TEMA 3

Programación de ficheros M

1. Introducción

La potencia de MATLAB reside en su capacidad para ejecutar una larga serie de comandos almacenados en un fichero de texto. Estos ficheros reciben el nombre de ficheros M ya que sus nombres tienen la forma *nombre_fichero.m*.

Los ficheros M comerciales se venden agrupados en librerías llamadas *toolboxes*. El usuario de MATLAB puede editar y modificar los ficheros M de las *toolboxes* y también puede crear sus propios ficheros M.

Para ver qué *toolboxes* están instaladas, así como su versión, teclear **`>>ver`** en la ventana de comandos.

Para ver qué ficheros hay en cada *toolbox* y qué hacen se puede hacer **`>>help nombre_toolbox`**, por ejemplo, **`>>help matlab\general`** o **`>>help control`**.

1.1 Editor de ficheros M

Puesto que los ficheros M son simples ficheros de texto, cualquier editor de textos (como, por ejemplo, el bloc de notas o *notepad*) es válido. Sin embargo, se recomienda trabajar con el editor de ficheros M del propio MATLAB puesto que dispone de facilidades para la depuración de los programas y, además, el código de colores que utiliza con las palabras reservadas hace la programación mucho más agradable.

Para abrir el editor de ficheros M clicar en el icono  . También se puede hacer doble clic sobre cualquier fichero M ya existente, o bien, desde la barra de menús, seleccionar **`File → New → M-File`**.

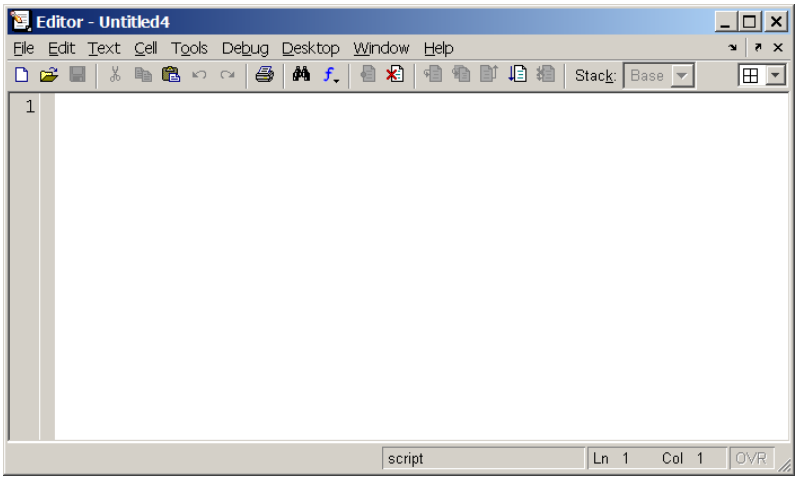


Fig. 1. Editor de ficheros M en MATLAB v6

Directorio de trabajo: Los ficheros creados por el usuario se guardan, por defecto, en el directorio de trabajo `<work>`. Aparte del fichero M de interés, el editor genera archivos de texto de recuperación con el mismo nombre que el fichero M pero con extensión `*.asv` (de *autosave*).

Si se quiere ver el contenido de un fichero M en la ventana de comandos (sin tener que abrir el editor), basta con teclear **`>>type nombre_fichero`**. P. ej.: **`>>type roots`**. Nota: Las funciones *built-in* no son ficheros M pero llevan asociado un fichero M con los comentarios de ayuda. Por ejemplo, probar **`>>type who`** y **`>>type who.m`**.

Opciones de la barra de menús: La barra de menús y la barra de herramientas presentan muchas utilidades. Se recomienda echar un vistazo a las opciones disponibles e intentar deducir para qué sirven.

Algunas opciones interesantes que destacamos son:

- **`File → Preferences…`** (para cambiar los parámetros generales referentes a color, control de cambios, fuentes…),
- **`Cell`** (para los casos en que se desee ejecutar tan sólo una parte del código),
- **`Tools → Check Code with M-Lint`** (para obtener sugerencias para corregir o mejorar el código),
- **`Debug → Run`** (para salvar y ejecutar el fichero M. Esta opción es equivalente a clicar sobre el icono)

1.2 Depuración de ficheros M

Semáforo: El editor M de la v7 presenta una especie de semáforo en el marco (ver Fig. 2). Si MATLAB tiene sugerencias (*warnings*) a hacer (como, por ejemplo, poner el punto y coma en la instrucción de la línea 1) ello lo indica con el color naranja. Si todo le parece bien lo indica con el color verde. Y, si detecta errores, aparece el color rojo.

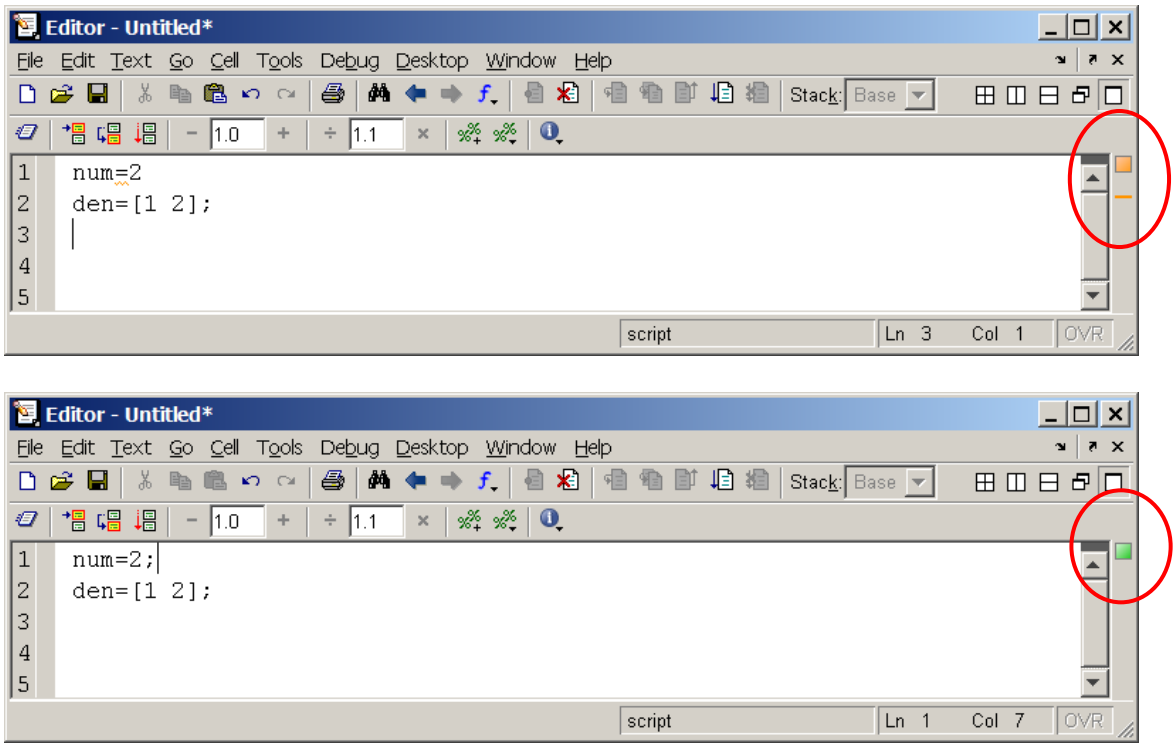
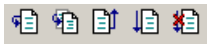
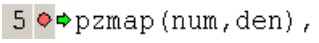


Fig. 2. Editor de ficheros M en MATLAB v7

Breakpoints: Es posible poner *breakpoints* para depurar el código. Clicar en los botones  para poner o quitar *breakpoints* en las líneas deseadas o bien usar las opciones de **Debug** en la barra de menús.

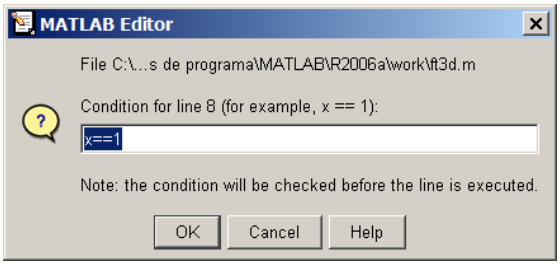
Para iniciar la ejecución clicar en  . Usar el resto de iconos  para pasar de una instrucción a otra, para eliminar los *breakpoints*, etc.

Mientras se ejecuta el programa con *breakpoints*, el *prompt* de Matlab pasa a ser **K>>**. Asimismo, la línea actual se marca dentro del fichero M por medio de una flecha verde:

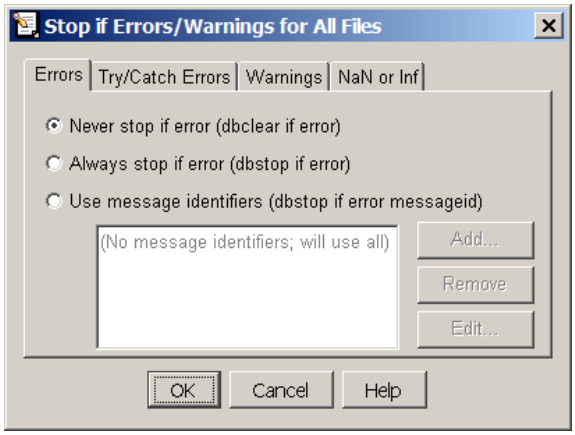


Los *breakpoints* pueden ser de tipo

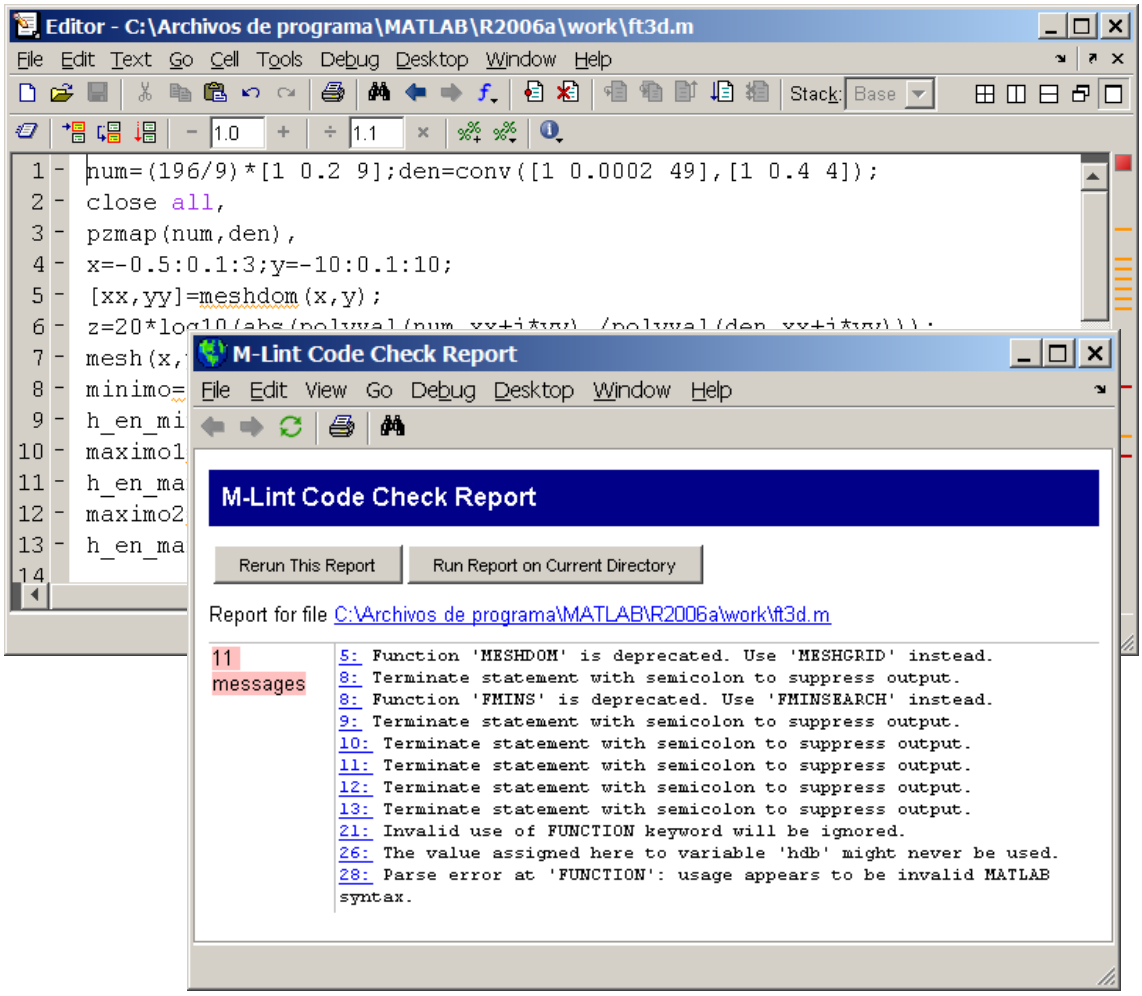
- Estándar: son de color rojo. Cuando el programa llega a ellos se detiene su ejecución.
- Condicionales: son de color amarillo. Para ponerlos hay que situarse en la línea deseada y seleccionar **Debug → Set/Modify Conditional Breakpoint…** Se abrirá un cuadro de diálogo donde se puede indicar cuál es la condición que detendrá el programa:



- De error: Son de color rojo. Para ponerlos hay que situarse en la línea deseada y seleccionar **Debug → Stop if Errors/Warnings…** Se abrirá un cuadro de diálogo donde se puede indicar cuál es el tipo de error, *warning* o valor (**NaN**, **Inf**) que detendrá el programa:



M-Lint Code Check: Para activarlo ir a **`Tools → Save and Check Code with M-Lint`**. A continuación se muestra un ejemplo:



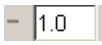
El informe nos dice lo siguiente:

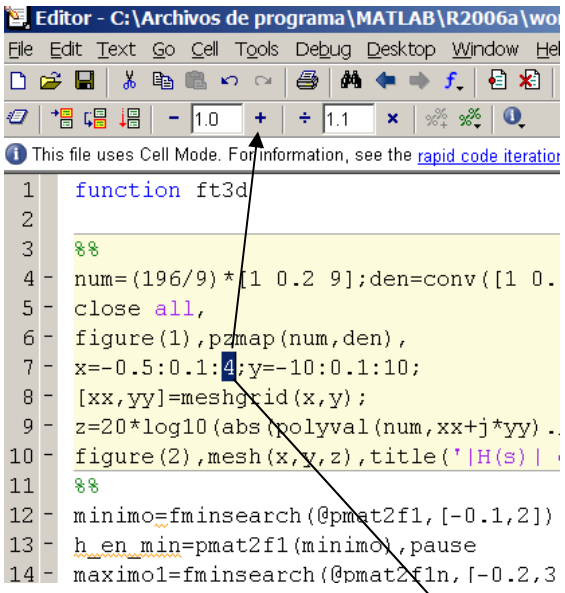
- Nos informa de que las funciones **`meshdom`** y **`fmins`** ya no existen y que hay que usar **`meshgrid`** y **`fminsearch`** en su lugar.
- Nos indica en qué líneas no hemos puesto punto y coma.
- Nos avisa de que hay variables que hemos creado pero que no hemos usado (**`hdb`**).
- Y, finalmente, nos avisa que no podemos declarar una función en medio de un *script* (sólo pueden declararse funciones dentro de funciones). Por tanto hay que convertir el *script* **`ft3d`** en una función. Para ello la primera línea deberá ser **`function ft3d`**.

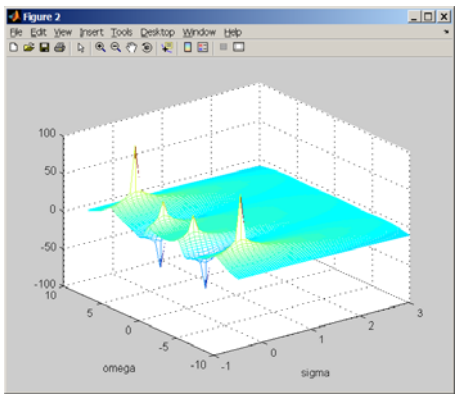
Cell mode: Para activar este modo ir a **`Cell → Enable Cell Mode`**. Aparecerá la barra

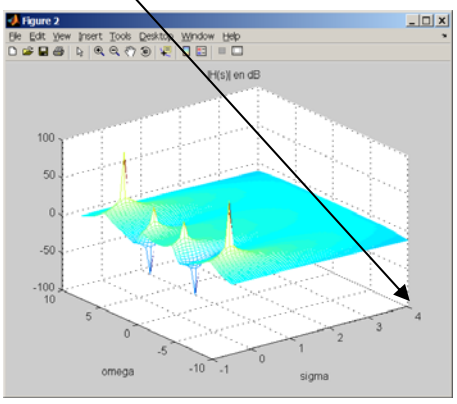
Este modo resulta útil para ejecutar partes del código o cambiar parámetros sin tener que salvar y ejecutar el fichero M entero.

Para delimitar las instrucciones que forman una celda teclear el doble tanto por ciento, **%%**, al inicio y al final (o usar las opciones de la barra de menús). Para ejecutarla, ponerse sobre ella y clicar en  .

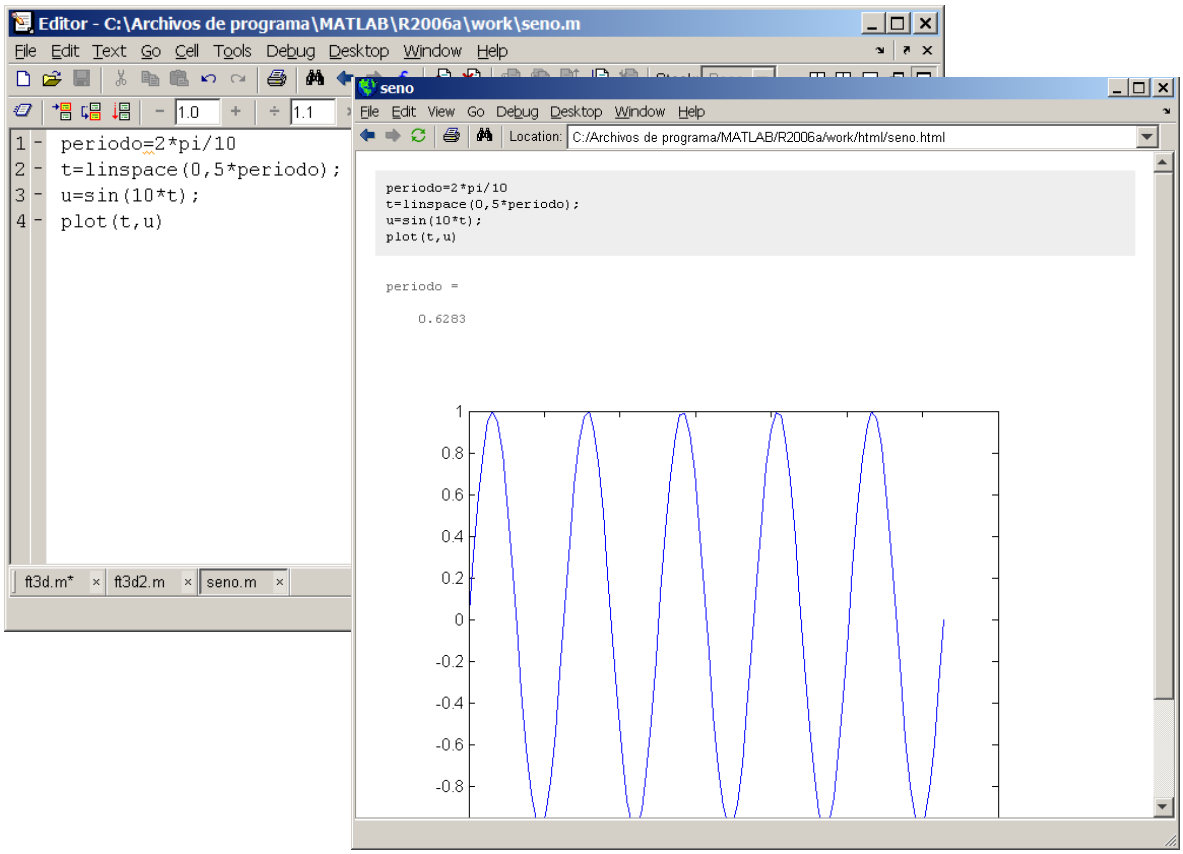
Para cambiar un parámetro y ver inmediatamente su efecto, ponerse sobre él y usar, p.ej., los signos + y - del botón  . La celda se ejecutará automáticamente (es el modo llamado *rapid code iteration*).



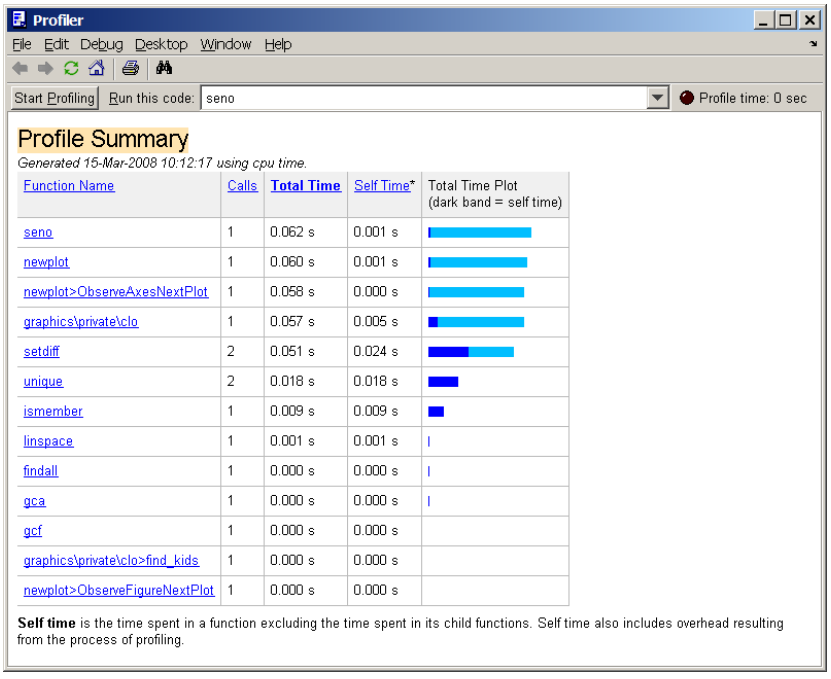




Publicar: Para crear un documento *html* con las instrucciones de un *script* así como con los resultados de la ejecución, clicar en . Por ejemplo:



Profiler: Para abrirlo se puede hacer desde **`Tools` → `Open Profiler`** y, una vez dentro, clicar en el botón **`Start Profiling`**. Esta herramienta permite ver cuanto tiempo consumen las instrucciones del código así como cuantas veces son llamadas. Ello ayuda a eliminar redundancias y hacer más eficiente la programación.



2. *Scripts* y funciones

Existen dos tipos de ficheros M: los *scripts* y las funciones (*functions*).

Scripts:

- No tienen argumentos de entrada ni de salida.
- Para ejecutar un *script* basta con teclear su nombre (sin extensión) en la ventana de comandos.
- Todas las variables creadas por del *script* aparecen en el *workspace*.

Funciones:

- Las funciones sí tienen argumentos de entrada y/o de salida.
- Para ejecutar una función hay que especificar el valor de sus argumentos de entrada.
- Las variables internas de la función no aparecen en el *workspace* (a no ser que se indique lo contrario declarándolas como globales o que sean argumentos de salida de la función).

2.1 Programación de *scripts*

Un *script* es un fichero de texto, de extensión `*.m`, que contiene una secuencia ordenada de comandos MATLAB (instrucciones primitivas, funciones y/o llamadas a otros *scripts*) escritos tal y como se introducirían en la ventana de comandos (pero sin el *prompt* >>).

Ejemplo 1. Programación de *scripts*

La siguiente figura muestra el *script* `curva.m`.

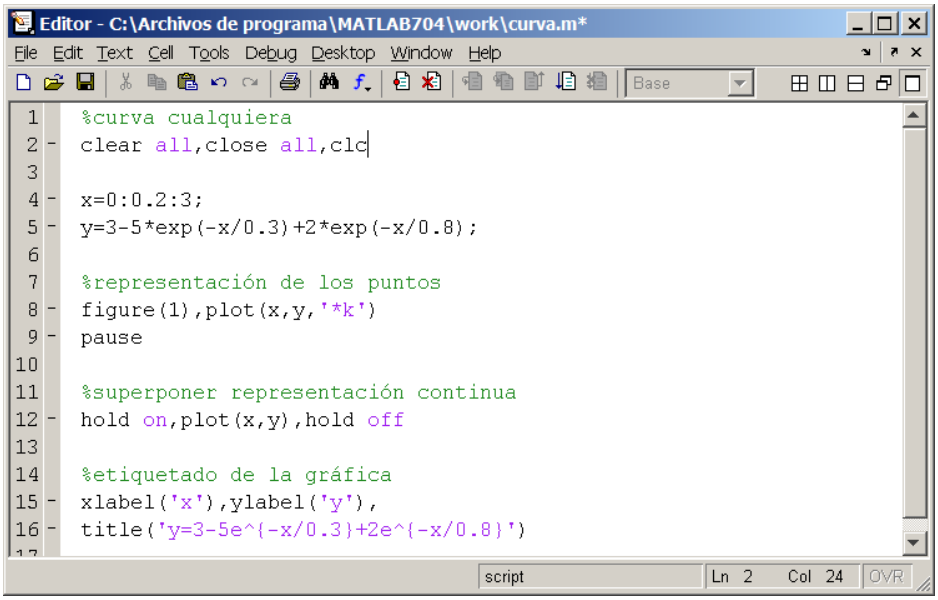
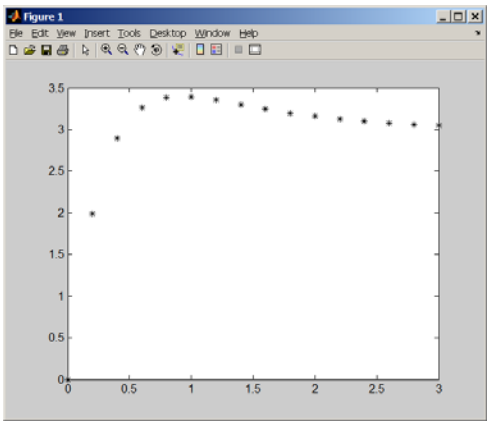
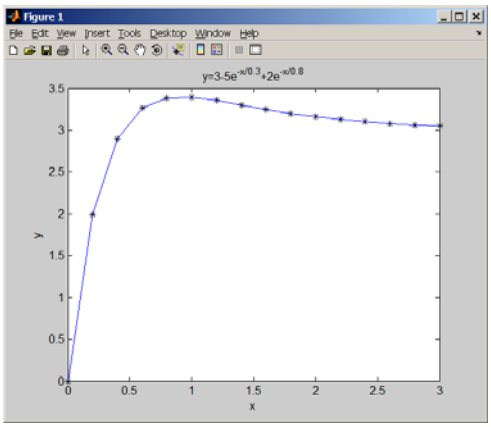


Fig. 3. Ejemplo de fichero M tipo *script*

Su ejecución da como resultado las siguientes figuras:





Los *scripts* utilizan las variables almacenadas en el *workspace* de MATLAB y las variables creadas por el *script*, a su vez, también se almacenan en el *workspace*.

Para ejecutar un *script* basta con teclear su nombre (sin extensión) en la ventana de comandos, o bien seleccionar la opción **`Debug→Run (F5)`**, o bien clicar en el icono del editor de ficheros M.

Incluir comentarios: Para poner un comentario basta con poner el símbolo % al principio del comentario (no hace falta cerrar el comentario con otro %). Notar que, por defecto, el editor presentará el comentario en color verde.

Instrucciones de "limpieza": Aunque no es obligatorio, lo habitual es poner al inicio del *script* un conjunto de instrucciones "de limpieza" del tipo **`hold off`**, **`axis`**, **`subplot`**, **`clear all`**, **`close all`**, **`close all hidden`**, **`clc`**...

Algunas funciones útiles:

- **`pause`**: Detiene la ejecución del *script* y se queda a la espera de que el usuario pulse una tecla para continuar. También es posible detener la ejecución del programa un tiempo determinado, por ejemplo, **`pause(2)`**.

- **`disp`**: Muestra texto por la pantalla de comandos. Por ejemplo, la ejecución de las instrucciones:

```
N=15.6;
disp(['Y el valor de N es... ',num2str(N),'!!'])
```

  da como resultado el siguiente texto en la ventana de comandos:

```
Y el valor de N es... 15.6!!
```

El nombre de la función `num2str` corresponde al inglés *number-to-string*. Notar que el argumento de entrada de `disp` es una concatenación de cadenas de caracteres o *strings*. También es posible hacer lo siguiente:

```
>> disp({'hola';'adios'})
    'hola'
    'adios'
```

- **`figure`**: Si antes de un plot, se incluye la instrucción **`figure`** o **`figure(1)`**, **`figure(2)`**, etc., en el momento de ejecutar el *script*, la ventana de figura aparecerá delante automáticamente (y así no habrá que irla a buscar con el ratón o la combinación de teclas `<Alt><Tab>`).

- **`echo on/off`**: Se usa en las demos. Sirve para que las instrucciones del *script* vayan saliendo por la ventana de comandos.

- **`tic`** y **`toc`**, o bien **`etime`**: Sirven para ver la duración de los cálculos.

- `...` (tres puntos): para escribir una misma instrucción en dos líneas diferentes (por ejemplo, cuando entramos matrices grandes o usamos las funciones **`set/get`** con muchos argumentos es muy habitual que una línea de código no baste)

2.2 Programación de funciones

Un fichero M tipo *function* también es una secuencia ordenada de comandos MATLAB pero, a diferencia de los *scripts*, para su ejecución necesitará que se le introduzcan argumentos de entrada y, como respuesta, generará argumentos de salida.

Ejemplo 2. Programación de *functions*

La siguiente figura muestra la función `suma_resta.m`.

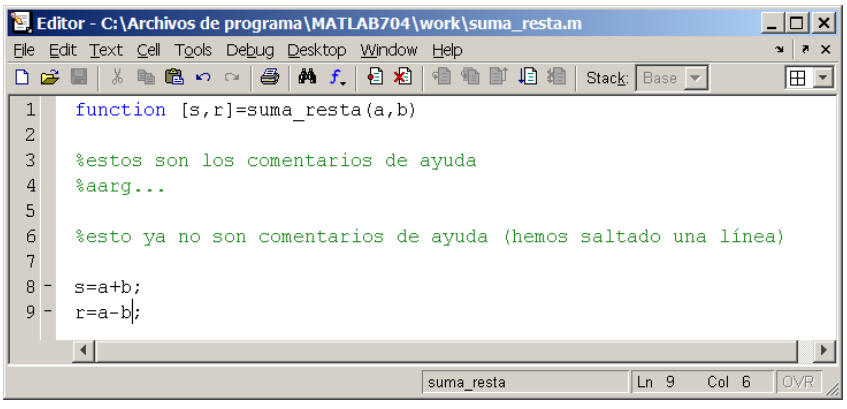


Fig. 4. Ejemplo de fichero M tipo función.

La declaración de los parámetros tanto de entrada como de salida así como el nombre de la función se realiza en la primera línea del fichero M.

Es posible declarar varias funciones, una detrás de otra, en un fichero M de tipo *function*. Por defecto, el nombre del fichero M debe coincidir con el nombre de la primera función.

Ejecución: La siguiente figura ilustra la ejecución de la función **`suma_resta`**:

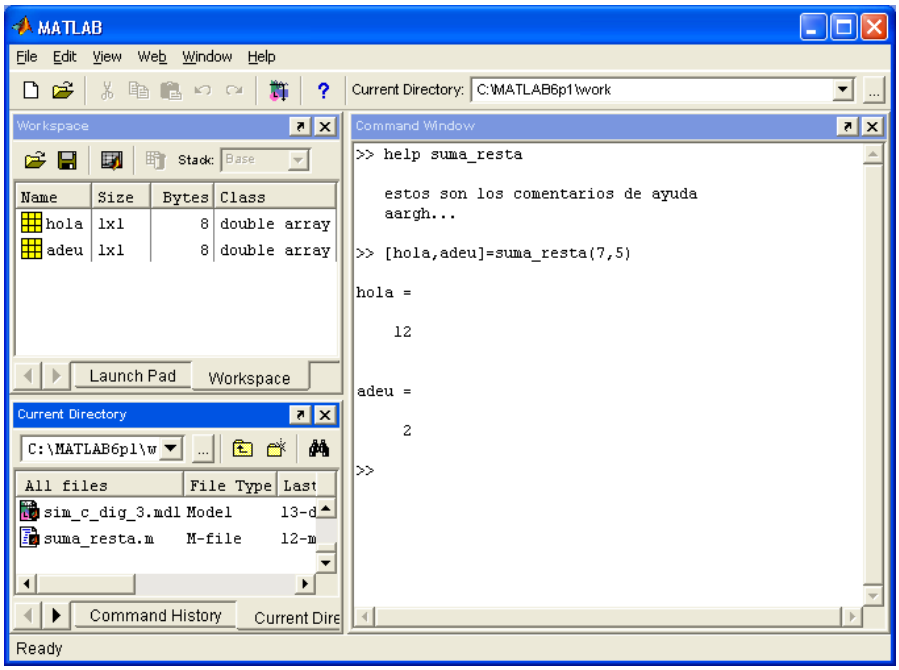


Fig. 5. Help y ejecución de la función suma_resta en la ventana de comandos.

Estructura: Un fichero-M de función se estructura en tres partes:

- Cabecera: Es donde se declara que es una función. Por ejemplo:

```
function [seno,coseno,tangente]=func1(ang)
```

- Comentarios de ayuda: Esto es opcional, van a continuación de la declaración de función y se suele dejar una línea en blanco entre ellos y la declaración. Los comentarios de ayuda es lo que aparece en pantalla al teclear `>>help nombre_función`.

```
% Calcula el seno, el coseno y la tangente del
% ángulo indicado por la variable 'ang'
```

- Colección de instrucciones:

```
seno=sin(ang);
coseno=cos(ang);
tangente=tan(ang);
```

Notar que:

- Los argumentos de salida van entre corchetes.
- Los argumentos de entrada van entre paréntesis.
- No es necesario terminar el fichero con la instrucción `end` (aunque, si se decide hacerlo, entonces todas las funciones del fichero deberán terminar en `end`).
- El nombre del fichero coincide con el nombre de la primera función declarada.
- Los comentarios van precedidos del símbolo % (que no es necesario cerrar al final del comentario)
- Las ayudas de las funciones de las *toolboxes* indican qué hace la función y cual es la sintaxis de uso. Aunque en estas ayudas el nombre de la función aparece siempre en mayúsculas, las funciones en MATLAB se invocan generalmente en minúsculas.

Variables globales: A diferencia de lo que ocurre con los *scripts*, las variables que se generan dentro de una función (y que no son argumentos de salida de ésta) no pasan al *workspace*. La única excepción es cuando se declaran (dentro y fuera de la función) como variables globales, por ejemplo:

```
>>global PERIODO_MUESTREO
```

Por una cuestión de estilo las variables globales suelen recibir nombres largos y en mayúsculas, pero no es obligatorio.

Número de argumentos de entrada y salida. Parámetros opcionales: Son las funciones **`nargin`** (*number of input arguments*) y **`nargout`** (*number of output arguments*); **`varargin`** (*variable number of input arguments*) y **`varargout`** (*variable number of output arguments*); **`nargchk`** (*check number of input arguments*) y **`nargoutchk`** (*check number of output arguments*).

Estas funciones permiten que la función haga cosas diferentes según el número de parámetros de entrada o salida con que es llamada.

Por ejemplo, la función **`bode`**, cuando se invoca sin argumentos de salida **`>>bode(num,den)`** representa el diagrama de Bode pero no genera ningún resultado numérico; si se invoca con una variable de salida **`>>mag=bode(num,den)`** guarda en **`mag`** los valores del módulo de la respuesta frecuencial y no representa nada; si se invoca con un único argumento de entrada **`>>bode(G)`** interpreta que **`G`** es un objeto **`tf`** (*transfer function*), etc…)

Ejemplo 3. Uso de nargin y nargout

```
function [numz,denz]=discret(nums,dens,Type)

%Sintaxis:  [numz,denz]=discret(nums,dens,Type)
%           Type puede ser 'Tustin', 'BwdRec' o 'FwdRec'

if nargin==0
    disp('Venga!!! Entra un sistema!!!')
```

```
    else
        if Type=='Tustin'
                instrucciones que llevan a cabo la discretización  s = 2/T (z-1)/(z+1)
        end
        if Type=='BwdRec'
                instrucciones que llevan a cabo la discretización  s = (z-1)/(Tz)
        end
        if Type=='FwdRec'
                instrucciones que llevan a cabo la discretización  s = (z-1)/T
        end
    end

    if nargout==0
        disp('buf!')
    end
```

```
>> help discret

 Sintaxis:  [numz,denz]=discret(nums,dens,Type)
            Type puede ser 'Tustin', 'BwdRec' o 'FwdRec'

>> [a,b]=discret

Venga!! Entra un sistema!!!
Warning: One or more output arguments not assigned during call to 'discret'.

>> discret(2,3,'Tustin')

buf!

>>
```

Ejemplo 4. Uso de varargin y varargout

Si queremos que una función pueda ser llamada con un número variable de argumentos de entrada la podemos declarar así:

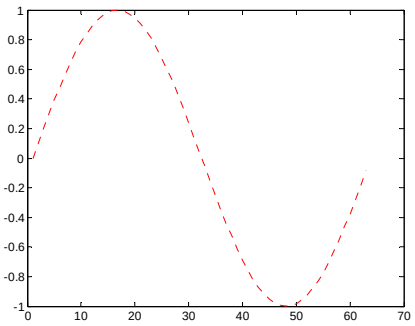
```
function y=func1(x,varargin)

%con, por ejemplo, estas instrucciones
figure,plot(x,varargin{:})
varargin,
x1=varargin{1},
x2=varargin{2},
x3=varargin{3},
x4=varargin{4},
```

Así, a partir de x podemos usar un número variable de argumentos de entrada. Si llamamos la función así (con 4 argumentos de entrada adicionales a x):

```
>>func1(sin(0:.1:2*pi),'color',[1 0 0],'linestyle',':')
```

La variable varargin interna a la función será un cell-array 4x1 donde sus componentes serán, respectivamente, varargin{1}='color', varargin{2}=[1 0 0], etc.

```
varargin =
    'color'    [1x3 double]    'linestyle'    ':'

x1 =
color

x2 =
     1     0     0

x3 =
linestyle

x4 =
:
```



Para que una función genere un número variable de argumentos de salida podemos usar `varargout`

```
function [varargout]=func2(x,varargin)
```

Funciones auxiliares y anidadas: Un fichero de función puede contener declaraciones de otras funciones en su interior.

En el caso de funciones auxiliares que se usan en una función principal, hay que ponerlas todas al final. No es obligatorio terminar las funciones con un **end** (pero si se termina una de las funciones con **end**, entonces todas las funciones deben terminar con **end**). En el siguiente ejemplo **pmat2f1** y **pmat2f1n** son funciones auxiliares de la función principal **ft3d**.

Para acceder al help de las funciones auxiliares usar el símbolo >.

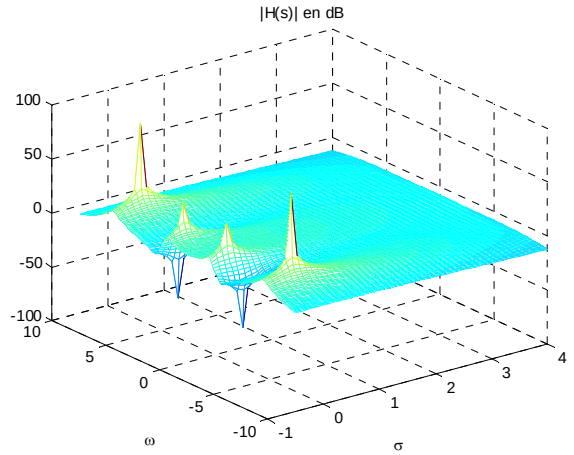
```
>> help ft3d>pmat2f1
    |H(s)| en dB
```

Ejemplo 5. Funciones auxiliares

```
function ft3d

num=(196/9)*[1 0.2 9];den=conv([1 0.0002 49],[1 0.4 4]);
x=-0.5:0.1:4;y=-10:0.1:10;[xx,yy]=meshgrid(x,y);
z=20*log10(abs(polyval(num,xx+j*yy)./polyval(den,xx+j*yy)));
mesh(x,y,z),title('|H(s)| en dB'),xlabel('\sigma'),ylabel('\omega'),

minimo=fminsearch(@pmat2f1,[-0.1,2])
h_en_min=pmat2f1(minimo),pause
```

```
maximo1=fminsearch(@pmat2f1n,[-0.2,3])
h_en_max_1=pmat2f1(maximo1),pause
maximo2=fminsearch(@pmat2f1n,[-0.002,9])
h_en_max_2=pmat2f1(maximo2),pause


function [hdb]=pmat2f1(in)
%   |H(s)| en dB
s=in(1);
w=in(2);
num=(196/9)*[1 0.2 9];den=conv([1 0.0002 49],[1 0.4 4]);
hdb=20*log10(abs(polyval(num,s+j*w)./polyval(den,s+j*w)));

function [hdb]=pmat2f1n(in)
%   |1/H(s)| en dB
s=in(1);
w=in(2);
num=(196/9)*[1 0.2 9];den=conv([1 0.0002 49],[1 0.4 4]);
hdb=-20*log10(abs(polyval(num,s+j*w)./polyval(den,s+j*w)));
```

La ejecución de `>>ft3d` da como resultado:




```
minimo =
   -0.1000    2.9983
h_en_min =
 -317.0491

Exiting: Maximum number of function evaluations has been exceeded
         - increase MaxFunEvals option.
         Current function value: -309.711370
maximo1 =
   -0.0001    7.0000
h_en_max_1 =
  309.7114

maximo2 =
   -0.0001    7.0000
h_en_max_2 =
  306.8104
```

En el caso de funciones anidadas, éstas se declaran en medio del código y, por tanto, hay que terminarlas siempre con un **end**. La función principal también deberá terminar con un **end**.

```
function principal
        ...
        function anidada
        ...
        end
        ...
end
```

Funciones locales (inline functions): Es posible crear funciones locales en la ventana de comandos, en *scripts* y en funciones. La ventaja es que no hace falta guardarlas en un fichero aparte, pero tienen limitaciones (no se pueden anidar y sólo tienen un argumento de salida).

Ejemplo 6. Funciones locales (`inline`)

La función local **`func1`** implementa la siguiente expresión: $h(x) = x^2 \sin(ax) - b$

```
>> func1=inline('x.^2.*sin(a*x)-b','x','a','b')
func1 =
     Inline function:
     func1(x,a,b) = x.^2.*sin(a*x)-b

>> func1=inline('x.^2.*sin(a*x)-b','x','a','b');
>> h=func1([0:pi/3:pi],4,1)
h =
   -1.0000   -1.9497    2.7988   -1.0000
```

Funciones privadas: Son las funciones que están dentro de los directorios de nombre `<private>`. Sólo pueden ser vistas y llamadas por las funciones y *scripts* que están en el directorio inmediatamente superior al directorio `<private>`. Puesto que sólo son visibles por unas pocas funciones su nombre puede ser el mismo que otras funciones estándar de MATLAB, por ejemplo, `bode.m`. Cuando desde el directorio inmediatamente superior a `<private>` se llame a la función `bode`, la que se ejecutará será la función `bode.m` de `<private>` y no la estándar de MATLAB.

3. Lenguaje de programación

3.1 Comandos de entrada y salida

Los comandos de entrada y salida son los que permiten al programa la comunicación con el usuario. Se recomienda hacer el **`help`** de las funciones que se presentan a continuación y probar las diferentes opciones que aparecen.

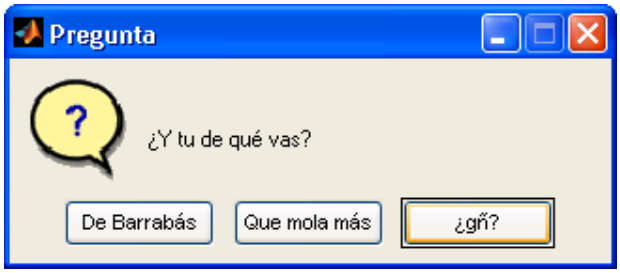
Entrada por teclado: Se usa la función **`input`**

```
>> x=input('Entrar un número: ')
Entrar un número: 7

x =
     7

>> y=input('Entrar un texto: ','s')
Entrar un texto: hola

y =
hola
```

(Notar que, en el *workspace*, `y` es un `char array` de dimensiones `1×4` y `8 bytes`. Esto lo podemos ver tanto en la ventana *Workspace* como al teclear `>>whos` en la ventana de comandos)

Entrada por ratón: También es posible abrir una caja de diálogo con **`questdlg`**,

```
respuesta=questdlg('¿Y tu de qué vas?','Pregunta',...
'De Barrabás','Que mola más','¿gñ?','¿gñ?')
```



(Notar cómo se ha indicado cuál es la selección por defecto)

Salida por pantalla: Las funciones **`disp`** y **`sprintf`** muestran texto y resultados en la ventana de comandos.

`disp`	(para escribir cadenas de caracteres en la ventana de comandos)
`sprintf`	(para presentar datos en un formato determinado)

Con **`disp`** pueden utilizarse algunas funciones de conversión tales como:

- `num2str` (*number-to-string*),
- `int2str` (*integer-to-string*),
- `mat2str` (*matrix-to-string*),
- `lab2str` (*label-to-string*),
- `poly2str` (*polynomial-to-string*),
- `rats` (*rational approximation*),

```
>> disp(['polinomio=' poly2str([3 4 5],'z')])
polinomio=   3 z^2 + 4 z + 5

>> mat2str(ones(2))
ans =
[1 1;1 1]

>> rats(3.4)
ans =
     17/5
```

Para **`sprintf`** existen diversas opciones: `\n`, `\r`, `\t`, `\b`, `\f`, `\\`, %%,... (por ejemplo, `sprintf('\n')` es el cambio de línea) y formatos: `%d`, `%i`, `%o`, `%u`, `%x`, `%X`, `%f`, `%e`, `%E`, `%g`, `%G`, `%c`, `%s`

```
>> sprintf('%0.2f',(1+sqrt(5))/2)
ans =
1.62

>> sprintf('El resultado es %0.5f',(1+sqrt(5))/2)
ans =
El resultado es 1.61803
```

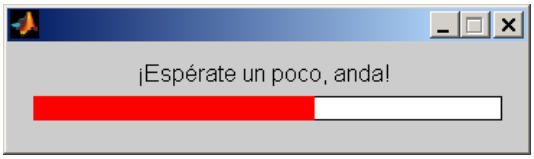
Otras opciones útiles:

```
>> title('\omega_n')
>> xlabel('\Theta_n')
>> text(0.2,0.2,'\it \phi^n')
```

Nota: en el ejemplo, la función **`text`** escribe el texto ϕ^n en las coordenadas (0.2,0.2) del último gráfico creado. Para más información, teclear **`>>help LaTex`**.

Barras de estado: Por ejemplo,

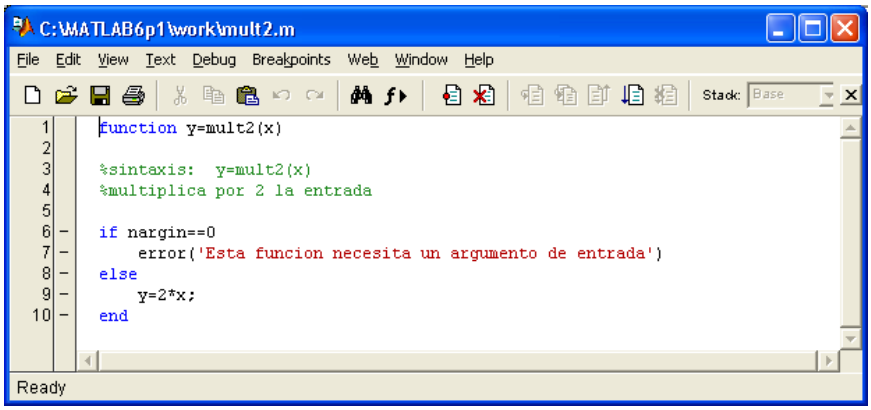
```
h=waitbar(0,'¡Espérate un poco, anda!');
for i=1:5,
  pause(1),waitbar(i/5,h);
end,
pause(1),close(h)
```



Mensajes de error: La función es **`error.`**

Ejemplo 7. Mensajes de error

Se ha editado una función **`mult2.m`**



Al intentar utilizarla sin argumentos de entrada en la ventana de comandos, el resultado ha sido:

```
>> z=mult2
??? Error using ==> mult2
Esta funcion necesita un argumento de entrada
```

Nota: `nargin` (*number of input arguments*) indica con cuántos argumentos se ha llamado a la función.

3.2 Indexado en MATLAB

Índices: En Matlab, el primer índice es 1.

x	=	[0.3	0.2	0.5	0.1	0.6	0.4	0.4];
Índice:		*1*	*2*	*3*	*4*	*5*	*6*	*7*

Para acceder a las diversas componentes de vectores y matrices hay que usar los paréntesis, ( ).

Comprobar que `>>x(0)` da un mensaje de error:

```
>> x(0)
??? Subscript indices must either be real positive integers
or logicals.

>> x(5)
ans =
     0.6
```

Función ***find****:* Sirve para encontrar los índices correspondientes a los elementos de un vector (o matriz) que cumplen cierta condición booleana.

```
>> x=[0.3 0.2 0.5 0.1 0.6 0.4 0.4];

>> i=find(x==0.9)
i =
     []

>> i=find(x>0.4)
i =
     3     5
```

Nota: Para matrices la sintaxis es: **`[i,j]=find(A~=0)`**

Otras funciones útiles son **`sort`**, **`rot90`**, **`flipud`**, **`fliplr`** (en el caso de vectores, **`fliplr(x)`** y **`flipud(x)`** son equivalentes a **`rot90(x,2)`**)

Arrays multidimensionales: Son una extensión de las matrices. Una matriz es un array bidimensional formado por filas y columnas. Un array tridimensional está formado por filas, columnas y páginas (las dimensiones superiores a 3 no tienen nombre definido). A continuación se muestra un array de 2 filas, 2 columnas y 3 páginas:

```
>> A=[1 2;3 4];
>> A(:,:,2)=eye(2);
>> A(:,:,3)=eye(2)*2;
>> A
A(:,:,1) =
     1     2
     3     4
A(:,:,2) =
     1     0
     0     1
A(:,:,3) =
     2     0
     0     2
```

Otro ejemplo: Cuando la función **`bode`** se aplica a objetos *transfer function*, el resultado son arrays tridimensionales. Si queremos quedarnos con los valores del módulo, basta con aplicar la función **`squeeze`** para eliminar las dimensiones unitarias:

```
>> mag=squeeze(mag);
>> G=tf(1,[1 1]);
>> [mag,fase]=bode(G);
>> size(mag)
ans =
     1     1    45
>> mag=squeeze(mag);
>> size(mag)
ans =
    45     1
```

Tipos de datos cell y struct: A veces se quiere guardar en una única variable datos de diferentes tipos y/o dimensiones. En ese caso, las variables de tipo **`cell`** y **`struct`** son útiles.

Ejemplo con cell.

```
>> matrices_1=cell(1,3)
%creamos la variable de tipo cell de dimensiones 1 por 3
matrices_1 =
     []     []     []

>> matrices_1{1}=1
%para acceder a cada celda hay que indicar su índice con {   }
matrices_1 =
    [1]     []     []

>> matrices_1{2}=eye(2)
matrices_1 =
    [1]    [2x2 double]     []

>> matrices_1{3}=eye(3)
matrices_1 =
    [1]    [2x2 double]    [3x3 double]

>> matrices_1{3}
ans =
     1     0     0
     0     1     0
     0     0     1

>>
```

Ejemplo con struct: Se sugiere hacer `>>help struct`

```
>> mis_cosillas.cosa1='hola'

mis_cosillas =

    cosa1: 'hola'

>> mis_cosillas.cosa2=1:5

mis_cosillas =

    cosa1: 'hola'
    cosa2: [1 2 3 4 5]

>> mis_cosillas.cosa3=zeros(3)

mis_cosillas =

    cosa1: 'hola'
    cosa2: [1 2 3 4 5]
    cosa3: [3x3 double]

>> mis_cosillas.mas_cosas.cosa4=8.2

mis_cosillas =
```

```
        cosa1: 'hola'
        cosa2: [1 2 3 4 5]
        cosa3: [3x3 double]
    mas_cosas: [1x1 struct]

>>
```

Formato de fechas en Matlab: MATLAB guarda las fechas en formato numérico. La fecha 1 corresponde al 1 de enero del año 0.

La función **`datestr`** muestra la fecha en caracteres alfanuméricos

```
>> datestr(1)
ans =
01-Jan-0000
```

La función **`datenum`** convierte una fecha alfanumérica en formato numérico. Para asegurar que la fecha está en nuestro sistema (día-mes-año) y no en el sistema americano (mes-día-año), se recomienda explicitar el segundo argumento de entrada (correspondiente al formato)

```
>> datenum('1/2/07')
ans =
      733044
>> datestr(ans)
ans =
02-Jan-2007

>> datenum('1/2/07','dd/mm/yy')
ans =
      733074
>> datestr(ans)
ans =
01-Feb-2007
```

Otras funciones útiles son: **`now`**, **`today`**.

3.3 Bucles

Bucles FOR

Sintaxis: `>>for (índices en sentido creciente),`
`instrucciones,`
`end`

El contador puede ser `i=1:length(x)` o bien `i=0:0.1:7` o bien `i=10:-1:0`, etc.

```
>> for i=1:3
disp('bah!')
end
bah!
bah!
bah!
```

Ejemplo 8. Bucle FOR. Cálculo de los intereses totales en un préstamo bancario

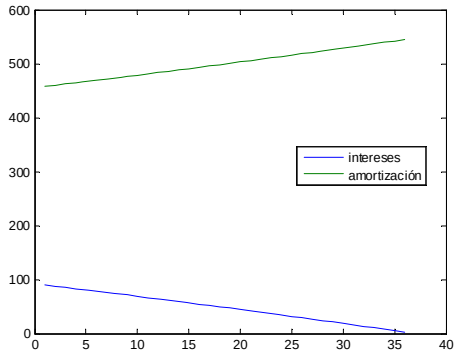
(Magrab,05)
La cuota mensual *c* de un préstamo de *P* euros viene dada por la siguiente expresión

$$c = \frac{iP}{1-(1+i)^{-m}}$$

donde *m* es el número de mes e *i* es el interés expresado mensualmente $i = I_a/1200$ (I_a es el tipo de interés anual y *m* el número de mensualidades). Cada pago c tiene una parte de intereses pagados al banco, i_n, y una parte de amortización, a_n=c-i_n, n=1,2,...,m. Estas cantidades varían mes a mes (los primeros meses la amortización es pequeña y los intereses grandes y en los últimos meses es al revés). El capital pendiente b_n después de cada pago va disminuyendo con la amortización, $b_n = b_{n-1} - a_n$ (antes de ningún pago, el capital pendiente es b_0=P) y los intereses que se pagan cada mes vienen dados por el capital pendiente del mes anterior, $i_n = ib_{n-1}$.

Suponer que tenemos un préstamo de 18000€ a 36 meses con un tipo anual del 6%. ¿Cuántos intereses pagaremos en total?

```
>> P=18000;TAE=6;m=36;
>> i=TAE/1200;
>> c=P*i/(1-(1+i)^-m)
c =
  547.5949

>> in=zeros(1,m);
>> an=in;bn=in;
>> bn(1)=P;%bo=P
>> for n=2:m+1
      in(n)=i*bn(n-1);%intereses
      an(n)=c-in(n);%amortización
      bn(n)=bn(n-1)-an(n);%capital pendiente
end
>> fprintf(1,'Intereses totales=%5.2f€\n',sum(in))
Intereses totales=1713.42€

>> plot(1:m,in(2:end),1:m,an(2:end))
>> legend('intereses','amortización')
```



Antes de ejecutar un bucle for es recomendable inicializar las variables con la función zeros a fin de optimizar la ejecución (si no, MATLAB asignará memoria dinámicamente para las expresiones del bucle).

Bucles WHILE

Sintaxis:
```
>> while (condición booleana),
        instrucciones,
        end
```

```
>> i=3;
>> while gt(i,1)
i=i-1;
disp('burp')
end
burp
burp
>>
```

Operadores relacionales: Los operadores relacionales son `==` (igualdad), `~=` (desigualdad), `<`, `<=` (menor, menor o igual), `>`, `>=` (mayor, mayor o igual).

También existen en formato función: `eq` (*equal*), `ne` (*not equal*), `ge` (*greater than or equal*), `lt` (*less than*),... Por ejemplo, `gt(i,1)` indica si el valor de la variable "i" es mayor que "1" (el nombre de la función corresponde a *greater than*).

Para ver las posibilidades y significados, escribir help y a continuación un punto:

```
>>help .
```

Ejemplo 9. Operadores relacionales

Si se quiere evaluar la siguiente expresión [mag,05]

$$f(x)=\begin{cases} e^{-x/2} & a \le x < b \\ 0 & \text{otro caso} \end{cases}$$

para a=-1 y b=2, podemos hacer

```
>> a=-1;b=2;
>> x=-3:3;
>> g=exp(x/2).*(a<=x & x<b)
g =
         0         0    0.6065    1.0000    1.6487         0         0
```

donde la expresión de la condición booleana es

```
>> (a<=x & x<b)
ans =
     0     0     1     1     1     0     0
```

Operadores lógicos: Los operadores lógicos son `and` (&), `not` (~), `or` (|), `xor` (ésta es una función).

Otras funciones útiles:
`isempty` (*is empty?*), `ischar`, `isnan`, `isinf`, `isfinite`, `isglobal`,
El valor `1` corresponde a verdadero mientras que el valor `0` corresponde a falso.

```
>> x=[]
x =
     []

>> isempty(x)
ans =
     1
```

`strcmp` (*string comparison*)…

```
>> y='hola'
y =
hola

>> strcmp(x,'adios')
ans =
     0
```

Otros comandos:

Para abortar la ejecución del código

`break`	sale del bucle. Aborta la ejecución del *script*.
`return`	vuelve a teclado o a la función que había llamado al bucle

Para redondear

`ceil`	`ceil(1.4)=2`	`ceil(-1.4)=-1`
`floor`	`floor(1.4)=1`	`floor(-1.4)=-2`
`fix`	`fix(1.4)=1`	`fix(-1.4)=-1`
`round`	`round(1.4)=1`	`round(-1.4)=-1`

3.4 Estructuras condicionales

Estructura condicional IF: La sintaxis es la siguiente

```
if (condición booleana)
      Instrucciones
elseif (condición booleana)
      Instrucciones
else
      Instrucciones
end
```

Opciones múltiples SWITCH: Suponer que `evalua` es una variable que puede valer `'si'`, `'no'` o `'puede'`:

```
switch evalua
case {'si','puede'}
 Instrucciones
case 'no'
 Instrucciones
otherwise
 Instrucciones
end
```

4. Funciones que utilizan otras funciones

Function handle: Hay funciones que usan como argumentos de entrada otras funciones. En concreto, el argumento de entrada se conoce como *function handle* y éste se compone del símbolo **@** seguido del nombre de la función a llamar, **@nombre_función**. A continuación se ven varios ejemplos de aplicación:

4.1 Solución numérica de ecuaciones

En este apartado se presentan las funciones **fzero** y **fsolve**. En general, las funciones de MATLAB que empiezan por la letra "f" (**fzero**, **fsolve**, **fminsearch**, **feval**,…) son funciones cuyos argumentos de entrada son otras funciones. (Como siempre, se recomienda teclear **>>help nombre_función** antes de usarlas por primera vez. Ver también **>>help foptions**).

Ejemplo: Se quiere resolver la ecuación $tg^{-1}\left(\frac{2}{\beta}\right) = e^{\beta}$ de forma numérica, con ayuda de las funciones **fzero** y/o **fsolve**.

Creación de la función: En primer lugar, hay que crear un fichero M tipo *function* que tenga como argumento de entrada **beta** y como argumento de salida **y**, $y = tg^{-1}\left(\frac{2}{\beta}\right) - e^{\beta}$. El contenido del fichero **ecuacion2.m** es el siguiente:

```
function y=ecuacion2(beta)

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

Ejecución de fzero: La función **fzero** busca el cero en funciones no lineales de una variable. El primer argumento contiene la ecuación a resolver. El segundo argumento es la apuesta inicial de donde está la solución (puede ser un escalar o un intervalo). Si la apuesta está muy alejada de la solución **fzero** puede no dar ninguna solución.

```
>> format long
>> beta=fzero(@ecuacion2,1)
beta =
   0.33865534225523

>> beta=fzero(@ecuacion2,10)
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
```

Ejecución de fsolve: La función **fsolve** permite resolver sistemas de ecuaciones no lineales con diversas variables. La sintaxis es la misma que en **fzero**,

```
>> beta=fsolve(@ecuacion2,1)
Optimization terminated: first-order optimality is less than
options.TolFun.
beta =
   0.33865534237264

>> beta=fsolve(@ecuacion2,10)
Optimization terminated: first-order optimality is less than
options.TolFun.
beta =
   0.33865534667084
```

Notar que el resultado es ligeramente distinto puesto que el algoritmo subyacente también lo es (**fsolve** pertenece a la *Optimization Toolbox* mientras que **fzero** pertenece al *kernel* de MATLAB)

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

Más opciones: Es posible utilizar el tercer argumento de entrada (opcional) para por ejemplo, mostrar las iteraciones y métodos utilizados. Teclear **>>help optimset** a fin de ver más opciones.

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

```
>> beta=fzero(@ecuacion2,[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(@ecuacion2,0.3,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
```

Funciones anónimas: Si la ecuación a resolver es sencilla o simplemente no queremos tener que crear una función cada vez que queremos resolver una ecuación, lo que podemos hacer es definir una función anónima. Ello se hace escribiendo lo que hace la función en el propio *function handle*, por ejemplo,

```
>> 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
```

La sintaxis de las funciones anónimas es `@(argumentos)(expresión)`

También se pueden entrar parámetros:

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

4.2 Simulación numérica de ecuaciones diferenciales

Para resolver ecuaciones diferenciales ordinarias (EDO) las funciones más importantes son:

ode23: para EDOs de orden 2 o 3
ode45: para EDOs de orden 4 o 5

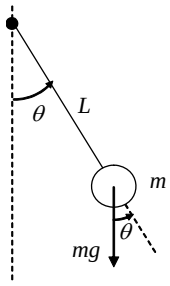
La sintaxis es:

```
>> [t,y]=ode23(@nom_func,[Tinicial Tfinal],cond_inic);
```

El fichero **nom_func.m** debe contener la ecuación diferencial de orden *n* a resolver descompuesta como sistema de *n* ecuaciones diferenciales de primer orden. Veámoslo con un ejemplo:

Ejemplo 10. Solución de ecuaciones diferenciales

Dinámica del péndulo simple: La ecuación diferencial que describe el comportamiento de un péndulo es $\ddot{\theta} + \alpha\dot{\theta} + \gamma \sin\theta - \beta = 0$.



Obtención de las ecuaciones de estado: En primer lugar hay que traducir la ecuación diferencial (de orden *n*) a (*n*) ecuaciones de estado (que son ecuaciones diferenciales de primer orden).

Se hace de la siguiente forma:

Ecuación original: $\ddot{\theta} + \alpha\dot{\theta} + \gamma \sin\theta - \beta = 0$

A θ la llamaremos x_1: $\ddot{x}_1 + \alpha\dot{x}_1 + \gamma \sin x_1 - \beta = 0$

A $\dot{x}_1$ la llamaremos x_2 (notar que $\dot{x}_1 = x_2 = \dot{\theta} = \omega$): $\dot{x}_2 + \alpha x_2 + \gamma \sin x_1 - \beta = 0$

Así, el sistema de ecuaciones de estado queda como:

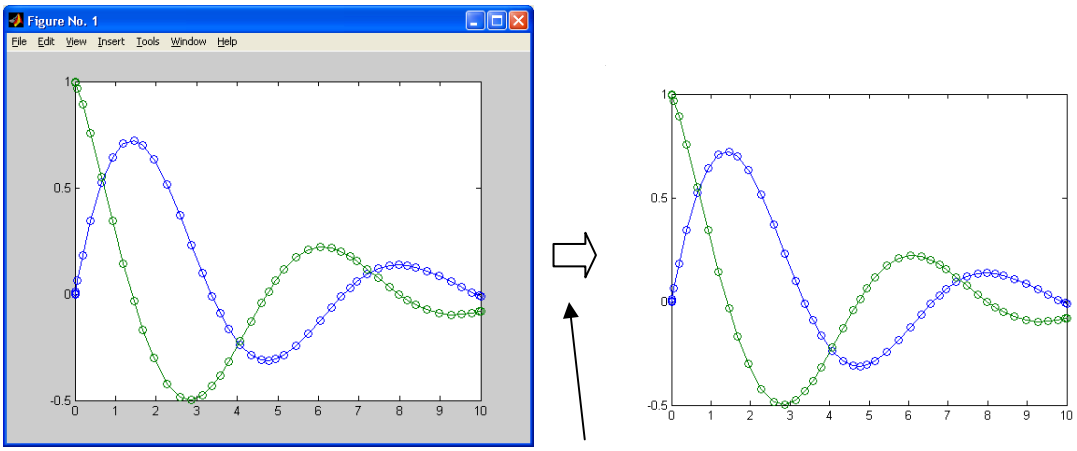
$$\left.\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\gamma \sin x_1 - \alpha x_2 + \beta \end{aligned}\right\}$$

Creación del fichero M: A continuación hay que editar una función con dicha descripción:

```
function     xpunto=pendulo(t,x)

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

Ejecución de ode23: Finalmente, se ejecuta una llamada desde la ventana de comandos:

```
>> Tinicial=0;Tfinal=10;cond_inic=[0;1];
>> ode23(@pendulo,[Tinicial Tfinal],cond_inic);
>>
```



Edit → Copy Figure (y, desde Word, Paste)

4.3 Optimización

MATLAB dispone de diversas funciones para optimización.

Programación lineal: La función es `linprog`. La programación lineal se usa en problemas donde la función objetivo y las restricciones son lineales,

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

Ejemplo 11. Programación lineal

Suponer que se fabrican dos productos A y B a lo largo de dos cadenas de producción [mag,05]. Cada cadena dispone de 200 horas. El producto A da un beneficio de 4€ por unidad y necesita 1h de la primera cadena y 1.25h de la segunda cadena. El producto B da un beneficio de 5€ por unidad y necesita 1h de la primera cadena y 0.75h en la segunda. Hay una demanda máxima potencial en el mercado de 150 unidades de B. Se quiere saber cuántas unidades de A y B maximizan el beneficio del fabricante. Definiendo x_1 y x_2 como las unidades de los productos A y B respectivamente, la función objetivo y las restricciones quedan como:

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

sujeto a:
$g_1: \; x_1 + x_2 \le 200$
$g_2: \; 1.25x_1 + 0.75x_2 \le 200$
$g_3: \; x_2 \le 150$
$(x_1, x_2) \ge 0$

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

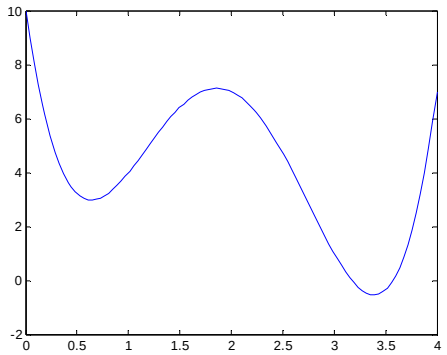
$$\begin{bmatrix} 1 & 1 \\ 1.25 & 0.75 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 200 \\ 200 \\ 150 \end{pmatrix}$$

```
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
```

Programación no lineal: Son problemas de optimización donde el criterio y/o las restricciones son no lineales. En el caso de optimización sin restricciones, $\min_{\mathbf{x}} f(\mathbf{x})$, las funciones son `fminunc` y `fminsearch`. La primera usa técnicas derivativas.

Ejemplo: Considerar la función $f(x) = 1.625x^4 - 12.75x^3 + 31.375x^2 - 26.25x + 10$. Para representarla, puesto que es un polinomio, se puede usar la función `polyval`.

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



Función conteniendo la curva de la que se quieren obtener los mínimos locales

```
function y=curva2(x)

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

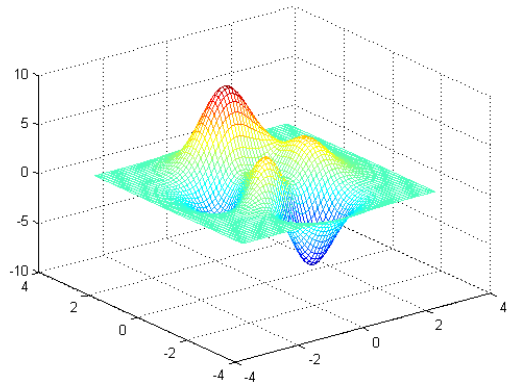
Uso de `fminsearch`:

```
>> fminsearch(@curva2,0.5)
ans =
    0.6425

>> fminsearch(@curva2,3)
ans =
    3.3853
```

Obtención de máximos locales: Basta con cambiar de signo la curva y volver a ejecutar `fminsearch`.

Optimización en funciones de 2 variables: La función sigue siendo `fminsearch`. Por ejemplo:

```
function z=superficie(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(@superficie,[0 0])

ans =

    0.2964    0.3202

>>
```

4.4 Temporizadores

MATLAB permite programar temporizaciones. Éstas combinan variables tipo *struct* con llamadas a funciones.

Ejemplo con temporizadores: Suponer que se edita un fichero `prova.m` con las siguientes instrucciones:

```
function prova

%Es crea un objecte "timer"
t=timer('StartDelay',2,'Period',1,'TasksToExecute',2,'Executi
onMode','fixedRate');

t.StartFcn = {@suma,'Inici:  ',1,2};
%Inicialització:  per començar sumem 1+2
t.StopFcn = {@suma, 'Final:  ',3,4};
%Terminació:  per acabar sumem 3+4
t.TimerFcn = { @suma, 'executant...',6,7};
%com a execució normal farem la suma de 6 i 7

%S'inicia el timer
start(t)


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

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

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

Su ejecución da el siguiente resultado:

```
resultat =
     3

Inici:   StartFcn  15-Mar-2010 18:03:24
resultat =
    13

executant... TimerFcn  15-Mar-2010 18:03:26
resultat =
    13

executant... TimerFcn  15-Mar-2010 18:03:27
resultat =
     7
Final:   StopFcn  15-Mar-2010 18:03:28
>>
```

5. Otras operaciones

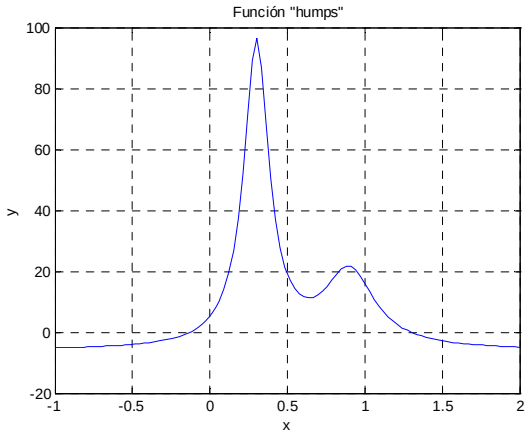
5.1 Integración y derivación

En este apartado se usará la función *humps* que ya está predefinida en MATLAB. El fichero M llamado **`humps.m`** contiene la función:

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

Para representarla, teclear las siguientes instrucciones:

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

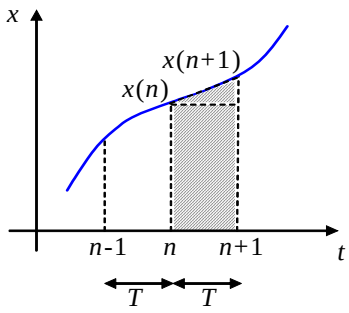


El valor de la integral de la función *humps* entre -1 y 2 es I=26.34496047137833.

Integrar una función es equivalente a calcular el área bajo dicha función. MATLAB dispone de comandos que aproximan numéricamente la integral de funciones.

Las más importantes son: **`trapz`**, **`cumtrapz`**, **`quad`**, **`quadl`**, **`dblquad`**, y **`triplequad`**.

Integración numérica por medio de áreas trapezoidales de igual base (trapz): Es posible integrar toda el área $I = \int_{x_1}^{x_2} f(x)dx$ por medio de trapecios de igual base con ayuda de **`trapz`** (regla trapezoidal o de Simpson). La precisión del resultado será mejor cuanto más pequeñas sean las bases de dichos trapecios:



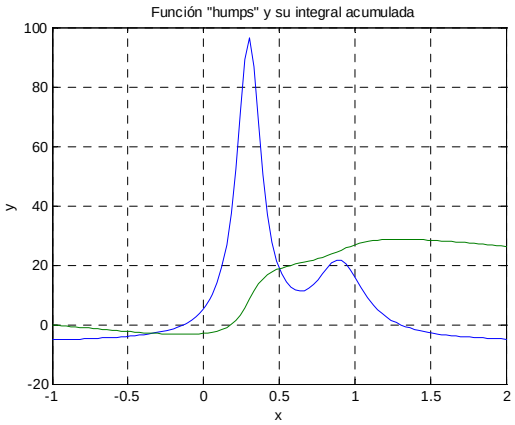
El área sombreada es $\frac{x(n)+x(n+1)}{2}T$.

```
>> 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
```

Integral acumulada (cumtrapz): Si se quiere evaluar la integral como función de x, $I(x)=\int_{x_1}^{x} f(x)dx$, se puede utilizar **`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')
```



Integración numérica por quadratura (quad y quadl): Estas funciones modifican el ancho de los trapecios individuales según la forma de la función con lo que se obtienen resultados mejores que con **`trapz`**. La función **`quadl`** (la l viene de adaptive Lobato) es ligeramente mejor que **`quad`**. Ambas funciones necesitan una función con la descripción de la curva:

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

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

Derivada e integral de un polinomio: Aunque la derivada de un polinomio es fácil de obtener, para los más perezosos MATLAB dispone de la función **`polyder`**:

Ejemplo: Dado $p(x) = x^3 + 2x^2 + 3x + 4$, su derivada es $dp(x)/dx = 3x^2 + 4x + 3$

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

También existe la función **`polyint`** para la integración de polinomios, $\int (3x^2 + 4x + 3)dx = 3\frac{x^3}{3} + 4\frac{x^2}{2} + 3x + cte$. El segundo argumento de entrada de `polyint` corresponde a la constante de integración.

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

Derivada numérica: La función es **`diff`** y calcula la diferencia progresiva entre muestras contiguas, es decir, realiza la siguiente aproximación:

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

Se recomienda usar esta función sólo en los casos en que la curva es relativamente suave. Si no es así, como por ejemplo en el caso de datos discretos, se recomienda antes de derivar ajustar los puntos por una curva suave.

Integración y derivación simbólicas: Es posible integrar y derivar en simbólico, gracias a las funciones de la *symbolic toolbox*. Para más detalles se recomienda teclear

```
>> help symbolic.
```

En cualquier caso, antes de realizar cualquier operación simbólica hay que declarar las variables implicadas como objetos simbólicos. Por ejemplo:

```
>> syms x n
```

Integración simbólica: La función es **int**. Por ejemplo, la integral de x^n es $\int x^n dx = \frac{x^{n+1}}{n+1}$. Las instrucciones son:

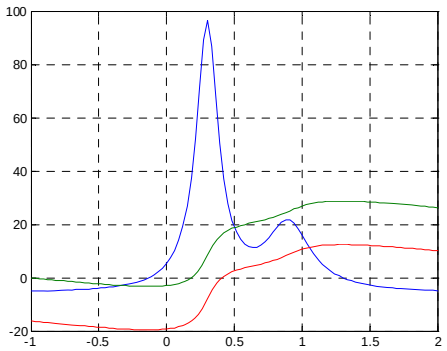
```
>> syms x n
>> f=x^n;
>> int(x^n)
ans =
x^(n+1)/(n+1)
```

Otro ejemplo: Obtener la integral en función de x de la función humps,

```
>> 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
```

y representarla:

```
>> 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 definida dentro de un intervalo: Es posible obtener el valor numérico de una integral definida en un intervalo. Siguiendo con el ejemplo de humps,

```
>> 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
```

Derivada simbólica: La función es **`diff`** pero aplicada sobre objetos definidos previamente como simbólicos. Por ejemplo:

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

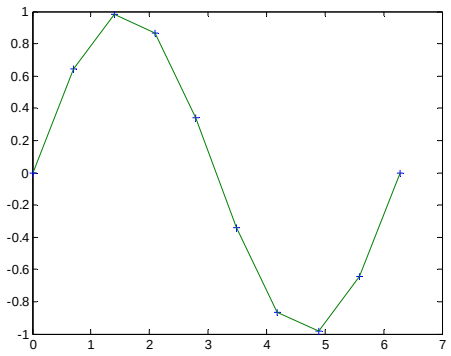
5.2 Interpolación y regresión. Ajuste polinomial de curvas

Interpolar es estimar valores intermedios a un conjunto de datos conocidos. Se trata de una operación importante en análisis de datos y ajuste de curvas.

Interpolación en una dimensión (interp1): Usa técnicas polinomiales: ajusta un polinomio entre cada par de puntos y estima su valor en el punto de interpolación deseado.

Ejemplo: Suponer que tenemos unos pocos puntos de una curva sinusoidal:

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



Y queremos saber el valor de la curva en 0.5. Para interpolar dicho valor se puede usar la función **`interp1`**. Esta función implementa diversos métodos de interpolación:

```
>> 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
```

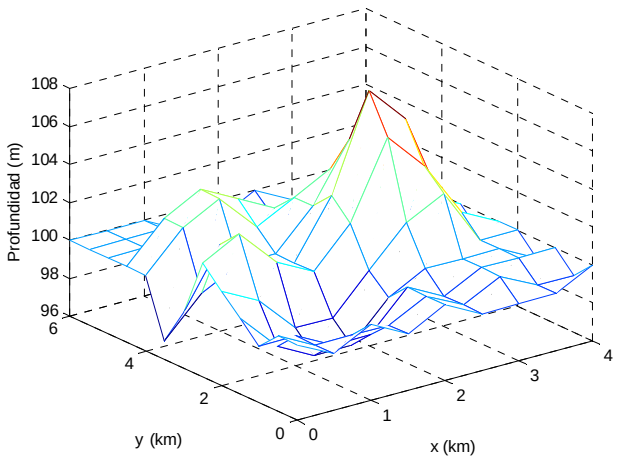
El valor exacto es sin(0.5)=0.4794. Por tanto, la opción ‘spline’ es la que da mejor resultado en nuestro ejemplo.

Interpolación en dos dimensiones (interp2): Se usa igual que **`interp1`** pero ahora interpolamos puntos en un plano.

Ejemplo: El script **`sonda.m`** contiene datos sobre la profundidad de un sector del lecho marino

```
%sonda.m

%medidas de profundidad marina

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)')
```



Si queremos interpolar la profundidad en el punto (2.2, 3.3), podemos hacer:

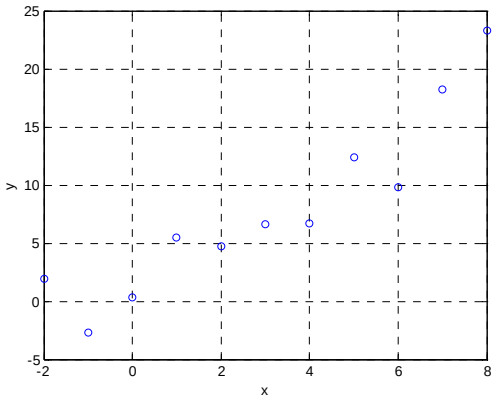
```
>> interp2(x,y,z,2.2,3.3,'cubic')
ans =
  104.1861
```

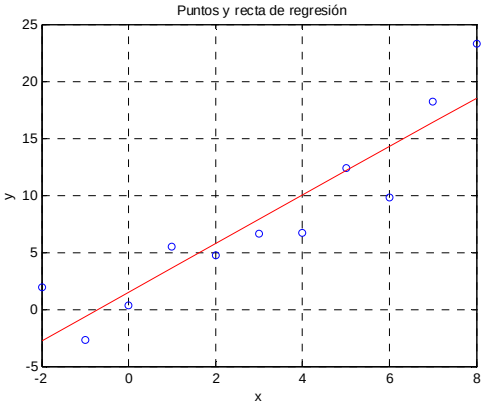
Recta de regresión: Suponer que se tienen diversos pares de puntos (x, y) y se quiere encontrar la recta que "más" se ajusta a todos ellos.

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

Para representarlos basta con editar un *script* con las instrucciones:

```
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')
```





Encontrar la recta $y_r = ax + b$ es equivalente a especificar su pendiente *a* y su ordenada en origen *b*. Puesto que no existe una recta que pase por todos ellos, se busca una recta óptima en el sentido que minimiza la suma de los errores cuadráticos en los puntos conocidos.

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

Para representar la recta $y_r = 2.1317x + 1.4985$, se puede usar la función polyval,

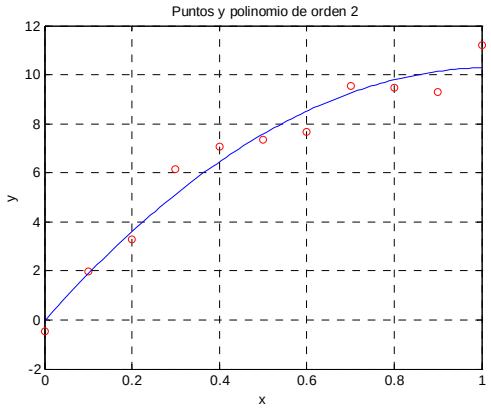
```
yr=polyval(coefs,x);
hold on,plot(x,yr,'r'),title('Puntos y recta de regresión'),
hold off
```

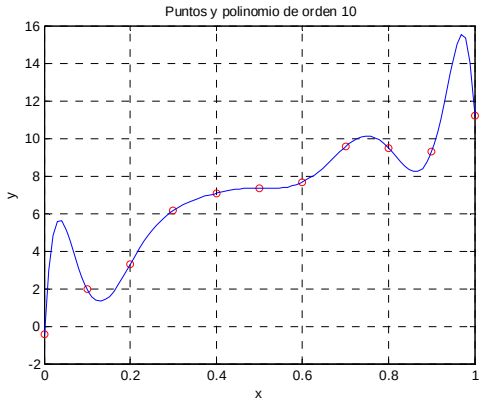
También podríamos haber hecho **`yr=coefs(1)*x+coefs(2)`**.

Ajuste con polinomios de orden superior: Para el caso de polinomios de orden superior, el procedimiento es el mismo:

```
%regresión cuadrática
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')
```

```
%curva que pasa exactamente por los puntos
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
```

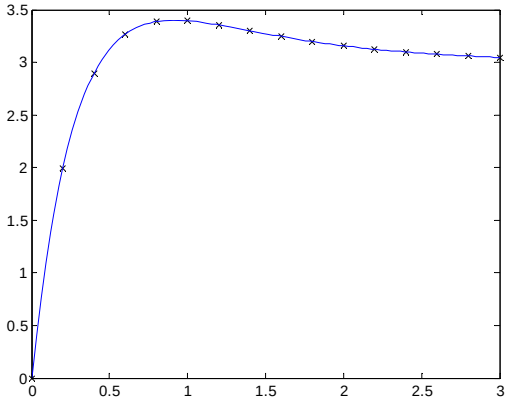
```
>> coefs2
coefs2 =
  1.0e+006 *
  Columns 1 through 7
   -0.4644    2.2965   -4.8773    5.8233   -4.2948    2.0211
-0.6032
  Columns 8 through 11
    0.1090   -0.0106    0.0004   -0.0000
```

Pseudoinversa. Ajustes de curvas lineales en los parámetros (función pinv):

La matriz pseudoinversa puede usarse para estimar expresiones lineales en los parámetros, por ejemplo, $y(x) = a_0 + a_1 e^{-\frac{x}{0.3}} + a_2 e^{-\frac{x}{0.8}}$; donde los valores desconocidos son $(a_0, a_1, a_2) = (3,-5,2)$.

```
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
```



5.3 Procesado de audio

MATLAB puede importar ficheros de sonido (función **`wavread`**) y puede acceder a la tarjeta de sonido a fin de escucharlos (función **`sound`**).

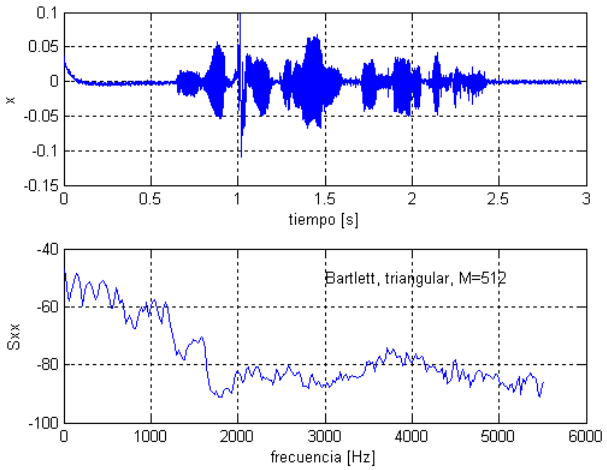
Ejemplo 12. Obtención de la frecuencia fundamental (pitch) de una voz

En un fichero **`*.wav`** se ha grabado la frase "el golpe de timón fue sobrecogedor" con las siguientes opciones: frecuencia de muestreo 11025Hz, mono y 16bits.

A continuación se representa la señal, se estima su espectro por el método de Barlett (con ventana triangular de M=512 muestras) y se busca la frecuencia fundamental (que resulta ser de 140Hz, una voz masculina)

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

%truncado de la secuencia
pot=fix(log2(length(y)));
N=2^pot;
x=y(1:N);
sound(x,fs,bits)

%estimación espectral
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;

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

%representación frecuencial
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')
```



Apéndice. Estimación de densidades espectrales. Métodos

Tenemos una secuencia de muestras $x[n]$, $n = 0,\ldots, N\text{-}1$, en el dominio temporal y queremos estimar [el módulo de] su densidad espectral (o sea su contenido frecuencial).

Fast Fourier Transform

Sin entrar en detalles y abusando un poco del lenguaje, para pasar del dominio temporal al dominio frecuencial lo que se hace es aplicar la transformada de Fourier a las muestras temporales. Si hay muchas muestras puede ser espantoso. Por suerte existe un algoritmo rápido llamado FFT (*Fast Fourier Transform*) que lo hace de forma rápida y está diseñado para ser aplicado cuando el número de muestras de la secuencia temporal es potencia de 2. MATLAB implementa la FFT.

Por ello, antes de aplicar la FFT se pueden hacer dos cosas a la secuencia original:

- bien se trunca (y nos quedamos con las primeras 2^k muestras)
- bien se rellena la secuencia original con los ceros necesarios para que N sea un número potencia de 2 (*zero padding*).

En los apuntes se ha hecho un truncado.

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

%truncado de la secuencia
pot=fix(log2(length(y)));
N=2^pot;
x=y(1:N);
sound(x,fs,bits)
```

Métodos

A partir de ahí, además, tenemos varias opciones para obtener espectros. Podemos coger trozos de la secuencia original, enventanándolas o no, obtener la FFT de cada uno de ellos y promediar al final; poner o no solapamiento en las ventanas, etc. Todas estas estrategias buscan mejoras en la estimación (en la carga de los cálculos, en la capacidad de discriminar picos de frecuencia, en el factor de calidad de la estimación, en el suavizado de la estimación,…). Los principales métodos son los siguientes:

Método del periodograma: Se aplica directamente la fórmula

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

donde la expresión obtenida para $\hat{S}_x(f)$ se denomina periodograma.

Método de Bartlett o promedio de periodogramas: Consiste en dividir la secuencia $x[n]$ de N puntos en K segmentos de M puntos cada segmento y promediar los K periodogramas resultantes:

- Segmento k: $x_k[m] = x[m + kM]$, $k = 0,..., K-1$, $m = 0,..., M-1$
- Periodograma del segmento k: $\hat{S}_x^k(f) = \frac{1}{M}\left|\sum_{m=0}^{M-1} x_k[m] e^{-j2\pi fm}\right|^2$, $k = 0,..., K-1$
- Promedio de los periodogramas de los K segmentos: $\hat{S}_x(f) = \frac{1}{K}\sum_{k=0}^{K-1} \hat{S}_x^k(f)$

Método de Welch: Este método, también conocido como promedio de periodogramas modificados, es una extensión del método de Bartlett y se diferencia de éste en dos aspectos: se introduce un solapamiento D entre segmentos y además éstos se enventanan con $w[n]$. Así pues, el procedimiento es el siguiente:

Se divide la secuencia $x[n]$ de N puntos en K segmentos de M puntos cada segmento pero introduciendo un solapamiento D entre segmentos:

- Segmento k: $x_k[m] = x[m + kM - D]$, $k = 0,..., K-1$, $m = 0,..., M-1$ (si $D = 0.5M$ el solapamiento es del 50%, si $D = 0.1M$ es del 10%)
- Los K segmentos se enventanan con $w[n]$ y se obtienen los periodogramas. Se normalizan los periodogramas para tener en cuenta el efecto de la ventana:
- Periodograma del segmento k: $\hat{S}_x^k(f) = \frac{1}{MU}\left|\sum_{m=0}^{M-1} x_k[m] w[m] e^{-j2\pi fm}\right|^2$, $k = 0,..., K-1$, siendo $U = \frac{1}{M}\sum_{m=0}^{M-1} w^2[m]$.
- Finalmente se promedian los K periodogramas resultantes: Promedio de los periodogramas de los K segmentos: $\hat{S}_x(f) = \frac{1}{K}\sum_{k=0}^{K-1} \hat{S}_x^k(f)$

Método de Blackman-Tukey o alisado de periodogramas: A partir de las N muestras de $x[n]$ se obtiene la autocorrelación $R_{xx}[k]$. Las muestras de $R_{xx}[k]$ más alejadas del origen se consideran despreciables puesto que son producto de una estimación con pocas muestras de $x[n]$. Por ello, se enventana la autocorrelación con una ventana par de M muestras y centrada en el origen. Para obtener la densidad espectral basta con aplicar la FFT a la correlación enventanada $R_{xx,v}[m]$.