UNIT 1

MATLAB Fundamentals and Graphics

1. Introduction

History: MATLAB was originally written by Cleve Moler, founder of MathWorks Inc., with the aim to provide easy access to the matrix software developed in the UNIX projects EISPACK (Eigen system package) and LINPACK (linear system package).

Versions and history: The first version was written in the late 1970s in Fortran. The matrix, which did not need any *a priori* pre-sizing, was its unique data structure. The name “MATLAB” derives from MATrix LABoratory.

MATLAB was rewritten in C in the 1980s. Version 3 for MS-DOS appeared in the early 1990s. Version 4 for Windows 3.11 appeared in 1993 and included the first version of Simulink. The current version (at July 27, 2018) is R2018a.

MATLAB is currently a *de facto* standard in engineering. Dedicated conferences are held at many universities and several companies sell their toolboxes as third parties. Many users also share their programs on the Internet with free access. For more information, refer to www.mathworks.com.

Related software exists, which is compatible with MATLAB, except for some specific tasks or for the automatic GUI generation tools. Notably, GNU-Octave is free software that is almost script-compatible with MATLAB. You can see https://www.gnu.org/software/octave/ for more information.

Main features:

- The MATLAB language is simple, yet powerful and fast. In a typical working session, it is not necessary to compile or to create executable files. Since M-files are text files, the memory requirements are small.

- A number of functions for mathematical operations and for signals and systems analysis are already implemented and available in commercial toolboxes. Users can access the corresponding M-files if they want to modify them. They can also create their own functions and specific toolboxes.

- MATLAB provides powerful tools for visualizing graphics in two and three dimensions, including effects and animations.

- MATLAB allows for the development of complex applications using graphics user interface (GUI) tools.

- MATLAB can communicate with other languages and environments and with hardware components such as audio cards, data acquisition cards and digital signal processors (DSPs).

Constitutive parts: There are three main parts.

- Environment (windows, variables and files).
- Graphic objects.
- Programming language.

Next section presents the MATLAB environment.

2. The MATLAB environment

The environment consists of a set of tools for working as a user or as a programmer. These tools can be used to import/export data, create/modify files, generate graphics and animation effects, and develop user applications.

Windows: There are several window types. The desktop includes the main windows that correspond to the MATLAB core. However, a number of secondary windows are opened and closed in a typical session in order to show figures, user interfaces, variable workspace, file editor, SIMULINK models and libraries, etc. There are also specific help and demo windows.

Variables: They are stored in the workspace but by default they are temporary objects (when the user exits MATLAB, all variables are deleted).

Files: They are steady objects (not deleted when MATLAB is closed) that have their own text editor and folder structure. In addition to the basic files that constitute the basic core of MATLAB, there exist user-created files and commercial files corresponding to the libraries, also called toolboxes. A toolbox is a set of files developed for specific applications, e.g., the *Curve Fitting Toolbox* is designed to find mathematical expressions that fit arbitrary curves.

There are also two “special” toolboxes whose hierarchy is above MATLAB (see Fig. 1): *Simulink*, for the numerical simulation of systems (dynamical, communication, etc.) and *Stateflow*, for the simulation of state machines.

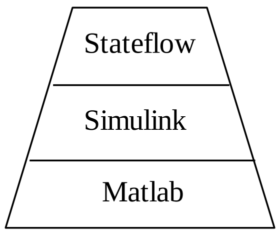
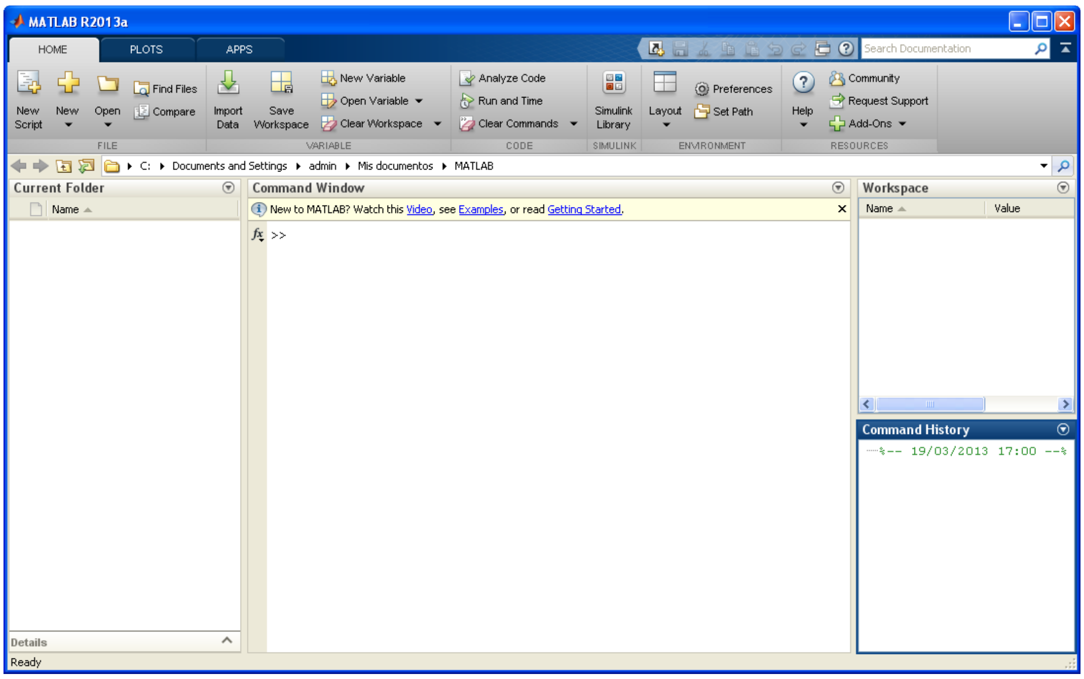


Fig. 1. MATLAB modules

2.1 Windows and desktop

Desktop: When MATLAB is started, a desktop opens as shown in the following figure (which correspond to an older realease, but looks assentially the same as in R2018a).



New to MATLAB? Watch this Video, see Examples, or read Getting Started.

```
fx >>
```

```
%-- 19/03/2013 17:00 --%
```

Fig. 2. MATLAB desktop

Command window: It is the main window. Commands and statements are written after the prompt >>. Operating system instructions can be executed from this window by simply typing the sign "`!`" after the prompt (example: `>>!dir`). The <↑> key can be used to recover previously typed statements.

Other desktop windows: The default desktop also includes the following windows.

- *Command History*: Contains the commands used in previous MATLAB sessions, indicating the date and time.
- *Current Directory*: Lists the files and folders in the current directory. By default, the selected current directory is <work>.
- *Workspace*: Shows the current variables as well as their type and value.

Desktop layout: It is possible to change the desktop layout. The available options can be explored in the menu bar. Changes to the desktop can be saved (*Desktop* → *Save Layout*). The corresponding file is `startup.m`. If this file exists, it is one of the first ones loaded when MATLAB is launched. Another initialization file is `matlabrc.m`, which contains the default values concerning fonts, colours, dimensions, etc. of the different program elements. Preferences can be adjusted from *File* → *Preferences…*

Secondary windows: Apart from desktop windows, other windows open and close during the session as long as commands are executed (or controls are activated) to generate plots, execute demos, etc.

Other commands: Other useful commands are

- `clc`: close command window
- `close [all] [all hidden]`: close (all) figure windows
- `exit, quit`: exit MATLAB
- `diary [on/off]`: export a session on a text file
- `ver`: list the installed toolboxes

For more information, type `>>help command_name.`

2.2 Variables and workspace

Workspace: Within a session, the variables generated by the commands are stored in the workspace. Workspace variables can be modified and/or used in other commands. This storage is temporary and only for the current session. Workspace variables are erased when you exit MATLAB.

Array Editor: To view the contents of a variable **`var1`**, simply type its name in the command window **`>>var1`**. You can also display its contents in the *Array Editor*. To open it, simply go to the *Workspace* window and double click on the icon of the desired variable ⊞var1 . Another way to open the *Array Editor* is to type **`>>open var1`** in the command window.

Importing and exporting variables: The workspace variables can be saved in a data file (`save` command) and this file can then be loaded in a later session (`load` command). The data files in MATLAB have the extension `*.mat`. The default name for the data file is `matlab.mat`.

Variable names: If a name is not assigned to a variable, it receives the default name `ans` (for *answer*). The next unnamed variable will also be called `ans`, so it will substitute the first one.

MATLAB is case sensitive. Lowercase and uppercase names refer to different variables. Hence, `a` and `A` correspond to two different variables.

It is recommended not to assign a name to a variable that refers also to a function. A common error is to create a variable with the name `axis` and then to try to execute the function `axis`. In such a case, MATLAB gives an error message because it considers `axis` to be a variable, not a function. To solve this problem, type `>>clear axis`. To check if a name corresponds to a function, you can simply type: `>> help name`.

Other commands: Other useful commands include

- `who,whos`: to see the variables in the workspace
- `size`: to see a matrix dimension (`>>[nrow,ncol]=size(A)`)
- `length`: to see a vector length (`>>N=length(v)`)
- `clear [all]`: to erase variables (`>>clear` erases all of them, `>>clear var1` only erases the variable `var1`)

For more information, type `>>help command_name.`

Example 1. Command window and workspace

```
» x=[1 2 3]

x =

     1     2     3                 Semicolon

» y=[4;5;6];  <-
»
» who

Your variables are:

x         y

» A=[1 2 3;4 -5 6;7 8 9];
» q=y'  <-                         Transpose

q =

     4     5     6

» x.*q
     <-                            Element-to-element operation
ans =

     4    10    18

» whos
  Name      Size         Bytes  Class

  A         3x3             72  double array
  ans       1x3             24  double array
  q         1x3             24  double array
  x         1x3             24  double array
  y         3x1             24  double array

Grand total is 21 elements using 168 bytes

»
```

Practice: Variables and workspace (I). Entering and viewing data

Variable generation: In the command window, enter a scalar, a matrix, a character string and a couple of commands. For instance:

```
>> x=2
>> A=[1 2;3 4;5 6];
>> A
>> s='hello'
>> a=2/0
>> 0/0
```

Notice that the typed commands are saved in the command history window.

Answer the following questions.

What is the semicolon for?
Does MATLAB distinguish between lowercase and uppercase?
If you type `>> y=40.5` and later on `>> y=102.3`, what happens to variable `y`?
What does `ans` mean?
What does `NaN` mean?
Now type `>>who` and `>>whos`. What do these two commands do?

Practice: Variables and workspace (II). Save and load.

To save variables `a` and `A` to the file `bah.mat`:

```
   >> save bah a A     or     >> save('bah','a','A')
```

Delete `s`

```
   >> clear s
   >> who
```

Delete all variables

```
   >> clear, who
```

Load a data file `bah.mat`

```
   >> load bah, who
```

For more details, type `>> help command_name`

2.3 Files and search path

While variables are the temporary information of MATLAB, files constitute the permanent information that is not deleted when a session is closed.

File types: There are several types of files.

- Data files: Files with extension `*.mat`. Several formats are available (ASCII, binary, etc.) and they are created and loaded by executing the `save` and `load` commands, respectively, or using the options on the workspace window menu bar. The default name for data files is `matlab.mat`.

MATLAB can also import data from other types of files (`*.txt`, `*.xls`, etc.). If the file in question is in the Current Directory, simply click on its icon to open the Import Wizard (see Exercise 6). Data can also be accessed from a file through the command window, for example, `>>ImportData('test.txt')`. Other related functions are `fopen`, `fread`, `fprintf`, ...

- M-files: Text files with the extension `*.m` contain commands as if they were written in the MATLAB command window. M-files can be created by the user and are stored by default in the working directory `<work>`. To edit the M-files, it is advisable to use MATLAB's M-file editor. However, any other text editor (for example, Windows Notepad) is also valid.

Commercial M-files are organized in libraries named toolboxes (each toolbox is stored in a different folder) and the user can edit and modify them, too.

In general, each command or MATLAB function corresponds to a file. For instance, the `roots` function corresponds to the `roots.m` file.

- Built-in files: These files cannot be modified by the user, their extensions are diverse (`*.dll`, `*.exe`, `*.mex`) and they correspond to the files of the MATLAB kernel (for example, the code of function `who` is not available to the user. The file `who.m` only contains the help of this function).

Names for files: MATLAB is case sensitive, e.g., see the difference between `>>who` and `>>WHO`. In general, functions should be typed in lowercase.

A name like `exercise1-2.m` is not valid because MATLAB will try to take `2` from the variable named `exercise1`.

Search path: The user created files are saved by default in the <work> directory, but you can save files anywhere in the directory tree. To force MATLAB to consider the new location, we have to widen its path. To do so, select `File → Set Path...` in the main menu. Or click on the icon next to the Current Directory in the desktop toolbar.

You can also use the function `addpath` to make MATLAB take into account the functions of specific directories for the current session. For example:

```
addpath c:\matlab\work\my_folder
addpath d:\projects
```

Scripts and functions: The power of MATLAB relies on the possibility of executing a large series of commands stored in a file. Such files are the so-called *M-files* since their name extension is *m*, *filename.m*. Apart from the commercial files provided in the toolboxes, users can create their own files. M-files can be script files or function files:

- Scripts declare neither input arguments nor output arguments. To execute script files, simply type their name (without an extension) in the command window or click on the (save and run) button in the M-file editor.

- Functions have input and output arguments declared. The function name is usually written in lowercase. Input arguments are inside parentheses ( ) while output arguments are inside brackets [ ].

Other commands: Other useful commands related to files are

▪ `what`:	lists the files that are in the current directory
▪ `which`:	to see the complete path for a function (`>>which bode`)
▪ `lookfor`:	to search for functions (see next example)
▪ `type`:	to see the M-files code (`>>type roots`)

For more information, type `>>help command_name.`

Example 2. Looking for functions

If we want to compute the determinant of a matrix but we do not remember the particular MATLAB function, the function `lookfor` can be used (this function searches the required word in all of the names and help sentences of the functions).

```
>> lookfor determinant
DET    Determinant.
DET    Symbolic matrix determinant.
DRAMADAH Matrix of zeros and ones with large determinant or inverse.
>>
>> help det

 DET    Determinant.
    DET(X) is the determinant of the square matrix X.

    Use COND instead of DET to test for matrix singularity.

    See also COND.

 Overloaded methods
    help sym/det.m

>>

>> det([1 3;-2 4])

ans =

    10

>> type det
det is a built-in function.
>>
```

Practice: Toolboxes and search path

1) Identify the default working folder (see Current Directory window or toolbar).

2) Identify which MATLAB version is installed and which commercial toolboxes are available. To do so, type `>>ver` in the command window.

3) See the folder tree by using the menu options `File` → `Set Path…` See where the files that correspond to the different toolboxes are stored.

4) Add any folder to the MATLAB path.

5) Use `which` to determine in which folder the file that corresponds to function `fminsearch` is stored. Idem with function `roots`.

6) See what functions `fminsearch` and `roots` are for (function `help`) and see if their code can be edited by the user (function `type`).

2.4 Helps and demos

MATLAB presents several help levels. All functions and installed toolboxes can be listed on the first level by using the command:

```
» help
```

On the second level, information about all of the functions of a particular toolbox can be retrieved by typing the command:

```
» help toolbox_name     (» help stats)
```

Finally, to obtain complete information about the usage and syntax of each function, use:

```
» help command_name     (» help bode)
```

In addition, the demonstration files (demos) are a set of scripts that offer a perspective of the toolboxes through the automatic execution of the most representative functions.

3. Programming language

3.1 Basic symbols and commands

Matrix: The basic computational unit in MATLAB is the rectangular matrix with real or complex elements. Square dimensional matrices, vectors and scalar variables are considered to be particular cases of this basic data structure.

Matrix input: Matrix elements are entered inside brackets `[ ]`. Elements from the same row can be separated with a space or a comma. A different row is specified by using semicolons or the Enter key ↵.

Example: Entering a matrix.

```
» A=[1 2;3 4]      (key ↵)
A =
      1   2
      3   4
```

Semicolon: It avoids the presentation of results in the command window.

```
» A=[1 2;3 4];     (key ↵)
»
```

Submatrix reference: Given a matrix **A**, if we want to pick out the sub-matrix **B** defined in rows 3 to 5 and columns 4 to 7 of matrix **A**, two possibilities are

```
»B=A(3:5,4:7)     or     »B=A([3,4,5],[4 5 6 7])
```

To obtain sub-matrix **C** defined in rows 1, 3 and 4 of **A**, the command is

```
»C=A([1 3 4],:)
```

The comma separates the specification of rows from the specification of columns. Notice that the colon symbol **:** after the comma means “all columns”.

Special symbols: There are several predefined variables. Imaginary number $\sqrt{-1}$ can be expressed either as **`1i`** or **`1j`**, **`Inf`** corresponds to $+\infty$ and **`pi`** refers to π. **`NaN`** (*Not-a-Number*) is obtained in non-definite operations, such as 0/0.

Example: Rectangular matrix with special symbols.

```
» A = [sin(pi/2) -4*j 9;sqrt(2) 0/0 log(0)]
Warning:    Divide by zero
Warning:    Log of zero
A =
      1.0000            0-4.0000i          9.0000
      1.4142         NaN                     -Inf
```

Ans: When no output variable is specified (the command is simply `»function_name(…)`), MATLAB assigns the result to the `ans` (for *answer*) variable.

```
» 12.4/6.9   (key ↵)
ans =
      1.7971
```

Stop execution: Key combination `<Control> <c>`.

Command syntax: MATLAB is interactive software in the sense that it establishes a “dialogue” with the user by means of a language based on commands or statements. In response to a statement, MATLAB executes it or gives the corresponding error message.

It is possible to enter several commands in the same row. These commands must be separated by commas or semicolons.

Commands can be

- Mathematical operations
- Script invocations (to execute a script, simply type the script name – no extension is needed)
- Function invocations (to execute a function, you must specify input and/or output arguments)

Calling functions: Input arguments are inside parentheses and output arguments (if there are two or more) are inside brackets.

```
» [output_arg]=function_name(input_arg)   (key ↵)
» function_name(input_arg)     (key ↵)
```

In a typical session, each command produces new variables that are stored in the workspace. These variables can be used later as input arguments for new commands introduced by the user. When the user exits MATLAB, all variables in the workspace are deleted.

Example 3. Function syntax

Although the input and output variables can have any name, in order to interpret the result their order is important. For instance, function **`eig`** computes both the eigenvalues and eigenvectors of a given matrix.

Function **`eig`** with only one output argument (or with no output argument at all), computes the eigenvalues:

```
>> A=[-3 2;0 1];
>> eig(A)
ans =
    -3
     1
```

If you call **`eig`** with two output arguments, the function gives a first output variable containing the modal matrix (a matrix whose columns are the eigenvalues) and a second output variable containing the Jordan form of input matrix **A** (that is, a diagonal matrix with eigenvales in the main diagonal):

```
>> [eigenvect,diag]=eig(A)

eigenvect =
    1.0000    0.4472
         0    0.8944

diag =
    -3     0
     0     1

>>
```

In fact, most functions give different results depending on the number of input and/or output arguments. Therefore, it is advisable to check the help file before using a function for the first time, e.g. **`>>help eig`**.

Special matrices generation: It is possible to generate matrices (or vectors) whose elements are equal to “1” (function **`ones`**) or equal to “0” (function **`zeros`**). It is also possible to generate matrices of random elements: use **`randn`** to generate matrices of Gaussian-distributed elements or **`rand`** to generate uniform distributed numbers. Identity matrices can be generated with the **`eye`** function.

Generation of vectors with equally spaced elements: The two main functions are **`linspace`** and the colon **`:`**. Notice that the former can be used to specify the number of points, while the latter can be used to specify the step between samples.

```
>> x=linspace(0,1,5)   %generate 5 points between 0 and 1
```

```
x =
         0    0.2500    0.5000    0.7500    1.0000

>> x=linspace(0,1);   %100 points between 0 and 1  (default)
```

```
>> y=0:5
y =
     0     1     2     3     4     5

>> y=0:0.5:5
y =
  Columns 1 through 7
    0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
  Columns 8 through 11
    3.5000    4.0000    4.5000    5.0000
```

Polynomials and transfer functions: Polynomials are entered as row vectors where the elements are the polynomial coefficients.

For instance, polynomial $s^4 + 5s^2 + 3s - 10$ is introduced as:

```
» polinomio=[1 0 5 3 -10];
```

Notice that the coefficient corresponding to s^3 is zero.

Transfer functions are introduced by separately typing the numerator and denominator polynomials. Thus the function

$$H(s) = \frac{s}{s^3 + 2s + 5}$$

will be defined by the instructions

```
»num = [1 0];
»den = [1 0 2 5];
```

It is also possible to combine the numerator and denominator into a single variable of class **tf** (transfer function).

```
»H = tf (num, den)
Transfer function:
     s
-------------
s ^ 3 + 2 s + 5
```

Polynomial product: To multiply polynomials, you can use the convolution product function, **conv**. It only allows two input arguments at a time but the functions can be nested. To get the product $(s+2)\cdot(s^2+4s)\cdot(3s^2+7s+2)$, type the following command:

```
>> conv([1 2],conv([1 4 0],[3 7 2]))
ans =
     3    25    68    68    16     0
```

That is, $3s^5 + 25s^4 + 68s^3 + 68s^2 + 16s$.

Other data types: If you want to store only one variable data of different types, you can use the *struct* or *cell array* variables.

For instance, to store the maximum temperature of several days along with the labels “day” and “max temp”, you can use a *cell* type variable. Type `>>help cell` for more information.

```
>> temp={'day','max temp';1,25.7;2,25.5;3,25.4}

temp = 

    'day'     'max temp'
    [  1]     [ 25.7000]
    [  2]     [ 25.5000]
    [  3]     [ 25.4000]
```

To organize different information types using several fields, you can use *struct* type variables. Type `>>help struct` for more information. For instance, if you want to save the input and output records of an experiment along with the realization date, type:

```
>> x=0:10;
>> y=2*x;

>> experiment.input=x;
>> experiment.output=y;
>> experiment.date='1mar10';
>> experiment

experiment = 
      input: [0 1 2 3 4 5 6 7 8 9 10]
     output: [0 2 4 6 8 10 12 14 16 18 20]
       date: '1mar10'
```

Format: The **`format`** command can be used to change the numerical format in which MATLAB presents the results. Several formats are

`short`: fixed comma with 4 decimals (default)
`long`: fixed comma with 15 decimals
`bank`: two decimals
`rational`: to show the rational numbers as the ratio of two integers.

Two useful instructions for submitting results are **`disp`** and **`sprintf`**:

```
>> disp 'hello'
hello
>> disp ('hello')
hello
>> disp(sprintf('\n\t\t\t Table 1'))

                   Table 1
```

Click `>>help command_name` for more information.

3.2 Mathematical functions and table of operators

Next tables show the basic operators and functions.

Mathematical operators	
`+`	sum
`-`	subtract
`*`	product
`/`	right division
`\`	left division
`^`	power
`'`	conjugate transpose

Table 1.1

Basic functions	
`abs`	absolute value
`angle`	phase
`sqrt`	square root
`real`	real part
`imag`	imaginary part
`conj`	conjugate
`exp`	exponential basis *e*
`log`	natural logarithm
`log10`	basis 10 logarithm

Table 1.2

Trigonometric functions	
`sin`	sine
`cos`	cosine
`asin`	arcsin
`acos`	arcosine
`tan`	tangent
`atan`	arctangent
`sinh`	hyperbolic sine
`cosh`	hyperbolic cosine
`tanh`	hyperbolic tangent

Table 1.3

A dot before a mathematical operator: A dot before an operator between two vectors or matrices indicates that the operation must be performed element-to-element. For instance, `A*B` is the matrix product of matrices **A** and **B**. If **A** and **B** have the same dimension, typing `C=A.*B` results in a **C** matrix whose c_{ij} elements are computed as $c_{ij}=a_{ij}\times b_{ij}$.

3.3 Logical operators

The instructions in MATLAB may undergo a process of programming, such as running a particular set of functions only if it satisfies some Boolean condition, performing loops, etc.

Boole algebra: Relational operators are `==` (equality), `<`, `<=` (less than, less than or equal to) and `>`, `>=` (greater than, greater than or equal to). There are also commands for Boolean comparisons, e.g., `gt(i,1)` means i is greater than 1.

Boolean conditions are expressed inside parentheses, for instance, (`a==2`). If this condition is true, the output value is “1”; if it is false, the output value is “0”.

Logical operators are & (*and*), `|` (*or*) and ~ (*not*); `xor` is not an operator but a function.

Flux control: Basic instructions for the flux control are `for...end`, `if... elseif... else...end`, `while...end`, `switch...case...otherwise...end`.

Example: We determine the sum of a vector’s components by using a loop.

```
»x=[1 2 3 4 5];
»add=0;
»for i=1:length(x)
          add=add+x(i);
end
»add
add =
      15
```

(Note: this loop was for illustration purposes only. To determine the sum of all the vector elements we can simply do: `sum([1 2 3 4 5])`)

Parentheses and brackets: Note that in the above example, we used parentheses to access the "i" element of the vector x. Parentheses ( ) are only used to index, to set a priority on mathematical operations and to contain the input parameters of functions. Brackets [ ] are used to define vectors and matrices and to contain the output variables in functions that give more than one.

Helpful instructions: Two instructions that are very useful are `size` (returns the size of a variable) and `length` (if the variable is a vector, it returns its length), since they are used to verify that the commands are running correctly. Also useful in programming is the `find` function, which returns the index of the elements within a vector or matrix that satisfy a given Boolean condition.

3.4 Functions created by the user

For the user, editing functions makes sense in situations when the same program structure will be used several times for different parameters that can be introduced externally.

To create the file containing the commands to be executed by our function or script, open a text editor and write the sequence of commands to be executed. In the case of a function, the file name has to coincide with the function name (e.g., `func1.m`). An *M-file* is structured in three parts:

Calling: Use the following syntax to call a function.

```
»[output_arguments] = function_name (input_arguments)
```

Structure: An M-file consists of three parts.

- Header (only in functions): `function` [output_args]=function_name(input_args)

```
function [sine,cosine,tangent]=func1(ang)
```

- Help comments (optional): Help comments appear in the command window when you type `>>help function_name`

```
% FUNC1 Test function
% [sine,cosine,tangent]=func1(ang) computes the sine,
% cosine and tangent of the angle 'ang'
```

- Commands collection:

```
sine=sin(ang);
cosine=cos(ang);
tangent=tan(ang);
```

Practice: Creation of functions by the user
1) Open the M-file editor (click on 🗋) and write the instructions of the previous example.
2) Save the file using the same name as the function (`func1.m` in our case).
3) Type >> `help func1` in the command window.
4) Call the function `func1` for various values of `ang` and see the results.

3.5 Generation of simple plots

To generate a simple *x-y* plot in MATLAB, follow these steps:

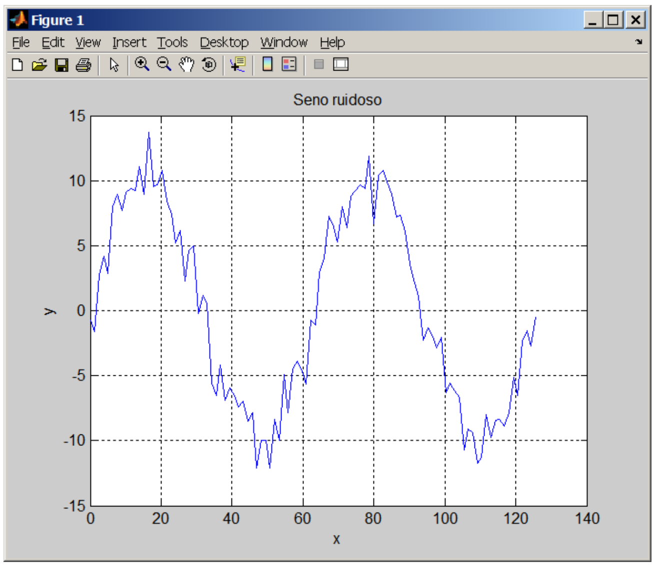
1. Generate (or load) a vector **`x`** containing the abscissas axis values.
2. Generate (or load) a vector **`y`** containing the y-axis values. The lengths of vectors **`x`** and **`y`** must be the same. You can also use names other than **`x`** and **`y`**.
3. Execute any plot command, e.g., **`>>plot(x,y)`**
4. Optionally, you can grid the plot **`(>>grid`**), label the axes (**`xlabel`**, **`ylabel`**) and insert a title (**`title`**).

Note: To draw two plots in the same figure, type **`>>plot(x1,y1,x2,y2).`**

Example 4. Simple plot

Next, we generate and plot two cycles of a sinusoid of frequency 0.1rad/s and amplitude 10 that is corrupted by additive white Gaussian noise with zero mean and variance 3:

```
>> x=linspace(0,2*2*pi/0.1);
>> y=10*sin(0.1*x)+randn(size(x))*sqrt(3);
>> plot(x,y)
>> grid,xlabel('x'),ylabel('y'),title('Seno ruidoso')
```



To copy the figure in a document (Word, for example), simply select *Edit* → *Copy Figure* in the figure window menu bar.

3.6 Using toolboxes

To check which toolboxes are available in your MATLAB, simply type `>>ver`. Here we illustrate the use of some functions belonging to the *Signal Processing Toolbox*.

Example 5. Signal spectrum. Power spectral density

Let us obtain and plot the power spectral densities (PSDs) for the following signals:

Multisinusoid: $x(t) = 0.5\,\mathrm{sen}(2\pi 3t) - 0.2\,\mathrm{sen}(2\pi 9t)$.
Square signal with frequency 0.3Hz (T = 3.33s) and pulse width 0.3s (duty cycle of 10%).

Solution:
First, we generate a time vector of length $N = 2^n$, n integer. For instance,

```
t=0:0.01:10.23;
```

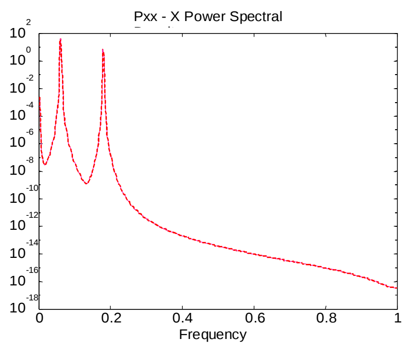
contains 1024 samples. Note that the sampling period is $T_s = 0.01s$ and hence the sampling frequency is $f_s = \frac{1}{T_s} = 100Hz$.

Second, we generate and plot the multisinusoid signal and the square signal:

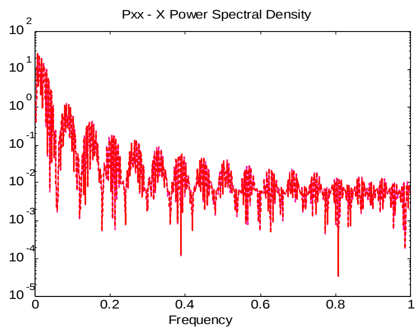
```
x1=0.5*sin(2*pi*3*t)-0.2*sin(2*pi*9*t);     plot(t,x1)
x2=0.5*square(2*pi*0.3*t,10)+0.5;           plot(t,x2)
```

To obtain the PSD, we use the function **`spectrum`**. To plot the results, we use **`specplot`**.

```
px1=spectrum(x1,1024);specplot(px1)
px2=spectrum(x2,1024);specplot(px2)
```



PSD for the multisinusoid.



PSD for the square signal.

Note that **`specplot`** plots the spectrum up to Nyquist frequency $f_s/2$. Moreover, it is normalized to 1. In the case of the multisinusoid signal, if we use the function **`ginput`**, we obtain the normalized frequencies $f_{1n} = 0.06$ and $f_{2n} = 0.18$. If we multiply them by $f_s/2 = 50$, the result is $f_1 = 3$ and $f_2 = 9$, as we might expect.

Example 6. White noise autocorrelation

Let us compute and plot the autocorrelation for a zero mean Gaussian distributed white noise with covariance 4.

Solution:
First, we generate a time vector of length $N = 2^n$, n integer. For instance,

```
Ts=0.01;fs=1/Ts;fn=fs/2;
t=0:Ts:10.23;
```

contains 1024 samples. The sampling period is $T_s = 0.01s$. Hence, the sampling frequency is $f_s = \frac{1}{T_s} = 100 Hz$ and the Nyquist frequency is $f_N = \frac{f_s}{2} = 50 Hz$.

Second, we generate (and plot) the noise sequence,

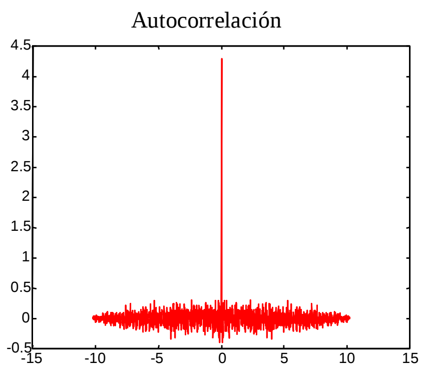
```
n=randn(size(t))*sqrt(4);plot(t,n)
```

and we check the mean and covariance values

```
mean(n),cov(n)
```

The autocorrelation sequence is obtained using the function **`xcorr`**. Its length is twice minus one the length of the original noise sequence. Use **`ginput`** or **`max`** to check that the central value coincides with the signal power.

```
rn=xcorr(n,'biased');
plot(-10.23:0.01:10.23,rn)
```



---

4. MATLAB graphic objects

4.1 Introduction

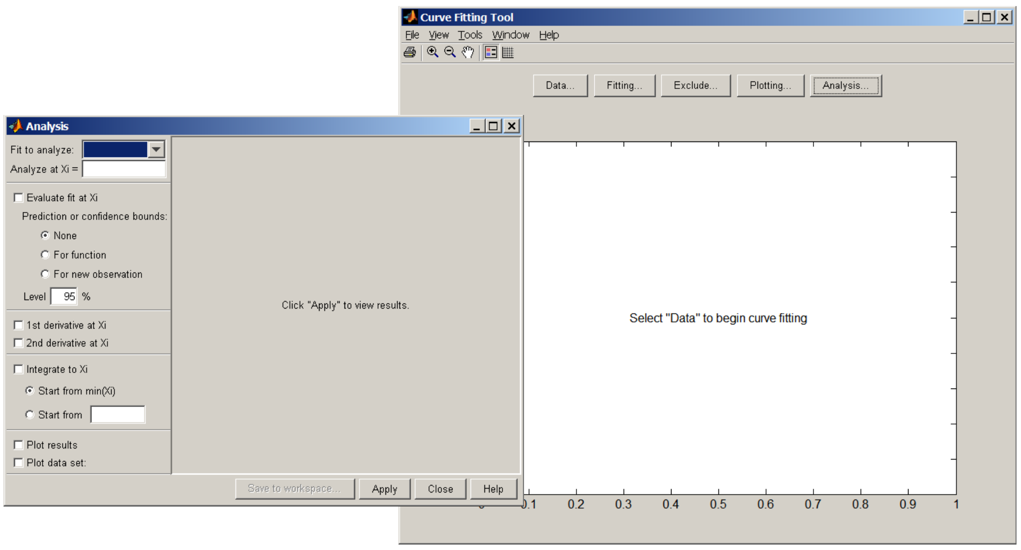
To get information about the functions related to MATLAB graphics, type:

```
>> help graph2d    %Two-dimension (2D) plots
>> help graph3d    %Three-dimension (3D) plots
>> help specgraph  %specific plots

>> help graphics   %low level commands
```

MATLAB Handle Graphics allows:

- High-level commands for data presentation. This includes two-dimensional and three-dimensional plots, photograph/image processing, specific charts for presentations (such as bar diagrams or pie diagrams), and special effects (movies, lighting, camera movements).

- Low-level commands for creating and modifying objects. This constitutes the so-called graphics user interface (GUI) for developing complex applications, consisting of figures, menu bars and control elements (such as buttons). With these utilities you can design windows like the ones shown in Fig. 3. These two windows correspond to the Curve Fitting Toolbox. The command is **`>>cftool`**.



Click "Apply" to view results.

Select "Data" to begin curve fitting

Fig. 3. Example of a graphics user interface

In the first case you work as a user whereas in the second case you work as a software developer. The second case refers to a more advanced level of MATLAB, which will be discussed in Unit 4. In this unit we introduce the two basic commands **`set`** and **`get`**.

MATLAB graphic objects present the following hierarchy:

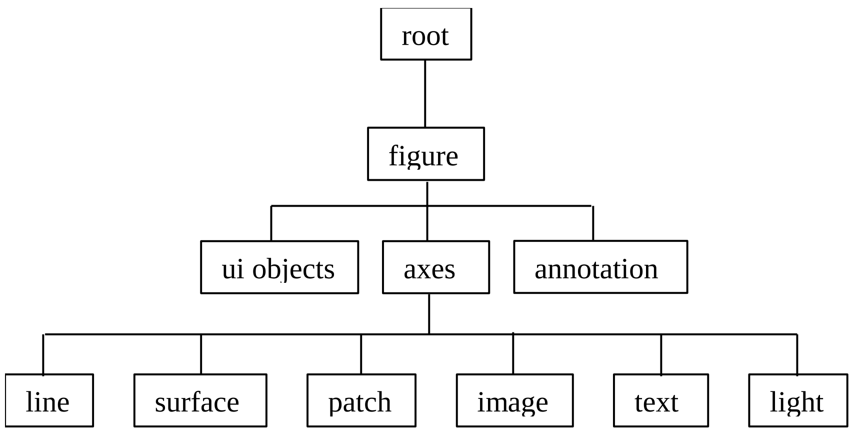


Fig. 4. Hierarchy of graphic objects in MATLAB.

(Note: There are more objects and groups of objects but they are not shown here to make the presentation clearer. For more details, see "graphics objects" in the MATLAB help file.)

The object **`root`** is the command window. If **`root`** does not exist then neither do the objects shown in Fig. 4. In other words, if MATLAB is not active, it is not possible to generate **`figure`**, **`axes`**, etc.

When we type **`>>plot(t,y)`** in the **`root`** object, all objects needed for the representation are automatically generated, **`figure`** → **`axes`** → **`line`**, if they did not already exist.

```
>> x=0:10;
>> plot(x,x)
```

Handle: Every graphical object is associated to a “handle” (a number that identifies the object) and to a set of “properties” (**`color`**, **`Position`**, etc.). Some properties can be modified by the user, whereas others cannot. The handle for the object **`root`** is 0.

To get the handle of a **`figure`** object, you can use the **`gcf`** command (get current figure).

```
>> gcf
ans =
     1
```

To get the handle of an **`axes`** object, you can use the **`gca`** command (get current axes).

```
>> gca
ans =
  158.0017
```

To see all of the properties of an object, you can use get, e.g., **`get(gca)`**

```
>> get(gca)
        ActivePositionProperty = outerposition
        ALim = [0 1]
         ⋮
        ZTickMode = auto

        BeingDeleted = off
         ⋮
        Visible = on
```

To change the object properties, you can use set, e.g.,

```
>> set(gcf,'NumberTitle','off')
>> set(gcf,'Color','r')
```

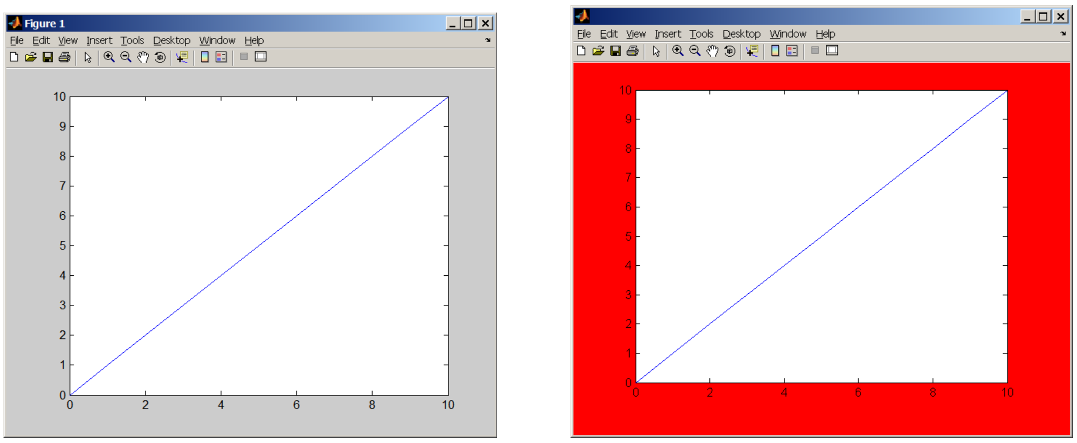


Fig. 5. Using the low-level commands: **`get`** and **`set`**.

These options are also available from the figure window menu bar: **`Edit`** → **`Figure Properties…`**.

4.2 LINE object

2D plots: General steps for two-dimensional curve plotting are listed below. However, it is often not necessary to follow all of these steps explicitly because the functions available in commercial toolboxes automatically execute some of them.

Step 1) x axis: Generate a vector with the values for the abscissas axis.

Functions: two points (**`:`**), **`linspace`**, **`logspace`**.

For instance,

`>>x=0:.2:12;` (initial value : space between values : final value)

or alternatively,

`>>x=linspace(0,12,200);` (200 equally spaced values, 0 being the first value and 12 being the last)

When logarithmic scales are needed, e.g., for a Bode diagram, we can use the function **`logspace`**:

`>>w=logspace(-1,3);` (50 logarithmically spaced values between 10^{-1}=0.1 and 10^{3}=1000)

Note that the third input (vector length) argument in both **`linspace`** and **`logspace`** is optional. Default values are 100 and 50, respectively. Also, in function **`:`**, if the step between values is not specified, the default value is 1 (e.g., **`>>x=1:3`** generates a row vector with elements **`1 2 3`**).

Step 2) y axis: Generate a vector containing the values corresponding to the y-axis.

Note that vectors x and y must be of equal length. Actually, `y` is often computed as a function of `x`, which means that the dimension compatibility is guaranteed.

For instance,

```
>>y1=bessel(1,x);
>>y2=bessel(2,x);
>>y3=bessel(3,x);
```

If we want to plot a constant value along the x axis, we can type **`>>plot(x,2*ones(size(x))`**. The function **`ones`** generates a matrix with the same dimension as `x` and with all components equal to "1". Another possibility is **`>>plot(x,x*0+2).`**

Step 3) Plot: Call a graphics generation command.

Functions: **`plot`**, **`semilogx`**, **`semilogy`**, **`loglog`**, **`polar`**, **`plotyy`**, **`stem`**, **`stairs`**.

For instance:

```
>>plot(x,y1,x,y2,x,y3)
```

When working with complex numbers (e.g., **`>>n=3+j*5;`**), statement **`>>plot(n,'x')`** is equivalent to **`>>plot(real(n),imag(n),'x').`**

Step 4) Zoom: To change the axes values, you can use **`axis`** and **`zoom`** (**`zoom on`** and **`zoom off`**). The syntax for **`axis`** is as follows:

```
axis([xmin xmax ymin ymax])
```

The following are also possible:

```
axis square
axis normal
```

Step 5) Grid: To add a grid, you can use function **`grid`** (also **`grid on`** and **`grid off`**). See also **`box on/off`** and **`axis on/off`**.

```
>> th=linspace(0,2*pi,101);
>> x=sin(th);
>> y=sin(2*th+pi/4);
>> plot(x,y,'k-')
>> grid on
>> box off
>> axis off
```

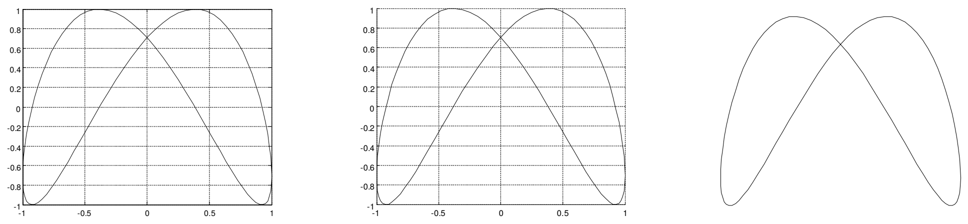


Fig. 6. Grid, box off and axis off

Step 6) Line properties: It is possible to change the line style (solid or not) and use different colours and symbols. We recommend exploring the statement **`>>help plot`**. Different options can be combined (the order is not important) and are called inside symbol `‘’` (`‘*r’`).

Colours: `‘r’, ‘b’, ‘g’, ‘m’, ‘k’, ‘c’, ‘y’,...`
Line style: `‘-‘, ‘-.’, ‘--',’.’, ‘o’, ‘x’,’o’,’*’,’+’,...`
Symbols: `‘s’, ‘h’, ‘p’, ‘d’,`...

For instance,

```
>>plot(t,y1,t,y2,’r--',t,y2,’-.g’)
```

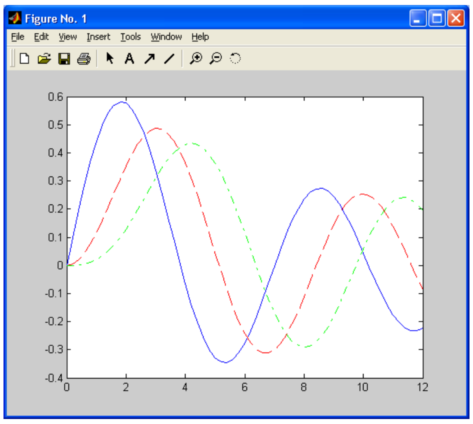


Fig. 7. Line styles

Step 7) Holding plots: To add new graphics over existing ones, use the function `hold` (also `hold on`, `hold off`).

```
>>plot(x,y1),hold
Current plot held
>>plot(x,y2)
>>plot(x,y3)
>>hold
Current plot released
```

Alternatively,

```
>>plot(t,y1),hold on
>>plot(t,y2)
>>plot(t,y3),hold off
```

Step 8) Subplots: Use **`subplot(a,b,c)`** to subdivide the figure window into several graphic areas, where **`a`** is the number of rows, **`b`** is the number of columns and **`c`** refers to the current plotting area (from **`1`** to **`a×b`**).

For instance, two plots:

```
>> subplot(212),plot(x,y2)
>> subplot(211),plot(n,'og')
```

Four plots:

```
>> x=linspace(0,12);y1=bessel(1,x);y2=bessel(2,x);y3=bessel(3,x);
>> subplot(221),plot(x,y1)
>> subplot(222),plot(x,y2)
>> subplot(223),plot(x,y3)
>> subplot(224),plot(x,x)
```

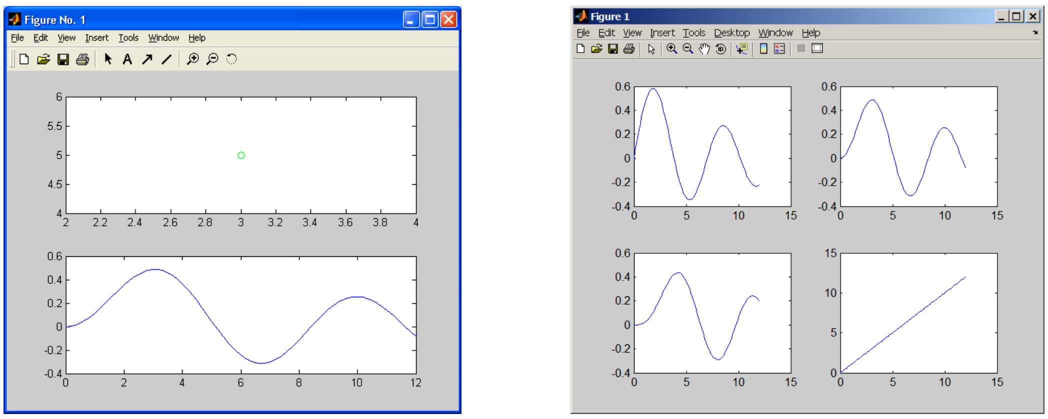


Figure No. 1
File Edit View Insert Tools Window Help
Figure 1
File Edit View Insert Tools Desktop Window Help

Fig. 8. Use of the **`subplot`** command

Step 9) Graphics input: To capture coordinate values *x*, *y* from a plot, use **`ginput`** (*graphics input*). The simplest usage is:

```
>>ginput
```

A cross appears over the current plot. Use the mouse to capture several points and when you have enough, return to the command window to see the captured points by pressing the <*Enter*> key.

Step 10) Save and open figures: To save a figure object, select the following options in the menu bar: **`File`** → **`Save`**. The figure will be saved in a file with the extension **`*.fig`**, for instance, **`figu.fig`**. To recover the figure, simply type:

```
>>openfig('figu')
```

3D plots: Three-dimensional line plots follow the same steps as two-dimensional line plots. The only difference is in the graphic commands. We use commands such as **`plot3`** or **`comet3`**.

```
>> plot3(y1,y2,y3)
```

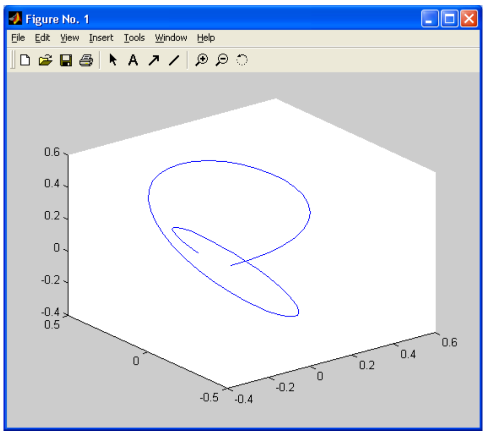


Fig. 9. 3D line object

Axes can be modified using the axis command:

```
axis([xmin xmax ymin ymax zmin zmax])
```

4.3 TEXT object

Again, consider the graphical representation of the three Bessel functions. To label the representation, we can use the following functions: **`xlabel`**, **`ylabel`**, **`zlabel`**, **`title`**, **`text`**, **`gtext`**, **`legend`**. Observe how we establish the format for sub- and super-indexes.

```
>> xlabel('x')
>> ylabel('y_1 , y_2 , y_3')
>> title('Funciones de Bessel de primera especie')
>> legend('1^e^r orden','2^o orden','3^e^r orden',-1)
>> text(6,0.5,'hoolaaa')
```

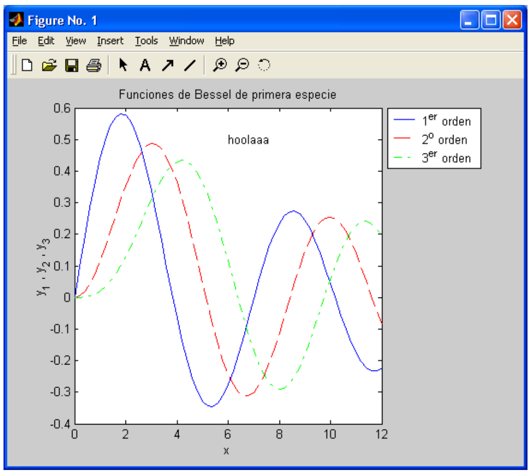


Figure No. 1
File Edit View Insert Tools Window Help
Funciones de Bessel de primera especie
hoolaaa
1er orden
2o orden
3er orden
y_1, y_2, y_3
x

Fig. 10. TEXT objects

Function **`gtext`** (*graphics text*) is similar to **`text`** but it inserts the text using the mouse instead of indicating the coordinates for the text. To write several text lines, use the symbol {}:

```
>>gtext({'y_1: 1^s^t order','y_2: 2^n^d order','y_3: 3^r^d order'});
```

It is possible to use symbols from the Greek alphabet. Just type symbol \ before the symbol name: **`\alpha`**, **`\beta`**,…

```
      >>title('y_1(\phi)=(\phi-sin(\phi))/2');
```

We suggest typing **`>>help TeX`**. This utility also allows for mathematical expressions such as rational functions, square roots, and so on.

Finally, you can insert other symbols, such as arrows:

```
     >> text(0.3,0.4,'\downarrow','FontSize',10)
```

Another useful function for use within a text object is **`num2str`** (number to string):

```
     >> r=2.5;
     >> text(0.4,0.3,['radio = ',num2str(r)])
```

4.4 PATCH object

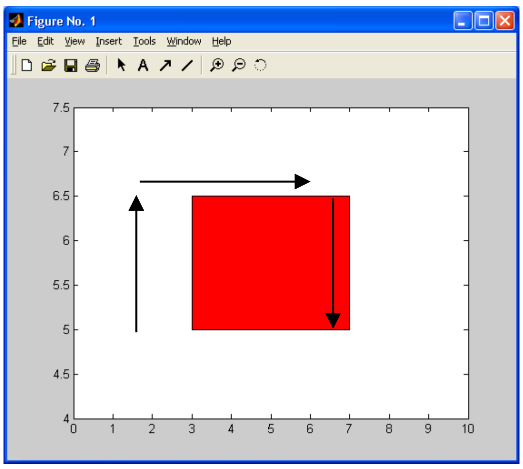
Patch objects are objects composed of one or more polygons that may or may not be connected. They are useful for presentations and animations since they can be used to draw complex pictures.

The three basic functions are **`fill`**, **`fill3`** and **`patch`**. The user must specify the polygon vertices and the fill-in colours.

The order in the vertices specification is very important. See the following example:

```
>> x=[3 3 7 7];
>> y=[5 6.5 6.5 5];
>> fill(x,y,'r')
>> axis([0 10 4 7.5])
```

```
>> x=[3 3 7 7];
>> y=[5 6.5 5 6.5];
>> fill(x,y,'r')
>> axis([0 10 4 7.5])
```



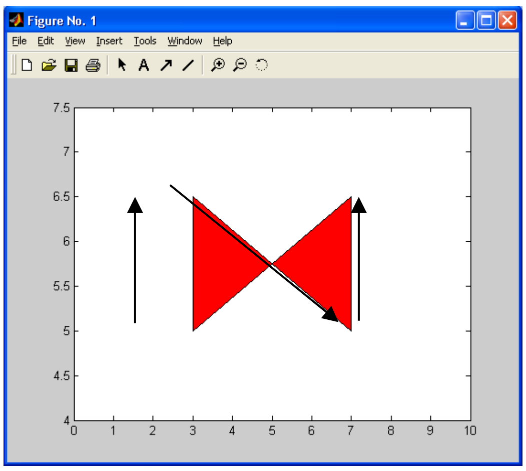


Fig. 11. Specification of vertices in a PATCH object

An alternative is **`fill(x1,y1,c1,x2,y2,c2,...).`**

It is possible to use pre-specified colours or to define our own colours using the triple [r g b], where the three components (red, green, blue) take values between 0 and 1.

<u>Example</u>:

Dark red= [0.5 0 0]
Copper = [1 .62 .4]
Grey = [0.5 0.5 0.5]

Pre-specified:

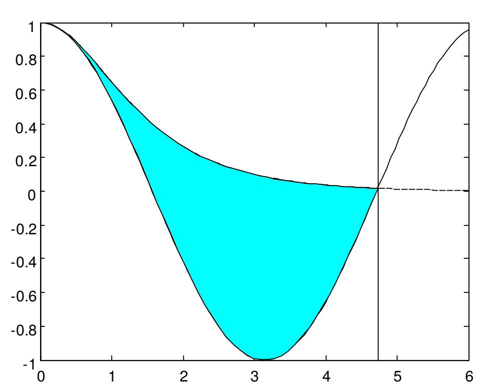
Red= [1 0 0] 'r'	Green = [0 1 0] 'g'
Blue = [0 0 1] 'b'	Cyan = [0 1 1] 'c'
Magenta = [1 0 1] 'm'	Yellow = [1 1 0] 'y'
Black = [0 0 0] 'k'	White = [1 1 1] 'w'

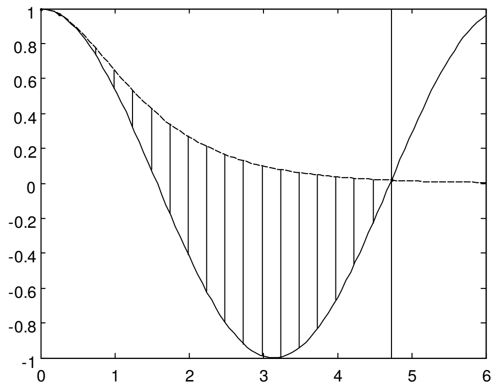
There also exist pre-specified colour maps (`colormap`): **`hsv`**, **`hot`**, **`cool`**, **`summer`**, **`gray`**, **`jet`**.

Example 7. PATCH objects

This example shows how to fill in an area [mag,05] :

```
>> x=linspace(0,6,100);
>> plot(x,cos(x),'k-',x,1./cosh(x),'k--',[4.73 4.73],[-1 1],'k')
>> hold on
>> xn=linspace(0,4.73,50);
>> fill([xn,fliplr(xn)],[1./cosh(xn),fliplr(cos(xn))],'c')
```





```
>> x=linspace(0,6,100);
>> plot(x,cos(x),'k-',x,1./cosh(x),'k--',[4.73 4.73],[-1 1],'k')
>> hold on
>> xx=linspace(0,4.73,20);
>> plot([xx;xx],[cos(xx);1./cosh(xx)],'k-')
```

4.5 SURFACE object

The procedure for plotting surfaces in three-dimensional axes is as follows,

First, define the values for axes x, y

```
>> x=-10:0.1:10;
>> y=x;
```

Then combine the values of `x` and `y` (vectors) to obtain the grid xx, yy (matrix) where we are going to represent the `z` values.

```
>> [xx,yy]=meshgrid(x,y);
```

`z` values are computed over the grid values `xx`, `yy`:

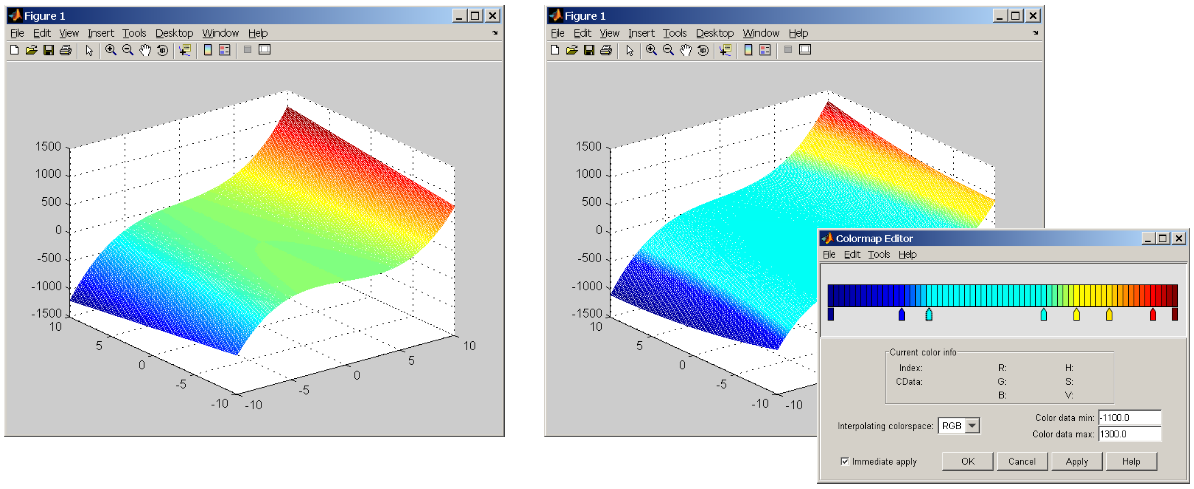
```
>> z=xx.^3+yy^2+2*xx*yy;
```

Finally, plot the surface (use **`mesh`**, **`surf`**, **`surface`**, **`waterfall`**). If you want to change the viewpoint, use the **`view`** function.

```
>> mesh(x,y,z)
```

Contours are represented with **`contour`** or **`meshc`** (which combines the surface and the contour plot). The contour diagrams are labelled with **`clabel.`**

To change the representation colour properties, use **`colormap`** (e.g., **`>>colormap gray),`** **`shading`**, **`hidden`**, **`brighten`**. You can also use the menu bar (*Edit* → *Colormap…*)



Interpolating colorspace: RGB
Color data min: -1100.0
Color data max: 1300.0
[✓ Immediate apply]

Fig. 12. SURFACE object.

Example 8. SURFACE objects

The following example can be found in the MATLAB demos:

```
z = peaks;
surf(z); hold on
shading interp;
[c ch] = contour3(z,20); set(ch, 'edgecolor', 'b')
[u v] = gradient(z);
h = streamslice(-u,-v); % downhill
set(h, 'color', 'k')
for i=1:length(h);
  zi = interp2(z,get(h(i), 'xdata'), get(h(i),'ydata'));
  set(h(i),'zdata', zi);
end
view(30,50); axis tight
```

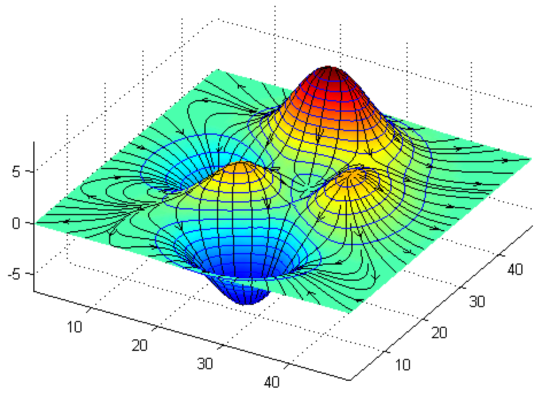


Fig. 13. SURFACE objects (peaks function).

To change the axes, use **`axis`**, **`zoom`**, **`grid`**, **`box`**, **`hold`**, **`axes`**, **`subplot`**. You can rotate the plot using **`rotate3d`**, **`viewmtx`** or **`view`** (in the menu bar: **`View`** → **`Camera Toolbar`**). We recommend exploring the options from the menu bar in the figure window.

```
>> z=peaks;surf(z)
```

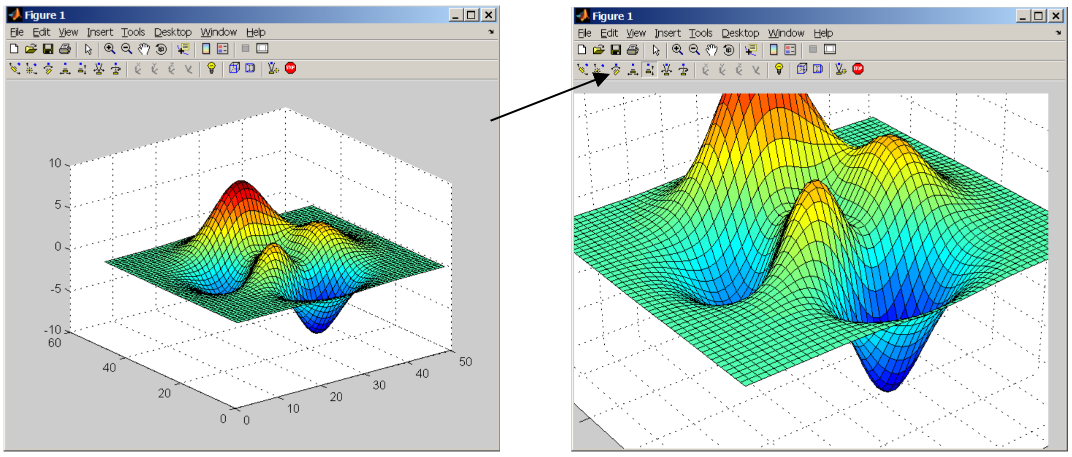


Fig. 14. Toolbar in the figure window.

For more information, type `>>help graph2d` y `>>help graph3d`.

Prespecified volumes: Use the functions `cylinder`, `sphere`, `ellipsoid`.

```
>> cylinder([2 1 1 0.5],20);
>> sphere(50),axis('square')
```

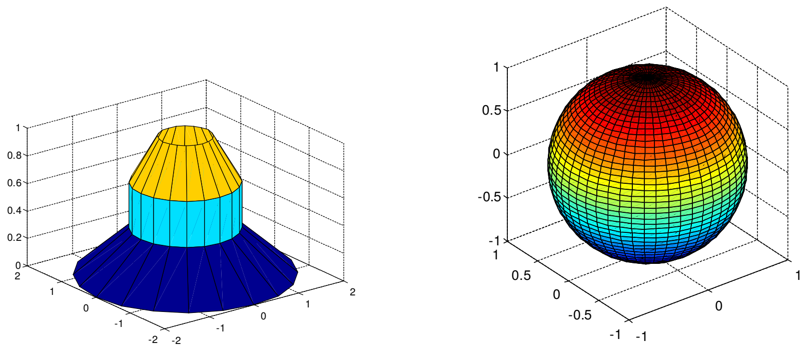


Fig. 15. Prespecified volumes.

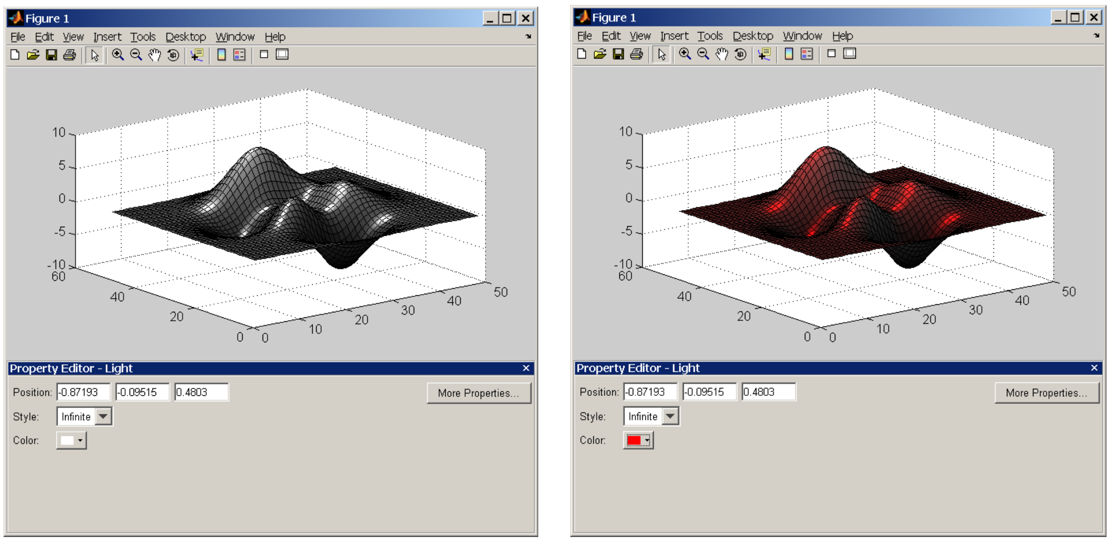
4.6 LIGHT object

This object is used to change the appearance of 3D representations. The most important functions are `lighting (flat, none, phong, gouraud)`, `material (metal, dull, shiny)`, `surfl`, `specular`, `diffuse`, `surfnorm`.

Example:

```
>> z=peaks;surf(z)
>> colormap('gray')
>> lighting phong
```

Select *Insert* → *Light* on the menu bar.



Position: -0.87193 -0.09515 0.4803
Style: Infinite
Color: ⬜

Position: -0.87193 -0.09515 0.4803
Style: Infinite
Color: 🟥

Fig. 16. LIGHT object.

4.7 IMAGE object

MATLAB writes/reads several graphical formats (TIFF, JPEG, BMP, PCX, XWD, HDF). The main functions are **`imread`**, **`imwrite`** and **`imfinfo`**.

```
>>X=imread('earth1','gif');
>>X=imread('earth1.gif');

> imfinfo('earth1.gif')

ans =

           Filename: 'earth1.gif'
        FileModDate: '17-May-2000 01:49:46'
           FileSize: 58178
             Format: 'GIF'
      FormatVersion: '87a'
              Width: 601
             Height: 353
           BitDepth: 7
          ColorType: 'indexed'
    FormatSignature: 'GIF87a'
    BackgroundColor: 0
        AspectRatio: 0
         ColorTable: [128x3 double]
         Interlaced: 'no'
```

There are two data types available to display the image: **`double`** (floating double precision, 64 bits) and **`uint8`** (unsigned integer, 8 bit). Functions are **`image`** and **`imagesc`**. It is possible to add a bar showing the present colours, **`colorbar`**.

An image consists of one data matrix X (each component is a pixel) and one matrix containing the colours for every pixel in X. There are four image types: indexed, of intensity, binary and truecolor.

Index image: A colormap matrix **map** is defined with 3 columns corresponding to R, G, and B, and as many rows as colours are present in the image. The elements of the image matrix **X** refer to a row number for the colour matrix.

```
>> load earth
>> image(X),colormap(map),colorbar('vert')
```

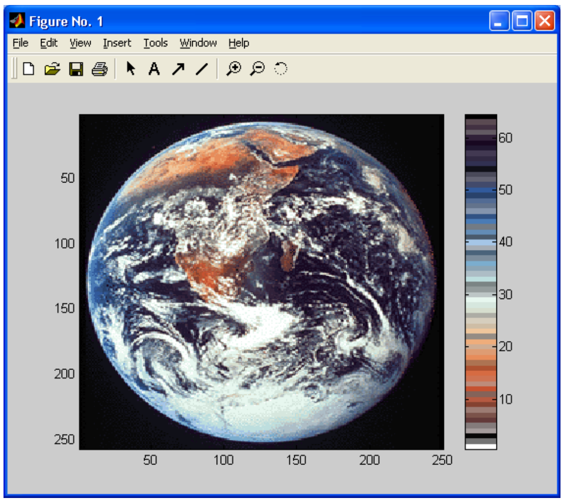


Fig. 17. Index image

Intensity image: Matrix **I** contains intensities (grey levels). These levels are from black to white (from 0 to 1, from 0 to 255 or from 0 to 65535)

```
>> Y=X/64;
>> imagesc(Y,[0 1]),colormap(gray),colorbar('vert')
```

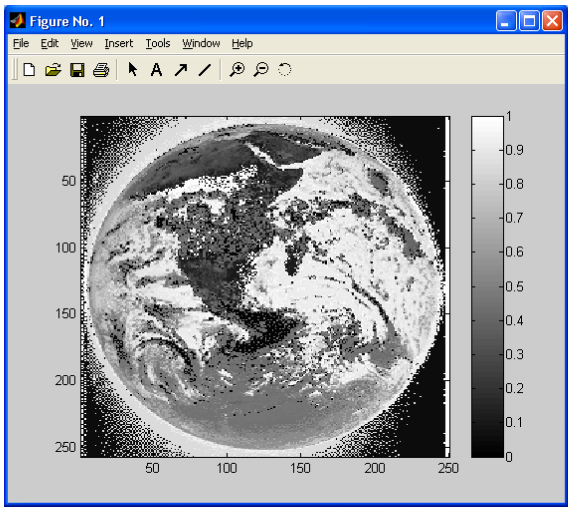


Fig. 18. Intensity image

Binary image: Components of matrix **X** are 1s and 0s.

Truecolor image: A 3D image that does not use a colormap. X has dimensions m×n×3. Each pixel in matrix X, X(m,n) is defined by three numbers: RGB(m,n,1), which corresponds to the red level; RGB(m,n,2), which corresponds to the green level; and RGB(m,n,3), which corresponds to the blue level.

```
>> rgb=imread('ngc6543a.jpg');
>> size(rgb)
ans =
   650   600     3

>> image(rgb)
```

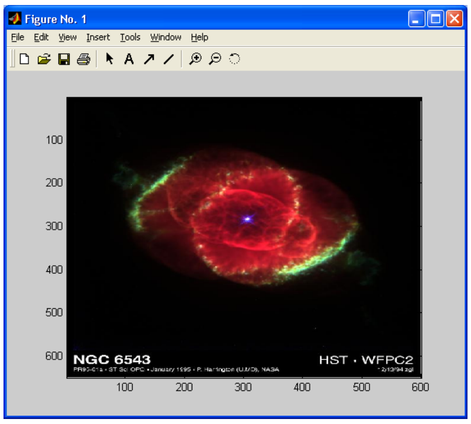


Fig. 19. Truecolor image

The default colormap is **`colormap('default')`** which corresponds to the **`hsv`** (*Hue Saturation Value*). Statement **`>>help graph3d`** gives information about other colour maps.

There are specific toolboxes that make a more intensive use of the images. For examples, see the demos of the Image Processing Toolbox, Mapping Toolbox Virtual Reality Toolbox.

5. Plots for specific applications

Depending on the applications, special graphics are required. For instance, in statistical analysis, data are presented by means of histograms, scattering diagrams, error bars, etc. Specific toolboxes include special representations facilities.

5.1 Plots for presentations

Pie diagramas: The command is **`pie`**. If the sum of values is less than 1, the pie is not complete.

```
>> x = [.19 .22 .41];
>> pie(x)
>> pie3(x)
```

If you want to take a piece:

```
>> x = [1 3 0.5 2.5 2];
>> pct=x/sum(x)
pct =
    0.1111    0.3333    0.0556    0.2778    0.2222

>> piece = [0 1 0 0 0];
>> pie(x,piece)
>> pie3(x,piece),colormap summer
```

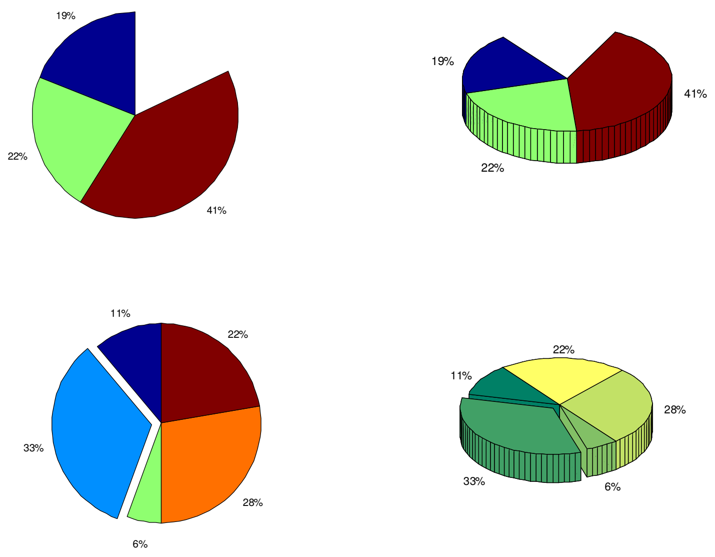


Fig. 20. Pie plots

Histograms: Functions are **`hist`**, **`histfit`**

```
yn = randn(10000,1);
hist(yn),
colormap autumn

Y = randn(10000,3);
hist(Y),
colormap summer

r = normrnd(10,1,100,1);
histfit(r)
```

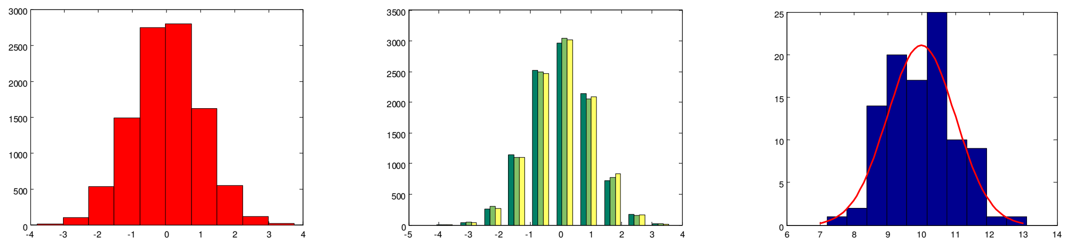


Fig. 21. Histograms

Bar plots: Functions are **`bar`**, **`barh`**

```
Y= round(rand(5,3)*10);
subplot(2,2,1),bar(Y,'group'),title('Grupo')
subplot(2,2,2),bar(Y,'stack'),title('Pila')
subplot(2,2,3),barh(Y,'stack'), title('Pila horizontal')
subplot(2,2,4),bar(Y,1.5),title('Grosor 1.5')
```

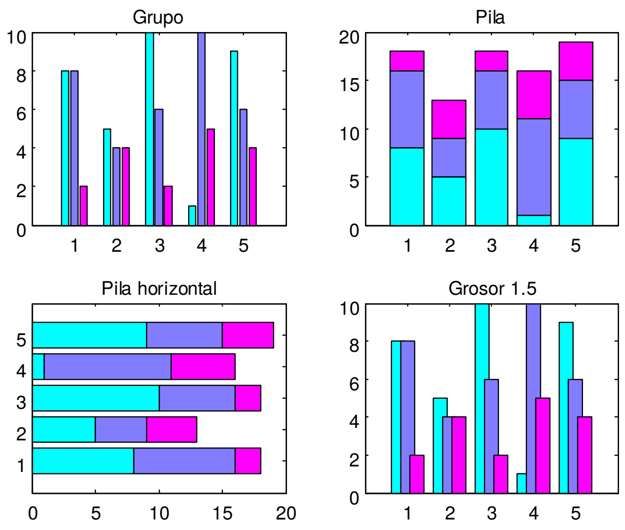


Fig. 22. Bar plots

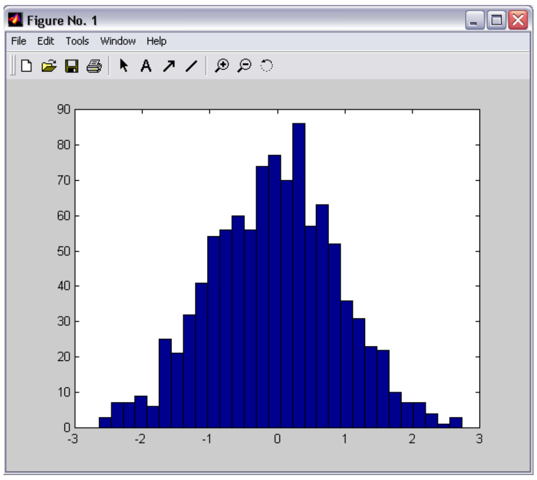
5.2 Probability and statistics

Example 9. Histograms

The histogram in the first figure has been generated as:

```
>>data=randn(1000,1);  %data is a vector of 1000 random
                       %elements
>>hist(data,30)  %30 bin histogram
```

Note that the data probability distribution does indeed correspond to a zero mean and unit variance Gaussian distribution (“n” in **`randn`** comes from “normal”). Which probability distribution corresponds to **`rand`**?



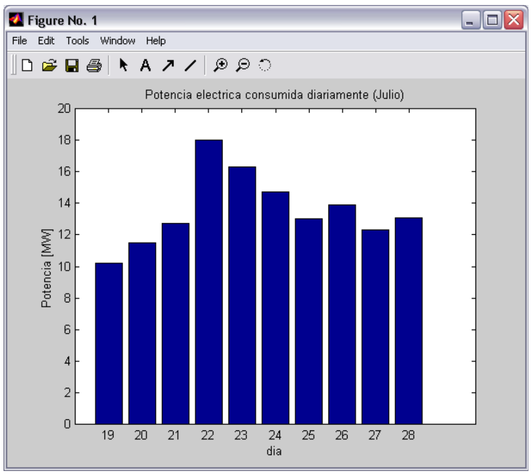


Figure No. 1
File Edit Tools Window Help
Potencia electrica consumida diariamente (Julio)
Potencia [MW]
dia

The second figure illustrates the energy (MW) spent by a town for 10 days. The commands used are:

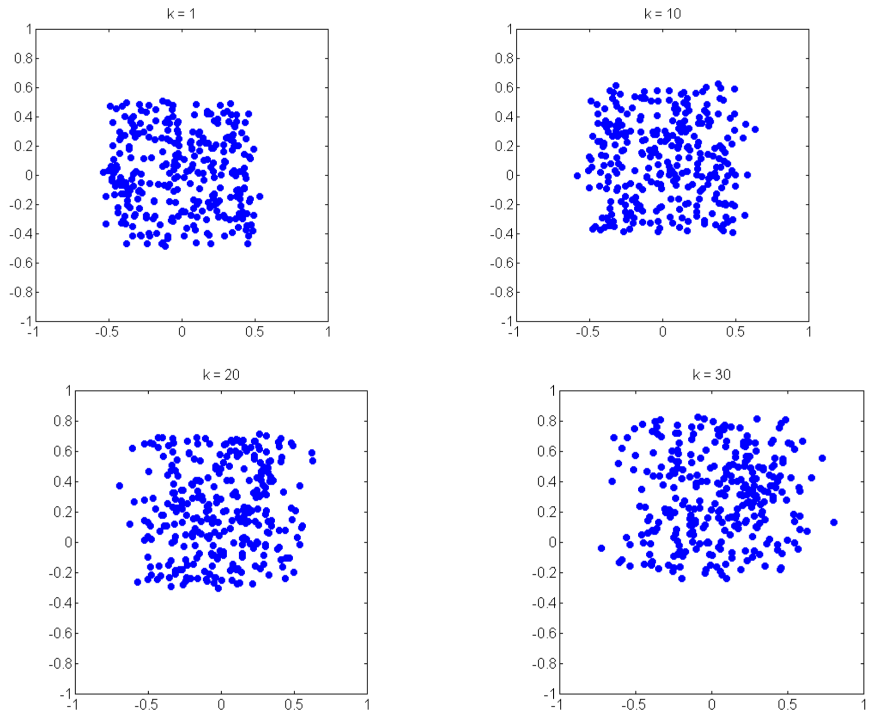
```
» days=19:28;
» power=[10.2 11.5 12.7 18 16.3 14.7 13.0 13.9 12.3 13.1];
» bar(days,power);
```

5.3 Movies

Movies: The main functions are **`moviein`** (beginning), **`getframe`** (to capture each frame) and **`movie`**. (Note: **`moviein`** is not necessary in the newer versions.)

Example 10. Brownian noise

```
n=300;s=0.02;
n_tr=50;
x=rand(n,1)-0.5;
y=rand(n,1)-0.5;
h=plot(x,y,'.')
set(h,'MarkerSize',18)
axis([-1 1 -1 1])
axis square
grid off

M=moviein(n_tr);

for k=1:n_tr
   x=x+s*randn(n,1);
   y=y+s*randn(n,1);
   set(h,'XData',x,'YData',y)
   M(:,k)=getframe;
end
movie(M,5)
```



Try other movie options, e.g., **`movie(M,-1,37)`**.

When you run the movie function, first it does a quick preview of the animation and then it presents the animation itself. You have several options, for example, movie (M, 0) executes the quick preview but not the animation, **`movie(M,-1)`** executes the quick pass and then plays the animation forward and then backwards.

By default, the video timing is that of the capture. If you want to change the timing, you must use the third argument. In **`movie(M,-2,50)`**, the timing is 50 frames per second. This means that a film with 50 frames will be shown in 1 second.

Video files: To generate a video file, you can use **`avifile`**. Frames are added to the video by using the **`addframe`** function. Finally, close the video with **`close`**.

The following example illustrates the generation of a video file (named *noise.avi*) based on the Brownian noise of the previous section.

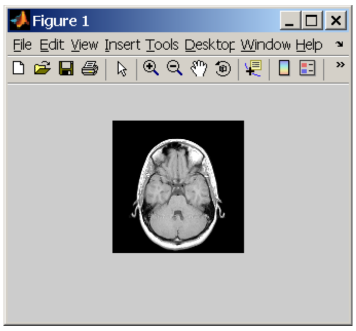
Example 11. Creating an avi file

```
mov = avifile('noise.avi')

n=300;
s=0.02;
x=rand(n,1)-0.5;
y=rand(n,1)-0.5;

h=plot(x,y,'.')
set(h,'MarkerSize',18)
axis([-1 1 -1 1]),axis square,grid off

n_frames=50;
for k=1:n_frames
   x=x+s*randn(n,1);
   y=y+s*randn(n,1);
   set(h,'XData',x,'YData',y)
   F=getframe(gca);
   mov=addframe(mov,F);
end
mov=close(mov);
```

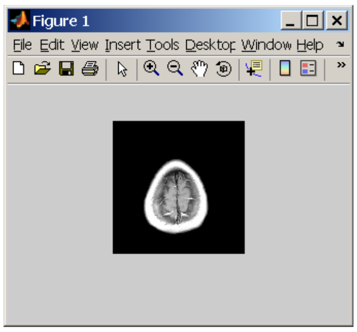
The generated video file can then be inserted to other programs, such as PowerPoint (Insert → Movies and sounds → Movie file). This is useful for presentations of projects or master's theses.

Animated images: To display animated images (multi-frame files, etc.), you can use the **`immovie`** function. To illustrate it, see the following example from MATLAB:

```
>> load mri
>> mov=immovie(D,map);
>> movie(mov)
```



…



To capture an image frame, you can do the following:

```
        [x,map]=imread('nombre_fichero.extensión',1:num:fotogramas);
```

or

```
        [x,map]=imread('nombre_fichero.extensión','frames','all');
```

You can also create an image by taking different frames separately and putting them together using the cat function:

```
A = cat (4, A1, A2)
```

Here, A1 and A2 are the two images that form the animation. The 4 is because this type of variable is an array of 4 dimensions m x n x 3 x 2, where m x n are the pixels of the image, 3 corresponds to a true-colour image (contains an array of three columns rgb, and each pixel refers to one of these 256 colours) and 2 is the number of frames.