Ministry of Higher Education and Scientific Research
Djilali Bounaâma University, Khemis Miliana
Faculty of Matter Sciences and Computer Science
Mathematics Department

PROGRAMMING TOOLS 2

Course and Laboratory Works

L2 Mathematics

By

Dr. Abdesselam KALI

November , 2025

Scanné avec CamScanner

Preface

This module, *Programming Tools 2*, introduces essential computational skills with **Scilab** that are increasingly vital for mathematics students.

This textbook reflects six years of teaching experience in the Department of Mathematics at Djilali Bounaâma University, Khemis-Miliana, and is designed for second-year undergraduate students majoring in Mathematics. The goal is not to make students specialists in programming, but to show how Scilab can be used effectively to perform numerical calculations, manipulate vectors and matrices, solve equations, analyze functions, and visualize mathematical concepts efficiently. By integrating programming with mathematical reasoning, students can deepen their understanding, explore complex problems, and develop practical skills that are indispensable in both academic research and real-world applications. Each chapter includes exercises with detailed solutions to support classroom learning and laboratory works.

The material follows the official syllabus outlined in the most recent Canevas for the **2018–2019** academic year. Programming techniques are introduced step by step, with emphasis on two main aspects: mastering commands and practicing them to solve exercises. The presentation is limited to what is necessary, ensuring clarity without oversimplification.

The primary bibliographic source used in preparing this textbook is the help documentation of the latest version of Scilab (Scilab-2025.1.0), available at [https://help.scilab.org/docs/2025.1.0/en_US/]. Other works cited in the bibliography have also been used as supporting material to enrich and complement the content presented here.

Scanné avec CamScanner

Students are expected to have a basic background in algorithms, data structures, and programming languages to fully benefit from this course. The recommended weekly schedule includes **1** hour and **30** minutes of lectures and **1** hour and **30** minutes of laboratory work. Assessment consists of continuous evaluation and a final exam. The module carries a coefficient of **1** and is worth **3** academic credits.

We would like to express our sincere gratitude to everyone who contributed to the preparation of this work. Our special thanks go to Dr. **Sakri Redha** Associate Professor (Class A), Department of Urban Hydraulics, National Higher School of Hydraulics (ENSH), and to Dr. **Bensaid Chaima** Associate Professor (Class A), Computer Science Department, Faculty of Matter Sciences and Computer Science, Djilali Bounaâma University, Khemis-Miliana, for their careful review and valuable suggestions. We are also thankful to our colleagues with whom we have shared many years of teaching mathematics in the Department of Mathematics and Computer Science.

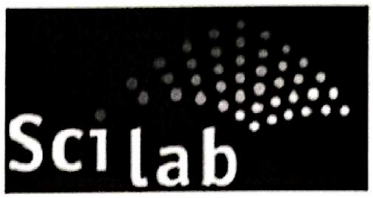
Scanné avec CamScanner

Module Content

Scanné avec CamScanner

Scanné avec CamScanner

Chapter 1. Getting Started

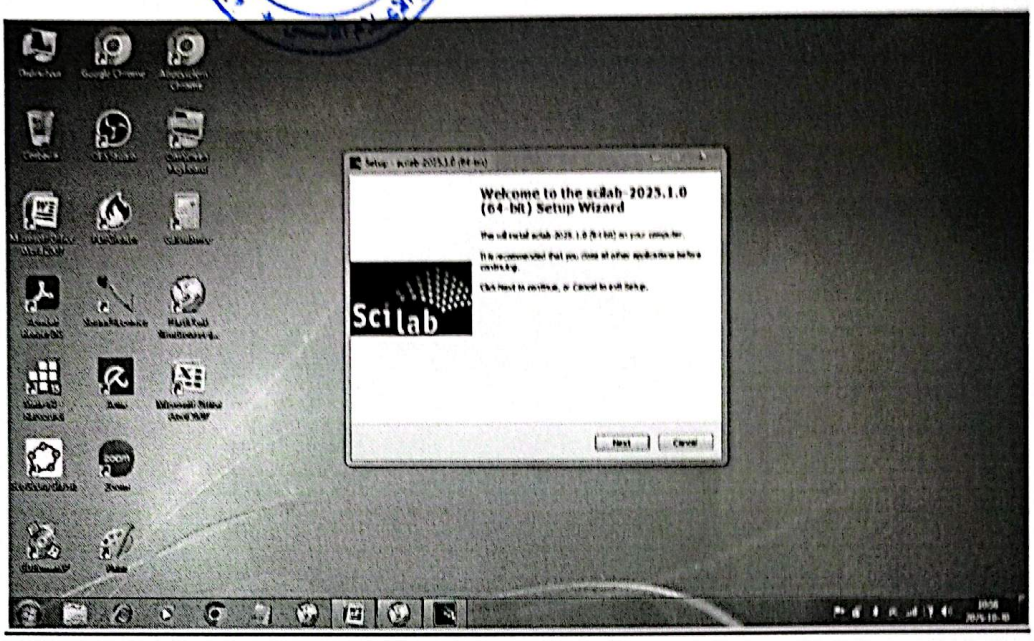


Scilab (a contraction of **Sci**entific **Lab**oratory) is open-source software available for download from the official webpage https://scilab.org. Scilab is supported on UNIX, Macintosh, and Windows environments. It includes libraries of a large number of built-in mathematical functions and a wide range of packages of pre-written programs, called *toolboxes*, which are managed through **ATOMS** (**A**utomated **To**olbox **M**anagement **S**ystem). Scilab also works as a programming language like C, C++ etc, and the built-in functions can be used into the users' programs. In addition, it has a variety of graphical capabilities. This feature of Scilab makes it user friendly interactive software. In Scilab, all variable types; including integer, real, complex, Boolean, string, polynomial, are treated as matrices. This chapter introduces the basic concepts needed to begin working with Scilab. It explains how to start the program, manage variables, set the working directory, save the work environment, and use essential functions and commands.

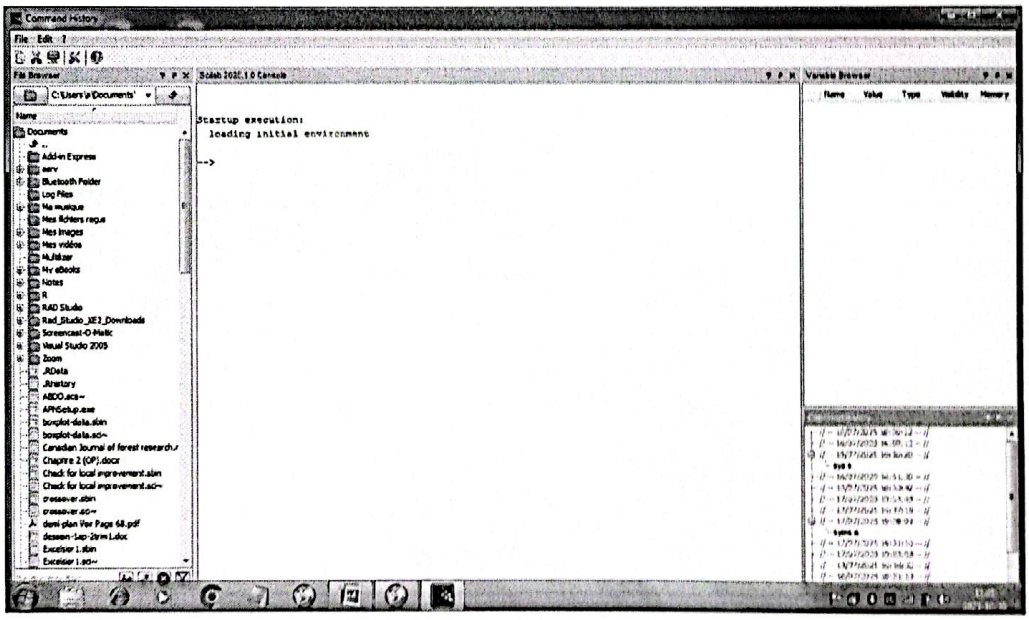
Scanné avec CamScanner

1. Startup and variable help:

The installation of Scilab is easy:



Once installed on your computer system, Scilab is ready to use. You can open Scilab by double-clicking the Scilab icon on your desktop or by clicking the icon that appears after typing "Scilab" in the Windows search bar. The Scilab window should then appear on your screen, looking like this:



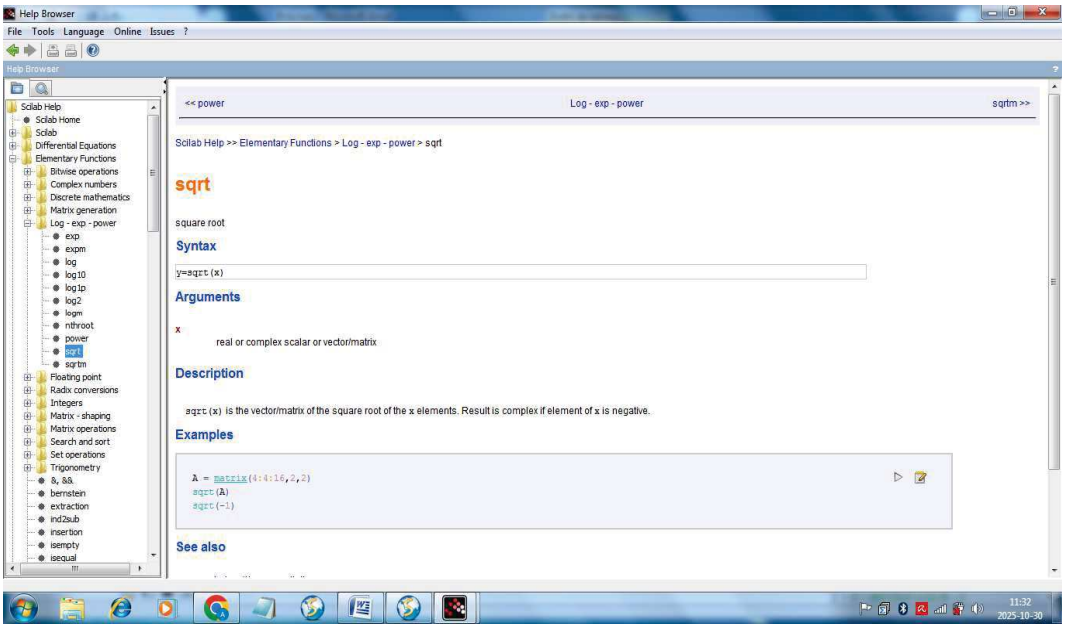
Scanné avec CamScanner

The window on the left, the *File Browser*, allows you to locate your files. In the center, the *Console*, is the area where you type commands and where the results are displayed. This is the part you will use most often. In this area, the prompt, appearing as **-->**, indicates that Scilab is waiting for a command. At the top right, the *Variable Browser* there is a summary of all the variables used, along with their properties. Below that is the *Command History*. If you need to find a command you've already entered, there's no need to retype it; it has been stored in memory and is available in this frame.

Scilab has an integrated *help* system that allows you to obtain information about commands and functions. For example, to get help on the **sqrt** (square root) command, simply type:

```
--> doc sqrt
```

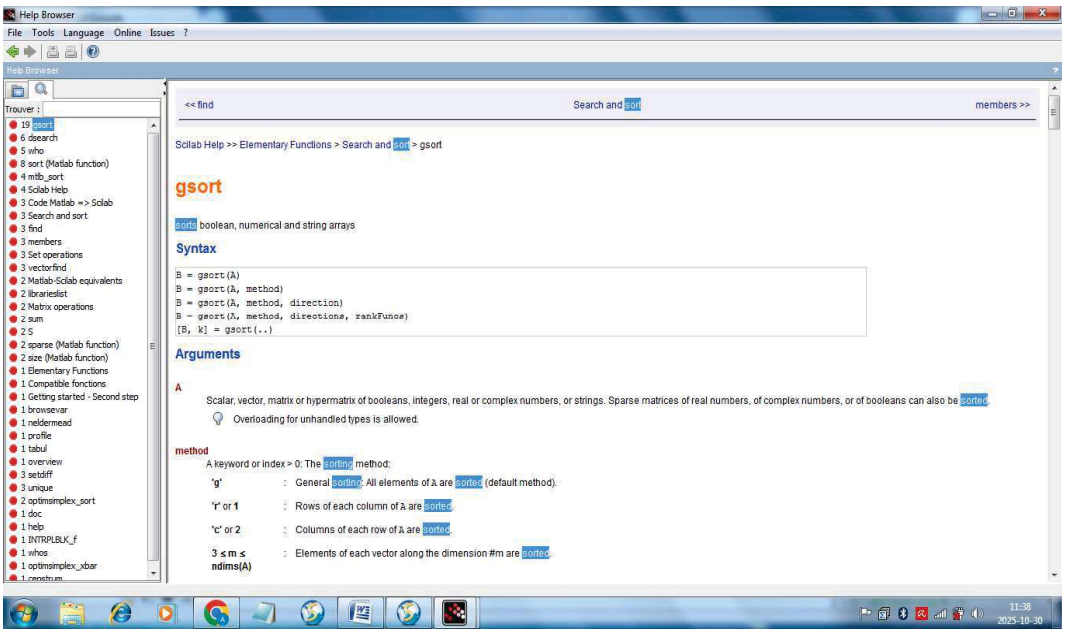
The help window should then appear on your screen, looking like this:



The Scilab help pages provide a detailed description of the **sqrt** command, its syntax, usage examples, and links to similar or related commands. When searching for a command to perform a particular task without knowing its name, you can use the **apropos** command, which searches the help pages for text containing a specific string of characters. For example, if you want to sort a vector or matrix, you would type:

```
--> apropos sort
```

The help browser will then suggest the **gsort** command, which is an excellent suggestion:



In general, you can also simply type **doc** and navigate through the table of contents.

3. Variables:

Scilab allows you to create variables to store data. Use the equal sign **=** to assign a value to a variable. In this example:

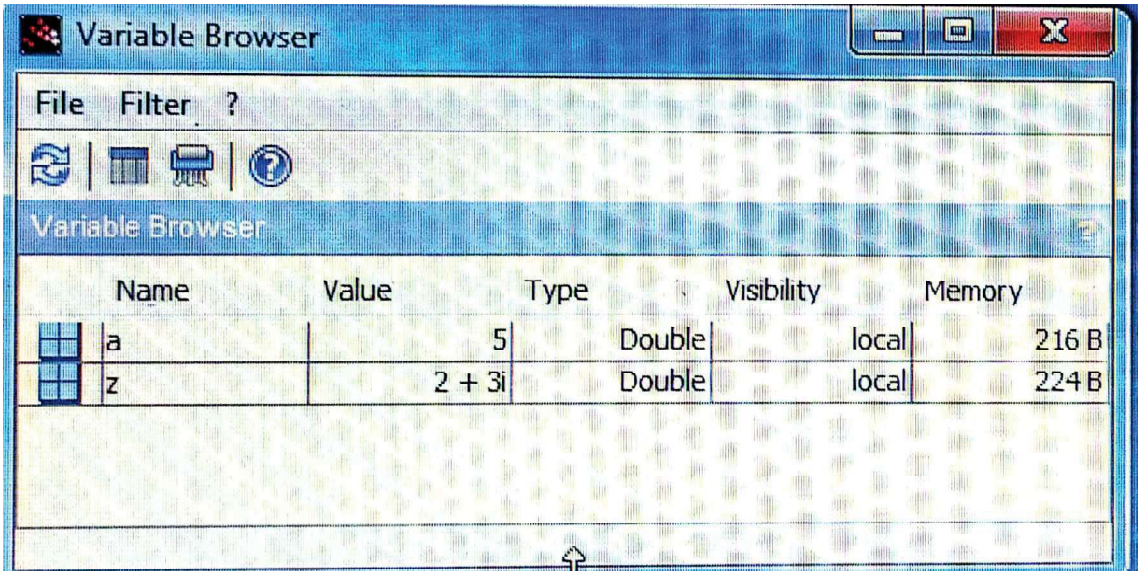
```
-->a = 5
```

We have created a variable **a** to which we have assigned the value **5**.
We can also create complex variables by using the **%i** notation for the imaginary part. In the example:

```
-->z = 2 + 3*%i
```

The variable **z** now contains the complex number **2 + 3i**.

The properties of the variables **a** and **z** are summarized in the *Variable Browser* window below.



For example, in Scilab, the complex variable **z** has the **type "double"**, meaning it stores numerical values in double-precision format. Its **visibility** is **local**, which means it is only accessible within the function or workspace where it was created, and its **memory usage** is **224 B**.

4. Working directory:

The working directory is the folder on your computer where Scilab looks for files by default. You can find out the current working directory using the **pwd** (**P**rint **W**orking **D**irectory) command. This will display the path of the current working directory.

You can use the **cd** (**C**hange **D**irectory) command to change the working directory. For example, to change the working directory to "Documents", type:

```
--> cd("C:\...\Documents")
--> pwd()
```

This will change the working directory to the "Documents" folder and display the new path.

5. Saving the workspace:

To exit Scilab, simply type **quit** or **exit**. It is possible to save the workspace before exiting by using the **save** command. To load a previously saved workspace, use the **load** command. For example:

```
--> a = 1, b = -9
--> save('vals.sod', 'a', 'b')
--> quit
--> load('vals.sod', 'a', 'b')
--> a, b
```

6. Functions and commands:

Functions are essential in Scilab for performing mathematical operations and returns a result. Scilab offers many built-in functions. You usually call a function with parentheses ( ) and sometimes with input arguments. Examples:

```
--> date()
--> x = -10
--> abs_x = abs(x)
--> disp(abs_x)
```

date(): shows the current date and time.

abs(x): gives the absolute value of a number.

`disp(abs_x)`: displays a value or message on the screen. It shows the result stored in **abs_x**.

A *command* is an instruction that tells Scilab to do something, but it may not return a value. Examples:

```
--> clear
--> clc
--> quit
```

`clear`: this command removes all variables from the workspace (memory).

`clc`: this command clears the console window, it removes all text shown on the screen but keeps the variables in memory.

`quit:` this command closes Scilab completely.

Laboratory Work 1

1. Starting Scilab:

a) Download and install the latest version of Scilab from the official website.

b) Launch Scilab on your computer.

c) Take a screenshot of the main Scilab window and add it to your report.

d) Use the **`doc`** command to get information on the following commands: **`who`**, **`who_user`**, **`whos`**.

e) Capture the screen displaying the detailed description of the **`whos`** command in the help window.

f) Use the **`apropos`** command to search for the command related to creating a complex number.

g) Capture the screen showing the command suggested by Scilab.

2. Creating variables:

a) Open Scilab.

b) Create a variable **`pi`** and assign it the value of π (3.14159265359) using **`%pi`**.

c) Create a second variable **`radius`** and assign it a radius of your choice.

d) Use these two variables to calculate the circumference of a circle.

e) Display the result in the console.

f) Create a variable **`z1`** containing a complex number of your choice.

g) Create a second variable **`z2`** containing another complex number of your choice.

h) Perform a complex arithmetic operation using these two variables.

i) Display the result in the console.

3. Working directory:

a) Open Scilab.

b) Use the **`pwd`** command to display the current working directory.

c) Create a new folder on your computer.

d) Use the **`cd`** command to change the working directory to the new folder you just created.

e) Use the **`pwd`** command again to confirm that you are now in the new directory.

f) Capture the screen showing the path of the new working directory.

4. Saving the workspace:

a) Open Scilab.

b) Create three variables with values of your choice.

c) Use the **`save`** command to save these variables in a file with a custom name.

d) Use the **`quit`** command to exit Scilab.

e) Open Scilab again, **`load`** only the first two variables from the file, and display their values.

f) Ensure that the third variable has not been loaded.

5. Functions and Commands:

1. Open Scilab.
2. Use the **`complex`** function to create a complex number of your choice.
3. Display the result in the console.
4. Describe the functionality of each of the following commands: **`edit`**, **`exec`**, and **`abort`**.

Solution of Laboratory Work 1

2. Creating Variables:

```
--> pi = %pi;
--> radius = 5;
--> circumference = 2 * pi * radius;
--> disp("circumference = ", circumference);
"circumference = "
   31.415927
```

```
--> z1 = 2 + 3 * %i;

--> z2 = 1 - 2 * %i;

--> result = z1 + z2;

--> disp("result",result);

"result"

   3. + i
```

3. Working Directory:

```
--> pwd();
--> cd("C:\Users\a\Desktop\Scilab");
--> pwd();
```

4. Saving the Workspace:

```
--> a = 10;
--> b = 20;
--> c = 30;
--> save('my_variables.sod', 'a', 'b','c');
--> quit;
--> load('my_variables.sod', 'a', 'b');
--> disp(a,b);
--> disp(c);
```

5. Functions and Commands:

```
--> Z=complex(1,2);
--> disp(Z, "Z = ");
```

edit: Opens the Scilab editor, allowing you to create or modify script (**.sce**) or function (**.sci**) files.

exec: Executes a Scilab script file, running the commands or functions written inside it.

abort: Immediately stops the execution of a running program or script.

Chapter 2. Numbers in Scilab

1. Natural integers:

An integer in Scilab always contains the **"."** character, even if the decimal point is optional during input. Example:

```
-->  n = 5

  n =

        5.
```

2. Representation of real numbers:

The basic objects in the Scilab language are real numbers, which are stored as double-precision floating-point numbers following the **IEEE 754** standard. Implementation of IEEE Standard for Floating-Point Arithmetic can be found at https://en.wikipedia.org/wiki/IEEE_754. In this format, each number occupies **64** bits: **1** bit for the sign (positive or negative), **11** bits for the exponent (which sets the scale), and **52** bits for the mantissa (the significant digits). Although up to **18** digits may appear on the screen, only the first **15–16** digits are truly accurate. Scilab displays real numbers in decimal notation, using an optional decimal point "**.**" and the signs "**+**" or "**–**". In scientific notation, Scilab uses the letter "**D**" to indicate the power-of-ten scaling factor. For example, **-123456.789123456789** is represented as **-1.23456789123456789D+5**, where the mantissa is **1.23456789123456789**, the sign is negative, **D+5** indicates the exponent (meaning 10^5), the signed exponent is **5** (positive), and the sign is negative for the entire number.

In Scilab, real numbers have an absolute value ranging from about 2×10^{-308} to 1.8×10^{308}. The value range can change depending on the version of Scilab, the computer's hardware, and system configuration. For example, when the value exceeds the upper limit (around **1.8D+308**), such as **2D+308**, Scilab displays **%inf**, which represents infinity:

```
-->2D+308
ans=
     %inf
```

The result of a calculation operation is by default displayed with **7** digits after the decimal point (ten characters: the sign and the decimal point included). Example, by default, Scilab displays the number **-1.23456789123456789D+5** such as **-123456.79**, but maintains full precision internally for accurate calculations.

```
--> -1.23456789123456789D+5
ans=
     -123456.79
```

If more precision (more significant decimals) is needed, the **format** function is used. For example:

```
---> format(25)
---> -1.23456789123456789D+5
ans  =
     - 123456.78912345679418650
```

To return to the default format, type:

```
---> format('v', 10)
```

Notice that the valid range for the **format** function in Scilab is **[2, 25]**.

To obtain the scientific notation, the **format e** function is used. For example:

```
---> x=0.476190547
--> format e
---> x
ans=
     4.762D-01
```

Scilab provides the usual mathematical operations (addition, subtraction, multiplication, division) and elementary functions such as sine, cosine, exponential, etc.

Addition	Subtraction	Multiplication	Division	Exponentiation
+	**-**	*****	**/**	**^ or ****

The evaluation of an expression is executed from left to right, considering the operation priority indicated in the following table:

Operation	Parentheses ( )	Exponentiation	Multiplication and division	Addition and subtraction
Priority	1	2	3	4

For this example:

```
--> (%e^8-5)*(6+9/2)
 ans =
   31247.559
```

The order of calculation is shown below:

$$\underbrace{\left(\underbrace{\underbrace{e^{8}}_{1}-5}_{2}\right)}_{1}*\underbrace{\left(\underbrace{6+\underbrace{9/2}_{1}}_{2}\right)}_{2}$$

In the expression **(%e^8 - 5) * (6 + 9/2)**, Scilab follows the standard order of operations. First, it evaluates the parentheses: **(%e^8 - 5)** and

`(6 + 9/2)`. Next, it computes the exponent **`%e^8`**, then performs the division **9/2 = 4.5**. After that, it completes the addition and subtraction: **6 + 4.5 = 10.5** and `%e^8- 5`. Finally, it multiplies the two results to give $(e^8 - 5) \times 10.5$.

Some of the most commonly used functions include the following:

Function	Meaning
`sin(x)`	sine of x (in radians)
`cos(x)`	cosine of x (in radians)
`tan(x)`	tangent of x (in radians)
`sind(x), cosd(x), tand(x)`	sine, cosine, tangent of x (in degrees)
`asin(x)`	arcsine of x
`acos(x)`	arccosine of x
`atan(x)`	arctangent of x
`sqrt(x)`	square root of x
`abs(x)`	absolute value of x
`exp(x)`	exponential of x
`log(x)`	natural logarithm
`log10(x)`	logarithm to the base 10
`round(x)`	rounds a number to the nearest integer
`floor(x)`	rounds a number down to the nearest integer
`ceil(x)`	rounds a number up to the nearest integer
`sign(x)`	gives the sign of x

3. Complex numbers:

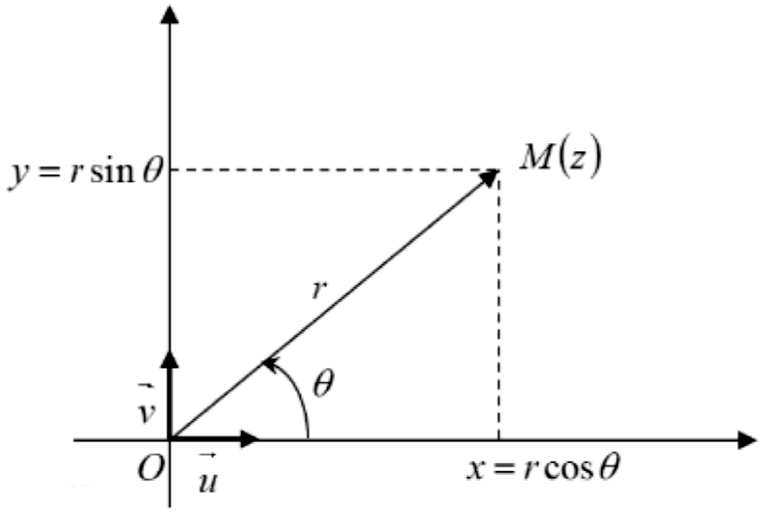
Scilab provides complex numbers that are stored as pairs of real numbers. Complex numbers consist of a real part and an imaginary part. In Scilab, complex numbers are represented using `%i` for the imaginary part. Complex numbers can be written in Cartesian form as **x + i y** or in polar form as $re^{i\theta}$, where **x**, **y**, **r**, θ are real numbers. Example:

```
--> z1 = 2 + 3 * %i
z 1=
      2. + 3.i
--> z2=-3*exp(4*%i)
z2=
      1.9609309+2.2704075i
```

Scilab allows mathematical operations with complex numbers, such as addition, subtraction, multiplication, and division. Example:

```
--> z1^2
 ans  =
       -5. + 12.i
--> z1 + z2
 ans  =
       3.9609309 + 5.2704075i
--> z1 - z2
 ans  =
       0.0390691 + 0.7295925i
--> z1 * z2
 ans  =
       -2.8893607 + 10.423608i
--> z1 / z2
 ans  =
       1.1925649 + 0.1491086i
```

In the $\mathbb{R}^2$ plane, a specific point **M(x,y)** is represented by the complex number **z = x + i y**, where **x** corresponds to the projection onto the real axis and **y** corresponds to the projection onto the imaginary axis.



The value $\mathbf{r} = |\mathbf{z}|$ (the modulus of **z**) and the argument are given by:

$$\begin{cases} \mathbf{r} = \sqrt{\mathbf{x}^2 + \mathbf{y}^2} \\ \boldsymbol{\theta} = \tan^{-1}\left(\dfrac{\mathbf{y}}{\mathbf{x}}\right) \end{cases}$$

.

Since $\mathbf{x} = \mathbf{r}\cos(\boldsymbol{\theta})$ and $\mathbf{y} = \mathbf{r}\sin(\boldsymbol{\theta})$, then $\mathbf{z} = \mathbf{r}(\cos(\boldsymbol{\theta}) + \mathbf{i}\sin(\boldsymbol{\theta})) = \mathbf{re}^{\mathbf{i}\boldsymbol{\theta}}$.

Let **z** be a complex number defined in Scilab

```
--> z = 2 + 3*%i
```

Some of the most commonly used functions on complex numbers include:

Function	Result	Description
`real(z)`	2	Returns the real part of z
`imag(z)`	3	Returns the imaginary part of z
`abs(z)`	3.605513	Returns the modulus of z
`atan(imag(z),real(z))`	0.9827937	Returns the angle θ
`conj(z)`	2 - 3i	The conjugate of z
`imult(z)`	-3 + 2i	The multiplication by i

Laboratory Work 2

Exercise 1:

1. **a)** Declare two variables, **`a`** and **`b`**, containing natural integers of your choice.
   b) Calculate the sum, difference, product, and quotient of **`a`** and **`b`**.
2. **a)** Declare two variables, **`a`** and **`b`**, containing real numbers of your choice.
   b) Calculate the sum, difference, product, and quotient of **`a`** and **`b`**.

Exercise 2:

1. **a)** Declare a variable containing a natural integer, for example, **`integer=5`**.
   b) Use the **`typeof`** function to determine the data type of the integer variable.
2. **a)** Declare a variable containing a real number, for example, **`real = 3.14`**.
   b) Use the **`typeof`** function to determine the data type of the real variable.
3. How does Scilab treat integers compared to real numbers in terms of data type?

Exercise 3:

1. Declare a variable containing a real number, for example, **`real_number = 3.14159265359`**.
2. Use the **`format()`** function to limit the display of **`real_number`** to **`2`**, **`15`**, and **`26`** decimal places.
3. Use the **`floor()`** function to round **`real_number`** down to the nearest integer (the nearest lower integer).
4. Use the **`ceil()`** function to round **`real_number`** up to the nearest integer (the nearest upper integer).
5. Use the **`round()`** function to round **`real_number`** to the nearest integer.

Exercise 4:

1. Give the Scilab command to calculate the following expression:

$$\frac{1}{\sqrt{8^3+1}} - \frac{2\sin\left(\frac{\pi}{4}\right)}{e^2} + \log(4)$$

2. Same question with decomposition:

$$\underbrace{\frac{1}{\sqrt{8^3+1}}}_{A} - \underbrace{\frac{2\sin\left(\frac{\pi}{4}\right)}{e^2}}_{B} + \underbrace{\log(4)}_{C}$$

Exercise 5:

1. Write a very small real number, for example, **`0.000000123`**, using scientific notation **`D`** and store it in a variable named **`small_number`**.

2. Write a very large real number, for example, **`1234567890`**, using scientific notation **`D`** and store it in a variable named **`large_number`**.

Exercise 6:

1. Identify the range of double-precision real numbers (approximately from 2.2×10^{-308} to 1.8×10^{308}).

2. Choose a positive real number within this range and store it in a variable named **`positive_number`**.

3) Choose a negative real number within this range and store it in a variable named **`negative_number`**.

Exercise 7:

1. Declare a variable **`epsilon`** and assign it the value of **`%eps`**.

2. Add **`epsilon`** to **`1`** and store the result in a variable **`result_eps`**.

3. What does **`%eps`** signify in Scilab?

4. Declare a variable **`infinity`** and assign it the value of **`%inf`**.

5. Add infinity to a positive number and store the result in a variable **`result_inf_positive`**.

6. Add infinity to a negative number and store the result in a variable **`result_inf_negative`**.

7. How do operations with **`%inf`** behave in calculations?

Exercise 8:

1. Declare two complex numbers **`z1`** and **`z2`** of your choice.
2. Perform the following operations and store the results in corresponding variables:
   - Addition of **`z1`** and **`z2`**.
   - Subtraction of **`z1`** by **`z2`**.
   - Multiplication of **`z1`** and **`z2`**.
   - Division of **`z1`** by **`z2`**.
3. Determine the conjugate of **`z1`**.
4. Calculate the real and imaginary parts of **`z2`** and store them in variables **`real_z2`** and **`imaginary_z2`**.
5. Calculate the modulus of **z1** and store it in a variable **`modulus_z1`**.
6. Calculate the argument of **`z1`** in radians and store it in a variable **`argument_z1`**.
7. Use the **`plot`** function to graph the complex numbers **`z1`** and **`z2`** on a complex plane.

Solution of Laboratory Work 2

Exercise 1 :

```
--> a = 7; b = 3;
--> sum = a + b;
--> difference = a - b;
--> product = a * b;
--> quotient = a / b;
--> a = 5.2; b = 1.8;
--> sum = a + b;
--> difference = a - b;
--> product = a * b;
--> quotient = a / b;
```

Exercise 2 :

```
--> integer = 5;
--> integer_type = typeof(integer);
--> real = 3.14;
--> real_type= typeof(reel);
```

Scilab treats integers and real numbers as **`constant`** data, meaning they are of the **double** data type, which is used for floating-point numbers.

Exercise 3 :

```
--> real_number = 3.14159265359
 real_number  =
               3.1415927
--> format(5)
--> real_number
 real_number  =
               3.14
--> format(26)
format: Wrong value for input argument #1: Must be in
the interval [2, 25].
--> floor(real_number)
 ans  =
        3.
--> ceil(real_number)
 ans  =
        4.
--> round(real_number)
 ans  =
        3.
```

Exercise 4 :

```
--> 1/sqrt(8^3+2)-2*sind(45)/exp(2)+log(4)
--> A=1/sqrt(8^3+2);
--> B=2*sind(45)/exp(2);
--> C=log(4);
--> disp(A-B+C)
```

Exercise 5 :

```
--> small_number = 1.230D-07
--> large_number= 1.234567890D+9
```

Exercise 6 :

The range of real numbers in double precision goes from approximately 2.2×10^{-308} to 1.8×10^{308}.

```
--> positive_number = 1.23456789012345D-100;
--> negative_number = -1.23456789012345D-100;
```

Exercise 7 :

```
--> epsilon = %eps
 epsilon =
           2.220D-16
--> result_eps = 1 + epsilon
 result_eps =
               1.
-->%eps+1==1
 ans  =
        F
```

In Scilab, double-precision numbers can range roughly from 2.2×10^{-308} to 1.8×10^{308}. The constant **%eps** ($\approx 2.22 \times 10^{-16}$) does not represent the smallest number in this range, but instead measures the precision of floating-point calculations, it is the smallest number that can, in theory, makes **1 + %eps** slightly greater than **1**. It is used to assess the precision of numerical calculations.

```
--> infinity = %inf
 infinity =
            Inf
--> result_inf_positive = infinity + 1000

 result_inf_positive =
                         Inf
--> result_inf_negative = infinity - 1000
  result_inf_negative =
                         Inf
```

Operations with **%inf** behave in such a way that they generate special results. Adding **%inf** to a positive number gives **%inf**, and adding **%inf** to a negative number also gives **%inf**. This reflects the behavior of mathematical infinities.

Exercise 8 :

```
--> z1=complex(2,1), z2=complex(1,2)
--> addition = z1 + z2,  subtraction  = z1 - z2
--> multiplication  = z1 * z2, division = z1 / z2
--> z1_bar=conj(z1),  real_z2 = real(z2)
--> imaginary_z2 = imag(z2)
--> modulus_z1 = abs(z1)
--> argument_z1 = atan(imag(z1), real(z1))
--> [radius, angle]=polar(z1)
--> figure(0)
--> clf()
--> hf=gcf()
--> hf.background=-2
--> ha=gca()
--> ha.data_bounds=[-5 -5; 5 5]
--> xgrid()
--> plot([0 2],[0 1],'b','LineWidth',3)
--> plot([0 1],[0 2],'r','LineWidth',3)
--> xlabel('Real axis (Re)','FontSize',2)
--> ylabel('Imaginary axis (Im)','FontSize',2)
--> legend('$\Large{z_{1}}$','$\Large{z_{2}}$')
--> plot(0,0,'sk')
--> plot(2,1,'sk')
--> plot(1,2,'sk')
--> xstring(2,1,'$\Large{z_{1}=2+i}$')
--> xstring(1,2,'$\Large{z_{2}=1+2i}$')
--> title('Complex numbers','FontSize',1)
```

Chapter 3. Vectors and Matrices

1. Operations on vectors and matrices

1.1. Vector and Matrix in Scilab:

A vector can be a row vector or a column vector.

In Scilab, a row vector contains elements in a single row. The elements are separated by commas "**,**", **spaces**, or both, and enclosed in square brackets **[ ]**. For example:

```
--> v = [1,2,3];    // Using commas
--> v = [1,2,3];    // Using spaces
--> v = [1,2 3];    // both comma and space
```

A column vector contains elements in a single column. The elements are separated by semicolons "**;**" and enclosed in square brackets **[ ]**. For example:

```
--> v = [1;2;3];    // Using semicolons
```

Specific vectors can be generated using the **":"** (colon) operator, as shown below:

```
--> x=1:5

 x  =

   1.   2.   3.   4.   5.
```

This assigns the integers from **1** to **5** to vector **x**. You can specify a different increment, for example:

```
--> y = 0:%pi/4:%pi

 y  =

   0.    0.7853982    1.5707963    2.3561945    3.1415927
```

This command always produces a row vector. Negative increments are also possible:

```
--> y = 6:-1: 1

 y  =

   6.    5.    4.    3.    2.    1.
```

You can create tables using these commands, which can later be used for graphical representations. For example:

```
--> x = [0: 0.2:1];
--> y = exp(-x).*sin(x);
--> [x' y']
 ans  =
   0.    0.
   0.2   0.1626567
   0.4   0.2610349
   0.6   0.3098824
   0.8   0.3223289
   1.    0.3095599
```

To specify only the minimum, maximum values, and the number of desired values, use the **linspace** command:

```
k = linspace(-%pi ,%pi ,5)
 k  =

  -3.1415927  -1.5707963   0.   1.5707963   3.1415927
```

A matrix in Scilab is defined as a set of row vectors (resp. column vectors) separated by a semicolons (resp. commas) and enclosed in brackets **[ ]**.

The matrix $\mathbf{A}=\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 0\end{pmatrix}$ can be written in Scilab using one of the following syntaxes :

```
--> A=[1 2 3;4 5 6;7 8 0]
--> A=[1 2 3
       4 5 6
       7 8 0]
--> A=[[1 2 3];[4 5 6];[7 8 0]]
```

The number of elements in each row (column) must be the same across all rows (columns) of the matrix.

An element of a matrix is referenced by its row and column numbers. **A(i,j)** denotes the element in the *i*-th row and *j*-th column of matrix **A**. For example:

```
--> A(2,3)
 ans  =
   6.
--> A(3,3)=A(1,3)+A(3,1)
 A  =
   1.    2.    3.
   4.    5.    6.
   7.    8.    10.
```

1.2. Submatrix Manipulation:

A(i,:) : Extracts row **i**.

A(:,j): Extracts column **j**.

A(:): Gives all elements of **A** as a column vector.

A(i:k,j:l): Extracts the submatrix between rows **i** and **k** and columns **j** and **l**.

A([i k],[j l]): Extracts the submatrix containing rows **i** and **k**, and columns **j** and **l**. For example:

```
--> A(2,:)
 ans  =
   4.   5.   6.
--> A(:,3)
 ans  =
   3.
   6.
   10.
--> A(:)
 ans  =
   1.
   4.
   7.
   2.
   5.
   8.
   3.
   6.
   10.
--> A(1:2,1:3)
 ans  =
   1.   2.   3.
   4.   5.   6.
--> A([1 3],[2 1])
 ans  =
   2.   1.
   8.   7.
```

1.3. Deleting rows or columns:

Deleting rows or columns is equivalent to inserting an empty matrix **[ ]**. For example:

```
--> A([1 2],:)=[]
 A  =

   7.   8.   10.
```

This instruction is equivalent to:

```
--> A=A([3:$],:)
 A  =
   7.    8.    10.
```

The symbol **$** designates the last element. The symbol **$** avoids the need to compute matrix dimensions. Example: Swap the first and last rows of matrix **A,**

```
--> A([1 $],:)=A([$ 1],:)
 A  =

   7.    8.    10.
   4.    5.    6.
   1.    2.    3.
```

1.3. Insert rows/columns:

This instruction inserts a row in the last position:

```
--> A($+1,:)=[11 12 13]
 A  =

   1.    2.    3.
   4.    5.    6.
   7.    8.    10.
   11.   12.   13.
```

And this instruction insert a column at the third position:

```
--> A=[A(:,1:2),[11;12;13],A(:,3)]
 A  =

   1.    2.    11.   3.
   4.    5.    12.   6.
   7.    8.    13.   10.
```

1.4. Basic Operations on Matrices:

Transposition

The special character "’" (prime or apostrophe) indicates the transposition operation. For example:

```
--> A=[1 2 3; 4 5 6; 7 8 0]
 A  =

   1.   2.   3.
   4.   5.   6.
   7.   8.   0.

--> B=A'
 B  =

   1.   4.   7.
   2.   5.   8.
   3.   6.   0.

--> X=[-1 0 2]'
 X  =

  -1.
   0.
   2.
```

The transposition operation transposes matrices in the classical sense. If **Z** is a complex matrix, then **Z'** gives the conjugate transpose, which both transposes the matrix and takes the complex conjugate of each element. While the command **Z.'** gives the transpose of the matrix **Z** without changing the sign of the imaginary parts. For example:

```
--> Z = [1 + 2*%i,3 - %i; -%i,4 + 3*%i]
 Z  =
   1. + 2.i   3. - i
  -i          4. + 3.i

--> Z_conjugate_transpose =Z'
 Z_conjugate_transpose  =
                           1. - 2.i   i
                           3. + i     4. - 3.i
--> Z_transpose = Z.'
 Z_transpose  =
                  1. + 2.i  -i
                  3. - i     4. + 3.i
```

Addition and Subtraction:

The operators "**+**" and "-" act on matrices and vectors. These operations are valid as long as the dimensions of the matrices or vectors are the same. For example, with the matrices from the previous example, the addition:

```
--> A=[1 2 3; 4 5 6; 7 8 0]
 A  =
   1.   2.   3.
   4.   5.   6.
   7.   8.   0.
--> X=[-1 0 2]'
 X  =
  -1.
   0.
   2.
--> A+X
Inconsistent row/column dimensions.
```

produces an "Inconsistent row/column dimensions" error because **A** is **3×3** and **X** is **3×1**. However, the following operation is valid:

```
--> A
 A  =
   1.   2.   3.
   4.   5.   6.
   7.   8.   0.
```

```
--> B
 B  =
   1.   4.   7.
   2.   5.   8.
   3.   6.   0.
--> C=A+B
 C  =
   2.    6.    10.
   6.    10.   14.
   10.   14.   0.
```

Addition and subtraction are also defined if one of the operands is a scalar, that is, a **1×1** matrix. In this case, the scalar is added to or subtracted from all elements of the other operand. For example:

```
--> z=X-1
 z  =
  -2.
  -1.
   1.
```

Multiplication:

The symbol "*" represents the matrix multiplication operator. This operation is valid as long as the dimensions of the operands are compatible, meaning that the number of columns of the left operand must equal the number of rows of the right operand. For example, the following operation is not valid:

```
--> X*z
Inconsistent row/column dimensions.
```

produces an "Inconsistent row/column dimensions" error. However, the following command:

```
--> X'*z
 ans  =
   4.
```

gives the dot product of **X** and **z**. Another valid command is:

```
--> b=A*X
 b  =
   5.
   8.
  -7.
```

Multiplying a matrix by a scalar is, of course, always valid:

```
--> A*2
 ans  =
   2.    4.    6.
   8.    10.   12.
   14.   16.   0.
```

Matrix inversion and division:

The inverse of a square matrix can be easily obtained with the **inv** command. For example:

```
--> B=inv(A)
 B  =
  -1.7777778   0.8888889  -0.1111111
   1.5555556  -0.7777778   0.2222222
  -0.1111111   0.2222222  -0.1111111
```

```
--> C=B*A
 C  =
   1.           4.441D-16    0.
  -2.220D-16    1.           0.
   0.           0.           1.
```

Note the minor errors due to finite calculation precision (matrix **C** should ideally be the identity matrix).

Matrix *division* is implemented in Scilab with the following meaning: the expression **A/B** yields the result of the operation $\mathbf{AB}^{-1}$. This is equivalent to the command **A*inv(B)**. For left division, the expression **A\B** yields the result of the operation $\mathbf{A}^{-1}\mathbf{B}$. This is equivalent to the command **inv(A)*B**. Dimension compatibility between the two matrices must be respected for this division to be meaningful.

Left division is commonly used when solving a linear system. For example, to solve the linear system **Ay=X**, with the equations:

$$\begin{cases} \mathbf{y}_1 + 2\mathbf{y}_2 + 3\mathbf{y}_3 = -1 \\ 4\mathbf{y}_1 + 5\mathbf{y}_2 + 6\mathbf{y}_3 = 0 \\ 7\mathbf{y}_1 + 8\mathbf{y}_2 = 2 \end{cases}$$

in Scilab, we write:

```
--> y=A\X
 y  =
   1.5555556
  -1.1111111
  -0.1111111
```

When Scilab interprets this expression, it does not invert matrix A before multiplying it by **X** on the right. Instead, it effectively solves the system of equations using Gaussian elimination, which is approximately three times faster than if we had written:

```
--> y=inv(A)*X

 y  =

   1.5555556

  -1.1111111

  -0.1111111
```

because in this case, we first need to calculate the inverse of the matrix (solving 3 linear systems) and then perform a matrix-vector multiplication. The accuracy of the results can be verified as follows:

```
--> A*y-X

 ans  =

   0.

  -2.220D-16

  -1.776D-15
```

As in the previous example, note the presence of errors on the order of machine precision.

Element-wise Operations:

The usual matrix operations can be performed element by element. For addition and subtraction, both viewpoints are the same since these two operations already act element by element on matrices.

The symbol "**.***" represents element-wise multiplication. If **A** and **B** have the same dimensions, then **A.*B** represents the array whose elements are simply the products of the individual elements of **A** and **B**. For example, if we define **x** and **y** as follows:

```
--> x=[1 2 3]
 x  =
   1.   2.   3.
--> y=[4 5 6]
 y  =
   4.   5.   6.
```

then the command:

```
--> z=x.*y
```

produces the result:

```
 z  =

   4.   10.   18.
```

Division works in the same way with "**./**":

```
--> z=x./y
 z  =
   0.25   0.4   0.5
```

The symbol "**.^**" represents element-wise exponentiation:

```
--> x.^y
 ans  =
   1.   32.   729.
--> x.^2
 ans  =
   1.   4.   9.
--> 2.^x
 ans  =
   2.   4.   8.
```

Note that for "**.^**", one of the operands can be a scalar.

Relational Operators:

Six relational operators are available to compare two matrices of equal dimensions:

< less than

<= less than or equal to

> greater than

>= greater than or equal to

== equal to

<> not equal

Scilab compares corresponding pairs of elements; the result is a matrix of boolean constants, with **F** (false) representing **false** and **T** (true) representing **true**. For example:

```
--> 2+2<>4
 ans  =
  F
```

Logical operators allow you to view the layout of elements in a matrix that satisfy certain conditions. For example, take the matrix:

```
--> A = [1 -1 2; -2 -4 1; 8 1 -1]
 A  =
   1.  -1.   2.
  -2.  -4.   1.
   8.   1.  -1.
```

The command:

```
--> P=(A<0)

P  =

 F  T  F

 T  T  F

 F  F  T
```

returns a matrix **P** with **T** indicating the negative elements of **A**.

Logical Operators:

The operators "**&**", "**|**", and "**~**" represent the logical operators **and**, **or**, and **not**, respectively. They are used to create logical expressions. For example, with the matrix from the previous example, the command:

```
--> P = (A < 0) & (modulo(A,2) == 0)
 P  =
  F  F  F
  T  T  F
  F  F  F
```

identifies the elements in **A** that are both negative and multiples of **2**.

2. Basic mathematical functions for matrix processing:

Function	Meaning
`ones(n), ones(n,m)`	Generates an n×n or n×m matrix with all elements equal to 1.
`zeros(n), zeros(n,m)`	Generates an n×n or n×m matrix with all elements equal to 0.
`eye(n)`	Generates an n×n identity matrix.

size	Calculates the number of rows and columns of a matrix.
Det	Calculates the determinant of a matrix.
inv	Calculates the inverse of a matrix.
rank	Calculates the rank of a matrix**.**
trace	Calculates the trace of a matrix**.**
spec	Calculates the eigenvalues**.**
norm	Calculates the norm**.**
diag(V)	Creates a matrix with vector V as the diagonal and 0 elsewhere.

Examples:

```
--> A=zeros(3,2)
 A  =

   0.   0.
   0.   0.
   0.   0.

--> B=ones(3,2)
 B  =

   1.   1.
   1.   1.
   1.   1.

--> I = eye(3,3)
 I  =

   1.   0.   0.
   0.   1.   0.
   0.   0.   1.

--> A=[1 2 3; 4 5 6; 7 8 9]; size(A)
 ans  =

   3.   3.
```

```
--> v=[3 5 1 2 4]; [n,m]=size(v)
 m  =

   5.

 n  =

   1.

--> det(A)
 ans  =

   6.661D-16

--> inv(A)
 ans  =

  -4.504D+15   9.007D+15  -4.504D+15
   9.007D+15  -1.801D+16   9.007D+15
  -4.504D+15   9.007D+15  -4.504D+15

--> rank(A)
 ans  =

   2.

--> trace(A)
 ans  =

   15.

--> spec(A)
 ans  =

   16.116844
  -1.116844
  -1.304D-15

--> norm(A)
 ans  =

   16.848103

--> diag(A)
 ans  =

   1.
   5.
   9.
```

Laboratory Work 3

Exercise 1 :

1. Define the matrix $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ e^{-3} & m/(l * \cos\left(\frac{2\pi}{3}\right)) \end{pmatrix}$ for m = 10 and l = 1.3.
2. Define the matrix $B = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$ for $\theta = \frac{\pi}{4}$.
3. Compute the product of A and B.
4. Add 1 to the elements of matrix A.
5. Compute the transpose of A.
6. Raise each element of A to the power of 3.
7. Compute $B + I$, where I is the identity matrix.
8. Define the matrix $X = \begin{pmatrix} A & -A \\ 0 & I \end{pmatrix}$.
9. Extract the submatrix formed by rows 2 and 3 and column 2 of X.
10. Swap columns 2 and 4 of X.
11. Remove the first row of X.
12. Insert the vector (1, 1, 3, 4) into the second row of X.
13. Solve the linear system $BX = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

Exercise 2 :

1. Using the **`diag`** and **`ones`** commands, define the following square matrix:

$$\mathbf{A}=\begin{pmatrix}\mathbf{1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{3} & \mathbf{1} & \mathbf{2} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{3} & \mathbf{1} & \mathbf{2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{3} & \mathbf{1} & \mathbf{2}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{3} & \mathbf{1}\end{pmatrix}$$

Exercise 3 :

1. Generate a diagonal matrix with random numbers.
2. Reverse rows or columns to create an anti-diagonal matrix.
3. Generate two random matrices **A** and **B** and find the Boolean matrix corresponding to elements of A greater than those of B.
4. Generate a random matrix of size **5×5** and subtract **1** from elements greater than **0.5**.
5. Generate a random matrix of size **5×5** with integers between **0** and **10**, and find the positions of elements equal to zero.
6. Let $\mathbf{x}=(1,2,...,10)'$. Calculate its Euclidean norm.

Exercise 4 :

1. Define a vector **`x`** that represents the uniform discretization of the interval **[-4, 2]** into **9** equal intervals.
2. Determine **`x(1)`**, **`x(4)`**, and explain why **`x(0)`** does not exist.
3. Explain **`x($)`**, **`length(x)`**, **`x(2:3)`**, **`x(:)`**, and **`x([1,1])`**.
4. Reverse the vector **`x`**.

Exercise 5 :

1. Let the vector **C = [-2, 5, -1 , 0, 2, 13, 6, 9, -4, 6, 1, 8, 2, -3, 11]**. Find the elements of **C** that are greater than **-1** and less than **7**, then replace them with **1**.
2. Construct a vector **D** containing the unique elements of **C** without duplicates.

Solution of Laboratory Work 3

Exercise 1 :

```
 --> m=10; l=1.3;
--> A=[1 2; 3 4; %e^(-3)  m/(l*cos(2*%pi/3))]
 A  =
   1.          2.
   3.          4.
   0.0497871  -15.384615
-->   theta=%pi/4;   B=[cos(theta)       sin(theta)   ;
-sin(theta)  cos(theta)]
 B  =
   0.7071068   0.7071068
  -0.7071068   0.7071068
--> A*B
 ans  =
  -0.7071068   2.1213203
  -0.7071068   4.9497475
   10.913771  -10.843361
--> 1+A
 ans  =
   2.          3.
   4.          5.
   1.0497871  -14.384615
--> A'
 ans  =
   1.   3.   0.0497871
   2.   4.  -15.384615
--> A.^3
```

```
 ans  =
   1.          8.
   27.         64.
   0.0001234  -3641.3291
--> B+eye(B)
 ans  =
   1.7071068   0.7071068
  -0.7071068   1.7071068
--> X=[A  -A ; zeros(A)  eye(A)]
 X  =
   1.          2.          -1.          -2.
   3.          4.          -3.          -4.
   0.0497871  -15.384615   -0.0497871    15.384615
   0.          0.           1.           0.
   0.          0.           0.           1.
   0.          0.           0.           0.
--> X([2 3] , 2)
 ans  =
   4.
  -15.384615
--> X(: , [2 4])=X(: , [4 2])
 X  =
   1.         -2.          -1.           2.
   3.         -4.          -3.           4.
   0.0497871   15.384615   -0.0497871  -15.384615
   0.          0.           1.           0.
   0.          1.           0.           0.
   0.          0.           0.           0.
--> X( 1 , :)=[ ]
 X  =
   3.         -4.          -3.           4.
   0.0497871   15.384615   -0.0497871  -15.384615
   0.          0.           1.           0.
   0.          1.           0.           0.
   0.          0.           0.           0.
```

```
--> X=[X( 1 , :) ; [ 1  1  3  4];   X(2 : 4 , : )]
 X  =
   3.          -4.          -3.           4.
   1.           1.           3.           4.
   0.0497871    15.384615   -0.0497871   -15.384615
   0.           0.           1.           0.
   0.           1.           0.           0.
--> X= inv(B)*[1 ;  1 ]
 X  =
   0.
   1.4142136
```

Exercise 2 :

```
--> diag(3*ones(4,1),-1) + diag(ones(5,1)) +
diag(2*ones(4,1),1)
      ans  =
        1.   2.   0.   0.   0.
        3.   1.   2.   0.   0.
        0.   3.   1.   2.   0.
        0.   0.   3.   1.   2.
         0.   0.   0.   3.   1.
```

Exercise 3 :

```
--> A=rand(1,5), B=diag(A)
 A  =
   0.2113249     0.7560439     0.0002211     0.3303271
0.665 3811
 B  =
   0.2113249   0.           0.           0.           0.
   0.          0.7560439    0.           0.           0.
   0.          0.           0.0002211    0.           0.
   0.          0.           0.           0.3303271    0.
   0.          0.           0.           0.     0.6653811
```

```
--> B([1:$],:)=B([$:-1:1],:)
 B  =
   0.          0.          0.          0.   0.6653811
   0.          0.          0.          0.3303271   0.
   0.          0.          0.0002211   0.          0.
   0.          0.7560439   0.          0.          0.
   0.2113249   0.          0.          0.          0.
--> A=rand(3,4), B=rand(3,4), C=A>B
 A  =
   0.6283918   0.8782165   0.6623569   0.5442573
   0.8497452   0.068374    0.7263507   0.2320748
   0.685731    0.5608486   0.1985144   0.2312237
 B  =
   0.2164633   0.3076091   0.312642    0.5664249
   0.8833888   0.9329616   0.3616361   0.4826472
   0.6525135   0.2146008   0.2922267   0.3321719
 C  =
  T T T F
  F F T F
  T T F F
--> A=rand(5,5), k=find(A>0.5), A(k)=A(k)-1
 A  =
 0.5935095  0.4051954 0.4148104   0.1121355  0.4062025
   0.5015342  0.9184708 0.2806498 0.6856896  0.4094825
   0.4368588 0.0437334 0.1280058 0.1531217   0.8784126
   0.2693125 0.4818509 0.7783129  0.6970851   0.113836
   0.6325745 0.2639556 0.211903  0.8415518   0.1998338
 k  =
   1.   2.   5.   7.   14.   17.   19.   20.   23.
 A  =
-0.4064905 0.4051954 0.4148104  0.1121355   0.4062025
-0.4984658 -0.0815292 0.2806498-0.3143104   0.4094825
0.4368588 0.0437334 0.1280058   0.1531217  -0.1215874
0.2693125 0.4818509  -0.2216871  -0.3029149   0.113836
-0.3674255  0.2639556  0.211903 -0.1584482   0.1998338
```

```
--> A=grand(5,5,"uin",0,10),  position=find(A==0)
 A  =
   6.    7.    2.    5.    6.
   3.    0.    3.    2.    5.
   10.   6.    3.    3.    7.
   9.    5.    8.    2.    8.
   4.    4.    0.    7.    6.
 position  =
   7.   15.
--> X=[1:10]', norm(X)
 X  =
   1.
   2.
   3.
   4.
   5.
   6.
   7.
   8.
   9.
   10.
 ans  =
   19.621417
```

Exercise 4 :

```
--> x=linspace(-4,2,9)
x  =
 -4.  -3.25  -2.5  -1.75  -1.  -0.25   0.5   1.25   2.
--> x(1),x(4)
      ans  =
       -4.
      ans  =
       -1.75
--> x(0)
Invalid index.
```

```
-> x($)
 ans  =
   2.
--> length(x)
 ans  =
   9.
--> x(2:3)
 ans  =
  -3.25  -2.5
--> x(:)
 ans  =
  -4.
  -3.25
  -2.5
  -1.75
  -1.
  -0.25
   0.5
   1.25
--> x([1,1])
 ans  =
  -4.  -4.
--> x([1:length(x)])=x([length(x):-1:1])
 x  =
  2.   1.25   0.5  -0.25  -1.  -1.75  -2.5  -3.25  -4.
```

x(1) and **x(4)** are respectively the first and the fourth elements of the vector **x**. The element with index **0**, **x(0)**, does not exist in Scilab (indices in Scilab start from **1**). **x($)** is the last element of the vector **x**. **length(x)** is the length of the vector **x**. **x(2:3)** represents the elements of **x** with indices **2** and **3**. **x(:)** gives the elements of **x** as a column vector. **x([1,1])** extracts a submatrix containing the two elements with index **1**.

Exercise 5 :

```
--> C = [-2, 5, -1 , 0, 2, 13, 6, 9, -4, 6, 1, 8, 2, -
3, 11]
 C  =
  -2.    5.  -1.    0.    2.    13.    6.    9.  -4.    6.
1.    8.    2.  -3.    11.
--> k=find(C > 1 & C < 7)
 k  =
   2.    5.    7.    10.    13.
--> C(k) = 1
 C  =
  -2.    1.  -1.    0.    1.    13.    1.    9.  -4.    1.
1.    8.    1.  -3.    11.
--> [s,k]=gsort( C )
 k  =
   6.    15.    8.    12.    2.    5.    7.    10.    11.
13.    4.    3.    1.    14.    9.
 s  =
   13.    11.    9.    8.    1.    1.    1.    1.    1.    1.
0.  -1.  -2.  -3.  -4.
--> k2=find(s(1: $-1)==s(2:$))
 k2  =
   5.    6.    7.    8.    9.
--> s(k2)=[]
 s  =
   13.    11.    9.    8.    1.    0.  -1.  -2.  -3.  -4.
--> D=s
 D  =
   13.    11.    9.    8.    1.    0.  -1.  -2.  -3.  -4.
```

Chapter 4. Programming Basics

The way Scilab has been used in the previous chapters may give the impression that it is simply an "enhanced calculator" capable only of executing commands entered via the keyboard and displaying the result. In reality, in Scilab, programs can be written either as *scripts* or as *functions*, and this mode of operation is preferred as soon as the number of command lines becomes sufficiently large.

1. Scripts:

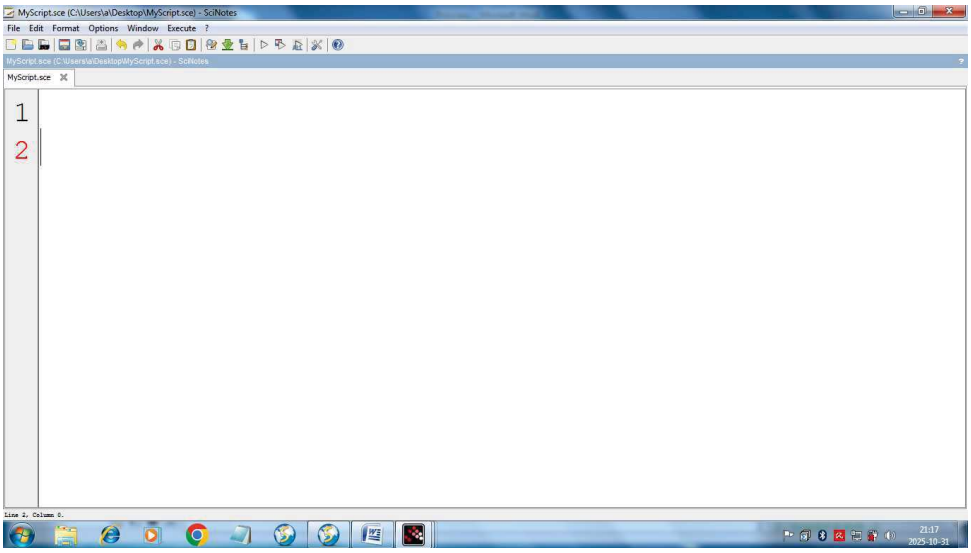
A *script* in Scilab is a text file that contains a list of commands written just as they would be entered in the console, and when executed, Scilab runs all these commands automatically in sequence. It is usually saved with the extension **.sce** and created using **SciNotes**, Scilab's integrated text editor. A script shares the same workspace as the main environment, meaning that all variables defined in it remain available after execution. Scripts are especially useful for repeating a set of commands or performing long calculations without having to retype them.

To accomplish this, you can launch the script editor directly from the console using the command:

```
--> scinotes
```

Alternatively, you can select **SciNotes** from the Scilab menu.

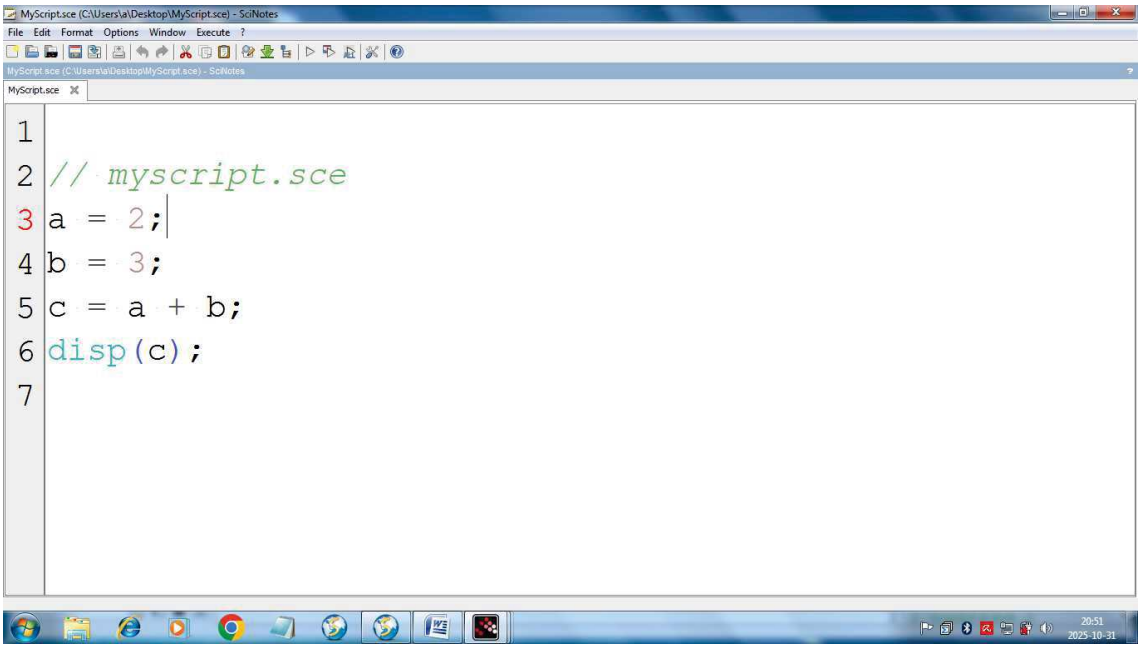
The file created is named **Untitled** by default. To change this name, simply select **Save As** from the **File** menu and choose the name **MyScript.sce**. The created window should look like this:



You can now enter the following lines in the editor window:

```
// MyScript.sce
a = 2;
b = 3;
c = a + b;
disp(c);
```

The created window should look like this:



Once the text has been entered in the window, save the file (Use the save button or by selecting **Save** from the **File** menu).

In a script, parentheses, loop endings, function closures, and test commands are automatically generated.

To execute the script "MyScript.sce", simply click on the button  , which should display the following lines in the command window:

```
--> exec('C:\Users\a\Desktop\MyScript.sce', -1)
   5.
```

You can also execute the file by selecting **…file with echo** from the **Execute** menu or by entering the following command in the console:

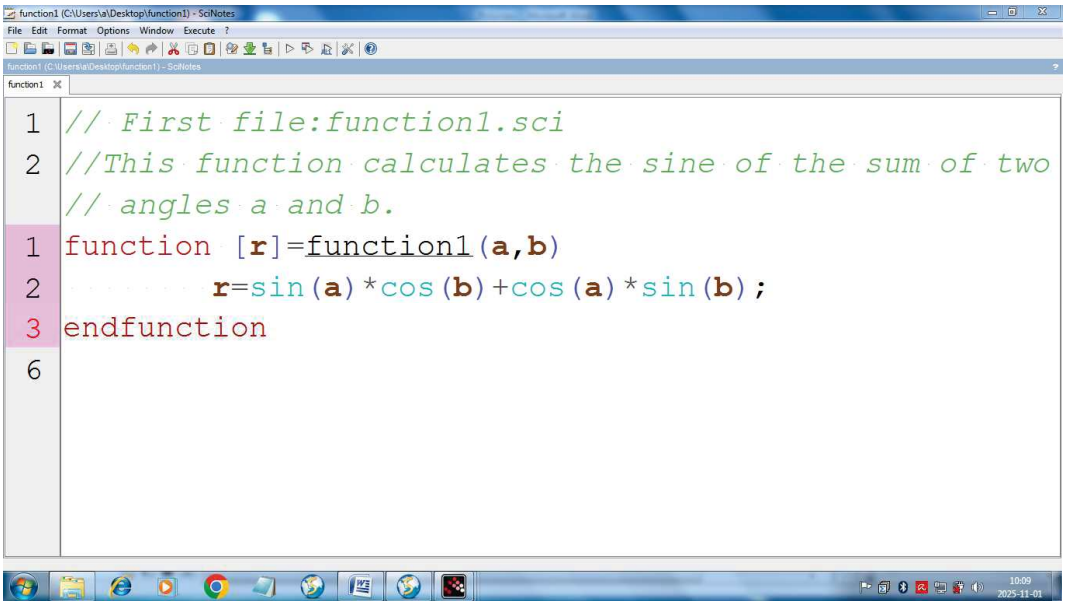
```
--> exec('C:\Users\a\Desktop\MyScript.sce')
```

2. Functions:

A *function* is a block of instructions stored in a **.sci** file that can take input arguments and return one or more results. Functions help structure and organize code, making it easier to manage complex or repeated computations. The general syntax for defining a function is as follows:

```
function [r1, r2, ..., rm] = function_name(arg1, arg2,..., argn)
    // Function body
    r1 = ...  // Value returned by r1
    ...
    rm = ...  // Value returned by rm
endfunction
```

As an example, let's create a function to calculate the **`sine`** of the sum of two angles **`a`** and **`b`**.



The file name can be different from the name of the function it defines, and a single file may contain several functions. The function is then loaded into the environment and can be executed.

```
--> exec('C:\Users\a\Desktop\function1')

--> function1(2,3)

ans =

  -0.9589243
```

It is also possible to define an "inline" function, i.e., directly from the Scilab command line. This is practical when the function is very short to write. For example:

```
--> deff("c=plus(a,b)","c=a+b");
--> plus(1,2)
 ans =
       3.
```

In Scilab, most of the time, we work with vectors and matrices. The operators and basic functions are designed to support this kind of manipulation and, more generally, to allow program vectorization. Of course, the Scilab language includes conditional operations (**`if`**–**`then`**–**`else`**, **`elseif`**), loops (**`while`**, **`for`**), and recursive programming. However, vectorization helps to reduce the use of these features, which are not very efficient in an interpreted language. The interpretation overhead can be dramatically penalizing compared to what a compiled **C** or **Fortran** program can achieve when performing mainly scalar calculations. Efficiency losses by a factor of **10** or even **100** can occur. Therefore, it is important to minimize the interpreter's workload by vectorizing programs as much as possible.

Control structures are instructions that define and manage the order of execution in a program. They allow the program to make decisions based on conditions or to repeat actions through loops for specific processes.

3. Control loops:

"for" Loop:

The **`for`** loop is commonly used for repetition, executing a block of instructions a predetermined number of times. Syntax:

```
for var = start : step : end
   instructions
end
//"var" takes values from "start" to "end", increasing
by "step".
```

For example:

```
--> for i = 1 : 5 : 20
  >     i*i
  > end
 ans =
   1.
 ans =
   36.
 ans =
   121.
 ans =
   256.
```

Alternatively, you can replace **start : step : end** with a vector. For example:

```
--> v=[1 3 7 8]
 v = [1x4 double]
   1.   3.   7.   8.
--> for i=v
  >     i/2
  > end
 ans =
   0.5
 ans =
   1.5
 ans =
   3.5
```

```
   ans =

     4.
```

Loops can also be used to define matrices:

```
--> n=4; a=zeros(n,n);

--> for i=1 : n-1

  >   a(i,i)=2;  a(i,i+1)=1; a(i+1,i)=-1;

  > end

--> a(n,n)=2;

--> a

 a = [4x4 double]

   2.   1.   0.   0.

  -1.   2.   1.   0.

   0.  -1.   2.   1.

   0.   0.  -1.   2.
```

Use **break** to exit a loop:

```
--> for i=0:0.01:1
  > test=i-exp(-i);
  > if (test>0) then
  > i
  > break
  > end
  > end
 i =
   0.5700000
```

"While" Loop:

The **while** loop executes instructions an indeterminate number of times, based on a logical condition. The syntax is:

```
while condition do
            instructions
    // instructions to execute while the condition is
true
End
```

For example**:**

```
--> // Example: Display numbers from 1 to 5
--> i = 1; // initialization
--> while i < 3 do
  >     disp(i);   // display the current value of i
  >     i = i + 1; // increment i
  > end
   1.
   2.
```

4. <u>Conditional statements:</u>

The "**if**" statement is the simplest and most commonly used control structure. It executes different actions based on whether a condition is true or false. The syntax of the if control structure is:

```
if condition then
  instructions
// statements executed when the condition is true
End
```

or:

```
if condition then
instructions //statements executed when the condition is true
else
Instructions //statements executed when the conditions is false
End
```

For example:

```
--> a=3.5;
--> if a>1 then
  >    c=log(a-1)
  > end
--> c
 c =
   0.9162907
```

Using `if`, `then`, and `else`: The "instructions1" are executed if the logical expression "condition" is true; otherwise, the "instructions2" are executed.

```
  -> a=0.9;
  --> if a>0 then
    > if a>1 then
    > f=1;
    > else
    > f=a*a;
    > end
    > else
    > f=0;
    > end
  --> f
   f =
     0.81
```

If **`else`** is followed by another **`if`**, you can use:

```
if condition then

      instructions
    // statements executed when the condition is true
elseif another_condition then
      instructions
    // statements executed when the second condition
    //is true
```

```
else
      instructions
    // statements executed when all conditions are
    // false
End
```

5. Reading and Displaying Variables (Input and Output):

The **input** command is used for user input. The syntax is:

```
variable = input('prompt text')
```

Example:

```
-> A=input('Enter the matrix A:  ')
Enter the matrix A:  [1,2;3,4]
 A = [2x2 double]
   1.   2.
   3.   4.
```

Second example:

```
--> name = input('Enter your name:  ', 'string')
Enter your name:  Scilab
 name =
  "Scilab"
```

The "string" option allows the input of character strings.

The **disp** command is used to display variables. Its syntax is:

```
disp(var)
```

Here, "var" can be a number, vector, matrix, string, or expression. Example:

```
--> t = 5;
--> disp("The solution is = ",(1 + sqrt(t)) / 2 )
  "The solution is = "
   1.6180340
```

Laboratory Work 4

Exercise 1:

Write a Scilab function that takes a real number **x** as input and returns $\mathbf{e^{-x}}$ if $\mathbf{x \geq 0}$, and **0** in all other cases.

Exercise 2:

Write a Scilab function that takes a real number **x** as input and returns **x + 1** if $\mathbf{x \in [-1, 0]}$, **−x + 1** if $\mathbf{x \in [0, 1]}$, and **0** in all other cases.

Exercise 3:

Write a Scilab function that calculates a factorial of a natural integer.

Exercise 4:

Write an Scilab function that calculates a few terms of the Fibonacci sequence, defined as follows:

$$\begin{cases} \mathbf{u}_0 = 1,\ u_1 = 1 \\ \mathbf{u}_{n+2} = \mathbf{u}_{n+1} + \mathbf{u}_n \end{cases}$$

Exercise 5:

Write a Scilab function that calculates a binomial coefficient $\mathbf{C_n^k = \frac{n}{k} C_{n-1}^{k-1}}$.

Exercise 6:

Write a **script file** in Scilab to solve the equation $\mathbf{ax^2 + bx + c = 0}$ for real numbers **a**, **b** and **c**. The script should prompt the user for the values of **a**, **b**, and **c**, compute the solution(s), and display the results.

Solution of Laboratory Work 4

Exercise 1:

```
--> function y = f(x)
  >     if x >= 0 then
  >         y = exp(-x);
  >     else
  >         y = 0;
  >     end
  > endfunction

--> f(3.4)

 ans =
   0.0333733

--> f(-0.1)

 ans =
   0.
```

Exercise 2:

```
--> function y = f(x)
  >     if x >= -1 & x <= 0 then
  >         y = x + 1;
  >     elseif x > 0 & x <= 1 then
  >         y = -x + 1;
  >     else
  >         y = 0;
  >     end
  > endfunction
```

```
--> f(-0.5)
 ans =
   0.5
--> f(0.8)
 ans =
   0.2000000
--> f(7)
 ans =
    0.
```

Exercise 3:

```
--> function [f]=fact(n)
   > if n <= 1 then
  > f = 1
  > else
  > f = n*fact(n-1)
  > end
  > endfunction

--> fact(5)
 ans =
     120.
```

Exercise 4:

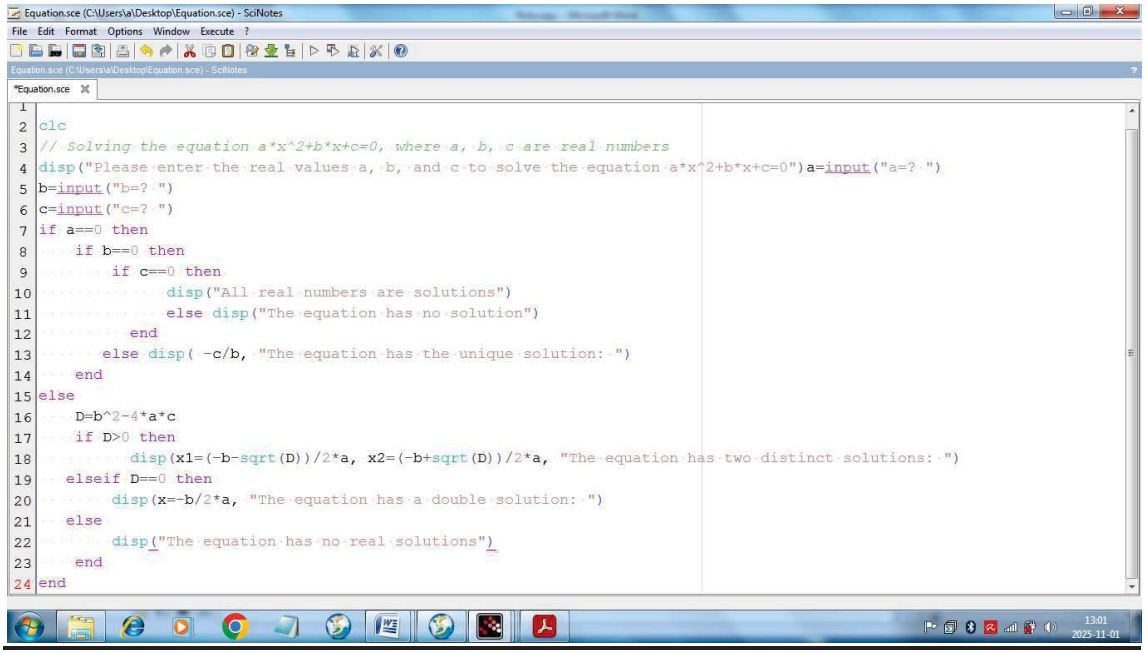
```
--> function [u]=fib(n)
  > // calcul du n ieme terme
  > // de la suite de Fibonnaci :
  > // fib(0) = 1, fib(1) = 1,
  > // fib(n+2) = fib(n+1) + fib(n)
  > if n <= 1 then
```

```
       > u = 1
       > else
       > u = fib(n-1) + fib(n-2)
       > end
       > endfunction

     --> fib(7)
      ans =
            21.
```

Exercise 5:

```
--> function C = binomial(n, k)
  > //Calculates the binomial coefficient C(n, k)
  >     if k == 0 | k == n then
  >         C = 1;
  >     else
  >         C = (n / k) * binomial(n - 1, k - 1);
  >     end
  > endfunction

--> binomial(10, 7)

 ans =

   120.
```

Exercise 6:




```
Clc
// Solving the equation a*x^2+b*x+c=0, where a,
// b and c are real numbers
disp("Please enter the real values a, b, and c to
solve the equation a*x^2+b*x+c=0 : ")
a=input("a=? ")
b=input("b=? ")
c=input("c=? ")
if a==0 then
    if b==0 then
        if c==0 then
        disp("All real numbers are solutions")
        else disp("The equation has no solution")
        end
    else disp( "The equation has the unique
solution: ", -c/b)
    end
else
```

```
 D=b^2-4*a*c
 if D>0 then
 disp("The equation has two distinct solutions:",
 x1=(-b-sqrt(D))/2*a, x2=-b+sqrt(D))/2*a)
 elseif D==0 then
 disp("The equation has a double solution: ",
x=-b/2*a)
 else
  disp("The equation has no real solutions")
  end
end
```

Chapter 5. Polynomials

Scilab provides robust tools for working with polynomials as formal mathematical objects. This chapter covers the definition, operations, and functions applications to polynomials and rational fractions.

1. Polynomials and rational fractions:

Scilab uses the predefined variable **%s** to represent the monomial of degree 1. Polynomials and rational fractions can be created through simple expressions:

```
--> p = 1 + %s + 2 * %s^2
 p = [polynomial] of s
  1 +s +2s^2
--> q=p/(1+%s)
 q = [rational] of s
   1 +s +2s^2
   ----------
      1 +s
```

To define a polynomial explicitly, use the **poly** command:

```
--> x = poly(0, 'x')
 x = [polynomial] of x
  x
--> p = x^2 + 2*x + 1
 p = [polynomial] of x
  1 +2x +x^2
```

A polynomial can also be defined by specifying its **roots**:

```
--> p1 = poly([0, 1, 2], 's', 'roots')

 p1 = [polynomial] of s

  2s -3s^2 +s^3
```

Alternatively, a polynomial can be created from its **coefficients**, listed in ascending order of degree:

```
--> p2 = poly([1, 2, 3], 's', 'coeff')

 p2 = [polynomial] of s

  1 +2s +3s^2
```

The characteristic polynomial of a square matrix **A** is defined as **det(A-xI)=p(x)**:

```
--> A = [1 2; 3 4]

 A = [2x2 double]

   1.   2.

   3.   4.

--> p = poly(A, 'x')

 p = [polynomial] of x

  -2 -5x +x^2
```

2. Zeros of a polynomial:

In Scilab, the zeros (or roots) of a polynomial are the values of **x** for which the polynomial equals zero. To find them, we use the function `roots()`. For example, consider the polynomial $\mathbf{P(x) = x^3 - 6x^2 + 11x - 6}$. Its coefficients are represented by the vector **`[-6 11 -6 1]`**. The instruction **`r = roots(p)`** gives the zeros of **P(x)**.

```
--> p = poly([-6 11 -6 1], 's', 'coeff')
 p = [polynomial] of s
  -6 +11s -6s^2 +s^3
--> roots(p)
 ans = [3x1 double]
   3. + 0.i
   2. + 0.i
   1. + 0.i
```

3. Operations on Polynomials:

Polynomials with the same variable can be added, subtracted, multiplied, divided, or raised to a power:

```
--> p1 = poly([0, 1, 2], 's', 'roots')
 p1 = [polynomial] of s
  2s -3s^2 +s^3
--> p2 = poly([1, 2, 3], 's', 'coeff')
 p2 = [polynomial] of s
  1 +2s +3s^2
--> p1 + p2
 ans = [polynomial] of s
  1 +4s +s^3
--> p1 - p2
 ans = [polynomial] of s
```

```
  -1 -6s^2 +s^3
--> p1*p2
 ans = [polynomial] of s
  2s +s^2 +s^3 -7s^4 +3s^5

--> p1 / p2
 ans = [rational] of s
   2s -3s^2 +s^3
   -------------
    1 +2s +3s^2
--> p1^4
 ans = [polynomial] of s
  16s^4 -96s^5 +248s^6 -360s^7 +321s^8 -180s^9 +62s^10
-12s^11 +s^12
```

Scilab provides commands for Euclidean and polynomial division:

pdiv(p1, p2) performs Euclidean division.

ldiv(p1, p2, k) divides polynomials following increasing powers.

```
   --> pdiv(p1, p2)
    ans = [polynomial] of s
     -1.2222222 +0.3333333s
   --> ldiv(p1,p2,4)
    ans = [4x1 double]
     -1.5
      1.75
     -0.875
      0.4375
```

Scilab offers several functions for polynomial manipulation:

Function	Description
`degree(p)`	Returns the degree of polynomial p.
`derivat(p)`	Computes the derivative of p.
`coeff(p)`	Returns the coefficients of p in ascending order.
`roots(p)`	Finds the roots of p.
`varn(p)`	Returns the variable name of p.
`factors(p)`	Factorizes p into simpler components.
`bezout(p, q)`	Computes the greatest common divisor.
`horner(p, t)`	Evaluates p at value t.
`simp(q)`	simplifies rational fractions q.

Laboratory Work 5

Exercise 1:

1. Create the polynomial **p(s) = 3 + 2s + s^2** and display it.
2. Define a rational fraction **q(s) = p(s)/(1 + 2s)** and display its form.
3. Using `poly(0, 'x')`, create a polynomial variable **x**, then define **f(x) = x^3- 4x + 2.**
4. Define a polynomial from its **roots**: **[-2, 0, 2]**.
5. Define another polynomial from its **coefficients [2, -3, 1, 4]** in ascending order of degree.
6. Compute the characteristic polynomial of $\mathbf{A} = \begin{pmatrix} 2 & 1 \\ -1 & 3 \end{pmatrix}$.

Exercise 2:

1. Define the polynomial **P(x) = x^3 - 5x^2 + 2x + 8** then, find its zeros.
2. Verify one of the zeros.
3. Define another polynomial **R(x) = x^4 - 2x^3 - 7x^2 + 8x + 12** and compute its zeros.

Exercise 3:

1. Define: **p1(s) = s^3 - 2s + 1** and **p2(s) = 2s^2 + s + 3**. Perform the following operations:
   p1 + p2 , **p1 - p2**, **p1×p2**, **p1/p2**, **$p1^3$**.
2. Perform the Euclidean division of **p1** by **p2**.
3. Compute the increasing powers division of **p1** by **p2**.

Exercise 4:

Let **$p(s) = s^4 - 3s^3 + 2s - 1$** and **$q(s) = s^3 - s^2 + s - 1$**. Perform the following:

1. Find the degree of **p**.
2. Compute its derivative.
3. Display its coefficients.
4. Get its variable name.
5. Factorize both **p** and **q**.
6. Compute:
   Least Common Multiple of **p** and **q**.
   Greatest Common Divisor of **p** and **q**.
7. Evaluate **p(s)** for **s = 1.5**.
8. Simplify the rational fraction **r = P/q**.

Solution of Laboratory Work 5

Exercise 1:

```
clc;

clear;

// 1. Create p(s) = 3 + 2s + s^2 and display it

--> s = poly(0, 's');

--> p = 3 + 2*s + s^2;

--> disp("p(s) =", p);

"p(s) ="

  3 +2s +s^2

// 2. Define a rational fraction q(s) = p(s)/ (1 + 2s)

--> q = p/(1 + 2*s);

--> disp("q(s) =",q);

  "q(s) ="

   3 +2s +s^2

   ----------

     1 +2s

// 3. Create a polynomial variable x and define f(x) =
x^3 - 4x + 2

--> x = poly(0, 'x');
```

```
--> f = x^3 - 4*x + 2;

--> disp("f(x) =",f);

"f(x) ="

  2 -4x +x^3

// 4. Define a polynomial from its roots [-2, 0, 2]

--> r = poly([-2, 0, 2], 'x', 'roots');

--> disp("Polynomial from roots =", r);

"Polynomial from roots ="

  -4x +x^3

// 5. Define a polynomial from coefficients [2, -3, 1,
4] (ascending degree)

--> p2 = poly([2, -3, 1, 4], 'x', 'coeff');

--> disp("Polynomial from coefficients =",p2);

  "Polynomial from coefficients ="

  2 -3x +x^2 +4x^3

//  6.  Compute  the  characteristic  polynomial  of  a
matrix

--> A = [1 2; 3 4];

--> char_poly = poly(A, 'x');

-->   disp("Characteristic   polynomial   of   A   =",
char_poly);

"Characteristic polynomial of A ="

  -2 -5x +x^2
```

Exercise 2:

```
clc;

// 1. Define P(x) = x^3 - 5x^2 + 2x + 8 and find its
zeros

--> x = poly(0, 'x');

--> P = x^3 - 5*x^2 + 2*x + 8;

--> zP = roots(P);

--> disp("Zeros of P(x) =", zP);

  "Zeros of P(x) ="

   4. + 0.i

   2. + 0.i

  -1. + 0.i

// 2. Verify one of the zeros (first one)

--> test_val = horner(P, zP(1));

disp("Verification of first zero:", test_val);

 "Verification of first zero:"

  -6.217D-14

// 3. Define R(x) = x^4 - 2x^3 - 7x^2 + 8x + 12 and
compute its zeros

--> R = x^4 - 2*x^3 - 7*x^2 + 8*x + 12;

--> zR = roots(R);

--> disp("Zeros of R(x) =", zR);

  "Zeros of R(x) ="

   3. + 0.i
```

```
   2. + 0.i

  -2. + 0.i

  -1. + 0.i
```

Exercise 3:

```
clc;

// 1. Define p1(s) and p2(s)

--> s = poly(0, 's');

--> p1 = s^3 - 2*s + 1;

--> p2 = 2*s^2 + s + 3;

// Operations

--> disp("p1 + p2 =", p1 + p2);

  "p1 + p2 ="

  4 -s +2s^2 +s^3

--> disp("p1 - p2 =", p1 - p2);

 "p1 - p2 ="

  -2 -3s -2s^2 +s^3

--> disp("p1 * p2 =",p1 * p2);

  "p1 * p2 ="

  3 -5s -s^3 +s^4 +2s^5

--> disp("p1 / p2 =" , p1 / p2);

  "p1 / p2 ="
```

```
   1 -2s +s^3
   ----------
   3 +s +2s^2
--> disp("p1^3 =",p1^3);
"p1^3 ="
  1 -6s +12s^2 -5s^3 -12s^4 +12s^5 +3s^6 -6s^7 +s^9
// 2. Euclidean division of p1 by p2
--> [q, r] = pdiv(p1, p2);
--> disp("Quotient =", q);
"Quotient ="
  1.75 -3.25s
--> disp("Remainder =", r);
"Remainder ="
  -0.25 +0.5s
// 3. Increasing powers division (use pfactors)
--> disp("Increasing powers division of p1 by p2 =",
pdiv(p1, p2));
"Increasing powers division of p1 by p2 ="
  -0.25 +0.5s
```

Exercise 4:

```
clc;
// Define p(s) and q(s)
--> s = poly(0, 's');
--> p = s^4 - 3*s^3 + 2*s - 1;
--> q = s^3 - s^2 + s - 1;
// 1. Degree of p
--> deg_p = degree(p);
--> disp("Degree of p =", deg_p);
"Degree of p ="
   4.
// 2. Derivative of p
--> dp = derivat(p);
--> disp("Derivative of p =", dp);
"Derivative of p ="
  2 -9s^2 +4s^3
// 3. Coefficients of p
--> coeff_p = coeff(p);
--> disp("Coefficients of p =", coeff_p);
"Coefficients of p ="
  -1.   2.   0.  -3.   1.
```

```
// 4. Variable name

--> vname = varn(p);

--> disp("Variable name =", vname);

"Variable name ="

  "s"

// 5. Factorize p and q

-> disp("Factorization of p =", factors(p));

  "Factorization of p ="

  (1) = -2.7889732+%s

  (2) = 0.8947036+%s

  (3) = 0.4007527-1.1057304*%s+%s^2

--> disp( "Factorization of q =", factors(q));

"Factorization of q ="

  (1) = 1+8.882D-16*%s+%s^2

  (2) = -1+%s

// 6. LCM and GCD

--> g = bezout(p, q);

--> l = (p * q) / g;

--> disp("GCD(p, q) =", g);

  "GCD(p, q) ="

               1

--> disp("LCM(p, q) =",l);
```

```
  "LCM(p, q) ="

   1 -3s +3s^2 -2s^4 +4s^5 -4s^6 +s^7

   ---------------------------------

                    1

// 7. Evaluate p(s) for s = 1.5

--> val = horner(p, 1.5);

--> disp("p(1.5) =", val);

  "p(1.5) ="

  -3.0625

// 8. Simplify rational fraction r = p/q

--> r = p / q;

--> r_simplified = simp(r);

-->   disp("Simplified   rational   fraction   r   =",
r_simplified);

  "Simplified rational fraction r ="

   -1 +2s -3s^3 +s^4

   -----------------

    -1 +s -s^2 +s^3
```

Chapter 6. Graphics in Scilab

Plotting and graphics are among the most effective ways to analyze and present data. Scilab offers powerful built-in tools for creating and customizing different types of plots such as 2D plots, contour plots, and 3D surface plots. In this chapter, we will learn how to generate these basic graphics and how to enhance them with titles, axis labels, and legends for clearer presentation.

1. Displaying 2D and 3D plots:

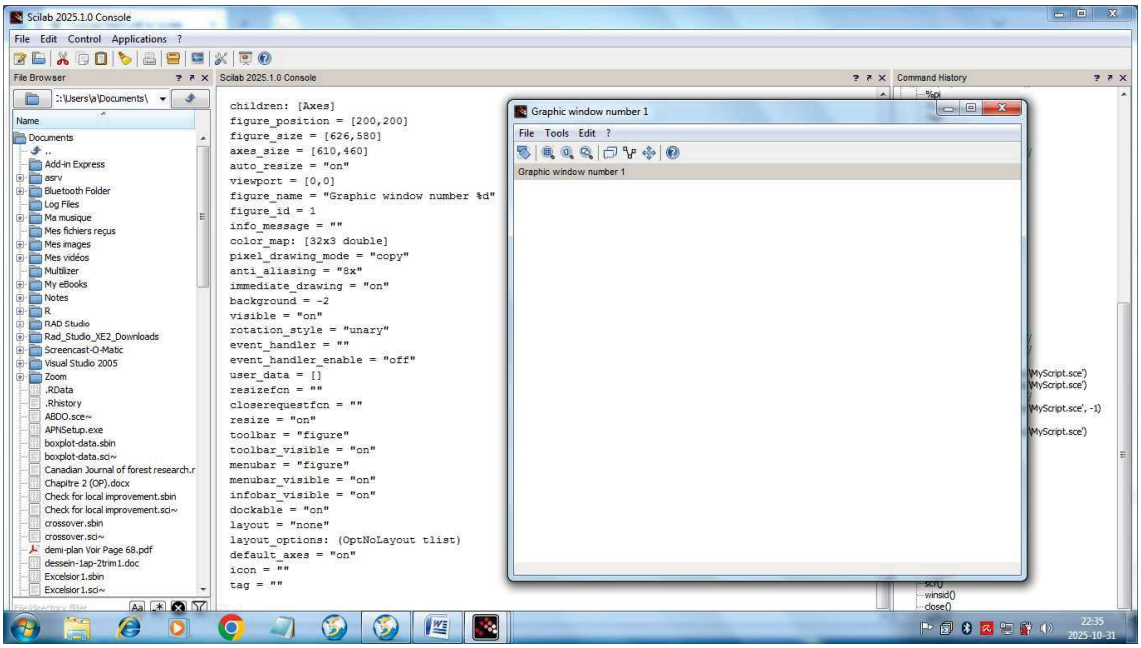
The graphics window opens automatically with plotting commands. To open one manually, use **`scf(number)`**:

```
-->scf(1)
```

or

```
--> figure(1).
```

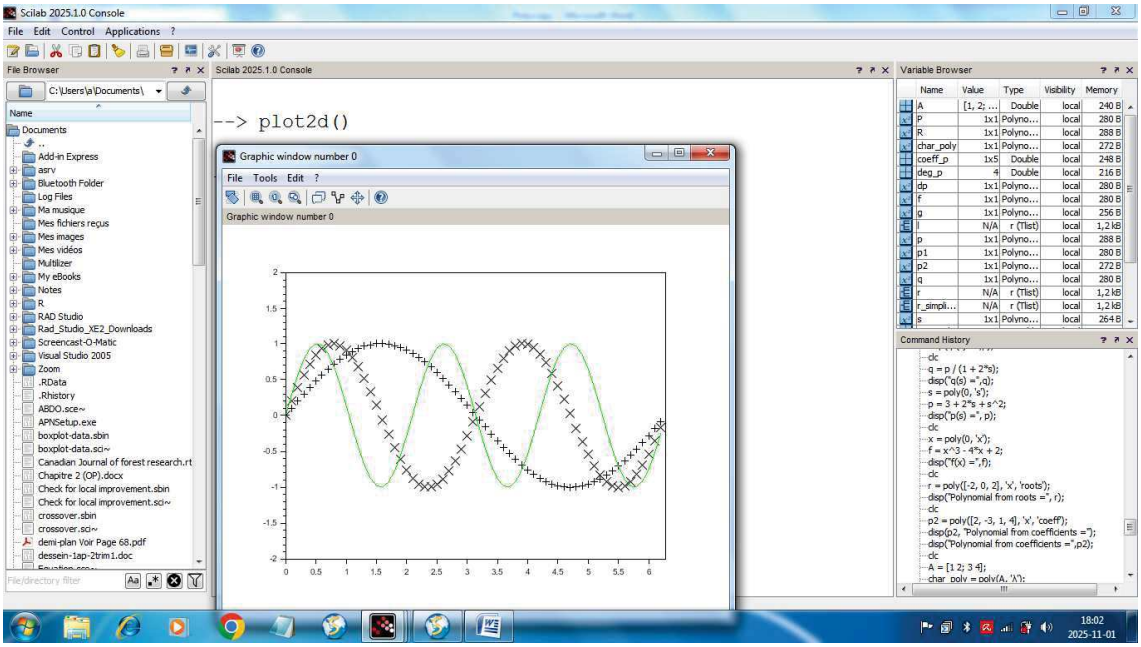
The graphic window looks like this:

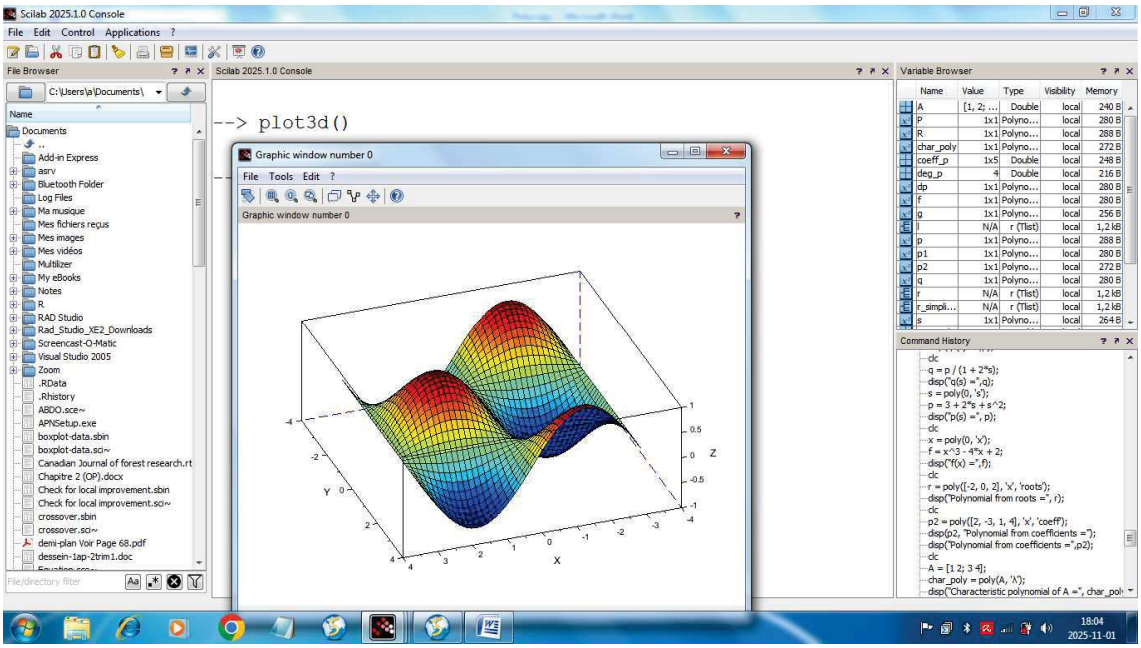


You can close the graphic window using:

```
--> close()
```

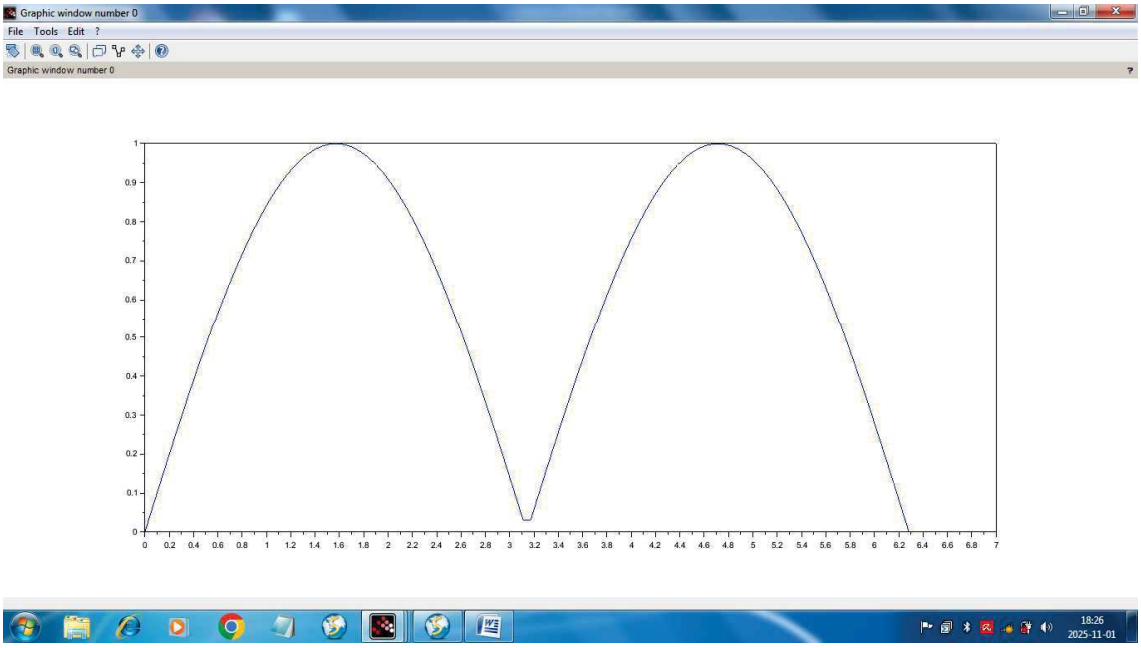
The **`scf(n)`** function, which stands for *Set Current Figure*, selects the graphics window with the number **`n`** as the current one. When you execute plotting commands such as **`plot()`** or **`plot3d()`**, Scilab sends the results to this active window. The following two graphics windows display 2D and 3D plots, respectively.



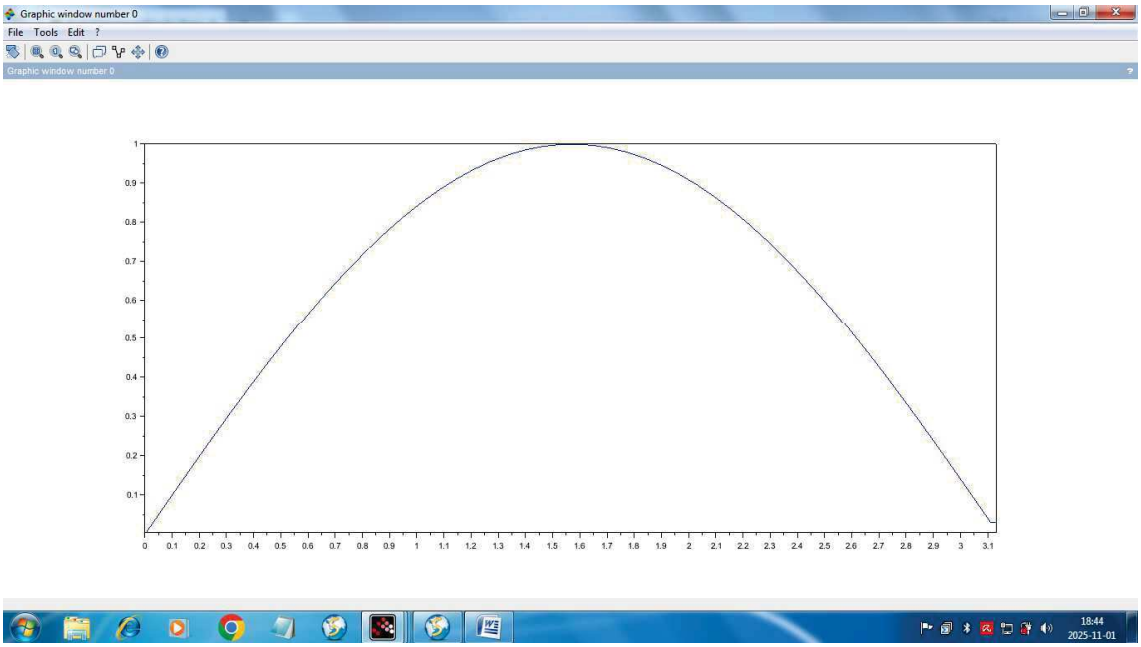


2. Graphs of functions:

```
--> function y=f(x), y=abs(sin(x)), endfunction

--> plot(linspace(0,2*%pi,100),f)
```



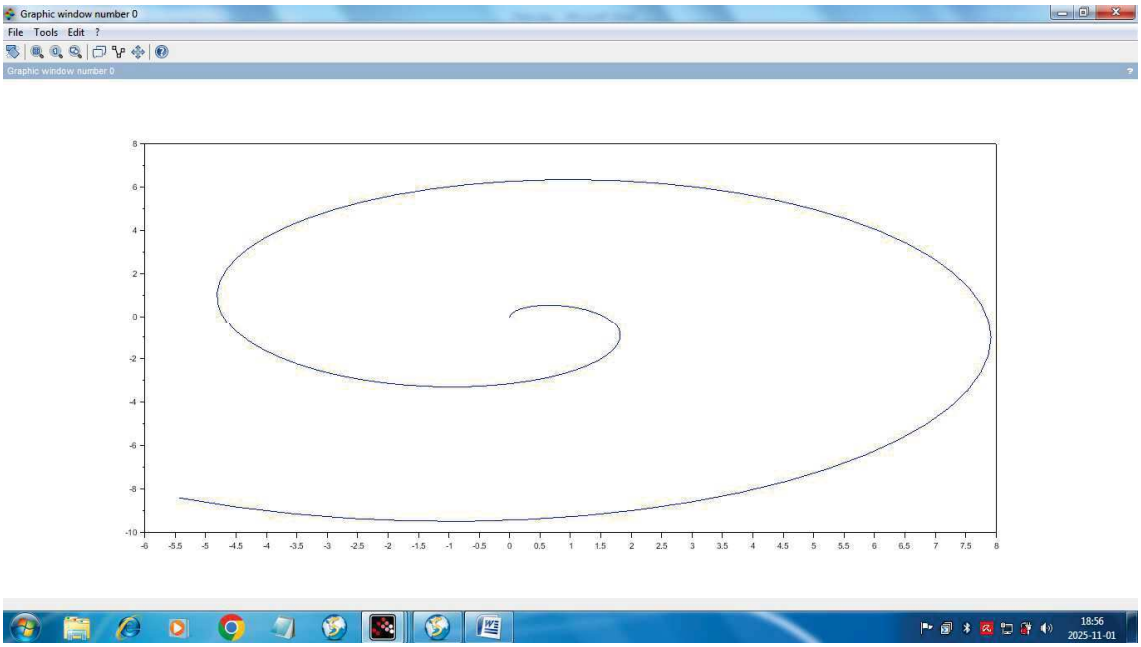
This graph and the following ones display only the contents of the drawing area. The **Tools** menu allows you to view details. Simply click on this menu, then select **Zoom** and choose a point in the window and drag the mouse to define the rectangle to zoom in on. This rectangle will then be displayed using the entire window. The figure below shows the graph obtained by zooming in on a region of the previous plot:



It is possible to repeat the operation to view more and more details. The "Original view" option in the "Tools" menu restores the original graph.

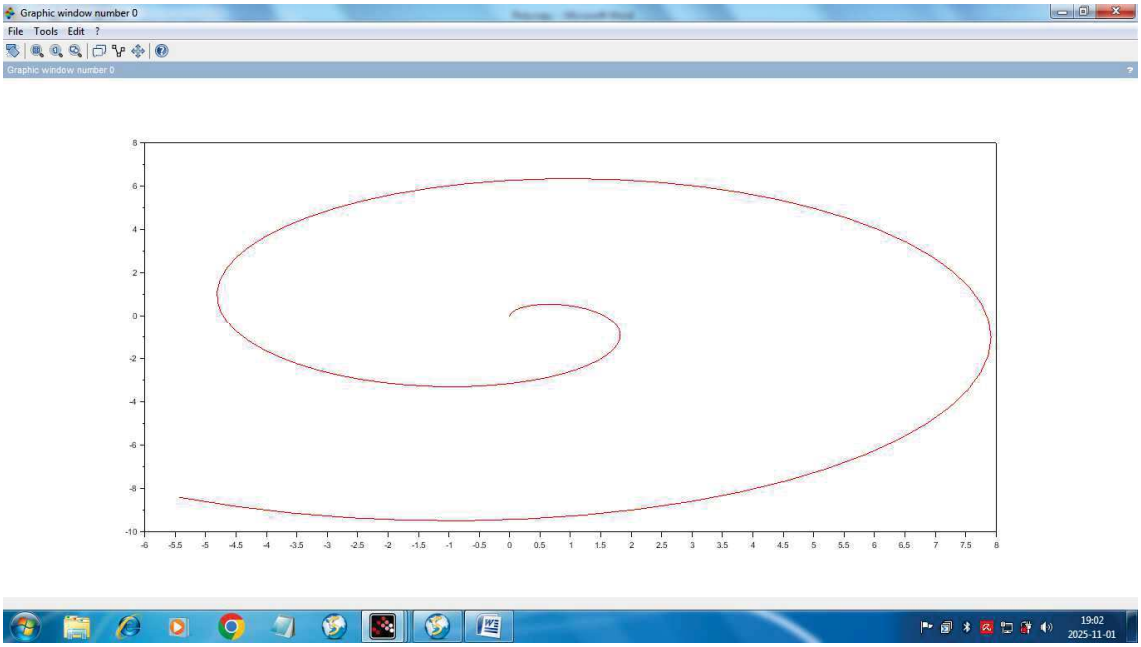
Very often, a function is represented by a set of data points **(x(i), y(i))**. In this case, the plot function can still be used.

```
--> p=linspace(0,10,100);

--> x=p.*sin(p);

--> y=p.*cos(p);

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

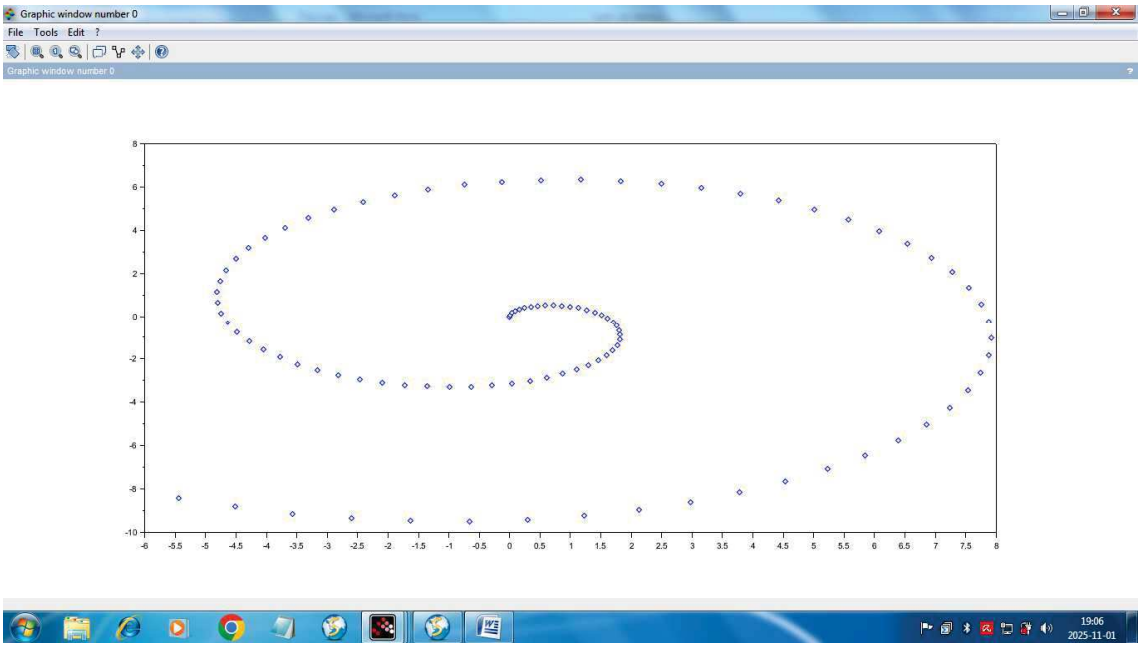


Many options can be specified through the arguments of the **`plot`** function (see the help on **`LineSpec`**).

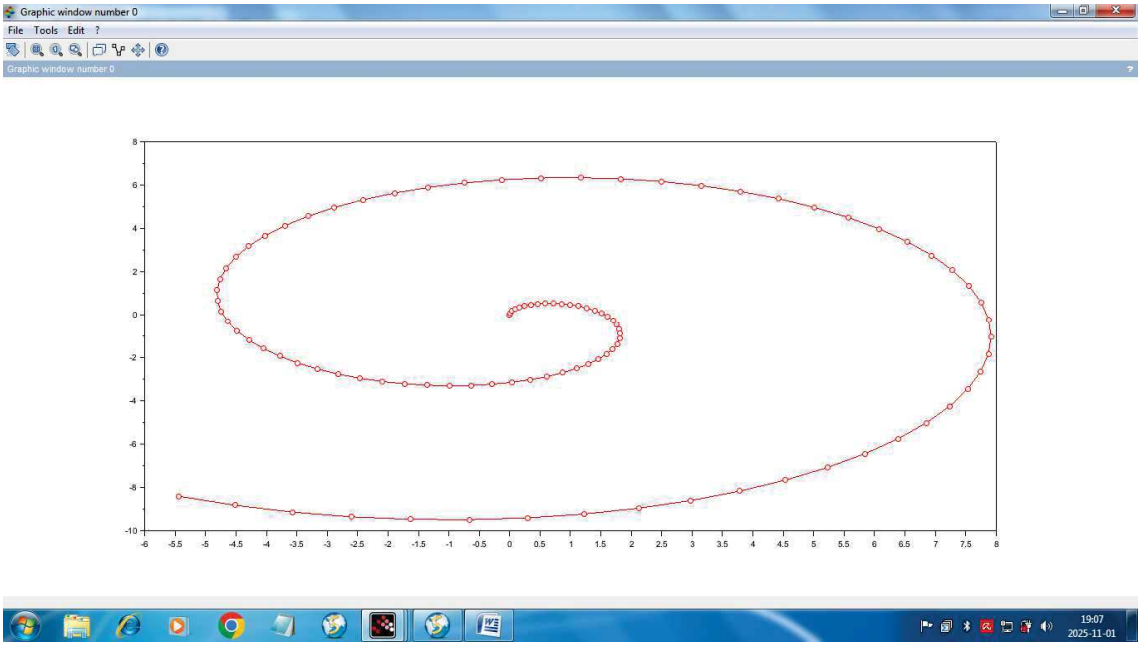
```
--> clf(), plot(x,y,'r')
```




```
--> clf(), plot(x,y,'d')
```

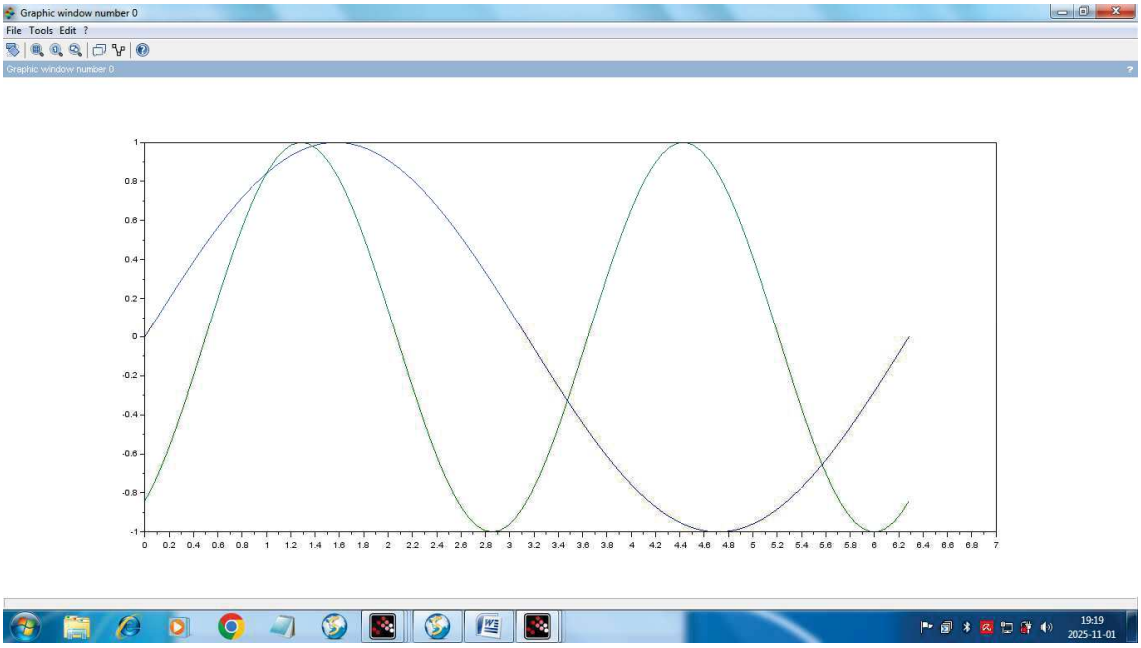



```
--> clf(), plot(x,y,'r-o')
```



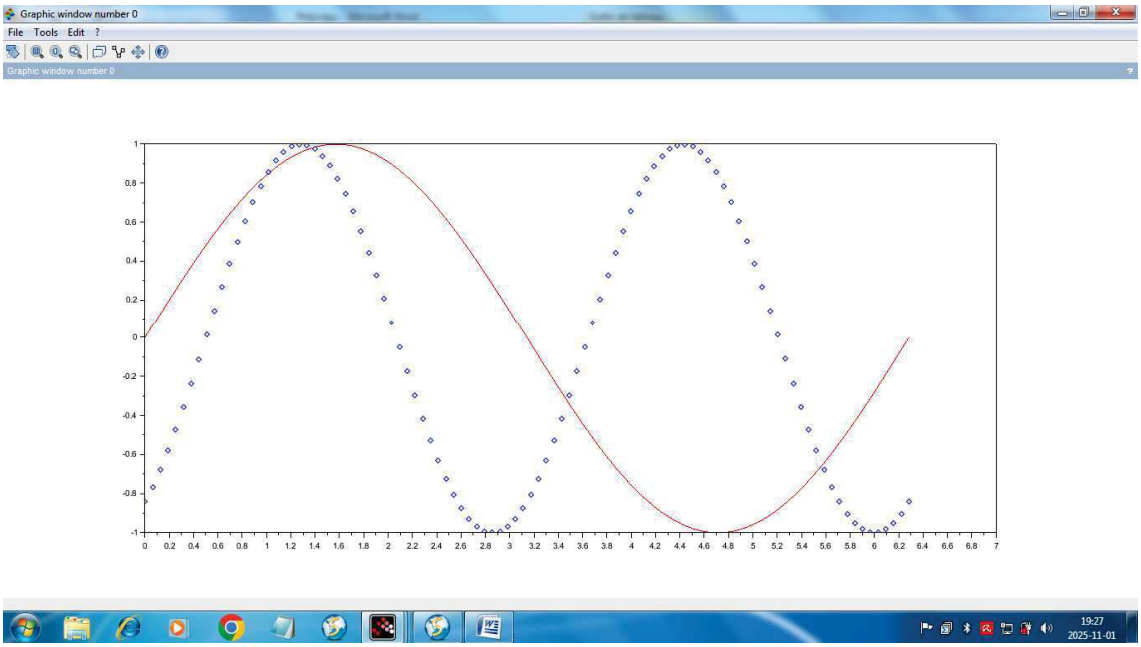
The user often wants to draw several curves in the same window to compare their behavior. Two different situations may occur: all the y-data correspond to the same discretization of the x-values.

```
--> x=linspace(0,2*%pi,100);
--> y=[sin(x)',sin(2*x-1)'];
--> clf(); plot(x,y)
```



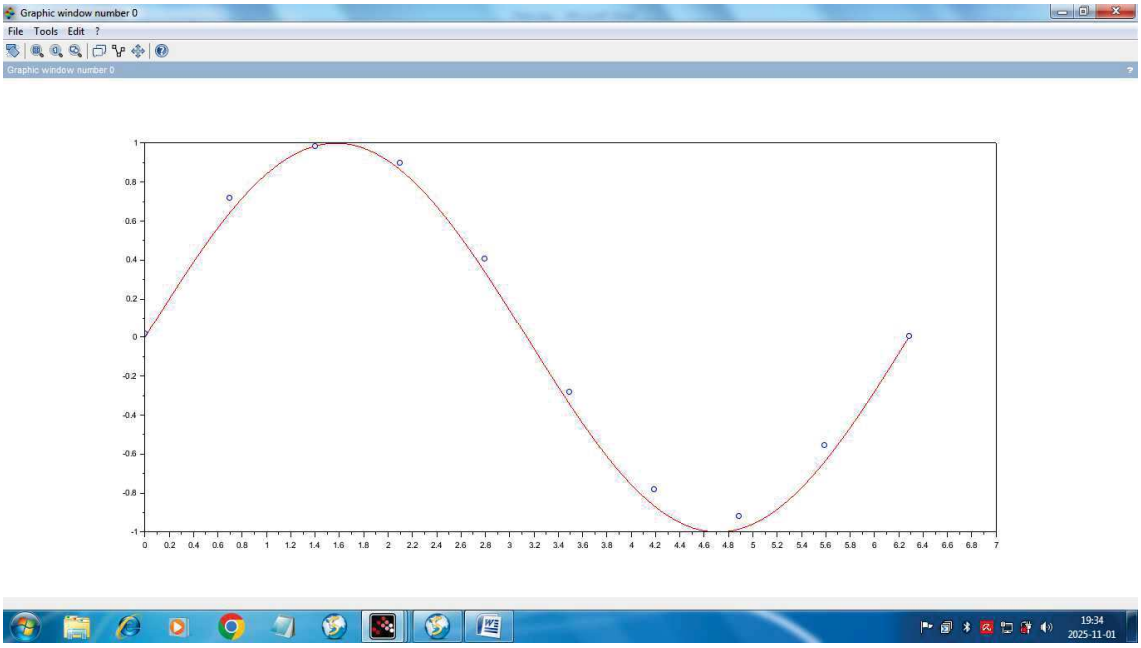
Note that each curve corresponds to a column of the matrix *y*. Each curve is drawn with a different color. It is also possible to pass several curves as arguments to the same **`plot`** function.

```
--> clf(); plot(x,sin(x),'r',x,sin(2*x-1),'d')
```



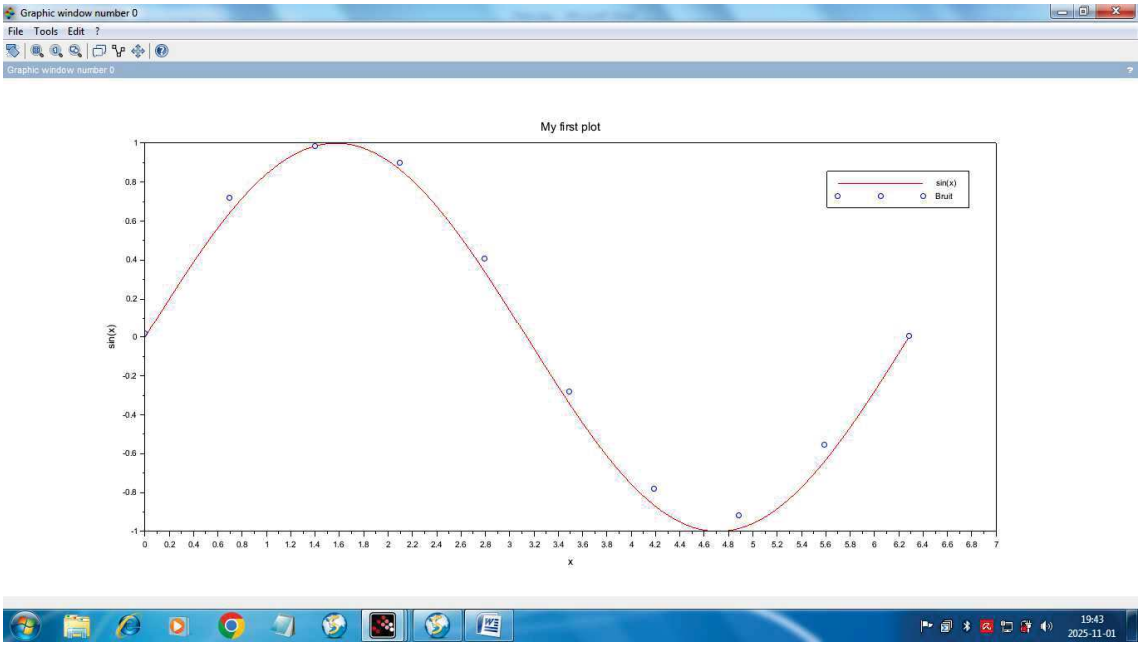
If each function has its own discretization of the x-values, the **`plot`** function can then be called several times.

```
--> x1=linspace(0,2*%pi,100);
--> x2=linspace(0,2*%pi,10);
--> clf();
--> plot(x1,sin(x1),'r')
--> plot(x2,sin(x2)+rand(x2)/10,'O')
```



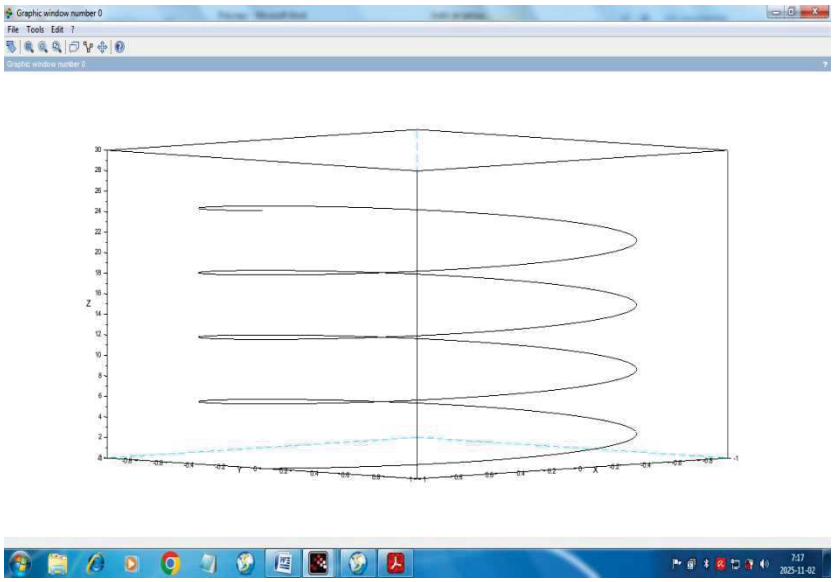
The following instructions allow you to add a legend (positioned by the user), a title, and axis labels to the graph.

```
--> legend(['sin(x)','Bruit'],5)
--> xtitle('My first plot','x','sin(x)')
```



It is, for example, possible to plot curves in 3D using the **param3d** command, such as a helix:

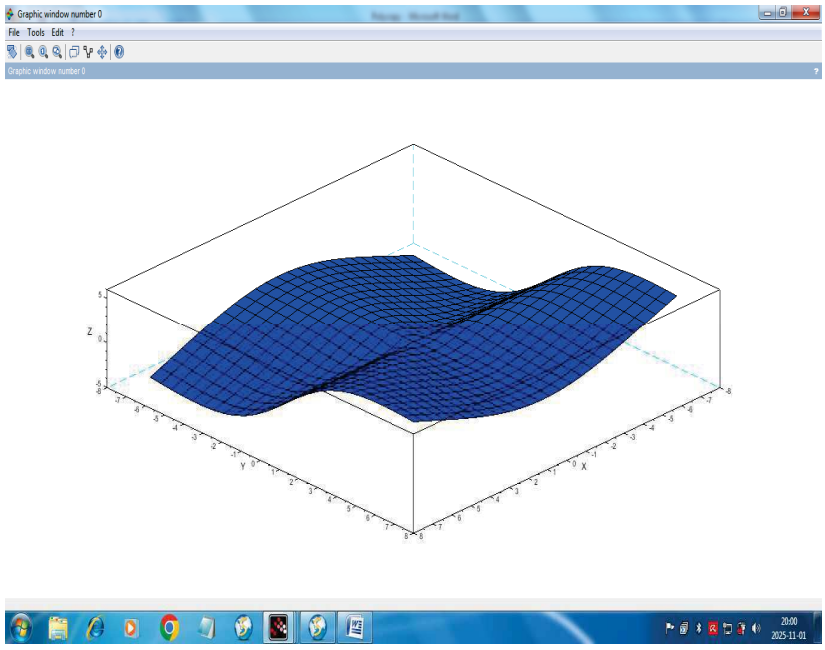
```
--> clf
--> t = 0:%pi/32:8*%pi;
--> param3d(cos(t), sin(t), t)
```



3. Analytical surfaces:

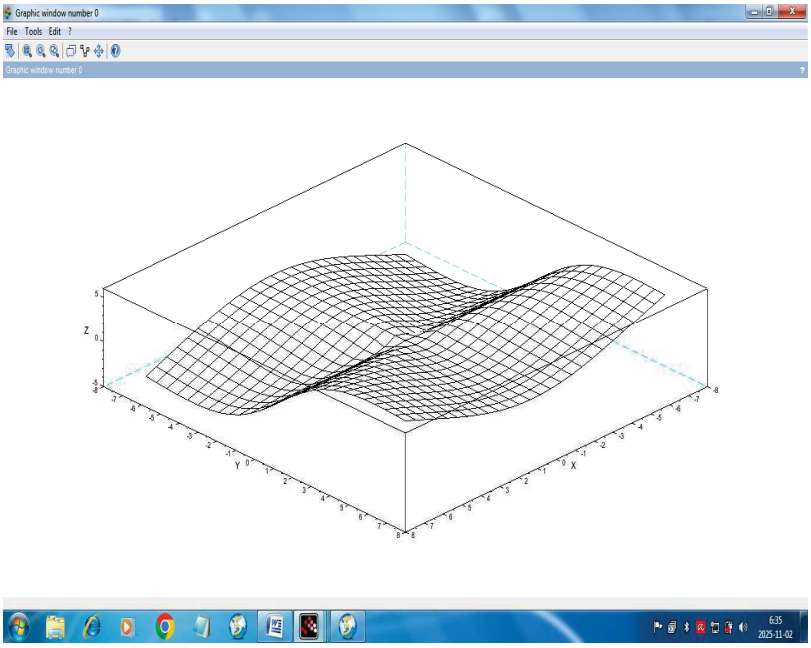
Let us first draw a surface defined by a function **z = F(x,y)**.

```
--> function z=F(x,y),...
  > z=(2*x^2*y+y^2)/(x^2+2*y^2), endfunction
--> x=-7:0.55:7;
--> clf();fplot3d(x,x,F)
```



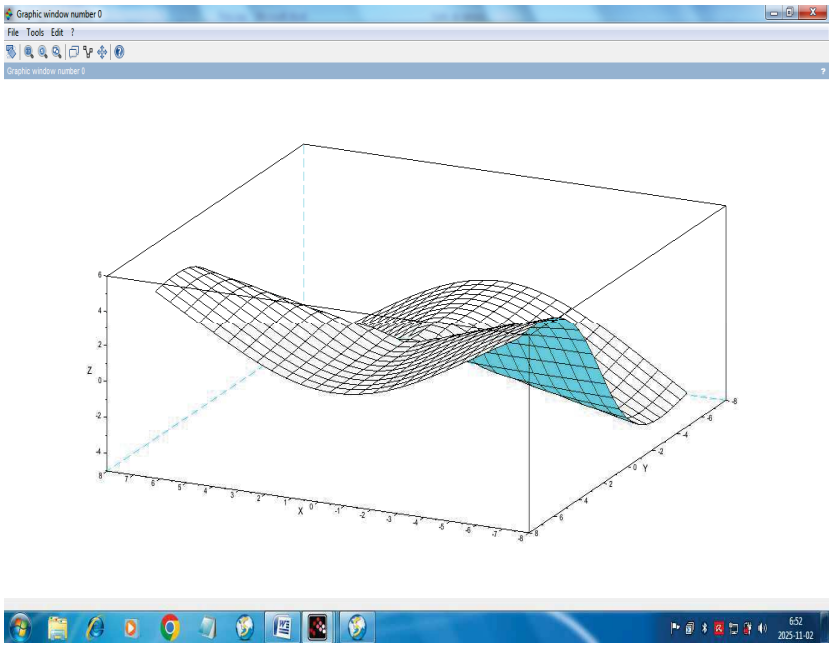
The visible surface is, by default, displayed in color number 2 from the color table. This setting can be changed through the **color_mode** property.

```
-->gce().color_mode=8
```



The **2D/3D Rotation** feature in the **Tools** menu enables adjustment of the viewing angles **θ** and **α**. The same result can be obtained by providing the optional parameters θ (theta) and α (alpha).

```
-->fplot3d(x,x,F,theta=115,alpha=63)
```



Alternatively, a surface can be represented using two vectors, **x** and **y**, specifying the discretization along the **x**- and **y**-axes, and a matrix **z**, where **z(i,j)** denotes the elevation at the point (**x(i), y(j)**). The surface corresponding to the function **F** below can then be constructed by computing the matrix **z** as follows:

```
--> z=feval(x,x,F);
--> plot3d(x,x,z)
```

The surface, represented in this manner, can be visualized using the **surf** function.

```
--> surf(x,x,z)
```

Laboratory Work 6

Exercise 1:

1. Draw the curve of the function **y(x) = xsin(x)** on the interval **[0,10]** with a step of **0.05**, specifying only the essential parameters.
2. Plot the same curve three times in succession:
   - in green
   - with markers
   - a red dashed line.
3. Superimpose the curves **y(x) = x** and **y(x) = xsin(x)** using one or two successive calls to the plot function.
4. Use the zoom tool to closely inspect the points of tangency.
5. Add a legend to the most recent plot.

Exercise 2:

1. Plot the surface defined by $\mathbf{z(x,y)=x^4-y^4}$ for **x=−3:0.3:3** and **y=x** using the **`fplot3d`** command. Then, use the **2D/3D Rotation** option from the **Tools** menu to adjust the viewing angle.
2. Calculate the matrix **Z**, where **Z(i,j)** corresponds to **Z(x(i), y(j))**, using the **`feval`** function. Next, plot the surface again with the **`surf`** command.

Exercise 3:

Découper la fenêtre graphique en deux sous-fenêtres identiques et reafficher les dessins D1 et D2 des deux exercices précédents. On pourra utiliser la **`subplot`**.

Solution of Laboratory Work 6

Exercise 1:

```
// 1. Draw the curve y(x) = x*sin(x) on [0,10] with
step 0.05

x = 0:0.05:10;

y = x .* sin(x);

plot(x, y);

xtitle("Plot of y(x) = x sin(x)", "x", "y");

// 2. Plot the same curve three times in succession

clf(); // Clear the graphics window

x = 0:0.05:10;

y = x .* sin(x);

// a) in green

plot(x, y, 'g');

xtitle("Same curve in different styles", "x", "y");

xgrid();

sleep(1000); // short pause before next plot

// b) with markers

clf();

plot(x, y, 'o');
```

```
xtitle("Curve with markers", "x", "y");

xgrid();

sleep(1000);

// c) red dashed line

clf();

plot(x, y, 'r--');

xtitle("Red dashed curve", "x", "y");

xgrid();

// 3. Superimpose the curves y = x and y = x*sin(x)

clf();

y1 = x;

y2 = x .* sin(x);

plot(x, y1, 'b', x, y2, 'r--');

xtitle("Superimposed Curves", "x", "y");

xgrid();

// 4. Use the zoom tool manually from the Tools menu

// (no code needed — user action)

// 5. Add a legend to the most recent plot

legend(["y = x", "y = x sin(x)"], "in_upper_left");
```

Exercise 2:

```
// 1. Plot the surface z(x,y) = x^4 - y^4 using
fplot3d
deff('z = F(x,y)', 'z = x.^4 - y.^4'); // Define the
function
x = -3:0.3:3;
```

```
y = x;
clf(); // Clear current figure
fplot3d(x, y, F);
xtitle("Surface z(x,y) = x^4 - y^4 using fplot3d",
"x", "y", "z");
// Use Tools • 2D/3D Rotation to change the viewing
angle manually
// 2. Compute the matrix Z using feval and redraw the
surface with surf
[X, Y] = ndgrid(x, y);   // Create the coordinate
grids
Z = feval(X, Y, F);      // Evaluate z = F(x,y) at all
points
fplot3d(X, Y, F);
clf(); // Clear the window before redrawing
surf(x, y, Z);
xtitle("Surface z(x,y) = x^4 - y^4 using surf", "x",
"y", "z");
xgrid();
```

Exercise 3:

```
// Divide the window into two sub-windows to display
D1 and D2

clf();

// --- Left subplot: D1 (superimposed curves from
Exercise 1) ---
subplot(1,2,1);
x1 = 0:0.05:10;
y1 = x1;
y2 = x1 .* sin(x1);
plot(x1, y1, 'b', x1, y2, 'r--');
xtitle("D1: y = x and y = x sin(x)", "x", "y");
```

```
xgrid();
legend(["y = x", "y = x sin(x)"], "in_upper_left");

// --- Right subplot: D2 (surface from Exercise 2) ---
subplot(1,2,2);
x2 = -3:0.3:3;
y2 = x2;
deff('z = F2(x,y)', 'z = x.^4 - y.^4');
Z2 = feval(X2, Y2, F2);
fplot3d(x2, y2, Z2);
xtitle("D2: Surface z(x,y) = x^4 - y^4", "x", "y",
"z");
xgrid();
```

Chapter 7. Symbolic computation

Symbolic computation allows exact manipulation of mathematical expressions, such as polynomials, equations, derivatives, integrals, and series. Unlike numerical computation, which gives approximate values, symbolic computation preserves the algebraic form of expressions. Scilab can perform symbolic calculations by integrating **Maxima**, a well-established computer algebra system, through the free **Symbolic Toolbox for Scilab**, implemented by **Jean-Francois Magni (2006)** and available at https://github.com/sengupta/scilab-maxima.git. Most functions of this toolbox have a syntax similar to the **Matlab Symbolic Toolbox**, although the functionality of this toolbox is limited to the functionality available in **Maxima**. This chapter is a practical guide to using **Symbolic** computation capabilities in Scilab.

1. Using the SciMax Symbolic Toolbox:

1.1. Symbolic Objects:

The Symbolic Toolbox defines a new Scilab data type called a **symbolic object**. Internally, a symbolic object is a data structure that stores a string representation of a symbol. The toolbox uses symbolic objects to represent symbolic variables, expressions, and matrices. Actual computations involving symbolic objects are primarily performed by **Maxima**, an open-source system for symbolic and numerical mathematics. Maxima was originally developed at the Massachusetts Institute of Technology as part of the **Macsyma** project in the 1960s. It was released under the **GNU** General Public License (**GPL**) in **1998** and continues to be actively maintained by a community of users and developers.

1.2 How to declare symbolic variables and handle symbolic expressions:

To create a symbolic expression that is a constant, you must use the **sym** command. For example, to convert the number **5** into a symbolic constant, enter:

```
--> f = sym('5')
  f =
  5
--> a=sqrt(f)
  a =
  5^(1/2)
```

Use the command **syms** to declare variables as symbolic. For example, the commands:

```
--> syms x y z
```

declare **x**, **y**, and **z** as symbols, not numbers. This allows you to create expressions like:

```
--> expr = x^2 + 3*y - z
 expr  =
z - 3*y + x^2
```

To substitute a symbolic variable with a value, you must use the **subs** command. For example:

```
--> subs(expr, x, 2)
ans=
4 + 3*y - z
```

To simplify an expression, use **simple** command. For example:

```
--> syms a b;
```

```
--> f = (a + b)^2 - (a - b)^2;
--> simple(f)
ans =
4*a*b
```

2. **Expanding and Functionalizing an Expression:**

The **expand** function develops expressions into their standard form. For example:

```
--> syms x;
--> f = (x + 2)^3;
-->expand(f)
ans=
x^3 + 6*x^2 + 12*x + 8
```

With a trigonometric expansion example:

```
-->syms x y;
-->f = sin(x + y);
--> expand(f)
ans =
 sin(x)*cos(y) + cos(x)*sin(y)
```

3. **Derivatives and Integrals of a Function:**

The **diff** function computes derivatives:

```
// First derivative
--> syms x
--> f = x^3 + 2*x^2 + x;
-->df = diff(f, x)
df =
  3*x^2 + 4*x + 1
```

```
//Higher-order derivatives:
Df2=diff(f, x, 2)  // Second derivative
df2=
     6*x + 4
//Partial Derivatives:
--> syms x y
-->f = x^2*y + sin(y);
-->dx=diff(f, x)//Partial derivative with respect to x:
dx=
   2*x*y
-->dy=diff(f, y)// Partial derivative with respect to y:
dy=
    x^2 + cos(y)
```

The **integ** function is used to integrate a symbolic expression:

```
// Indefinite integral:
--> syms x
--> f = x^2;
-->integ(f, x)
ans=
   x^3 / 3
// Definite integral:
--> integ(f, x, 0, 1)
ans=
1/3
```

4. Calculating the Taylor Expansion of a Function:

The **taylor** function for Taylor expansion. The syntax of this function is as follow :

```
--> taylor(f, x, x0, n) //  "f": function to expand,
"x": variable,  "x0": point of expansion,  "n": order.
```

For example:

```
// sin(x) around 0 up to order 5
-->syms x
-->taylor(sin(x), x, 0, 5)
ans=
 x - x^3/6 + x^5/120
// log(1 + x) around x = 0 up to order 4
-->syms x
-->taylor(log(1 + x), x, 0, 4)
ans=
    x - x^2/2 + x^3/3 - x^4/4
```

Laboratory Work 7

Exercise 1:

1. Define symbolic variables **u**, **v** and form the expression **(u + v)^2**.

2. Evaluate the expression **x^2 + 2*x + 1** for **x = -1**.

3. Simplify the expression **(x + 1)^2 - x^2 - 2*x - 1**.

Exercise 2:

1. Expand the expression **(x - 3)^2 * (x + 2)**.

2. Create a symbolic function **g(x) = x^4 - 2*x^2 + 1** and compute **g(1)** and **g(-1)**.

3. Replace **y** with **2*x** in the expression x + y^2.

Exercise 3:

1. Compute the first and second derivatives of **f(x) = x^3 - 4*x + 6**.

2. Find the partial derivatives of **f(x, y) = x*y^2 + log(x)**.

3. Evaluate the definite integral of **f(x) = 1/(1 + x^2)** from **0** to **1**.

Exercise 4:

1. Compute the 4th-order Taylor expansion of **cos(x)** around **x = 0**.

2. Compute the 5th-order Taylor series of **exp(x)** around **0** .

Solution of Laboratory Work 7

Exercise 1:

```
//Define symbolic variables "u", "v" and form the
//expression "(u + v)^2":
-->syms u v;

-->expr = (u + v)^2

expr =

u^2 + 2*u*v + v^2

// Evaluate the expression x^2 + 2*x + 1" for x = -1:

-->syms x;

-->expr = x^2 + 2*x + 1;

-->subs(expr, x, -1)

ans =

     0

// Simplify (x + 1)^2 - x^2 - 2*x - 1:

-->syms x

-->expr = (x + 1)^2 - x^2 - 2*x - 1;

-->simple(expr)

ans:

    0
```

Exercise 2:

```
// Expand (x - 3)^2 * (x + 2):

-->syms x

-->expr = (x - 3)^2 * (x + 2);

-->expand(expr)

ans:

 x^3 - 4*x^2 - 3*x + 18

//Define g(x) = x^4 - 2*x^2 + 1 and compute g(1) and
// g(-1)

-->syms x

-->g = x^4 - 2*x^2 + 1;

-->subs(g, x, 1)

ans:

 0

-->subst(g, x, -1)

ans:

 0

// Substitute y = 2*x in x + y^2:

-->syms x y

-->expr = x + y^2;

-->subs(expr, y, 2*x)

ans:

x + (2*x)^2 = x + 4*x^2
```

Exercise 3:

```
// Derivatives of  f(x) = x^3 - 4*x + 6:
-->syms x

-->f = x^3 - 4*x + 6;

-->diff(f, x)

ans:

3*x^2 - 4

-->diff(f, x, 2)

ans=

  6*x

//Partial derivatives of  f(x, y) = x*y^2 + log(x):

-->syms x y

-->f = x*y^2 + log(x);

-->diff(f, x)

ans;

y^2 + 1/x

-->diff(f, y)

ans=

2*x*y
```

// Evaluate $\int_0^1 \frac{1}{1+x^2} dx$:

```
-->syms x

-->integrate(1/(1 + x^2), x, 0, 1)

ans: %pi/4
```

Exercise 4:

```
// 1. 4th-order Taylor expansion of "cos(x)" around 0:
-->syms x

-->taylor(cos(x), x, 0, 4)

ans:

1 - x^2/2 + x^4/24

// Taylor expansion of exp(x)around 0 up to degree 5:

-->syms x

-->taylor(exp(x), x, 0, 5)

ans:

 1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120
```

Bibliography

1. Affouf, M. (2021). A guide to Scilab with applications. Lulu.
2. Allaire, G., & Kaber, S. M. (2002). Introduction à Scilab : Exercices pratiques corrigés d'algèbre linéaire. Paris : Ellipses.
3. Berrou, J.-M. (n.d.). TP d'informatique en ECS1 avec Scilab (E.D.E.N.). Retrieved from https://www.scilab.org/tp-dinformatique-en-ecs1-avec-scilab-eden
4. Bunks, C., Chancelier, J.-P., Delebecque, F., Gomez, C., Goursat, M., Nikoukhah, R., & Steer, S. (1999). Engineering and scientific computing with Scilab. Boston : Birkhäuser.
5. Campbell, S. L., Chancelier, J.-P., & Nikoukhah, R. (2001). Modeling and simulation in Scilab/Scicos. Springer.
6. Chancelier, J.-P., Delebecque, F., Gomez, C., Goursat, M., Nikoukhah, R., & Steer, S. (2001). Introduction à Scilab (1re éd.). Paris : Springer.
7. Chouhan, D. (2022). Basic programming concepts of Scilab. Orange Books Publication.
8. Goyal, R., & Dhingra, M. (2019). Programming in Scilab. Alpha Science International.
9. Magni, J.-F. (2006). Symbolic Toolbox for Scilab [Computer software]. GitHub. https://github.com/sengupta/scilab-maxima.git
10. Mottelet, S. (2018, février). Introduction à Scilab. Université de Technologie de Compiègne, Laboratoire TIMR – EA 4297.
11. Nagar, S. (2017). Introduction to Scilab: For engineers and scientists. Apress.
12. Ramchandran, H., & Nair, A. S. (2011). Scilab (A free software to MATLAB). Springer.

13. Roux, P. (2016). Scilab : I. Fundamentals – From theory to practice. Éditions D-BookeR.
14. Rouxph. (2011, June 15). SciLab / Symbolic toolbox? Message posted to Comp.soft-sys.math.scilab mailing list.
15. http://lists.scilab.org/pipermail/users/2011-June/004395.html
16. Scilab Enterprises. (2013). Scilab pour l'enseignement des mathématiques [PDF]. https://www.scilab.org/scilab-pour-l%E2%80%99enseignement-des-math%C3%A9matiques-french
17. Scilab Enterprises. (2025). Scilab Online Help (Version 2025.1.0) – English. https://help.scilab.org/docs/2025.1.0/en_US/
18. Urroz, G. E. (2001). Numerical and statistical methods with Scilab for science and engineering (Vols. 1–2). Springer.
19. Verma, A. K. (2018). Scilab — A beginner's approach. Cengage India.
20. Vrillon, L. (2015). Mathématique & Informatique. Retrieved from https://www.scilab.org/math%C3%A9matique-informatique
21. Wikipedia contributors. (n.d.). IEEE 754. In Wikipedia, The Free Encyclopedia. Retrieved November 5, 2025, from https://en.wikipedia.org/wiki/IEEE_754