الجمهورية الجزائرية الديمقراطية الشعبية
PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
وزارة التعليم العالي و البحث العلمي
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
جامعة محمد بوضياف بالمسيلة
MOHAMED BOUDIAF UNIVERSITY OF M'SILA
كلية التكنولوجيا
FACULTY OF TECHNOLOGY
قسم الهندسة الكهربائية
DEPARTMENT OF ELECTRICAL ENGINEERING

رقم:...438../ق هـ ك/2024 المسيلة في : 01 ديسمبر 2024

شهادة إدارية

بخصوص مطبوعة الدروس الخاصة بالأستاذ

روباش توفيق

بناءً على محضر اللجنة العلمية لقسم الهندسة الكهربائية تحت رقم: 365/ ق.هـ.ك/2024المنعقد بتاريخ 06نوفمبر2024 والمتضمن تعيين الخبراء:الأستاذ غضبان إسماعيل أستاذ محاضر -أ- بجامعة المسيلة ،الأستاذ زغلاش سمير أستاذ بجامعة المسيلة والأستاذ بن صافية ياسين أستاذ بجامعة البويرة وذلك لتقييم مطبوعة الدروس الخاصة بالأستاذ روباش توفيق أستاذ محاضر "أ" بقسم الهندسة الكهربائية لجامعة المسيلة تحت عنوان: "**Logiciels de simulation**" مطبوعة دروس مكتوبة باللغة الإنجليزية تحت عنوان "**Softwares of Simulation**" وبعد إطلاع رئيس اللجنة العلمية ورئيس القسم على التقارير الواردة و التي كانت كلها ايجابية، وعليه فإن اللجنة لا ترى مانعا أن تتخذه سندا في تدريس طلبة السنة الثالثة ليسانس كهروتقني، شعبة كهروتقني ،ميدان علوم و تكنولوجيا و أن تعتمد في أي تقييم للمسار العلمي للأستاذ المعني.

رئيس اللجنة العلمية
أ .د بوقرة عبد الرحمان

رئيس القسم
دنداف المبروك

ملاحظة: سلمت هذه الشهادة للمعني(ة) لاستعمالها في حدود ما يسمح به القانون.

Abstract

This polycopy of courses is an introduction to softwares of simulation. It allows studying the necessary programming tools using the MATLAB language, scientific calculation software. Moreover, it aims to prepare the student for practical work in Modeling of electrical machines, Power Electronics and Numerical Analysis in which this tool is intensively used for the implementation and simulation of the theoretical principles presented in class. In addition, this manual offers the student the opportunity to train in other widely used professional softwares, namely: PSpice, PSim,....

Keywords: Simulation software, MATLAB, SIMULINK, PSpice

DEMOCRATIC AND POPULAR ALGERIAN REPUBLIC
وزارة التعليم العالي والبحث العلمي
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

University of Mohamed Boudiaf – M'sila
Faculty of Technology
Department of Electrical Engineering

جامعة محمد بوضياف - المسيلة
كلية التكنولوجيا
قسم الهندسة الكهربائية

Polycopy of Courses

Softwares of Simulation

UET 3.1
Intended for 3rd year undergraduate students
Specialty: Electrotechnics

Directed by:
Dr. Toufik ROUBACHE

Course: Softwares of Simulation

Semester 5:
Teaching unit: UET 3.1
VHS: 22h30 (Course: 1h30)
Credits: 1
Coefficient: 1

Teaching objective

The objective of the course is to know different softwares of simulation, and then be able to reproduce an electro-energetic system for study and simulation.

Recommended and useful knowledge:

- ✓ Programming concepts
- ✓ Matlab concepts.

Outline

Chapter 1: Getting started with MATLAB

1.1 - Introduction

1.2 - MATLAB environment

1.3 - Starting MATLAB

Command window, Defined variables window (the workspace), Directory window Workstation, Command History Window

1.4 - Presentation and generalities

Get help, Getting started, The workspace, Syntax of a line of instructions, Management files in the working directory, Arithmetic operations, Operations and functions relating to scalars, Special variables and constants, Format of numbers and precision of calculations, Order History

Chapter 2: Data types and variables

2.1 - Data types
2.2 - Variables
Complex numbers, Boolean variables, Character strings, Vectors, Matrices, Polynomials.

Chapter 3: The graphics

3.1 - Management of graphic windows

3.2 - 2D graphic representation

Graphs in Cartesian coordinates, Improve the readability of a figure, Graphs in polar coordinates, Diagrams.

3.3 - 3D graphics

3D Curves, Surfaces

Chapter 4: Programming in MATLAB

4.1 - Arithmetic, logical operators and special characters

4.2 - M-Files

4.3 - Scripts and functions

(Scripts, Functions)

4.4 - Control instructions

(FOR loop, WHILE loop, Conditioned IF statements)

Chapter 5: Getting started with SIMULINK

5.1 - SIMULINK libraries

Libraries Sources, Sinks, Continuous, Math Operations, Commonly Used Blocks, Signal Routing, Logic and Bit Operations, User-Defined Functions, Ports & Subsystems,…….

5.2- Quick start

5.3 - Masks and subsystems

5.3.1 - Subsystems

5.3.2 - Masking of subsystems

Hiding the subsystem, Using Callbacks

5.4 - Study of some simulation examples

Chapter 6: Power System Blockset (PSB)

6.1 - Presentation of the Power System Blockset

6.2 - Study of a simulation example

Chapter 7: Simulation and co-simulation with other software

7.1 - Simulation by PSim and Simulink-PSim co-simulation

7.2 - Simulation with other software: PSpice, Proteus, Scilab,….

Evaluation method:

Exam: 100%.

Tables of contents

Chapter 1: Getting started with MATLAB

Chapter 2: Data types and variables

Chapter 3: The graphics

Chapter 4: Programming in MATLAB

Chapter 5: Getting started with SIMULINK

Chapter 6: Power System Blockset (PSB)

Chapter 7: Simulation and co-simulation with other softwares

Chapter 1: Getting started with MATLAB

1.1– Introduction

MATLAB is an abbreviation of Matrix LABoratory. Originally written in Fortran by C. Moler, MATLAB was intended to facilitate access to the matrix software developed in the LINPACK and EISPACK projects. The current version, written in C by MathWorks Inc., exists in a professional version and a student version. Its availability is ensured on several platforms: Sun, Bull, HP, IBM, PC compatible (DOS, Unix or Windows), Macintoch, iMac and several parallel machines [1-3]. There are two operating modes [4]:

- Interactive mode: MATLAB executes instructions as they are given by the user.
- Executive mode: MATLAB executes a ".m" file (program in MATLAB language) line by line.

Generally speaking, Matlab is used to do computational experiments very quickly.

In this chapter we present the Matlab environment, then the following syntax elements:

- Starting MATLAB

- Presentation and general information

1.2 - MATLAB environment

MATLAB is a complete, open and extensible environment for calculation and visualization. As in Figure 1.1, the main interface is broken down into the following elements:

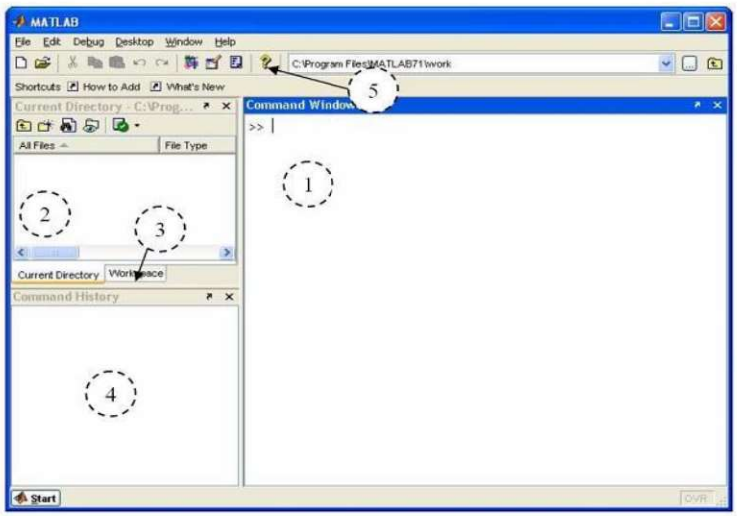


Figure 1.1 *MATLAB environment*

A. Command window: Allows you to execute commands outside of the program and displays the results.

Example: Tap 1+3

B. Current Directory: Content of the current directory where your programs should be located.

C. Workspace: Displays all the variables used.

D. History command: Allows you to view the last commands executed. It is possible to drag these commands to the command window. You can also access it by pressing the up arrow or for older commands by typing the first letter of the expression then up arrow.

E. Choice of current directory: This is the folder where your programs should be located (*.m files). You can put your programs in another folder but in this case you must include it in File >> Set Path

1.3 - Start, exit

To start running MATLAB:

- ✓ under Windows, you must click on Startup, then Program, then MATLAB,
- ✓ under other systems, refer to the installation manual.

The MATLAB '>>' prompt should then appear in the command window, following which the commands will be entered.

The "quit" function allows you to exit MATLAB:

>>quit

1.4 - Help, online documentation

Online help can be obtained directly in the session by typing help command or function name. The "help" command allows you to provide help on a given problem.

Example:

>> help cos

COS Cosine.

COS(X) is the cosine of the elements of X.

-List of help commands:

help ⟶ gives help on a function or toolkit (help)
helpdesk ⟶ hypertext documentation (requires Netscape or other)
helpwin ⟶ online help in a separate window.
lookfor ⟶ search for a keyword (slow).
which ⟶ locates functions and files.
what ⟶ list of Matlab files in the current directory.
exist ⟶ check if a function or variable exists in the workspace.
who, whos ⟶ list of variables in the workspace.

```
>> who          % provides the list of functions defined in the workspace
>> whos         % gives more information than who
```

1.5 - Presentation and generalities

MATLAB allows interactive work either in command mode or in programming mode; while still having the possibility of making graphical visualizations. Considered one of the best programming languages, it has the following particularities compared to these languages:

- Easy programming,
- Continuity among whole, real and complex values,
- The extended range of numbers and their precisions,
- The very comprehensive mathematical library,
- The graphical tool which includes graphical interface functions and utilities,
- The possibility of linking with other classic programming languages.

1.5.1- Calculations in Command mode

Let's start with the most common operators: +, -, *, /, ^. The last one means “power”. Parentheses are used in the classic way. We have everything to make a first calculation:

Type any mathematical expression and press <Enter>. For example:

```
>> (3*2)/(5+3)
ans =
0.7500
```

The result is automatically put into a variable called ans (answer). This can be used for the following calculation, for example:

```
>> ans*2

ans =
1.5000
```

Then you will notice that the result is displayed with 5 significant figures, which does not mean that the calculations are made with such low precision. If you want to display numbers more precisely, type the long format command. To return to the initial behavior: short format.

1.5.2- Command history

All the commands you have taped in MATLAB can be found and edited using the arrow keys. Press ↑ to go back through previous commands, ↓ to go back down, and use → and ← to edit a command. To restart a command, there is no need to return the cursor to the end, you press the <Enter> key directly. Even stronger: you can find all the commands starting with a group of letters. For example, to find all commands starting with <plot>, type plot, then press ↑ several times.

1.5.3- Functions on scalars

abs: Absolute value or complex module

conj: Complex conjugate

imag: Imaginary part

real: Real part

prod: Product of all elements of a vector argument

sum: Sum of all elements of a vector argument

round: Rounds all elements to the nearest integers.

1.5.4- Batch files (scripts)

Batch scripting consists of a series of commands to be executed by the command-line interpreter, stored in a plain text file (batch file). Batch files can have any name, but must end with the extension ".m".

To create a command file, tap MATLAB commands inside the integrated editor, save for example under the name " toyou.m", then under MATLAB, tap:

>> toyou

All commands in the file will be executed in bulk.

1.5.5- Comments and self-documentation

White lines are considered comments. Anything after the "%" symbol will also be considered a comment. It is also possible to document your order files. So, define a series of uninterrupted comment lines at the beginning of your toyou.m file. When you type:

>> help toyou

These comment lines will appear on the screen. This is how MATLAB online help works.

It's so simple that it would be a shame to miss out on it.

1.5.6- Removing the display

So far, we have seen that all commands typed in MATLAB displayed the result. For certain commands (creation of large tables), this can be tedious. We can therefore place the character ";" at the end of a command line to tell MATLAB that it should not display the result.

1.6- Exercises

A. First program in Matlab

Write the following example, save it under the name 'firstprog' and run it from the command window.

```
% first program under Matlab
disp('This is my first program')
disp(' It displays the first 10 integers on the screen')
for i=1:10
i
end
% end of program
```

Edit the program by adding a '**;**' After the '**i**'. What does its execution give now?

B. Input and format

Write a script that prompts the user to enter a number, displays it in short simple format then short with power of 10. Indices: input, format. Is the precision in a calculation with each of these numbers affected?

C. Addpath (adds a path)

Write a simple program and save it in a directory that is not the current directory. Try to run it by calling it from the command window. Use the addpath command to make the call effective.

Chapter 2: Data types and variables

2.1– Introduction

Like any programming language, MATLAB allows you to define variable data. A variable is designated by an identifier which is made up of a combination of letters and numbers. Please note, the first character of the identifier must necessarily be a letter. For MATLAB any variable is considered to be an array of elements of a given type.

In this chapter, we will see in particular how to create a variable, give it a value, and use it.

2.2– Types of variables

The three main types of variables used by Matlab are: real, complex and strings. The logical type is associated with the result of certain functions [5].

Variable names must follow a few rules:

• The length of the name is arbitrary;

• The name may contain letters in lowercase or uppercase, without accent;

• The name may contain numbers;

• The name cannot contain special characters (space or punctuation);

• The name must start with a letter.

Example:

```
>> a=1.2
a =
1.2000
```

We can now include this variable in new mathematical expressions, to define a new one:

```
>> b = 5*a^2+a
b =
8.4000
```

and then use these two variables:

```
>> c = a^2 + b^3/2
c =
297.7920
```

I now have three variables a, b and c. These variables are not permanently displayed on the screen. But to see the content of a variable, nothing could be simpler, we tap its name:

```
>>b
```

b =
8.4000

or double-click on its name in the workspace.

2.2.1– Complex

The imaginary unit is designated by **i** or **j**. Complex numbers can be written in Cartesian form:

a+ib (a+i*b (or a+b*i))

Example:

>> Z=a+ b*i

Z=

1.2000 + 8.4000i

The symbol * can be omitted if the imaginary part is a numerical constant. All previous operators work in complex. For example:

>> Z = (a+b*i)^2

Z =

-69.1200 +20.1600i

Here are some commands regarding complex numbers:

imag(Z): returns the imaginary part of Z

real(Z): returns the real part of Z

abs(Z): returns the modulus of Z

angle (Z): returns the angle of Z

conj(Z): returns the conjugate of Z (Z *)

2.2.2– Character string

A string is an array of characters. Character string (char) data is represented in the form of a series of characters surrounded by single apostrophes (‘).

Example:

>> ch1='hello'

ch1 =

Good morning

2.2.3– Logic

A logical type result is returned by certain functions or in the case of certain tests. The logical type has two forms: true or false (1 or 0).

2.3- Clearing variables

The **clear** command clears part or all of the variables defined so far.

Syntax:

clear a b c . . .

If no variables are specified, all variables will be cleared.

2.4- Predefined variables

There are a number of pre-existing variables. We have already seen "**ans**" which contains the last calculation result, as well as i and j which represent.

There is also **pi**, which represents **π**, and a few others. Remember that "**eps**", the name we often tend to use, is a predefined variable. ***WARNING:*** These variables are not protected, so if you assign them, they do not keep their initial value. This is often the problem for **i** and **j** which we often use spontaneously as loop indices, so that we can no longer define a complex.

2.5- Predefined functions

All common functions and many of the less common ones exist. Most of them operate in complexes. Please note that to apply a function to a value, you must put the latter in parentheses.

Example:

```
>> sin(pi/12)
ans =
0.16589613269342
```

Here is a non-exhaustive list:

- trigonometric and inverse functions: sin, cos, tan, asin, acos, atan
- hyperbolic functions (we add "h"): sinh, cosh, tanh, asinh, acosh, atanh
- root, logarithms and exponentials: sqrt, log, log10, exp
- error functions: erf, erfc
- Bessel and Hankel functions: besselj, bessely, besseli, besselk, besselh.

2.6- Matrices and tables

Despite the title of this section, MATLAB does not differentiate between the two. The concept of table is important because it is the basis of the graph: typically for a curve of n points, we will define a table of n abscissa and a table of n ordinates. But we can also define rectangular arrays with two indices to define matrices in the mathematical sense of the term, and then perform operations on these matrices.

2.6.1- Definition of a table

We use the square brackets [and] to define the start and end of the matrix. So to define a variable M containing the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$, we will write:

```
>> M = [1,2,3;11,12,13;21,22,23]
M =
```

1 2 3
11 12 13
21 22 23

We use the symbol ‘,’ which serves as a column separator and ‘;’ line separator. We can also define vectors, row or column, using this syntax (a space is interpreted as a comma).

Example:

>> U = [1 2 3]
U =
1 2 3
defines a line vector, whereas (an "enter" command is interpreted as a semicolon):
>> V = [11
12
13]
V =
11
12
13

Defines a column vector. The transition from one command line to the next is carried out by hitting the <Enter> key.

We could also have defined the latter by:

>> V=[11;12;13]

2.6.2- Accessing an array element

Simply enter the name of the table followed in parentheses by the index(s) whose value you want to read or write.

Example:

If I want the value of M, line 3, column 2:

>> M(3,2)

ans =

22

To modify only one element of an array, we use the same principle. For example, I want M32 to equal 32 instead of 22:

>> M(3,2)=32

M =

1 2 3

11 12 13

21 32 23

You will notice that MATLAB suddenly redisplays the entire matrix, taking the modification into account.

We may wonder what happens if we affect the component of a matrix that does not yet exist.

Example:

```
>> P(2,3) = 3
P =
0 0 0
0 0 3
```

Here's the answer: MATLAB automatically constructs a table large enough to reach the specified indices, and puts zeros everywhere except at the term in question. You will notice that unlike traditional languages, there is no need to size the tables in advance: they are built gradually.

2.6.3- Extracting sub-arrays

It is often useful to extract blocks from an existing table. To do this, we use the '**:**' character. The start value and the end value must be specified for each index. The general syntax is therefore as follows (for an array with two indices):

array(start: end, start: end)

So to extract the block $\begin{bmatrix} 2 & 3 \\ 12 & 13 \end{bmatrix}$, we will tap:

```
>> M(1:2,2:3)
ans =
2 3
12 13
```

We use the "**:**" character alone to take all possible indices.

Example:

```
>> M(1:2,:)
ans =
 1 2 3
11 12 13
```

This is very useful for extracting rows or columns from a matrix. For example to obtain the second line of M:

```
>> M(2,:)
ans =
11 12 13
```

2.6.4- Construction of tables by blocks

You know this principle in mathematics. For example, from the previously defined matrices and vectors, we want to define the matrix $\begin{bmatrix} M & V \\ U & 0 \end{bmatrix}$ which is a 4x4 matrix. To do this in MATLAB, we act as if the blocks were scalars, and we simply write:

```
>> N=[M V
U 0]
```

```
N =
 1 2 3 11
 11 12 13 12
 21 32 23 13
 1 2 3 0
```

or by using the characters ‘,’ and ‘;’

```
>> N=[M, V; U, 0]
```

This syntax is widely used to extend vectors or matrices, for example, if I want to add a column to M, constituted by V:

```
>> M=[M V]
M =
1 2 3 11
1 12 13 12
21 32 23 13
```

If I want to add a row to it, consisting of U:

```
>> M = [M;U]
M =
1 2 3
11 12 13
21 32 23
1 2 3
```

2.6.5- Table operations

2.6.5.1- Addition and subtraction

The two operators are the same as for scalars. As long as the two tables concerned have the same size, the resulting table is obtained by adding or subtracting the terms of each table.

2.6.5.2- Multiplication, division and power term by term

These operators are noted ".*", "./" and ".^" (be careful not to forget the period). They are designed to carry out forward operations on two tables of the same size. These symbols are important when you want to draw curves.

2.6.5.3- Multiplication, division and power in the matrix sense

Since we want to manipulate matrices, it seems interesting to have matrix multiplication. This is simply noted * and should not be confused with term-to-term multiplication. It goes without saying that if we write ***A*B*** the number of columns of ***A*** must be equal to the number of rows of ***B*** for the multiplication to work.

The division makes sense with respect to the inverses of matrices. Thus ***A/B*** represents ***A*** multiplied (in the sense of matrices) by the inverse matrix of ***B***. There also exists a division on the left which is noted \. Thus ***A\B*** means the inverse of ***A*** multiplied by ***B***. This symbol can also be

used to solve linear systems: if v is a vector, $A\backslash v$ mathematically represents $A^{-1}v$, that is to say the solution of the linear system $Ax = v$.

The n^{th} power of a matrix represents this matrix multiplied n times in the sense of matrices by itself.

To clearly show the difference between the ".*" and "*" operators, a small example involving the identity matrix multiplied by the matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.

Here is the multiplication in the sense of matrices:

```
>> [1 0; 0 1] * [1 2; 3 4]

ans =
1 2
3 4
```

and now the multiplication term by term:

```
>> [1 0; 0 1].* [1 2; 3 4]

ans =
1 0
0 4
```

Example:

A. Matrix and extraction

Write a script that defines the following matrix M and displays the last row:

```
 1 2 3 4 5
 9 10 11 12 13
-1 -2 -3 -4 -5
```

B. Random element matrix

Determine the syntax of the rand function, create 2 matrices ***A*** and ***B*** of size 5X6, with random values between -10 and 10. Display ***C= A+B***

C. Operations on a matrix

A=[1.4;6.8]; Is it possible to apply the sin function to the matrix a? What is the result? Write a script that displays the successive results of sin(a).^n with n varying from 1 to 10 and using a for...end loop.

2.6.5.4- Transposition

The transpose operator is the ' (apostrophe) character and is often used to transform row vectors into column vectors and vice versa.

2.6.6- Table lengths

The size function applied to a matrix returns an array of two integers: the first is the number of rows, the second the number of columns. The command also works on vectors and returns 1 for the number of rows (resp. columns) of a row (resp. column) vector.

For vectors, the length command is more convenient and returns the number of components of the vector, whether row or column.

2.6.7- Fast table generation

2.6.7.1- Classic matrices

We can define matrices of a given size containing only **0s** with the zeros function, or containing only **1s** with the ones function. You must specify the number of rows and the number of columns. Here are two examples:

```
>> ones(2,3)
ans =
1 1 1
1 1 1
>> zeros(1,3)
ans =
0 0 0
```

The identity is obtained with **eye**. We only specify the dimension of the matrix (which is square...)

```
>> eye(3)
ans =
1 0 0
0 1 0
0 0 1
```

There is also a **diag** function for creating diagonal matrices (see the built-in help).

2.6.7.2- Lists of values

This notion is crucial for the construction of curves. This involves generating in a vector a list of values equidistant between two extreme values. The general syntax is:

variable = start value: step: end value

This syntax always creates a row vector. For example to create a vector x of values equidistant from 0.1 between 0 and 1:

```
>> x = 0:0.1:1
x =
Columns 1 through 7
0 0.1000 0.2000 0.3000 0.4000 0.5000 0.6000
Columns 8 through 11
0.7000 0.8000 0.9000 1.0000
```

It is recommended to put a semicolon at the end of this type of instruction to avoid tedious display of the result.

Another example to create 101 equally distributed values on the interval [0; 2π]:

```
>> x = 0: 2*pi/100: 2*pi;
```

You can also use the linspace function:

```
>> x=linspace(0,2*pi,101);
```

Values can be distributed logarithmically with the logspace function.

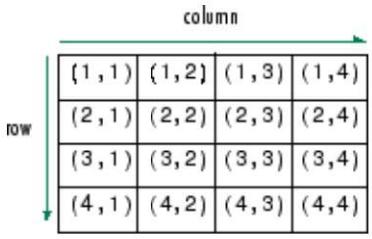
y = linspace(x_1,x_2,n) generates n points. The spacing between the points is (x_2-x_1)/(n-1).

Example: Create a vector of 7 evenly spaced points in the interval [-5,5].

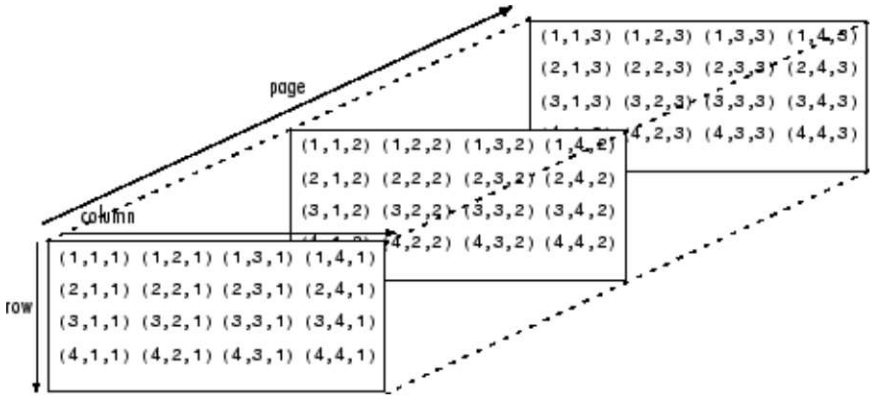
```
y1 = linspace(-5,5,7)
y1 = 1×7
   -5.0000   -3.3333   -1.6667         0    1.6667    3.3333    5.0000
```

2.6.8- Multidimensional Arrays

A multidimensional array in MATLAB is an array with more than two dimensions. In a matrix, the two dimensions are represented by rows and columns.



Each element is defined by two subscripts, the row index and the column index. Multidimensional arrays are an extension of 2-D matrices and use additional subscripts for indexing. A 3-D array, for example, uses three subscripts. The first two are just like a matrix, but the third dimension represents *pages* or *sheets* of elements.



2.6.8.1- Creating Multidimensional Arrays

You can create a multidimensional array by creating a 2-D matrix first, and then extending it. For example, first define a 3-by-3 matrix as the first page in a 3-D array [6].

```
A = [1 2 3; 4 5 6; 7 8 9]
A = 3×3
```

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

Now add a second page. To do this, assign another 3-by-3 matrix to the index value 2 in the third dimension. The syntax A(:,:,2) uses a colon in the first and second dimensions to include all rows and all columns from the right-hand side of the assignment.

```
A(:,:,2) = [10 11 12; 13 14 15; 16 17 18]
A =
A(:,:,1) =

    1    2    3
    4    5    6
    7    8    9

A(:,:,2) =

   10   11   12
   13   14   15
   16   17   18
```

The **cat** function can be a useful tool for building multidimensional arrays. For example, create a new 3-D array ***B*** by concatenating ***A*** with a third page. The first argument indicates which dimension to concatenate along.

```
B = cat(3,A,[3 2 1; 0 9 8; 5 3 7])
B =
B(:,:,1) =

    1    2    3
    4    5    6
    7    8    9
B(:,:,2) =

   10   11   12
   13   14   15
   16   17   18


B(:,:,3) =

    3    2    1
    0    9    8
    5    3    7
```

Another way to quickly expand a multidimensional array is by assigning a single element to an entire page. For example, add a fourth page to ***B*** that contains all zeros.

B(:,:,4) = 0
B =
B(:,:,1) =

1	2	3
4	5	6
7	8	9

B(:,:,2) =

10	11	12
13	14	15
16	17	18

B(:,:,3) =

3	2	1
0	9	8
5	3	7

B(:,:,4) =

0	0	0
0	0	0
0	0	0

Examples:

A. Solving a system of equations

- Consider the system of 5 equations with 5 unknowns:

x+2y+3z+4u+5v=1
x+2y+3z+4u+v=2
x+2y+3z+u+v=3
-x-2y+3z+4u+v=4
x+y+z+u+v=5

- Write the script to resolve this system

B. Fast matrix generation

Let A=**ones**(2,3). Is it possible to calculate A-1, A+A, 2*A, A*A, A.*A? Analyze the results.

C. Fast vector generation

In a command line, create the column vector having the following values (without writing them explicitly):
[10-60;10-59; ...; 1059; 1060]
hint: logspace

D. manipulation of matrices

-Create the following matrix using the **reshape** command:

1 11 21 31 41 51 61 71 81 91
2 12 22 32 42 52 62 72 82 92
3 13 23 33 43 53 63 73 83 93
4 14 24 34 44 54 64 74 84 94
5 15 25 35 45 55 65 75 85 95
6 16 26 36 46 56 66 76 86 96
7 17 27 37 47 57 67 77 87 97
8 18 28 38 48 58 68 78 88 98
9 19 29 39 49 59 69 79 89 99
10 20 30 40 50 60 70 80 90 100

-Use **triu** to create:
1 11 21 31 41 51 61 71 81 91
0 12 22 32 42 52 62 72 82 92
0 0 23 33 43 53 63 73 83 93
0 0 0 34 44 54 64 74 84 94
0 0 0 0 45 55 65 75 85 95
0 0 0 0 0 56 66 76 86 96
0 0 0 0 0 0 67 77 87 97
0 0 0 0 0 0 0 78 88 98
0 0 0 0 0 0 0 0 89 99
0 0 0 0 0 0 0 0 0 100

2.7- Polynomials

A polynomial is a function of the form:

$f(x) = a_n x_n + a_n{-}1x_{n-1} + \ldots + a_2x_2 + a_1x + a_0$.

The degree of a polynomial is the highest power of x in its expression. Constant (non-zero) polynomials, linear polynomials, quadratics, cubics and quartics are polynomials of degree 0, 1, 2 , 3 and 4 respectively. The function $f(x) = 0$ is also a polynomial, but we say that its degree is 'undefined'.

2.7.1- Roots of polynomial functions

You may recall that when $(x - a)(x - b) = 0$, we know that ***a*** and ***b*** are roots of the function: $f(x) = (x - a)(x - b)$. Now we can use the converse of this, and say that if a and b are roots, then the polynomial function with these roots must be $f(x) = (x - a)(x - b)$, or a multiple of this. For example, if a quadratic has roots $\boldsymbol{x} = 3$ and $x = -2$, then the function must be $f(x) = (x - 3)(x + 2)$, or a constant multiple of this. This can be extended to polynomials of any degree. For example, if the roots of a polynomial are $x = 1$, $x = 2$, $x = 3$, $x = 4$, then the function must be $f(x) = (x - 1)(x - 2)(x - 3)(x - 4)$, or a constant multiple of this [7].

Syntax:
r = roots(p)

Example : Solve the equation $3x^2-2x-4=0$.

*Create a vector to represent the polynomial, then find the roots.

```
p = [3 -2 -4];
r = roots(p)
r = 2×1

    1.5352
   -0.8685
```

*Create vectors `u` and `v` containing the coefficients of the polynomials x^2+1 and $2x+7$.

```
u = [1 0 1];
v = [2 7];
```

Use convolution to multiply the polynomials.

```
w = conv(u,v)
w = 1×4

     2     7     2     7
```

`w` contains the polynomial coefficients for $2x^3+7x^2+2x+7$.

Exercises

1. What is a polynomial function?

2. Which of the following functions are polynomial functions?

(a) $f(x) = 4x^2 + 2$ *(b)* $f(x) = 3x^3 - 2x + \sqrt{x}$ *(c)* $f(x) = 12 - 4x^5 + 3x^2$

(d) $f(x) = \sin(x) + 1$ *(e)* $f(x) = 3x^{12} - 2/x$ *(f)* $f(x) = 3x^{11} - 2x^{12}$

3. Write down one example of each of the following types of polynomial function:

(a) cubic (b) linear (c) quartic (d) quadratic

Chapter 3: The graphics

3.1– Introduction

A 2D curve is represented by a series of abscissas and a series of ordinates. The software usually draws segments between these points. The MATLAB function is called **plot**.

The "help graph2d" instruction provides an introduction to 2D graphics and the table of available functions.

Elementary X-Y graphs:

plot : Linear plot.

loglog : Log-log scale plot.

semilogx : Semi-log scale plot.

semilogy : Semi-log scale plot.

polar : Polar coordinate plot.

plotyy : Graphs with y tick labels on the left and right.

This chapter presents the most important commands for developing the most common graphic representations. There are, however, a large number of commands and scripts for creating more sophisticated graphics.

3.2– The plot instruction

The simplest use of the **plot** instruction is as follows:

plot (abscissa vector, ordinate vector) or, plot ([x1 x2 ... xn], [y1 y2 ... yn])

The vectors can be either row or column, provided they are both of the same type. In general they are lines because the generation of lists of values seen previously provides line vectors by default.

Example:

If we want to plot sin(x) on the interval [0; 2π], we start by defining a series (of reasonable size, say 100) of equidistant values on this interval:

>> x = 0: 2*pi/100: 2*pi;

then, as the sin function can be applied term by term to an array:

>> plot(x, sin(x))

which, provides the graph opposite in the graphics window:

We see that the axes automatically adapt to the extreme values of the abscissa and ordinate.

By the way, all plot requires is an abscissa vector and an ordinate vector. The x-coordinates can therefore be a function of x rather than x itself. In other words, it is therefore possible to draw parameterized curves:

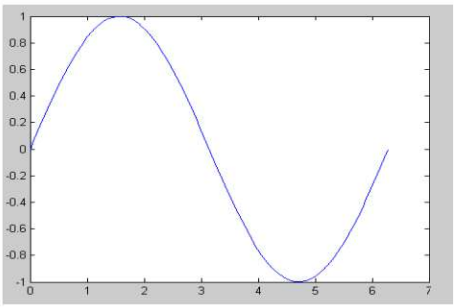
>> plot(cos(x), sin(x))

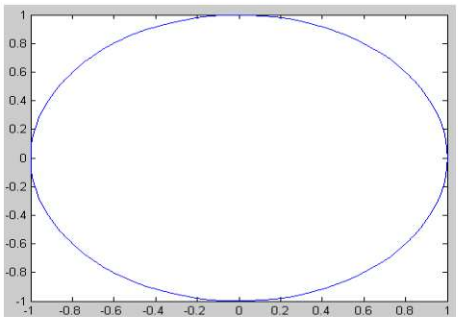
3.2.1– Superimpose several curves

To superimpose several curves, you can proceed in two ways:

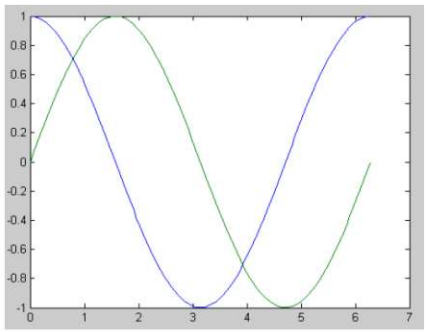
-Specify as many pairs (abscissa, ordinate) as there are curves to trace in the plot command. For example to superimpose sin and cos:

>> plot(x, cos(x),x, sin(x))





The two curves are actually in different colors. This method works even if the abscissa of the two curves are not the same.



-Or draw the first curve on a Plot figure (abscissa1, ordinate1, 'color1') and paste the other curves on it using the "Hold On" command, then draw one by one the Plot curves (abscissae2, ordinate2, 'color2'), Plot(abscissae3, ordinate3, 'color3'),… this technique requires specifying the color of the curve to be able to distinguish them.

3.2.2– Curve attributes

MATLAB assigns default colors to curves. It is possible to modify the color, the style of the line and that of the points, by specifying after each pair (abscissa, ordinate) a character string (between primes) which can contain the following codes (obtained by taping help plot) [8]:

Specifier	Line Style
-	solid line (default)
--	dashed line
:	dotted line
-.	dash-dot line

Specifier	Color
r	red
g	green
b	blue
c	cyan
m	magenta
y	yellow
k	black
w	white

Specifier	Marker Type
+	plus sign
o	circle
*	asterisk
.	point
x	cross
s	square
d	diamond
^	upward pointing triangle
v	downward pointing triangle
>	right pointing triangle
<	left pointing triangle
p	five-pointed star (pentagram)
h	six-pointed star (hexagram)

Codes can be combined with each other. For example:

>> plot(x,sin(x),':',x,cos(x),'r-.')

3.3– Decoration of graphics

3.3.1– Title

It is the "**title**" instruction to which you must provide a character string. The title appears at the top of the window chart:

>> plot(*x*,cos(*x*),*x*,sin(*x*))

>> title('Sin and cos functions')

3.3.2– Labels

This involves displaying something below the x-axis and next to the y-axis:

>> plot(*x*,cos(*x*))

>> xlabel('Abscissa')

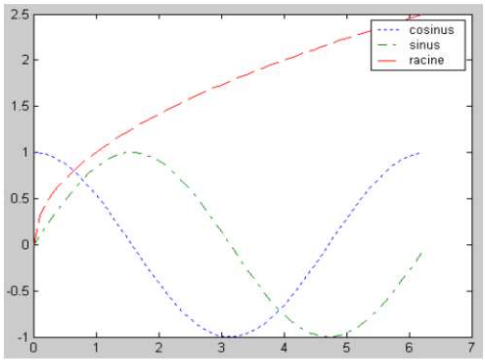
>> ylabel('Ordinated')

3.3.3– Legends

Using the legend statement. It is necessary to communicate as many character strings as curves drawn on the screen. A box is then drawn on the graph, which displays opposite the style of each curve, the corresponding text. For example :

>> plot(x,cos(x), ':', x, sin(x), '-.', x, sqrt(x),'--')

>> legend('cosine', 'sine', 'root')



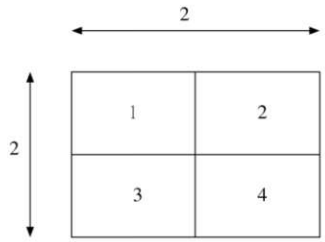
3.3.4– Drawing a grid

This is the "grid" instruction, which is used after a plot instruction displays a grid on the curve. If we type "grid" again, the grid disappears.

To have a fine grid, you must use the "grid minor" instruction.

3.4– Show multiple graphs with the (subplot) function

This is a very useful feature for presenting several results on the same graphic page. The general idea is to divide the graphics window into tiles of the same size, and to display a graph in each tile. We use the "subplot" instruction by specifying the number of tiles on the height, the number of tiles on the width, and the number of the tile in which we are going to plot:



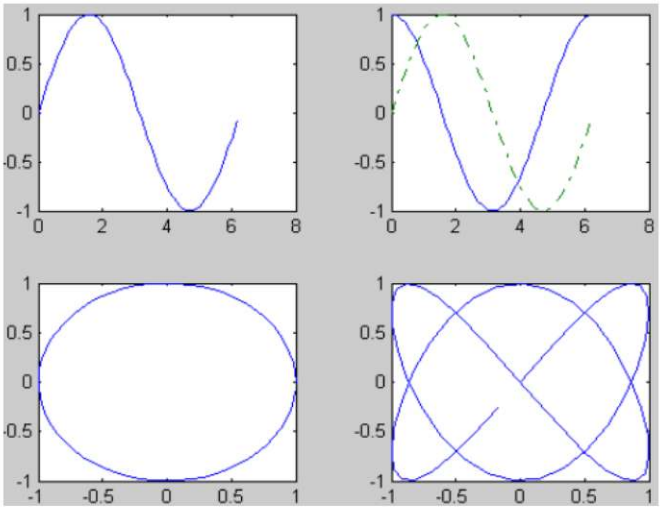
subplot (Number of tiles on height, Number of tiles on width, Number of tiles)

The comma can be omitted. The blocks are numbered in the direction of reading a text:

from left to right and top to bottom:

Once a "subplot" command is typed, all subsequent graphics commands will be executed in the specified pad. Thus, the following graph was obtained from the sequence of instructions:

```
>> subplot(221)
>> plot(x,sin(x))
>> subplot(222)
>> plot(x,cos(x),x,sin(x),'-.')
>> subplot(223)
>> plot(cos(x),sin(x))
>> subplot(224)
>> plot(sin(2*x),sin(3*x))
```



3.5– Axes and zoom

There are two ways to modify the extreme values on the axes, in other words to zoom on the curves. The simplest is to use the graphics window button. You can then:

- Frame an area to zoom with the left mouse button, or
- Click on a point with the left button. The clicked point will be the center of the zoom, the latter being carried out with an arbitrary factor,
- Click on a point with the right button to zoom out

To zoom more precisely, use the axis command (see the integrated help).

3.6– Clearing the graphics window

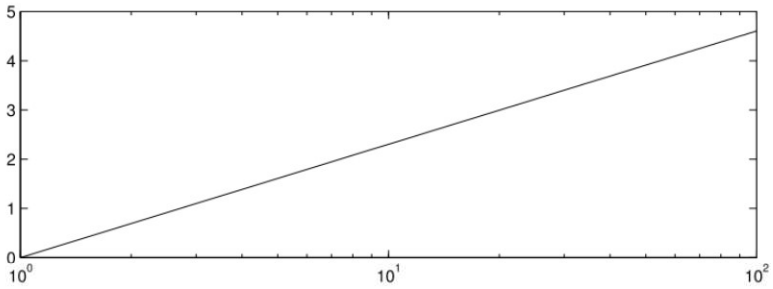
Just tap **clf**. This command also cancels any subplot and hold commands placed.

3.7– Entering a point with the mouse

The **ginput**(N) command allows you to click N points in the graphics window. The command then returns two vectors, one containing the abscissa, the other the ordinates. Used without the N parameter, the command loops until the "Enter" key is typed.

3.8– Logarithmic scales

We can plot log scales on the abscissa, on the ordinate or both. The corresponding functions are called **semilog*x*, semilog*y*** and **loglog**, respectively. They are used in exactly the same way as plot. For example:



>> *x*=1:100;

>> semilog*x*(*x*,log(*x*))

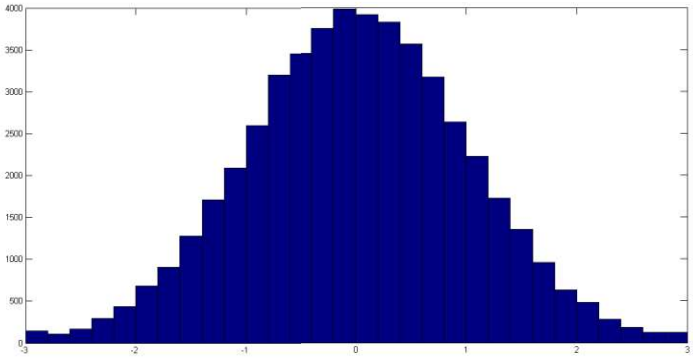
gives the curve opposite.

3.9– histograms: hist

>>*x*=-2.9: 0.2: 2.9; % Sets the interval and width of the histogram channels.

>>*y*=**randn**(50000,1); %Generates random numbers distributed according to a normal distribution

>> **hist**(*y*,*x*) % Draw the histogram of the figure below



3.10– Summary quote of some other graphic functions

- axis([xmin xmax ymin ymax],'option'); option: square, fill, tight, image...

xlim([xmin xmax]);

xlim('option'); option = mode, auto, manual

- **grid on**: allows you to add a grid

- **grid off**: hide the grid in the graph

- **clf** : clear all graphs in the current window or clear the graph (fig) mentioned clf('reset') or clf(fig)

-**close**: delete the figure(s) close fig or close all

-**refresh** : redraws the current figure refresh or refresh(fig)

-**cla** : delete current axes all its graphic objects **cla reset** or **cla(ax)**

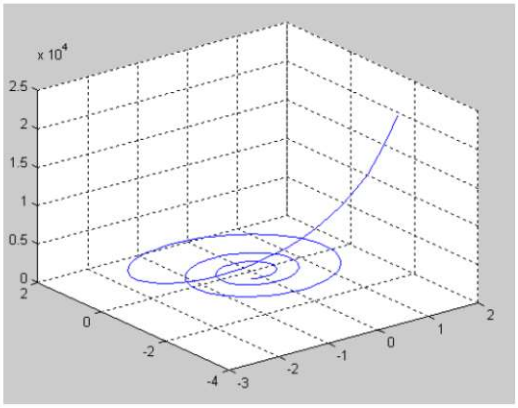
3.11– 3D graphics and animations

3.11.1– 3D curves

A 2D curve is defined by a list of doublets (*x; y*) and a 3D curve by a list of triplets (*x; y; z*). Since the plot instruction expected two arguments, one for *x*, one for *y*, the **plot3** instruction expects three: *x, y and z*.

3.11.1.1– Parameterized curve (figure opposite)

```
>> t=-10:0.1:10;
>> plot3(exp(-t/10).*sin(t), exp(-t/10).*cos(t), exp(-t))
>> grid
```



For more details:

>>help graph3d % introduction to 3D graphics and table of available functions

3.11.1.2– Generation of points (meshgrid)

To define a surface, you need a set of triples (x; y; z). In general the points (x; y) trace a regular mesh in the plane but this is not an obligation. The only constraint is that the number of points is the product of two integers m×n (we will understand why very soon).

If we have a total of m×n points, this means that we have m×n values of x, m×n values of *y* and m×n values of *z*. It therefore appears that abscissa, ordinate and dimensions of the points on the surface can be stored in tables of size m×n.

All surface drawing instructions, for example surf, will therefore have the following syntax:

surf (abscissa table, ordinate table, dimensions table)

$$\begin{bmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & \vdots & x_{2n} \\ \vdots & \vdots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \quad \begin{bmatrix} y_{11} & \cdots & y_{1n} \\ y_{21} & \vdots & y_{2n} \\ \vdots & \vdots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{bmatrix} \quad \begin{bmatrix} z_{11} & \cdots & z_{1n} \\ z_{21} & \vdots & z_{2n} \\ \vdots & \vdots & \vdots \\ z_{m1} & \cdots & z_{mn} \end{bmatrix}$$

It now remains to construct these tables. Let us take the example of the surface defined by $z = x2 + y2$ for which we want to trace the representative surface on the plane [-1; 1] × [-2; 2].

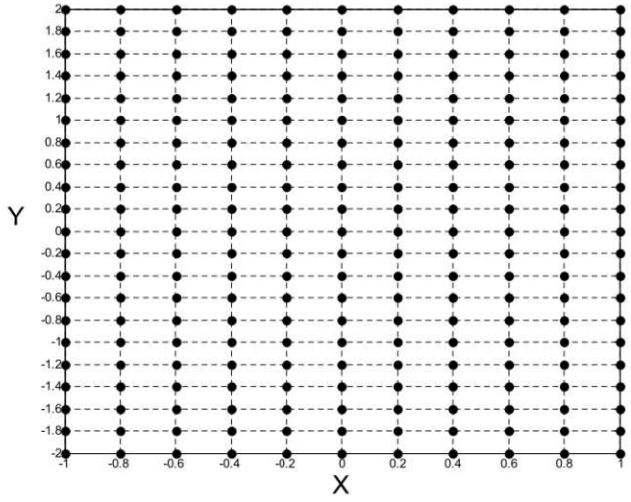
To define a grid of this rectangle, you must first define a series of values x1; ... ; xm for x and a sequence of values y1; ... ; yn for y, for example:

```
>> x = -1:0.2:1
>> y = -2:0.2:2
```

Combining all these values of x and y, we obtain m×n points in the (x; y) plane. We must now construct two tables, one containing the 11×21 abscissa of these points and the other the 11×21 ordinates, i.e.:

$$X = \begin{bmatrix} -1 & -0.8 & \cdots & 1 \\ -1 & -0.8 & \cdots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ -1 & -0.8 & \cdots & 1 \end{bmatrix} \qquad Y = \begin{bmatrix} -2 & -2 & \cdots & -2 \\ -1.8 & -1.8 & \cdots & -1.8 \\ \vdots & \vdots & \vdots & \vdots \\ 2 & 2 & \cdots & 2 \end{bmatrix}$$

Thus, if we consider for example the first column of these matrices, we obtain all the pairs (*x, y*) for *x*=-1: (-1, -2), (-1, -1.8), (-1, -1.6), (-1, -1.4), ..., (-1, 1.8) and (-1, 2). The other columns cover the rest of the portion of the *xy* plane. Below, the corresponding mesh:

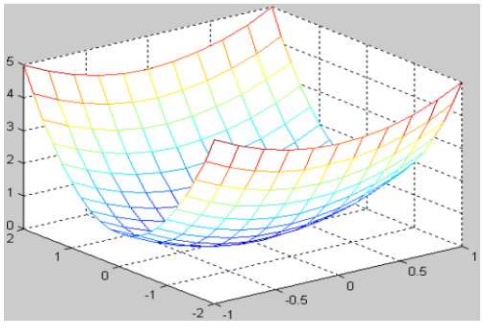


Don't worry, no need for loops, the meshgrid function is responsible for generating these 2 matrices:

```
>> [X,Y] = meshgrid(x, y);
```

It now remains to calculate the corresponding z = f(x, y). This is where term-to-term calculations on matrices still show their effectiveness: we apply the formula directly to tables X and Y, without forgetting to put a point in front of the operators *, / and ^

```
>> Z = X.^2 + Y.^2;
```



Then we can use all the surface drawing functions, for example mesh:

```
>> mesh(X,Y,Z)
```

The most common instructions are:

- mesh which draws a series of lines between points on the surface
- meshc which works like mesh but adds the contour lines in the plane (x; y)

- **surf** which "paints" the surface with a color depending on the coast

- **surfl** which "paints" the surface as if it were illuminated.

- **surfc** which works like mesh but adds the contour lines in the (x, y) plane

Finally, note that there are conversion functions between Cartesian, cylindrical and spherical coordinates, making it possible to easily draw curves defined in one of these coordinate systems. For example, we will look at the cart2pol documentation.

Example :

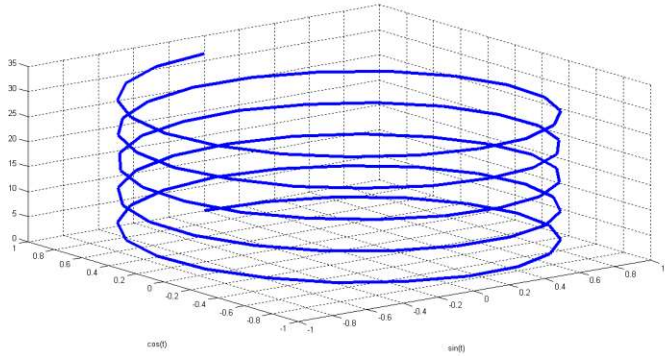
has. line in space:

>> t = linspace(0, 10*pi);

>> plot3(sin(t), cos(t), t)

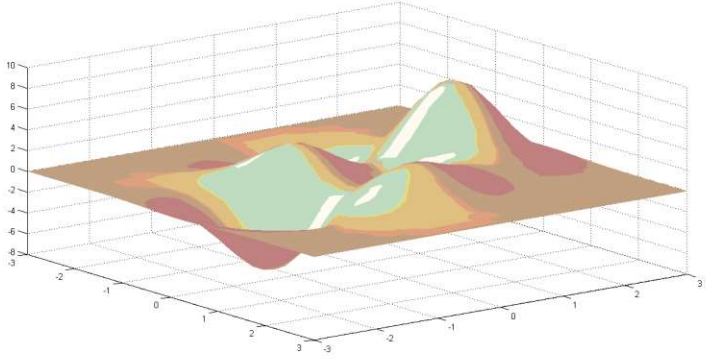
>> xlabel('sin(t)'), ylabel('cos(t)'), zlabel('t')

>> grid on % Figure below

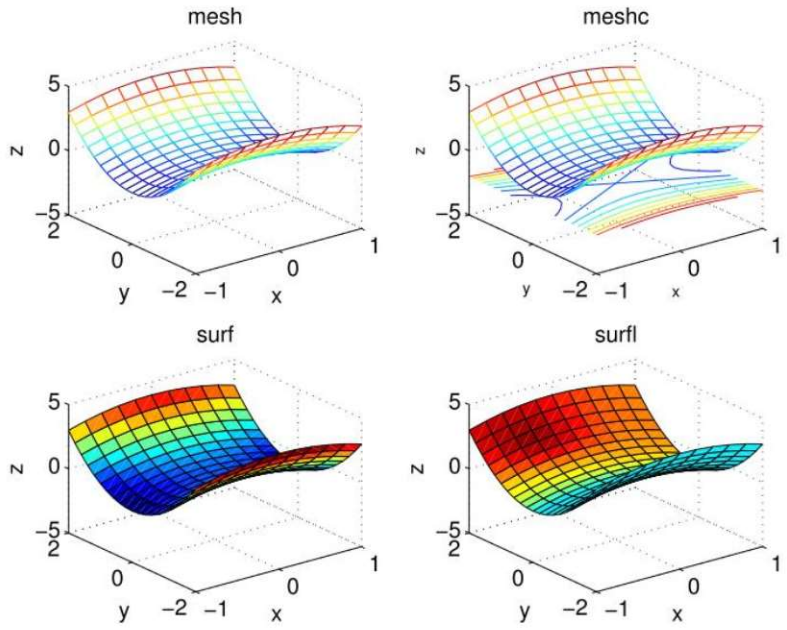


3.11.1.3– Surface with illumination: surfl

>> clf
>> [X,Y,Z] = peaks(30);
>> surfl(X,Y,Z) % graph with illumination
>> shading interp % best interpolation
>> colormap pink %choice of a predefined color palette
>> view(-37.5+90, 30) % change point of view: view(azimuth, elevation)



3.11.1.4– Differences between mesh, meshc, surf and surfl instructions



3.11.1.5– Colormaps

The colormap instruction defines the sequence of colors, or "palette", for painting surfaces. You can give color tables directly or use predefined palettes (for example 'bone', 'winter', 'hot', etc.). Refer to the documentation for more information.

3.11.1.6– Interpolation

You will have noticed that MATLAB plots surfaces with facets. It is possible to smooth these surfaces with the shading instruction. We can test for example, following an instruction on fl (this is the most spectacular) see the figure in example 4.b:

>> shading interp

3.11.1.7– Function Usage

plot3: Plotting a parametric line in 3D

mesh: Trace of a surface in 3D, from mesh matrices

meshgrid: Defining mesh matrices from two vectors

surf: Drawing of a 3D surface with color gradient, from mesh matrices

surfc: Trace of a 3D surface with color gradient and iso-value lines

ezmesh ,ezmeshc: Easy surface plotting (mesh matrices set by default)

ezsurf, ezsurfc: Easy surface drawing with color gradient (mesh matrices defined by default)

sphere: Definition of mesh matrices for drawing a sphere

cylinder: Definition of mesh matrices for drawing a cylinder

3.12– Animations in Matlab

MATLAB treats an animation as a succession of images whose scrolling speed can be controlled.

Creating an animation begins by reserving the memory space necessary for the different images or frames.

The moviein command does this.

The command syntax is:

Anim = moviein(n)

This command initializes the memory for recording an animation. An animation matrix is created to be able to store n images. Each column of the matrix corresponds to an image of the animation.

To generate an animation whose images contain the entire graphics window and not just a graphic, we will use the following syntax:

Anim = moviein(n,gcf)

An image is an instantaneous photo, the animation is therefore a series of instantaneous images. Images are captured using the commands below:

getframe(h): allows you to capture an image of the graphic object whose pointer or handle is h (root, figure or axis)

getframe(h,rect): allows you to capture an image with specification of a rectangle as capture area, the rectangle is indicated by its coordinates in units and in relation to the lower left corner of the graphics window.

Movie(n): allows the display of the animation,

More generally, you can create an animation in two different ways:

• Either by saving a certain number of figures and scrolling through them later like a film,

• Or by erasing and redrawing the objects on the screen as you go.

Create a movie using the movie and getframe commands

Example:

```
>>for i=1:5
>>plot(sin(i*pi*[0:0.025:2]));
>>M(:,i) = getframe;
>>end;
>>movie(M);
```

Create an animation using the **hold on** and **pause** commands

• When two plot commands follow one another, the second erases the first on the figure. The hold on command keeps the result of the two plot commands in the figure.

• The **pause** command allows you to stop temporarily a program, in particular between two plot commands. To restart the program, simply press any key on the keyboard.

• The pause(x) command allows you to pause for a duration fixed by x in a program. The bigger x is, the longer the pause!

Example:

```
>>y=0:0.1:2;
>>plot(y,sin(y*pi),'-r');
>>hold on;
>>for x=0:0.1:2
plot(x,sin(x*pi),'*');
break(1);
end;
>>y=0:0.1:2;
>>for x=0:0.1:2
plot(y,sin(y*pi),'-r',x,sin(x*pi),'*');
break(0.5);
end;
```

3.12.1– Import/export animations

It is possible to export a Matlab movie to an MPEG file using mpgwrite, or movie2avi, commands found on the Mathworks website. Likewise, it is possible to load an MPEG file into a Matlab movie with mpgread.

Chapter 4: Programming in MATLAB

4.1– Introduction

We will see in this chapter, the simplest type of MATLAB program is called a *script*. A script is a file that contains multiple sequential lines of MATLAB commands and function calls. You can run a script by typing its name at the command line.

4.2– Functions

We have seen a number of predefined functions. It is possible to define your own functions. The first method allows you to define simple functions on a command line. The second, much more general, allows you to define very advanced functions by defining it in a file.

4.2.1– Inline functions

Let's say I want to define a new function that I call sincos defined mathematically by:

sincos(x) = sin(x) - x cos(x)

We will write:

>> **sincos = inline('sin(x)-x*cos(x)', 'x')**
sincos =
Inline function:
*sincos(x) = sin(x)-x*cos(x)*

It's very simple. Don't forget primes, which define character strings in MATLAB. We can now use this new function:
>> sincos(pi/12)
years =
0.0059

Now let's try applying this function to an array of values:

>> **sincos(0:pi/3:pi)**

??? Error using ==> inline/subsref

*Error in inline expression ==> sin(x)-x*cos(x)*

*??? Error using ==> **

Inner matrix dimensions must agree.

It doesn't work. Did you understand why? (this is the same problem as for the plot(x,x*sin(x)) instruction seen previously) MATLAB tries to multiply x row vector by cos(x) also row vector in the sense of matrix multiplication! Therefore it is necessary to use a term by term multiplication .* in the definition of the function:

>> **sincos = inline('sin(x)-x.*cos(x)')**

sincos =

Inline function:

*sincos(x) = sin(x)-x.*cos(x)*

>> **sincos(0:pi/3:pi)**

ans =

0 0.3424 1.9132 3.1416

We can therefore state the following rule:

When defining a function, it is preferable to systematically use the term-by-term operators: .*, ./ and .^ instead of * / and ^ if we want this function to be able to apply to tables .

We can similarly define functions of several variables:

>> **sincos = inline('sin(x)-y.*cos(x)', 'x', 'y')**

sincos =

Inline function:

*sincos(x,y) = sin(x)-y.*cos(x)*

The order of the variables (displayed on the screen) is that in which they appear in the function definition. To try :

>> sincos(1,2)

4.2.2- Functions defined in a file

In mathematics we often write functions in the form:

y = f(x) or z = g(x, y) or ....

We will see that it is the same thing under MATLAB. Function names should be more explicit than f or g. For example, a function returning the square of an argument x could be named square (we avoid accents):

y = square(x)

In the file defining the function, this syntax is also preceded by the function keyword:

General form of a function in a functionname.m file:

function Result = functionname(Parameters_Passes_to_the_Function)

% comment lines (descriptive help appearing in the % command window if you type: help functionname)

command lines

...

Result=...

Let's start by taking the previous example (sincos). The order of operations is as follows:

1. Edit a new file called sincos.m

2. Type the following lines:

function s = sincos(x)

% sincos calculates sin(x)-x.*cos(x)

s = sin(x)-x.*cos(x);

3. Save

The result is the same as before:

>> sincos(pi/12)

years =

0.0059

We will notice several things:

- the use of .* so that the function is applicable to tables.

- the variable s is only there to specify the output of the function.

- the use of; at the end of the calculation line, otherwise MATLAB would display the function result twice.

So one more rule:

When defining a function in a file, it is preferable to put '**;**' at the end of each command constituting the function. However, be careful not to put any on the first line.

Another important point: the file name must have the extension .m and the file name without suffix must be exactly the name of the function (appearing after the function keyword)

As for where in the tree the file should be placed, the rule is the same as for scripts.

Now let's see how to define a function with multiple outputs. We want to create a function called cart2pol which converts Cartesian coordinates (x; y) (inputs of the function) into polar coordinates (r; fi) (outputs of the function). Here is the content of the cart2pol.m file:

function [r, theta] = cart2pol (x, y)

% cart2pol converts Cartesian coordinates (x; y) (function inputs)

% in polar coordinates (r; fi) (function outputs)

% sqrt = square root (Matlab internal function)

% atan = reciprocal tangent function (internal function of Matlab)

r = sqrt(x.^2 + y.^2);

theta = atan (y./x);

Note that the two output variables are enclosed in square brackets and separated by a comma.

To use this function, we will write for example:

>> [module,phase] = cart2pol(1,1)

module =

1.4142

phase =

0.7854

We therefore simultaneously assign two variables module and phase with the two outputs of the function, by putting these two variables in square brackets, and separated by a comma.

It is possible to retrieve only the first output of the function. MATLAB often uses this principle to define functions that have a main output and optional outputs.

So for our function, if only one output variable is specified, only the polar radius value is returned. If the function is called without an output variable, ans takes the value of the polar radius:

```
>> cart2pol(1,1)
ans =
1.4142
```

Exercise: Create a function: vec2col

Write the vec2col.m function which transforms any vector (row or column) passed as an argument into a column vector. Error processing (error function) will test that the variable passed to the function is indeed a vector and not a matrix (use size and the conditional if statement described later in the course).

Exercise: First and second degree equations

Complete the 2 functions whose first 2 lines are respectively:

```
function x=equ1(a,b)
% resolves a*x+b=0
```

And

```
function x=equ2(a,b,c)
% resolves a*x^2+b*x+c=0
```

which solve the first and second degree equations. Use the nargin function so that the call to equ2(a,b, c) is valid. Check that the solutions are correct.

Exercise: Nargin, nargout

Research with help the syntax and usage of these commands. Apply them to equ1 and equ2.

4.2.3-Variable scope

Inside functions like the one we just wrote, you have the right to manipulate three types of variables:

- The function's input variables. You cannot modify their values.
- The output variables of the function. You must assign a value to them.
- Local variables. These are temporary variables to split calculations for example. They only have meaning inside the function and will not be “seen” from the outside. And THAT'S ALL!

Now imagine that you write a command file, and a function (in two different files of course), the command file using the function. If you want to pass a value A from the batch file to the function,

you normally have to pass through an input to the function. However, sometimes we would like the variable A of the command file to be usable directly by the function.

For this we use the global directive. Let's take an example: you have a relationship giving the Cp of a gas as a function of the temperature T:

Cp(T) = AT3 + BT2 + CT + D

and you want to make it a function. We could make a function with 5 inputs, for T, A, B, C, D, but it is more natural for this function to have only T as an input.

How to pass A,B,C,D which are constants? Answer: we declare these variables globally both in the command file and in the function, and we assign them in the command file.

Command file:

```
global A B C D
A = 9.9400e-8;
B = -4.02e-4;
C = 0.616;
D = -28.3;
T = 300:100:1000; % Temperature in Kelvins
plot(T, cp(T)) % We plot the CP as a function of T
```

The function:

```
function y = cp(T)
global A B C D
y = A*T.^3 + B*T.^2 +C*T + D;
```

Exercise: Local variable

Write a base name script containing the single line:

```
a=12345;
```

Run base, then type in the command window:

```
a ↵
```

Type the local name function containing the 2 lines into the editor:

```
function local
a=0
```

Execute the local function, then type in the command window:

```
a ↵
```

Explain the results displayed.

Exercise: Global variable

Write a globa name function containing the lines:

```
% function illustrating the concept of global variable
```

% see also globb

global A

A=101010;

Write a globb name function containing the lines:

function globb

% function illustrating the concept of global variable

% see also globa

global A

available(A)

Execute globa, then globb. Tap A↵ in the command window. Now tap global A in the command window, then A↵. Explain the results displayed. Explain the difference between the clear A and clear global A commands.

4.3-Control structures

We are going to talk here about tests and loops. Let's start with comparison operators and logical operators.

4.3.1-Comparison and logical operators

Let us first note the following important point, inspired by the C language:

MATLAB represents the logical constant "**FALSE**" as **0** and the constant "**TRUE**" as **1**. This is particularly useful, for example for defining functions piecewise. Then, the following two tables contain most of what you need to know:

4.3.1.1-MATLAB Syntax Operator

We may wonder what happens when we apply a comparison operator between two arrays, or between an array and a scalar. The operator is applied term by term. So:

Operator	MATLAB syntax
equal to	==
different from	~=
greater than	>
greater than or equal to	>=
less than	<
less than or equal to	<=
negation	~
or	|
and	&

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

ans =
0 1
1 0
```

Terms of A greater than 2 give 1 (true), the others 0 (false).

An important application of logical operators concerns the automatic processing of large matrices. Let's create a matrix (2X5, small for the example):

```
>> A=[10, 12, 8, 11, 9; 10, 10, 128, 11, 9];
```

A =

10 12 8 11 9

10 10 128 11 9

Element 128 deviates greatly from all others. For example, it may be an aberrant point that appeared during an experimental measurement and we wish to replace values greater than 20 with the value 10. For a small matrix such as A, this is easily achievable, simply tap: A(2, 3)=10; But this method is unrealistic if we have to do with a 1000X1000 matrix! Instead, let's use:

>> A(A>20)=10;

And that's it:

A =

10 12 8 11 9

10 10 10 11 9

We can also construct functions piecewise. Let's imagine that we want to define the following function:

f(x) = sin(x) if x > 0 otherwise f(x) = sin(2x)

Here's how to write the function:

>> f = inline('sin(x).*(x>0) + sin(2*x).*(~(x>0))', 'x')

f =

Inline function:

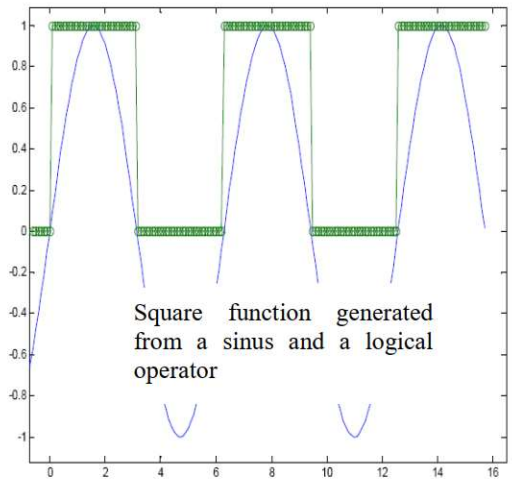
f(x) = sin(x).*(x>0) + sin(2*x).*(~(x>0))

We add the two expressions sin x and sin 2x by weighting them by the logical condition defining their domains of validity. Simple but effective! Let us represent the function thus defined:

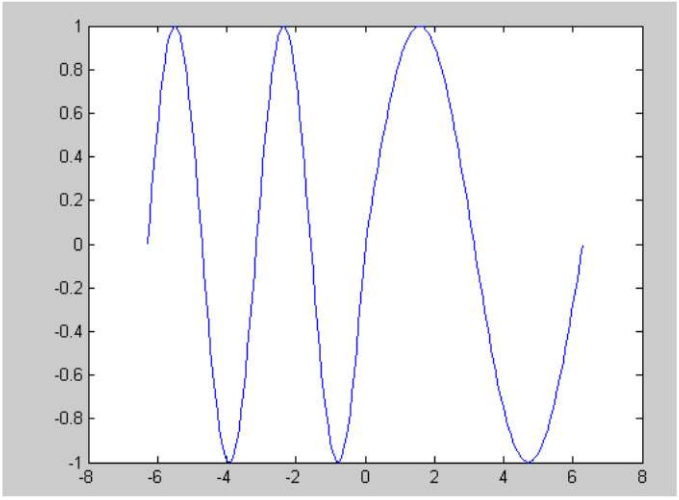
>> x=-2*pi:2*pi/100:2*pi;

>> plot(x,f(x))

gives the following curve on the left:



Square function generated from a sinus and a logical operator



Example: Using logical operators

From a sine function, create a square function such as the one above on the right.

4.3.2-The find command

The find command is useful for simply extracting elements from an array according to a given logical criterion.

>> k = find(x)

returns in the variable k the list of indices of the array x whose elements are non-zero.

Knowing that "FALSE" and 0 are identical in MATLAB, this allows you to find all the indices of a table respecting a given logical criterion:

>> x = [-1.2, 0, 3.1, 6.2, -3.3, -2.1]

x =

-1.2000 0 3.1000 6.2000 -3.3000 -2.1000

>> inds = find (x < 0)

inds =

1 5 6

We can then easily extract the subarray containing only the negative elements of x by simply writing:

>> y = x(inds)

y =

-1.2000 -3.3000 -2.1000

Remember this example carefully. This type of sequence proves very useful in practice because here again, it saves an (if) test. Finally, note that the find command also applies to 2D tables.

Exercise: Reshape and find

Explain precisely what the following command does:

>>a=reshape(fix((rand(1,400)-0.5)*100),20,20);

Find all elements with value equal to 7.

Transform matrix a into a column vector b, sort b in ascending order. How many elements have a value of 7?

Hints: find, sort.

WARNING: unless you add an explicit comment, this type of expression should be avoided because it makes understanding the code difficult.

4.3.3-Conditional if statements

The syntax is as follows:

if logical condition

instructions

elseif logical condition

instructions

...

else

instructions

end

Note that there is no need for parentheses around the logical condition. Note that it is often possible to avoid blocks of this type by directly exploiting the comparison operators on arrays (see the square function and the piecewise function defined in the previous section), as well as the find command.

*Exercise: Function with if and is**

A number of functions detect the state of MATLAB entities. Create a parity(x) function which returns the parity of the number x ('x is even' or 'x is odd'), specifies whether it is a prime number ('x is prime'), processes the case where x is not a number and where x is not integer (using the integer function). The function must use the isnumeric, isprime functions and the if...elseif...end control commands.

4.3.4-for loops

for variable = start value: step: end value

instructions

end

Example:

```
for i = 1:m
for j = 1:n
H(i,j) = 1/(i+j);
end
end
```

Exercise: Conditional loop, break

How many times is this loop evaluated? What are the values of i displayed on the screen?

```
for i=12:-2:1;
available(i)
if i==8
break
end
end
```

Exercise: Tick and knock

Write a script using the tic and toc functions and one (or more) loop(s) such that the elapsed time is of the order of a second. Is the result always the same?

4.3.5-While loops

while logical condition

instructions

end

Exercise: Loop while

What does the following script do?

```
eps = 1;
while (1+eps) > 1
eps = eps/2;
end
eps = eps*2
```

4.3.6-return

A return to the calling function, or command window, is caused when MATLAB encounters the return command.

4.3.7-Switch, case, otherwise

Exercise: Switch

What does the script below do?

```
clc
color = input('Enter a color: ', 's');
color=lower(color);
color switch
'red' box
string='The chosen color is: red';
'green' box
string='The chosen color is: green';
box 'blue'
string='The chosen color is: blue';
otherwise
string='Unknown color';
end
disp(string)
```

Exercise: Implicit loops

Explicitly write the vector u defined by implicit loops:

u(1:2:10)= (1:2:10).^2;

u(2:2:10)= (2:2:10).^3;

Explain the results.

Chapter 5: Getting started with SIMULINK

5.1-Introduction

Simulink is the MATLAB graphics extension for representing functions mathematics and systems in block diagram form, and to simulate the operation of these systems. The objectives of this course are:

- Get to grips with the Simulink Matlab simulation software;
- See some essential blocks to use in practical exercises;
- Learn how to modify block parameters and simulate a Simulink model.

5.2- SIMULINK libraries

To open Simulink type "simulink" in the matlab command window or click on the icon present in the main MATLAB window. A dialog box containing the list of libraries appears on the screen. Each library contains a set of blocks used to create diagrams, or block diagrams [9].

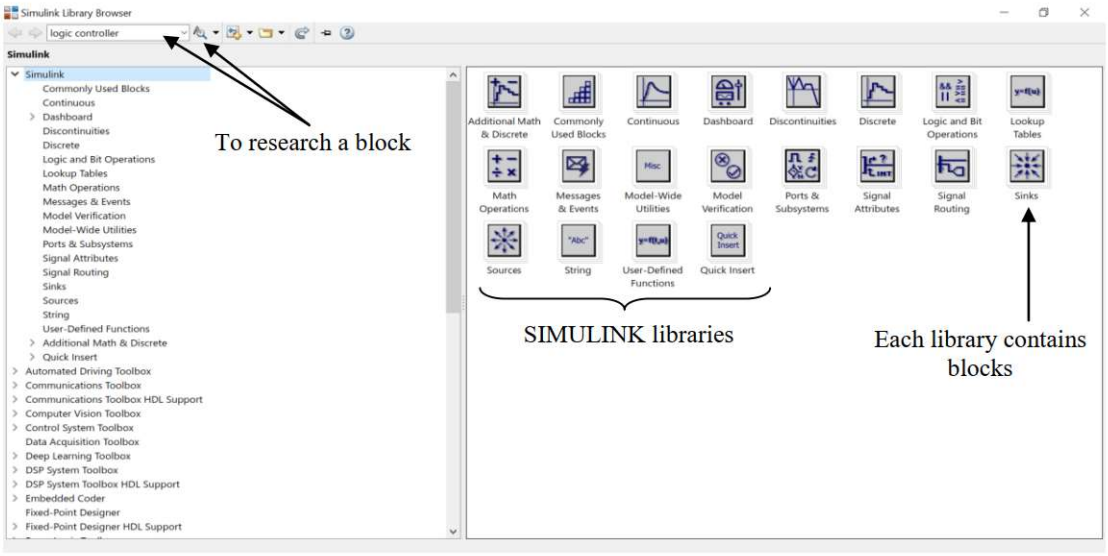


Figure 5.1 *Simulink library browser*

5.3- Quick start

Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. It is integrated with MATLAB, enabling you to incorporate MATLAB algorithms into models and export simulation results to MATLAB for further analysis.

To create a Simulink model follow the following steps:

- From the Simulink Library Browser menu select File > New > Model or click on the icon, an Untitled working window will open.

- Open the block collections by double clicking (You can search for any block by typing its name in the search bar of the Simulink Library Browser);
- Drag the blocks you need to build the diagram into the working window (You can take a copy of a block by right-clicking and dragging it);
- Make connections between the blocks using the mouse;
- Change the parameters of any block by double clicking on this block, a window will open.
- When you have finished the diagram, save the model to a file: File > Save or the icon registration and give a name to the model (*.mdl).

5.4 - Masks and subsystems

5.4.1 – Subsystems

To group multiple blocks into a single block (subsystem), select the blocks you want contained in the subsystem, then select Create subsystem from the Edit menu (or right mouse button)

5.4.2 - Masking of subsystems

To mask a subsystem file:

- Open the subsystem file to be masked.
- In the Simulink toolstrip, on the Subsystem tab, click Create System Mask. Alternatively, right-click anywhere on the canvas and select Mask and click Create System Mask. The Mask Editor dialog opens.
- Add mask parameters and click OK.

Simulink invokes the **callback** functions to disable or hide the controls whenever you open the dialog boxes. Disable a Button on a Dialog Box.

5.5- Study of some simulation examples

The simple model consists of three blocks: Step, Transfer Function, and Scope. The Step is a **Source** block from which a step input signal originates. This signal is transferred through the **line** in the direction indicated by the arrow to the *Transfer Function* **Continuous** block. The *Transfer Function* block modifies its input signal and outputs a new signal on a line to the *Scope*. The *Scope* is a **Sink** block used to display a signal much like an oscilloscope.

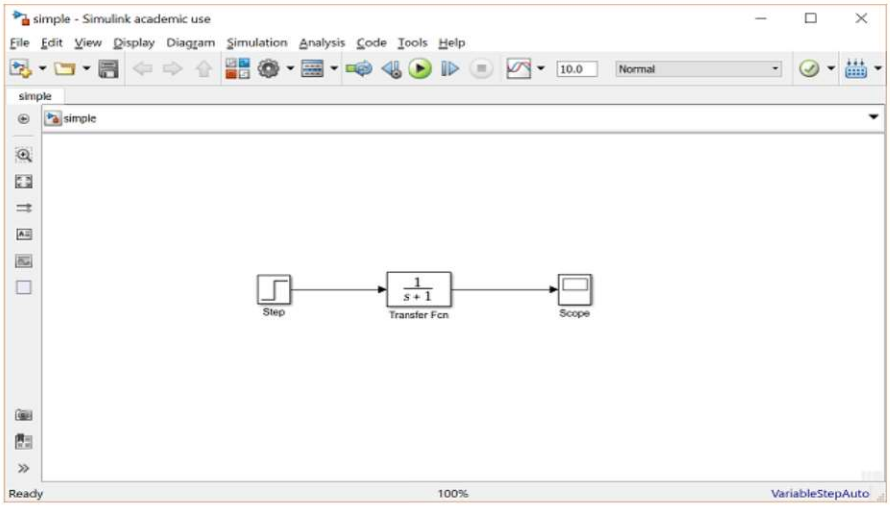
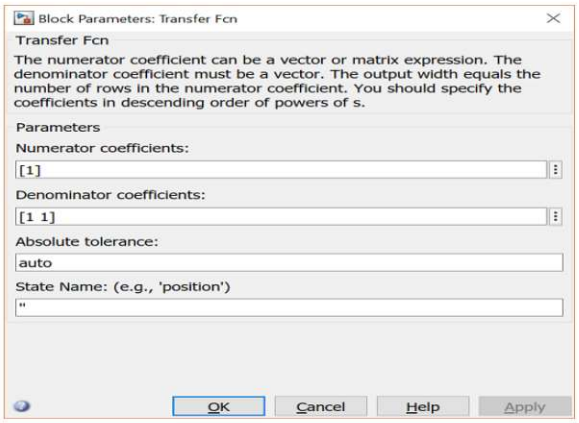


Figure 5.2 *Simple simulation example*

There are many more types of blocks available in Simulink, some of which will be discussed later. Right now, we will examine just the three we have used in the simple model.

5.5.1- Modifying Blocks

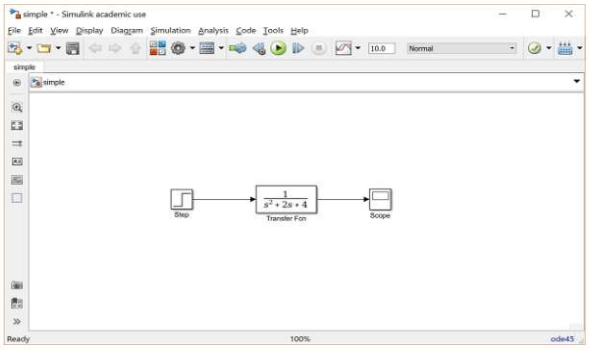
A block can be modified by double-clicking on it. For example, if you double-click on the *Transfer Function* block in the Simple model, you will see the following dialog box. This dialog box contains fields for the numerator and the denominator of the block's transfer function. By entering a vector containing the coefficients of the desired numerator or denominator polynomial, the desired transfer function can be entered. For example, to change the denominator to: s^2+2s+4



enter the following into the denominator field

[1 2 4]

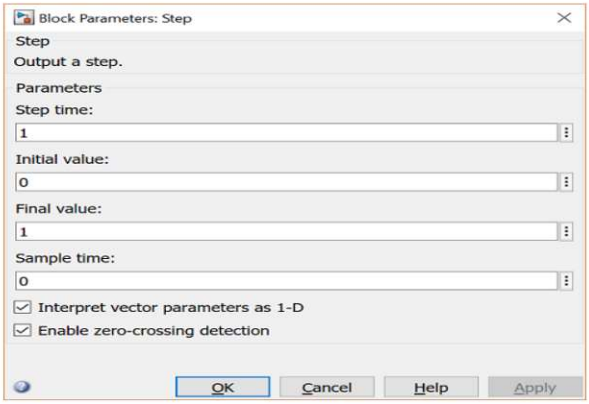
and hit the close button, the model window will change to the following, which reflects the change in the denominator of the transfer function.



The *Step* block can also be double-clicked, bringing up the following dialog box.

The default parameters in this dialog box generate a step function occurring at time = 1 sec, from an initial level of zero to a level of 1 (in other words,

a unit step at t = 1). Each of these parameters can be changed. Close this dialog before continuing.



When a simulation is performed, the signal which feeds into the scope will be displayed in this window. Detailed operation of the scope will not be covered in this tutorial.

5.5.2- Running Simulations

Before running a simulation of this system, first open the scope window by double-clicking on the scope block. Then, to start the simulation, either select **Run** from the **Simulation** menu, click the **Play** button at the top of the screen, or hit **Ctrl-T**.

The simulation should run very quickly and the scope window will appear as shown below.

5.5.3- Simulation parameters

- Open the configuration settings by selecting Configuration Parameters in the Simulation menu.
- Set simulation times using the Start time fields and Stop time in the Solver section of the dialog box Configuration Parameters.

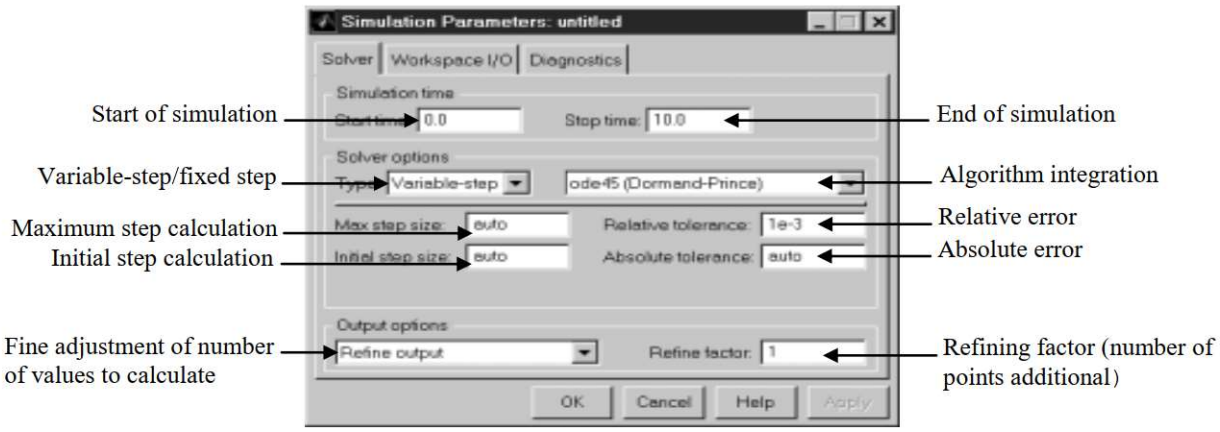


Figure 5.3 *Configuration parameters*

Chapter 6: Power System Blockset (PSB)

6.1-Introduction

Power System Blockset (PSB) is a Simulink toolbox that allows dynamic simulation of electric power systems. It models elements like generators, transformers, ...etc.The blockset uses the MATLAB/Simulink environment to represent common components and devices found in electrical power networks [10].

6.2 - Presentation of the Power System Blockset

6.2.1- Block library

Elements in the PSB block library are classified in various groups according to their nature: Electrical Sources, Elements, Power Electronics, Machines, Connectors, and Measurements.

Simulation results can be visualized with Simulink scopes connected to outputs of measurement blocks available in the PSB library. These measurement blocks acts as an interface between the electrical blocks and the Simulink blocks. The voltage and current measurement blocks can be used at selected points in the circuit to convert electrical signals into Simulink signals. Nonlinear elements requiring control, such as power electronic devices, have a Simulink input that allows control from a Simulink system.

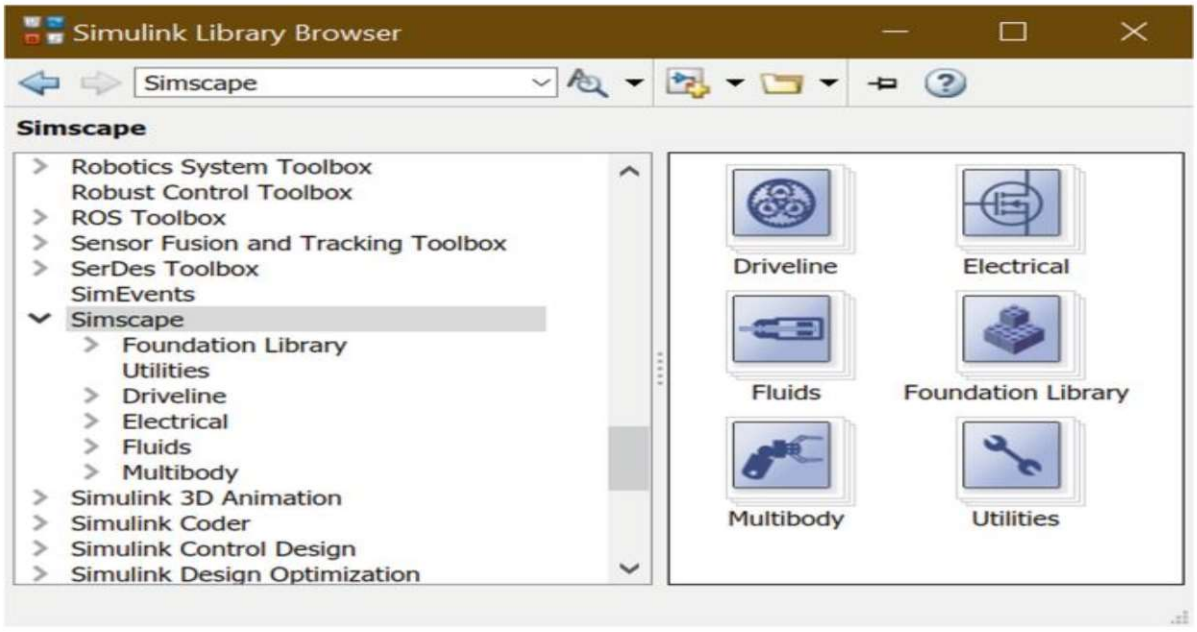


Figure 6.1 *The Power System Blockset Library*

6.2.2 - Study of a simulation example

6.3- Example

This example illustrates a Single-phase half-wave diode rectification (not controlled). This circuit will be used to illustrate simulate an AC/DC converter with different load and configuration.

The circuit built with the PSB schematic diagram is shown on Figure 6.1. The input is simulated with a AC Voltage source. The rectifier is a semiconductor device that converts AC voltage into DC voltage using a single diode. Thus, the analysis of the voltage and current evolutions at the output of the converter and semiconductors with resistive and inductive load, respectively are done. In addition, the study of inductive loads with Freewheeling Diode.

Three oscilloscopes are used to display waveforms while the simulation is running.

A. Resistive Load

For the shown half-wave rectifier, the source is a sinusoid of 100 Vrms at a frequency of 50 Hz. The load resistor is 10 Ω [11], [12].

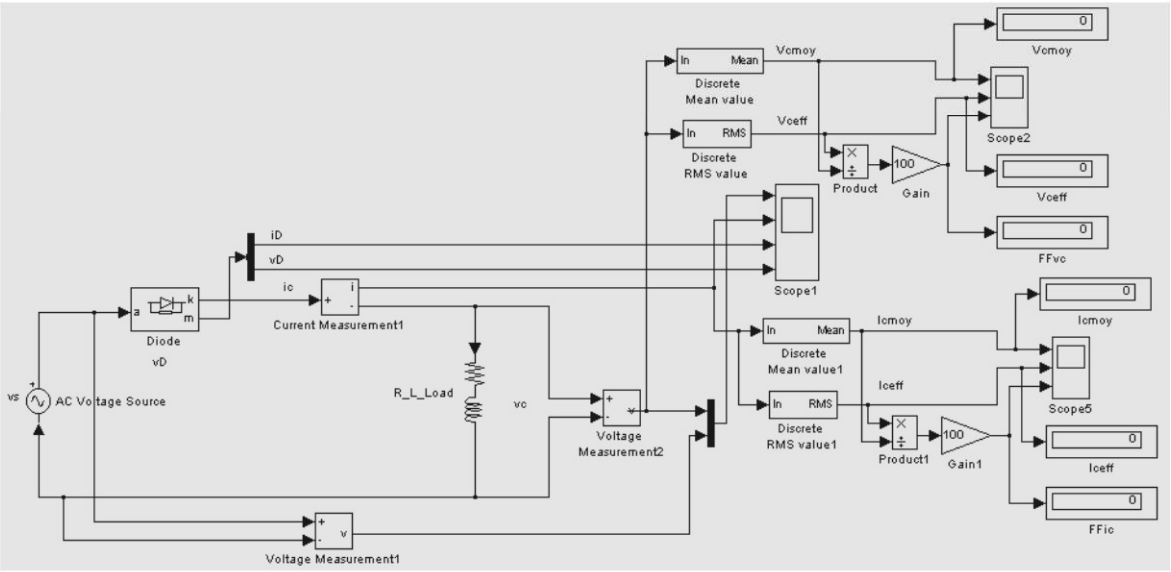


Figure 6.2 *Single Phase Half Wave Rectifier (not controlled)*

Simulation results obtained for a Single Phase Uncontrolled Half Wave Rectifier with Resistive Load are shown on Figure 6.3.

- For the positive half-cycle of the source in this circuit, the diode is on (forward-biased). Considering the diode to be ideal, the voltage across a forward-biased diode is zero and the current is positive. On the other hand, for the negative half-cycle of the source, the diode is reverse-biased, making the current zero. The voltage across the reverse-biased diode is the source voltage, which has a negative value.

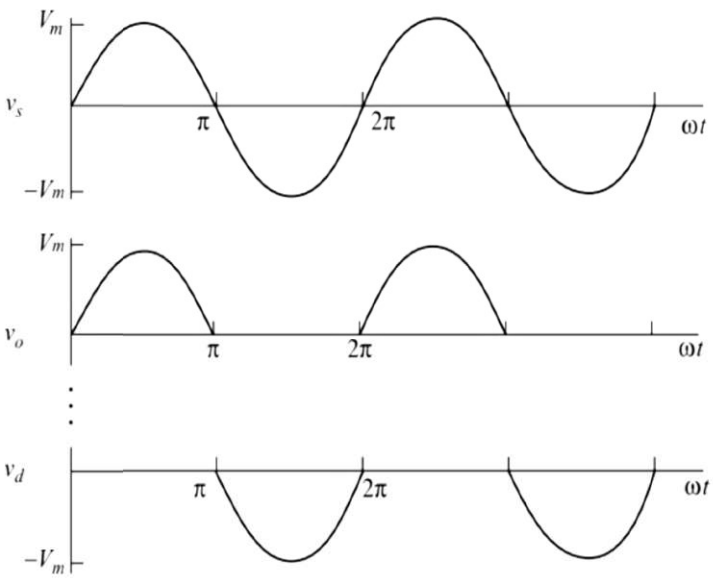


Figure 6.3 *Diagrams of the evolution of the supply voltages, rectified voltage and reverse voltage applied across the diode*

B. Inductive Load

In this case, the half-wave rectifier is the same circuit of Figure 6.2, with slight change in the load (R=10 Ω and L=0.1 H). The simulation results of this case are illustrates in Figure 6.4.

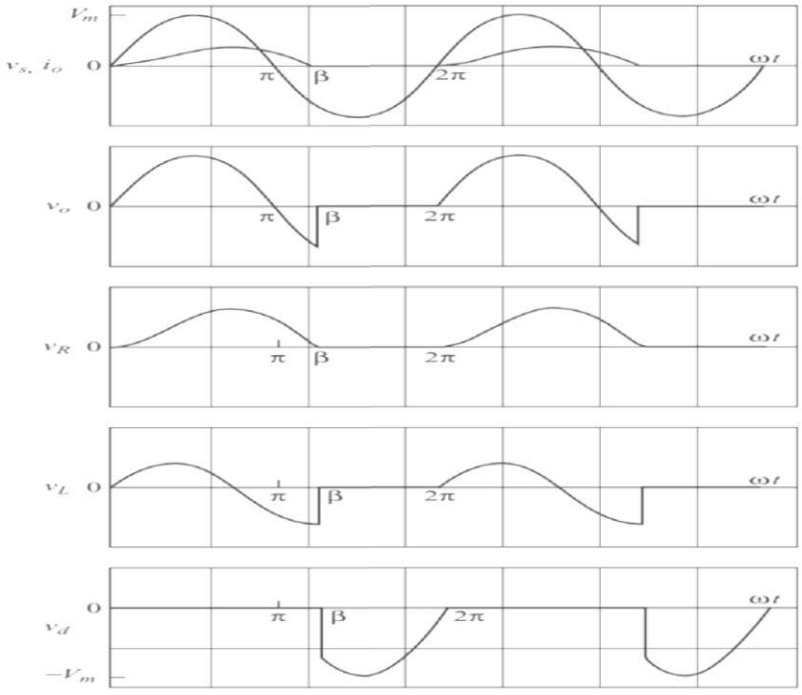


Figure 6.4 *Results of the evolution of the supply voltages, rectified current and voltage, resistance and inductance voltage, and reverse voltage applied across the diode*

C. Inductive Load with Freewheeling Diode

In this case, a freewheeling diode D_2, can be connected across an RL load as shown in Figure 6.5.

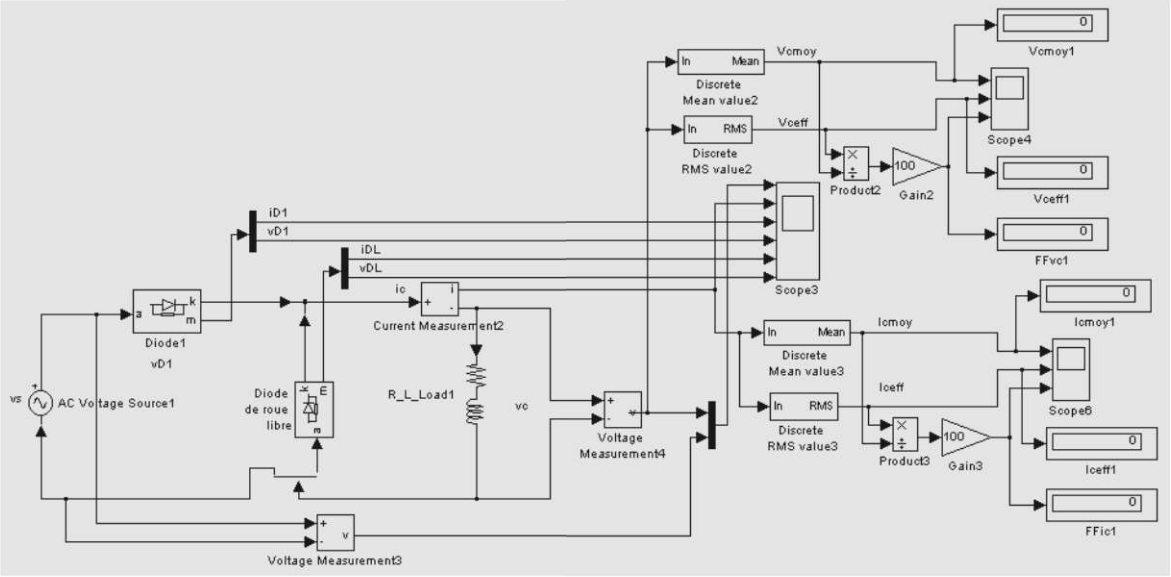


Figure 6.5 *Single-phase half-wave diode rectification (RL load) with freewheeling diode*

➢ Both diodes cannot be forward-biased at the same time. Diode D_1 will be **ON** when the source is positive, and diode D_2 will be **ON** when the source is negative.

- For a positive source voltage ($\mathbf{D_1}$ is on. D_2 is off):

➢ The voltage across the RL load is the same as the source.

- For a negative source voltage ($\mathbf{D_1}$ is off. D_2 is on):

➢ The voltage across the RL load is zero.

Simulation results of steady state load, source, and diode currents are shown in the Figure 6.6

.

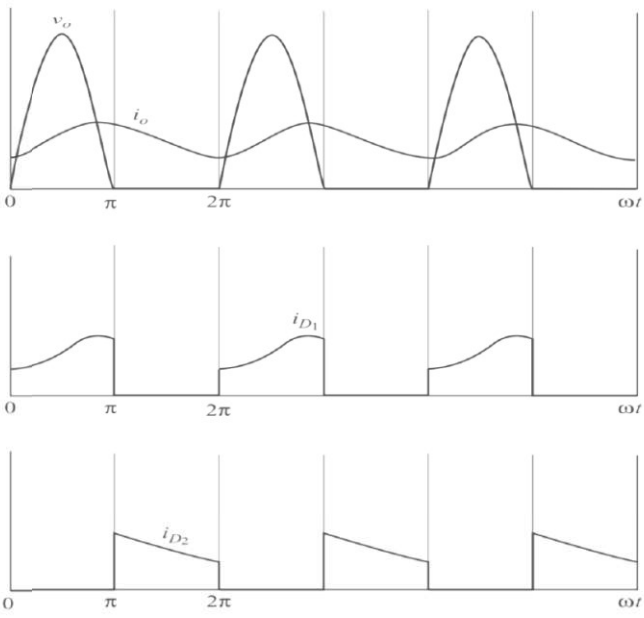


Figure 6.6 *The evolution of rectified current and voltage, current across D_1 and D_2*

Chapter 7: Simulation and co-simulation with other software

7.1-Introduction

The Simulink platform supports co-simulation between components. In Simulink, co-simulation is between components with local solvers or involving simulation tools. For example, co-simulation can be an S-function implemented as a co-simulation gateway between Simulink and third-party tools or custom code, or a Functional Mockup Unit in co-simulation mode imported to Simulink.

In this chapter we will talk about the different software simulation like PSim, PSpice, Proteus,…etc.

7.2 - Simulation by PSim and Simulink-PSim co-simulation

PSim is simulation software specially designed for the analysis and design of power electronics and motor drives. It provides a powerful simulation and design environment for switch mode power supplies, analog/digital control, and electric motor drives.

PSim simulation tool is widely used among research scholars because of its easy to use, fast simulation and flexible control representation [13].

7.2.1 - Setting up Co-Simulation with Matlab/Simulink

PSim can perform co-simulation with Matlab/Simulink. When PSim is installed, if the Sim Coupler Module is enabled in the license, the installation will set up the co-simulation between PSim and Matlab/Simulink automatically. If you change the PSim folder name after PSim is installed, you must set up the co-simulation again by running Utilities >> SimCoupler Setup.

If you are running multiple versions of PSIM or Matlab/Simulink, to associate a specific version of PSim with a specific version of Matlab/Simulink, from that PSim version, run Utilities >> SimCoupler Setup, and select the desired Matlab/Simulink version.

To see how the co-simulation works, follow the steps below to simulate the sample circuit "TP1.slx":

1. From PSim, run Utilities >> SimCoupler Setup. This will set up PSim for the co-simulation. Note that this needs to be run only once.
2. Launch Matlab. From Matlab, launch Simulink.
3. In Simulink, load the file "TP1.slx" from the "examples\Simulink co- simulation" sub-folder in the PSim directory.

4. Double click on the SimCoupler block (the block with the orange color). Make sure that the path for the file "TP1_psim.sch" is correct. If it is not, click on the Browser button and locate the file in the "examples\Simulink co-simulation" sub-folder in the PSim directory.

5. In Simulink, start simulation. After simulation is completed, SIMVIEW will be launched to view waveforms.

To start, double click on the Pspice SV icon and Pspice schematic will open. The figure below is schematic

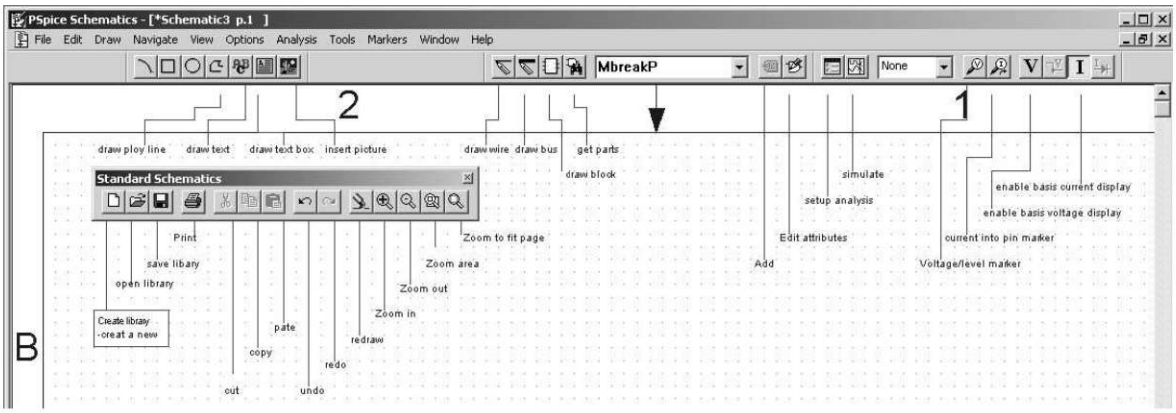


Figure 7.1 *PSpice Schematics*

Example:

7.3 - Simulation with other software

7.3.1 - Search for components

If we select a library, for example "eval", we will appear in the "Part List" the list of components constituting this library. By choosing a component In particular, it can be viewed at the bottom right of the main window. Using the arrows direction of the keyboard, up and down, you can visualize all the components of this library [14],[15].

We select the component, then "Ok".

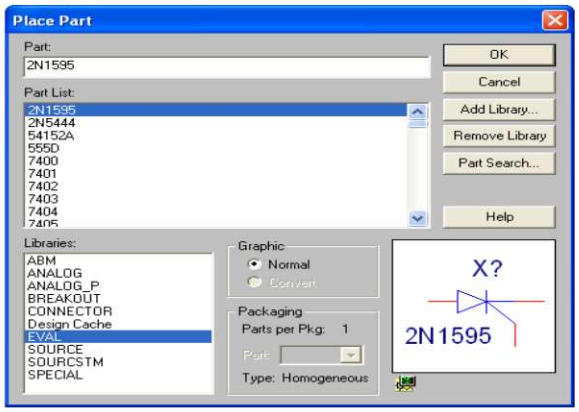


Figure 7.2 *Symbol placement window*

7.3.2 - Automatic search in a library

You can also launch an automatic search directly in the library chosen, by entering the "Part" field. You must use the * character to replace a part of the component name of which we are not certain. we are looking for example for the 2n2222 transistor. Just mention "*22*" in the "Part" field, click on "Ok". The research editor will find:

7.3.3 - Scheme entry

7.3.3.1 - Component Placement

Component placement is done by clicking on the icon (Place Part) or Place→Part, or keyboard shortcut "p".

This opens the following window:

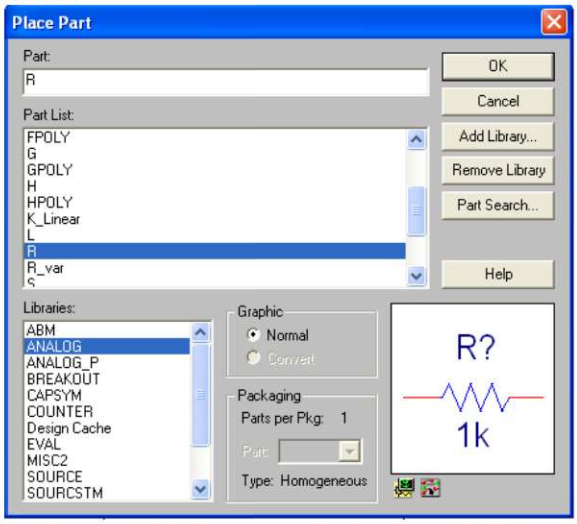


***Figure 7.3** Component selection window*

7.3.3.2 - Simulation

7.3.3.2.1 - Creation of a simulation profile

To carry out a simulation, you must define the parameters of the variable quantities [16].

That is to say the variation range, the calculation step, as well as other information which will be seen later, but we must first create a simulation profile by clicking on the icon (new simulation) or pspice→ new simulation profile, and indicate a name on the window which is displayed:

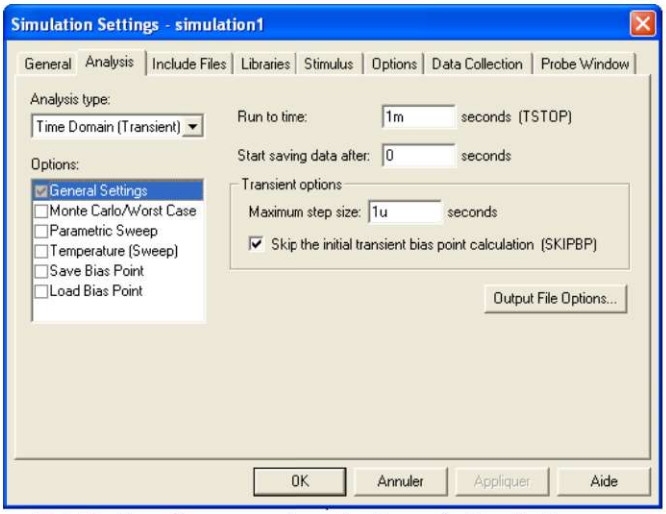


Figure 7.4 *Define the type of analysis and simulation parameters*

7.3.3.2.2 - Simulation types

7.3.3.2.2.1 - Temporal analysis "Time Domain (transient)"

A temporal analysis (transient or TRAN) corresponds to a simulation based on time. The following window explains the (Time Domain) option in detail:

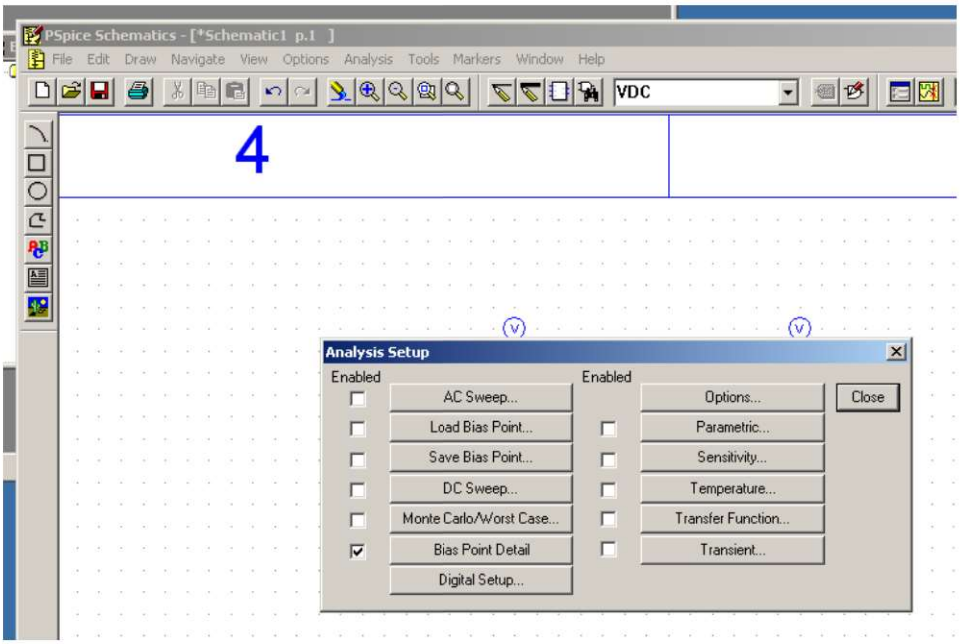


Figure 7.5 *Analysis setup*

References

[1] P.Tona, 'Introduction à MATLAB et SIMULINK', National Higher School of Electrical Engineers of Grenoble, France, 2023.

[2] H. Le-Huy,'Introduction à MATLAB et Simulink', Laval University, Québec, Canada, 1998.

[3] M. Etique, 'Introduction au logiciel MATLAB', Institute of Industrial Automation, canton of Vaud (VD) Switzerland,

[4] C. Vilain, 'Formation Initiation à Matlab', GIPSA-Lab, 2014.

[5] O. Louisnard, 'Initiation à MATLAB', 2003.

[6] https://www.mathworks.com

[7] www.mathcentre.ac.uk, mathcentre 2009.

[8] Alexandre Loye, 'Représentations graphiques', Lausanne University, MATLAB Course UNIL-FGSE – Hiver 2009-2010.

[9] MATLAB help 2021

[10] Gilbert Sybille, Hoang Le-Huy 'Digital simulation of power blockset systems and power electronics using the MATLAB/SIMULINK power system', IEEE Power Engineering Society Winter Meeting, 4(4), 2973 – 2981, 2000.

[11] T. Roubache, 'Power Electronics Workshop', Department of Electrical Engineering, University of M'sila, 2023, Algeria.

[12]F. Obeidat, 'Power Electronics Single Phase Uncontrolled Half Wave Rectifiers', Pheladelphia University, 2023.

[13] https://powersimtech.com.

[14]R. Chabane, K. Bouharati, 'Etude du logiciel de simulation PSpice et ses applications en electronique', State Engineer's Thesis in Electronics, Tizi-Ouzou University, 2010, Algeria.

[15]https://engineering.purdue.edu/~ee255d3/readings.html.

[16]https://powersimtech.com.

APPENDIX

Trigonometric functions

sin, asin, sinh, asinh, cos, acos, cosh, acosh, tan, atan, tanh, atanh, cot, acot, coth, acoth, sec, asec, sech, asech, csc, acsc, csch, acsch,

Basic Mathematical Functions

abs: absolute value or module

angle: argument of a complex

sqrt: square root

real: real part

imag: imaginary part

conj: complex conjugated

gcd: PGCD

lcm: PPCM

round: rounded to the nearest whole number

fix: truncation

floor: rounded towards -∞

ceil: rounded towards +∞

sign: sign of

rem: remainder of division

exp: exponential

log: natural log

log10: log decimal

Column-wise data analysis

max: max value

min: min value

mean: average value

median: median value

std: standard deviation

sort: in ascending order

sum: sum of elements

prod: produces elements

cumsum: vector of cumulative partial sums

cumprod: vector of cumulative partial products

hist: histogram

Options du graphe

title Définir le titre du graphe

xlabel Label de l'axe des x

ylabel Label de l'axe des y

zlabel Label de l'axe des z

legend Ajouter une légende sur le graphe

text Permet d'ajouter du texte sur le graphe

axis Définir xmin, xmax, ymin et ymax du graphe. voir aussi axis equal

Graphiques 2D : principales fonctions

plot Graphe en 2D avec une échelle linéaire

plotyy Graphe avec deux axes y différents à gauche et à droite

loglog Graphe en 2D avec une échelle logarithmique pour les deux axes

semilogx Graphe en 2D avec une échelle logarithmique pour l'axe des x

semilogy Graphe en 2D avec une échelle logarithmique pour l'axe des y

figure, close Permet d'ouvrir ou de fermer une figure

subplot Tracer plusieurs graphes alignés sur une même figure

Graphiques 3D : fonctions usuelles

plot3 Tracé d'une ligne paramétrique en 3D

mesh Tracé d'une surface en 3D, à partir de matrices de maillage

meshgrid Définition de matrices de maillage à partir de deux vecteurs

surf Tracé d'une surface en 3D avec dégradé de couleur, à partir de matrices de maillage

surfc Tracé d'une surface en 3D avec dégradé de couleur et lignes d'iso-valeurs

ezmesh ,ezmeshc Tracé facile de surface (matrices de maillage définies par défaut)

ezsurf , ezsurfc Tracé facile de surface avec dégradé de couleur (matrices de maillage définie par défaut)

sphere Définition de matrices de maillage pour le tracé d'une sphère

cylinder Définition de matrices de maillage pour le tracé d'un cylindre

polynomials

poly: construct a polynomial from the roots

roots: calculation of roots

roots1: calculation of roots

polyval: one-point assessment

polyvalm: evaluation in a point matrix

conv: multiplication

deconv: division

residue: decomposition into simple elements and residues

polyfit: approximation polynomial

polyder: differentiation