help command_name`

to see the details of the considered command

`doc command_name`

to see the documentation of the command

Why not…? Google



Programming with Matlab

MATLAB makes use of an **interpreted programming language**:

No need to compile the whole script!

Instructions can be executed one by one…

- From the **command window**
- Selecting code lines **from the editor**

…or executing everything is in the editor.

This makes **Programming & Testing a lot easier…**

…but execution may be **slower.**



Programming from the command window

The simplest way to use MATLAB is doing simple computations such as (+, -, *, /, ^)

```
>> result=(18*2+19)/88
```

The general syntax:

```
>> variable=expression
```

All the variables are stored in the **workspace**

If you use **; at the end of the instruction**, the result of the computation is not displayed.

Matrices and vectors

Syntax for arrays:

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

M =

     1     2

     3     4

>> Mt=M'

Mt =

     1     3

     2     4
```

; works as **row delimiter**.

' is the **trasnspose operator.**

Matrices and vectors

Alternative syntax for arrays :

```
>> x=1:5

x =

1  2  3  4  5
```

: for **equally spaced arrays**

```
>> x=1:0.5:2

x =

1.0000  1.5000  2.0000
```

Array indexing

`A(i,j)`	select the entries of i-th row and j-th column
`A(:,j)`	select the j-th column of A
`A(i,:)`	select the i-th row of A
`A(:,end)`	select the last column of A
`A(end,:)`	select the last row of A
`A=[]`	create an empty vector or matrix
`v(i)`	select the i-th element of vector v
`[m,n]=size(A)`	determine the size of A (m rows, n columns)
`m=length(v)`	determine the number of elements of v

Array indexing

`A(i,j)`	select the entries of i-th row and j-th column
`A(:,j)`	select the j-th column of A
`A(i,:)`	select the i-th row of A
	th element of vector v
`[m,n]=size(A)`	determine the size of A (m rows, n columns)
`m=length(v)`	determine the number of elements of v

INDEXING STARTS FROM 1

i,j>0



Array indexing

`A(i,j)` ← `i` and `j` can be **variables!**

```
>> x=10:0.5:12
x =
   10.0000   10.5000   11.0000   11.5000   12.0000

>> idx=[3 5]
idx =
     3     5

>> x(idx)
ans =
    11    12
```

Array indexing

```
A(find(condition))
```

logical indexing

```
>> x=10:0.5:12
x =
   10.0000   10.5000   11.0000   11.5000   12.0000

>> idx=find(x>11)
idx =
     4     5

>> x(idx)
ans =
   11.5000   12.0000
```

Math functions

sin	`sin(z)`	`sind(z)`
cosin	`cos(z)`	`cosd(z)`
tangent	`tan(z)`	`tand(z)`
arctan	`atan(y)`	`atand(y)`
exponent	`exp(x)`	
natural logarithm	`log(x)`	
10 logarithm	`log10(x)`	
root square value	`sqrt(x)`	

The element-wise computations are executed as `.+` `.*` `.^` `./`

Polynomial functions

In order to represent a polynomial function, it is necessary to write a vector containing the **coefficients in a decreasing order**.

For instance $p=t^3 -6t+3$ is

```
p=[1 0 -6 3]
```

while

```
r=roots(p)
```

allows one to find the root values of p

Cycles and logic conditions

It is possible to write typical coding structures (for, while, if).
Note that it is necessary to close them by using the instruction **end**.

for condition …instructions … **end**

while condition …instructions … **end**

if condition …instructions … **else** … instructions **end**

```
s=0;
for i=1:9
   s=s+i^2/(i+1);
end
>> s =
37.9290
```

.m-files - Scripts

All the **files containing instructions** which can be executed by MATLAB are called **m-files**. (e.g. namefile.m).



Run (F5) to execute the script!

```
% is used for comments
%% is used to create sections in the .m file
```

```
%% time specs

%time array
t=0:0.01:1;
```

Plots

plot is the function used to draw graphics

```
>> figure(1)
plot(t,q,'b','linewidth',1.1);
title('M');
xlabel('time (s)');
ylabel('M (V)');
grid on
xlim([-5,20])
ylim([0,0.6])
```

To open a new figure

Plot figure with line width 1.1

Define labels for the axes

Draw the grid

Define the limit for the x y axes

Useful commands

`clear all` or `clear variable_name`

Clears all the variables in the workspace or clears only variable_name

`close all` or `close figure i`

Closes all figures or closes only figure i

`clc` clears the Command Window

`save file_name`

Save the content of the workspace (*.mat)

`load file_name`

Loads all the variables contained in file_name (*.mat)



.m-files - Scripts

EXERCISE 1: create a script to plot a simple S1P - T2 decay in the rotating frame.

```
%Mxy=M0*exp(-t/T2)
```

The sampling time must be properly chosen to avoid signal distortion.

The end time must be chosen to show the entire decay.

M0,T2, sampling time (Ts) and end time (Tend) must be variables.



Structures

A **structure** is a data type that groups related data using data containers called fields. Each field can contain any type of data.

```
>>Experiment=struct('time',[0:10],'signal',[10:20])
```

Access data in a structure using dot notation of the form

```
structName.fieldName
```

```
>> Experiment.time

ans =

     0   1   2   3   4   5   6   7   8   9   10
```

m-files - Functions

A MATLAB m-file containing input and output arguments is a **function file.**

```
function y=my_linear (alpha,beta,x)
y= alpha + beta *x;
```

No «return» statement needed

Outputs, Inputs.
Formal parameters , i.e. its name identifies the variable only inside the function.

```
>> vett=[1 4 5 9];
>> my_linear(1,4,vett)
ans =

5    17    21    37
```

m-files - Functions

A MATLAB m-file containing input and output arguments is a **function file.**

```
function y=my_linear (alpha,beta,x)
y= alpha + beta *x
```

THE .M FILE MUST BE SAVED WITH THE SAME NAME OF THE FUNCTION IT CONTAINS!

```
3     17     21     37
```



.m-files - Functions

EXERCISE 2: create a script to plot a simple S1P - experiment in the rotating frame.

Create a Matlab function with the following declaration:

```
[S1P_experiment] = S1P(M0,T1,T2)
```

S1P_experiment= structure with fields time and Signals (Mxy,Mz)

Inf and NaN

```
>>Inf
```

Representation for **positive infinity**.

```
>>NaN
```

Not a Number : these values result from operations which have **undefined numerical results**.

Useful Matlab functions:

```
Bool= isinf(A)
Bool= isnan(A)
```

Many more `is*` functions!



Files

Matlab offers a number of built-in function to **read** and **write data** from/to **files**.

Some examples:

Text files: `textscan()`, `dlmread()`, `csvread()`, `fileread()`,… **and write.**

Spreadsheet files: `xlsread()`,… **and write.**

Image files: `imread()`,… **and write.**



Files and NaNs

EXERCISE 3: Use the S1P function to generate data. Randomly introduce NaN values in the T2 decay values and save (time and signal) on an Excel file.

EXERCISE 4: Read the Excel file to recover your data. Plot the data. Remove the NaN samples and the corresponding timestamps and plot again.

