UNIT 3

Toolboxes

1. Introduction

The MATLAB powerfulness is due to the fact that it is possible to run a large number of commands from a text file. These files are known as M-files since their names are of the form **`filename.m`**. M-files can be *functions* that accept input arguments and produce an output, or they can be *scripts* that execute a series of MATLAB statements.

Commercial toolboxes and other free access toolboxes contain M-files. MATLAB users can edit and modify the M-files of the toolboxes and they can create their own M-files as well.

To list the toolboxes installed in your computer, together with their versions and date, type **`>>ver`** in the command window.

To list the contents of a particular toolbox, you can simply enter **`>>help toolbox_name`**. For example, **`>>help matlab\general`** or **`>>help control`**.

2. Functions use other functions as input arguments

Function handle: Several functions have input arguments that are other functions. Such an input argument is called "function handle" and it is composed by the symbol **`@`** followed by the function name, **`@function_name`**. In the following sections, several application examples are presented.

2.1 Solving equations

Main functions are **`fsolve`** and **`fzero`**. Usually, functions that begin with "f" (**`fzero`**, **`fsolve`**, **`fminsearch`**, **`feval`**,…) are functions which input arguments are other functions. To list the default parameters of the optimization subroutines, type **`>>help foptions`**.

Example: We want to solve the following equation: $tg^{-1}\left(\frac{2}{\beta}\right)=e^{\beta}$.

First at all, edit an M-file **`equation.m`** containing the following function

```
function y=equation(beta)

y=atan(2/beta)-exp(beta);
```

Input argument is β and output argument is $y=tg^{-1}\left(\frac{2}{\beta}\right)-e^{\beta}$.

Using fzero: **fzero** finds the zero in nonlinear functions of one variable. First input argument is the equation to be solved. The second input argument is the initial guess about where the zero is (this guess can be a scalar value or an interval). If the initial value is too far from the solution **fzero** would fail and give no solution at all.

```
>> format long
>> beta=fzero(@equation,1)
beta =
   0.33865534225523

>> beta=fzero(@equation,10)
Exiting fzero: aborting search for an interval containing a sign change
    because NaN or Inf function value encountered during search.
(Function value at 829.2 is -Inf.)
Check function or try again with a different starting value.

beta =
   NaN
```

Using fsolve: **fsolve** solves nonlinear multivariable equation systems. Its syntax is the same than **fzero**.

```
>> beta=fsolve(@equation,1)
Optimization terminated: first-order optimality is less than
options.TolFun.
beta =
   0.33865534237264

>> beta=fsolve(@equation,10)
Optimization terminated: first-order optimality is less than
options.TolFun.
beta =
   0.33865534667084
```

Notice that the result is slightly different since the solving method is different too (**fsolve** belongs to the Optimization Toolbox whereas **fzero** is part of the MATLAB kernel)

```
>> which fsolve
C:\Archivos de programa\MATLAB704\toolbox\optim\fsolve.m
>> which fzero
C:\Archivos de programa\MATLAB704\toolbox\matlab\funfun\fzero.m
```

More options: The third input argument in **fzero** and **fsolve** is optional and includes information about the iterations and numerical methods. Type **>>help optimset** for more information.

```
>> options=optimset('Display','iter');
```

```
>> beta=fzero(@equation,[0.3 0.4],options)
 Func-count    x          f(x)             Procedure
    2             0.3     0.0720476        initial
    3        0.337826    0.00156645        interpolation
    4        0.338656 -5.18785e-007        interpolation
```

```
    5         0.338655  1.50686e-010         interpolation
    6         0.338655             0         interpolation

Zero found in the interval [0.3, 0.4]
beta =
   0.33865534225523
```

```
>> beta=fsolve(@equation,0.3,options)

                                    Norm of     First-order   Trust-region
Iteration  Func-count     f(x)      step        optimality    radius
0          2           0.00519085                  0.132           1
1          4           9.85129e-007  0.0391806     0.00188         1
2          6           3.33215e-014  0.000525202   3.45e-007       1
Optimization terminated: first-order optimality is less than
options.TolFun.

beta =
   0.33865543888303
```

Anonymous functions: If the equation to be solved is simple or simply we do not want to create an M-file every time we want to solve a function, we can use the so-called anonymous functions. An anonymous function is defined directly in the function handle. For instance:

```
>> func_anon=@(beta)(atan(2/beta)-exp(beta));

>> x=fsolve(func_anon,1)
Optimization terminated: first-order optimality is less than
options.TolFun.
x =
    0.3387
```

Syntax for anonymous functions is: **`@(arguments)(expression)`**

It is also possible to use parameters:

```
ct=2;func_anon=@(beta)(atan(ct/beta)-exp(beta));
```

2.2 Solving differential equations

The main functions for solving ordinary differential equations (ODE) are:

`ode23`: for 2nd or 3rd order ODEs
`ode45`: for 4th or 5th order ODEs

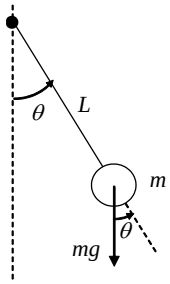
The syntax is:

```
>> [t,y]=ode23(@function_name,[Tini Tfin],init_cond);
```

The M-file **`function_name.m`** must contain the state equations corresponding to the differential equation to be solved. See the next example:

Example 1. Solving differential equations

Simple pendulum dynamics: The dynamics of the pendulum of the figure are described by the following equation: $\ddot{\theta} + \alpha\dot{\theta} + \gamma \sin\theta - \beta = 0$.



State equations: Firstly, we have to translate the ODE (of *n*-th order) to (*n*) state space equations (which are differential equations of first order).

Original equation:	$\ddot{\theta} + \alpha\dot{\theta} + \gamma \sin\theta - \beta = 0$
θ will be denoted x_1:	$\ddot{x}_1 + \alpha\dot{x}_1 + \gamma \sin x_1 - \beta = 0$
$\dot{x}_1$ will be denoted x_2 (note that $\dot{x}_1 = x_2 = \dot{\theta} = \omega$):	$\dot{x}_2 + \alpha x_2 + \gamma \sin x_1 - \beta = 0$

Thus, the description in the state space can be expressed as:

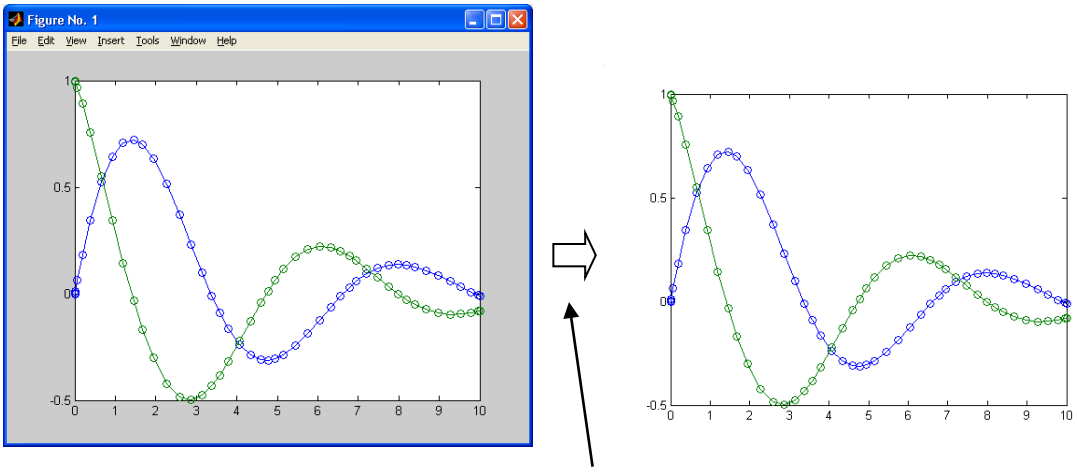
$$\left.\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\gamma \sin x_1 - \alpha x_2 + \beta \end{aligned}\right\}$$

M-file: Then, we have to create a function containing such a description:

```
function     xdot=pendulum(t,x)

alpha=0.5;gamma=1;beta=0;
xdot(1,:)=x(2);
xdot(2,:)=-alpha*x(2)-gamma*sin(x(1))+beta;
```

Calling ode23: Finally, we call **ode23** from the command window:

```
>> Tinitial=0;Tfinal=10;init_cond=[0;1];
>> ode23(@pendulum,[Tinitial Tfinal],init_cond);
>>
```



Edit → Copy Figure (in Matlab) and Paste (in the document processor)

2.3 Optimization

MATLAB has several functions for optimization.

Linear Programming: The function is `linprog`. Linear programming is used in problems where the objective function and constraints are linear,

$$\min_{\mathbf{x}} \mathbf{f}^T\mathbf{x}$$
$$\text{constraints}: \mathbf{A}\mathbf{x} \le \mathbf{b}$$
$$\mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq}$$
$$\mathbf{lb} \le \mathbf{x} \le \mathbf{ub}$$

Example 2. Linear programming

Assume that two products A and B are produced along two production lines [mag, 05]. Each line has 200 hours. The product A makes a profit of 4€ per unit and needs 1h of the first line and 1.25h of the second line. The product B gives a profit of 5€ per unit and needs 1h of the first line and 0.75h of the second. There is a potential maximum demand in the market of 150 units of B. We want to know how many units of A and B maximize the benefit of the manufacturer. Defining x1 and x2 as the units of products A and B respectively, the objective function and constraints are as:

Minimize $f(x_1, x_2) = -4x_1 - 5x_2$

Subject to: $g_1: \; x_1 + x_2 \le 200$

$g_2: \; 1.25x_1 + 0.75x_2 \le 200$

$g_3: \; x_2 \le 150$

$(x_1, x_2) \ge 0$

Thus, $\mathbf{f}^T = \begin{pmatrix} -4 & -5 \end{pmatrix}$

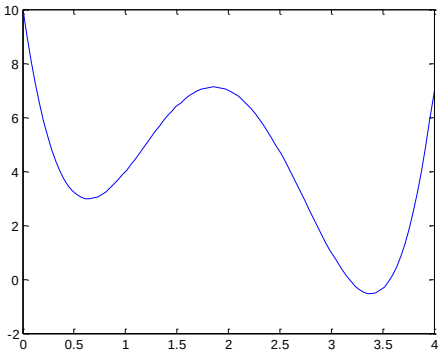
$$\begin{bmatrix} 1 & 1 \\ 1.25 & 0.75 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 200 \\ 200 \\ 150 \end{pmatrix}$$

```
f=[-4 -5];
A=[1 1;1.25 0.75;0 1];
b=[200 200 150];
lb=[0 0];
x=linprog(f,A,b,[],[],lb,[])
Optimization terminated.
x =
   50.0000
  150.0000
```

Nonlinear programming: These are optimization problems where the criterion and/or constraints are nonlinear. In the case of unconstrained optimization, $\min_{\mathbf{x}} f(\mathbf{x})$, the functions are `fminunc` and `fminsearch`. The first one uses derivative techniques

Example: Consider the function $f(x) = 1.625x^4 - 12.75x^3 + 31.375x^2 - 26.25x + 10$. To plot it, since it is a polynomial, you can use the **`polyval`** command,

```
>> coefs=[1.625  -12.75   31.375  -26.25   10];
>> x=linspace(0,4);
>> y=polyval(coefs,x);
>> plot(x,y)
```



The function that contains the curve of which we are going to find the local minima is:

```
function y=curve2(x)

coefs=[1.625  -12.75   31.375  -26.25   10];
y=polyval(coefs,x);
```

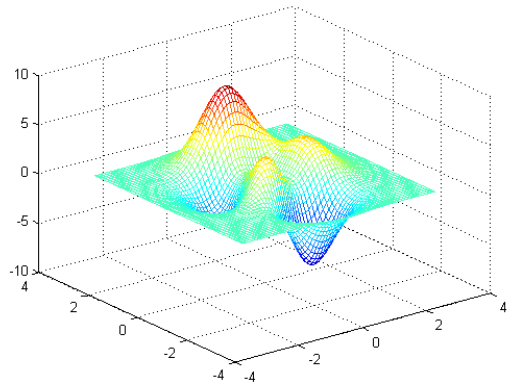
Using **`fminsearch`**:

```
>> fminsearch(@curve2,0.5)
ans =
    0.6425

>> fminsearch(@curve2,3)
ans =
    3.3853
```

Finding local maxima: It is enough to change the sign of the curve and use **`fminsearch`** again.

Optimization in 2-variable functions: The command is again **`fminsearch`**. For instance,

```
function z=surface(u)

x=u(1);
y=u(2);
z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```




```
>> fminsearch(@surface,[0 0])

ans =

    0.2964    0.3202

>>
```

2.4 Timers

Timers are implemented by combining **`struct`** type variables and functions that call other functions.

Example: Edit a file **`trial.m`** with the following commands:

```
function trial

%we create a "timer" object
t=timer('StartDelay',2,'Period',1,'TasksToExecute',2,'Executi
onMode','fixedRate');

t.StartFcn = {@suma,'Initial:  ',1,2};
%We begin by adding 1+2
t.StopFcn = {@suma, 'Final:  ',3,4};
%We terminate by adding 3+4
t.TimerFcn = { @suma, 'running...',6,7};
%Meanwhile we add 6+7

%Start the timer
start(t)


function suma(obj,event,txt,arg_in1,arg_in2)
result=arg_in1+arg_in2

event_type = event.Type;
event_time = datestr(event.Data.time);

msg = [txt,' ',event_type,'  ',event_time];
disp(msg)
```

The result is:

```
result =
     3

Initial:   StartFcn  15-Mar-2013 18:03:24
result =
    13

running... TimerFcn  15-Mar-2013 18:03:26
result =
    13

running... TimerFcn  15-Mar-2013 18:03:27
result =
     7
Final:   StopFcn  15-Mar-2013 18:03:28
>>
```

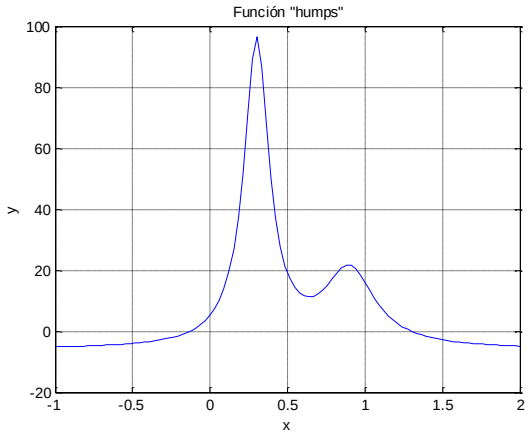
2.5 Integration and derivation

In this section we use the function “humps” which is already defined in MATLAB. The file **`humps.m`** contains the following function:

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6$$

To plot the function, type the following sentences:

```
>> x=linspace(-1,2);
>> y=humps(x);
>> plot(x,y),grid,xlabel('x'),ylabel('y'),title('Función "humps"')
```

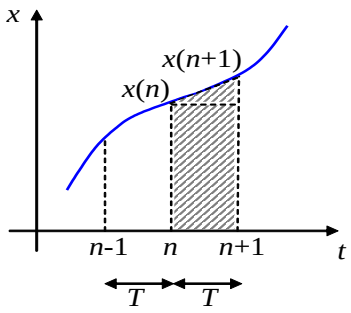


The integral value *I* for the function *humps* between -1 and 2 is *I*=26.34496047137833.

Integrate a function is equivalent to compute the area below such a function. MATLAB presents commands that numerically approximate the integral of functions.

Main functions are: **`trapz`**, **`cumtrapz`**, **`quad`**, **`quadl`**, **`dblquad`**, and **`triplequad`**.

Approximate integral by means trapezoid areas of equal base (trapz): It is possible to integrate the whole area $I = \int_{x_1}^{x_2} f(x)dx$ by means the sum of the areas of equal base trapezoids (Simpson rule). The function is **`trapz`**. The result will be more accurate if the trapezoids base is small:



The shadowed area is $\frac{x(n)+x(n+1)}{2}T$.

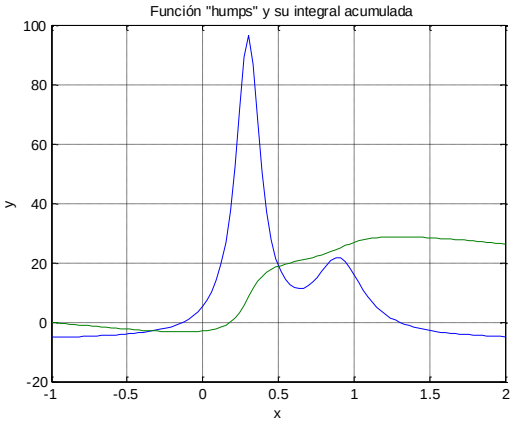
```
>> x=linspace(-1,2,30);y=humps(x);area1=trapz(x,y),paso=x(2)-x(1),
area1 =
   26.2102
paso =
    0.1034
```

```
>> x=linspace(-1,2);y=humps(x);area1=trapz(x,y),paso=x(2)-x(1),
area1 =
   26.3447
```

```
paso =
    0.0303
```

Cummulative integral (cumtrapz): If you want to evaluate the integral as a function of x, $I(x) = \int_{x_1}^{x} f(x)dx$, you can use **`cumtrapz`**.

```
>> x=linspace(-1,2);y=humps(x);I=cumtrapz(x,y);
>> plot(x,y,x,I),grid,xlabel('x'),ylabel('y'),
>> title('Función "humps" y su integral acumulada')
```



Quadrature integral (quad and quadl): Depending on the curve shape, these functions modify the base of each of the trapezoid areas in order to improve the results obtained with **`trapz`**. Function **`quadl`** (*l* comes from adaptive Lobato) is slightly better than quad. Both functions need a function with the curve description.

```
>> format long
>> quad(@humps,-1,2)
ans =
  26.34496050120123

>> quadl(@humps,-1,2)
ans =
  26.34496047137897
```

Integration and derivation of polynomials: Although a polynomial derivative is easy to compute, there exists the **`polyder`** function.

Example: Given $p(x) = x^3 + 2x^2 + 3x + 4$, its derivative is $dp(x)/dx = 3x^2 + 4x + 3$

```
>> p=[1 2 3 4];
>> dp=polyder(poli)
dp =
     3     4     3
```

There also exist the function **`polyint`** to integrate polynomials, $\int (3x^2 + 4x + 3)dx = 3\frac{x^3}{3} + 4\frac{x^2}{2} + 3x + ct$. The second input argument of the function corresponds to the integration constant.

```
>> polyint(dp)
ans =
     1     2     3     0
>> polyint(dp,4)
ans =
     1     2     3     4
```

Derivation: The function is **`diff`** and it computes the progressive difference between samples,

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

It is recommended to use this function only in the cases the curve is smooth. If it is not de case (for instance in the case of discrete data), it is recommended to smooth the curve before using **`diff`**.

Symbolic integration and derivation: It is possible to use the function of the Symbolic Toolbox. For more details, type **`>>help symbolic`**.

Before using a symbolic operation, you must declare the selected variables as symbolic objects:

```
>> syms x n
```

Symbolic integration: The function is **`int`**. For instance, the integration of x^n is $\int x^n dx = \frac{x^{n+1}}{n+1}$. The commands are:

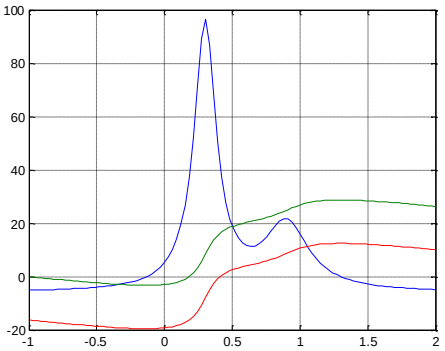
```
>> syms x n
>> f=x^n;
>> int(x^n)
ans =
x^(n+1)/(n+1)
```

Another example: To obtain the "humps" integral as a function of x:

```
>> syms x y
>> y = 1 / ((x-.3)^2 + .01) + 1 / ((x-.9)^2 + .04) - 6;
>> int(y,x)
ans =
10*atan(10*x-3)+5*atan(5*x-9/2)-6*x
```

and to plot it:

```
>> clear x z,
>> x=linspace(-1,2);z=10*atan(10*x-3)+5*atan(5*x-9/2)-6*x;
>> plot(x,y,x,I,x,z),grid,
```



Integral in an interval: Consider again the humps example:

```
>> int(y,-1,2)
ans =
10*atan(17)-18+5*atan(11/2)+10*atan(13)+5*atan(19/2)

>> 10*atan(17)-18+5*atan(11/2)+10*atan(13)+5*atan(19/2)
ans =
  26.34496047137833
```

Symbolic derivation: The function is **`diff`** but applied to symbolic objects:

```
>> syms x
>> diff(log(x))
ans =
1/x
```

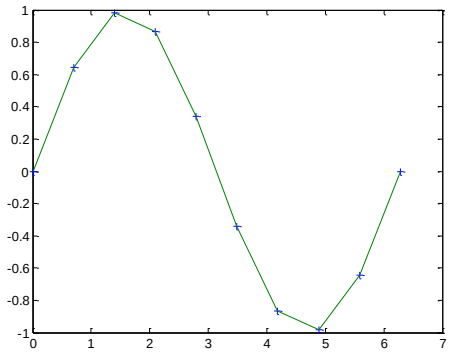
2.6 Interpolation and regression. Polynomial fitting of curves

To interpolate is to estimate unknown intermediate values in a set of known values. It is an important operation in data analysis and curve fitting.

Interpolation in one dimension (interp1): It uses polynomial techniques. It fits a polynomial between each couple of points and it estimates the value in the desired interpolation point.

Example: Assume that we have few points of a sinusoid:

```
>> x=linspace(0,2*pi,10);y=sin(x);figure(1),plot(x,y,'+',x,y)
```



We want to know the curve value at point 0.5. To interpolate such a value we can use the **`interp1`** function. There are several interpolation techniques available:

```
>> s=interp1(x,y,0.5,'linear')
s =
    0.4604

>> s=interp1(x,y,0.5,'cubic')
s =
    0.4976

>> s=interp1(x,y,0.5,'spline')
s =
    0.4820
```

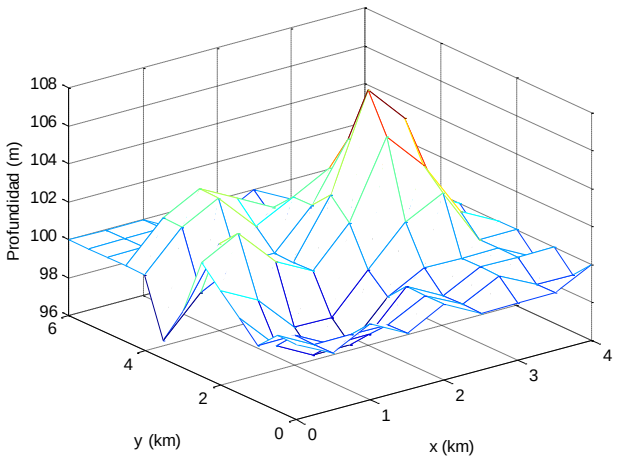
The exact value is sin(0.5)=0.4794. Therefore, the 'spline' option is the one that gives the better result in this case.

Interpolation in two dimensions (interp2): It is like **`interp1`** but now we interpolate in a plane.

Example: The script **`sea_bed.m`** contains data from the sea bed depth measurements.

```
%sea_bed.m

%sea bed depth measurements

x=0:0.5:4; %km
y=0:0.5:6; %km

z=[100  99 100  99 100  99  99  99 100;
   100  99  99  99 100  99 100  99  99;
    99  99  98  98 100  99 100 100 100;
   100  98  97  97  99 100 100 100  99;
   101 100  98  98 100 102 103 100 100;
   102 103 101 100 102 106 104 101 100;
    99 102 100 100 103 108 106 101  99;
    97  99 100 100 102 105 103 101 100;
   100 102 103 101 102 103 102 100  99;
   100 102 103 102 101 101 100  99  99;
   100 100 101 101 100 100 100  99  99;
   100 100 100 100 100  99  99  99  99;
   100 100 100  99  99 100  99 100  99];
```

```
figure(1),mesh(x,y,z),
xlabel('x (km)'),ylabel('y (km)'),zlabel('Profundidad (m)')
```



If we want to interpolate the depth in the point (2.2,3.3), we can do:

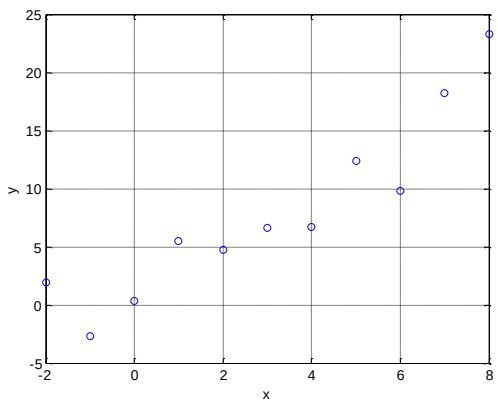
```
>> interp2(x,y,z,2.2,3.3,'cubic')
ans =
  104.1861
```

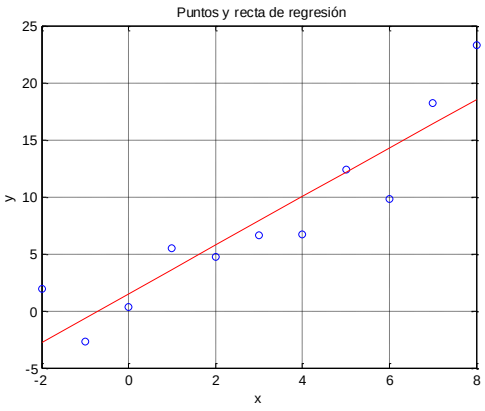
Regression line: Consider several points (*x*, *y*). We want to find the curve that best fits all of them.

x	-2	-1	0	1	2	3	4	5	6	7	8
y	1.94	-2.71	0.34	5.50	4.77	6.61	6.70	12.38	9.79	18.24	23.27

To plot them, simply create a script with the following commands:

```
x=[-2,-1,0,1,2,3,4,5,6,7,8];
y=[1.94,-
2.71,0.34,5.50,4.77,6.61,6.70,12.38,9.79,18.24,23.27];
figure(1),plot(x,y,'o'),grid,xlabel('x'),ylabel('y')
```





To find the line $y_r = ax + b$ is equivalent to specify its slope *a* and its ordinate in origin *b*. Since there is no straight line that contains all the points, we have to look for an optimal straight line in the sense that it minimizes the sum of the squared errors in the known points.

```
>> coefs=polyfit(x,y,1)
coefs =
    2.1317    1.4985
```

To plot the line $y_r = 2.1317\,x + 1.4985$, use the **`polyval`** function,

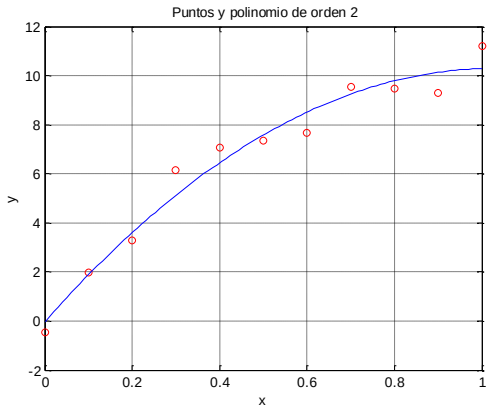
```
yr=polyval(coefs,x);
hold on,plot(x,yr,'r'),title('Puntos y recta de regresión'),
hold off
```

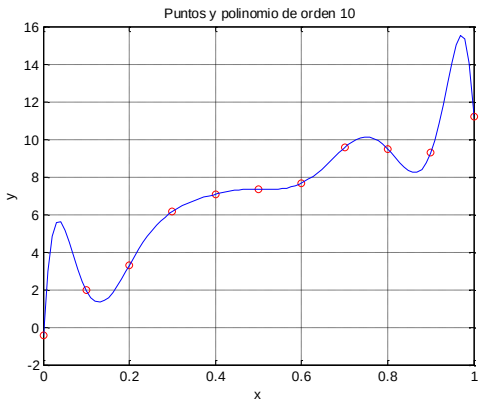
It is also possible to type **`yr=coefs(1)*x+coefs(2)`**.

Polynomial fitting: For the case of polynomials of order 2 or more, the procedure is the same:

```
%quadratic regression
x=0:0.1:1;
y=[-.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.3 11.2];
coefs1=polyfit(x,y,2);
x1=linspace(0,1);y1=polyval(coefs1,x1);
figure(2),plot(x,y,'or',x1,y1),grid,xlabel('x'),ylabel('y'),
title('Puntos y polinomio de orden 2')
```

```
%curve that fits all the points exactly
n=length(x);
coefs2=polyfit(x,y,n-1);
x2=linspace(0,1);y2=polyval(coefs2,x1);
figure(3),plot(x,y,'or',x2,y2),grid,xlabel('x'),ylabel('y'),
title(['Puntos y polinomio de orden ',num2str(n-1)])
```






```
>> coefs1
coefs1 =
   -9.8108   20.1293   -0.0317
```

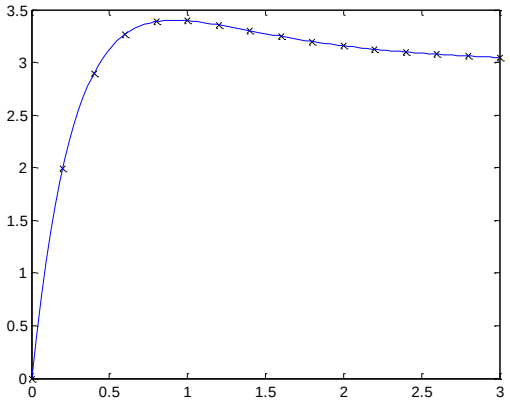
```
>> coefs2
```

```
coefs2 =
  1.0e+006 *
  Columns 1 through 7
   -0.4644    2.2965   -4.8773    5.8233   -4.2948    2.0211
-0.6032
  Columns 8 through 11
    0.1090   -0.0106    0.0004   -0.0000
```

Pseudoinverse. Fitting of curves linear in the parameters (pinv function): The pseudoinverse matrix can be used to estimate expressions that are linear in the parameters such as $y(x) = a_0 + a_1 e^{-\frac{x}{0.3}} + a_2 e^{-\frac{x}{0.8}}$; where the unknown values are $(a_0, a_1, a_2) = (3,-5,2)$.

```
x=0:0.2:3;
y=3-5*exp(-x/0.3)+2*exp(-x/0.8);
figure(4),plot(x,y,'xk')
H=[ones(size(x')) -exp(-x'/0.3)  exp(-x'/0.8)];
coefs=pinv(H)*y'
pause,
xrep=linspace(0,3);
Hrep=[ones(size(xrep')) -exp(-xrep'/0.3)  exp(-xrep'/0.8)];
hold on,plot(xrep,Hrep*coefs)
```

```
coefs =
    3.0000
    5.0000
    2.0000
```



3. Signal Processing

3.1 Audio processing

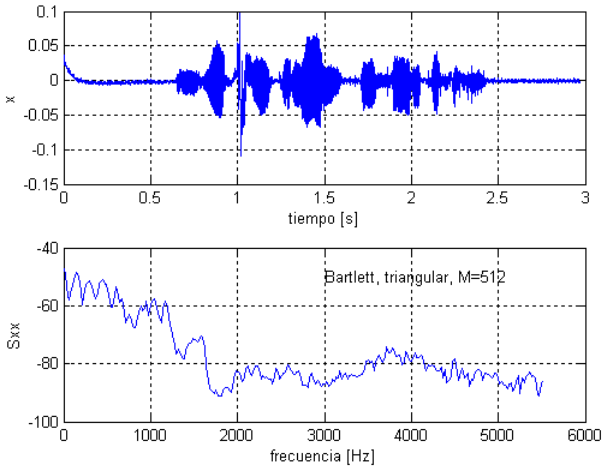
MATLAB can import audio files (function **`wavread`**) and can access the audio hardware to listen to them (function **`sound`**).

Example 3. Finding the pitch frequency of a human voice

In a **`*.wav`** file we have recorded the sentence "el golpe de timón fue sobrecogedor" with the following options: sampling frequency 11025Hz, mono and 16bits.

Next we plot the signal, we estimate the spectral density by means the Barlett method (with triangular window of M=512 samples) and we search for the fundamental pitch frequency (it happens to be 140Hz, a male voice).

```
[y,fs,bits]=wavread('frase');
fs,bits

%cut the sequence
pot=fix(log2(length(y)));
N=2^pot;
x=y(1:N);
sound(x,fs,bits)

%spectral estimation
M=512;K=N/M;w=triang(M);
Sxx=zeros(1,M);
for i=1:K
    xi=x( (i-1)*M+1 : i*M )'.*w';
    Xi=fft(xi);
    Sxxi=(abs(Xi).^2)/M;
    Sxx=Sxx+Sxxi;
end
Sxx=Sxx/K;

%time plot
t=0:1/fs:(N-1)/fs;
subplot(211),plot(t,x),grid,xlabel('tiempo [s]'),ylabel('x')

%frequency plot
f=linspace(0,fs/2,M/2);
subplot(212),plot(f,10*log10(Sxx(1:M/2))),grid,ylabel('Sxx'),
xlabel('frecuencia [Hz]'),text(3000,-50,'Bartlett, triangular, M=512')
```



3.2 Fast Fourier Transform (FFT)

Consider a sequence of samples $x[n]$, $n = 0,\ldots, N-1$, in the time domain. We want to estimate the magnitude of its spectral density.

Fast Fourier Transform: Roughly speaking, to pass from the time domain to the frequency domain what we do is to apply the Fourier transform to the time samples. If N is large this is unfeasible. Fortunately there exists one fast algorithm called FFT (*Fast Fourier Transform*) that can be used when the number of samples is a power of 2. MATLAB implements the FFT.

Before applying the FFT we can do two things:

- Truncate the original sequence (we take only the first 2^k samples)
- Complete the original sequence with a number of "0" such that the final length N a power of 2 (*zero padding*).

Here we truncate the original sequence:

```
[y,fs,bits]=wavread('frase');
fs,bits

%sequence truncation
pot=fix(log2(length(y)));
N=2^pot;
x=y(1:N);
sound(x,fs,bits)
```

Methods for spectral density estimation: Now we have several options to obtain spectra. We can take pieces of the original sequence, apply a window to them or not, compute the FFT of each piece, find the average among all pieces, use or not overlapping between windows, etc. All these strategies aim to improve the estimation (computing load, improve the discrimination of the frequency peaks, improve the quality factor, smooth the estimate,…). Main methods are the following:

Periodogram method: Consists of directly applying the expression:

$$\hat{S}_x(f) = \frac{1}{N}\left|\sum_{n=0}^{N-1} x[n]e^{-j2\pi fn}\right|^2,$$

where $\hat{S}_x(f)$ is the so-called periodogram.

Bartlett method or periodogram average: Consists of Split the sequence $x[n]$ of N samples in K segments of M samples each segment and average the K resulting periodograms:

- Segment k: $x_k[m] = x[m + kM]$, $k = 0,..., K-1$, $m = 0,..., M-1$
- Periodogram of segment k: $\hat{S}_x^k(f) = \frac{1}{M}\left|\sum_{m=0}^{M-1} x_k[m]e^{-j2\pi fm}\right|^2$, $k = 0,..., K-1$
- Average of the periodograms of the K segments: $\hat{S}_x(f) = \frac{1}{K}\sum_{k=0}^{K-1}\hat{S}_x^k(f)$

Welch method: This method is also known as average of modified periodograms. It is an extension of the Barlett method with two differences: we introduce overlapping D between segments and these are windowed with $w[n]$. the procedure is the following:

Split the N samples sequence $x[n]$ into K segments of M samples each one and introduce an overlap of D samples between segments:

- Segment k: $x_k[m] = x[m + kM - D]$, $k = 0,..., K-1$, $m = 0,..., M-1$ (if $D = 0.5M$ the overlap is 50%, if $D = 0.1M$ it is 10%)
- The K segments are windowed with $w[n]$ and the periodograms are computed. The periodograms are normalized in order to take into account the window effect:
- Periodogram of segment k: $\hat{S}_x^k(f) = \frac{1}{MU}\left|\sum_{m=0}^{M-1} x_k[m]w[m]e^{-j2\pi fm}\right|^2$, $k = 0,..., K-1$, where $U = \frac{1}{M}\sum_{m=0}^{M-1} w^2[m]$.
- Finally the K resulting periodograms are averaged: Average of the periodograms of the K segments: $\hat{S}_x(f) = \frac{1}{K}\sum_{k=0}^{K-1}\hat{S}_x^k(f)$

Blackman-Tukey method or peridogram smoothing: We take the N samples of $x[n]$ and compute their correlation $R_{xx}[k]$. The samples of $R_{xx}[k]$ farthest from the origin are not considered since they have been obtained from a few samples of $x[n]$. Therefore, the correlation is windowed with a M length window centered at the origin. To compute the spectral density we apply the FFT to the windowed correlation $R_{xx,v}[m]$.

4. Probability and Statistics

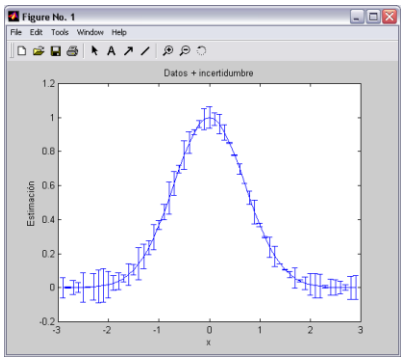
4.1 Plots for statistics studies

Next examples show several of the Matlab utilities for statistics applications.

Example 4. Error intervals

Consider a system whose output y is the decreasing exponential of the squared input u. Output measurements for different input values u are uncertain (uncertainty here has been generated in a random manner). Such uncertainty can be represented using **errorbar**:

```
» u=-2.9:0.1:2.9;
» e=0.1*rand(size(u));
» y=exp(-u.*u);
» errorbar(u,y,e)
```



Example 5. Box and scatter diagrams

The file **carsmall.mat** contains data about 100 cars: *Acceleration*, *Cylinders*, *Displacement*, *Horsepower*, *MPG* (consumption: miles-per-gallon), *Model*, *Model_Year*, *Origin*, *Weight.*

```
load carsmall
```

Scatter diagrams:

```
figure,scatter(Weight,MPG),xlabel('Weight'),ylabel('MPG')

figure,gscatter(Weight,MPG,Model_Year,'bgr','xos')
```

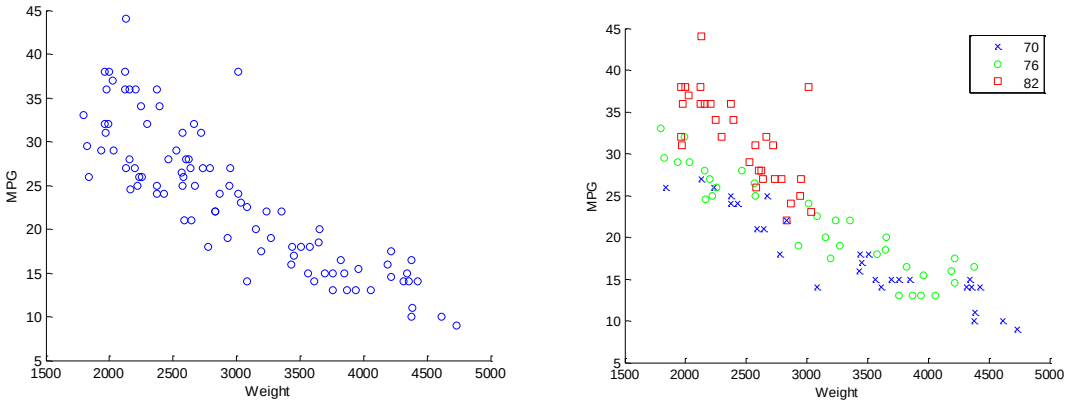


Fig. 1. Scatter plots

Box plots:

```
boxplot(MPG, Origin)
```

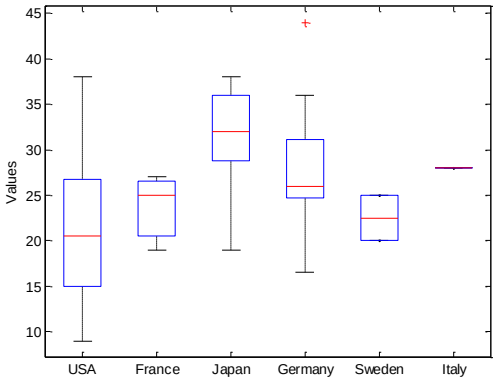


Fig. 2. Box plots

There is one outlier: a car with the characteristic MPG>40. You can use **`find`** to identify it:

```
>> find(MPG>40)
ans =
    97
```

It is a German car:

```
>> Origin(97,:)
ans =
Germany
```

We can also identify the model and year:

```
>> Model(97,:)
ans =
vw pickup

>> Model_Year(97,:)
ans =
    82
```

4.2 Probability distributions and density functions

Example 6. Testing a distribution. Probability plots

Distribution diagrams:

Normal probability plot: It is used to determine if a given sample is Gaussian distributed. The solid line connects 25 and 75 percentiles.

```
x=normrnd(10,3,100,1);
figure,normplot(x)

x=exprnd(10,100,1);
figure,normplot(x)
```

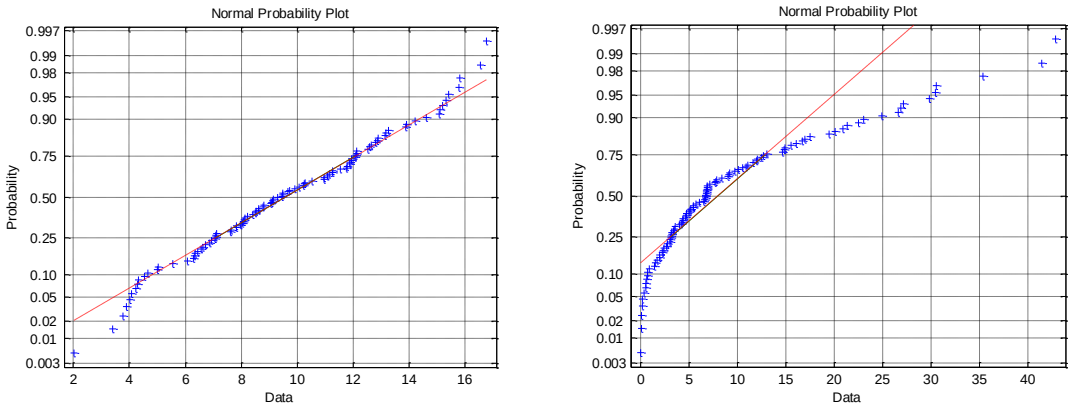


Fig. 3. Normal probability plots

Clearly, the second figure does not correspond to a normal distribution. Another way to see that it is not normal is by means of the Kolmogorov-Smirnov test:

```
>> h=kstest(x)
h =
     1
```

The interpretation is as follows: if the result is h = 1, we can reject the null hypothesis. The null hypothesis is that the sample x is standard normal distribution (mean 0 and variance 1). The test result says that you can reject this hypothesis. Therefore, the sample is not distributed as N (0,1).

We may also ask whether the distribution is normal but with a different mean and deviation, N(m,σ). We already know that this is not the case for the PP plot, but we will make sure:

```
>> [m,s]=normfit(x); %buscar media y desv que ajusten la muestra
>> [h,p]=kstest(x,[x normcdf(x,m,s)])  %y aplicar el test
h =
     1
p =
    0.0032
```

Since h = 1 again, we can reject the null hypothesis that the sample is distributed as N(m,σ). We are left with the alternative hypothesis, which says that the sample is not distributed as N (m, σ).

The **`kstest`** function rejects the null hypothesis (h = 1) by default if the significance level is 5%, that is, if the value of p is less than 0.05, as is true in our case (if the p value had been higher than 0.05, h would have given 0).

For other distributions beyond Gaussian, use *probplot*.

```
x=wblrnd(3,3,100,1);
probplot('weibull',x)
```

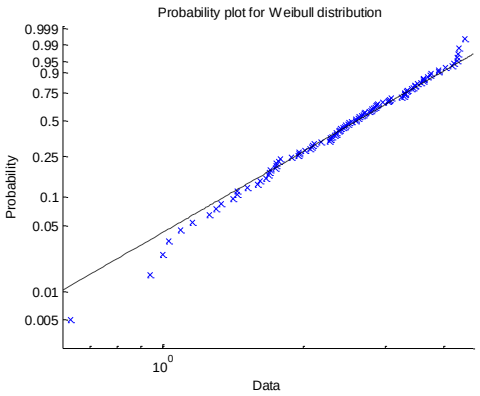


Fig. 4. Probability plot (other distributions)

Q-Q (quantile-quantile) plot: shows whether two samples are from the same probability distribution family.

```
x=poissrnd(10,50,1);
y=poissrnd(5,100,1);
figure,qqplot(x,y)

x=normrnd(5,1,100,1);
y=wblrnd(2,0.5,100,1);
figure,qqplot(x,y)
```

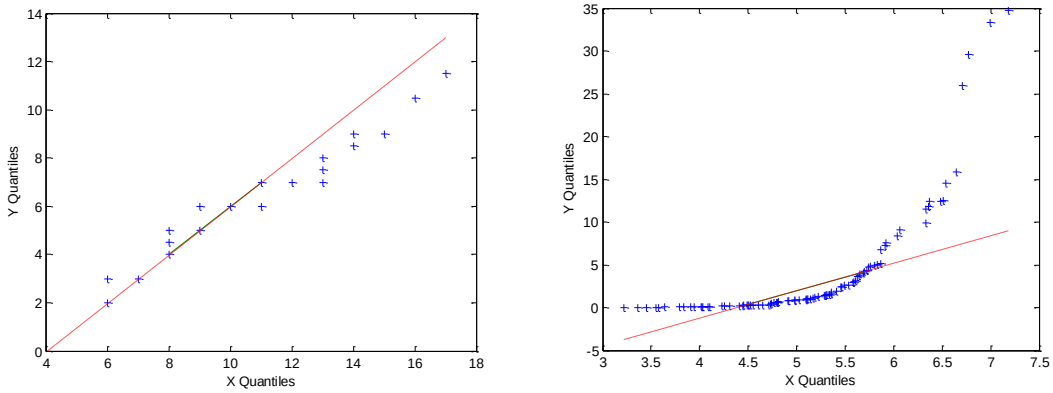
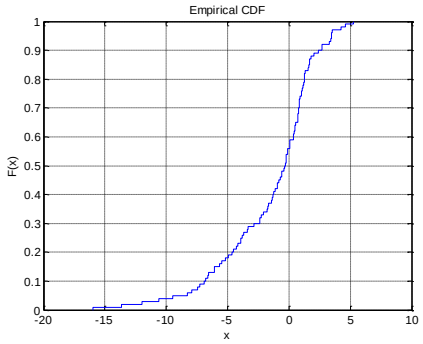


Fig. 5. Q-Q plots

Cumulative distribution diagram: Function is *cdfplot*.

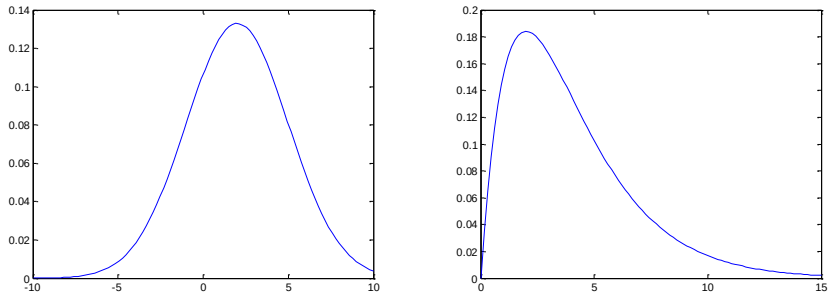
```
y=evrnd(0,3,100,1);
figure,cdfplot(y)
```



Example 7. Probability density function generation

Generating pdfs: Functions that generate probability density functions (pdfs) end with "…*pdf*" and start with the probability family name. Hence *normpdf* generates the normal distribution pdf and *chi2pdf* generates the pdf corresponding to the chi-square distribution.

```
x=linspace(-10,10);y=normpdf(x,2,3);figure,plot(x,y)
x=linspace(0,15);y=chi2pdf(x,4);figure,plot(x,y)
```

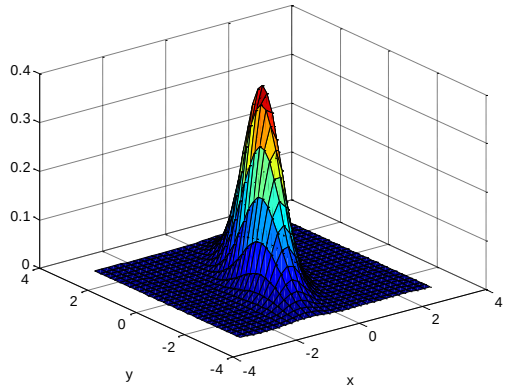


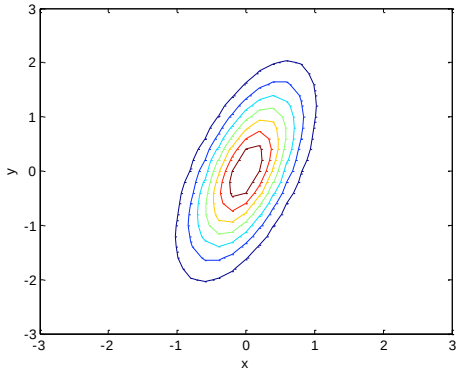
Other functions are betapdf (Beta), binopdf (binomial), exppdf (exponential), unifpdf (uniform), etc

Multivariable normal distribution: The function is *mvnpdf*.

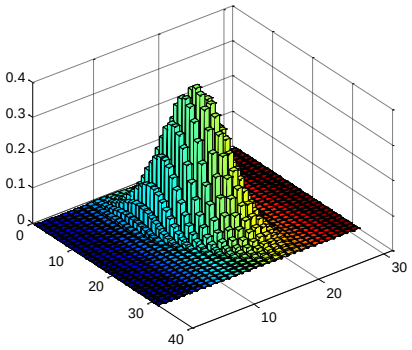
```
media=[0 0];
matriz_cov=[.25 .3;.3 1];

x=-3:.2:3;
y=x;
[xx,yy]=meshgrid(x,y);

F=mvnpdf([xx(:) yy(:)],media,matriz_cov);
F=reshape(F,length(xx),length(yy));
```

```
figure,surf(x,y,F),xlabel('x'),ylabel('y')
figure,contour(x,y,F),xlabel('x'),ylabel('y')
```






```
bar3(F)
```



---

5. Dynamic systems

The *Control Systems Toolbox* contains functions that allow the computation and representation of frequency responses and time responses, and includes functions to design controllers on the basis of root locus plots.

Transfer function: Enter the numerator polynomial and the denominator polynomial (notice the brackets). For instance, $H(s)=\frac{2}{s^2+0.5s+1}$, is introduced as

```
>> num=2;
>> den=[1 0.5 1];
```

Newest versions use transfer function objects (function tf)

```
>> H=tf(num,den)

H =
```

```
          2
---------------
s^2 + 0.5 s + 1

Continuous-time transfer function.
```

5.1 Frequency response of linear systems

Functions: The main functions are **`bode`** (for Bode diagrams), **`nyquist`** (for polar plots), **`nichols`** (for phase-gain diagrams) and **`freqs`** (to obtain the complex value for the frequency response). We recommend exploring the help of the functions presented: `>>help function_name`.

Syntax: There are several levels.

The simplest one (see Fig. a) is:

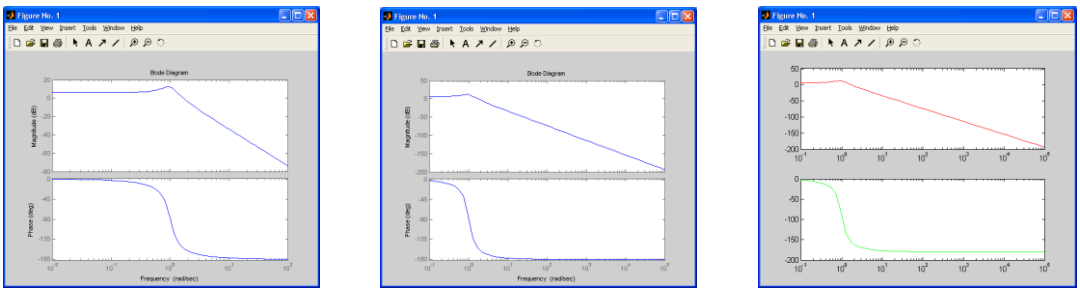
```
>> bode(num,den)
```

To specify the frequencies axis (see Fig. b), use logspace:

```
>> w=logspace(-1,5); %frequencies from 0.1 to 1e5
>> bode(num,den,w)
```

To store the values for the phase and magnitude (to represent them later,) use output arguments (see Fig. c):

```
>> [mag,fase]=bode(num,den,w);
>> subplot(211),semilogx(w,20*log10(mag),'r')
>> subplot(212),semilogx(w,fase,'g')
```



(a) (b) (c)

Fig. 6. Frequency response

5.2 Time response of linear systems

Functions: The main functions are **`impulse`** (for impulse responses), **`step`** (for step responses) and **`lsim`** (*linear simulation*, for arbitrary excitations such as ramps, sinusoids, mixed signals, etc.). For more information, type `>>help function_name`.

Syntax: There are several levels (notice the semicolon use).

The simplest syntax (see Fig. a) is:

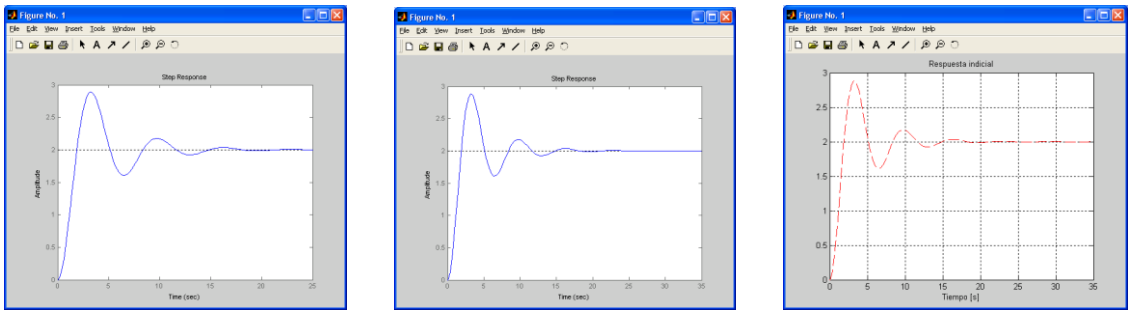
```
>> step(num,den)
```

To specify time span (see Fig. b), type:

```
>> t=linspace(0,35);
>> step(num,den,t)
```

To generate a variable with the time response samples to plot it later (see Fig. c), type:

```
>> y=step(num,den,t);
>> plot(t,y,'r--')
>> grid,title('Respuesta indicial'),xlabel('Tiempo [s]')
```

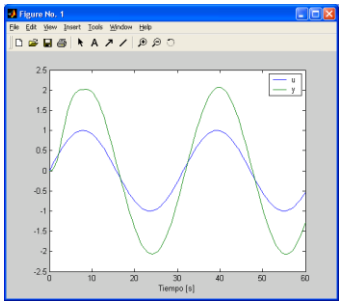


(a) (b) (c)

Fig. 7. Time response

To simulate general excitations, first define them:

```
>> t=linspace(0,60,100);
>> u=sin(0.2*t);
>> y=lsim(num,den,u,t);
>> plot(t,u,t,y)
>> legend('u','y'),
>> xlabel('Tiempo [s]')
```

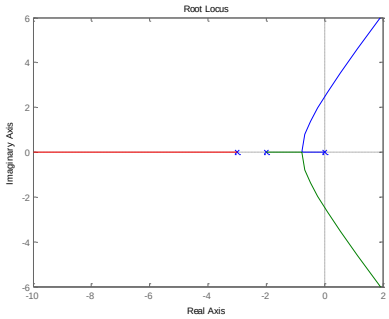


(d)

5.3 More functions for control systems

To compute and plot the Evans root locus, you can use the command `rlocus`:

```
>> num=1;
>> den=conv([1 3 0],[1 2]);
>> rlocus(num,den)
```



Finally **`sisotool`** opens a GUI for the design of control systems:

```
>> sisotool
```