

Linear Systems

SECOND YEAR GSC/GSA/GTT/GT

LAB SESSION 1– ACADEMIC YEAR 2017-2018

1. Introduction

1.1. Basic Matlab features

MATLAB (**MAT**rix **LAB**oratory) is a high-performance language for technical computing. It integrates *computation*, *visualization*, and a *programming environment*. Furthermore, MATLAB is a modern programming language environment: it has sophisticated *data structures*, contains built-in editing and *debugging tools*, and supports *object-oriented programming*. These factors make MATLAB an excellent tool for teaching and research.



MATLAB has many advantages compared to conventional computer languages (e.g., C, FORTRAN) for solving technical problems. MATLAB has powerful built-in routines that enable a very wide variety of computations. It also has easy to use graphics commands that make the visualization of results immediately available.

Specific applications are collected in packages referred to as *toolbox*. There are toolboxes for signal processing, symbolic computation, control theory, simulation, optimization, and several other fields of applied science and engineering.

1.2. Working environment

When you start MATLAB, a special window called the *MATLAB desktop* appears. The desktop is a window that contains *other* windows. The major tools within or accessible from the desktop are: the command window (used to execute commands and display errors), the command history (used to check already executed commands), the workspace (used to check stored variables), the current directory (used to check the files stored in the current working directory) and the help browser (used to find information regarding the usage and meaning of the Matlab commands). In the following page you can see a figure with the MATLAB desktop.

Some of the most important features of the Command Window are the following:

- It distinguishes uppercase letters and lowercase letters, i.e. it is not the same to refer to the variable `Var` or to `var`. In other words, MATLAB is *case sensitive*.
- The up and down arrows ( and ) allow you to repeat previously written commands.
- Highlighting a the name of a function and pressing **F1** opens a small help window, which is useful as it usually provides some plots and examples about it. The **help** command can also be used to obtain information about it, but this latter is simply displayed on the *command window* and does not contain any plot. The **doc** command

opens a new Matlab window where you can find the information regarding all the Matlab commands.

- If you don't store the information of the last mathematical operation in any other variable, Matlab will automatically assign the value to a variable named **ans**.

- If you finish the instruction line with a semicolon ";", the line will be executed but the result will not be displayed in the command window.

- You can write more than one instruction in a single line if you separate them with a comma "," or a semicolon ";".

- The command **save** allows you to save the workspace (including all the variables and their corresponding values) in a file, with a **.mat** extension, so you can retrieve the workspace anytime with the **load** command.

The Command Window also allows you to control the current working directory with the commands: **cd**, **dir**, **mkdir**, **delete**, etc.

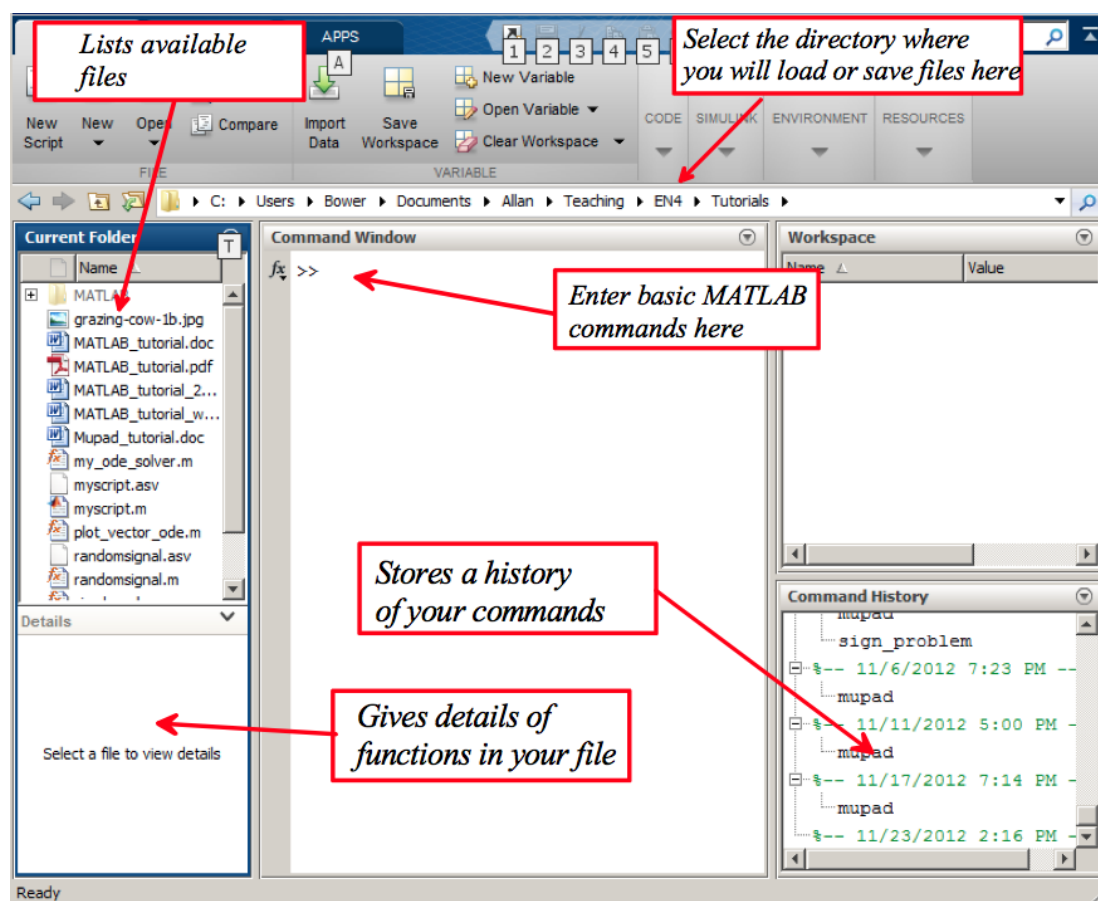


Figure: The graphical interface to the MATLAB workspace.

Variables, vectors and matrices

MATLAB works indistinctly with matrices and scalars, since a scalar is considered as a matrix of size 1x1.

1.3. Matrix generation

The definition of a variable is straight forward, usually the name of the variable is specified with an alphabetical character (either a letter or a word) and the value of the variable is assigned with a number (or another alphabetical variable with and assigned number, e.g., **pi**). A variable can be also set with a mathematical operation:

```
>> A=2                                >> B=pi/4
A =                                    B =
2                                     0.7854
```

When a variable is generated it is stored in the **workspace** until it is overwritten or deleted with the **clear** command.

```
>> A*B
ans =
1.5708
```

The way the numbers are *displayed* (not computed!) can be set with the command `format`, and a more precise representation can be used (short, long, etc.).

The vectors and matrices can be set in different ways. The simplest method consists on assigning a value to each of the elements of the vector or matrix one by one:

```
>> Vector(1)=2 ; Vector(2)=2; Vector(3)=9
Vector =
2      2      9
```

Notice that the index of a matrix must always be a positive integer number.

```
>> Vector(0)=33
??? Subscript indices must either be real positive integers or
logicals.
```

Another way of initializing a vector is assigning values to all the elements of the vector (or matrix) within square brackets (`[]`). This is done separating the elements of the same row with a comma “,” or a white space, and separating the rows with a semicolon “;”.

```
>> Matrix=[1 2 3 ; 0 0 1; 2 4 6]
```

```
Matrix =
```

```
     1     2     3
     0     0     1
     2     4     6
```

Another practical way of defining vectors is generating a string of values specifying the initial value, the final value and the step size (all separated by a colon ":").

```
>> V2=[10:2:30]
```

```
V2 =
```

```
    10    12    14    16    18    20    22    24    26    28    30
```

We can also use the command **linspace** to create a string of numbers. With this command we have to specify the initial value, the final value and the number of elements we wish the vector to have (all separated by a comma ","):

```
>> time=linspace(3,5,8)
```

```
time =
```

```
    3.0000    3.2857    3.5714    3.8571    4.1429    4.4286
    4.7143    5.0000
```

There are a few commands to initialize vectors or matrices to specific values (zeros, ones, etc.) **zeros**, **ones**, **eye**, **diag**, etc., in which you only have to specify the size of the vector you wish to generate:

```
>> I=eye(3)
```

```
I =
```

```
     1     0     0
     0     1     0
     0     0     1
```

There also exist some **functions** that return matrices as a result of a mathematical operation which we will look into in the next section.

1.4. Built-in functions and operators

There are many MATLAB built-in functions as well as operators that can be used to manipulate variables. The most common operators are arithmetic operators such as addition (+), subtraction (-), matrix multiplication (*), matrix left division (e.g. $x=A \setminus B$ finds which matrix, x , is a solution of the linear system $Ax=B$), matrix right division (/), matrix power (^), etc. The mathematical order in which to perform the operators is the same as arithmetic rules, it also gives preference to mathematical operations within round brackets "()".

```
>> angle=pi/9; num=24; t=12; result=((num +3 * t - angle)^2)/1000

result =

    3.5582
```

Below we can see an example with matrices.

```
>> M=[1 0 ; 0 2] ; V=[3 4]; V*M

ans =

     3     8
```

We can manipulate a vector or a matrix with an **apostrophe** (') or with the command **ctranspose** to obtain its corresponding conjugate transpose, a.k.a. Hermitian. If the aim is to simply obtain the transpose, the command **transpose** or **period and an apostrophe** (.') may be used.

```
>> v1=[1 2 3]; v2=[5 0 3] ; scalar_prod= v1*v2'

scalar_prod =

    14
```

All the matrix operators mentioned above can be applied **element-wise** just by adding a period in front of them (.*)(.\)(./). This way, the operation will be performed along the corresponding elements of the vectors, that is, the operations will be performed between element {i,j} of one matrix and element {i,j} of the other one.

```
>> A=[1 1; 2 5]; B=[9 2; 5 0]; C=A.*B

C =

     9     2
    10     0
```

There are also some functions that work both for scalars and matrices, in which the operations are applied element-wise. Some of the examples are the following: **sin, cos, exp, log, abs, sqrt, round**.

```
>> angle=pi/4; result=sin(angle)

result =

    0.7071
```

We may use the letters “i” or “j” to denote the imaginary unit, i.e. $\sqrt{-1}$. However, it is recommended (for clarity) to use either 1i, or 1j, instead.

```
>> angles=j*[pi/2 pi 3*pi/2]; output=exp(angles)

output =

    0.0000 + 1.0000i   -1.0000 + 0.0000i   -0.0000 - 1.0000i
```

There are also other functions available that return a single scalar value when applied to a vector. Some of these functions are: **max**, **min**, **sum**, **mean**, **std**, ... When applied in matrices, we must specify if we wish to apply it to the columns or the rows (number 1 is used to choose the columns and number 2 to choose the rows). If we don't specify it by default the columns are chosen.

```
>> vector=[3.67 5.51 9.02 3.27 1.91] ; mean(vector)

ans =

    4.6760
```

Matlab also contains specific matrix functions, such as: **eig**, **inv**, **det**, **size**.

```
>> A=[1 1; 2 5]; inverse_A = inv(A)

inverse_A =

    1.6667   -0.3333
   -0.6667    0.3333
```

1.5. Indexing of matrix elements: Submatrices and Concatenation.

Occasionally, we will need to slice part of the data from a vector or matrix. To do so, we can access each element of a matrix $\{i,j\}$ using round brackets, **()**, and a comma separating the different dimensions (i.e. the first dimension goes along the rows, the second dimension goes along the columns, etc.).

```
>> M_Test=[9 -1 0; 2 4 3; 41 42 40] ; M_Test(3,2)

ans =

    42
```

If we wish to access any **submatrices** of a matrix, we can access it with the brackets **()** and a colon **:**. For example, if we wish to the elements of column 2 and 3 from the first row we must type:

```
>> M_Test(1,2:3)

ans =

    -1     0
```

If we don't specify a range of indexes and we simply type the colon operator ":" we will select all the elements of the specified row or column, for example, to select all the rows of the third column we type:

```
>> M_Test(:,3)

ans =

     0
     3
    40
```

The colon operator ":" can also be used to convert a matrix into a unique column. In the following example we also use an apostrophe to make it a row vector:

```
>> M=M_Test(:) '

M =

     9     2    41    -1     4    42     0     3    40
```

Another way to extract submatrices is to use logical operators (e.g. <, >, ==, ~=). Applying a logical operator implies returning the values of the matrix that fulfill the specified condition. Notice that the logical negation corresponds with the tilde symbol, which is obtained by simultaneously pressing **Alt Gr** and **4**, and then the **space bar**.

```
>> M2=M_Test(M_Test >= 3) '

M2 =

     9    41     4    42     3    40
```

To concatenate submatrices we only need to initialize them within square brackets, **[]**, making sure they match in size.

```
>> Matrix=[M_Test ones(3,1); zeros(1,3) 9]

Matrix =

     9    -1     0     1
     2     4     3     1
    41    42    40     1
     0     0     0     9
```

2. Data visualization

As we mentioned in the introduction, one of the strengths of MATLAB is its powerful visualization capabilities.

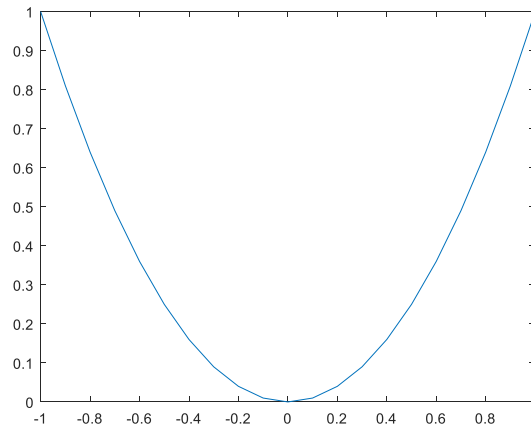
The most important and widely used functions are: **plot** for two dimensional visualization and **mesh**, **surf** and **contour** for three dimensional visualization.

A common usage of the **plot** function is to display vector "x" against "y", where $y=f(x)$.

If we execute the following commands

```
>> x=-1:0.1:1;y=x.^2;  
>> plot(x,y)
```

we will obtain the following figure:

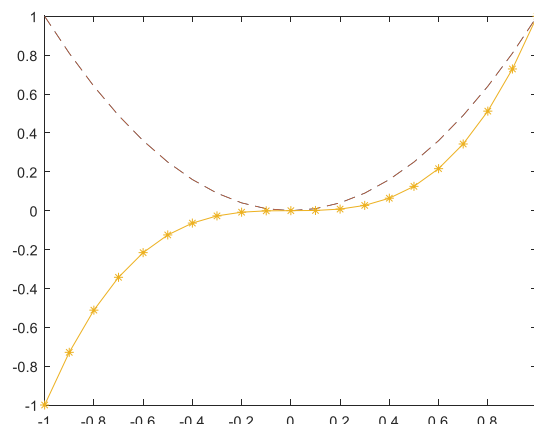



If we don't specify the values of "x" when calling the plot function, the assigned values for the x-axis by default will be the indexes of the displayed vector, that is, 1, 2,..., n.

That is, we are working with a finite number of elements and we are evaluating them. In the figure above we represented the computed elements with a line that links them and so it looks like a smooth function.

If we call the "plot" function again the previous drawing will disappear. If we wish to see several plots together we may use the **hold on** command. And if we wish to clear the figure and draw a new one we can then type the **hold off** command.

```
>>x=-1:0.1:1; y1=x.^2; y2=x.^3;  
>> plot(x,y1,'--');  
>> hold on  
>> plot(x,y1,'--');  
>> plot(x,y2,'*-');
```



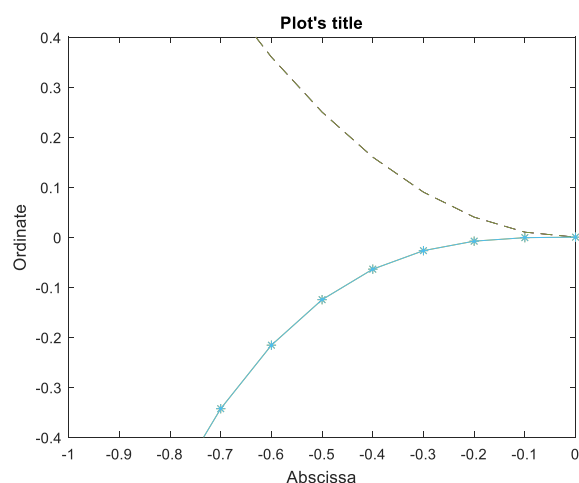
 Note that in some earlier versions of MATLAB, the two curves may be displayed in blue color.


We may represent the elements of a vector with different lines and colors. It is highly recommended that you type **help plot** in the command window to obtain a list of the different available options.

There also exist the commands **xlabel** (used to label the x-axis), **ylabel** (used to label the y-axis) and **title** (to give a name to the figure). The **axis** command controls the range of values we wish to display. You may type the following commands to observe the changes in the figure:

```
>> xlabel('Abcissa')
>> ylabel('Ordinate');
>> title('Plot's title');%' to write a single quote inside a
%string
>> axis([-1 0 -0.4 0.4])
```

You should obtain the following:



 Note that in some earlier versions of MATLAB, the two curves may be displayed in blue color

If we wish to keep the obtained figure but we need to represent a new figure we can type the command **figure** and a new figure window will appear. In fact, the plot command draws in the current active window (selected with a command or with the mouse).

3. Scripts and functions

As we have seen in the previous section, there are certain instructions that we may wish to execute in one go, as a whole unit. We can obviously execute them in the command window however it may be an arduous task.

Scripts solve this problem: they are text files that contain MATLAB instructions written in the same way we would in the command window. They are stored with the extension “.m” and to execute them we only need to type the name of the script file (avoiding the “.m” extension) in the command window. Note that the file **MUST** be stored in the current working directory or folder, else MATLAB will not be able to find it and execute it.

```
% Script to generate the figures depicted
% in the MATLAB manual
x=-1:0.1:1; y1=x.^2; y2=x.^3;
plot(x,y1,'--');
hold on
plot(x,y1,'--');
plot(x,y2,'*-');
xlabel('Abscissa')
ylabel('Ordinate');
title('Plot's title');
axis([-1 0 -0.4 0.4])
```

“GenerateFigures.m”

To call the script we only need to type

```
>> GenerateFigures
```

If we wish to write a **comment** in the script, which obviously will not be executed, we need to put a percentage sign “%” in front of it.

On the other hand, functions allow us to specify **input** and **output** variables to a group of instructions. As with the scripts, functions need to be stored with the extension “.m”. The main difference between scripts and functions is the need to start the text file with the word **function** following the *name by which we stored the file with* and include the input and output variables. For example:

```
% Function that computes the distance between two points P1 and P2
function [modulus,v]=distance(P1,P2)

% Computes the vector that links the two points
v=P2-P1;
% Computes the modulus of the vector
modulus =norm(v,2);
```

“distance.m”

Where the output variable is `modulus` and `v`, and the input variables are `P1` and `P2`. To call the function we would have to type

```
>> P1=[1 2];
>> P2=[-5 8];
>> Dist=distance(P1,P2)

Dist =

    8.4853
```

The main difference between *scripts* and *functions* are to do with the usage of them. Even though for the example shown `P1` and `P2` are variables that have the same value in and out the function that will not be the case always. The scripts use exactly the same workspace that existed when it was called whilst in the functions the variables are local and they are deleted when we come out of the function (except from the output variables). Therefore, after calling “GenerateFigures”, the variables `x`, `y1` and `y2` will still be available in the workspace.

Notice that within scripts and functions is we can perform debugging, using a similar interface as in Visual C o Eclipse we may *pause* the execution, explore and modify the content of the variables, etc. To debug a script or a function, you can put “breakpoint” in a line of the editor using the specified icon or the **F12** button, which will stop the execution in that particular line and will give you access to the command window.

4. Loops and data flow control

MATLAB have some built-in commands to create loops or to diverge the execution. Note that even though in languages like C to perform certain calculations is necessary and recommended to use loops, in MATLAB you can perform many operations in a single line using matrix computations.

In fact, MATLAB is faster performing matrix operations rather than loops, therefore is highly recommended to avoid loops if possible.

If is necessary to perform loops we may use the commands **for** and **while**.

```
% Plot 3 sinusoids of various different frequencies
x=0:0.01:0.5;
for f=2:0.5:3
y=sin(2*pi*f*x);
plot(x,y), hold on;
end
```

```
% Compute Euler's number with an error smaller than 0.01
n=1;
error=1;
while (abs(error)>0.01)
aprox=(1+1/n)^n;
error=aprox-exp(1)
n=n+1
end
```

We may also use **if** and **switch** and specify more conditions nesting “**if**” with “**else**” and/or “**elseif**”. Note that **if**, **for**, **while** and **switch** statements need to be finished with “**end**”.

```
% Function returning relative dimension of a matrix

function MatrixDimension(A)

Width=size(A,2)
Hight=size(A,1);
if (Width> Height)
    disp 'Width larger than height'
elseif (Width< Height)
    disp 'Width smaller than height'
else
    disp 'Square matrix'
end
```

5. Exercises

Exercise 1

Write a MATLAB function named *generate_sinusoid* (and remember to save it as *generate_sinusoid.m*), that generates and display sinusoidal signals (use the MATLAB built-in **sin** function to compute signals). The function should have as input variables: initial time instant (*n0*), final time instant (*n1*), sampling period (*step*), angular frequency (*w0*) and phase (*phi*). The function should return the generated sequence (*y*) and the corresponding time reference vector (*ref*), as well as a plot the signal in a figure.

Exercise 1.1

Use the function defined in Exercise 1 and the MATLAB built-in command **subplot** to draw in a **single figure** the sinusoids corresponding to frequencies: π , 2π and 4π . Use 0 as the initial time instant and 2 as the final time instant. Set *phi*=0 in all cases.

Exercise 2

Write a MATLAB function named *generate_exponential* (and remember to save it as *generate_exponential.m*), that generates and displays sinusoidal signals from the following input variables: initial time instant (*n0*), final time instant (*n1*), sampling period (*step*) and the base of the exponential (*b*). The function should return the generated sequence (*x*) and the corresponding time reference vector (*ref*), as well as a plot the signal in a figure.

Exercise 2.1

Please use the function defined in Exercise 2 and the MATLAB built-in command **subplot** to draw in a **single figure** the exponential signals corresponding to bases: $\frac{1}{2}$, $\frac{1}{4}$ and e . Use 0 as the initial time instant and 2 as the final time instant.

Exercise 3

Write a script in which the following instructions are executed sequentially.

- Define a discrete time vector, *N*, that takes values between -50 and +50, in which the time elapsed between two consecutive samples is one time unit.
- Define the signal named *pulse*, which will be a pulse of amplitude one for the time instants between -10 and +10.
- Define the signal named *deltas*, which should be the sum of two deltas: one delta displaced 20 units to the right and a delta displaced 20 units to the left and multiplied by -1.
- Compute the convolution of these two signals (*deltas* and *pulse*) using the function **conv**, and store the result in *result1*.
- Draw the signal *result1* using the command **stem**.

Exercise 4

Generate and draw each one of the following sequences:

$$x_1[n] = 3 \sin \left(\frac{\pi}{7} n \right) + j 4 \cos \left(\frac{\pi}{7} n \right), \quad 0 \leq n \leq 20$$

$$x_2[n] = (1.1)^n \cos \left(\frac{\pi}{11} n + \frac{\pi}{4} \right), \quad 0 \leq n \leq 50$$

In the case of the complex signal, draw (using the command **subplot**) the real and imaginary parts. Then, in a different figure, use **subplot** again to draw the modulus (using the **abs** command) and the phase in radians (**angle**).

Next, express the sinusoids by means of complex exponentials (using Euler's formula) and generate two new vectors, *x11* and *x22* with the function *exp*. Once again plot the real and imaginary parts, the modulus (*abs*) and phase (*angle*), and check that results match those obtained in the first part.