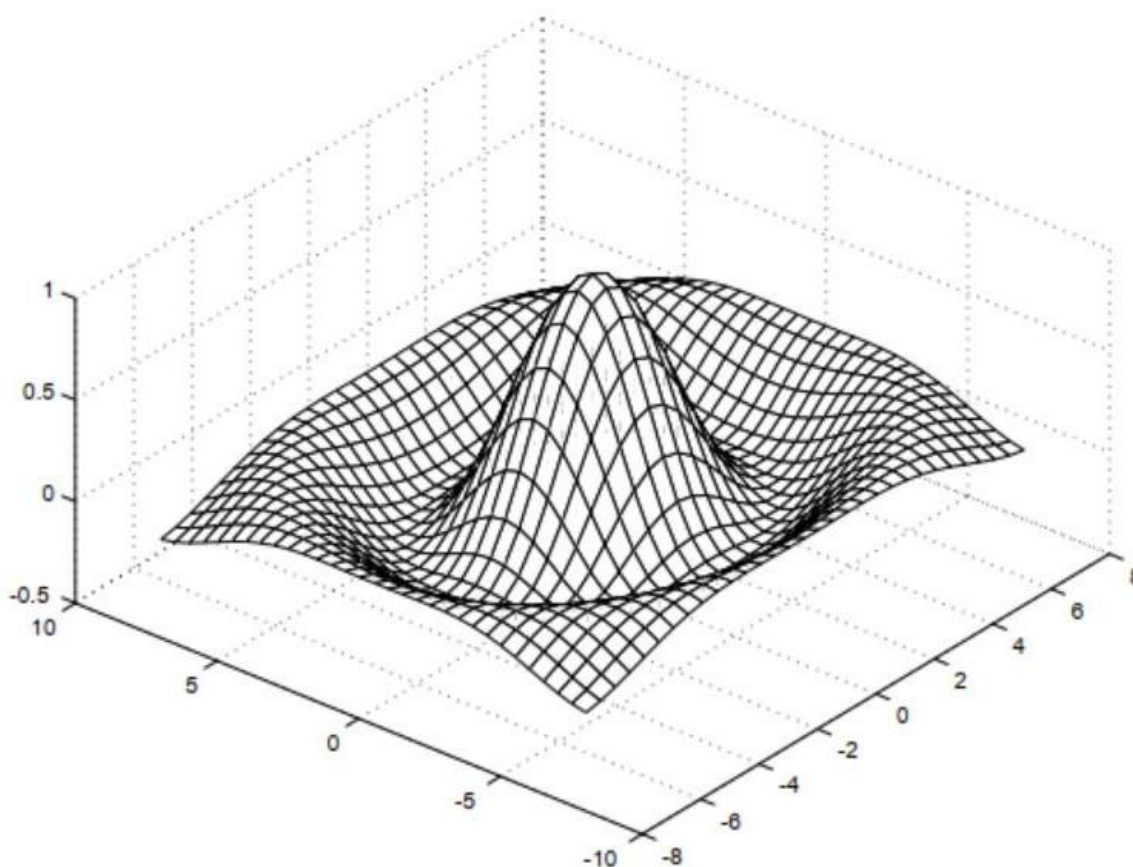


Programming Tools PT 2

(MATLAB)

Tutorial for Second-Year Math Students



Assma Leulmi
University of Ferhat Abbas Setif-1
Faculty of sciences
DEPARTEMENT OF MATHEMATICS



Table of Contents

Introduction.....	1
Chapter I: Introduction to the Matlab environment.....	3
1. Introduction and Opening a Matlab Session.....	3
1. General.....	3
2. MATLAB environnement.....	4
A. Editor/Debugger	6
B. M-Files	6
D. Help	8
2.1 First interaction with MATLAB.....	9
2.2 The numbers in MATLAB.....	11
2.3 The main constants, functions and commands.....	14
2.4 The priority of operations in an expression	15
TP1	17
Exercice 01	17
Exercice 02	17
Exercice 03	18
Exercice 04	18
Solution TP 1	21
Chapter II: Vectors and Matrices	21
1. The vectors.....	21
1.1 Referencing and access to vector elements.....	23
1.2 Element-by-element operations for vectors.....	24
1.3 The linspace function.....	25
2. The matrices	26
2.1 Referencing and access to matrix elements	28
2.2 Automatic generation of matrices	30
Example 1:	32
Example 2:	32
2.3 Basic operations on the matrices.....	32

2.4	Useful functions for matrix processing.....	33
TP2		36
Exercise 01		36
Exercise 02		36
Exercise 03		36
Exercise 04		37
Exercise 05		37
Exercise 06		37
Exercise 07		38
Solution TP 2		39
Chapter III: Introduction to programming with Matlab		45
1.	General.....	45
1.1	Comments.....	45
1.2	Writing long expressions.....	45
1.3	Reading data in a program (Inputs)	46
1.4	Writing data in a program (Outputs).....	46
2.	Logical expressions.....	47
2.1	Logical operations	47
2.2	Matrix comparison	50
3.	Flow control structures	51
3.1	The if statement.....	52
3.2	The switch statement	54
3.3	The for statement.....	55
3.4	The while statement	56
	Example of a <code>for</code> loop in MATLAB	57
	Replacing the <code>for</code> loop with a <code>while</code> loop in MATLAB.....	57
4.	Summary the control structures.....	58
5.	Summary exercise.....	59
6.	The functions.....	60
6.1	Creating a function in an M-Files	60
6.2	Comparison between a program is a function	61

6.3	Exercices with Solution.....	64
7.	Polynomials:	78
7.1	Polynomials in MATLAB	78
7.2	Polynomial zeros.....	78
7.3	Polynomial operations	78
Chapter V: Graphs and data visualization in Matlab.....		80
1.	The plot function	81
2.	Change the appearance of a curve	83
3.	Annotation of a figure.....	84
4.	Draw multiple curves in the same figure	84
4.1	The hold command.....	84
4.2	Use plot with multiple arguments	85
4.3	Using matrices as argument for the plot function.....	86
4.4	Using the fplot function	86
5.	Manipulating the axes of a figure	87
6.	Other types of graphs.....	89
7.	Transfer figures to a Word document:	92
8.	Figure Editor:	93
	Exercise	94
9.	Symbolic Calculation.....	95
9.1	Calling the symbolic toolbox	95
9.2	Expanding and transforming expressions	95
9.3	Derivatives and integrals of a function.....	95
9.4	Taylor series expansion of a function	95
TP3.....		96
	Exercice 01	96
	Exercice 02	94
	Exercice 03	96
	Exercice 04	95
	Exercice 05	97
	Exercice 06	97
	Exercice 07	98

Solution TP 3	99
Command Catalogue	108
Content of the subject	114
Bibliography	117

Introduction

Mathematicians, faced with increasingly complex and large-scale problems, encounter significant challenges when trying to solve equations or model complex systems. These problems can be too vast or abstract to solve by hand. That's why programming tools are essential. They allow for automating calculations, handling large datasets, and simulating complex phenomena, greatly facilitating mathematical research and discoveries. These tools enable mathematicians to focus on theoretical analysis while relying on computational power to solve larger and more difficult problems.

Programming tools play a crucial role in the evolution of mathematics. In fact, computational methods and software have significantly transformed the way mathematicians, scientists, and engineers approach mathematical problems. Mathematics often deals with complex problems that may be impractical or impossible to solve by hand. Programming allows for the handling of large data sets, performing symbolic computations, and simulating complex systems that would be unmanageable manually. For example:

- Numerical Analysis: Solving equations and optimizing functions in cases where analytical solutions don't exist.
- Simulation and Modeling: Tools like MATLAB, Scilab and Python (with NumPy and SciPy), and R are used to simulate real-world phenomena in physics, biology, economics, etc.

Programming tools (MATLAB, Scilab and Python) are very important for mathematics students. They help by:

1. Simplifying complex calculations: They solve equations and handle complex math problems easily.
2. Visualization: They allow students to plot graphs and visualize mathematical concepts.
3. Practical learning: Students can experiment with numerical methods and apply them to real-world problems.

In short, MATLAB and Scilab are essential for math students, as they simplify calculations, simulations, and modeling.

This booklet serves as a support for the Programming Tools (Matlab) course in the second year of the Bachelor's degree in Mathematics and Technology (ST), Bachelor's degree in Material Sciences (SM) (Physics and Chemistry), and engineering training programs (ST, ME, etc.). The objective of this course is to familiarize students with a set of tools (algorithms) aimed at efficiently solving various problems.

The chapters are illustrated with application examples, and a series of exercises is provided at the end of each chapter.

It covers: Introduction to the Matlab environment, Vectors and matrices,

Introduction to programming with Matlab, and Graphs and data visualization in Matlab.

Given the time allocated to this unit (1 lecture + 1 practical session) per week, the short duration of the third semester (S3), and the fact that the students are not specialists in computer science, we have chosen not to go into too much detail. Interested readers may refer to the literature for further details.

We have focused our efforts on Matlab in order to enable students to implement it with their instructors on their computers during the practical sessions (TP).

In preparing this handout, we have wisely used a set of references, presented at the end of the document, to deepen the topics discussed. Finally, please let me know if you spot any errors, whether in the content or in the format of this handout.

MATLAB

Chapter I: Introduction to the Matlab environment

1. Introduction and Opening a Matlab Session:

Matlab is software designed for optimizing scientific calculations. Initially developed for matrix computation, hence the abbreviation MATrix LABoratory, Matlab enables problem-solving through algorithms, graphs, simulations, and more.

The following tutorial is a brief introduction to the world of Matlab. It will cover the essential concepts needed to navigate Matlab.

Note: The examples in this tutorial were created in m-files. This explains why the instructions are grouped and separated from the results displayed.

1. General:

MATLAB (MATrix LABoratory) is an interactive programming environment for scientific computing, programming and data visualization.

It is widely used in the fields of engineering and scientific research, as well as in higher education institutions. Its popularity is mainly due to its strong and simple interaction with the user but also to the following points:

- ✓ Its functional richness: with MATLAB, it is possible to perform complex mathematical manipulations by writing few instructions. It can evaluate expressions, draw graphs and run classic programs. Above all, it allows the direct use of several thousand predefined functions.
- ✓ The ability to use toolboxes (toolboxes): which encourages its use in several disciplines (simulation, signal processing, imaging, artificial intelligence ...etc.).
- ✓ The simplicity of its programming language: a program written in MATLAB is easier to write and read compared to the same program written in C or PASCAL.
- ✓ Built-in Functions: It includes thousands of predefined functions for a variety of mathematical tasks, such as trigonometric functions (like sin and exp), linear algebra, statistics, numerical analysis and optimization.
- ✓ Its way of managing everything as matrices, which frees the user to deal with data typing and thus avoid the problems of transtyping.

Originally MATLAB was designed to make mainly calculations on vectors and matrices hence its name '**Matrix Laboratory**', but subsequently it was improved and increased to be able to deal with many more areas.

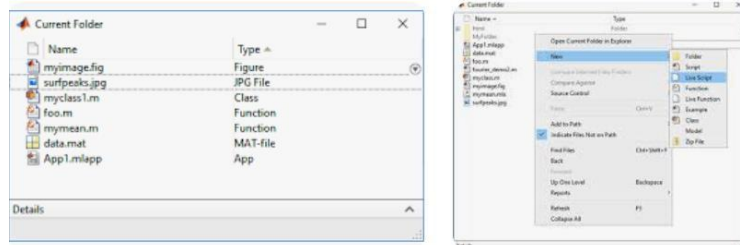
MATLAB is not the only scientific computing environment available because there are other competitors, the most important of which are Maple and Mathematica. There are even free software that are clones of Matlab like Scilab and Octave.

EXERCISE: Start MATLAB.

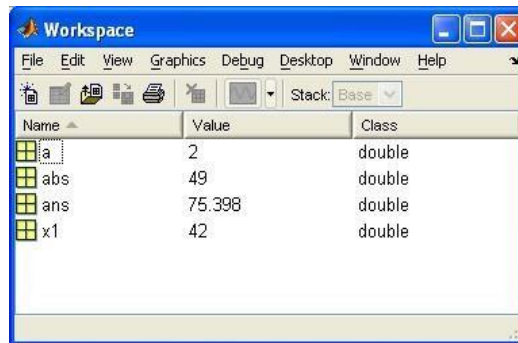
2. MATLAB environnement:

Currently MATLAB is at version 7.x and at startup it displays several windows. Depending on the version you can find the following windows:

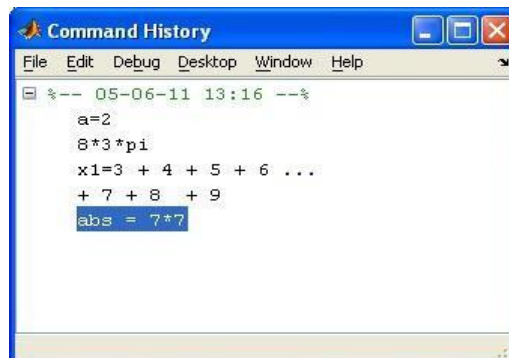
- **Current Folder:** indicates the current directory and existing files.



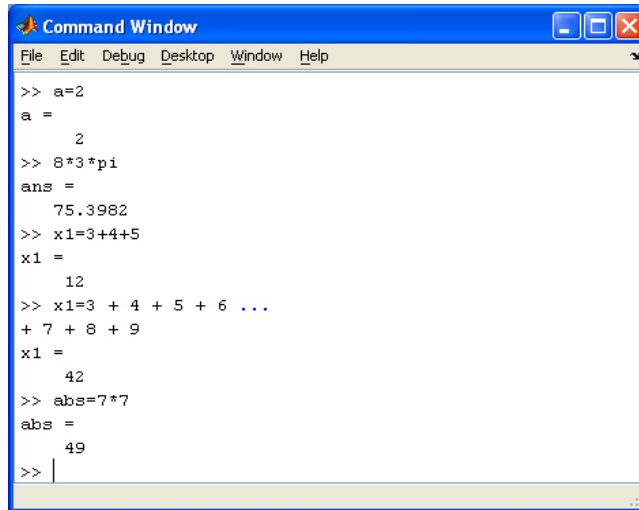
- **Workspace:** indicates all existing variables with their types and values.



- **Command History:** keeps track of all commands entered by the user.



- **Command Window:** we use to formulate our expressions and interact with MATLAB, and this is the window we use throughout this chapter.



```

>> a=2
a =
    2
>> 8*3*pi
ans =
   75.3982
>> x1=3+4+5
x1 =
    12
>> x1=3 + 4 + 5 + 6 ...
+ 7 + 8 + 9
x1 =
    42
>> abs=7*7
abs =
    49
>>
    
```

The MATLAB interface corresponds to this one.

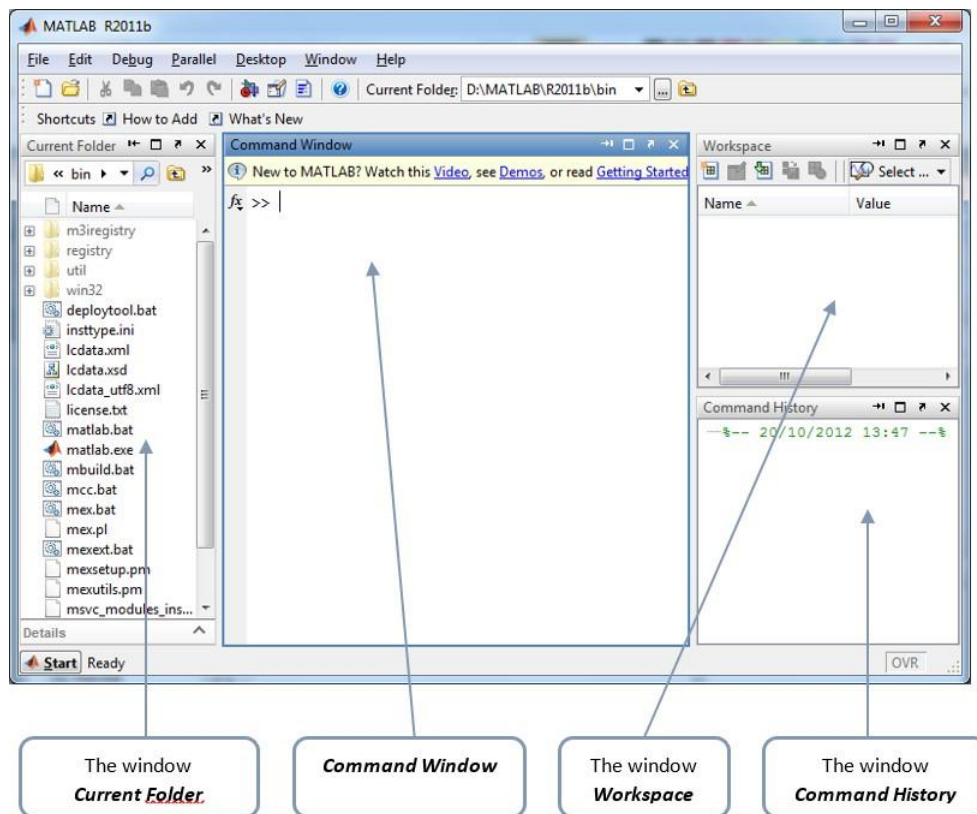
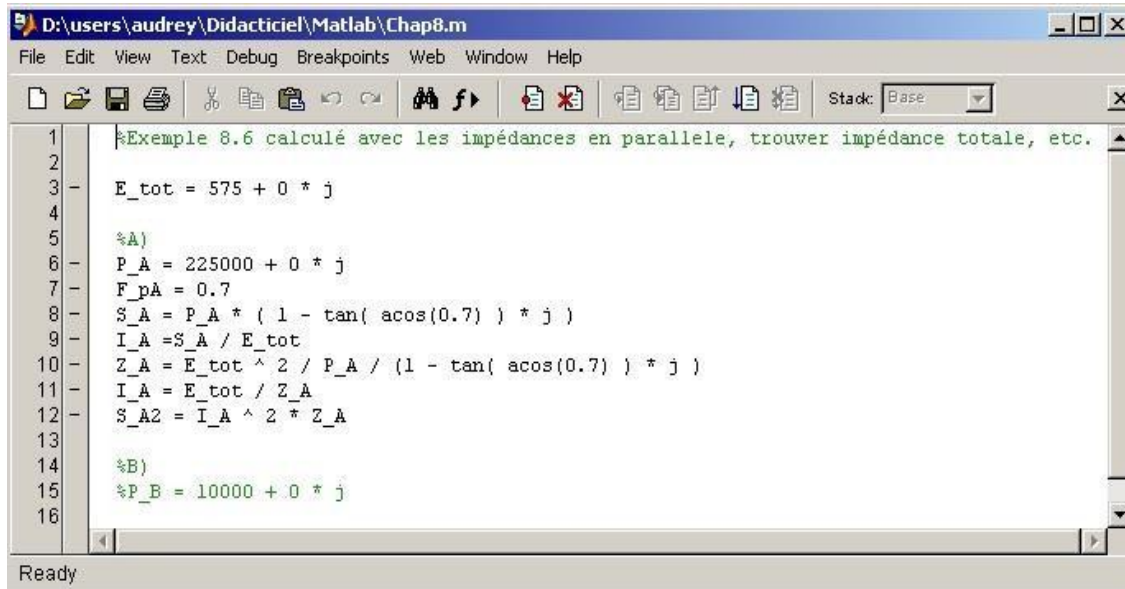


Figure 1 : MATLAB environnement (version 2011b where 7.13)

A. Editor/Debugger:

To work with M-files, you need to use MATLAB's editor/debugger. This window is not part of the basic MATLAB interface and opens when an M-file is opened or when a new M-file is created.



On the toolbar, one button is essential: the RUN button compiles the program, i.e., it executes the commands in the program. It can also be executed with F5.

Another useful feature of the Editor is that if you hover over a variable, its value appears (if the program has been compiled at least once).

Demonstrations:

For step-by-step demonstrations on the MATLAB environment, open MATLAB, then type `demo` in the command window. Select **MATLAB - Desktop Tools and Development Environment**. You will find about half a dozen demonstrations.

B. M-Files:

To avoid retyping a series of commands, you can create a MATLAB program, known as an "M-file," with the name derived from the ".m" extension of these files. Using the MATLAB editor, you create a text file that contains a series of MATLAB commands. To create an M-file, go to the menu **File → New → M-File** or click on the blank page icon. The file is typically saved in the current directory. Once the file is saved (e.g., under the name `filename.m`), you can call it in MATLAB using the command:

```
>> filename
```

The commands stored in the M-file will then be executed, and the results will appear in the Command Window. If you need to modify your series of commands, simply edit the corresponding line in the M-file and re-run the M-file by typing the file name in MATLAB again (try the `↑` key).

Other ways to execute the M-file include clicking on the Run button, going to the Debug → Run menu in the Editor/Debugger, or pressing F5.

M-files prevent you from having to repeatedly type the same commands and allow you to save your instructions, commands, and calculations. This is the recommended procedure for your practical assignments.

EXERCICE:

Creating an M-File – Procedure:

- Create a file using the button or the menu.
- Write some instructions. For example, write the following instructions:

```
A=8  
B=7  
C=A+B
```

- Save the M-file in the current directory. If you don't, MATLAB will prompt you to save it before compiling.
- Return to MATLAB: the results should appear in the command window. These results should be:

```
>> A =  
      7  
B =  
      9  
C =  
     16
```

C. The ";" and "..."

The semicolon at the end of a line tells MATLAB not to display the result of the operation on the screen. A common practice is to place semicolons at the end of all lines and remove some of them when something isn't working as expected in our program, in order to see what's going on.

In the editor/debugger as well as in the command window, the use of "..." is useful to continue the current instruction onto the next line.

The two instructions are identical except for the semicolon at the end of the second one 2e.

```
>> B = pi *
3^2 B =
      28.274
>> B = pi * 3^2;
>>
```

The following two instructions give the same result.

```
>> A = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8
+ 9 A =
      45
>> A = 1 + 2 + 3 + 4 + 5 ...
+ 6 + 7 + 8 + 9
A =
      4
      5
```

D. Help:

1. Help:

In addition to the basic features of MATLAB, a vast library of functions (called 'toolboxes' in MATLAB language) is available to you. To get a list of available function families, enter the command `help`. To see the list of functions within a specific family, you can enter `help matfun`, for example, to view the list of matrix functions. To get information on a specific function, simply use the `help` command followed by the function name, such as `help sin` to get information on the sine function. If the function has not been compiled to improve execution speed, you can view the source code by entering `type arrow`, for example.

2. Finding a Function:

The `lookfor` command allows you to search through the documentation directories for all functions containing the searched word. For example, to see the list of functions related to randomness, you can enter `lookfor random` and it will return 13 functions that use randomness.

2.1 First interaction with MATLAB:

The easiest way to use MATLAB is to write directly to the command window (Command Window) right after the cursor (prompt) >>

MATLAB can be seen as an extremely powerful calculator. Simple operations can be entered directly, and the result is obtained by pressing the "Enter" key.

To calculate a mathematical expression just write it like this:

```
>> 7+6
ans =
    13
```

Then click on the Enter key to see the result

If we want an expression to be calculated but without displaying the result, we add a semicolon ';' at the end of the expression as follows:

```
>> 7+6 ;
>>
```

To create a variable we use the simple structure: 'variable = definition' without worrying about the type of the variable.

For example:

```
>> a=10 ;
>> u=cos(a) ;
>> v=sin(a) ;
>> u^2+v^2
```

```
ans =
    1
```

```
>> ans+11
```

```
ans =
    12
```

```
>>
```

It is possible to write several expressions in the same line by making them separated by commas or semicolons. **For example:**

```
>> 7+6, 2*4-1, 12-2
```

```
ans =
    13
ans =
    7
```

```
ans =
    10
>> 5+6; 2*4-1, 12-2;
ans =
     7
>>
```

The name of a variable must contain only alphanumeric characters or the symbol '_' (underscore) and must start with an alphabet. We must also pay attention to capital letters because the MATLAB is case-sensitive (**A** and **a** are two different identifiers).

The basic operations in an expression are summarized in the following table:

Operation	meaning
+	addition
-	subtraction
*	multiplication
/	division
\	Left division (or reverse division)
^	power
'	The transposed
(and)	Parentheses specify the order of evaluation

To see the list of variables used, either look at the window '**Workspace**' we use the commands either 'whos' or 'who':

whos gives a detailed description (variable name, type and size), but **who** just give the names of the variables.

For example, in this course we used 3 variables **a**, **u** and **v**:

```
>> who
Your variables are:
ans u      v

>> whos
      Name      Size      Bytes  ClassAttributes
      a         1x1         8    double
      ans        1x1         8    double
      u          1x1         8    double
      v1x1       1x1         8    double
```

```
>> who
```

```
DATAE DATAG DATAI DATAR E I
```

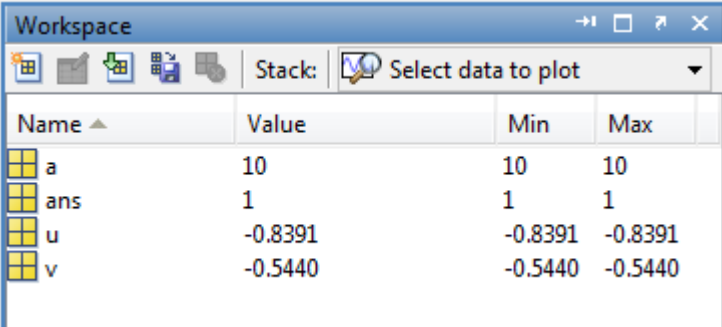
```
>> whos
```

Name	Size	Bytes	Class
DATAE	1x7	56	double array
DATAG	7x7	392	double array
DATAI	1x7	56	double array
DATAR	7x7	392	double array
E	7x1	56	double array
I	7x1	56	double array

Grand total is 126 elements using 1008 bytes

Let's see that all the objects created are matrices with components stored as double-precision real numbers, even those we entered as integers.

The use of these two commands can be omitted cards variable information are visible directly in the workspace window.



Name	Value	Min	Max
a	10	10	10
ans	1	1	1
u	-0.8391	-0.8391	-0.8391
v	-0.5440	-0.5440	-0.5440

2.2 The numbers in MATLAB:

MATLAB uses conventional decimal notation, with an optional decimal point '.' and the sign '+' or '-' for signed numbers. Scientific notation uses the letter 'e' to specify the power scale factor of 10. Complex numbers use characters 'i' and 'j' (indifferently) to design the imaginary part. The following table gives a summary:

type	Exemples
Integer	5 -83
Real in decimal notation	0.0205 3.1415926
Real in scientific notation	1.60210e-20 6.02252e23 (1.60210x10 ⁻²⁰ et 6.02252x10 ²³)
Complex	5+3i -3.14159j

MATLAB always uses real numbers (double precision) to make the calculations, which allows obtaining a calculation accuracy of up to 16 significant digits.

But it should be noted the following points:

- The result of a calculation operation is displayed by default with four digits after the decimal point.
- To display more numbers use the command **format long** (14 digits after the decimal point).
- To return to the default view, use the **format short**.
- To display only 02 digits after the decimal point, use the **format bank**.
- To display the numbers as a ration, use the **format rat**.

control	meaning
format short	displays numbers with 04 digits after the decimal point
format short e	Scientific with 04 digits.
format long	displays numbers with 14 (or 16) digits after the decimal point
format long e	Scientific with 14 (or 16) digits.
format bank	displays numbers with 02 digits after the decimal point
format hex	Hexadecimal
format +	Use the symbols +, - and space to display positive, negative, and zero numbers
format rat	displays the numbers as a ration (a/b)

Example:

```
>> 8/3
      ans =
      2.6667

>>format long

>> 8/3
      ans =
      2.666666666666667

>>format bank

>> 8/3
      ans =
      2.67

>>format short

>> 8/3
      ans =
      2.6667

>> 7.2*3.1
      ans =
      22.3200

>>format rat

>> 7.2*3.1
      ans =
      558/25

>> 2.6667
      ans =
      26667/10000
```

vpa function can be used to force the compute to have more significant decimals by specifying the desired number of decimals.

Example :

```
>> sqrt(2)
      ans =
      1.4142

>> vpa(sqrt(2),50)
      ans =
      1.4142135623730950488016887242096980785696718753769
```

2.3 The main constants, functions and commands:

MATLAB defines the following constants:

Constant	its value
Pi	$\pi=3.1415...$
$exp(1)$	$e=2.7183...$
I	$= \sqrt{-1}$
J	$= \sqrt{-1}$
Inf	∞
NaN	Not a Number (Pas un numéro)
Eps	$\varepsilon \approx 2 \times 10^{-16}$

Frequently used functions include:

For example:

The notation **exp(x)** represents the exponential function, which is defined as e^x , where e is Euler's number (approximately equal to 2.71828). This function is commonly used in mathematics to model growth processes, such as compound interest or population growth. In MATLAB, you can calculate the exponential of xxx using the `exp` function.

MATLAB defines the following function:

function	its meaning
$\sin(x)$	I the sine of x (in radian)
$\cos(x)$	I Cosine of x (in radian)
$\tan(x)$	The tangent of x (in radian)
$\sin^{-1}(x)$	The arc sine of x (in radian).(the inverse sine function $\sin^{-1}$ of x)
$\cos^{-1}(x)$	The arc cosine of x (in radian). (the inverse sine function $\cos^{-1}$ of x)
$\tan^{-1}(x)$	The arc tangent of x (in radian).(the inverse tangent function $\tan^{-1}$ of x)
$\sqrt{x}$	The square root of x $\rightarrow \sqrt{x}$
$ x $	The absolute value of x $\rightarrow x $
e^x	The exponential function, which is defined as e^x

log(x)	Natural logarithm of x $\rightarrow \ln(x)=\log_e(x)$
log10(x)	Logarithm based on 10 of x $\rightarrow \log_{10}(x)$
imag(x)	The imaginary part of the complex number x
real(x)	The actual part of the complex number x
round(x)	Round a number to the nearest integer
floor(x)	round a number to the smallest integer $\rightarrow \max\{n \mid n \leq x, n \text{ entire} \}$
ceil(x)	round a number to the largest integer $\rightarrow \min\{n \mid n \geq x, n \text{ entire} \}$

MATLAB offers many commands for user interaction. For the moment, we are content with a small set, and we will expose the others as the course progresses.

control	its meaning
who	Displays the name of the variables used
whos	Displays information about the variables used
clear x y	Deletes the x and y variables
clear, clear all	Deletes all variables
clc	Clears the control screen
exit, quit	Close the MATLAB environment
format	Sets the output format for numeric values <code>format long</code> : displays numbers with 14 digits after the decimal point <code>format short</code> : displays numbers with 04 digits after the decimal point <code>format bank</code> : displays numbers with 02 digits after the decimal point <code>format rat</code> : displays the numbers as a ration (a/b)

2.4 The priority of operations in an expression:

The evaluation of an expression runs from left to right considering the priority of the operations indicated in the following table:

operations	priority (1=max, 4=min)
The parentheses ^ (and)	1
Power and the transposed ^and ‘	2
Multiplication and division * and /	3
Addition and subtraction + and -	4

For example:

$5+2*3 = 11$ and $2*3^2 = 18$

Summary exercise:

Create an x variable and set it to 2, then write the following expressions:

- $3x^3-2x^2+4x$
- $\frac{e^{1+x}}{1-\sqrt{2x}}$
- $|\sin^{-1}(2x)|$
- $\frac{\ln(x)}{2x^3}-1$
- $\cos(x^5)-52$

Solution:

```
>> x=2 ;
>> 3*x^3-2*x^2+4*x ;
>> exp(1+x)/(1-sqrt(2*x)) ;
>> abs(asin(2*x)) ;
>> log(x)/(2*x^4)-1 ;
>> cos(x^5)-52 ;
```

TP1

Exercice 01:

Guess the answer given by Matlab for each of the following commands:

```
>> a=5 ; b=a+2 ; c=b-3 ; clear a, who
```

```
>> a=-2.5 ; B=a+2, A=B ; B=A*2
```

```
>> temp=27.48 ; poids= 15.63 ; floor(temp), ceil(poids) ;  
round(poids)
```

```
>> var1=7+3^2 ; var2=8\var1 , var1+var2 ; ans/6
```

```
>> sqrt(2), format bank, sqrt(2), 3/2
```

```
>> format rat, sin(pi/6)
```

Exercice 02:

Provide Matlab commands to evaluate the following expressions:

$$a) -x^5 - \frac{7}{3}x^3 + x^2 + 1, \text{ for } x = 1 \quad e) \frac{x^3 \sin(\frac{3\pi}{4})}{\cos(2\pi-1)}, \text{ for } x = e^3$$

$$b) \frac{5+x^2}{3x-4}, \text{ for } x = 2 \quad f) -2\ln(5x) + \sqrt{4x^3+1}, \text{ for } x = -3i$$

$$c) \frac{2\sqrt{x+1}}{e^{x+3}+5}, \text{ for } x = 3.2 \quad g) (0.5 + 12i)^3 + 4x, \text{ for } x = 2 - 3j$$

$$d) \frac{(\sin(e^x) + 2)\sqrt{\sqrt{|x|} + 1}}{\tan^{-1}(x^2) + (\ln(\sqrt{|x|} + 1))^2}, \text{ for } x = -1.5$$

Exercice 03:

1. Propose Matlab commands to generate the following vectors:

$$V_1 = [2, 3, 4, \dots, 9, 10]$$

$$V_4 = [\frac{1}{99}, \frac{1}{97}, \dots, \frac{1}{5}, \frac{1}{3}, 1]$$

$$V_2 = [-1.5, 0, 1.5, \dots, 4.5, 6]$$

$$V_5 = [\frac{1}{4}, \frac{1}{16}, \frac{1}{36}, \frac{1}{64}, \frac{1}{100}]$$

$$V_3 = [1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \dots, \frac{1}{81}, \frac{1}{100}]$$

2. Create a row vector U that starts at $-\pi/3$ and ends at $7\pi/3$, containing exactly 4 uniformly spaced elements.

3. Create a vector V that contains all the elements of the vectors V_1, V_2 and V_3 consecutively.

4. Propose a Matlab command to reverse the elements of the vector U.

5. Propose a Matlab command to display the elements of the vector V from the 5th position to the 11th in reverse order.

6. Propose a Matlab command to display the second third of the vector V.

7. Propose a Matlab command to display the last quarter of the vector V.

Exercice 04:

Propose instructions (as simple as possible) to produce the matrix A of size 50×50 with the following shape:

$$\begin{bmatrix} \pi & 0 & 0 & \cdots & -1 \\ 0 & \pi & 0 & 0 & 0 \\ 0 & 0 & \pi & 0 & 0 \\ \vdots & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \pi \end{bmatrix}$$

Solution of TP 01

Exercice 01:

Guess the response given by Matlab for each of the following commands:

```
>> a=5 ; b=a+2 ; c=b-3 ; clear a, who
```

Your variables are:

b c

```
>> a=-2.5 ; B=a+2, A=B ; B=A*2
```

B =

-0.5000

B =

-1

```
>> temp=27.48 ; poids=15.63 ; floor(temp), ceil(poids) ; round(poids)
```

ans =

27

ans =

16

```
>> var1=7+3^2 ; var2=8\var1 , var1+var2 ; ans/6
```

var2 =

2

ans =

3

```
>> sqrt(2) , format bank, sqrt(2), 3/2
```

ans =

1.4142

ans =

1.41

ans =

1.50

```
>> format rat, sin(pi/6)
```

ans =

1/2

Exercice 04 :

```
>> A = pi*eye(50); A(50,1)=1; A(1,50)=-1
```

Exercise 02 :

```

a) >> x=1; -x^5-(7/3)*x^3+x^2+1
b) >> x=2; (5+x^2)/(3*x-4)
c) >> x=3.2; (-2*sqrt(x)+1)/(exp(x+3)+5)
d) >> x=-1.5;
>> ((sin(exp(x))+2)*sqrt(sqrt(abs(x))+1))/(atan(x^2)+log(sqrt(abs(x))+1)^(3/2))
e) >> x=exp(3); (x^3*sin(3*pi/4)^2)/cos(2*pi-1)
f) >> x=3i; -2*log(5*x)+sqrt(4*x^3+1)
g) >> x=2-3j; (0.5+12i)^3+4*x

```

Exercise 03 :

1. Creation of vectors v_1 , v_2 , v_3 , v_4 and v_5 :

```

>> v1 = [2:10]
>> v2 = [-1.5:1.5:6]
>> v3 = (1./[1:10]).^2
>> v4 = 1./[99:-2:1]
>> v5 = 1./([2:2:10].^2)

```

2. Creation of the vector U:

```

>> U = linspace(-pi/3 , 7*pi/3 , 4)

```

3. Creation of the vector V:

```

>> V = [v1,v2,v3]

```

4. Invert the vector U:

```

>> U(end:-1:1)

```

5. Display the elements of V from the 5th position to the 11th in reverse order:

```

>> V(11:-1:5)

```

6. Display the second third of the vector V:

```

>> tiersPosition = round(length(V)/3)
>> V(tiersPosition+1 : 2*tiersPosition)

```

7. Display the last quarter of the vector V:

```

>> quartPosition = round(length(V)/4)
>> V(3*quartPosition+1 : end)

```

MATLAB

Chapter II: Vectors and Matrices

Matlab was initially created to provide mathematicians, scientists, and engineers with a user-friendly way to work with linear algebra. As a result, handling vectors and matrices in Matlab is both intuitive and efficient.

The name "MATLAB" stands for matrix laboratory, and as the name suggests, the foundation of the software is matrices and vectors. A matrix allows multiple values to be stored at once, with each value being accessible by its position. For example, in the following matrix, each value can be accessed independently.

1. The vectors:

A vector is an ordered list of elements. If the elements are arranged horizontally, we say that the vector is a vector row, on the other hand if the elements are arranged vertically we say that it is a vector column.

To create a vector line just write the list of its components in square brackets [and] and separated by spaces or commas as follows:

```
>> V = [5, 2, 13, -6]           % Create a vector line V
      V =
         5         2        13        -6
```

```
>> U = [4 -2 1]                % % Create a vector line U
      U =
         4        -2         1
```

To create a column vector, you can use one of the following methods:

1. Write the components of the vector in square brackets [and] and semicolons separated (;) as follows:

```
>> U = [4;-2;1]                % Creating a column vector U
      U =
         4
        -2
         1
```

2. Write vector vertically :

```
>> U = [
4
-2
1
]
      U =
         4
        -2
         1
```

3. Calculate the transpose of a line vector:

```
>> U = [4-21]' % Creating a column vector U
U =
     4
    -2
     1
```

If the components of a vector **X** are ordered with consecutive values, it can be represented using the following notation:

X = first_element : last_element (Square brackets are optional in this case)

This generates a vector **X** that starts from **first_element** and increments by 1 until it reaches **last_element**. For example:

X=1: 5

This will result in the vector:

X=[1 , 2 , 3 , 4 , 5]

For example:

```
>> X=1:8 % you can also write colomn (1,8)
X =
     1     2     3     4     5     6     7     8
>> X = [1:8]
X =
     1     2     3     4     5     6     7     8
```

If the components of a vector **X** are ordered with consecutive values but with a pitch (increment/decrement) different from 1, the pitch can be specified using the following notation:

X = first_element : step: last_element (Brackets are optional)

Or: **X=start : pitch : end**

For example, for a vector with values starting at 1, incrementing by 2, and ending at 9, it would be written as:

>> X = 1:2:9

The notation **x = 1:2:9** in MATLAB defines a vector **X** that starts at 1, increments by 2, and ends at 9. The resulting vector will be:

X = [1, 3, 5, 7, 9]

This syntax is a compact way to generate vectors with a specified start value, step size (pitch), and end value.

For example:

```
>> X = [0:2:10] % vector X contains even numbers < 12
```

```
X =
    0    2    4    6    8   10

>> X= [-4:2:6]           % on can also write colon (-4,2,6)
X =
   -4   -2    0    2    4    6

>> X= 0:0.2:1           % you can also write colon (0,0.2,1)
X =
    0    0.2000    0.4000    0.6000    0.8000    1.0000
```

You can write more complex expressions like:

```
>> V = [1:2:5, -2:2:1]
V =
    1    3    5   -2    0

>> A = [1 2 3]
A =
    1    2    3

>> B = [A, 4, 5, 6]
B =
    1    2    3    4    5    6
```

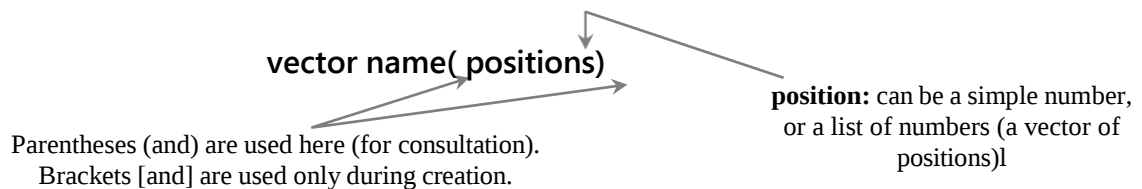
Example:

```
>> t = [1 2 0 0 ; 0 2 3 1 ; 0 0 2 1]
t =
    1    2    0    0
    0    2    3    1
    0    0    2    1

>> x = t(2, 3)
x =
    3
```

1.1 Referencing and access to vector elements:

The elements of a vector can be accessed using the following syntax:



Or “**vector(index)**”

Here, **vector** refers to the name of the vector, and **index** specifies the position of the element you want to retrieve. MATLAB uses 1-based indexing, meaning the first element is accessed with index=1. For example, for the vector:

```
X=[10,20,30,40]
```

To access the second element, you would use:

```
X(2)
```

which returns 20.

Examples:

```
>> V=[5, -1, 13,-6, 7]           % creation of vector V which contains 5 elements
V =
5     -1     13     -6      7
```

```
>> V(3)                          % the 3rd position
ans =
13
```

```
>> V(2:4)                        % from second to fourth position
ans =
-1     13     -6
```

```
>> V(4:-2:1)                     % from the 4th pos to the 1st with the pitch = -2
ans =
-6     -1
```

```
>> V(3:end)                      % from the 3rd position to the last
ans =
13     -6      7
```

```
>> V([1,3,4])                   % 1st, 3rd and 4th position only
ans =
5     13     -6
```

```
>> V(1)=8                       % set the first element to 8
V =
8     -1     13     -6      7
```

```
>> V(6)=-3                      % add a sixth element with the value -3
V =
8     -1     13     -6      7     -3
```

```
>> V(9)=5                       % add a ninth element with value 5
V =
8     -1     13     -6      7     -3      0      0      5
```

```
>> V(2)=[]                      % Remove second item
V =
8     13     -6      7     -3      0      0      5
```

```
>> V(3:5)=[]                    % Delete from 3rd to 5th element
V =
8     13      0      0      5
```

1.2 Element-by-element operations for vectors:

With two vectors $\vec{u}$ and $\vec{v}$, it is possible to perform element-by-element calculations using the following operations:

operation	Meaning	Exemple with :
		<pre>>> u= [-2,6,1] ; >> v= [3,-1, 4] ;</pre>
+	Addition of vectors	<pre>>> u+2 ans = 0 8 3 >>u+v ans = 1 5 5</pre>
-	Vector subtraction	<pre>>> u-2 ans = -4 4 -1 >> u-v ans = -5 7 -3</pre>
.*	Element-by-element multiplication	<pre>>> u*2 ans = -4 12 2 >> u.*2 ans = -4 12 2 >> u.*v ans = -6 -6 4</pre>
./	Division element by element	<pre>>> u/2 ans = -1.0000 3.0000 0.5000 >> u./2 ans = -1.0000 3.0000 0.5000 >> u./v ans = -0.6667 -6.0000 0.2500</pre>
.^	Power item by item	<pre>>> u.^2 ans = 4 36 1 >> u.^v ans = -8.0000 0.1667 1.0000</pre>

Writing an expression such as: u^2 generates an error car this expression refers to a matrix multiplication ($u*u$ must be rewritten $u*u'$ or $u'*u$ to be valid).

1.3 The linspace function:

The creation of a vector with components ordered by a regular interval and a specified number of elements can be achieved using the linspace function in MATLAB.

The syntax is as follows:

```
linspace(start, end, number of elements).
```

```
X = linspace(start_value, end_value, number_of_elements)
```

Here, **start_value** is the first element of the vector, **end_value** is the last element, and **number_of_elements** specifies how many elements should be evenly spaced between the two values.

Matlab calculates the increment step automatically according to the formula:

$$\text{step} = \frac{\text{end} - \text{start}}{\text{number of elements} - 1}$$

For example:

```
>> X=linspace(1,10,4) % a vector of four elements from 1 to 10
```

```
X =
    1    4    7   10
```

```
>> Y = linspace(13,40,4) % a vector of four elements of 13 to 40
```

```
Y =
   13   22   31   40
```

The size of a vector (i.e., the number of its components) can be obtained using the length function in MATLAB, as follows:

```
n = length(X)
```

Here, **X** is the vector for which you want to find the size, and **n** will hold the number of elements in the vector.

For example:

```
>> length(X) % the size of vector X
```

```
ans =
     4
```

2. The matrices:

A matrix is a rectangular array of (two-dimensional) elements. Vectors are matrices with a single row or column (monodimensional).

To insert a matrix, follow these rules:

- Items should be bracketed [and].
- Spaces or commas are used to separate items in the same row.
- Comma (or enter) is used to separate lines.

To illustrate this, considering the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

This matrix can be written in Matlab with the following parameters:

```
>> A=[1,2,3,4;5,6,7,8;9,10,11,12] ;
```

```
>> A=[1 2 3 4;5 6 7 8;9 10 11 12] ;
```

```
>> A = [1,2,3,4
5,6,7,8
9,10,11,12] ;
>> A=[[1;5;9] , [2;6;10] , [3;7;11] , [4;8;12]] ;
```

The number of elements in each row (number of columns) must be the same in all rows in the matrix, otherwise an error will be reported by Matlab. **For example:**

```
>> X=[1 2 ; 4 5 6]
Error using vertcat
CAT arguments dimensions are not consistent.
```

A matrix can be generated by vectors as shown in the following examples:

```
>> x = 1:4 % vector creation x
x =
     1     2     3     4
```

```
>> y = 5:5:20 % vector creation y
y =
     5    10    15    20
```

```
>> z = 4:4:16 % vector creation z
z =
     4     8    12    16
```

```
>> A = [x ; y ; z] % A is formed by line vectors x, y and z
A =
     1     2     3     4
     5    10    15    20
     4     8    12    16
```

```
>> B = [x' y' z'] % Best formed by column vectors x, y and z
B =
     1     5     4
     2    10     8
     3    15    12
     4    20    16
```

```
>> C = [x ; x] % It is formed by the same vector x2 times
C =
     1     2     3     4
     1     2     3     4
```

Example 2:

```
>> t1 = [1 2 ; 2 3]
t1 =
     1     2
     2     3
```

```
>> t1 = [3 4 ; 6 7]
t2 =
     3     4
     6     7
```

```
>> tL = [t1 , t2] % ou [t1 t2]
tL =
     1     2     3     4
     2     3     6     7
```

```
>> tc = [ t1 ; t2 ]
tc =
     1     2
     2     3
     3     4
     6     7
```

Exercise:

MATLAB is particularly well-suited for numerical applications involving matrices. Let's look at some methods for manipulating them.

1. Define a matrix $M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and then try the following operations in the interpreter:

```
>> 2 * M + 3
>> M + M
>> sqrt(M)
>> M * M
>> M .* M
>> ones(4)
>> ones(3, 5)
```

2. What is the difference between the $*$ and $.*$ operators? What does the `ones` function do?
3. How can we easily create a 54×42 matrix containing only 7s?

Réponse:

2. L'opérateur $*$ désigne le produit de deux matrices (comme vous l'avez vu en maths), tandis que $.*$ désigne le résultat du produit terme à terme de deux matrices de mêmes dimensions : la case $(i; j)$ du résultat est le produit des cases $(i; j)$ de chacune des deux matrices de départ. La fonction `ones` crée une matrice ayant les dimensions indiquées ne contenant que des 1.

3. `>> M = 7 * ones(54, 42);`

2.1 Referencing and access to matrix elements:

The elements of a matrix are accessed using the following general syntax:

nom_matrice(positions_lignes , positions_colonnes)

Parentheses (and) are used here (for consultation).
Brackets [and] are used only during creation.

positions: can be a simple number,
or a list of numbers (a vector of
positions)

Or: **matrix(row,column).**

It is worth noting the following possibilities:

- Access to an element in row i and column j is done by: **$A(i,j)$** .
- Access to the whole number i line is via: **$A(i,:)$** .
- Access to the entire column number j is by: **$A(:,j)$** .

Examples:

```
>> A = [1,2,3,4 ; 5,6,7,8 ; 9,10,11,12] % creation of matrix A
```

```
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

```
>> A(2,3) % element on the 2nd row in the 3rd column
```

```

ans =
     7

>> A(1,:) % all 1st line items
ans =
     1     2     3     4

>> A(:,2) % all items in 2nd column
ans =
     2
     6
    10

>> A(2:3,:) % all elements of the 2nd and 3rd line
ans =
     5     6     7     8
     9    10    11    12

>> A(1:2,3:4) % The upper right sub matrix of size 2x2
ans =
     3     4
     7     8

>> A([1,3],[2,4]) % La sub-matrix: rows(1,3) and columns (2,4)
ans =
     2     4
    10    12

>> A(:,3)=[] % Delete the third column
A =
     1     2     4
     5     6     8
     9    10    12

>> A(2,:)=[] % Delete the second line
A =
     1     2     4
     9    10    12

>> A=[A,[0;0]] % add a new column {or A(:,4)=[0;0]}
A =
     1     2     4     0
     9    10    12     0

>> A=[A;[1,1,1,1]] % Add a new line {or A(3,:)=[1,1,1,1]}
A =
     1     2     4     0
     9    10    12     0
     1     1     1     1
  
```

The dimensions of a matrix can be obtained using the **size** function. When applied to a matrix **A** of dimension $m \times n$, this function returns a vector with two components: the first component represents the number of rows **m**, and the second component represents the number of columns **n**.

The syntax is as follows:

```
dims = size(A)
```

For example:

```
>> d = size(A)
```

```
d =
     3     4
```

Here, the variable **d** contains the dimensions of the matrix **A** as a vector. To obtain the separate dimensions can use the syntax:

```
>> d1=size (A, 1)           % d1 contains the number of rows (m)
d1 =
     3
```

```
>> d2=size (A, 2)           % d2 contient le nombre de colonne (n)
d2 =
     4
```

2.2 Automatic generation of matrices:

In Matlab, there are functions that automatically generate particular matrices. In the following table we have the most used :

La fonction	Signification
zeros(n)	Generates an $n \times n$ matrix with all elements = 0
zeros(m,n)	Generates a matrix $m \times n$ with all elements = 0
ones(n)	Generates an $n \times n$ matrix with all elements = 1
ones(m,n)	Generates a $m \times n$ matrix with all elements = 1
eye(n)	Generates a $n \times n$ dimension identity matrix
magic(n)	Generates a $n \times n$ dimension magic matrix
rand(m,n)	Generates a matrix of dimension $m \times n$ de random values

For example:

```
>> ones(2)
ans =
     1     1
     1     1
```

```
>> ones(3,6)
ans =
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
```

```
>> zeros(3)
ans =
     0     0     0
     0     0     0
     0     0     0
```

```
>> zeros(2,7)
ans =
```

```
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

```
>> eye(4)
ans =
    1     0     0     0
    0     1     0     0
    0     0     1     0
    0     0     0     1
```

In MATLAB, the `magic(n)` function generates an **n x n magic square**, which is a square matrix where the sum of every row, column, and diagonal is the same. This number is known as the magic constant.

```
M = magic(n)
```

Input:

- `n`: The size of the magic square. It must be a positive integer.

Output:

- `M`: An `n × n` matrix where the sum of every row, column, and diagonal is equal.

Example:

```
>> n = 3;
>> M = magic(n);
>> disp(M);
ans =
     8     1     6
     3     5     7
     4     9     2
```

In this 3×3 magic square, the sum of each row, column, and diagonal is 15, which is the magic constant for a 3×3 square.

Magic Constant Formula:

The magic constant `C` for an `n×n` magic square is given by:

$$C = \frac{n(n^2 + 1)}{2}$$

For example, for a 3×3 square, $C = \frac{3(9+1)}{2} = 15$.

Properties of Magic Squares:

- The elements of a magic square are distinct integers from 1 to n^2 .
- The magic square is symmetric with respect to its center.

In MATLAB, the `rand(m, n)` function generates an $m \times n$ matrix of random numbers that are uniformly distributed between 0 and 1.

`A = rand(m, n)`

Input:

- `m`: The number of rows in the output matrix.
- `n`: The number of columns in the output matrix.

Output:

- `A`: An $m \times n$ matrix where each element is a random number between 0 and 1, drawn from a uniform distribution.

Example 1:

Generate a 3×2 matrix of random numbers:

```
>> A = rand(3, 2);
>> disp(A);
d1 =
    0.2761    0.5439
    0.8059    0.9021
    0.4923    0.6242
```

Example 2:

Generate a 4×4 matrix of random numbers:

```
>> B = rand(4, 4);
>> disp(B);
d1 =
    0.9397    0.1261    0.0142    0.6077
    0.6175    0.5161    0.7350    0.8002
    0.9601    0.3401    0.4602    0.0503
    0.3833    0.9069    0.5592    0.7212
```

2.3 Basic operations on the matrices:

L'opération	Signification
+	Addition
-	Subtraction
.*	The multiplication element by element (Element-wise multiplication)
./	The division element by element (Element-wise division)
.\	The division inverse element by element (Element-wise inverse division)
.^	The power element by element

*	Matrix multiplication (La multiplication matricielle)
/	Matrix division $(A/B) = (A*B^{-1})$ (La division matricielle)

The element-by-element operations on matrices are the same as those for vectors (the only condition necessary to make an element-by-element operation is that both matrices have the same dimensions). However, the multiplication or division of matrices requires some constraints (see a course on matrix algebra for more details).

Example:

```
>> A=ones (2 , 3)
```

```
A =
     1     1     1
     1     1     1
```

```
>> B=zeros (3,2)
```

```
B =
     0     0
     0     0
     0     0
```

```
>> B=B+3
```

```
B =
     3     3
     3     3
     3     3
```

```
>> A*B
```

```
ans =
     9     9
     9     9
```

```
>> B=[B , [3 3 3]'] % ou bien B(:,3)=[3 3 3]'
```

```
B =
     3     3     3
     3     3     3
     3     3     3
```

```
>> B=B(1:2, :) % ou bien B(3,:)=[]
```

```
B =
     3     3     3
     3     3     3
```

```
>> A=A*2
```

```
A =
     2     2     2
     2     2     2
```

```
>> A.*B
```

```
ans =
     6     6     6
     6     6     6
```

```
>> A*eye (3)
```

```
ans =
     2     2     2
     2     2     2
```

2.4 Useful functions for matrix processing:

Here are some of the most used functions regarding matrices:

fonction	usefulness	Example of use
Det	Determinant calculation of a matrix	<pre>>> A=[1,2;3,4] ; >>det(A) ans = -2</pre>
Inv	Calculates the inverse of a matrix	<pre>>> inv(A) ans = -2.0000 1.0000 1.5000 -0.5000</pre>
rank	Calculates the rank of a matrix	<pre>>> rank(A) ans = 2</pre>
trace	Calculates the trace of a matrix	<pre>>> trace(A) ans = 5</pre>
eig	Calculates the eigenvalues	<pre>>> eig(A) ans = -0.3723 5.3723</pre>
dot	Calculates the scalar product of 2 vectors	<pre>>> v=[-1,5,3]; >> u=[2,-2,1]; >> dot(u,v) ans = -9</pre>
norm	Calculates the standard of a vector	<pre>>> norm(u) ans = 3</pre>
cross	Calculates the vector product of 2 vectors	<pre>>> cross(u,v) ans = -11 -7 8</pre>
diag	Returns the diagonal of a matrix	<pre>>> diag(A) ans = 1 4</pre>
diag(V)	Creates a matrix with vector V in the diagonal and 0 elsewhere.	<pre>>> V=[-5,1,3] >> diag(V) ans = -5 0 0 0 1 0</pre>

		0 0 3
tril	Returns the lower triangular part	<pre> >> B=[1,2,3;4,5,6;7,8,9] B = 1 2 3 4 5 6 7 8 9 >> tril(B) ans = 1 0 0 4 5 0 7 8 9 >> tril(B,-1) ans = 0 0 0 4 0 0 7 8 0 >> tril(B,-2) ans = 0 0 0 0 0 0 7 0 0 </pre>
triu	Returns the upper triangular part	<pre> >> triu(B) ans = 1 2 3 0 5 6 0 0 9 >> triu(B,-1) ans = 1 2 3 4 5 6 0 8 9 >> triu(B,1) ans = 0 2 3 0 0 6 0 0 0 </pre>

Exercise 01:

Create the following vectors in two different ways.

$$V1 = (5 \quad 2 \quad 3 \quad 7 \quad 9), V2 = \begin{pmatrix} -1 \\ 3 \\ 6 \\ -5 \\ 8 \end{pmatrix}$$

Exercise 02:

Execute and understand the following commands:

```
V1*V2
V1*V2'
V1.*V2'
V3=[4 5 6 7 8]
V1+V3
V1/V3
V1./V3
V1^3
V1.^3
```

Exercise 03:

Let the three matrices A, B, and C be:

$$A = \begin{pmatrix} 1 & 2 \\ 7 & 2 \end{pmatrix}, B = \begin{pmatrix} 3 & -2 \\ 0 & 1 \end{pmatrix} \text{ and } C = \begin{pmatrix} -1 & 3 \\ 0 & 1 \\ -1 & -1 \\ 4 & 8 \end{pmatrix}$$

1) Calculate the following expressions:

- | | | |
|------------------------|-----------------------------|---|
| ▪ $A*B-3$ | ▪ $C*B+1+\text{zeros}(4,2)$ | ▪ $C(1:2,:) ^2$ |
| ▪ $A.*B-3$ | ▪ $A' .^B/2$ | ▪ $C(2:3,:) .^2$ |
| ▪ $A^2-\text{ones}(2)$ | ▪ $C*\text{eye}(2)$ | ▪ $C(\text{end}:-1:1,2) . \setminus 24$ |

2) Create the matrix M which contains matrices A and B stacked on top of each other to define the 1st and 2nd columns, and matrix C to define the 3rd and 4th columns.

$$M = \begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 7 & 2 \end{pmatrix} & \begin{pmatrix} -1 & 3 \\ 0 & 1 \\ -1 & -1 \\ 4 & 8 \end{pmatrix} \end{pmatrix}$$

3) Provide the Matlab result for each of the following commands:

- | | | |
|---------------------|-----------------------|--|
| ▪ $M(3,2) = 3$ | ▪ $M(2,:) - 7*M(1,:)$ | ▪ $M(2,:) = M(2,:) - 7*M(1,:)$ |
| ▪ $M(3,[2 \ 4])$ | | ▪ $M([1 \ 3],[1 \ 3]) = 10*\text{ones}(2)$ |
| ▪ $M(1:3,[2 \ 4])'$ | | ▪ $M([1,3],:) = []$ |

- `M(:,1) = []`
- `size(M)*M`
- `M(end:-1:1,end:-2:1)`
- `M = [[M;M] ones(4,1)]`
- `tril(M,-1)+triu(M,2)`

Exercise 04:

Execute and understand the following commands:

- | | |
|---|---|
| <ul style="list-style-type: none"> ▪ <code>U=ones(2,2)</code> ▪ <code>N=zeros(2,3)</code> ▪ <code>Y=eye(4,4)</code> ▪ <code>diag(A)</code> ▪ <code>diag(A,1)</code> ▪ <code>diag(A,-1)</code> ▪ <code>diag([2,4,6,8])</code> ▪ <code>help floor</code> ▪ <code>floor(B)</code> ▪ <code>help rand</code> ▪ <code>C=B.*B-4*A</code> ▪ Note: <code>B.*B</code> does not have the same meaning as <code>B*B</code> (test it). ▪ <code>D=B*B // or D=B^2</code> ▪ <code>M=floor(10*rand(3,3));</code> | <ul style="list-style-type: none"> ▪ <code>A=floor(10*rand(3,3))</code> ▪ <code>C=[A(:,2),B(:,3)]</code> ▪ <code>D=[B(1:2,:);A(2:3,:)]</code> ▪ <code>B=2*ones(4,4);</code> ▪ <code>M1=floor(10*rand(3,2))</code> ▪ <code>help find</code> ▪ <code>find(M1==3)</code> ▪ <code>find(M1>=3)</code> ▪ <code>find(M1>5)</code> ▪ <code>S=(M+M')/2;</code> ▪ <code>A=(M-M')/2;</code> ▪ <code>(M*M')'-M*M'</code> ▪ <code>A+S;</code> |
|---|---|

Exercise 05:

Consider the following three vectors:

$$X = (X_1, X_2, X_3, \dots, X_{n-1}, X_n), \quad Y = (Y_1, Y_2, Y_3, \dots, Y_{n-1}), \quad Z = (Z_1, Z_2, Z_3, \dots, Z_{n-1})$$

Propose Matlab instructions to design the matrices A, B, and C for $n=30$:

$$A = \begin{pmatrix} x_1 & z_1 & 0 & \cdots & 0 & 0 \\ y_1 & x_2 & z_2 & \ddots & 0 & 0 \\ 0 & y_2 & x_3 & \ddots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & x_{n-1} & z_{n-1} \\ 0 & 0 & 0 & \cdots & y_{n-1} & x_n \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 0 & \cdots & 0 & z_1 & x_1 \\ 0 & 0 & \ddots & z_2 & x_2 & y_1 \\ 0 & 0 & \ddots & x_3 & y_2 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ z_{n-1} & x_{n-1} & \ddots & 0 & 0 & 0 \\ x_n & y_{n-1} & \cdots & 0 & 0 & 0 \end{pmatrix}$$

$$C = \begin{pmatrix} 1 & 1/2^2 & 1/3^2 & \cdots & 1/n^2 \\ -1/2^2 & 1 & 1/2^2 & \ddots & 1/(n-1)^2 \\ -1/3^2 & -1/2^2 & 1 & \ddots & 1/(n-2)^2 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -1/(n-2)^2 & \vdots & \ddots & 1/2^2 & 1/3^2 \\ -1/(n-1)^2 & -1/(n-2)^2 & \ddots & 1 & 1/2^2 \\ -1/n^2 & -1/(n-1)^2 & \cdots & -1/2^2 & 1 \end{pmatrix}$$

Exercise 06:

For a matrix A, propose two general expressions for each of the following operations:

1. Delete the entire i-th row
2. Delete the entire j-th column
3. Add a row at the end of the matrix (a row vector x with **length(x) = size(A,2)**)
4. Add a column at the end of the matrix (a column vector y with **length(y) = size(A,1)**)

Exercise 07:

a) The matrices A, B, C by enumeration and D, E by description (i is written as %i in Matlab)

$$A = \begin{pmatrix} 1 & 4 \\ -2 & 5 \\ 3 & 0 \end{pmatrix}; B = \begin{pmatrix} 1/2 & 1 & 3/2 \\ 1/3 & 2/3 & 1 \\ 1/4 & 1/2 & 3/4 \end{pmatrix}; C = \begin{pmatrix} i & 0 & 1 \\ 0 & 1+i & 0 \\ 1 & 0 & i \end{pmatrix}; D = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}; E = \begin{pmatrix} 90 & 85 & 70 & 65 \\ 30 & 40 & 50 & 60 \\ 80 & 70 & 60 & 50 \\ 1 & 7 & 14 & 21 \end{pmatrix}$$

b) Provide the defined matrices as arguments to the functions **length** and **size**.

c) Conclude regarding the difference between these two functions.

d) Execute and understand the following commands:

```
>> C(2,2)=pi
```

```
>> C(:,3)=[1 :3]'
```

Solution TP TD N°02

Exercise 01:

1) Calculate the expressions:

```
>> A = [1 2 ; 7 2];
>> B = [3 -2 ; 0 1];
>> C = [-1 3 ; 0 1 ; -1 -1 ; 4 8];
```

```
>> A*B-3
```

ans =

```
0 -3
18 -15
```

A	B	A*B	A*B-3
1 2 7 2	3 -2 0 1	3 0 21 -12	0 -3 18 -15

```
>> A.*B-3
```

ans =

```
0 -7
-3 -1
```

A	B	A.*B	A.*B-3
1 2 7 2	3 -2 0 1	3 -4 0 2	0 -7 -3 -1

```
>> A^2-ones(2)
```

ans =

```
14 5
20 17
```

A	A^2	ones(2)	A^2-ones(2)
1 2 7 2	15 6 21 18	1 1 1 1	14 5 20 17

```
>> C*B+1+zeros(4,2)
```

ans =

```
-2 6
1 2
-2 2
13 1
```

C	B	C*B	zeros(4,2)	C*B+1+zeros(4,2)
-1 3 0 1 -1 -1 4 8	3 -2 0 1	-3 5 0 1 -3 -1 12 0	0 0 0 0 0 0 0 0	-2 6 1 2 -2 2 13 1

```
>> A' .^B/2
```

ans =

```
0.5000 0.0102
0.5000 1.0000
```

A	A'	B	A' .^B	A' .^B/2
1 2 7 2	1 7 2 2	3 -2 0 1	1 1/49 1 2	0.5 1/98 0.5 1

```
>> C*eye(2)
```

ans =

```
-1 3
0 1
-1 -1
4 8
```

C	eye(2)	C*eye(2)
-1 3 0 1 -1 -1 4 8	1 0 0 1	-1 3 0 1 -1 -1 4 8

```
>> C(1:2,:).^2
```

ans =

```
1 0
0 1
```

C	C(1:2,:)	C(1:2,:).^2
-1 3 0 1 -1 -1 4 8	-1 3 0 1	-1 3 -1 3 1 0 0 1 0 1 0 1

```
>> C(2:3,:).^2
```

```
ans =
```

```
0 1
1 1
```

C	C(2:3,:)	C(1:2,:).^2
$\begin{bmatrix} -1 & 3 \\ 0 & 1 \\ -1 & -1 \\ 4 & 8 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix}$	$\begin{bmatrix} (0)^2 & (1)^2 \\ (-1)^2 & (-1)^2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$

```
>> C(end:-1:1,2).\24
```

```
ans =
```

```
3
-24
24
8
```

C	C(end:-1:1,2)	C(end:-1:1,2).\24
$\begin{bmatrix} -1 & 3 \\ 0 & 1 \\ -1 & -1 \\ 4 & 8 \end{bmatrix}$	$\begin{bmatrix} 8 \\ -1 \\ 1 \\ 3 \end{bmatrix}$	$\begin{bmatrix} 24/8 & 3 \\ 24/-1 & -24 \\ 24/1 & 24 \\ 24/3 & 8 \end{bmatrix}$

2) Creation of the matrix **M**:

```
>> M = [ [A ; B] C ]
```

```
M =
```

```
1 2 -1 3
7 2 0 1
3 -2 -1 -1
0 1 4 8
```

1	2	-1	3
7	2	0	1
3	-2	-1	-1
0	1	4	8

3) The Matlab results for the expressions:

```
>> M(3,2) = 3
```

```
M =
```

```
1 2 -1 3
7 2 0 1
3 3 -1 -1
0 1 4 8
```

1	2	-1	3
7	2	0	1
3	3	-1	-1
0	1	4	8

```
>> M(3,[2 4])
```

```
ans =
```

```
3 -1
```

1	2	-1	3
7	2	0	1
3	3	-1	-1
0	1	4	8

```
>> M(1:3,[2 4])'
```

```
ans =
```

```
2 2 3
3 1 -1
```

M(1:3,[2 4])	M(1:3,[2 4])'
$\begin{bmatrix} 2 & 3 \\ 2 & 1 \\ 3 & -1 \end{bmatrix}$	$\begin{bmatrix} 2 & 2 & 3 \\ 3 & 1 & -1 \end{bmatrix}$

```
>> M(2,:)-7*M(1,:)
```

```
ans =
```

```
0 -12 7 -20
```

M(2,:)	M(1,:)	7* M(1,:)
$\begin{bmatrix} 7 & 2 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & -1 & 3 \end{bmatrix}$	$\begin{bmatrix} 7 & 14 & -7 & 21 \end{bmatrix}$

```
>> M(2,:) = M(2,:)-7*M(1,:)
```

```
M =
```

```
1 2 -1 3
0 -12 7 -20
3 3 -1 -1
0 1 4 8
```

1	2	-1	3
0	-12	7	-20
3	3	-1	-1
0	1	4	8

```
>> M([1 3],[1 3]) = 10*ones(2)
```

M =

```
10    2    10    3
 0   -12    7   -20
10    3    10   -1
 0    1     4     8
```

10	2	10	3
0	-12	7	-20
10	3	10	-1
0	1	4	8

```
>> M([1,3],:) = []
```

M =

```
 0   -12    7   -20
 0    1     4     8
```

0	-12	7	-20
0	1	4	8

```
>> M(:,1) = []
```

M =

```
-12    7   -20
 1     4     8
```

-12	7	-20
1	4	8

```
>> size(M)*M
```

ans =

```
-21    26   -16
```

M	size(M)	size(M)*M
$\begin{bmatrix} -12 & 7 & -20 \\ 1 & 4 & 8 \end{bmatrix}$	(2 3)	$\begin{bmatrix} -21 & 26 & -16 \end{bmatrix}$

```
>> M(end:-1:1,end:-2:1)
```

ans =

```
 8     1
-20   -12
```

```
>> M = [[M;M] ones(4,1)]
```

M =

```
-12    7   -20    1
 1     4     8     1
-12    7   -20    1
 1     4     8     1
```

-12	7	-20	1
1	4	8	1
-12	7	-20	1
1	4	8	1

```
>> tril(M,-1)+triu(M,2)
```

ans =

```
 0     0   -20    1
 1     0     0    1
-12    7     0     0
 1     4     8     0
```

tril(M,-1)	triu(M,2)
$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -12 & 7 & 0 & 0 \\ 1 & 4 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & -20 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

Exercise 02:

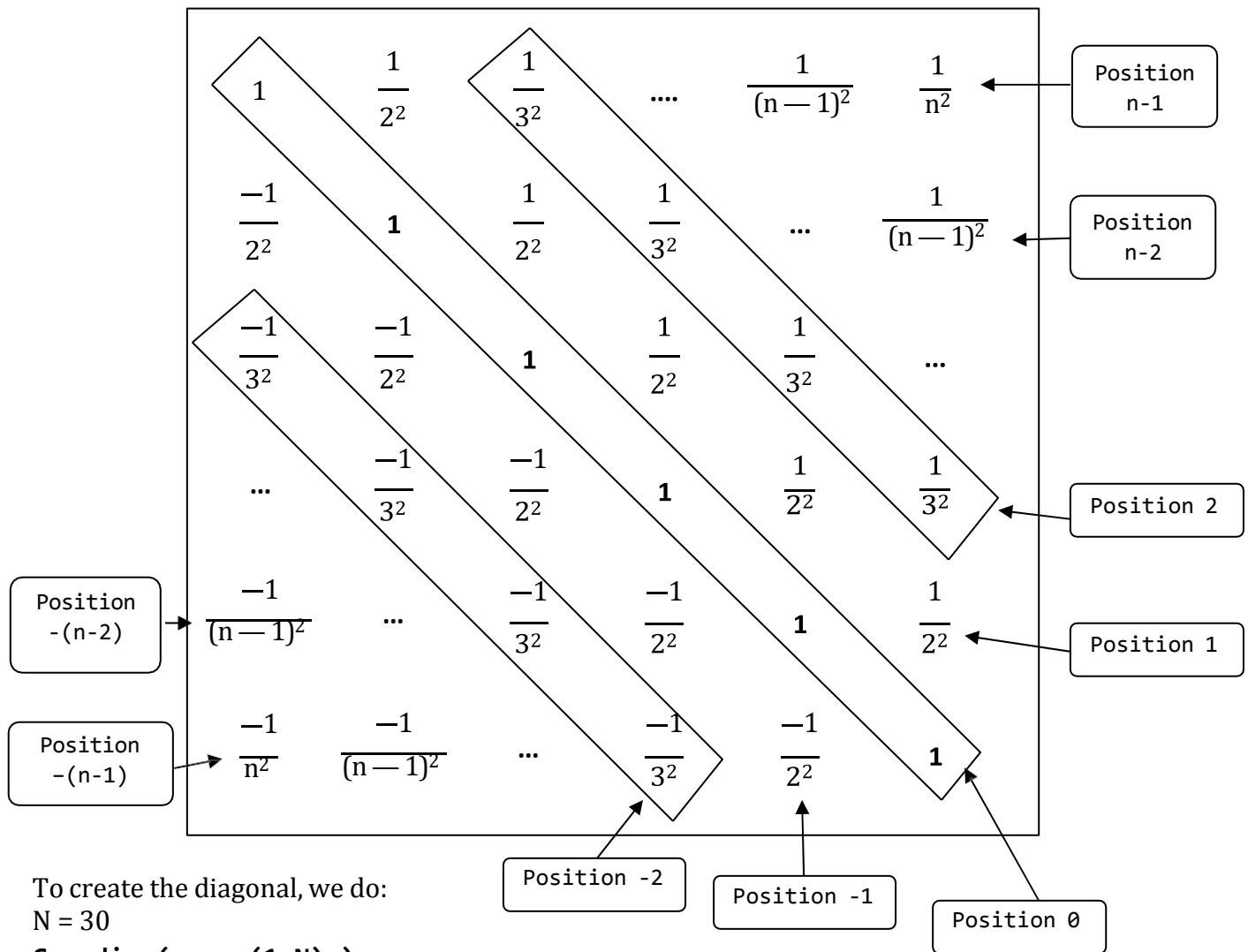
```
>> n=10 ;
>> mat1 = diag(x);
>> mat2 = diag(y,-1);
>> mat3 = diag(z,1);
>> A = mat1 + mat2 + mat3 ;
>> B = A(:, end:-1:1) ;
```

```
x1  0  0  ...  0  0
 0  x2  0  \  0  0
 0  0  x3  \  0  0
 \  \  \  \  \  \
 0  0  0  \  x_{n-1}  0
 0  0  0  ...  0  x_n
```

```
0  0  0  ...  0  0
y  0  0  \  0  0
 0  y2  0  \  0  0
 \  \  \  \  \  \
 0  0  0  \  0  0
 0  0  0  ...  y_{n-1}  0
```

```
0  z1  0  ...  0  0
0  0  z2  \  0  0
0  0  0  \  0  0
 \  \  \  \  \  \
0  0  0  \  0  z_{n-1}
```

Creation of the matrix C (this part of the exercise is optional):



To create the diagonal, we do:

$N = 30$

C = diag(ones(1,N));

% We can observe that $C = \text{diag}((\text{ones}(1,N))/1^2) , 0$;

To create the other elements, we do:

Above the diagonal

To create the vector: $1/2^2$ (position 1) above the diagonal, we do:

V1 = diag((ones(1,N-1))/2^2) , 1);

To create the vector: $1/3^2$ (position 2) above the diagonal, we do:

V2 = diag((ones(1,N-2))/3^2) , 2);

To create the vector: $1/4^2$ (position 3) above the diagonal, we do:

V3 = diag((ones(1,N-3))/4^2) , 3);

...

To create the vector: $1/n^2$ (position n-1) above the diagonal, we do:

Vn = diag((ones(1,1))/N^2) , N-1);

Below the diagonal

To create the vector: $-1/2^2$ (position -1) below the diagonal, we do:

U1 = diag(-(ones(1,N-1))/2^2) , -1);

To create the vector: $-1/3^2$ (position -2) below the diagonal, we do:

U2 = diag(-(ones(1,N-2))/3^2) , -2);

To create the vector: $-1/4^2$ (position -3) below the diagonal, we do:

U3 = diag(-(ones(1,N-3))/4^2) , -3);

...

To create the vector: $-1/n^2$ (position - (n-1)) below the diagonal, we do:

Un = diag(-(ones(1,1))/N^2) , -(N-1));

So, to create the vector V_i from the i -th position above the diagonal, we write:

```
Vi = diag( (ones(1,N-i)) / (i+1)^2 , i);
```

And to create the vector U_i from the i -th position below the diagonal, we write:

```
Ui = diag( -(ones(1,N-i)) / (i+1)^2 , -i));
```

The final matrix is the sum of all these matrices, so we can use a **for** loop as follows:

```
for i = 1:N-1
    V = (ones(1,N-i))/(i+1)^2;
    C = C + diag(V,i) + diag(-V,-i);
end
```

So the complete commands to generate this matrix are:

```
format rat
N = 30;
C = diag(ones(1,N));

for i=1:N-1
    V = (ones(1,N-i))/(i+1)^2;
    C = C + diag(V,i) + diag(-V,-i);
end

C
```

Second method:

```
format rat
N = 30;
C = ones(N,N);

for i = 1:N
    C(i,i+1:N)= 1./[2:N-i+1].^2;
    C(i+1:N,i)= -(1./[2:N-i+1].^2)';
end

C
```

Exercice 03:

1. Delete the entire i -th row

The 1st method:

$$A(i,:) = []$$

The 2nd method:

$$A = [A(1:i-1,:) ; A(i+1:end,:)]$$

The 3rd method:

$$A = A([1:i-1, i+1:size(A,1)], :)$$

2. Delete the entire j -th column

The 1st method:

$$A(:,j) = []$$

The 2nd method:

$$A = [A(:,1:j-1) , A(:,j+1:end)]$$

The 3rd method:

$$A = A(:, [1:j-1, j+1:size(A,2)])$$

3. Add a row at the end of the matrix (a row vector x)

The 1st method:

$$A = [A ; x]$$

The 2nd method:

$$A(end+1,:) = x$$

The 3rd method:

$$A(size(A,1)+1, :) = x$$

4. Add a column at the end of the matrix (a column vector y)

The 1st method:

$$A = [A y]$$

The 2nd method:

$$A(:,end+1) = y$$

The 3rd method:

$$A(:,size(A,2)+1) = y$$

MATLAB

Chapter III: Introduction to programming with Matlab

We have seen so far how to use Matlab to perform commands or to evaluate expressions by writing them in the command line (After prompt >>), so the commands used are usually written as a single statement (possibly on a single line).

However, there are problems whose description of their solutions requires several instructions, which require the use of several lines. For example, searching for the roots of a second-degree equation (taking into account all possible cases).

A collection of well-structured instructions for solving a given problem is called a program. In this part of the course, we will present the mechanisms of writing and executing programs in Matlab.

1. General:

1.1 Comments:

Comments are explanatory sentences ignored by Matlab and intended for the user to help him understand the part of the commented code.

In Matlab a comment starts with the % symbol and occupies the rest of the line.

For example:

```
>> A=B+C ;           % Give A the value of B+C
```

1.2 Writing long expressions:

If a long expression cannot be written in a single line, it can be divided into several lines by putting at the end of each line at least three points.

Example:

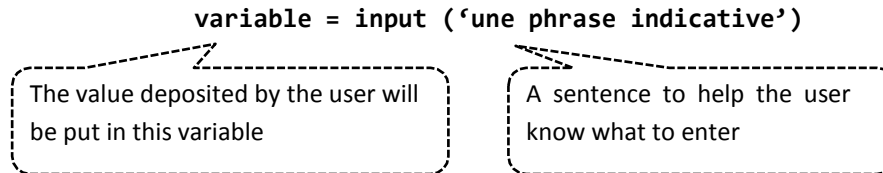
```
>> (sin(pi/3)^2/cos(pi/3)^2)-(1-2*(5+sqrt(x)^5/(-2*x^3-x^2)^1+3*x)) ;
```

This expression can be rewritten as follows:

```
>> (sin(pi/3)^2/cos(pi/3)^2)- ... ↵
>> (1-2*(5+sqrt(x)^5 ..... ↵
>> /(-2*x^3-x^2)^1+3*x)) ;    ↵
```

1.3 Reading data in a program (Inputs):

To read a value given by the user, it is possible to use the **input** command, which has the following syntax:



When Matlab executes such an instruction, the indicative phrase will be displayed to the user waiting for the latter to enter a value.

for example:

```
>> A = input ('Enter a whole number : ')
Enter a whole number : 5
A =
    5
>>
```

```
>> A = input (' Enter a whole number : ');
Enter a whole number : 5
>>
```

```
>> B = input ('Enter a vector line : ')
Enter a vector line : [1:2:8,3:-1:0]
B =
     1     3     5     7     3     2     1     0
```

1.4 Writing data in a program (Outputs):

We have already seen that Matlab can display the value of a variable by typing only the name of this last. For example :

```
>> A = 5 ;
>> A
A =
    5
% Ask Matlab to display the value of A
```

With this method, Matlab writes the name of the variable (A) then the sign (=) followed by the desired value. However, there are cases where only the value of the variable is displayed (without the name and without the sign =).

To do this, we can use the **disp** function, which has the following syntax: **disp(object)**
The value of the object can be a number, a vector, a matrix, a string or an expression.

It is reported that with an empty vector or matrix, **disp** displays nothing.

Example:

```
>> disp(A)           % Display the value of A without 'A = '
5
>> disp(A);          % Semicolon has no effect
5
>> B                  % Display vector B by the classical method
B =
    1     3     5     7     3     2     1     0
>> disp(B)           % Display the vector B without 'B = '
    1     3     5     7     3     2     1     0
>> C = 3 :1 :0        % Creating an empty C vector
C =
Empty matrix: 1-by-0
>> disp(C)           % disp displays nothing if vector is empty
```

2. Logical expressions:

2.1 Logical operations:

Logical operations in MATLAB are used to compare values and return logical values (true/false or 1/0). These operations are essential for conditional statements, loops, and decision-making in your programs. Below are the common logical operations:

1. Relational Operators

These compare two values and return true (1) or false (0).

- **Equal to (==):**

```
result = (x == y); % Returns true if x is equal to y
```

- **Not equal to (~=):**

```
result = (x ~= y); % Returns true if x is not equal to y
```

- **Less than (<):**

```
result = (x < y); % Returns true if x is less than y
```

- **Greater than (>):**

```
result = (x > y); % Returns true if x is greater than y
```

- **Less than or equal to (<=):**

```
result = (x <= y); % Returns true if x is less than or equal to y
```

- **Greater than or equal to ($\geq$):**

```
result = (x >= y); % Returns true if x is greater than or equal to y.
```

2. Logical Operators

These operators allow you to perform logical operations on arrays or scalar values.

- **AND (&):**

```
result = (x > 0) & (y > 0); % Returns true if both conditions are true
```

- **OR (|):**

```
result = (x > 0) | (y > 0); % Returns true if either condition is true
```

- **NOT (~):**

```
result = ~(x > 0); % Returns true if x is not greater than 0.
```

3. Logical Arrays

In MATLAB, logical operations can be performed on arrays. The result is an array of logical values.

- **Example:**

```
A = [1, 2, 3, 4];
B = [4, 3, 2, 1];
result = (A > B); % Compares element-wise, returns [false, false, true, true]
```

4. Short-circuit Operators

These operators are used to perform logical operations but stop evaluating as soon as the result is determined.

- **Short-circuit AND (&&):**

```
result = (x > 0) && (y > 0); % Stops if the first condition is false
```

- **Short-circuit OR (||):**

```
result = (x > 0) || (y > 0); % Stops if the first condition is true
```

5. Logical Functions

MATLAB also provides built-in logical functions for more complex operations:

- **all ():** Returns true if all elements are true.

```
result = all(A > 0); % Returns true if all elements in A are greater
than 0
```

- **any()**: Returns true if any element is true.

```
result = any(A > 0); % Returns true if any element in A is greater
than 0
```

- **find()**: Returns the indices of non-zero elements (true values).

```
indices = find(A > 2); % Finds the indices where A is greater than 2
```

Summary

Logical operations in MATLAB are essential for comparing values, making decisions, and controlling program flow. These operations are widely used in conditional statements, loops, and filtering data.

The basic logical operations in an expression are summarized in the following table:

The comparison operation	its meaning
==	equality
~=	inequality
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
logical operations	its meaning
&	the logical and
 	the logical OR
~	the logical negation

In Matlab a logical variable can take the values 1(true) or 0(false) with a small rule that assumes that:

- 1) Any value equal to 0 will be considered false (**= 0 $\Rightarrow$ false**)
- 2) Any value other than 0 will be considered true (**$\neq$ 0 $\Rightarrow$ true**).

The following table summarizes the operation of logical operations:

a	b	a & b	a b	~a
1 (true)	1(true)	1	1	0
1 (true)	0 (false)	0	1	0
0(false)	1 (true)	0	1	1
0 (false)	0 (false)	0	0	1

For example:

```
>> x=10;
>> y=20;
>> x < y          % displays 1 (true)
    ans =
         1
>> x <= 10         % displays 1 (true)
    ans =
         1
>> x == y          % displays 0 (false)
    ans =
         0
>> (0 < x) & (y < 30) % displays 1 (true)
    ans =
         1
>> (x > 10) | (y > 100) % displays 0 (false)
    ans =
         0
>> ~(x > 10)       % displays 1 (true)
    ans =
         1
>> 10 & 1           % 10 is considered true therefore 1 & 1 = 1
    ans =
         1
>> 10 & 0           % 1 & 0 = 0
    ans =
         0
```

Example:

```
>> (3 == 5) & (3 == (2 + 1))
    ans =
         0
>> (3 == 5) | (3 == (2 + 1))
    ans =
         1
```

2.2 Matrix comparison:

In MATLAB, matrix comparison involves element-wise operations where matrices are compared to each other or to scalar values. The result of these comparisons is a matrix of logical values (`true` or `false`), where each element corresponds to the result of comparing the elements of the matrices at the same position.

The comparison of vectors and matrices differs somewhat from scalars, hence the usefulness of the two functions 'isequal' and 'isempty' (which allow to give a concise answer for comparison).

Function	Description
Isequal	tests whether two (or more) matrices are equal (having the same elements everywhere). Returns 1 if so, and 0 otherwise.
Isempy	tests if a matrix is empty (contains no elements). Returns 1 if it is, and 0 otherwise.

To better perceive the impact of these functions follow the following example:

```
>> A=[5,2;-1,3]           % create the matrix A
    A =
         5         2
        -1         3

>> B=[5,1;0,3]           % create the matrix B
    B =
         5         1
         0         3

>> A==B                   % test whether A=B ? (1 or 0 depending on the position)
    ans =
         1         0
         0         1

>> isequal(A,B)           % Test if A and B are equal (the same)
    ans =
         0

>> C=[] ;                 % Create the empty matrix C

>> isempty(C)             % Test if C is empty (true = 1)
    ans =
         1

>> isempty(A)             % Test if A is empty (displays false = 0)
    ans =
         0
```

3. Flow control structures

Flow control structures are instructions for defining and manipulating the order of execution of tasks in a program. They offer the possibility to perform different treatments depending on the state of the program data, or to perform repetitive loops for a given process.

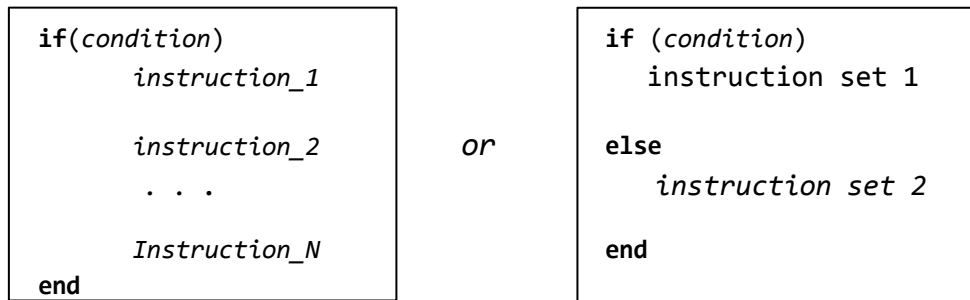
Matlab has eight flow control structures, namely:

- **if**
- **switch**
- **for**
- **while**
- **continue**
- **break**
- **try - catch**
- **return**

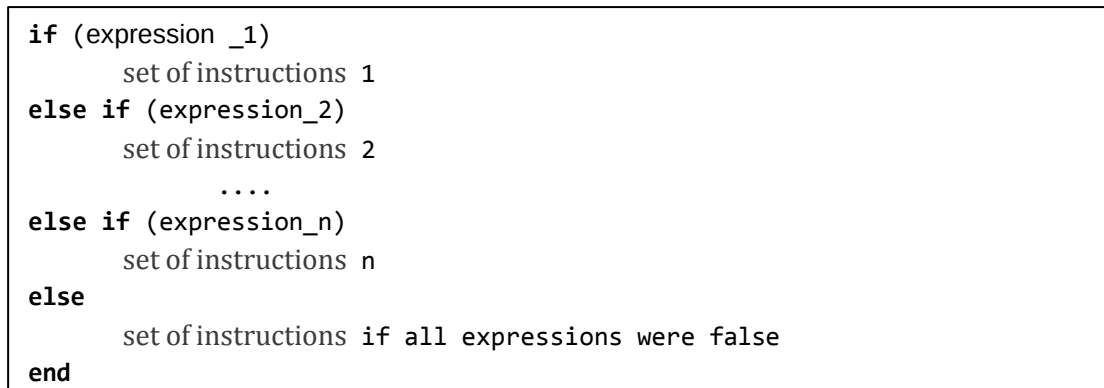
We expose the first four : (if, switch, for and while)

3.1 The if statement:

The **if** statement is the simplest and most widely used flow control structure. It guides the execution of the program according to the logical value of a condition. Its general syntax is as follows:



If the condition is evaluated to true, the instructions between the **if** and the **end** will be executed; otherwise, they will not be (or if an **else** exists, the instructions between the **else** and the **end** will be executed). If it is necessary to check multiple conditions instead of just one, you can use **else if** clauses for each new condition, and at the end, you can put an **else in** case no condition has been evaluated as true. Here is the general syntax:



For example, the following program defines you according to your age:

```

>> age = input('Enter your age : '); ...
if (age <2)
    disp('You are a fool')
elseif (age <13)
    disp('You are a child')
elseif (age < 18)
    disp ('You are an adolescent')
elseif (age <60)
    disp ('You are unadulterated')

```

```
else
    disp ('You are an old man')
end
```

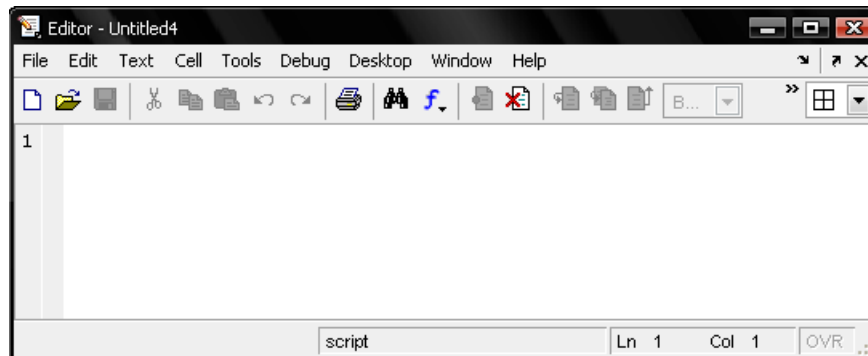
As you can see, writing a Matlab program directly after the command prompt (the prompt >>) is a bit unpleasant and annoying.

A more convenient method is to write the program to a separate file, and call that program (if necessary) by typing the file name in the command prompt.

This approach is defined in Matlab by M-Files, which are files that can contain data, programs (scripts) or functions that we develop.

To create an M-Files simply type the command `edit`, or simply go to the menu: File New M-Files (or click on the icon ).

In any case an editing window like this will appear:



All you have to do is write your program in this window and save it with a name (for example: 'Premier_Programme.m'). It is reported that the extension of the M-Files files is always '.m'. Now, if we want to run our program, just go to the usual command prompt (>>) and then type the name of our file (without the '.m') like this:

```
>> Premier_Programme
```

And the program will start running immediately.

To return to the editing window (after closing it) simply enter the command :

```
>> edit Premier_Programme
```

Example:

Let's create a program that finds the roots of a second-degree equation designated by: $ax^2+bx+c=0$. Here is the M-File that contains the program (it is saved with the name 'Equation2deg.m').

```
% Programme de résolution de l'équation  $a*x^2+b*x+c=0$ 

a = input ('Entrez la valeur de a : ');           % Lire a
b = input ('Entrez la valeur de b : ');           % Lire b
c = input ('Entrez la valeur de c : ');           % Lire c

delta = b^2-4*a*c ;                               % Calculer delta
if delta<0
    disp('Pas de solution')                        % Pas de solution
elseif delta==0
    disp('Solution double : ')                    % Solution double
    x=-b/(2*a)
else
    disp('Deux solutions distinctes: ')           % Deux solutions
    x1=(-b+sqrt(delta))/(2*a)
    x2=(-b-sqrt(delta))/(2*a)
end
```

If we want to run the program, just type the name of the program:

```
>> Equation 2 of g
Enter the value of a : -2 ↵
Enter the value of b : 1 ↵
Enter the value of c : 3 ↵
Two solutions:
x1 =
    -1
x2 =
    1.5000
```

Thus, the program will be executed following the instructions written in its M-File. If an instruction is terminated by a semicolon, then the value of the variable concerned will not be displayed, but if it ends with a comma or a line break, then the results will be displayed.

Note: There is the predefined **solve** function in Matlab to find the roots of an equation (and much more). If we want to apply it to our example, just write:

```
>> solve('-2*x^2+x+3=0', 'x')
years =
    -1
    3/2
```

3.2 The switch statement:

The **switch** statement executes groups of statements based on the value of a variable or expression. Each group is associated with a **case** clause that defines whether or not this group should be executed according to the equality of the value of this box with the evaluation result of the **switch** expression. If not all **cases** have been accepted, it is possible to add an **otherwise** clause that will be executed only if no box is executed.

Therefore, the general form of this instruction is:

```
switch (expression)
  case value_1
    Instruction group 1
  case value_2
    Instruction group 2
    . . .
  case value_n
    Instruction group n
  otherwise
    Package instructions where the boxes have failed
end
```

Example:

```
x = input ('Enter a number: ');
switch(x)
  case 0
    par ('x = 0 ')
  case 10
    par('x = 10 ')
  case 100
    par('x = 100 ')
  otherwise
    par('x n'' is not s 0 or 10 or 100')
end
```

The execution will give:

```
Enter a number : 50
x is not 0 or 10 or 100
```

3.3 The for statement:

The **for** statement repeats the execution of a group of instructions a specified number of times. It has the following general form:

```
for variable = expression_vector
  instruction group
end
```

The expression_vector corresponds to the definition of a vector: *start: not: end or start: end*

The variable will go through all the elements of the vector defined by the expression, and for each it will execute the group of instructions.

Example:

In the following table, we know three forms of the **for** statement with the Matlab result:

The instruction for		for i = 1 : 4 j=i*2 ; disp(j) end	for i = 1 : 2 : 4 j=i*2 ; disp(j) end	for i = [1,4,7] j=i*2 ; disp(j) end
The resultat of the execution		2 4 6 8	2 6	2 8 14

3.4 The while statement:

The **while** statement repeats the execution of a group of statements an indeterminate number of times depending on the value of a logical condition. It has the following general form:

```
while (condition)
    set of instructions
end
```

As long as the expression of **while** is evaluated to true, the instructions set will run in a loop.

Example:

```
1. a=1 ;
   while (a~=0)
       a = input ('Enter unnombre (0 to finish) : ') ;
   end
```

This program asks the user to enter a number. If this number is not equal to 0 then the loop repeats, otherwise (if the given value is 0) then the program stops.

```
2. count = 1;
   while count <= 5
       disp(['Count is: ', num2str(count)]);
       count = count + 1;
   end
```

This program repeatedly displays the current value of the variable count while incrementing it by 1 in each iteration, until count reaches 5. Specifically, it prints "Count is: " followed by the current value of count.

However, in its current form, the program does not interact with the user directly (e.g., it doesn't ask for any input). If the intent is for the program to ask the user something, you would need to modify it to include a prompt for user input, such as using `input()`.

For example, if you want the program to ask the user for confirmation before incrementing the counter:

```
count = 1;
while count <= 5
    disp(['Count is: ', num2str(count)]);
    input('Press Enter to continue...');
    count = count + 1;
end
```

In this modified version, the user is asked to press Enter to continue each time before the counter increments.

Remark:

In MATLAB, you can easily replace a `for` loop with a `while` loop by handling the initialization, condition, and increment manually. Here's an example to illustrate how this can be done.

Example of a `for` loop in MATLAB:

```
a = 16; % Example value
x = a;
for i = 1:10 % Perform 10 iterations
    x = 0.5 * (x + a / x);
end
disp(x)
```

This is an implementation of Newton's method for finding the square root with 10 iterations using a `for` loop.

Replacing the `for` loop with a `while` loop in MATLAB:

```
a = 16; % Example value
x = a;
i = 1; % Initialization of iteration counter
while i <= 10 % Perform 10 iterations
    x = 0.5 * (x + a / x);
    i = i + 1; % Increment the counter manually
end
disp(x)
```

Explanation:

- **Initialization:** `i = 1` initializes the counter.
- **Condition:** `while i <= 10` runs the loop as long as the counter `i` is less than or equal to 10.
- **Increment:** `i = i + 1` manually increments the counter at the end of each iteration.

This `while` loop will perform exactly the same task as the `for` loop, preserving the functionality of the program.

4. Summary the control structures:

MATLAB offers several flow control structures to manage the execution of code based on conditions or for repetitive tasks. These include conditional statements (if, else, elseif), loops (for, while), as well as error handling using try and catch. These structures are essential for creating flexible, robust, and efficient programs.

We gives now a glimpse of:

1. **break** and **continue** Statements

- **break:** Exits the loop completely, even if the loop condition is still true.
- **continue:** Skips the current iteration and moves to the next iteration of the loop.
- **Example (using break):**

```
for i = 1:10
    if i == 5
        break; % Exit the loop when i equals 5
    end
    disp(i);
end
```

- **Example (using continue):**

```
for i = 1:10
    if mod(i, 2) == 0
        continue; % Skip the rest of the loop for even numbers
    end
    disp(i); % Display only odd numbers end
```

2. **try** and **catch** Statements

The try and catch block is used for error handling. If an error occurs in the try block, control is passed to the catch block, allowing you to handle the error gracefully.

- **Syntax:**

```
try
    % Code that may cause an error
```

```
catch exception
    % Code to handle the error
end
```

• **Example:**

```
try
    A = [1, 2, 3];
    disp(A(4)); % This will cause an error because there is no 4th
element
catch exception
    disp('An error occurred:');
    disp(exception.message);
end
```

5. Summary exercise:

There are predefined functions in Matlab given in the table below. Let's try to program them (for a given vector V).

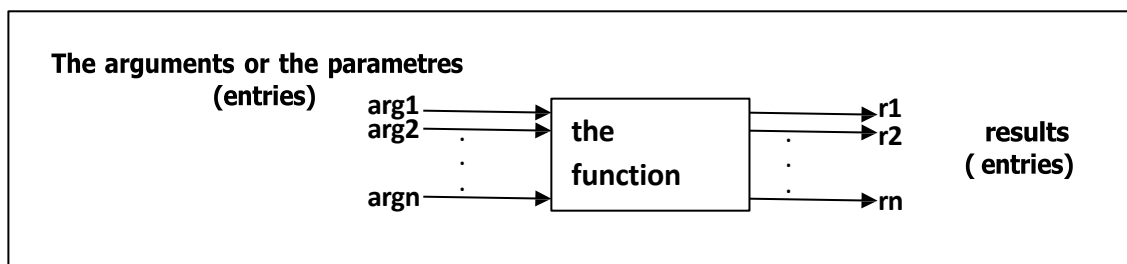
function	Description	The program that simulates it
sum (V)	The sum of the elements of a vector V	<pre>n = length(V); sum = 0 ; for i = 1 : n sum=sum+V(i) ; end disp(sum)</pre>
prod (V)	The product of elements of a vector V	<pre>n = length(V); product = 1 ; for i = 1 : n product=product*V(i) ; end disp(product)</pre>
mean (V)	The average of the elements of a vector V	<pre>n = length(V); moyenne = 0 ; for i = 1 : n moyenne = moyenne+V(i) ; end moyenne = moyenne / n</pre>
diag (V)	Create a matrix with vector V in the diagonal, and 0	<pre>n = length(V); A = zeros(n) ; for i = 1 : n A(i,i)=V(i) ; end disp(A)</pre>

sort(V)	Order elements of vector V in ascending order	<pre> n = length(V); for i = 1 : n-1 for j = i+1 : n if V(i) > V(j) tmp = V(i) ; V(i) = V(j) ; V(j) = tmp ; end end end disp(V) </pre>
----------------	---	---

6. The functions:

There is a difference in concept between functions in computer science or mathematics:

1. In computer science, a function is a routine (a sub-program) that accepts arguments (parameters) and returns a result.



2. In mathematics a function **f** is a relationship that assigns to each value **x** no more than one value **f(x)**.

6.1 Creating a function in an M-Files:

Matlab contains a large number of predefined functions such as **sin**, **cos**, **sqrt**, **sum**, ... etc. and it is possible to create our own functions by writing their source codes in M-Files (with the same function name) respecting the following syntax:

```

function [r1, r2, ..., rn] = nom_fonction (arg1, arg2, ..., argn)

    % The body of the function
    . . .
    r1 = . . . % the value returned for r1
    r2 = . . . % The value returned for r2
    . . .
    rn = . . . % the value returned for rn
end                                     % The end is optional

```

Or: $r_1...r_n$ are the values returned, and $\mathbf{arg}_1...arg_n$ are the arguments.

Example: Write a function that calculates the square root of a number by the Newton method (view in the TP).

Solution:

```
>> edit
```

The root file. m

```
function r =racine(nombre)
r = nombre/2;
precision = 6;
for i = 1:precision
    r = (r + nombre ./ r) / 2;
end
```

Execution:

```
>> x = root (9)
```

```
x =
```

```
3
```

```
>> x = root (196)
```

```
x =
```

```
14.0000
```

```
>> x = root ([16,144,9,5])
```

```
x =
```

```
4.0000    12.0000    3.0000    2.2361
```

Remark:

Unlike a program (a script), a function can be used in an expression for example:

2* root (9)-1.

6.2 Comparison between a program is a function:

<i>program</i>	<i>fonction</i>
<pre>a = input('Enter a positive number: '); x = a/2; Precision = 6; for i = 1:precision x = (x + a ./ x) / 2; end disp(x)</pre>	<pre>function r =root(number) r = number /2; Precision = 6; for i = 1: Precision r = (r + number ./ r) / 2; end</pre>
execution: <pre>>> root ↵ Enter a positive number: 16↵ 4</pre>	execution: <pre>>> root (16) ans = 4</pre>
one cannot write expressions such as: <pre>>> 2* root + 4</pre>	you can write phrases like: <pre>>> 2* root (x) + 4</pre>

Exercise: As long as there are sums

We have already seen how to use a for loop to repeat several instructions. However, it was necessary to know in advance how many times we wanted to repeat the loop. The keyword **while** is used to execute a sequence of instructions until a condition is met.

1. Run the following script.

```
x = 1;
while x < 1000
x
x = 2*x;
end
```

What does it do? Did you understand the meaning of the keyword *while*?

Answer:

This program calculates the smallest power of two greater than one thousand.

2. Write a function `cube(n)` that returns the largest cube less than or equal to n .

Answer:

```
function r=cube(n)
i=0;
while i^3<=n
i=i+1;
end
r=(i-1)^3;
```

3. Write a function `somme_carres(n)` that returns the largest integer k satisfying

$$\sum_{i=1}^k i^2 \leq n$$

Answer:

```
function r=somme_carres(n)
i=0; S=0;
while S<=n
i=i+1;
S=S+i^2;
end
r=i-1;
```

Exercise: Sum of the first n natural numbers

Task:

Write two MATLAB scripts that calculate the sum of the first n natural numbers (1, 2, 3, ..., n):

1. Using a for loop.
2. Using a while loop.

Formula:

The sum of the first n natural numbers is:

$$\text{Sum} = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

But for this exercise, you will calculate the sum manually using loops.

1. Using a for loop

```
% Input: n is the number of terms
n = input('Enter a positive integer: ');

% Initialize the sum
sum_for = 0;

% For loop to calculate the sum
for i = 1:n
    sum_for = sum_for + i;
end

% Display the result
fprintf('Sum of the first %d natural numbers using for loop:
%d\n', n, sum_for);
```

2. Using a while loop

```
% Input: n is the number of terms
n = input('Enter a positive integer: ');
% Initialize the sum and the counter
sum_while = 0;
i = 1;

% While loop to calculate the sum
while i <= n
    sum_while = sum_while + i;
    i = i + 1;
end

% Display the result
fprintf('Sum of the first %d natural numbers using while loop:
%d\n', n, sum_while);
```

Steps for Students:

1. Run the code for different values of n (e.g., $n = 5$, $n = 10$, $n = 100$).
2. Verify the results for both the for loop and the while loop.

3. Compare the outputs from both programs to the theoretical formula $\frac{n(n+1)}{2}$.

This exercise helps you understand:

- How for and while loops work.
- How to control loop execution.
- How to use counters to manage loop conditions.

Exercices with Solution

Exercise 1:

Write a script in Matlab, that asks the user for a number, and then informs them whether the number is positive or negative (ignore the case where the number is zero).

Solution of Exercise 1: Script to check if the number is positive or negative.

MATLAB Script:

```
% Ask the user for input
num = input('Please enter a number: ');

% Check if the number is positive or negative
if num > 0
    disp('The number is positive.');
```

```
elseif num < 0
    disp('The number is negative.');
```

```
end
```

Explanation:

- `input('Please enter a number: ')` prompts the user to input a number. The input is stored in the variable `num`.

- The if statement checks if num is greater than 0, and if so, it displays "The number is positive."
- The elseif statement checks if num is less than 0, and if so, it displays "The number is negative."
- We do not need to check for zero, as per the instructions.

Exercise 2:

Write a script in Matlab, that asks the user for two numbers and then informs them whether their product is negative or positive (ignore the case where the product is zero).

However, note: do not calculate the product of the two numbers.

Solution of Exercise 2: Script to check if the product of two numbers is positive or negative.

MATLAB Script:

```
% Ask the user for two numbers

num1 = input('Please enter the first number: ');
num2 = input('Please enter the second number: ');

% Check the signs of the numbers and determine the product sign
if (num1 > 0 && num2 > 0) || (num1 < 0 && num2 < 0)
    disp('The product is positive.');
```

```
elseif (num1 > 0 && num2 < 0) || (num1 < 0 && num2 > 0)
    disp('The product is negative.');
```

```
end
```

Explanation:

1. Input the numbers: The script prompts the user to enter two numbers using the input function.
2. Check the signs:
 - If both numbers are positive ($\text{num1} > 0 \ \&\& \ \text{num2} > 0$) or both are negative ($\text{num1} < 0 \ \&\& \ \text{num2} < 0$), then their product will be positive.
 - If one number is positive and the other is negative ($\text{num1} > 0 \ \&\& \ \text{num2} < 0$ or $\text{num1} < 0 \ \&\& \ \text{num2} > 0$), then the product will be negative.
3. Avoid calculating the product: We don't explicitly compute the product; instead, we use the signs of the individual numbers to determine the result.

This way, we follow the instruction to not calculate the product while still determining if the product is positive or negative.

Exercise 3:

Write a script in Matlab, that asks the user for a number and then informs them whether the number is positive or negative (this time include the case where the number is zero).

Solution of Exercise 3: Script to check if the number is positive, negative, or zero

MATLAB Script:

```
% Ask the user for a number

num = input('Please enter a number: ');

% Check if the number is positive, negative, or zero

if num > 0

    disp('The number is positive.');
```

```
elseif num < 0

    disp('The number is negative.');
```

```
else

    disp('The number is zero.');
```

end

Explanation:

1. **Input the number:** The script prompts the user to enter a number using the input function.
2. **Check conditions:**
 - If the number is greater than zero ($\text{num} > 0$), it will display "The number is positive."
 - If the number is less than zero ($\text{num} < 0$), it will display "The number is negative."
 - If the number is exactly zero (else case), it will display "The number is zero."

This script now correctly handles all three cases (positive, negative, and zero), as required by the exercise.

Exercise 4:

Write a script in Matlab, that asks the user for two numbers and then informs them whether the product is negative or positive (this time include the case where the product could be zero).

However, note: do not calculate the product.

Note: To solve **Exercise 4** in MATLAB, where you need to determine whether the product of two numbers is positive, negative, or zero **without actually calculating the product**, we can use the signs of the two numbers:

1. If both numbers are positive or both are negative, the product is positive.
2. If one number is positive and the other is negative, the product is negative.
3. If either number is zero, the product is zero.

Here's the solution:

Solution of Exercise 4: Script to check if the product of two numbers is positive, negative, or zero.

MATLAB Script:

```
% Ask the user for two numbers

num1 = input('Please enter the first number: ');
num2 = input('Please enter the second number: ');

% Check the signs of the numbers to determine the product sign

if num1 == 0 || num2 == 0

    disp('The product is zero.');
```

```
elseif (num1 > 0 && num2 > 0) || (num1 < 0 && num2 < 0) disp('The
product is positive.');
```

```
elseif (num1 > 0 && num2 < 0) || (num1 < 0 && num2 > 0)

    disp('The product is negative.');
```

```
end
```

Explanation:

1. Input: The script prompts the user to input two numbers using the input function.
2. Check for zero: If either num1 or num2 is zero ($\text{num1} == 0 \parallel \text{num2} == 0$), the script immediately informs the user that the product is zero.
3. Check for positive or negative product:
 - If both numbers are either positive or both negative ($(\text{num1} > 0 \ \&\& \ \text{num2} > 0) \parallel (\text{num1} < 0 \ \&\& \ \text{num2} < 0)$), the product is positive.
 - If one number is positive and the other is negative ($(\text{num1} > 0 \ \&\& \ \text{num2} < 0) \parallel (\text{num1} < 0 \ \&\& \ \text{num2} > 0)$), the product is negative.

This way, we do not calculate the product but deduce its sign based on the values of the numbers.

Exercise 5

Write a script in Matlab, that asks the user for a child's age. Then, inform them of the child's category:

- "Poussin" for 6 to 7 years old
- "Pupille" for 8 to 9 years old
- "Minime" for 10 to 11 years old
- "Cadet" for 12 years old and older

Can multiple equivalent scripts be created to achieve this result?

Solution to Exercise 5: In MATLAB, where we classify a child's age into categories:

MATLAB Script:

```
% Ask the user for the child's age
age = input('Please enter the child's age: ');
% Check the age range and categorize the child
if age >= 6 && age <= 7
    disp('Category: Poussin');
elseif age >= 8 && age <= 9
    disp('Category: Pupille');
elseif age >= 10 && age <= 11
    disp('Category: Minime');
elseif age >= 12
    disp('Category: Cadet');
else
    disp('The age entered is outside the valid range.');
```

end

Explanation:

1. **Input:** The script prompts the user to enter the child's age using `input`.
2. **Age Categorization:**
 - If the age is between 6 and 7 (inclusive), the script categorizes the child as "Poussin".
 - If the age is between 8 and 9 (inclusive), the child is categorized as "Pupille".
 - If the age is between 10 and 11 (inclusive), the child is categorized as "Minime".
 - If the age is 12 or older, the child is categorized as "Cadet".
3. **Error Handling:** If the age entered is less than 6, the script will inform the user that the entered age is outside the valid range.

Can Multiple Equivalent Scripts Be Created?

Yes, you can create multiple equivalent scripts to achieve the same result, in different ways.

Here are a few variations:

1. Using `switch` statement (alternative approach):

MATLAB Script:

`% Exercise 5: Alternative using switch-case`

`% Ask the user for the child's age`

`age = input('Please enter the child's age: ');`

`% Using a switch-case structure with ranges`

`switch true`

`case (age >= 6 && age <= 7)`

`disp('Category: Poussin');`

`case (age >= 8 && age <= 9)`

`disp('Category: Pupille');`

`case (age >= 10 && age <= 11)`

`disp('Category: Minime');`

`case (age >= 12) disp('Category: Cadet');`

`otherwise`

`disp('The age entered is outside the valid range.');`

`end`

2. Using logical indexing (another variation):

MATLAB Script:

```
% Ask the user for the child's age
```

```
age = input('Please enter the child's age: ');
```

```
% Define the age categories
```

```
categories = {'Poussin', 'Pupille', 'Minime', 'Cadet'};
```

```
age_limits = [6 8 10 12]; % Corresponding lower limits for categories
```

```
% Check which category the age falls into
```

```
if age >= 6 && age <= 7
```

```
    disp(['Category: ', categories{1}]);
```

```
elseif age >= 8 && age <= 9
```

```
    disp(['Category: ', categories{2}]);
```

```
elseif age >= 10 && age <= 11
```

```
    disp(['Category: ', categories{3}]);
```

```
elseif age >= 12
```

```
    disp(['Category: ', categories{4}]);
```

```
else
```

```
    disp('The age entered is outside the valid range.');
```

```
end
```

Conclusion:

- The solution using if-elseif is the simplest and most direct way.
- Using switch-case or logical indexing are valid alternatives and may suit different coding styles or use cases.
- These alternative scripts can achieve the same result, but the choice of which one to use depends on personal preference or specific coding requirements.

Exercise 6:

Test the following script in Matlab, for different values of the variable a. What does it do?

```
a=-25
if a>=0 then
    disp(a)
else
    disp(-a)
end
```

Solution of Exercise 6: The MATLAB script you've provided for **Exercise 6** is as follows:

MATLAB Script:

```
a = -25; % Set the value of a
if a >= 0
    disp(a) % If a is greater than or equal to zero, display a
else
    disp(-a) % If a is negative, display the positive value of a
end
```

What does this script do?

1. Assigns a value to a:
The value of a is set to -25 in this case.
2. Checks if a is greater than or equal to zero:
The script checks whether a is a non-negative number ($a \geq 0$).
3. Branches based on the condition:
 - If a is greater than or equal to zero ($a \geq 0$), the script will display the value of a using the `disp(a)` command.
 - If a is less than zero ($a < 0$), the script will display the positive version of a using `disp(-a)`.

For $a = -25$:

- Since a is **negative** ($a = -25$), the `else` block will be executed, and the script will display the **positive value** of a (i.e., 25).

Output:

Testing with Different Values of a:

Let's test the script for different values of a.

1. If a = 10:

MATLAB Script:

```
a = 10; % Positive value of a
if a >= 0
    disp(a)
else
    disp(-a)
end
```

- Since a is positive (a = 10), the script will display 10.

Output:

3. If a = 0:

MATLAB Script:

```
a = 0; % Zero value of a
if a >= 0
    disp(a)
else
    disp(-a)
end
```

- Since a is **zero** (a = 0), the condition a >= 0 is true, so the script will display 0.

Output:

3. If $a = -100$:

MATLAB Script:

```
a = -100; % Negative value of a
if a >= 0
    disp(a)
else
    disp(-a)
end
```

- Since a is negative ($a = -100$), the script will display 100 (the positive version of a).

Output:

What does the script do in summary?

- The script displays the absolute value of a .
 - If a is positive or zero, it displays a as is.
 - If a is negative, it displays the positive version of a (i.e., it takes the absolute value).

Key takeaway:

The script effectively outputs the absolute value of a , regardless of whether a is positive or negative.

Exercise 7:

We demonstrate that the partial sum series

$$S_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

tends to $+\infty$. The problem consists of finding the number of terms of this series, called the harmonic series, needed to exceed a given arbitrary value:

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} > \text{value}$$

1. Write a function `function [s] = harmonic(n)` that, for an integer `n` passed as a parameter, calculates the sum:

$$s = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

2. Write a MATLAB script that:
 - Asks the user for a value strictly greater than 1 and less than or equal to 8 (and keeps asking until the input satisfies the condition).
 - Displays the number of terms needed to exceed this value and the result of the sum (you can use the harmonic function defined above).

Solution of exercise 7:

1) Function `harmonic(n)`:

The first task is to create a function that calculates the harmonic sum up to a given integer `n`.

```
function [s] = harmonic(n)
    % Function to compute the harmonic sum up to n
    s = 0; % Initialize sum to zero
    for i = 1:n
        s = s + 1/i; % Add the reciprocal of i to the sum
    end
end
```

Explanation of the `harmonic` function:

- The function takes an integer n as input.
- It initializes the sum s to zero.
- It uses a `for` loop to iterate through all integers from 1 to n , adding $\frac{1}{i}$ to the sum for each iteration.
- Finally, it returns the sum s .

2) MATLAB Script to Find the Number of Terms:

Now, we will write a MATLAB script that repeatedly asks the user for a value between 1 and 8, and calculates the number of terms needed for the harmonic series to exceed that value.

```
% Script to find the number of terms needed to exceed a specified value
% Ask the user for a value strictly greater than 1 and less than or equal
to 8
while true
    value = input('Enter a value strictly greater than 1 and less than or
equal to 8: ');
    if value > 1 && value <= 8
        break; % Exit the loop if the value is valid
    else
        disp('The value must be greater than 1 and less than or equal to 8.
Please try again. ');
    end
end
% Initialize variables
n = 1; % Start from the first term
sum = 0; % Initialize the harmonic sum
% Find the number of terms needed to exceed the value
while sum <= value
    sum = harmonic(n); % Compute the harmonic sum for the current n
    n = n + 1; % Increment the number of terms
end

% Display the result
fprintf('Number of terms required: %d\n', n - 1);
```

```
fprintf('Harmonic sum for %d terms: %.4f\n', n - 1, sum);
```

Explanation of the MATLAB Script:

1. Input Validation: The while true loop asks the user to input a value greater than 1 and less than or equal to 8. If the user enters a value that doesn't satisfy this condition, it will keep asking for a valid input.
2. Initialization:
 - n is initialized to 1 (starting from the first term).
 - sum is initialized to 0 to begin summing the harmonic terms.
3. Loop to Find the Number of Terms:
 - The while sum <= value loop keeps adding terms from the harmonic series (sum = harmonic(n)) until the sum exceeds the input value.
 - The number of terms n is incremented until the sum exceeds the given threshold.
4. Displaying the Results: Once the sum exceeds the given value, the script prints the number of terms and the corresponding harmonic sum.

Example Run:

If the user enters 3.5 as the target value:

```
Enter a value strictly greater than 1 and less than or equal to 8:
```

```
3.5
```

```
Number of terms required: 6
```

```
Harmonic sum for 6 terms: 2.45
```

This indicates that 6 terms are required for the harmonic sum to exceed 3.5.

Conclusion:

This solution allows the user to interactively find out how many terms of the harmonic series are needed to exceed a given value between 1 and 8. The function harmonic(n) computes the sum, and the script uses it to determine how many terms exceed the specified value.

7. Polynomials in MATLAB

MATLAB provides powerful tools for working with polynomials, including the ability to define polynomials, find their roots (zeros), and perform various operations like addition, multiplication, and division.

7.1 Polynomials:

In MATLAB, polynomials can be represented using vectors, where each element corresponds to a coefficient of the polynomial. The vector elements are ordered in descending powers of the variable.

For example, the polynomial $p(x) = 2x^3 + 3x^2 - x + 5$ can be represented as:

```
>> p = [2 3 -1 5]; % Polynomial 2x^3 + 3x^2 - x + 5
```

Here, the vector [2 3 -1 5] represents the polynomial $2x^3 + 3x^2 - x + 5$ or $2x^3 + 3x^2 - x + 5$, with the first element being the coefficient of x^3 , the second element being the coefficient of x^2 , and so on.

7.2 Polynomial Zeros:

You can find the roots (zeros) of a polynomial using the `roots()` function, which calculates the values of x for which the polynomial equals zero.

For example, to find the zeros of the polynomial $p(x) = 2x^3 + 3x^2 - x + 5$:

```
>> zeros_p = roots(p); % Finds the roots of the polynomial p.
```

This will return the values of x that satisfy $p(x) = 0$.

7.3 Polynomial Operations:

MATLAB allows you to perform several operations on polynomials, such as addition, subtraction, multiplication, and division.

1. **Polynomial Addition and Subtraction:** Polynomials can be added or subtracted directly by adding or subtracting their coefficient vectors.

```
>> p1 = [2 3 -1 5]; % First polynomial.

>> p2 = [1 -4 2]; % Second polynomial.

>> sum_p = p1 + p2; % Polynomial addition.

>> diff_p = p1 - p2; % Polynomial subtraction.
```

2. Polynomial Multiplication: Polynomials can be multiplied using the **conv()** function, which performs the convolution of two polynomials.

```
>> prod_p = conv(p1, p2); % Multiplies p1 and p2,
```

3. Polynomial Division: To divide one polynomial by another, use the **deconv()** function, which returns both the quotient and the remainder.

```
>> [quotient, remainder] = deconv(p1, p2); % Divides p1 by p2.
```

Summary

In MATLAB, you can:

4. Represent polynomials using coefficient vectors.
5. Find the zeros (**roots**) of a polynomial using **roots()**.
6. Perform various polynomial operations like addition, subtraction, multiplication (**conv()**), and division (**deconv()**).

These tools make MATLAB a powerful environment for working with polynomials in numerical and symbolic computations.

MATLAB

Chapter V: Graphs and data visualization in Matlab

Based on the principle that an image is better than a long speech, Matlab offers a powerful visualization system that allows the presentation and graphical display of data in an efficient and easy way.

In mathematics, consider a function $f: \mathbb{R} \rightarrow \mathbb{R}$ defined on $\mathbb{R}$. The graph of such a function, denoted G_f , is defined as the set:

$$G_f = \{(x, f(x)) \mid x \in \mathbb{R}\}$$

The graph of a function is therefore the set of pairs $(x, y) \in \mathbb{R} \times \mathbb{R}$ that satisfy the equation:

$$y = f(x)$$

Formally, the curve of f corresponds to the plot of this infinite set of points. In practice, it is not possible to plot each of these points by hand. Therefore, we typically obtain an approximation of the curve through graphical analysis, such as studying the function's growth, calculating tangents at points of interest, and determining asymptotes.

Translation in Matlab:

The MATLAB approach is different from the mathematical approach used to obtain the shape of the curve. In fact, the "point-by-point" approach is used. Even though only a finite number of points can be represented, the computational power allows this number to be large enough to provide a very good approximation of the desired curve. The cloud of points obtained is then completed by connecting successive point pairs, which results in a continuous plot.

Two functions are available to create such plots: `plot2d` and `plot`. While the `plot` function can be used instead of `plot2d`, it is a version adapted from MATLAB (the paid software, with MATLAB being the free version). The `plot` function is more limited: it does not allow multiple curves to be plotted simultaneously. Therefore, it is better to use `plot2d`, which is a function specific to MATLAB.

Explanation:

In MATLAB, the graphical representation of a function is achieved through the "point-by-point" method. Although only a finite number of points can be computed, the large number of points can provide an excellent approximation of the curve. When the points are plotted, they are typically connected in a continuous manner, resulting in a smooth curve.

7. The `plot2d` function is a more flexible function for plotting in MATLAB, which allows for multiple curves to be plotted on the same graph.
8. The `plot` function, while available in many other software environments, is more limited in MATLAB in terms of functionality compared to `plot2d`, specifically when it comes to plotting multiple curves simultaneously.

4o mini

In this part of the course, we will present the basic principles necessary to draw curves in Matlab.

Graphic Windows:

In addition to the main window and the editor, there is a third type of window: graphic windows. These windows are named Figure followed by a number that corresponds to their creation order.

Multiple graphic windows can be opened simultaneously. Additionally, within the same window, multiple plots can be overlaid by drawing them one after the other.

The Edit tab of a graphic window allows you to modify the title, add a legend, etc. This tab also allows you to clear the figure (without closing the window).

To clear the content of all graphic windows simultaneously, the `clf` command is executed in the console.

1. The plot function:

The `plot` function can be used with vectors or matrices. It draws lines recording definite coordinate points in its arguments, and it has several forms:

- ❖ **If it contains two vectors of the same size as arguments:** it considers the values of the first vector as the elements of the X-axis (the abscissas), and the values of the second vector as the elements of the Y-axis (the ordinates).

Example:

```
>> A = [2, 5, 3, -2, 0]
```

A =

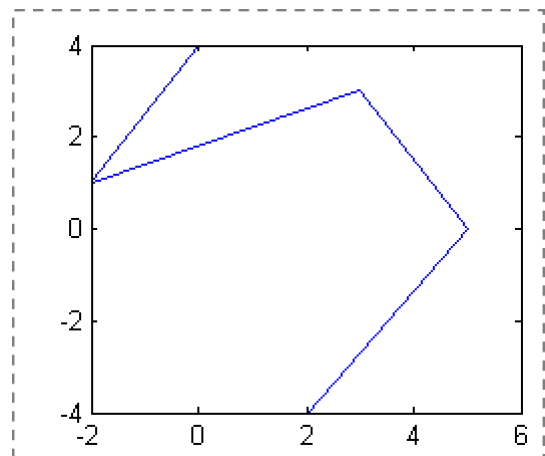
```
2      5      3     -2      0
```

```
>> B = [-4, 0, 3, 1, 4]
```

B =

```
-4      0      3      1      4
```

```
>> plot(A,B)
```



- ❖ **If it contains a single vector as an argument,** it considers the values of the vector as the elements of the Y-axis (the ordinates), and their relative positions will define the X-axis (the abscissas).

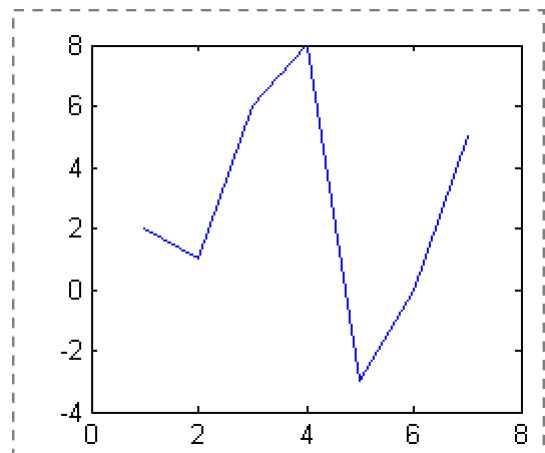
Example:

```
>> V = [2, 1, 6, 8, -3, 0, 5]
```

V =

```
2      1      6      8     -3      0      5
```

```
>> plot(V)
```



- ❖ **If it contains a single matrix as argument:** it considers the values of each column as the elements of the Y axis, and their relative positions (the row number) as the values of the X axis. Therefore, it will give several columns (one for each column).

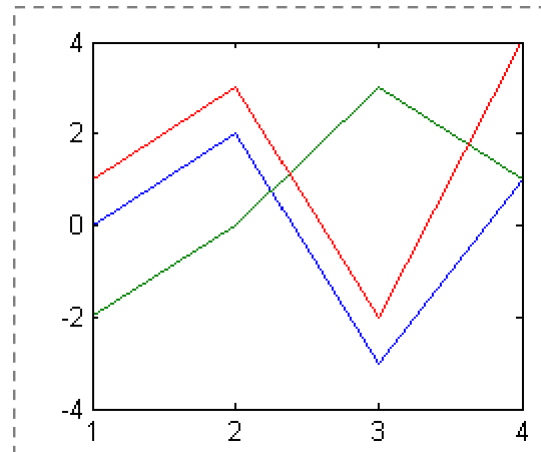
Example:

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

M =

0	-2	1
2	0	3
-3	3	-2
1	1	4

```
>> plot(M)
```



❖ If it contains two matrices as arguments:

it considers the values of each column of the first matrix as the elements of the X axis, and the values of each column of the second matrix as the values of the Y axis.

Example:

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

K =

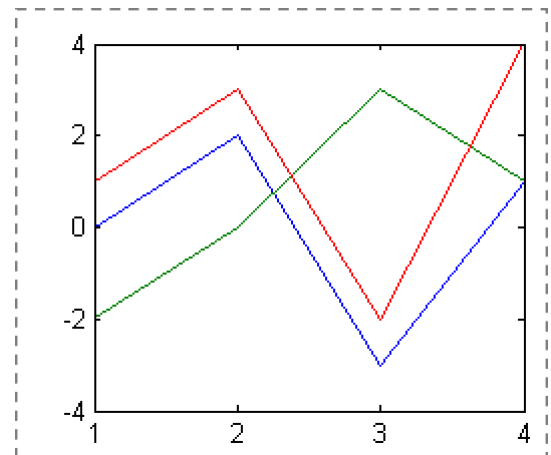
1	1	1
2	2	2
3	3	3
4	4	4

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

M =

0	-2	1
2	0	3
-3	3	-2
1	1	4

```
>> plot(K,M)
```



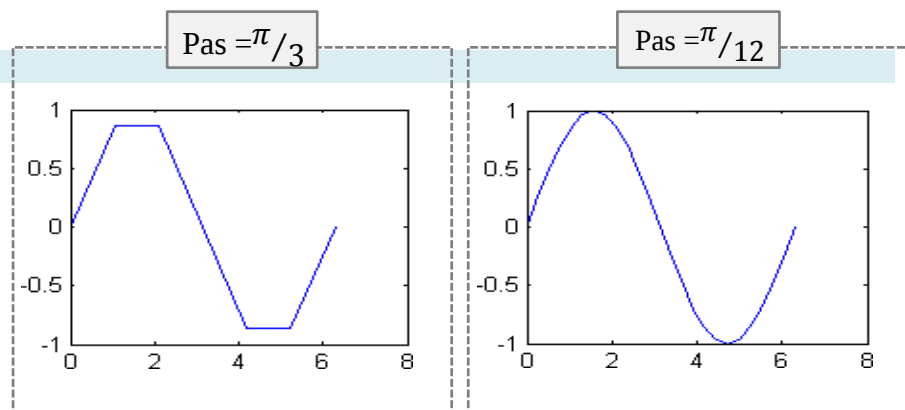
It is obvious that the higher the number of coordinates the more precise the curve becomes. For example to draw the curve of the function $y=\sin(x)$ on $[0, 2\pi]$ we can write:

The first figure

```
>> x=0:pi/3:2*pi;
>> y=sin(x);
>> plot(x,y)
```

The second figure

```
>> x=0:pi/12:2*pi;
>> y=sin(x);
>> plot(x,y)
```



2. Change the appearance of a curve:

It is possible to manipulate the appearance of a curve by changing the color of the curve, the shape of the coordinate points and the type of line connecting the points.

To do this, we add a new argument (which we can call a marker) of type string to the plot function like this:

plot (x, y, 'marke')

The content of the marker is a combination of a set of special characters collected in the following table:

Curve color		Point representation	
character	its effect	character	its effect
b or blue	blue curve	.	A point .
g or green	green curve	o	A circle ●
r or red	red curve	x	The symbole x
c or cyan	between green and blue	+	The symbole +
m or magenta	bright purplish red	*	A star *
y or yellow	yellow curve	s	A square ■
k or black	black curve	d	A diamond ◆
curve Style		v	lower triangle ▼
character	its effect	^	Upper triangle ▲
-	online full —	<	Left triangle ◀
:	dotted	>	triangle ▶
-.	A point dashed -.-.-.-	p	Pentagram ★
--	dash - - - - -	h	Hexagram ⬠

Example:

Let's try to draw the function $y = \sin(x)$ for $x = [0 \dots 2\pi]$ with a pitch = $\pi/6$.

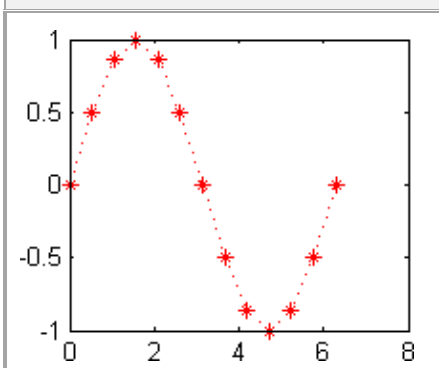
```
>> x=0:pi/6:2*pi;
```

```
>> y=sin(x);
```

Changing the marker produces different results, and here are some examples:

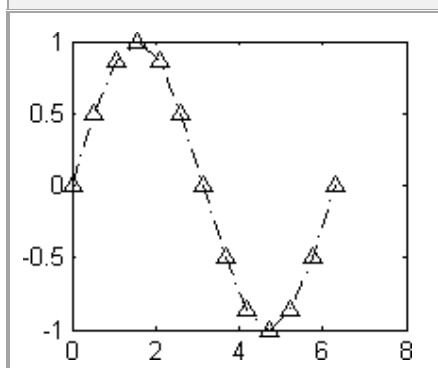
Color red, dotted and with stars

plot(x, y, 'r:*')



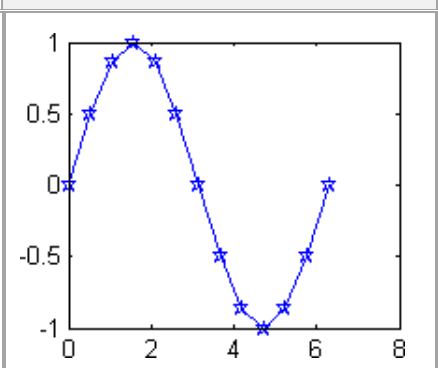
Color black, dashed and with rectangles sup

plot(x, y, 'black-.^')



Blue color, full line and with pentagrams

plot(x, y, 'pb-')



3. Annotation of a figure:

In a figure, it is better to put a textual description helping the user to understand the meaning of the axes and to know the purpose or interest of the visualization concerned.

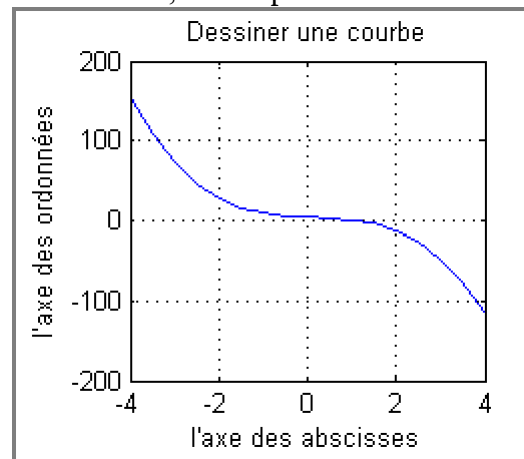
It is also very important to be able to point out locations or significant points in a figure by a comment indicating their importance.

- ✓ To give a title to a figure containing a curve we use the function **title** like this:
`>> title('titre of the figure')`
- ✓ To give a title for the vertical axis of y-ordinates, we use the **ylabel** function like this:
`>> ylabel('This is the Y-axis')`
- ✓ To give a title for the horizontal x-axis, we use the **xlabel** function like this:
`>> xlabel('This is the X-axis')`
- ✓ To write a text (a message) on the graphic window at a position indicated by the x and y coordinates, we use the **text** function like this:
`>> text(x, y, 'this point is important')`
- ✓ To put a text on a position chosen manually by the mouse, we use the **gtext** function, which has the following syntax:
`>> gtext('This point is selected manually')`
- ✓ To put a grid (**grid**), use the **grid** (or **grid on**) command. To remove it reuse the same **grid** (or **grid off**) command.

Example:

Let's draw the function: $y = -2x^3 + x^2 - 2x + 4$ for x varies from -4 to 4, with a pitch=0.5.

```
>> x=-4:0.5:4;
>> y=-2*x.^3+x.^2-2*x+4;
>> plot(x,y)
>> grid
>> title('Dessiner une courbe')
>> xlabel('l'axe des abscisses')
>> ylabel('l'axe des ordonnées')
```



4. Draw multiple curves in the same figure:

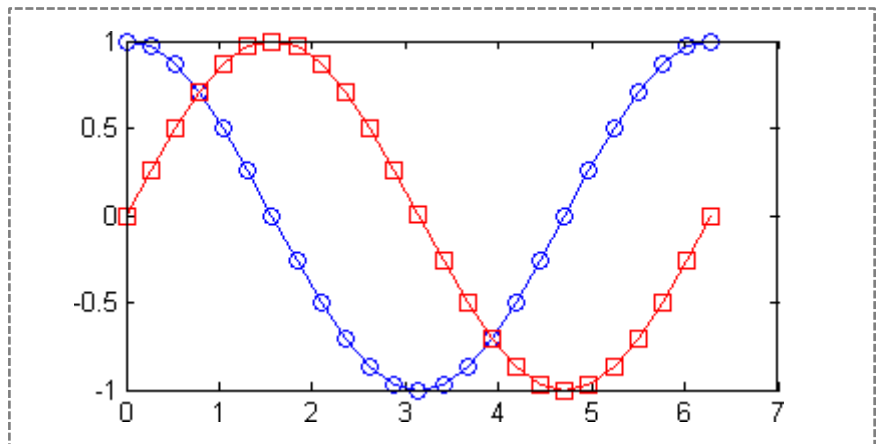
By default in Matlab, each new drawing with the plot command erases the previous one. To force a new curve to coexist with the previous curves, there are at least three methods:

4.1 The hold command:

The **hold** (or **hold on**) command activates the 'preserve old curves' mode which allows the display of several curves in the same figure. To cancel its effect just rewrite **hold** (or **hold off**).

For example, to draw the curve of the two functions $\cos(x)$ and $\sin(x)$ in the same figure, we can write:

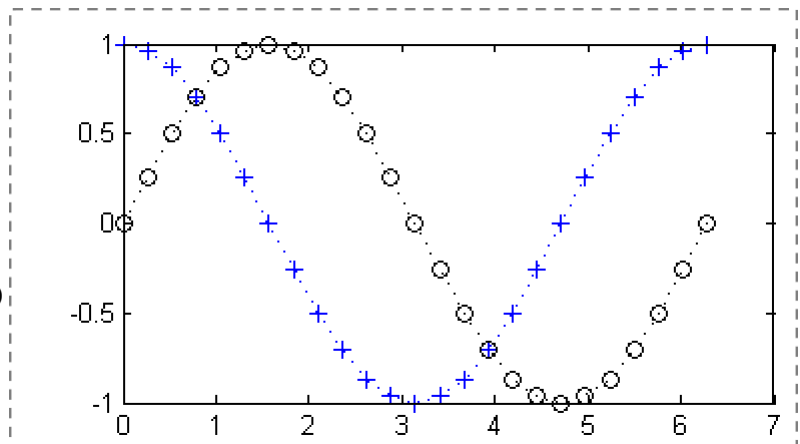
```
>> x=0:pi/12:2*pi;
>> y1=cos(x);
>> y2=sin(x);
>> plot(x,y1,'b-o')
>> hold on
>> plot(x,y2,'r-s')
```



4.2 Use plot with multiple arguments:

One can use plot with several couples (x,y) or triples (x,y, 'marker') as arguments. For example to draw the same previous functions one writes:

```
>> x=0:pi/12:2*pi;
>> y1=cos(x);
>> y2=sin(x);
>> plot(x,y1,'b:+',x,y2,'k:o')
```



To obtain several superimposed plots, simply enter them in the `plot(x1,y1,x2,y2, ...)` function. You can also use the **hold** on function to overlay graphics.

Example:

The following two sets of instructions produce the same graph:

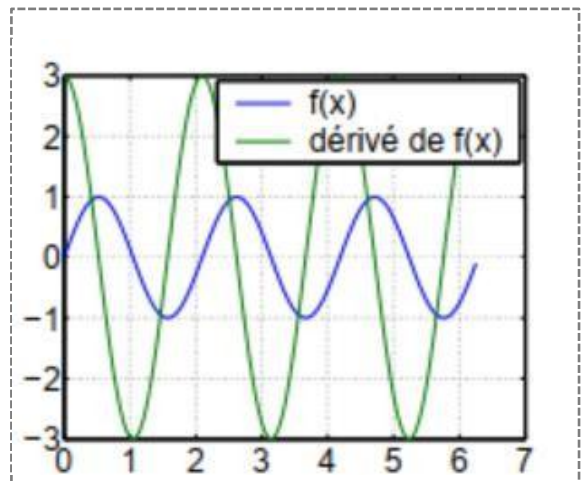
```
>> x=0:.05:2*pi;
>> y1=sin(3*x);
>> y2=3*cos(3*x);
```

1- Using the function `plot(x1,y1,x2,y2,...)`

```
>> plot(x,y1,x,y2)
>> legend('f(x)', 'derived from f(x)')
>> grid on
```

2- Using the hold function

```
>> plot(x,y1)
>> grid on
>> hold on
```



```
>> plot(x,y2)
>> legend('f(x)', 'derived from f(x)')
>> hold off
```

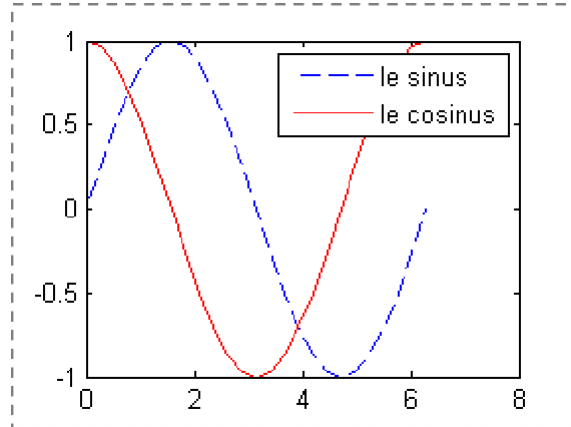
4.3 Using matrices as argument for the plot function:

In this case several curves are obtained automatically for each column (or sometimes the rows) of the matrix. This case has been presented earlier.

It is possible to distinguish them by putting a legend indicating the names of the curves.

To do this, we use the **legend** function, as illustrated by the following example which draws the curves of the two functions $\sin(x)$ and $\cos(x)$:

```
>> x=0:pi/12:2*pi;
>> y1=sin(x);
>> y2=cos(x);
>> plot(x,y1,'b--',x,y2,'-r')
>> legend('sinus','cosinus')
```

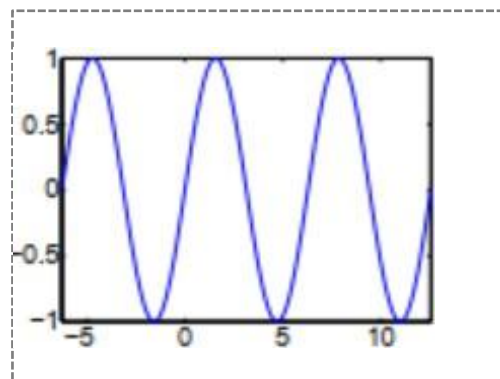


It is possible to move the legend (which is by default in the upper right corner) using the mouse with a drag and drop.

4.4 Using the fplot function

As for the `fplot` function, it is written as follows: `fplot(fon,lim)` where `fon` indicates the name of the function to be plotted in the form of a string (in quotes), and `lim` defines the axis limits. For the limits, they must be written in square brackets with the following syntax: `lim = [Xmin Xmax Ymin Ymax]`. To do this, we use the **fplot** function, as illustrated by the following example, which draws the curves of the two functions $\sin(x)$:

```
>> f='sin';
>> fplot(f,[-2*pi 4*pi])
```



5. Manipulating the axes of a figure:

Matlab calculates by default the limits (minimum and maximum) of the X and Y axes and automatically chooses the appropriate partitioning. But it is possible to control the aspect of the axes via the **axis** command.

To define the axis limits it is possible to use this command with the following syntax:

axis ([xmin xmax ymin ymax]) Or **axis ([xmin,xmax,ymin,ymax])**

With: **xmin** and **xmax** set the minimum and maximum for the x-axis.

ymin and **ymax** define the minimum and maximum for the y-axis.

To return to the default display mode, we write the command: **axis auto**

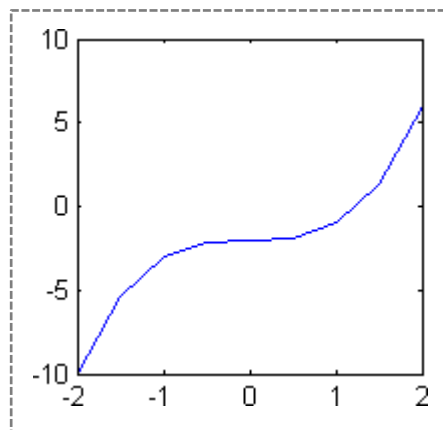
Example:

$$f(x) = x^3 - 2$$

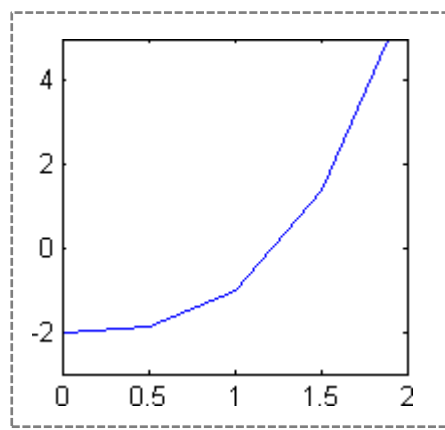
```
>> x=-2:0.5:2;
```

```
>> y=x.^3-2;
```

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



```
>> axis auto
```



```
>> axis([0,2,-3,5])
```

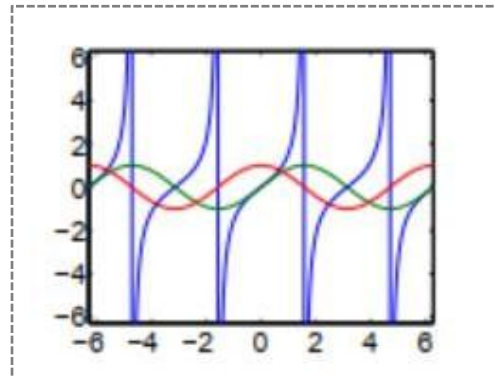
Other options in the axis command include:

- To make the size of the two axes identical (the size and not the partitioning), we use the **axis square** command. It is named as such because it adjusts the appearance of the axes to be square-shaped.
- To make the partitioning of the two axes identical we use the **axis equal** command.
- To return to the default display and undo the changes we use the **axis auto** command.
- To make the axes invisible we use the **axis off** command. To make them visible again we use the **axis on** command.

Remark:

For superimposed functions plotted using the **fplot** function, this involves creating a matrix with the names of the desired functions.

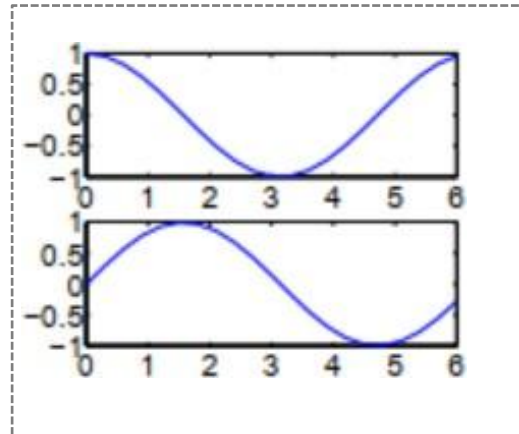
```
>> f=[tan(x),sin(x),cos(x)]';
>> fplot(f,2*pi*[-1 1 -1 1])
```



To obtain multiple graphs side by side, simply use the function `subplot(m,n,p)`, which divides the figure into an m by n grid of rectangular tiles. p indicates the position in the grid that the following `plot` function will occupy in the instructions. If you want to create a 2x2 grid for the graphs, the `subplot` function to write before each function to be displayed is defined as follows:

1	2
<code>subplot(2,2,1)</code>	<code>subplot(2,2,2)</code>
3	4
<code>subplot(2,2,3)</code>	<code>subplot(2,2,4)</code>

```
>> t = 0:0.3:2*pi;
>> x = cos(t);
>> y = sin(t);
>> figure
>> subplot(2,1,1);
>> plot(t,x)
>> subplot(2,1,2);
>> plot(t,y)
```

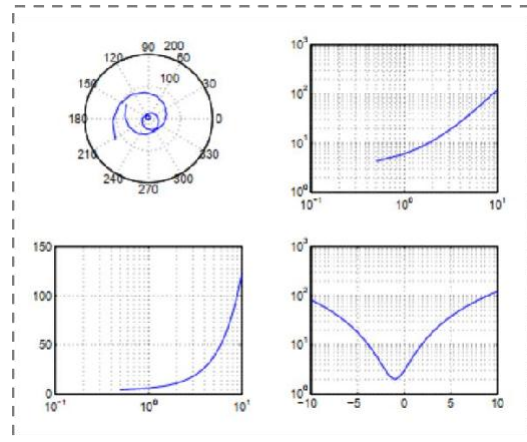


Note: Several functions for plotting graphs are available for different scales.

<code>plot(x,y)</code>	linear x-y	<code>polar(theta,r)</code>	polar
<code>loglog(x,y)</code>	logarithmic x-y	<code>bar(x,y)</code>	bars
<code>semilogx(x,y)</code>	logarithmic x, linear y	<code>stairs(x,y)</code>	staircase
<code>semilogy(x,y)</code>	linear x, logarithmic y		

Example:

```
>> x=-10:.5:10;
>> y=x.^2+x.*2+3;
>> subplot(2,2,1), polar(x,y), grid on
>> subplot(2,2,2), loglog(x,y),grid on
>> subplot(2,2,3), semilogx(x,y), grid on
>> subplot(2,2,4), semilogy(x,y), grid on
```



6. Other types of graphs:

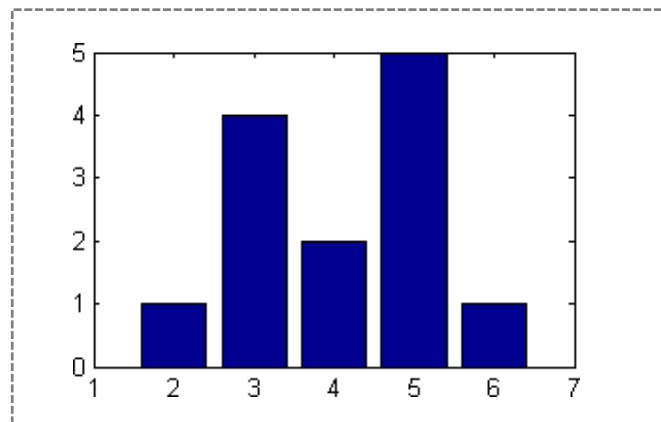
The Matlab language not only allows the display of points to draw curves, but it also offers the possibility of drawing bar graphs **and** histograms.

To draw a bar graph we use the bar function which has the same working principle as the **plot** function.

Example:

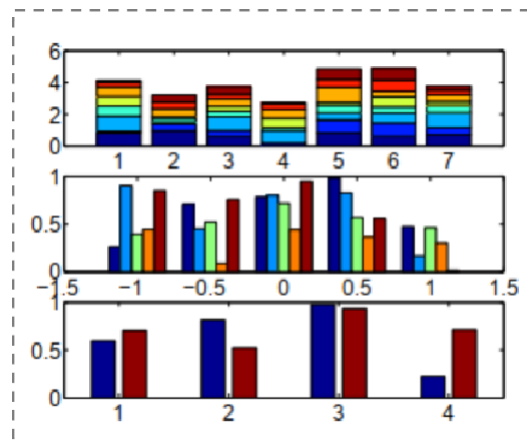
1-

```
>> X=[2,3,5,4,6];
>> Y=[1,4,5,2,1];
>> bar(X,Y)
```



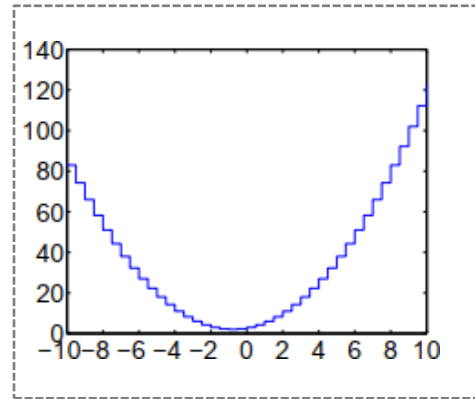
2-

```
>> subplot(3,1,1), bar(rand(7,8),'stacked')
>> subplot(3,1,2), bar(-1:.5:1,rand(5),1)
>> subplot(3,1,3), bar(rand(4,2),.75)
```



3-

```
>> x=-10:.5:10;
>> y=x.^2+x.*2+3;
>> stairs(x,y)
```



It is possible to change the appearance of the sticks, and there is the **barh** function that draws the sticks horizontally, and the **bar3** function that adds a 3D effect.

Among the very interesting drawing functions presented (lack of space) we can find: **hist**, **stairs**, **stem**, **pie**, **pie3**, ...etc. (that we encourage you to study them).

We also point out that Matlab allows the use of a coordinate system other than the Cartesian system such as the polar coordinate system (for more details look for functions **compass**, **polar** and **rose**).

We summarise all this chapter in:

1. Basic Plotting

- **Line Plot:** This fundamental type of plot is ideal for visualizing data trends over a continuous variable. **For example:**

```
x = 0:0.1:2*pi; % Generate a vector from 0 to 2π
y = sin(x); % Calculate the sine of each element in x
plot(x, y); % Plot y against x
title('Sine Function'); % Title for the plot
xlabel('x'); % X-axis label
ylabel('sin(x)'); % Y-axis label
grid on; % Enable grid lines
```

- **Scatter Plot:** Useful for displaying the relationship between two variables. **For example:**

```
x = rand(1, 100); % Generate random x values
y = rand(1, 100); % Generate random y values
scatter(x, y); % Create a scatter plot
title('Scatter Plot Example');
xlabel('X Values');
ylabel('Y Values');
```

2. Multiple Plots

- **Subplots:** This feature allows you to display multiple plots in a single figure window. **For example:**

```
subplot(2, 1, 1); % 2 rows, 1 column, first subplot
plot(x, y);
title('Sine Function');

subplot(2, 1, 2); % 2 rows, 1 column, second subplot
plot(x, cos(x));
title('Cosine Function');
```

3. Bar and Pie Charts

- **Bar Graphs:** Effective for comparing categorical data. **For example:**

```
categories = {'A', 'B', 'C', 'D'};
values = [10, 15, 7, 20];
bar(values); % Create a bar graph
set(gca, 'xticklabel', categories); % Set x-tick labels
title('Bar Graph Example');
```

- **Pie Charts:** Useful for showing proportions of a whole. **For example:**

```
data = [1, 2, 3, 4];
pie(data); % Create a pie chart
title('Pie Chart Example');
```

4. 3D Plots

- **3D Surface Plot:** Ideal for visualizing data in three dimensions. **For example:**

```
[X, Y] = meshgrid(-5:0.5:5, -5:0.5:5); % Create a grid of values
Z = sin(sqrt(X.^2 + Y.^2)); % Compute Z values
surf(X, Y, Z); % Create a 3D surface plot
title('3D Surface Plot');
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
```

5. Customizing Plots

MATLAB provides extensive options for customizing plots, including:

- **Adding Legends:**

```
legend('sin(x)', 'cos(x)'); % Add a legend to identify plots
```

- **Changing Colors and Markers:**

```
plot(x, y, 'r--o'); % Red dashed line with circular markers
```

- **Setting Axes Limits:**

```
axis([0 2*pi -1 1]); % Set limits for the x and y axes
```

6. Exporting Figures

You can save your figures in various formats (e.g., PNG, JPEG, PDF) using the `saveas` function:

```
saveas(gcf, 'myPlot.png'); % Save the current figure as a PNG file
```

7. Transfer figures to a Word document:

The `print` statement without options sends the figure to the default printer on the computer.

The `print` function includes a wide range of options. To see all of them, enter the command `help print`. Here are the most useful commands:

Option	Explication	Example
<nameofFile>	Save the figure under the defined name in the current directory. The default save format is Postscript. It is possible to add an extension to the file name, which will determine the format in which it is saved.	<code>print exp3</code>
<code>-f<nameofFigure></code>	Specifies the name of the figure to be processed (prints if no other option is specified).	<code>print -f2</code>
<code>-dbitmap</code>	Copies the open figure to the clipboard in bitmap format. The bitmap format creates a photographic copy of the graphic in question.	<code>print -dbitmap</code>
<code>-dsetup</code>	Let me know if you need further help!	<code>print -dsetup</code>
<code>-dmeta</code>	Copies the open figure to the clipboard in metafile format. The metafile format creates a copy of the graphic, which, once pasted into Word (see procedure following the table), can be edited (layout, text, etc.).	<code>print -dmeta</code>
<code>-djpeg<nn></code>	Saves in JPEG format with a quality of nn.	<code>print -djpeg90</code>

To copy a metafile graphic into Word, go to the **Edit** menu → **Paste Special**.

A window will appear. If you have a metafile image in your clipboard, simply click on **Picture (Enhanced Metafile)** and then click **OK** for the file to appear. Once the image is in the Word document, right-click on the image and select **Edit Picture** to modify it.

To use more than one option at a time, follow this order:

```
print -tool -options filename.
```

Instructions
<pre>print -dbitmap -f3 % copie dans le presse-papiers de la figure 3</pre>

8. Figure Editor:

options. Sometimes, it's also necessary to try several options to determine which one produces the best chart. This is why all the options covered in this section have a second way to be applied to the chart: they can be selected through the menus and buttons on the figure itself.

This allows us to modify them and see the results at our convenience without having to create a new figure each time.

Simply save the figure at the end to keep these options. By default, the figure is saved in the .fig format, which is Matlab's figure format, but it's also possible to save it in another format, such as bitmap (.bmp).

It is also possible to copy the figure once it's open without needing to use the commands.

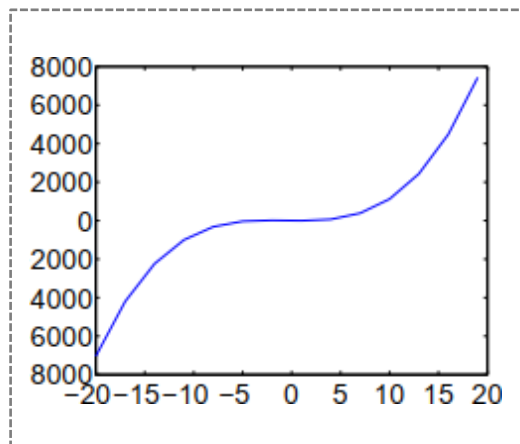
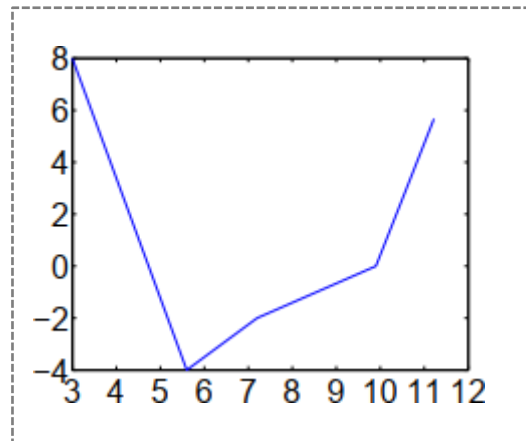
Simply go to the **Edit** menu → **Copy Figure**. To choose the format in which the figure will be saved, go to the **Edit** menu → **Copy Options**. In this preferences window, select the desired format. See the table in the previous section for the properties of each format and instructions on how to paste the copy into Word.

Using the buttons, you can create a new figure, open a figure, save the current figure, or print it.

It's also possible to modify the chart using the arrow icon. Once this button is pressed, simply double-click on an item in the chart to open the **Property Editor** window. With this editor, it's possible to modify the selected object using the various options available. Here are some examples of interesting properties of objects that we can modify through this editor.

Exercice:

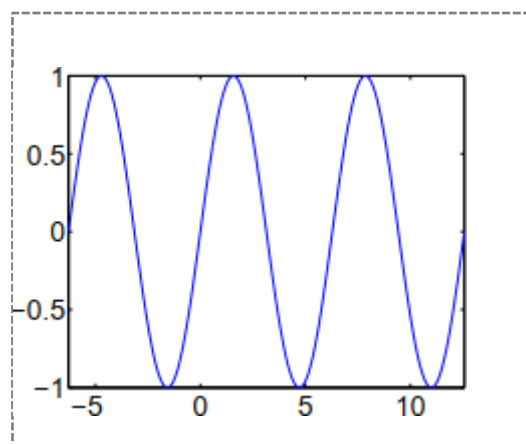
```
% experimental data
>> X=[3 5.6 7.2 9.9 11.22];
>> Y=[8 -4 -2 0 5.67];
>> plot(X,Y)
```



```
% function
```

```
>> X1=-20:3:20;
>> Y1=X1.^3+2*X1.^2-8*X1+1;
>> plot(X1,Y1)
```

```
>> f='sin';
>> fplot(f,[-2*pi 4*pi])
```



9. Symbolic Calculation in Matlab:

Matlab provides a Symbolic Math Toolbox for performing symbolic calculations. It allows you to perform algebraic manipulations, such as expanding, simplifying, and solving expressions symbolically. This toolbox is essential for handling exact mathematical expressions and solving problems analytically.

9.1 Calling the Symbolic Toolbox:

To use symbolic capabilities in Matlab, first, you need to create symbolic variables and expressions. You do this by using the `syms` command.

Example:

```
>> syms x y
```

```
>> f = x^2 + y^2;
```

Here, x and y are symbolic variables, and f is a symbolic expression.

9.2 Expanding and Transforming Expressions:

Matlab allows you to expand and simplify expressions using functions like `expand()` and `simplify()`. These functions help manipulate symbolic expressions to a more convenient form.

- **Expanding an Expression:**

```
>> g = (x + y)^2;
```

```
>> expanded_g = expand(g); % Expands (x + y)^2 to x^2 + 2xy + y^2
```

- **Simplifying an Expression:**

```
>> simplified_g = simplify(expanded_g); % Simplifies the expression if possible.
```

9.3 Derivatives and Integrals of a Function:

Matlab can calculate symbolic derivatives and integrals using the `diff()` and `int()` functions.

- **Derivative of a Function:**

```
>> f_prime = diff(f, x); % Derivative of f with respect to x.
```

- **Integral of a Function:**

```
>> integral_f = int(f, x); % Integral of f with respect to x.
```

9.4 Taylor Series Expansion of a Function:

The Taylor series expansion is a powerful tool for approximating a function around a specific point (usually 0). In Matlab, this can be done using the `taylor()` function.

- **Taylor Series Expansion:**

```
>> taylor_expansion = taylor(f, x); % Expands f(x) around x = 0
```

To specify a different point for the expansion, you can add a second argument:

```
>> taylor_expansion_at_a = taylor(f, x, 'Order', 6, 'Point', 1); %  
Expands around x = 1 up to the 6th order
```

Summary

Using Matlab's Symbolic Toolbox, you can:

- Define symbolic variables and expressions with `syms`.
- Perform algebraic manipulations, including expansion and simplification.
- Calculate derivatives and integrals symbolically.
- Find the Taylor series expansion of functions.

These tools are essential for solving complex mathematical problems analytically in Matlab.

Exercice N° 01 :

Let A = 10 and B = 12. Give the results of the following expressions:

```
>> C = (A>B) | ~(A==B) , D = A == B
>> ~((A-B > A) & (A+B == 22)) , ans & B, ans | 0 ; ans & 0
>> [4,2,-2:2:3] ~= 2*[2,1,-1,0,1]
>> [-1 2 ; 4 6 ; 3 -7] >= [2 2 ; -1 8 ; 4 -2]
>> isequal(2*ones(3),2+zeros(3))
>> isempty([4:1;-3:-1:-1;[]]) % cette matrice est-elle vide ?
```

Exercice N° 02 :

This program calculates the square root of a number using Newton's method.

1. Manually execute the program for a=16, a=5, a=169.

- 1) Remplacer l'instruction **for** par l'instruction **while** en préservant la fonctionnalité du programme.
- 2) Modify the program so that it applies to a vector and not just a single number.
- 3) What is the impact on the program's result if a value that is too small is chosen for the variable precision (for example, precision = 3)?

```
a = input('Enter a positive number: ');
x = a/2;
precision = 6;
for i = 1:precision
    x = (x + a / x) / 2;
end
disp(x)
```

Exercice N° 03 :

Here is a MATLAB program that finds the name of the day of the week for a valid date.

- 1) Using this program, find the name of the day for one of the following dates:
 01/11/1954 05/07/1962
 11/09/2001 01/03/2012
- 2) Find the day of your birthday.
- 3) Replace the **switch** statement with an **if** statement while keeping the functionality.
- 4) Write a program that verifies the validity of a date (day/month/year); for example, the date 29/02/2009 is incorrect because the year 2009 is not a leap year.

The function **mod(a, b)** calculates the remainder of the integer division of a by b. For example:
 $\text{mod}(12, 3) = 0$, $\text{mod}(27, 4) = 3$, $\text{mod}(54, 4) = 2$.
 (If $a = xb + y$, then $\text{mod}(a, b) = y$.)

```
day = input('Enter the day : ');
month = input('Enter the month : ');
year = input('Enter the year : ');
if mod(year,4) == 0
    DurationMonth =
        [0,3,4,0,2,5,0,3,6,1,4,6];
else
    DurationMonth =
        [0,3,3,6,1,4,6,2,5,0,3,5];
end
nbrDay = year - 1900;
daynbr = daynbr + floor((daynbr -1)/4);
daynbr = daynbr + DurationMonth (month) + Day;
daynbr = mod(daynbr,7);
switch(daynbr)
    case 0 , disp('Sunday')
    case 1 , disp('Monday')
    case 2 , disp('Tuesday')
    case 3 , disp('Wednesday')
    case 4 , disp('Thursday')
    case 5 , disp('Friday')
    case 6 , disp('Saturday')
end
```

Exercice N° 04 :

The Collatz conjecture is the mathematical hypothesis that assumes the sequence (to the right) always converges to the value 1 (or more precisely the sequence: 4, 2, 1), even though there is currently no proof for it.

The Collatz sequence
For a positive integer N ($N > 0$) :
 $U_0 = N$
 $U_{n+1} = \begin{cases} U_n/2 & \text{if } U_n \text{ is even (pair)} \\ 3U_n + 1 & \text{if } U_n \text{ is odd (impair)} \end{cases}$

- 1) Find the Collatz sequence for the numbers: 5 and 3.
- 2) Write a program that generates the Collatz sequence for a given number NNN.
- 3) Transform this program into a function.

Exercice N° 05 :

- 1) Write a program that calculates the factorial of an integer n ($n!$).
- 2) Write a program that calculates the following two sums:

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}, \quad \text{et} \quad \sum_{k=1}^n \frac{(-1)^k}{k^2} = -1 + \frac{1}{4} - \frac{1}{9} + \dots \mp \frac{1}{n^2},$$

$n > 0$ (n is an integer.)

- 3) Using the **mod** function, which calculates the modulo (the remainder of the integer division), write a function that can determine whether an integer a is prime or not.
(En utilisant la fonction **mod** qui calcule le modulo (le reste de la division entière), écrivez une fonction qui peut indiquer si un nombre entier **a** est premier ou pas.)

Exercice N° 06 :

Let the following two functions be: $f(x) = \sin(x - 2) + 4$

$$g(x) = -2x^3 + x^2 - 3$$

1. **Provide the necessary MATLAB instructions to plot the curve of the function $f(x)$ with a variation of x from 0 to 2π , and a step size of $\pi/12$.** (Donnez les instructions Matlab nécessaires pour tracer la courbe de la fonction $f(x)$ avec une variation de x de 0 jusqu'à 2π , et un pas = $\pi/12$.)
2. **Provide the necessary MATLAB instructions to plot the curve of the function $g(x)$ with a variation of x from -5 to 5, and a step size of 0.2.** (Donnez les instructions Matlab nécessaires pour tracer la courbe de la fonction $g(x)$ avec une variation de x de -5 jusqu'à 5, et un pas = 0.2.)
3. **Suggest two methods to draw the curves of these functions on the same figure.** (Proposez deux méthodes pour dessiner les courbes de ces fonctions dans la même figure.)
4. **How can you add a title to the figure and titles to both axes?** (Comment faire pour donner un titre pour la figure et un titre pour les deux axes ?)
5. **How can you plot the curve of $f(x)$ as a green dashed line with diamond-shaped markers? And how can you plot the curve of $g(x)$ as a blue dashed line with square-shaped markers?** (Comment faire pour dessiner la courbe de $f(x)$ en pointillé vert avec des points en forme de losanges ? et Comment faire pour dessiner la courbe de $g(x)$ en tirets bleus avec des points en forme de carrés ?)

Exercice N° 07 : (optional exercise)

- 1) Create the function defined by:
- 2) Then plot its curve in the interval:
[−4,4]

$$y = \begin{cases} -x^2 - 4x - 2, & \text{if } x < -1 \\ |x|, & \text{if } |x| \leq 1 \\ 2 - e^{\sqrt{x-1}}, & \text{if } x > 1 \end{cases}$$

Exercice N° 08 : (optional exercise.)

Write a MATLAB program in the form of a function called `determinant`, which calculates the determinant of a matrix using Cramer's method, defined by the recursive formula:

$$\det(A) = \begin{cases} a_{1,1} & \text{if } n = 1 \\ \sum_{j=1}^n \Delta_{i,j} a_{i,j} & \text{if } n > 1, \end{cases}$$

$\Delta_{i,j} = (-1)^{i+j} \det(A_{i,j})$ where $A_{i,j}$ is the matrix obtained by eliminating the i-th row and the j-th column of matrix A.

(Ecrivez un programme Matlab sous forme d'une fonction appelée `déterminant`, et qui calcule le déterminant d'une matrice par la méthode de Cramer définie par la formule récursive :

ou $\Delta_{i,j} = (-1)^{i+j} \det(A_{i,j})$ et $A_{i,j}$ est la matrice obtenue en éliminant la i-ème ligne et la j-ème colonne de la matrice A.)

Solution of TP N°03

Exercice 01:

```
>> C = (A>B) | ~(A==B) , D = A == B
```

C =

1

D =

0

A	B	(A>B)	(A==B)	~(A==B)	C
10	12	0	0	1	1
A==B	D				
0	0				

```
>> ~((A-B > A) & (A+B == 22)) , ans & B, ans | 0 ; ans & 0
```

ans =

1

ans =

1

ans =

0

(A-B > A)	(A+B == 22)	((A-B > A) & (A+B == 22))	~((A-B > A) & (A+B == 22))
0	1	0	1

ans	B	ans & B	ans	ans 0	ans	ans & 0
1	12	1	1	1	1	0

```
>> [4,2,-2:2:3] ~= 2*[2,1,-1,0,1]
```

ans =

0 0 0 0 0

[4, 2, -2 :2 :3] = [4,2,-2,0,2]
2*[2 , 1, -1,0,1] = [4,2,-2,0,2]

```
>> [-1 2 ; 4 6 ; 3 -7] >= [2 2 ; -1 8 ; 4 -2]
```

ans =

0 1
1 0
0 0

```
>> isequal(2*ones(3),2+zeros(3))
```

ans =

1

2*ones(3)	2+zeros(3)	=
1 1 1	0 0 0	2 2 2
2* 1 1 1	2+ 0 0 0	2 2 2
1 1 1	0 0 0	2 2 2

```
>> isempty([4:1;-3:-1:-1;[]])
```

ans =

1

Exercice 02:

1) Execution of the program:

For a = 16		For a = 5		For a = 169	
a = 16		a = 5		a = 169	
x = 8		x = 2.5000		x = 84.5000	
precision = 6		precision = 6		precision = 6	
i=1	x=(x+a/x)/2 x=(8+16/8)/2 x=5	i=1	x=(x+a/x)/2 x=(2.5000+5/2.5000)/2 x=2.2500	i=1	x=(x+a/x)/2 x=(84.5000+169/84.5000)/2 x=43.2500
i=2	x=(x+a/x)/2 x=(5+16/5)/2 x=4.1000	i=2	x=(x+a/x)/2 x=(2.2500+5/2.2500)/2 x=2.2361	i=2	x=(x+a/x)/2 x=(43.2500+169/43.2500)/2 x=23.5788
i=3	x=(x+a/x)/2 x=(4.1000+16/4.1000)/2 x=4.0012	i=3	x=(x+a/x)/2 x=(2.2361+5/2.2361)/2 x=2.2361	i=3	x=(x+a/x)/2 x=(23.5788+169/23.5788)/2 x=15.3731
i=4	x=(x+a/x)/2 x=(4.0012+16/4.0012)/2 x=4.0000	i=4	x=(x+a/x)/2 x=(2.2361+5/2.2361)/2 x=2.2361	i=4	x=(x+a/x)/2 x=(15.3731+169/15.3731)/2 x=13.1832
i=5	x=(x+a/x)/2 x=(4.0000+16/4.0000)/2 x=4.0000	i=5	x=(x+a/x)/2 x=(2.2361+5/2.2361)/2 x=2.2361	i=5	x=(x+a/x)/2 x=(13.1832+169/13.1832)/2 x=13.0013
i=6	x=(x+a/x)/2 x=(4.0000+16/4.0000)/2 x=4.0000	i=6	x=(x+a/x)/2 x=(2.2361+5/2.2361)/2 x=2.2361	i=6	x=(x+a/x)/2 x=(13.0013+169/13.0013)/2 x=13.0000
disp(4.0000)		disp(2.2361)		disp(13.0000)	

2) Replace the **if** statement with the **while** statement:

```

a = input('Entrez un nombre positif: ');
x = a/2;
precision=6;
i=1 ;
while i <= precision
    x = (x + a / x) / 2;
    i=i+1 ;
end
disp(x)

```

○ We need to create the variable *i* and ensure it is incremented in each iteration, because, unlike the **if** statement, the **while** statement does not automatically increment *i*.

We set the condition *i <= precision* to indicate the stopping criterion for the loop.

3) Modify the program so that it is applicable to vectors:

```
a = input('Entrez un vecteur de nombres positifs: ');
x = a / 2;
precision = 6;
for i = 1:precision
    x = (x + a ./ x) / 2;
end
disp(x)
```

The only thing to modify is the addition of element-wise operations instead of regular operations.
Instead of using `val1 / val2`, you use `val1 ./ val2`.

4) The variable **precision**, as its name suggests, determines the precision of the calculation. If we choose a value that is too small, the program may return a value for the square root that is close to the real root, but not close enough. We can observe this in the example where **a=169**, because if we stop after 3 iterations (**precision=3**), we get **x = 15.3731**, which is not exactly the true value. (Therefore, we need to allow enough iterations for the program to approach the root sufficiently). So, the larger the value of **precision**, the closer the result will be to the actual square root.

Exercice 03 :

1) Using this program, find the day of the week for one of the following dates:

01/11/1954 05/07/1962

11/09/2001 01/03/2012

The Matlab program	01/11/1954	05/07/1962	11/09/2001	01/03/2012
day = input('Enter the day : ');	day = 01	day = 05	day = 11	day = 01
month = input('Enter the month : ');	month = 11	month = 07	month = 09	month = 03
year = input('Enter the year: ');	year = 1954	year = 1962	year = 2001	year = 2012
if mod(year,4) == 0 DurationMonth=[0,3,4,0,2,5,0,3,6,1,4,6]; else DurationMonth=[0,3,3,6,1,4,6,2,5,0,3,5]; end	1954 = 488*4+2 mod(1954,4) = 2 ≠ 0 Then: DurationMonth = 0,3,3,6,1,4,6,2,5,0,3,5	1962 = 490*4+2 mod(1962,4) = 2 ≠ 0 Then: DurationMonth = 0,3,3,6,1,4,6,2,5,0,3,5	2001 = 500*4+1 mod(2001,4) = 1 ≠ 0 Then: DurationMonth = 0,3,3,6,1,4,6,2,5,0,3,5	2012 = 503*4 mod(2012,4) = 0 Then: DurationMonth = 0,3,4,0,2,5,0,3,6,1,4,6
daynbr = year - 1900;	daynbr = 1954-1900 = 54	daynbr = 1962-1900 = 62	daynbr = 2001-1900= 101	daynbr = 2012-1900= 112
daynbr = daynbr + floor((daynbr -1)/4);	54 + floor((54-1)/4) = 54 + floor(53/4) = 54 + 13 = 67	62 + floor((62-1)/4) = 62 + floor(61/4) = 62 + 15 = 77	101 + floor((101-1)/4) = 101 + floor(100/4) = 101 + 25 = 126	112 + floor((112-1)/4) = 112 + floor(111/4) = 112 + 27 = 139
daynbr = daynbr + DurationMonth(month) + day;	67 + DurationMonth (11) + 1 =67 + 3 + 1 = 71	77 + DurationMonth (7) + 5 =77 + 6 + 5 = 88	126 + DurationMonth (9) + 11 =126 + 5 + 11 = 142	139 + DurationMonth (3) + 1 =139 + 4 + 1 = 144
daynbr = mod(daynbr,7);	71 = 7*10+1 then: daynbr = mod(71,7) daynbr = 1	88 = 7*12+4 then: daynbr = mod(88,7) daynbr = 4	142 = 7*20+2 then: daynbr = mod(142,7) daynbr = 2	144 = 7*20+4 then: daynbr = mod(144,7) daynbr = 4
switch(daynbr) case 0 , disp('Sunday') case 1 , disp('Monday') case 2 , disp('Tuesday') case 3 , disp('Wednesday') case 4 , disp('Thursday') case 5 , disp('Friday') case 6 , disp('Saturday') end	Monday	Thursday	Tuesday	Thursday

- 2) Find the day of your birthday. (Depends on the student's choice)
- 3) Replace the **switch** statement with the **if** statement while keeping the functionality.

```
if (daynbr == 0)
    disp(' Sunday ')
elseif (daynbr == 1)
    disp(' Monday ')
elseif (daynbr == 2)
    disp(' Tuesday ')
elseif (daynbr == 3)
    disp(' Wednesday ')
elseif (daynbr == 4)
    disp(' Thursday ')
elseif (daynbr == 5)
    disp(' Friday ')
else
    disp(' Saturday ')
end
```

```
switch(daynbr)
case 0 , disp(' Sunday ')
case 1 , disp(' Monday ')
case 2 , disp(' Tuesday ')
case 3 , disp(' Wednesday ')
case 4 , disp(' Thursday ')
case 5 , disp(' Friday ')
case 6 , disp(' Saturday ')
end
```

- 4) Writing a program that checks the accuracy of a date (day/month/year):

```
day = input('Enter the day : ');
month = input('Enter the month : ');
year = input('Enter the year : ');

limitMonth = [31,28,31,30,31,30,31,31,30,31,30,31];

if mod(year,4)==0
    limitMonth(2)=29;
end

Validyear = year > 0;
Validmonth = (month > 0) & (month <= 12);
Validday = (day > 0) & (day <= limitMonth(month));

if (Validday & Validmonth & Validyear)
    disp(' Valid Date ')
else
    disp(' Incorrect Date ')
end
```

The limites the months of the year

Check if a date is a leap year. If so, then month 2 (February) will have 29 days.

Exercise 04:

For a positive integer N ($N > 0$):

$$U_0 = N$$

$$U_{n+1} = \begin{cases} \frac{U_n}{2} & \text{if } U_n \text{ is even} \\ 3U_n + 1 & \text{if } U_n \text{ is odd} \end{cases}$$

- 1) Find the Collatz sequence for the numbers: 5 and 3.

For $N = 5$

$U_0 = N = 5$	$U_1 = 3 \cdot 5 + 1$	$U_2 = 16/2$	$U_3 = 8/2$	$U_4 = 4/2$	$U_5 = 2/2$	...	...	...	U_∞
5	16	8	4	2	1	...	4	2	1

Convergence of the sequence to the sequence 4,2,1, so we stop here after 5 iterations.

For $N = 3$

$U_0 = N = 3$	$U_1 = 3 \cdot 3 + 1$	$U_2 = 10/2$	$U_3 = 3 \cdot 5 + 1$	$U_4 = 16/2$	$U_5 = 8/2$	$U_6 = 4/2$	$U_7 = 2/2$
3	10	5	16	8	4	2	1

- 2) Write a program that generates the Collatz sequence for a given number N :

```
% Conjecture de Collatz
n = input('Enter a positive integer: ');

while n ~= 1
    if ceil(n/2) == n/2 % Test if n is even
        n = n/2;
    else
        n = 3*n+1;
    end
    disp(n)
end
```

- 4) Transform the program into a function:

This function takes a number N as input and returns a vector V containing the Collatz sequence as output.

Exercise 05 :

- 1) The program that calculates the factorial of n:

```
n = input('Enter a positive integer: ');
fact = 1 ;
for i = 1:n
    fact = fact*i ;
end
fact % write the value of fact
```

- 2) The program that calculates the sum $\sum_{k=1}^n \frac{1}{k}$:

```
n = input('Enter a positif integer : ');
H = 0 ;
for k = 1:n
    H = H+1/k ;
end
disp(H)
```

We could have written
`H = sum(1./[1:n]) ;`

- 3) The program that calculates the sum $\sum_{k=1}^n \frac{(-1)^k}{k^2}$:

```
n = input('Enter a positive integer:'); H = 0
;
for k = 1:n
    H = H+(-1)^k/k^2 ;
end
disp(H)
```

- 4) The program that indicates whether a number is prime or not:

```
function P = isOdd(a)
P = 1;
i=2;
while i<=sqrt(a) & P
    if mod(a,i) == 0
        P = 0;
    end
    i=i+1;
end
```

The expression

(while i<=sqrt(a) & P)

Can be replaced by:

(while i < a & P)

It becomes more readable but less efficient.

The program consists of dividing the number **n** by all numbers smaller than it (or by its square root for efficiency). If it finds a divisor (remainder of the division = 0), then the number is not prime, and the loop is stopped.

Otherwise, if no divisor is found for n, it will be considered prime.

Exercice 06:

- 1) Plot the graph of the function $f(x)$:

```
>> x = 0:pi/12:2*pi;
>> f = sin(x-2)+4;
>> plot(x,f)
```

- 2) Plot the graph of the function $g(x)$:

```
>> x = -5:0.2:5;
>> g = -2*x.^3+x.^2-3;
>> plot(x,g)
```

- 3) To plot both curves at the same time, you can do:

- Use the command **hold on** as follows:

```
>> x1 = 0:pi/12:2*pi;
>> f = sin(x1-2)+4;
>> plot(x1,f)
>> hold on
>> x2 = -5:0.2:5;
>> g = -2*x2.^3+x2.^2-3;
>> plot(x2,g)
>> hold off
```

- Use four arguments with the **plot** function as follows:

```
>> x1 = 0:pi/12:2*pi;
>> f = sin(x1-2)+4;
>> x2 = -5:0.2:5;
>> g = -2*x2.^3+x2.^2-3;
>> plot(x1,f,x2,g)
```

- 4) To add titles, simply write:

```
>> title(' The title of the figure ')
>> xlabel(' The title of the x-axis ')
>> ylabel(' The title of the y-axis ')
```

- 5) Pour dessiner la courbe de $f(x)$ en pointillé vert avec des points en forme de losanges on écrit :

```
>> x = 0:pi/12:2*pi;
>> f = sin(x-2)+4;
>> plot(x, f, 'gd')
```

- 6) Pour dessiner la courbe de $g(x)$ en tirets bleus avec des points en forme de carrés on écrit :

```
>> x = -5:0.2:5;
>> g = -2*x.^3+x.^2-3;
>> plot(x, g, 'b--s')
```

Exercise 07 :

- 1) Creation of the function defined by: $y = \begin{cases} -x^2 - 4x - 2, & \text{si } x < -1 \\ |x|, & \text{si } |x| \leq 1 \\ 2 - e^{\sqrt{x-1}}, & \text{si } x > 1 \end{cases}$

```
function y = f(x)
    if (x < -1)
        y = -x^2-4*x-2;
    elseif (x > 1)
        y = 2-exp(sqrt(x-1));
    else
        y = abs(x);
    end
end
```

- 1) Plot its curve in the interval:]-4, 4[

```
>> x = -4:0.2:4;
```

Création des abscisses

```
>> N = length(x);
```

```
>> for i=1:N
```

```
    y(i)=f(x(i));
```

Creation of the ordinates

```
end
```

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

Plot the function

Exercise 07 :

The function that calculates the determinant using Cramer's recursive method is:

```
function d = determinant(A)
n = size(A,1); % The nombre of Line/colomns
if n == 1
    d = A(1,1);
else
    i = 1;
    d = 0;
    for j=1:n
        c=(-1)^(i+j);
        sousMat=A;
        sousMat(i,:)=[];
        sousMat(:,j)=[];
        sousDet=determinant(sousMat);
        d=d+A(i,j)*c*sousDet;
    end
end
end
```

If n=1, then the determinant is $a_{1,1}$

Calculate $(-1)^{i+j}$

sousMat is the matrix A without the i-th row and la j-th column

Calculate the determinant sub matrix sousMat

Calculate the sum of the determinants of the submatrices according to the formula

Command Catalogue:

Nom	Fonction	Description ou Commentaire
General Functions (Fonctions générales)		
Aide	Help	Matlab Help (Aide de Matlab)
Démonstrations	demo	List of demonstrations available in Matlab regarding its functionality. (Liste des démonstrations disponible dans Matlab quant à son fonctionnement)
Variables	who whos	Name the variables and provide the size of the memory space occupied. (Nomme les variables ainsi que donne la grandeur de l'espace mémoire occupé)
Recherche	lookfor	Find all functions containing the searched word. (Trouver toutes les fonctions contenant le mot recherché)
Dimensions matrices	size	Dimensions of the matrix. (Dimensions de la matrice)
Clear	clear	Clear the current workspace. (Efface l'espace de travail courant)
Quitter	quit, exit	Exit Matlab. (Met fin à Matlab)
Simple calculations (Calculs simples)		
Addition	-	Addition
Subtraction	+	Soustraction
Multiplication	*	Multiplication
Division	/	Division
Powers	$\wedge$	Puissance
Powers of e	exp(x)	Puissance de e
Natural Logarithm	log(x)	Logarithme naturel (ln)
Logarithm to base 2	log2(x)	Logarithme en base 2
Logarithm to base 10	log10(x)	Logarithme en base 10 (commun)
Square root	sqrt(x)	Racine carrée
Roots	roots(x)	Produit une matrice contenant les racines de x
Sinus	sin(x)	Sinus
Inverse of sinus	asin(x)	Inverse du sinus
Secant	sec(x)	Sécante
Hyperbolic sine	sinh(x)	Sinus hyperbolique
Inverse of hyperbolic Sine	asinh(x)	Inverse du sinus hyperbolique
Cosinus	cos(x)	Cosinus
Inverse of cosinus	acos(x)	Inverse du cosinus
Cosecant	csc(x)	Cosécante
Hyperbolic cosine	cosh(x)	cosinus hyperbolique
Inverse of hyperbolic cosine	acosh(x)	Inverse du cosinus hyperbolique
Tangent	tan(x)	Tangente
Cotangent	cot(x)	Cotangente

Command Catalogue:

Hyperbolic Tangent	tanh(x)	Tangente hyperbolique
Inverse tangent	atan(x)	Inverse de tangente
Inverse of tangent	atan2(x)	Inverse tangent in the 4th quadrant. (Inverse de tangente dans le 4e quadrant) ($-pi \leq ATAN2(Y,X) \leq pi$)
π	pi	The number pi (le nombre pi)
Infinit	inf	Infinit

Command Catalogue:

Nom	Fonction	Description ou Commentaire
Matrices (Matrices)		
Addition	$a + b$	Element-wise addition (Addition élément par élément)
Soustraction	$a - b$	Element-wise division (Division élément par élément)
Multiplication matricielle	$a * b$	Standard matrix multiplication where: (Multiplication matricielle standard où) $c(i, j) = \sum_{k=1}^n a(i, k)b(k, j)$
Multiplication de tableau	$a .* b$	Element-wise multiplication (Multiplication élément par élément)
Division matricielle de droite	a / b	Right matrix division defined by $a * \text{inv}(b)$ where $\text{inv}(b)$ is the inverse of b . (Division matricielle de droite définie par $a * \text{inv}(b)$ où $\text{inv}(b)$ est l'inverse de b)
Division de tableau de droite	$a ./ b$	Element-wise division of a by b. (Division de a par b élément par élément)
Division de matricielle de gauche	$a \backslash b$	Left matrix division defined by $\text{inv}(a) * b$ where $\text{inv}(a)$ is the inverse of a. (Division matricielle de gauche définie par $\text{inv}(a) * b$ où $\text{inv}(a)$ est l'inverse de a)
Division de tableau de gauche	$a .\backslash b$	Element-wise division of b by a. (Division de b par a élément par élément)
Puissance	$a .\wedge b$	Element-wise exponential of a to the power of b. (a exponentiel de b, élément par élément)
Identité	$\text{eye}(m)$	Produce an identity matrix of size $m \times m$. (Produit une matrice identité $m \times m$)
Zeros	$\text{zeros}(m,n)$	Produce an $m \times n$ matrix where all the entries are 0. (Produit une matrice $m \times n$ où toutes les données sont 0)
Un	$\text{ones}(m,n)$	Produce an $m \times n$ matrix where all the entries are 1. (Produit une matrice $m \times n$ où toutes les données sont 1)
Hasard	$\text{rand}(m,n)$	Produce an $m \times n$ matrix where all the entries are randomly selected. (Produit une matrice $m \times n$ où toutes les données sont choisies au hasard)
Déterminant	$\text{det}(A)$	Give the determinant of the matrix. (Donne le déterminant de la matrice)
Transposé	A'	Transpose the matrix. (Transposé la matrice)
Réduction	$\text{rref}(A)$	Reduce the matrix using the Gauss-Jordan method (Be careful with singular matrices). (Réduit la matrice par la méthode de Gauss-Jordan (Attention aux matrices singulières))
Inverse	$\text{int}(A)$	Inverse of matrix A (Be careful with singular matrices). (Inverse de la matrice A (Attention aux matrices singulières))

Command Catalogue:

Somme	sum(A)	Sum of the columns of matrix A. (Somme des colonnes de la matrice A)
Rang	rank(A)	Evaluate the number of linearly independent rows and columns. (Évalue le nombre de rangée et de colonnes linéairement indépendantes)
Norme	norm(A)	Calculate the norm of the matrix. (Produit la norme de la matrice)
Diagonalisation	diag(A)	Diagonalize the matrix. (Diagonalise la matrice)
Matrice triangulaire inférieure	tril(A)	Allows obtaining the lower triangular part of a matrix. (Permet d'obtenir la partie triangulaire inférieure d'une matrice)
Matrice triangulaire supérieure	triu(A)	Allows obtaining the upper triangular part of a matrix. (Permet d'obtenir la partie triangulaire supérieure d'une matrice)
Logarithme d'une matrice	logm(A)	Calculate the logarithm of matrix A. (Effectue le logarithme de la matrice A)
Exponentiel d'une matrice	expm(A)	Express the matrix as a power of the constant e. (Met la matrice comme puissance de la constante e)
Racine carrée d'une matrice	sqrtn(A)	Calculate the square root of the matrix. (Effectue la racine carrée de la matrice)
Fonction d'une matrice	funm(A, fonction)	Apply the function to the matrix. (Effectue la fonction de la matrice)
Valeurs propres	eig(A)	Return the eigenvalues of the square matrix A. (Renvoie les valeurs propres (eigenvalues) de la matrice carrée A)
Coefficient polynome caractéristique	poly(A)	Return the coefficients of the characteristic polynomial associated with matrix A. (Renvoie les coefficients du polynôme caractéristique associé à la matrice A)

Command Catalogue:

Nom	Fonction	Description ou Commentaire
Graphics (Graphiques)		
Tracé d'une courbe	plot(x,y)	Allows plotting a curve according to the matrix X by Y. (Permet de tracer une courbe selon la matrice X par Y)
Tracé d'une courbe	fplot(fon, lim)	Allows plotting a curve according to the function fon with axis limits lim. (Permet de tracer une courbe selon la fonction fon avec les limites d'axes lim)
Identification axe X	xlabel(texte)	Create a label for the X-axis. (Produit une étiquette de l'axe des X)
Identification axe Y	ylabel(texte)	Create a label for the Y-axis. (Produit une étiquette de l'axe des Y)
Titre	title(texte)	Create a title for the graph. (Produit un titre au graphique)
Légende	legend (fon1,fon2,pos)	Add a legend where fon1, fon2, etc., represent the functions and specify the position where the legend is placed in the graph. (Met une légende où fon1, fon2, . . .représentes les fonctions et pos la position où la légende se place dans le graphique)
Grille	grid	Grid the graph according to the chosen scale. (Quadrille le graphique selon l'échelle choisie)
Texte positionné d'avance	text(x,y,'mots')	Position the text 'mots' at coordinates x and y. (Positionne en x et y le texte 'mots')
Texte positionné manuellement	gtext('mots')	Manually position the text 'mots' while the graph is being executed. (Positionne le texte 'mots' manuellement lors de l'exécution du graphique)
Axes	axis	Several axis options are available. (Plusieurs options d'axes sont offertes)
Tracés superposés	hold on ; hold off	Allows overlaying graphs with hold on, and then stops the overlay with hold off. (Permet de superposer les graphiques avec hold on, puis arrête la superposition avec hold off)
Tracés côte à côte	subplot(m,n,p)	Place multiple graphs on the same figure. (Place sur une même figure plusieurs graphiques)
Échelle logarithmique	loglog(x,y)	Plot a graph with a logarithmic scale. (Trace un graphique avec une échelle logarithmique)
Échelle semi-logarithmique en X	semilogx(x,y)	Plot a graph with a semi-logarithmic scale on the x-axis. (Trace un graphique avec une échelle semi-logarithmique en x)
Échelle semi-logarithmique en Y	semilogy(x,y)	Plot a graph with a semi-logarithmic scale on the y-axis. (Trace un graphique avec une échelle semi-logarithmique en y)

Command Catalogue:

Échelle polaire	polar(theta,r)	Plot a graph with a polar scale. (Trace un graphique avec une échelle polaire)
Barres	bar(x,y)	Plot a graph with bars. (Trace un graphique avec des barres)
Escalier	stairs(x,y)	Plot a stair-step graph. (Trace un graphique en escalier)
Complex numbers (Nombres complexes)		
Nombre complexe	$a+bi$ $a+b*j$	Define a complex number $a + bi$. (Définit un nombre complexe $a+bi$)
Fonction complexe	complex(a,b)	Produce a complex number $a + bi$. (Produit un nombre complexe $a+bi$)
Partie réelle	real(x)	Return the real part of a number. (Renvoie la partie réelle d'un nombre)
Partie imaginaire	imag(x)	Give the imaginary part of the number x . (Donne la partie imaginaire du nombre x)

Semestre : 03

Unité d'enseignement: Méthodologique

Matière : Outils de Programmation 2

Crédits : 3

Coefficient : 1

Objectifs de l'enseignement : Donner aux étudiants les éléments fondamentaux pour la maîtrise d'outils de programmation en s'appuyant sur des langages à usage scientifique et technique.

Connaissances préalables recommandées : Algorithmique, structures de données et langages de programmation.

Contenu de la matière :

Chapitre 1: Prise en Main

Démarrage et aide variable - Variables - Répertoire de travail - Sauvegarde de l'environnement du travail - Fonctions et commandes.

Chapitre 2: Les nombre en Matlab avec licence ou Scilab

Entiers naturels - Représentation des réelles - Nombres complexe.

Chapitre 3: Vecteurs et Matrices

Opérations sur les vecteurs et les Matrices - Fonctions mathématiques élémentaires.

Chapitre 4: Eléments de programmation

Script - Fonction - Boucle de contrôle - Instruction conditionnelle.

Chapitre 5: Polynômes

Polynômes en Matlab avec licence ou Scilab - Zéros d'un polynôme - Opérations sur les polynômes.

Chapitre 6: Graphisme en Matlab avec licence ou Scilab

Affichage des courbes en dimension deux et dimension trois - Graphe d'une fonction – Surface Analytique.

Chapitre 7 : Calcul symbolique

Appel de la toolbox symbolique - Développement et mise en fonction d'une expression - Dérivée et primitive d'une fonction - Calcul du développement limité d'une fonction.

Mode d'évaluation : Examen (60%), contrôle continu (40%)

Références

- Calcul scientifique avec Matlab, Jonas-Koko, Ellipses.

- Introduction au Matlab, J. T. Lapresté, Ellipses.

Semester: 03

Teaching Unit: Methodological

Course: Programming Tools 2

Credits: 3

Coefficient: 1

Course Objectives:

To provide students with the fundamental elements for mastering programming tools, focusing on languages used for scientific and technical purposes.

Recommended Prerequisites:

Algorithmics, data structures, and programming languages.

Course Content:

Chapter 1: Getting Started

- Startup and variable help
- Variables
- Working directory
- Saving the work environment
- Functions and commands

Chapter 2: Numbers in MATLAB or Scilab

- Natural integers
- Real number representation
- Complex numbers

Chapter 3: Vectors and Matrices

- Operations on vectors and matrices
- Basic mathematical functions

Chapter 4: Programming Elements

- Script
- Function
- Control loops
- Conditional statements

Chapter 5: Polynomials

- Polynomials in MATLAB or Scilab

- Polynomial zeros
- Polynomial operations

Chapter 6: Graphics in MATLAB or Scilab

- 2D and 3D curve plotting
- Graph of a function
- Analytical surfaces

Chapter 7: Symbolic Calculation

- Calling the symbolic toolbox
- Expanding and transforming expressions
- Derivatives and integrals of a function
- Taylor series expansion of a function

Evaluation Method:

- Exam (60%)
- Continuous assessment (40%)

References:

- *Calcul scientifique avec MATLAB*, Jonas-Koko, Ellipses.
- *Introduction to MATLAB*, J. T. Lapresté, Ellipses.

Bibliography

English Books:

1. **"MATLAB for Engineers"** by Holly Moore
 - A practical approach for learning MATLAB, focused on applications relevant to engineering and mathematicians.
 2. **"MATLAB: A Practical Introduction to Programming and Problem Solving"** by Stormy Attaway
 - A beginner-friendly book that introduces MATLAB programming along with problem-solving techniques.
 3. **"MATLAB for Data Analysis"** by Victor Cheng
 - Focuses on using MATLAB for data analysis, visualization, and numerical computing for mathematicians.
 4. **"MATLAB for Dummies"** by Jim Sizemore and John Paul Mueller
 - An accessible, beginner-friendly guide to learning MATLAB, covering everything from the basics to advanced topics.
 5. **"MATLAB: An Introduction with Applications"** by Amos Gilat
 - This book offers a comprehensive introduction to MATLAB, with applications in engineering, science, mathematicians and economics.
 6. **"MATLAB for Engineers and Scientists"** by R. S. S. P. Vishnu
 - A practical guide for engineers and scientists with MATLAB-based solutions for real-world problems.
 7. **"MATLAB for Machine Learning"** by Jason Boyd
 - A specialized book that uses MATLAB to apply machine learning algorithms and concepts.
-

French Books:

1. **"MATLAB: Guide de l'utilisateur"** by Brian Hahn and Daniel T. Valentine
 - A comprehensive user guide that covers the basics and advanced features of MATLAB in French.
2. **"Introduction à MATLAB"** by Eric Lefebvre and Alain Gosselin
 - This book provides an introduction to MATLAB, with practical exercises and applications.
3. **"MATLAB pour les sciences et ingénierie"** by Jean-Pierre Bourguet
 - A practical book focusing on MATLAB for solving problems in science and engineering.
4. **"Programmation MATLAB pour les ingénieurs et scientifiques"** by Olivier Sauter
 - A French book aimed at engineers and scientists who want to learn how to use MATLAB for their computations and simulations.
5. **"Applications MATLAB pour les sciences et l'ingénierie"** by R. I. Finkel
 - This book offers practical applications and examples using MATLAB in various scientific and engineering contexts.