Apuntes de MATLAB
orientados a métodos numéricos elementales

Rosa Echevarría
Dpto. de Ecuaciones Diferenciales y Análisis Numérico
Universidad de Sevilla

15 de diciembre de 2017

Índice

1 Introducción y elementos básicos

MATLAB es un potente paquete de software para computación científica, orientado al cálculo numérico, a las operaciones matriciales y especialmente a las aplicaciones científicas y de ingeniería. Ofrece lo que se llama un entorno de desarrollo integrado (IDE), es decir, una herramienta que permite, en una sóla aplicación, ejecutar órdenes sencillas, escribir programas utilizando un editor integrado, compilarlos (o interpretarlos), depurarlos (buscar errores) y realizar gráficas.

Puede ser utilizado como simple calculadora matricial, pero su interés principal radica en los cientos de funciones tanto de propósito general como especializadas que posee, así como en sus posibilidades para la visualización gráfica.

MATLAB posee un lenguaje de programación propio, muy próximo a los habituales en cálculo numérico (Fortran, C, . . .), aunque mucho más *tolerante* en su sintaxis, que permite al usuario escribir sus propios *scripts* (conjunto de comandos escritos en un fichero, que se pueden ejecutar con una única orden) para resolver un problema concreto y también escribir nuevas funciones con, por ejemplo, sus propios algoritmos, o para *modularizar* la resolución de un problema complejo. MATLAB dispone, además, de numerosas *Toolboxes*, que le añaden funcionalidades especializadas.

Numerosas contribuciones de sus miles de usuarios en todo el mundo pueden encontrarse en la web de The MathWorks: http://www.mathworks.es

1.1 Comenzando

Al iniciar MATLAB nos aparecerá una ventana más o menos como la de la Figura 1.1 (dependiendo del sistema operativo y de la versión)

Si la ubicación de las ventanas integradas es diferente, se puede volver a ésta mediante:

Menú Desktop → Desktop Layout → Default

Se puede experimentar con otras disposiciones. Si hay alguna que nos gusta, se puede salvaguardar con

Menú Desktop → Desktop Layout → Save Layout . . .

dándole un nombre, para usarla en otras ocasiones. De momento, sólo vamos a utilizar la ventana principal de MATLAB: Command Window (ventana de comandos). A través de esta ventana nos comunicaremos con MATLAB, escribiendo las órdenes en la línea de comandos. El símbolo >> al comienzo de una línea de la ventana de comandos se denomina *prompt* e indica que MATLAB está desocupado, disponible para ejecutar nuestras órdenes.

1.2 Objetos y sintaxis básicos

Las explicaciones sobre las funciones/comandos que se presentan en estas notas están muy resumidas y sólo incluyen las funcionalidades que, según el parecer subjetivo de la autora, pueden despertar más

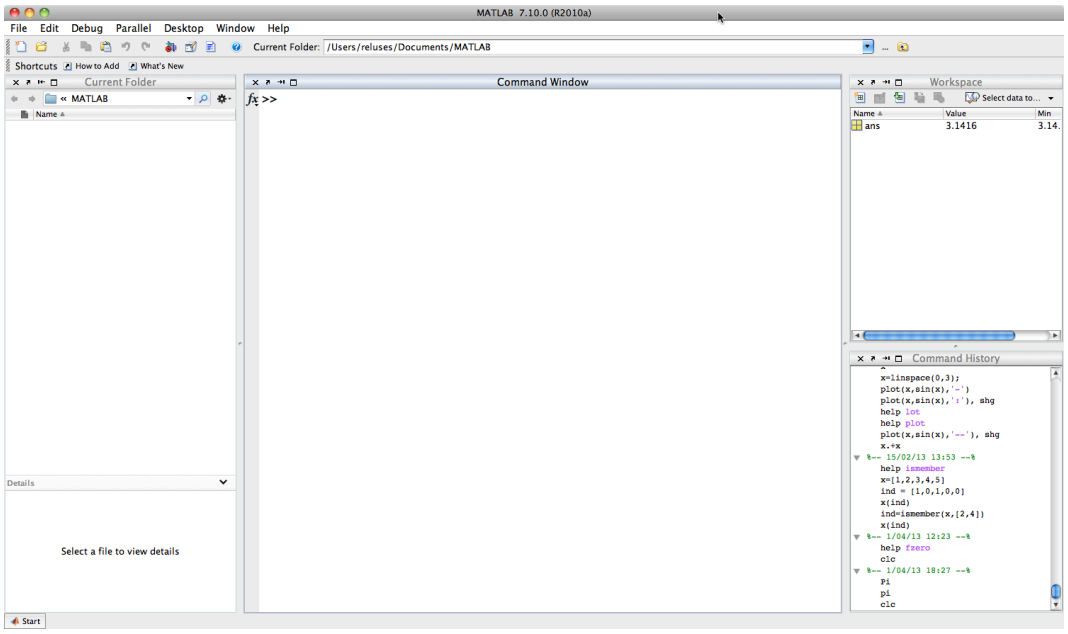


Figura 1.1: La ventana de MATLAB

interés. La mayoría de las funciones tienen mas y/o distintas funcionalidades que las que se exponen aquí. Para una descripción exacta y exhaustiva es preciso consultar la Ayuda on-line.

Los tipos básicos de datos que maneja MATLAB son números reales, booleanos (valores lógicos) y cadenas de caracteres (string). También puede manipular distintos tipos de números enteros, aunque sólo suele ser necesario en circunstancias específicas.

En MATLAB, por defecto, los números son codificados como números reales en coma flotante en doble precisión. La precisión, esto es, el número de bits dedicados a representar la mantisa y el exponente, depende de cada (tipo de) máquina.

MATLAB manipula también otros objetos, compuestos a partir de los anteriores: números complejos, matrices, *cells*, estructuras definidas por el usuario, clases Java, etc.

El objeto básico de trabajo de MATLAB es una matriz bidimensional cuyos elementos son números reales o complejos. Escalares y vectores son considerados casos particulares de matrices. También se pueden manipular matrices de cadenas de caracteres, booleanas y enteras.

El lenguaje de MATLAB es **interpretado**, esto es, las instrucciones se traducen a lenguaje máquina una a una y se ejecutan antes de pasar a la siguiente. Es posible escribir varias instrucciones en la misma línea, separándolas por una coma o por punto y coma. Las intrucciones que terminan por punto y coma no producen salida de resultados por pantalla.

Algunas constantes numéricas están predefinidas (ver Tabla 1.1)

MATLAB distingue entre mayúsculas y minúsculas: `pi` no es los mismo que `Pi`.

MATLAB conserva un historial de las instrucciones escritas en la línea de comandos. Se pueden recuperar instrucciones anteriores, usando las teclas de flechas arriba y abajo. Con las flechas izquierda y derecha nos podemos desplazar sobre la línea de comando y modificarlo.

Se pueden salvaguardar todas las instrucciones y la salida de resultados de una sesión de trabajo de MATLAB a un fichero:

```
>> diary nombre_fichero
>> diary off                    % suspende la salvaguarda
```

`i, j`	Unidad imaginaria : `2+3i`, `-1-2j`
`pi`	Número π
`Inf`	*Infinito*, número mayor que el más grande que se puede almacenar. Se produce con operaciones como $x/0$, con $x \neq 0$
`NaN`	*Not a Number*, magnitud no numérica resultado de cálculos indefinidos. Se produce con cálculos del tipo $0/0$ o ∞/∞. `(0+2i)/0` da como resultado `NaN + Inf i`
`eps`, `intmax`, `intmin` `realmax`, `realmin`	Otras constantes. Consultar la ayuda on-line.

Tabla 1.1: Algunas constantes pre-definidas en MATLAB. Sus nombres son reservados y no deberían ser usados para variables ordinarias.

1.2.1 Constantes y operadores

Números reales	`8.01  -5.2  .056  1.4e+5  0.23E-2` `-.567d-21  8.003D-12`
Números complejos	`1+2i  -pi-3j`
Booleanos	`true  false`
Caracteres	Entre apóstrofes: `'esto es una cadena de caracteres (string)'`
Operadores aritméticos	`+  -  *  /  ^`
Operadores de comparación	`==  ~= (ó <>)  <  >  <=  >=`
Operadores lógicos (lo dos últimos sólo para escalares)	`&  |  ~  &&  ||` `&&` y `||` no evalúan el operando de la derecha si no es necesario.)

Tabla 1.2: Constantes y operadores de diversos tipos.

1.2.2 Funciones elementales

Los nombres de las funciones elementales son los *habituales*.

Los argumentos pueden ser, siempre que tenga sentido, reales o complejos y el resultado se devuelve en el mismo tipo del argumento.

La lista de todas las funciones matemáticas elementales se puede consultar en:

Help → MATLAB → Functions: By Category → Mathematics → Elementary Math

Algunas de las más habituales se muestran en la Tabla 1.3:

1.2.3 Uso como calculadora

Se puede utilizar MATLAB como simple calculadora, escribiendo expresiones aritméticas y terminando por RETURN (`<R>`). Se obtiene el resultado inmediatamente a través de la variable del sistema `ans` (de *answer*). Si no se desea que MATLAB escriba el resultado en el terminal, debe terminarse la orden por punto y coma (útil, sobre todo en programación).

`sqrt(x)`	raiz cuadrada	`sin(x)`	seno (radianes)
`abs(x)`	módulo	`cos(x)`	coseno (radianes)
`conj(z)`	complejo conjugado	`tan(z)`	tangente (radianes)
`real(z)`	parte real	`cotg(x)`	cotangente (radianes)
`imag(z)`	parte imaginaria	`asin(x)`	arcoseno
`exp(x)`	exponencial	`acos(x)`	arcocoseno
`log(x)`	logaritmo natural	`atan(x)`	arcotangente
`log10(x)`	logaritmo decimal	`cosh(x)`	cos. hiperbólico
`rat(x)`	aprox. racional	`sinh(x)`	seno hiperbólico
`mod(x,y)` `rem(x,y)`	resto de dividir x por y Iguales si $x, y > 0$. Ver help para definición exacta	`tanh(x)`	tangente hiperbólica
`fix(x)`	Redondeo hacia 0	`acosh(x)`	arcocoseno hiperb.
`ceil(x)`	Redondeo hacia $+\infty$	`asinh(x)`	arcoseno hiperb.
`floor(x)`	Redondeo hacia $-\infty$	`atanh(x)`	arcotangente hiperb.
`round(x)`	Redondeo al entero más próximo		

Tabla 1.3: Algunas funciones matemáticas elementales.

Ejemplos 1.1

```
>> sqrt(34*exp(2))/(cos(23.7)+12)
   ans  =
        1.3058717

>> 7*exp(5/4)+3.54
   ans  =
        27.97240

>> exp(1+3i)
   ans  =
        - 2.6910786 + 0.3836040i
```

1.2.4 Variables

En MATLAB las variables no son nunca declaradas: su tipo y su tamaño cambian de forma dinámica de acuerdo con los valores que le son asignados. Así, una misma variable puede ser utilizada, por ejemplo, para almacenar un número complejo, a continuación una matriz 25×40 de números enteros y luego para almacenar un texto. Las variables se crean automáticamente al asignarles un contenido. Asimismo, es posible eliminar una variable de la memoria si ya no se utiliza.

Ejemplos 1.2

```
>> a=10
   a =
        10.
>> pepito=exp(2.4/3)
   pepito  =
        2.2255
>> pepito=a+pepito*(4-0.5i)
   pepito  =
        18.9022 - 1.1128i
>> clear pepito
```

Para conocer en cualquier instante el valor almacenado en una variable basta con teclear su nombre (Atención: recuérdese que las variables `AB`, `ab`, `Ab` y `aB` **son distintas**, ya que MATLAB distingue entre mayúsculas y minúsculas).

Otra posibilidad es hojear el Workspace ó espacio de trabajo, abriendo la ventana correspondiente. Ello nos permite ver el contenido de todas las variables existentes en cada momento e, incluso, modificar su valor.

Algunos comandos relacionados con la inspección y eliminación de variables se describen en la tabla siguiente:

`who`	lista las variables actuales
`whos`	como el anterior, pero más detallado
`clear`	elimina todas las variables que existan en ese momento
`clear a b c`	elimina las variables `a`, `b` y `c` (atención: sin comas!)

1.2.5 Formatos

Por defecto, MATLAB muestra los números en formato de punto fijo con 5 dígitos. Se puede modificar este comportamiento mediante el comando `format`

`format`	Cambia el formato de salida a su valor por defecto, short
`format short`	El formato por defecto
`format long`	Muestra 15 dígitos
`format short e`	Formato short, en coma flotante
`format long e`	Formato long, en coma flotante
`format rat`	Muestra los números como cociente de enteros

1.2.6 Algunos comandos utilitarios del sistema operativo

Están disponibles algunos comandos utilitarios, como

`ls` `dir`	Lista de ficheros del directorio de trabajo
`pwd`	Devuelve el nombre y la ruta (*path*) del directorio de trabajo
`cd`	Para cambiar de directorio
`clc`	Limpia la ventana de comandos Command Window
`date`	Fecha actual

Tabla 1.4: Comandos utilitarios.

1.3 Documentación y ayuda *on-line*

- Ayuda on-line en la ventana de comandos

```
>> help nombre_de_comando
```

  La información se obtiene en la misma ventana de comandos. Atención:

- Ayuda on-line con ventana de navegador

```
>> helpwin
```

  ó bien Menú Help ó bien Botón Start → Help.

Además, a través del navegador del Help se pueden descargar, desde The MathWorks, guías detalladas, en formato pdf, de cada capítulo.

1.4 *Scripts* y funciones. El editor integrado

1.4.1 *Scripts*

En términos generales, en informática, un *script* (guión o archivo por lotes) es un conjunto de instrucciones (programa), usualmente simple, guardadas en un fichero (usualmente de texto plano) que son ejecutadas normalmente mediante un intérprete. Son útiles para automatizar pequeñas tareas. También puede hacer las veces de un "programa principal" para ejecutar una aplicación.

Así, para llevar a cabo una tarea, en vez de escribir las instrucciones una por una en la línea de comandos de MATLAB, se pueden escribir las órdenes una detrás de otra en un fichero. Para ello se puede utilizar el Editor integrado de MATLAB. Para iniciarlo, basta pulsar el icono *hoja en blanco* (New script) de la barra de MATLAB, o bien

File → New → Script

UN *script* de MATLAB debe guardarse en un fichero con sufijo `.m` para ser reconocido. El nombre del fichero puede ser cualquiera *razonable*, es decir, sin acentos, sin espacios en blanco y sin caracteres «extraños».

Para editar un *script* ya existente, basta hacer *doble-click* sobre su icono en la ventana Current Folder.

Para ejecutar un *script* que esté en el directorio de trabajo, basta escribir su nombre (sin el sufijo) en la línea de comandos.

1.4.2 M-Funciones

Una función (habitualmente denominadas M-funciones en MATLAB), es un programa con una «interfaz» de comunicación con el exterior mediante argumentos de entrada y de salida.

Las funciones MATLAB responden al siguiente formato de escritura:

```
function [argumentos de salida] = nombre(argumentos de entrada)
%
% comentarios
%
....
instrucciones (normalmente terminadas por  ;  para evitar eco en pantalla)
....
```

Las funciones deben guardarse en un fichero con el mismo nombre que la función y sufijo `.m`. Lo que se escribe en cualquier línea detrás de `%` es considerado como comentario.

Ejemplo 1.3
El siguiente código debe guardarse en un fichero de nombre `areaequi.m`.

x

```
function [sup] = areaequi(long)
%
% areaequi(long) devuelve el area del triangulo
%                equilatero de lado = long
%
sup = sqrt(3)*long^2/4;
```

La primera línea de una M-función siempre debe comenzar con la claúsula (palabra reservada) `function`. El fichero que contiene la función debe estar *en un sitio en el que MATLAB lo pueda encontrar*, normalmente, en la carpeta de trabajo.

Se puede ejecutar la M-función en la misma forma que cualquier otra función de MATLAB:

Ejemplos 1.4

```
>> areaequi(1.5)
ans =
    0.9743
>> rho = 4 * areaequi(2) +1;
>> sqrt(areaequi(rho))
ans =
    5.2171
```

Los breves comentarios que se incluyen a continuación de la línea que contiene la claúsula `function` deben explicar, brevemente, el funcionamiento y uso de la función. Además, constituyen la ayuda *on-line* de la función:

Ejemplo 1.5

```
>> help areaequi
  areaequi(long) devuelve el area del triangulo
                 equilatero de lado = long
```

Se pueden incluir en el mismo fichero otras funciones, denominadas subfunciones, a continuación de la primera[1], pero sólo serán *visibles* para las funciones del mismo fichero. **NO** se pueden incluir M-funciones en el fichero de un *script*.

1.4.3 Funciones anónimas

Algunas funciones *sencillas*, que devuelvan el resultado de una expresión, se pueden definir mediante una sóla instrucción, en mitad de un programa (script o función) o en la línea de comandos. Se llaman funciones anónimas. La sintaxis para definirlas es:

```
nombre_funcion = @(argumentos) expresion_funcion
```

Ejemplo 1.6 (Función anónima para calcular el área de un círculo)

```
>> area_circulo = @(r) pi * r.^2;
>> area_circulo(1)
ans =
    3.1416
>> semicirc = area_circulo(3)/2
semicirc =
    14.1372
```

Las funciones anónimas pueden tener varios argumentos y hacer uso de variables previamente definidas:

Ejemplo 1.7 (Función anónima de dos variables)

```
>> a = 2;
>> mifun = @(x,t) sin(a*x).*cos(t/a);
>> mifun(pi/4,1)
ans =
    0.8776
```

Si, con posterioridad a la definición de la función `mifun`, se cambia el valor de la variable `a`, la función no se modifica: en el caso del ejemplo, seguirá siendo `mifun(x,t)=sin(2*x).*cos(t/2)`.

[1] También es posible definir funciones anidadas, esto es, funciones «insertadas» dentro del código de otras funciones. (Se informa aquí para conocer su existencia. Su utilización es delicada.)

1.5 *Workspace* y ámbito de las variables

Workspace (espacio de trabajo) es el conjunto de variables que en un momento dado están definidas en la memoria del MATLAB.

Las variables creadas desde la línea de comandos de MATLAB pertenecen al Base Workspace (espacio de trabajo base; es el que se puede «hojear» en la ventana Workspace). Los mismo sucede con las variables creadas por un *script* que se ejecuta desde la línea de comandos. Estas variables permanecen en el Base Workspace cuando se termina la ejecución del script y se mantienen allí durante toda la sesión de trabajo o hasta que se borren.

Sin embargo, las variables creadas por una M-función pertenecen al espacio de trabajo de dicha función, que es independiente del espacio de trabajo base. Es decir, las variables de las M-funciones son **locales**: MATLAB reserva una zona de memoria cuando comienza a ejecutar una M-función, almacena en esa zona las variables que se crean dentro de ella, y «borra» dicha zona cuando termina la ejecución de la función.

Esta es una de las principales diferencias entre un *script* y una M-función: cuando finaliza la ejecución de un *script* se puede «ver» y utilizar el valor de todas las variables que ha creado el script en el Workspace; en cambio, cuando finaliza una función no hay rastro de sus variables en el Workspace.

Para hacer que una variable local de una funcion pertenezca al Base Workspace, hay que declararla `global`: la orden

```
global a suma error
```

en una función hace que las variables `a`, `suma` y `error` pertenezcan al Base Workspace y por lo tanto, seguirán estando disponibles cuando finalize la ejecución de la función.

1.6 Matrices

Como ya se ha dicho, las matrices bidimensionales de números reales o complejos son los objetos básicos con los que trabaja MATLAB. Los vectores y escalares son casos particulares de matrices.

1.6.1 Construcción de matrices

La forma más elemental de introducir matrices en MATLAB es describir sus elementos de forma exhaustiva (por filas y entre corchetes rectos `[ ]`) : elementos de una fila se separan unos de otros por comas y una fila de la siguiente por punto y coma.

Ejemplos 1.8 (Construcciones elementales de matrices)

```
>> v = [1, -1, 0, sin(2.88)]                    % vector fila longitud 4

>> w = [0; 1.003; 2; 3; 4; 5*pi]                % vector columna longitud 6

>> a = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]  % matriz 3x4
```

Observación. El hecho de que, al introducirlas, se escriban las matrices por filas no significa que internamente, en la memoria del ordenador, estén así organizadas: **en la memoria las matrices se almacenan como un vector unidimensional ordenadas por columnas**.

Ejemplos 1.9 (Otras órdenes para crear matrices)

```
>> v1 = a:h:b      % crea un vector fila de números regularmente espaciados
                   %  a  a+h  a+2h ... hasta  c <= b < c+h

>> v2 = a:b        % como el anterior, con paso h=1

>> v3 = v2'        % matriz traspuesta (conjugada si es compleja)

>> v4 = v2.'       % matriz traspuesta sin conjugar
```

Se pueden también utilizar los vectores/matrices como objetos para construir otras matrices (bloques):

Ejemplos 1.10 (Matrices construidas con bloques)

```
>> v1 = 1:4;

>> v2 = [v1, 5; 0.1:0.1:0.5]

>> v3 = [v2', [11,12,13,14,15]']
```

$$v_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \quad v_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 0.1 & 0.2 & 0.3 & 0.4 & 0.5 \end{pmatrix} \quad v_3 = \begin{pmatrix} 1 & 0.1 & 11 \\ 2 & 0.2 & 12 \\ 3 & 0.3 & 13 \\ 4 & 0.4 & 14 \\ 4 & 0.5 & 15 \end{pmatrix}$$

Las siguientes funciones generan vectores de elementos regularmente espaciados, útiles en muchas circunstancias, especialmente para creación de gráficas.

`linspace(a,b,n)`	Si `a` y `b` son números reales y `n` un número entero, genera una partición regular del intervalo `[a,b]` con `n` nodos (`n-1` subintervalos)
`linspace(a,b)`	Como el anterior, con `n`=100

Las siguientes funciones generan algunas matrices especiales que serán de utilidad.

`zeros(n,m)`	matriz `n`×`m` con todas sus componentes iguales a cero
`ones(n,m)`	matriz `n`×`m` con todas sus componentes iguales a uno
`eye(n,m)`	matriz unidad `n`×`m`: diagonal principal = 1 y el resto de las componentes = 0
`diag(v)`	Si `v` es un vector, es una matriz cuadrada de ceros con diagonal principal = `v`
`diag(A)`	Si `A` es una matriz, es su diagonal principal

MATLAB posee, además, decenas de funciones útiles para generar distintos tipos de matrices. Para ver una lista exhaustiva consultar:

Help → MATLAB → Functions: By Category → Mathematics → Arrays and Matrices

1.6.2 Acceso a los elementos de vectores y matrices

El acceso a elementos individuales de una matriz se hace indicando entre paréntesis sus índices de fila y columna. Se puede así utilizar o modificar su valor

Ejemplos 1.11 (Acceso a elementos individuales de una matriz)

```
>> A = [1, 2, 3; 6, 3, 1; -1, 0, 0; 2, 0, 4]
A =
     1     2     3
     6     3     1
    -1     0     0
     2     0     4
>> A(2, 3)
ans =
     1
>> A(4, 2) = 11
A =
     1     2     3
     6     3     1
    -1     0     0
     2    11     4
```

Se puede acceder al último elemento de un vector mediante el *índice simbólico* `end`:

Ejemplos 1.12 (Último índice de un vector)

```
>> w = [5, 3, 1, 7, 3, 0, -2];
>> w(end)
ans =
    -2
>> w(end-1) = 101
w =
     5     3     1     7     3   101    -2
```

Se puede acceder a una fila o columna completa sustituyendo el índice adecuado por `:` (dos puntos)

Ejemplos 1.13 (Acceso a filas o columnas completas)

```
>> A = [1, 2, 3; 6, 3, 1; -1, 0, 0; 2, 0, 4];
>> A(:, 3)
ans =
     3
     1
     0
     4
>> A(2, :)
ans =
     6     3     1
>> A(3, :) = [5, 5, 5]
A =
     1     2     3
     6     3     1
     5     5     5
     2     0     4
```

También se puede acceder a una "submatriz" especificando los valores de las filas y columnas que la definen:

Ejemplos 1.14 (Acceso a "submatrices")

```
>> w = [5, 3, 1, 7, 3, 0, -2];
>> w([1,2,3])            % o lo que es lo mismo w(1:3)
ans =
     5     3     1
>> A = [1, 2, 3; 6, 3, 1; -1, 0, 0; 2, 0, 4];
>> A(1:2, :) = zeros(2, 3)
A =
     0     0     0
     0     0     0
    -1     0     0
     2     0     4
>> B = A(3:4, [1,3])
B =
    -1     0
     2     4
```

1.6.3 Operaciones con vectores y matrices

Los operadores aritméticos representan las correspondientes operaciones matriciales siempre que tengan sentido.

Sean `A` y `B` dos matrices de elementos respectivos a_{ij} y b_{ij} y sea `k` un escalar.	
`A+B`, `A-B`	matrices de elementos respectivos $a_{ij}+b_{ij}$, $a_{ij}-b_{ij}$ (si las dimensiones son iguales)
`A+k`, `A-k`	matrices de elementos respectivos $a_{ij}+k$, $a_{ij}-k$.
`k*A`, `A/k`	matrices de elementos respectivos $k\,a_{ij}$, $\frac{1}{k}\,a_{ij}$
`A*B`	producto matricial de `A` y `B` (si las dimensiones son adecuadas)
`A^n`	Si `n` es un entero positivo, `A*A*...*A`

Además de estos operadores, MATLAB dispone de ciertos operadores aritméticos que operan **elemento a elemento**. Son los operadores `.*` `./` y `.^`, muy útiles para aprovechar las funcionalidades vectoriales de MATLAB.

Sean `A` y `B` dos matrices de las mismas dimensiones y de elementos respectivos a_{ij} y b_{ij} y sea `k` un escalar.	
`A.*B`	matriz de la misma dimensión que `A` y `B` de elementos $a_{ij}\times b_{ij}$
`A./B`	ídem de elementos $\frac{a_{ij}}{b_{ij}}$
`A.^B`	ídem de elementos ${a_{ij}}^{b_{ij}}$
`k./A`	matriz de la misma dimensión que `A`, de elementos $\frac{k}{a_{ij}}$
`A.^k`	ídem de elementos a_{ij}^{k}
`k.^A`	ídem de elementos $k^{a_{ij}}$

Por otra parte, la mayoría de las funciones MATLAB están hechas de forma que admiten matrices como argumentos. Esto se aplica en particular a las funciones matemáticas elementales y su utilización debe entenderse en el sentido de **elemento a elemento**. Por ejemplo, si `A` es una matriz de elementos a_{ij}, `exp(A)` es otra matriz con las mismas dimensiones que `A`, cuyos elementos son $e^{a_{ij}}$.

Algunas otras funciones útiles en cálculos matriciales son:

`sum(v)`	suma de los elementos del vector `v`
`sum(A)` `sum(A,1)`	suma de los elementos de la matriz `A`, por columnas
`sum(A,2)`	suma de los elementos de la matriz `A`, por filas

`prod(v)`, `prod(A)`
`prod(A,1)`
`prod(A,2)`
	como la suma, pero para el producto

`max(v)`, `min(v)`	máximo/mínimo elemento del vector `v`
`max(A)`, `min(A)`	máximo/mínimo elemento de la matriz `A`, por columnas
`mean(v)`	promedio de los elementos del vector `v`
`mean(A)`	promedio de los elementos de la matriz `A`, por columnas.

1.7 Dibujo de curvas

La representación gráfica de una curva en un ordenador es una línea poligonal construida uniendo mediante segmentos rectos un conjunto discreto y ordenado de puntos: $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$.

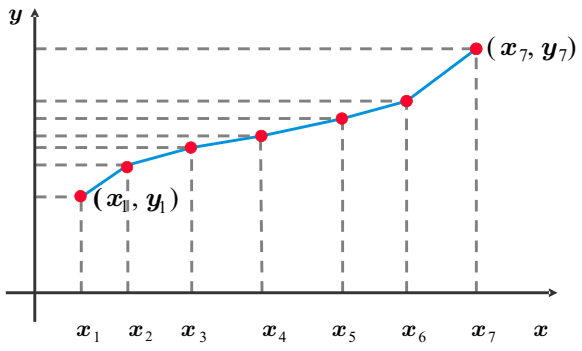


Figura 1.2: Línea poligonal determinada por un conjunto de puntos.

La línea así obtenida tendrá mayor apariencia de «suave» cuanto más puntos se utilicen para construirla, ya que los segmentos serán imperceptibles.

Para dibujar una curva plana en MATLAB se usa el comando

```
plot(x,y)
```

siendo `x` e `y` dos vectores de las mismas dimensiones conteniendo, respectivamente, las abscisas y las ordenadas de los puntos de la gráfica.

Ejemplo 1.15
Dibujar la curva $y = \dfrac{x^2+2}{x+5}$ para $x \in [-2, 3]$

```
>> f = @(x) (x.^2+2)./(x+5);
>> x = linspace(-2,3);
>> plot(x,f(x))
```

Se pueden dibujar dos o más curvas de una sóla vez, proporcionando al comando `plot` varios pares de vectores abscisas-ordenadas, como en el ejemplo siguiente.

Ejemplo 1.16
Dibujar las curvas $y = 2\,\mathrm{sen}^3(x)\,\cos^2(x)$ e $y = e^x - 2x - 3$ para $x \in [-1.5, 1.5]$

```
>> f1 = @(x) 2 * sin(x).^3 .* cos(x).^2;
>> f2 = @(x) exp(x) - 2*x -3;
>> x = linspace(-1.5,1.5);
>> plot(x,f1(x),x,f2(x))
```

A cada par de vectores abscisas-ordenadas en el comando `plot` se puede añadir un argumento opcional de tipo cadena de caracteres, que modifica el aspecto con que se dibuja la curva. Para más información, hojear el help (`help plot`).

Ejemplo 1.17
Las siguientes órdenes dibujan la curva $y = \mathrm{sen}^3(x)$ en color rojo (`r`, de **r**ed) y con marcadores $*$, en lugar de una línea contínua.

```
>> x = linspace(0,pi,30);
>> plot(x,sin(x).^3,'r*')
```

La orden siguiente dibuja la curva $y = \mathrm{sen}^3\, x$ en color negro (`k`, de blac**k**), con marcadores $*$ y con línea punteada, y la curva $y = \cos^2 x$ en color azul (`b`, de **b**lue) y con marcadores $+$

```
>> plot(x,sin(x).^3,'k*:', x, cos(x).^2,'b+')
```

Ademas, mediante argumentos opcionales, es posible modificar muchas otras propiedades de las curvas. Esto se hace siempre mediante un par de argumentos en la orden de dibujo que indican

```
'Nombre de la Propiedad', Valor de la propiedad
```

Esta propiedad afectará a todas las curvas que se dibujen con la misma orden.

Ejemplo 1.18
Para establecer un grosor determinado de las líneas se usa la propiedad `LineWidth` (atención a las mayúsculas):

```
>> plot(x,sin(x).^3,'k*:', x, cos(x).^2,'b+', 'LineWidth', 1.1)
```

Se pueden añadir elementos a la gráfica, para ayudar a su comprensión. Para añadir una leyenda que identifique cada curva se usa el comando siguiente, que asigna las leyendas a las curvas en el orden en que han sido dibujadas.

```
legend('Leyenda1', 'Leyenda2')
```

Para añadir etiquetas a los ejes que aclaren el significado de las variables se usan los comandos

```
xlabel('Etiqueta del eje OX')
ylabel('Etiqueta del eje OY')
```

Se puede añadir una cuadrícula mediante la orden

```
grid on
```

También es muy útil la orden, siguiente, que define las coordenadas mínimas y máximas del rectángulo del plano OXY que se visualiza en la gráfica.

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

Cada nueva orden de dibujo borra el contenido previo de la ventana gráfica, si existe. Para evitar esto existen la órdenes

```
hold on
hold off
```

La orden `hold on` permanece activa hasta que se cierre la ventana gráfica o bien se dé la orden `hold off`

Ejemplos 1.19
Las siguientes órdenes darán como resultado la gráfica de la figura 1.3

```
x = linspace(0,pi,30);
axis([-0.5,pi+0.5,-0.5,1.5])
hold on
plot(x,sin(x).^3,'g', x, cos(x).^2,'b+', 'LineWidth', 1.1)
x = linspace(-0.95, pi);
plot(x, log(x+1)/2, 'r', 'LineWidth', 1.1)
plot([-5,5], [0, 0], 'k', 'LineWidth', 1)
plot([0, 0], [-5,5], 'k', 'LineWidth', 1)
legend('Leyenda1', 'Leyenda2', 'Leyenda3')
xlabel('Etiqueta del eje OX')
ylabel('Etiqueta del eje OY')
hold off
shg
```

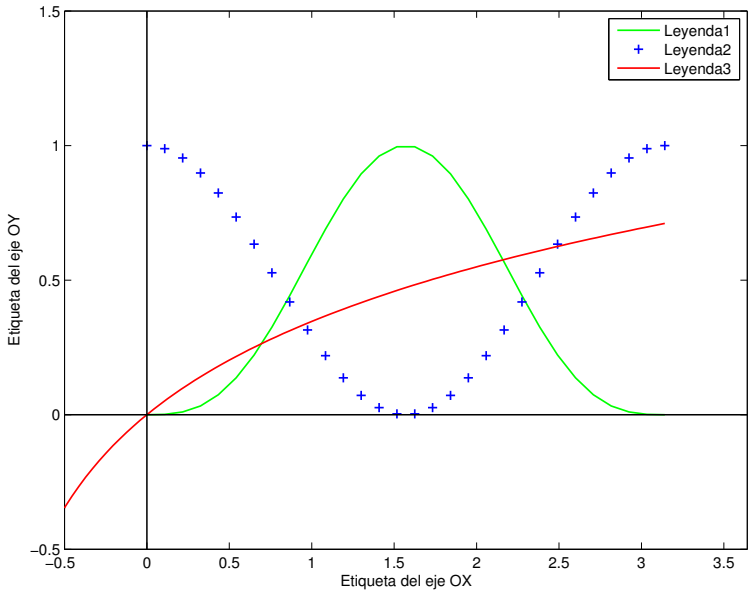


Figura 1.3: Varias curvas con leyenda y etiquetas.

2 Programación con MATLAB

Los condicionales y los bucles o repeticiones son la base de la programación estructurada. Sin ellas, las instrucciones de un programa sólo podrían ejecutarse en el orden en que están escritas (orden secuencial). Las estructuras de control permiten modificar este orden y, en consecuencia, desarrollar estrategias y algoritmos para resolver los problemas.

Los **condicionales** permiten que se ejecuten conjuntos distintos de instrucciones, en función de que se verifique o no determinada condición.

Los **bucles** permiten que se ejecute repetidamente un conjunto de instrucciones, ya sea un número pre-determinado de veces, o bien mientras que se verifique una determinada condición.

2.1 Estructuras condicionales: if

Son los mecanismos de programación que permiten "romper" el flujo secuencial en un programa: es decir, permiten hacer una tarea si se verifica una determinada **condición** y otra distinta si no se verifica.

Evaluar si se verifica o no una condición se traduce, en programación, en averiguar si una determinada **expresión con valor lógico** da como resultado **verdadero** o **falso**

Las estructuras condicionales básicas son las representadas en los diagramas de flujo siguientes:

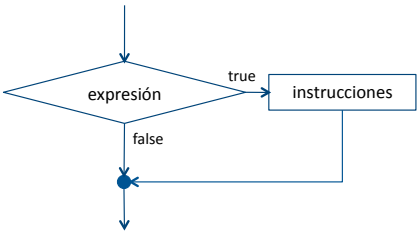


Figura 2.1: Estructura condicional simple: Si `expresión` toma el valor lógico `true`, se ejecutan las instrucciones del bloque `instrucciones` y, después, el programa se continúa ejecutando por la instrucción siguiente. Si, por el contrario, la `expresión` toma el valor lógico `false`, no se ejecutan las `instrucciones`, y el programa continúa directamente por la instrucción siguiente.

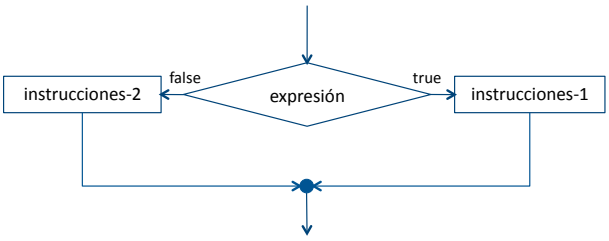


Figura 2.2: Estructura condicional doble: Si `expresión` toma el valor lógico `true`, se ejecutan las instrucciones del bloque `instrucciones 1`. Si, por el contrario, la `expresión` toma el valor lógico `false`, se ejecuta el bloque de `instrucciones 2`. En ambos casos, el programa continúa ejecutándose por la instrucción siguiente.

En casi todos los lenguajes de programación, este tipo de estructuras se implementan mediante una instrucción (o super-instrucción) denominada `if`, cuya sintaxis puede variar ligeramente de unos lenguajes a otros. En MATLAB, concretamente, su forma es la siguiente:

```
instruccion-anterior
if expresion
  bloque-de-instrucciones
end
instruccion-siguiente
```

Esta super-instrucción tiene el funcionamiento siguiente: Se evalúa `expresion`. Si el resultado es `true`, se ejecuta el `bloque-de-instrucciones` y, cuando se termina, se continúa por `instruccion-siguiente`. Si el resultado no es `true`, se va directamente a `instruccion-siguiente`.

```
instruccion-anterior
if expresion
  bloque-de-instrucciones-1
else
  bloque-de-instrucciones-2
end
instruccion-siguiente
```

Su funcionamiento es el siguiente: Se evalúa `expresion`. Si el resultado es `true`, se ejecuta el `bloque-de-instrucciones-1` y, cuando se termina, se continúa por `instruccion-siguiente`. Si el resultado no es `true`, se ejecuta el `bloque-de-instrucciones-2` y cuando se termina se va a la `instruccion-siguiente`.

Ejemplo 2.1 (Uso de un condicional simple)
Escribir una M-función que, dado $x \in \mathbb{R}$, devuelva el valor en x de la función definida a trozos

$$f(x) = \begin{cases} x+1 & \text{si } x < -1, \\ 1-x^2 & \text{si } x \geq -1. \end{cases}$$

```
function [fx] = mifun(x)
%
%  v = mifun(x)  devuelve el valor en x de la función
%                f(x) = x+1   si x < -1
%                f(x) = 1-x^2 si no
%
fx = x + 1;
if x > -1
  fx = 1 - x^2;
end
```

Ejemplo 2.2 (Uso de un condicional doble)
Escribir una M-función que, dados dos números $x, y \in \mathbb{R}$, devuelva un vector cuyas componentes sean los dos números ordenados en orden ascendente.

```
function [v] = Ordena(x, y)
%
%  v = Ordena(x, y)  es un vector que contiene los dos
%          numeros x e y ordenados en orden creciente
%
if x <= y
  v = [x, y];
else
  v = [y, x];
end
```

Observación. La orden de MATLAB `sort([x,y])` tiene el mismo efecto que esta M-función.

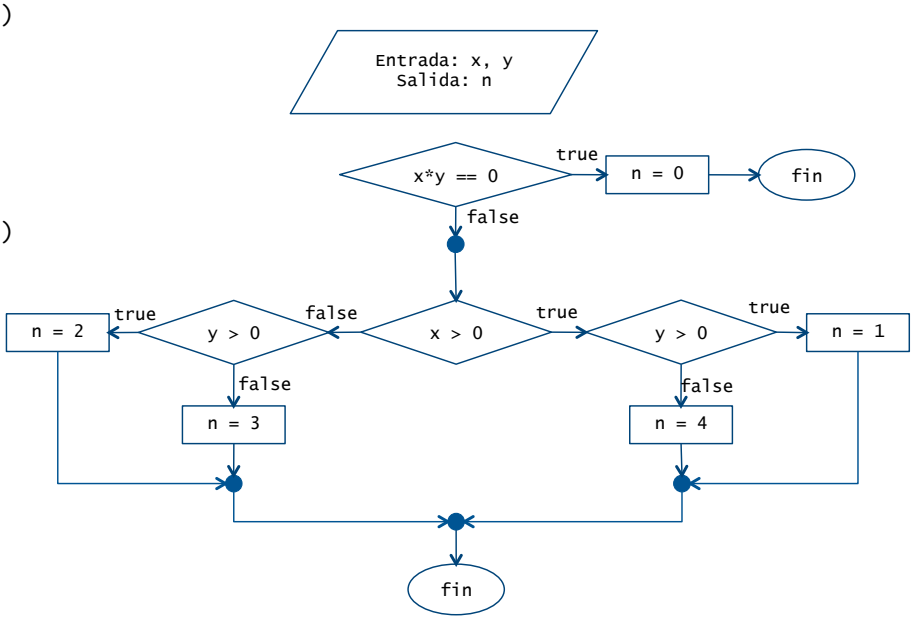
Estas estructuras se pueden "anidar", es decir, se puede incluir una estructura condicional dentro de uno de los bloques de instrucciones de uno de los casos de otra.

En ocasiones interesa finalizar la ejecución de un programa en algún punto "intermedio" del mismo, es decir, no necesariamente después de la última instrucción que aparece en el código fuente. Esto se hace mediante la orden

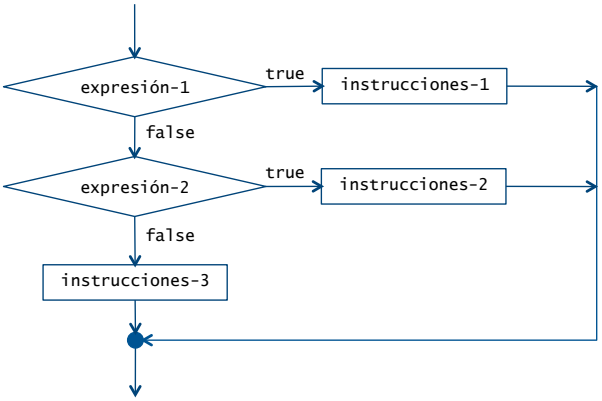
```
return
```

Ejemplo 2.3 (Condicionales anidados)
Escribir una M-función que, dados dos números $x, y \in \mathbb{R}$, devuelva el número del cuadrante del plano OXY en el que se encuentra el punto (x, y). Si el punto (x, y) está sobre uno de los ejes de coordenadas se le asignará el número 0.

```
function [n] = Cuadrante(x, y)
%
%  n = Cuadrante(x, y)  es el número del cuadrante del plano OXY
%          en el que se encuentra el punto (x,y).
%          n = 0 si el punto está sobre uno de los ejes.
%
if x*y == 0
  n = 0;
  return
end

if (x > 0)
  if (y > 0)
    n=1;
  else
    n=4;
  end
else
  if (y > 0)
    n=2;
  else
    n=3;
  end
end
```



Se pueden construir estructuras condicionales más complejas (con más casos). En MATLAB, estas estructuras se implementan mediante la versión más completa de la instrucción `if`:



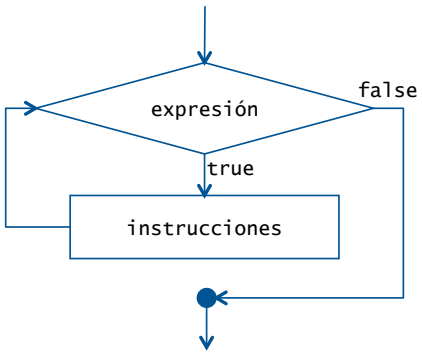

```
instruccion-anterior
if expresion-1
  bloque-de-instrucciones-1
elseif expresion-2
  bloque-de-instrucciones-2
else
  bloque-de-instrucciones-3
end
instruccion-siguiente
```

Esta estructura tiene el funcionamiento siguiente:
Se evalúa `expresion-1`.
Si el resultado es `true`, se ejecuta el `bloque-de-instrucciones-1` y, cuando se termina, se continúa por `instruccion-siguiente`.
Si el resultado de `expresion-1` no es `true`, se evalúa `expresion-2`.
Si el resultado es `true`, se ejecuta el `bloque-de-instrucciones-2` y, cuando se termina, se continúa por `instruccion-siguiente`.
Si el resultado de `expresion-2` no es `true`, se ejecuta el `bloque-de-instrucciones-3` y luego se va a `instruccion-siguiente`.

Es **muy importante** darse cuenta de que, en cualquier caso, se ejecutará **sólo uno** de los bloques de instrucciones.
La cláusula `else` (junto con su correspondiente bloque-de-instrucciones) puede no existir. En este caso es posible que no se ejecute ninguno de los bloques de instrucciones.

2.2 Estructuras de repetición o bucles: while

Este mecanismo de programación permite repetir un grupo de instrucciones **mientras que** se verifique una cierta condición. Su diagrama de flujo y su sintaxis en MATLAB son los siguientes:



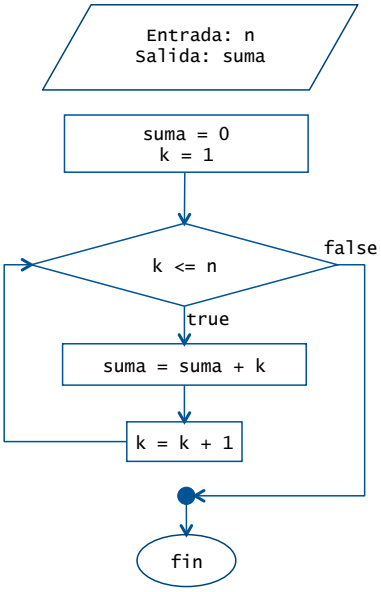

```
instruccion-anterior
while  expresion
  bloque-de-instrucciones
end
instruccion-siguiente
```

Esta estructura tiene el funcionamiento siguiente:
Al comienzo, se evalúa `expresion`.
Si el resultado **no es** `true`, se va directamente a la `instruccion-siguiente`. En este caso, no se ejecuta el `bloque-de-instrucciones`.
Si, por el contrario, el resultado de `expresion` es `true`, se ejecuta el `bloque-de-instrucciones`. Cuando se termina, se vuelve a evalúar la `expresion` y se vuelve a decidir.
Naturalmente, este mecanismo precisa que, dentro del `bloque-de-instrucciones` se modifique, en alguna de las repeticiones, el resultado de evaluar `expresion`. En caso contrario, el programa entraria

en un *bucle infinito*. Llegado este caso, se puede detener el proceso pulsando la combinación de teclas CTRL + C.

Ejemplo 2.4 (Uso de un while)
Escribir una M-función que, dado un número natural n, calcule y devuelva la suma de todos los números naturales entre 1 y n.

```
function [suma] = SumaNat(n)
%
%  suma = SumaNat(n)  es la suma de los n primeros números naturales
%
suma = 0;
k    = 1;
while  k <= n
  suma = suma + k;
  k = k + 1;
end
```



Observación. La orden de MATLAB `sum(1:n)` tiene el mismo efecto que `SumaNat(n)`.

Ejercicio.
Partiendo de la M-función `SumaNat` del ejemplo anterior, escribe una M-función que calcule y devuelva la suma de todos los números naturales impares entre 1 y n.

Ejemplo 2.5 (Uso de un while)
Escribir un *script* que genere de forma aleatoria un número natural del 1 a 9 y pida repetidamente al usuario que escriba un número en el teclado hasta que lo acierte.

Observaciones:
(a) La orden `randi(n)` genera de forma aleatoria un número natural entre 1 y n.
(b) La orden `var=input('Mensaje')` escribe `Mensaje` en la pantalla, lee un dato del teclado y lo almacena en la variable `var`.
(c) La orden `beep` emite un sonido.

```
%-----------------------------------------------------------------
%  Script Adivina
%  Este script genera un numero aleatorio entre 1 y 9
%  y pide repetidamente que se teclee un numero hasta acertar
%-----------------------------------------------------------------
%
n = randi(9);

disp(' ')
disp(' Tienes que acertar un numero del 1 al 9')
disp(' Pulsa CTRL + C si te rindes ...')
disp(' ')

M = 0;
while M~=n
    M=input('Teclea un numero del 1 al 9 :  ');
end
beep
disp(' ')
disp('  ¡¡¡¡Acertaste!!!!  ')
```

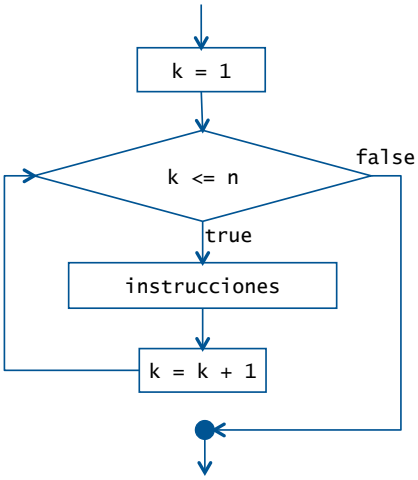
Ejercicio.
Partiendo del *script* `Adivina` del ejemplo anterior, escribe una M-función que reciba como argumento de entrada un número natural m, genere de forma aleatoria un número natural n entre 1 y m, y pida repetidamente al usuario que escriba un número en el teclado hasta que lo acierte.

2.3 Estructuras de repetición o bucles indexados: for

En muchas ocasiones, las repeticiones de un bucle dependen en realidad de una variable entera cuyo valor se va incrementando hasta llegar a uno dado, momento en que se detienen las repeticiones. Esto sucede, especialmente, con los algoritmos que manipulan vectores y matrices, que es lo más habitual cuando se programan métodos numéricos.

En estas ocasiones, para implementar el bucle es preferible utilizar la estructura que se expone en esta sección: **bucle indexado**. En MATLAB (igual que en muchos otros lenguajes) este tipo de mecanismos se implementan mediante una instrucción denominada `for`.

Por ejemplo, el bucle representado mediante el diagrama siguiente, se puede implementar con las órdenes que se indican:




```
instruccion-anterior
for k = 1 : n
  bloque-de-instrucciones
end
instruccion-siguiente
```

Estructura de repetición `for`: Para cada valor de la variable `k` desde `1` hasta `n`, se ejecuta una vez el `bloque-de-instrucciones`. Cuando se termina, el programa se continúa ejecutando por la `instrucción-siguiente`.
Se observa que, con esta instrucción, no hay que ocuparse ni de inicializar ni de incrementar dentro del bucle la variable-índice, `k`. Basta con indicar, junto a la cláusula `for`, el conjunto de valores que debe tomar. Puesto que es la propia instrucción `for` la que gestiona la variable-índice `k`, **está prohibido modificar su valor** dentro del bloque de instrucciones.

El conjunto de valores que debe tomar la variable-índice `k` tiene que ser de números enteros, pero no tiene que ser necesariamente un conjunto de números consecutivos. Son válidos, por ejemplo, los conjuntos siguientes:

```
for k = 0 : 2 : 20           % números pares de 0 a 20
for ind = 30 : -5 : 0        % números 30, 25, 20, 15, 10, 5, 0
for m = [2, 5, 4, 1, 7, 20]  % los números especificados
```

Observación. Siempre que en un bucle sea posible determinar *a priori* el número de veces que se va a repetir el bloque de instrucciones, es preferible utilizar la instrucción `for`, ya que la instrucción `while` es más lenta.

Ejemplo 2.6 (Uso de for)
Escribir una M-función que, dado un vector `v`, calcule el valor máximo entre todos sus componentes.

```
function [vmax] = Maximo(v)
%
%  Maximo(v) es el máximo de las componentes del vector v
%
vmax = v(1);
for k = 2:length(v)
  if v(k) > vmax
    vmax = v(k);
  end
end
```

Ejemplo 2.7 (Uso de for)
Observación:
La orden `error('mensaje')` imprime `mensaje` en la pantalla y detiene la ejecución del programa.

Escribir una M-función que calcule la traza de una matriz.

```
function [y] = Traza(A)
%
% Traza(A) es la suma de los elementos diagonales de la
%          matriz cuadrada A
%
[n,m] = size(A);
if n ~= m
    error('La matriz no es cuadrada')
end
y = 0;
for k = 1:n
    y = y + A(k,k);
end
```

2.4 Ruptura de bucles de repetición: break y continue

En ocasiones es necesario interrumpir la ejecución de un bucle de repetición en algún punto interno del bloque de instrucciones que se repiten. Lógicamente, ello dependerá de que se verifique o no alguna condición. Esta interrupción puede hacerse de dos formas:

- Abandonando el bucle de repetición definitivamente.
- Abandonando la iteración en curso, pero comenzando la siguiente.

Las órdenes respectivas en MATLAB son (tanto en un bucle `while` como en un bucle `for`):

```
break
continue
```

El funcionamiento de estas órdenes queda reflejado en los diagramas de flujo correspondientes (Figuras 2.3 y 2.4).

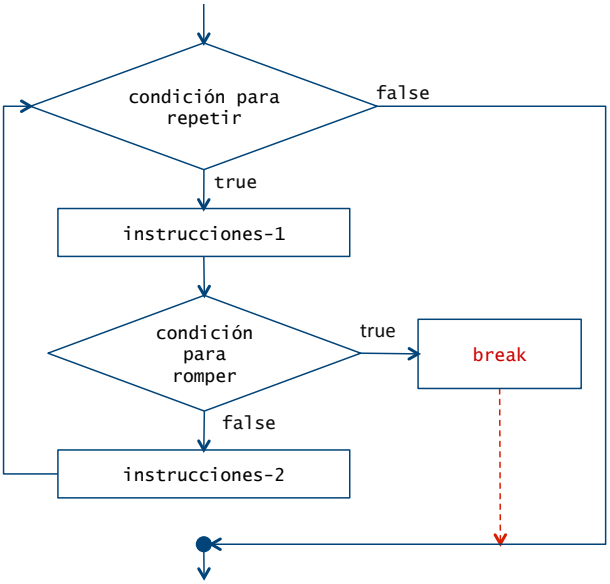


Figura 2.3: Ruptura de bucle `break`: Si en algún momento de un bucle de repetición (de cualquier tipo) se llega a una instrucción `break`, el ciclo de repetición se interrumpe inmediatamente y el programa continúa ejecutándose por la instrucción siguiente a la cláusula `end` del bucle. Si se trata de un bucle indexado `for`, la variable-índice del mismo conserva el último valor que haya tenido.

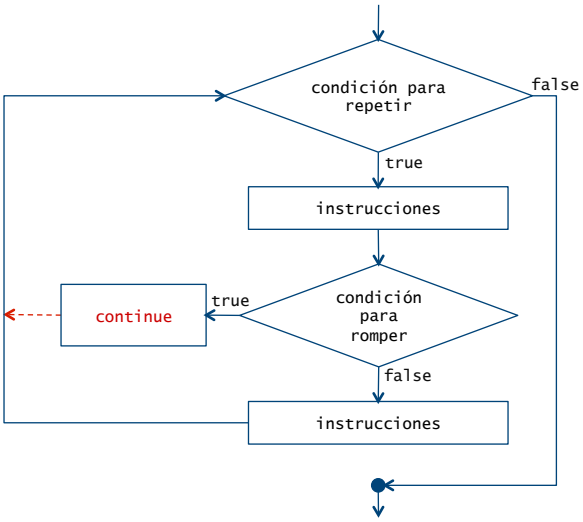


Figura 2.4: Ruptura de bucle `continue`: Si en algún momento de un bucle de repetición (de cualquier tipo) se llega a una instrucción `continue`, la iteración en curso se interrumpe inmediatamente, y se inicia la siguiente iteración (si procede).

2.5 Gestión de errores: warning y error

La instrucción siguiente se utiliza para alertar al usuario de alguna circunstancia no esperada durante la ejecución de un programa, pero no lo detiene. La orden

```
warning('Mensaje')
```

imprime `Warning: Mensaje` en la pantalla y continúa con la ejecución del programa.
Por el contrario, la instrucción

```
error('Mensaje')
```

imprime `??? Mensaje` en la pantalla y detiene en ese punto la ejecución del programa.

Además de estas funciones y sus versiones más completas, MATLAB dispone de otras órdenes para gestionar la detección de errores. Véase Error Handling en la documentación.

2.6 Operaciones de lectura y escritura

2.6.1 Instrucción básica de lectura: input

La instrucción `input` permite almacenar en una variable un dato que se introduce a través del teclado. La orden

```
var = input('Mensaje')
```

imprime `Mensaje` en la pantalla y se queda esperando hasta que el usuario teclea algo en el teclado, terminado por la tecla `return`. Lo que se teclea puede ser cualquier expresión que use constantes y/o variables existentes en el Workspace. El resultado de esta expresión se almacenará en la variable `var`. Si se pulsa la tecla `return` sin teclear nada se obtendrá una matriz vacía: `[ ]`.

2.6.2 Instrucción básica de impresión en pantalla: disp

La instrucción `disp` permite imprimir en la pantalla el valor de una (matriz) constante o variable, sin imprimir el nombre de la variable o `ans` y sin dejar líneas en blanco. Su utilidad es muy limitada.

```
disp(algo)
```

Ejemplos 2.8 (Uso de disp)
Observaciones:
(a) La orden `date` devuelve una cadena de caracteres con la fecha actual.
(b) La orden `num2str(num)` devuelve el dato numérico `num` como una cadena de caracteres.

```
>> disp(pi)
    3.1416

>> v = [1, 2, 3, pi];
>> disp(v)
    1.0000    2.0000    3.0000    3.1416

>> disp('El metodo no converge')
El metodo no converge

>> disp(['Hoy es ', date])
Hoy es 18-Feb-2015

>> x = pi;
>> disp(['El valor de x es: ',num2str(x)])
El valor de x es: 3.1416
```

2.6.3 Instrucción de impresión en pantalla con formato: fprintf

Esta orden permite controlar la forma en que se imprimen los datos. Su sintaxis para imprimir en la pantalla es

```
fprintf( formato, lista_de_datos )
```

donde:

`lista_de_datos` son los datos a imprimir. Pueden ser constantes y/o variables, separados por comas.

`formato` es una cadena de caracteres que describe la forma en que se deben imprimir los datos. Puede contener combibaciones de los siguientes elementos:

- Códigos de conversión: formados por el símbolo `%`, una letra (como `f`, `e`, `i`, `s`) y eventualmente unos números para indicar el número de espacios que ocupará el dato a imprimir.
- Texto literal a imprimir
- Caracteres de escape, como `\n`.

Normalmente el `formato` es una combinación de texto literal y códigos para escribir datos numéricos, que se van aplicando a los datos de la lista en el orden en que aparecen.

En los ejemplos siguientes se presentan algunos casos simples de utilización de esta instrucción. Para una comprensión más amplia se debe consultar la ayuda y documentación de MATLAB.

Ejemplos 2.9 (Uso de fprintf)

```
>> long = 32.067
>> fprintf('La longitud es de %12.6f metros \n',long)
La longitud es de    32.067000 metros
```

En este ejemplo, el formato se compone de:

- el texto literal `'La longitud es de '` (incluye los espacios en blanco),
- el código `%12.6f` que indica que se escriba un número (en este caso el valor de la variable `long`) ocupando un total de 12 espacios, de los cuales 6 son para las cifras decimales,
- el texto literal `' metros '` (también incluyendo los blancos),
- el carácter de escape `\n` que provoca un salto de línea.

```
>> x = sin(pi/5);
>> y = cos(pi/5);
>> fprintf('Las coordenadas del punto son x= %10.6f e y=%7.3f \n', x, y)
Las coordenadas del punto son x=   0.587785 e y=  0.809
```

Observamos que el primer código `%10.6f` se aplica al primer dato a imprimir, `x`, y el segundo código `%7.3f` al segundo, `y`.

```
>> k = 23;  err = 0.00000314;
>> fprintf('En la iteracion k=%3i el error es %15.7e \n', k, err)
En la iteracion k= 23 el error es   3.1400000e-06
```

- el código `%3i` indica que se escriba un número entero ocupando un total de 3 espacios,
- el código `%15.7e` indica que se escriba un número en formato exponencial ocupando un total de 15 espacios, de los cuales 7 son para los dígitos a la derecha del punto decimal.

2.7 Comentarios generales

Algunos comentarios generales sobre la escritura de programas:

- A los argumentos de salida de una M-función **siempre hay que darles algún valor**.
- Las instrucciones `for`, `if`, `while`, `end`, `return`, `...` no necesitan llevar punto y coma al final, ya que no producen salida por pantalla. Sí necesitan llevarlo, en general, las instrucciones incluidas dentro.
- Las M-funciones no necesitan llevar `end` al final. De hecho, a nivel de este curso, que lo lleven puede inducir a error. Mejor no ponerlo.
- Todos los programas deben incluir unas líneas de comentarios que expliquen de forma sucinta su cometido y forma de utilización. En las M-funciones, estas líneas deben ser las siguientes a la instrucción `function`, ya que de esta manera son utilizadas por MATLAB como el texto de `help` de la función. En los *scripts* deben ser las primeras líneas del archivo.
- Es bueno, en la medida de lo posible y razonable, respetar la notación habitual de la formulación de un problema al elegir los nombres de las variables. Ejemplos
  - Llamar `S` a una suma y `P` a un producto, mejor que `G` o `N`.
  - Llamar `i, j, k, l, m, n` a un número entero, mejor que `x` ó `y`.
  - Llamar `T` o `temp` a una temperatura, mejor que `P`.
  - Llamar `e`, `err` ó `error` a un erro, mejor que `vaca`.
- Hay que comprobar los programas que se hacen, dándoles valores a los argumentos para verificar que funciona bien en **todos los casos que puedan darse**. Esto forma parte del proceso de diseño y escritura del programa.
- Los espacios en blanco, en general, hacen el código más legible y por lo tanto más fácil de comprender y corregir. Se deben dejar uno o dos espacios entre el carácter `%` de cada línea de comentario y el texto del comentario (comparénse los dos códigos siguientes).

```
function[Ma,Mg]=Medias(N)
%Ma=Medias(N)  devuelve la media aritmetica
%de los numeros naturales menores o iguales
%que N
%[Ma,Me]=Medias(N) devuelve tambien la
media
%geometrica
Ma=0;
Mg=1;
for i=1:N
Ma=Ma+i;
Mg=Mg*i;
end
N1=1/N;
Ma=Ma*N1;
Mg=Mg^(N1);
```

```
function [Ma,Mg]=Medias(N)
%-------------------------------------
%  Ma=Medias(N)  devuelve la media
%     aritmetica de los numeros
%     naturales menores o iguales que N
%  [Ma,Me]=Medias(N) devuelve tambien la
%     media geometrica
%-------------------------------------
%

Ma = 0;
Mg = 1;

for i=1:N
  Ma = Ma+i;
  Mg = Mg*i;
end

N1  = 1/N;
Ma = Ma*N1;
Mg = Mg^(N1);
```

3 Operaciones de lectura y escritura en ficheros con MATLAB

Hasta ahora se ha visto:

- Cómo leer valores que el usuario escribe en el teclado (orden `input`).
- Cómo imprimir información en la ventana de comandos de MATLAB (órdenes `disp` y `fprintf`).

En este tema veremos operaciones avanzadas de lectura y escritura, esto es, que operan sobre el contenido de un fichero.

3.1 Introducción a la lectura y escritura en ficheros: órdenes save y load

3.1.1 Grabar en fichero el espacio de trabajo: orden save

La utilización más sencilla de la orden `save` es para guardar en un fichero todo el contenido del **workspace** (espacio de trabajo), es decir, el conjunto de las variables almacenadas en memoria:

```
save nombredefichero
save('nombredefichero')          % también se puede escribir así
```

Esta orden guarda todo el contenido del workspace (nombres de variables y sus valores) en un fichero de nombre `nombredefichero.mat`. Esto puede ser útil si se necesita abandonar una sesión de trabajo con MATLAB sin haber terminado la tarea: se puede salvar fácilmente a un fichero el workspace en su estado actual, cerrar MATLAB y luego recuperar todas las variables en una nueva sesión. Esto último se hace con la orden

```
load nombredefichero
load('nombredefichero')          % equivalente a la anterior
```

Ejercicio 3.1 Uso de save para salvar el workspace

1. Crea unas cuantas variables en tu espacio de trabajo. Por ejemplo:

```
A = rand(5,5);
B = sin(4*A);
x = 1:15;
frase = 'Solo se que no se nada';
```

2. Comprueba con la orden `who` qué variables tienes en el workspace:

```
who
```

3. Salva el contenido del workspace a un fichero de nombre `memoria.mat`:

```
save memoria
```

   Comprueba que en tu carpeta de trabajo aparece un fichero con este nombre.

4. Borra todas las variables de la memoria:

```
clear
```

   y comprueba con `who` que el workspace está vacío.

5. Recupera tus variables desde el fichero `memoria.mat`:

```
load memoria
```

6. Comprueba de nuevo, con `who`, que vuelves a tener todas tus variables.

También es posible salvar sólo parte de las variables:

```
save nombredefichero var1  var2 var3        (sin comas de separación)
```

Ejercicio 3.2 Uso de save para salvar algunas variables

1. Comprueba con `who` que tienes tus variables en el workspace. Si no es así, recupéralas usando la orden

```
load memoria
```

2. Salva, a otro fichero de nombre `memo.mat` el contenido de dos de las variables que tengas, por ejemplo:

```
save memo A frase
```

Comprueba que en tu carpeta de trabajo aparece un fichero de nombre `memo.mat`.

3. Borra las variables salvadas:

```
clear A frase
```

y comprueba con `who` que ya no existen en el workspace.

4. Recupera tus variables:

```
load memo
```

y comprueba con `who` que ahora están de nuevo en el workspace.

3.1.2 Distintos tipos de ficheros: de texto y binarios

Existen muchos tipos distintos de ficheros, que normalmente se identifican por distintas extensiones;

```
.txt      .dat      .m        .mat
.jpg      .xls      .pdf            etc.
```

Una de las cosas que diferencia unos ficheros de otros es el hecho de que su contenido sea o no legible con un editor de texto plano, también llamado **editor *ascii***[1]. El editor de MATLAB es un editor *ascii* y también lo es el Bloc de Notas de Windows y cualquier editor orientado a la programación. Word no es un editor *ascii*.

Desde el punto de vista anterior, existen dos tipos de ficheros:

- Ficheros **formateados** o **de texto**: son aquéllos cuyo contenido está formado por texto plano. Se pueden «ver» con cualquier editor de texto *ascii*.
- Ficheros **no formateados** ó **binarios**: contienen información de cualquier tipo codificada en binario, es decir, tal y como se almacena en la memoria del ordenador, utilizando ceros y unos. Ejemplos de ficheros binarios son ficheros de imágenes, ficheros que contienen programas ejecutables (es decir, escritos en código-máquina), etc. Sólo se pueden utilizar con programas adecuados para cada fichero.

Ejercicio 3.3 Los ficheros `.mat` no son editables

Los ficheros `.mat` creados por la orden `save` son ficheros binarios, es decir, no son editables. Veámos qué ocurre si lo abrimos con un editor *ascii*:

1. En la ventana Current Folder (Carpeta de trabajo), pon el cursor encima del icono del fichero `memoria.mat` y con el botón derecho del ratón elige Open as Text (Abrir como Texto).

[1] ASCII = American Standard Code for Information Interchange (Código Estándar Estadounidense para el Intercambio de Información).

2. Observarás un montón de signos extraños: has abierto como texto *ascii* un fichero que no lo es.
3. Cierra el editor

La orden `save` también puede ser usada para guardar datos en un fichero de texto:

```
save nombrefichero var1 var2 -ascii
```

En este caso, el fichero no tiene la extensión `.mat`, ya que la misma está reservada para los ficheros binarios de MATLAB. Se debe especificar la extensión junto con el nombre del fichero. Además, no se guardan en el fichero los nombres de las variables, solamente sus valores.

Ejercicio 3.4 Uso de save para salvar datos en un fichero de texto

1. Comprueba con `who` que tienes tus variables en el workspace. Si no es así, recupéralas usando la orden

   ```
   load memoria
   ```

2. Salva, a un fichero de texto de nombre `memo.dat` el contenido de algunas de las variables que tengas, por ejemplo:

   ```
   save memo.dat A frase -ascii
   ```

   El cualificador `-ascii` indica que se salvaguarden los datos en un fichero formateado. Comprueba que en tu carpeta de trabajo aparece un fichero de nombre `memo.dat`.

3. Puedes usar el editor de MATLAB para «ver» y modificar el contenido de este fichero. Abre el fichero desde el menú Open file ... o bien con la orden

   ```
   edit memo.dat
   ```

Es importante hacer notar que los datos guardados de esta manera en un fichero de texto no pueden ser recuperados en la memoria con la orden `load`, a menos que se trate de una sola matriz de datos numéricos, es decir que cada línea del fichero contenga la misma cantidad de datos. En ese caso hay que usar la versión `A=load('nombredefichero.extension')` de la orden para guardar los valores de la matriz en la variable `A`.

Ejercicio 3.5 Cargar datos desde un fichero y hacer una gráfica con ellos.

1. En primer lugar vas a crear el fichero que luego se leerá. Crea dos variables `x` e `y` con las abscisas y las ordenadas de los puntos de una curva. Crea también una variable con un título para la gráfica. Por ejemplo:

```
t = linspace(0,2*pi,200);
r = 2*sin(3*t);
x = r.*cos(t);
y = r.*sin(t);
titulo = 'Grafica de la funcion';
```

2. Salva, a un fichero `datos1.mat` el contenido de las variables `x`, `y` y `titulo`:

```
save  datos1  x  y  titulo
```

   Comprueba que en tu carpeta de trabajo se ha creado un fichero `datos1.mat`. Recuerda que se trata de un fichero binario que no se puede editar.

3. Escribe ahora un *script* de nombre `Plot_datos.m` que «cargue» las variables del fichero y genere la gráfica correspondiente. Para ello abre un fichero nuevo en el editor de MATLAB y sálvalo con el nombre `Plot_datos.m`. Dentro de la ventana del editor escribe:

```
file = input('Nombre del fichero a cargar : ','s');
load(file)
plot(x,y)
title(titulo)
```

   Salva el *script* y ejecútalo.

Ejercicio 3.6 Se dispone de un fichero `temperaturas.dat` en el que se almacenan datos meteorológicos de cierta ciudad. Las temperaturas están expresadas en grados Celsius, con un decimal. Concretamente, el fichero contiene la temperatura a lo largo del día, medida cada hora, desde las 0 hasta las 23 horas (en total 24 datos por día). Cada línea contiene las temperaturas de un día. El fichero contiene cuatro líneas.

Hay que realizar una gráfica que represente, mediante 4 curvas, la evolución de las temperaturas a lo largo de cada día.

1. Puesto que el fichero `temperaturas.dat` es de texto, se puede editar. Ábrelo para ver su estructura. Cierra el fichero.

2. Comienza por «cargar» en memoria los datos del fichero `temperaturas.dat`, que es un fichero de texto. Puesto que todas las filas del fichero contienen el mismo número de datos, esto se puede hacer con la orden `load`:

```
Temp = load('temperaturas.dat');
```

   que guarda en la variable `Temp` una matriz con 4 filas y 24 columnas conteniendo los datos del fichero. Compruébalo. Cada fila de `Temp` contiene las temperaturas de un día.

3. Crea un vector con 24 componentes representando las horas del día, para usarlo como abscisas:

```
hh = 0:23;
```

4. Se quiere construir una curva con cada fila de la matriz `Temp`. Esto se puede hacer con la orden

```
plot(hh,Temp(1,:),hh,Temp(2,:),hh,Temp(3,:),hh,Temp(4,:),)
```

Pero también se puede hacer mediante la orden

```
plot(hh,Temp)
```

Consulta en el Help de MATLAB este uso de la función `plot` en el que el segundo argumento es una matriz.

3.2 Lectura y escritura en ficheros avanzada

Las órdenes `save` y `load` son fáciles de usar, aunque apenas permiten diseñar su resultado y sólo sirven en algunos casos.

Por ejemplo, con la orden `save` no se pueden escribir datos en un fichero con un formato personalizado, como una tabla, mezclando texto y datos numéricos, etc.

Con la orden `load` no podemos cargar en memoria datos de un fichero de texto, a menos que se trate de una sola matriz de datos numéricos.

Con frecuencia es necesario utilizar ficheros procedentes de otros usuarios o creados por otros programas y cuya estructura y formato no se puede elegir o, por el contrario, es preciso generar un fichero con una estructura determinada.

Para todo ello es necesario recurrir a órdenes de lectura/escritura de «bajo nivel». Estas órdenes son más complicadas de usar, ya que requieren más conocimiento del usuario pero, a cambio, permiten elegir todas las caracterísitcas de la operación que se realiza.

Los pasos en el trabajo «de bajo nivel» con ficheros son:

Abrir el fichero e indicar qué tipo de operaciones se van a efectuar sobre él. Esto significa que el fichero queda listo para poder ser utilizado por nuestro programa.

Leer datos que ya están en el fichero, o bien **Escribir** datos en el fichero o **Añadir** datos a los que ya hay.

Cerrar el fichero.

Sólo hablamos aquí de operaciones con ficheros formateados. No obstante, en algunas ocasiones, como por ejemplo cuando hay que almacenar grandes cantidades de datos, puede ser conveniente utilizar ficheros binarios, ya que ocupan menos espacio y las operaciones de lectura y escritura sobre ellos son más rápidas.

3.2.1 Apertura y cierre de ficheros

Para **abrir** un fichero sólo para lectura con MATLAB se utiliza la orden

```
fid=fopen('NombreFichero');
```

`NombreFichero` es el nombre del fichero que se desea utilizar.

`fid` es un número de identificación que el sistema asigna al fichero, si la operación de apertura ha sido exitosa. Si no, `fopen` devolverá el valor `fid=-1`. En adelante se usará ese número para cualquier operación sobre el fichero que se quiera realizar.

De forma más general, la orden para abrir un fichero es:

```
fid=fopen('NombreFichero','permiso');
```

`permiso` es un código que se utiliza para indicar si se va a utilizar el fichero para leer o para escribir. Posibles valores para el argumento `permiso` son:

- `r` indica que el fichero se va a utilizar sólo para lectura. Es el valor por defecto. (En Windows conviene poner `rt`).
- `w` indica que el fichero se va a utilizar para escribir en él. En este caso, si existe un fichero con el nombre indicado, se abre y se sobre-escribe (el contenido previo, si lo había, se pierde). Si no existe un fichero con el nombre indicado, se crea. (En Windows conviene poner `wt`)
- `a` indica que el fichero se va a utilizar para añadir datos a los que ya haya en él. (En Windows conviene poner `at`)

  Para otros posibles valores del argumento `permiso`, ver en el Help de MATLAB la descripción de la función `fopen`.

Una vez que se ha terminado de utilizar el fichero, hay que cerrarlo, mediante la orden

```
fclose(fid)
```

donde `fid` es el número de identificación que `fopen` le dió al fichero. Si se tienen varios ficheros abiertos a la vez se pueden cerrar todos con la orden

```
fclose('all')
```

3.2.2 Leer datos de un fichero con fscanf

La orden

```
A = fscanf(fid,'formato');
```

lee los datos de un fichero hasta el final o hasta que se encuentre algún dato que no corresponde con el formato proporcionado.

`fid` es el número de identificación del fichero, devuelto por la orden `fopen`.

`formato` es una descripción del formato que hay que usar para leer los datos. Se utilizan los mismos códigos que con la orden `fprint`.

`A` es un vector columna en el que se almacenan los datos leídos, en el orden en que están escritos.

Ejercicio 3.7 Leer los datos del fichero `datos3.dat` con `fscanf` y almacenarlos en una matriz con 2 columnas.

1. Edita el fichero `datos3.dat` para ver su estructura y luego ciérralo:

```
1.000   3.000
2.000   1.500
3.000   1.000
4.000   0.750
5.000   0.600
etc.
```

2. Abre el fichero `datos3.dat` para lectura:

```
fid = fopen('datos3.dat','r');
```

3. Lee los datos del fichero con la orden:

```
mat = fscanf(fid,' %f');
```

```
1.0000
3.0000
2.0000
1.5000
3.0000
etc.
```

4. Para que los datos formen una matriz con 2 columnas como en el fichero hay que usar la orden

```
mat2 = reshape(mat,2,[])';
```

que en primer lugar organiza los datos como una matriz con 2 filas y luego la transpone. Comprueba que la matriz `mat2` contiene los datos organizados como en el fichero.

5. Cierra el fichero

```
fclose(fid);
```

Observación: el ejercicio 3.7 se podría haber hecho usando la orden `load`, ya que todas las filas están formadas por el mismo número de datos. Sin embargo la orden `fscanf` sirve para otros casos que no se podrían leer con `load`, como en el ejercicio siguiente.

Ejercicio 3.8 Se dispone de un fichero `contaminacion.dat` que contiene datos sobre contaminación de las provincias andaluzas.

Cada línea del fichero contiene el nombre de la provincia y tres datos numéricos.

Se quieren recuperar los datos numéricos del fichero en una matriz con tres columnas, es decir, ignorando el texto de cada línea.

1. Abre con el editor el fichero `contaminacion.dat` para ver su estructura:

   ```
   Almeria     3.7    1.4    2.0
   Cadiz       9.5    8.5    3.3
   Almeria     0.9    4.2    9.9
   etc.
   ```

   Cierra el fichero.

2. Intenta abrir el fichero con la orden `load`. Recibirás un mensaje de error, ya que, como hemos dicho, load no puede leer ficheros de texto con datos de distintos tipos:

   ```
   mat = load('contaminacion.dat');     % producira errores
   ```

3. Sin embargo, el fichero se puede leer con la orden `fscanf`. Abre el fichero para lectura:

   ```
   fid = fopen('contaminacion.dat','r');
   ```

4. Escribe la orden siguiente:

   ```
   mat = fscanf(fid,' %f');
   ```

   que indica que se va a leer el contenido del fichero hasta que se termine o bien hasta que los datos a leer no coincidan con el formato especificado. El formato especifica leer 1 dato real. Si después de éste siguen quedando datos en el fichero, se «re-visita» el el formato, es decir, es como si pusiéramos `%f %f %f %f %f %f ...` hasta que se acabe el fichero.

   Verás que el resultado es una matriz vacía. ¿Cual es el error?

5. Escribe ahora la orden:

   ```
   frewind(fid);
   ```

   Esta orden «rebobina» el fichero, es decir, coloca el apuntador de lectura al principio del mismo.

6. Escribe ahora la orden

   ```
   mat = fscanf(fid,' %s %f %f %f');
   ```

Esta orden indica que se han de leer del fichero 1 dato tipo *string* (cadena de caracteres), luego 3 reales y repetir. Ahora obtendrás una matriz con una sola columna. Observa sus valores. ¿Es lo que esperabas? Intenta comprender lo que has obtenido.

7. Rebobina de nuevo el fichero y escribe la orden:

```
mat = fscanf(fid,' %*s %f %f %f')
```

Esta orden indica que se **ignore** (`%*s`) un dato de tipo *string*, que luego se lean 3 reales y luego repetir. Observa el resultado.

8. Para dar a la matriz `mat` la estructura deseada hay que usar la orden

```
mat = reshape(mat,3,[])';
```

que indica que se re–interprete `mat` como una matriz con 3 filas y el número de columnas que salgan y luego se transpone.

3.2.3 Escribir datos en un fichero con fprintf

La orden

```
fprintf(fid,'formato',lista_de_variables);
```

escribe los valores de la variables de la `lista_de_variables` en el fichero con numero de identificación `fid` de acuerdo con el formato especificado.

`fid` es el número de identificación del fichero, devuelto por la orden `fopen`.

`formato` es una descripción del formato que hay que usar para escribir los datos.

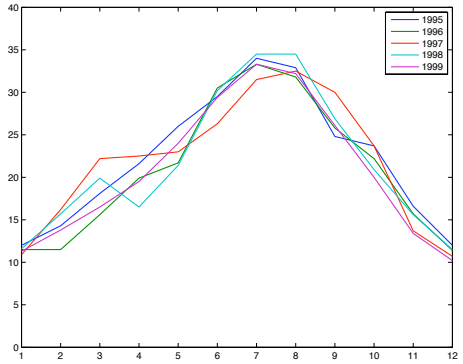
`lista_de_variables` son las variables o expresiones cuyos valores hay que escribir en el fichero.

Ejercicio 3.9 El siglo pasado era costumbre escribir los números de los años con sólo dos dígitos: 76, 77, 78, etc.

Se dispone de un fichero `temp_anuales.dat` en donde se almacena la temperatura máxima media de cada mes del año, para varios años del siglo XX. Los años vienen expresados con dos dígitos y las temperaturas vienen expresadas con una cifra decimal.

Se necesita generar a partir de este fichero otro que contenga los mismos datos, pero con los años expresados con 4 dígitos, para poder añadir los de este siglo. Las temperaturas deben ser escritas, también, con un decimal.

Genera también una gráfica que muestre la evolución de las temperaturas a lo largo del año (una curva para cada año). Dibuja cada curva de un color y añade a cada curva una leyenda con el número del año al que corresponde, como en la gráfica siguiente:



1. Puesto que el contenido del fichero `temp_anuales.dat` es una matriz de números, se puede leer con la orden `load`:

```
datos = load('temp_anuales.dat');
```

2. Los números de los años forman la primera columna de la matriz datos, así que un vector conteniendo los años con la numeración de 4 cifras se obtiene con:

```
anyos = datos(:,1)+1900;
```

Observación: para escribir los datos en un nuevo fichero no podemos usar la orden `save`, ya que los escribiría con formato exponencial (p.e. 9.5000000e+01) y queremos escribir los años sin decimales y los datos con solo un decimal.

Hay que usar órdenes de «bajo nivel», concretamente la orden `fprintf`.

3. Abrimos un nuevo fichero para escritura, en el que grabaremos los nuevos datos.

```
fid = fopen('temp_anuales_new.dat','w');
```

4. Recuperamos las dimensiones de la matriz `datos`, para saber cuántas filas tiene:

```
[nfils,ncols] = size(datos);        % nfils = numero de filas
```

5. Vamos a realizar un bucle con un índice `k` variando desde `1` hasta `nfils`. En cada iteración del bucle escribiremos una fila del nuevo fichero, mediante las órdenes:

```
fprintf(fid,' %4i',anyos(k));          % el año con formato %4i
fprintf(fid,' %6.1f',datos(k,2:13));   % los datos con formato %6.1f
fprintf(fid,'\n');                     % al final un salto de línea
```

Prueba a escribir el nuevo fichero mediante una estrategia distinta de esta.

6. Cerramos el fichero

```
fclose(fid);
```

7. La gráfica pedida se puede hacer con las órdenes

```
plot(1:12,datos(:,2:13),'LineWidth',1.1);
legend(num2str(anyos));
axis([1,12,0,40])
shg
saveas(gca,'temp_anuales.pdf','pdf')
```

Analiza estas órdenes pues usan algunas funcionalidades nuevas. La orden `shg` posiciona la ventana gráfica delante de todas las demás. La orden `saveas(gca,'temp_anuales.pdf','pdf')` salva la gráfica a un fichero **pdf**.

Ejercicio 3.10 Modifica alguno de los *scripts* de cálculo de series (por ejemplo `SumaExp.m`) para que la tabla de las iteraciones se escriba en un fichero en lugar de en la pantalla.

Ejercicio 3.11 Una triangulación es una partición de una región en triángulos que cumplen la condición de que «cubren» toda la región y de que dos triángulos cualesquiera sólo pueden tener en común un vértice o un lado completo. Queda definida mediante dos matrices:

1. Una matriz `coor` con `2` filas y `Np` columnas que contiene las coordenadas de los vértices de los triángulos. Por ejemplo si

   ```
   coor =    0.000    -0.300    -0.900    -0.500    ...
             1.000     0.400     0.300     0.300    ...
   ```

   indica que el vértice número 1 es el punto $(0, 1)$; el vértice número 2 es el punto $(-0.3, 0.4)$, etc. En total hay `Np` vértices.

2. Una matriz `trian` con `3` columnas y `Nt` filas que indica cuáles son los vértices de cada triángulo. Por ejemplo

   ```
   trian =     7    26    25
              26    52    25
              26    27    52
              ..... etc.
   ```

   indica que el triángulo número 1 tiene los vértices 7, 26 y 25; que el triángulo número 2 tiene los vértices 26, 52 y 25, etc. En total hay `Nt` triángulos.

Se dispone de dos ficheros: `puntos.dat` que contiene la matriz `coor` y `triangulos.dat` que contiene la matriz `trian`.

Este ejercicio consiste en leer los datos de los ficheros y dibujar la triangulación.

1. Vamos a escribir un *script* llamado `mallado.m` para hacer el ejercicio. Abre un fichero nuevo, sálvalo con ese nombre escribe en él lo que sigue.

2. Comenzaremos por dibujar los vértices con marcadores. Aunque no es necesario es un buen ejercicio. Puesto que el contenido del fichero `puntos.dat` es una matriz de números, se puede leer con la orden `load`:

```
coor = load('puntos.dat');
```

3. Recuperamos las dimensiones de la matriz `coor`:

```
[Np,nc]=size(coor);        % Np es el número de vértices
```

4. Abrimos una nueva ventana gráfica, fijamos los ejes y utilizamos `hold` ya que tendremos que dibujar muchos objetos distintos:

```
figure
axis([min(coor(:,1)),max(coor(:,1)),min(coor(:,2)),max(coor(:,2))])
hold on
```

5. Para dibujar todos los vértices con marcadores realizamos un bucle y en cada iteración marcamos un punto:

```
for k=1:Np
   plot(coor(k,1),coor(k,2),'rx','LineWidth',1);
end
```

   Con esto deben quedar marcados los vértices de la triangulación. Compruébalo salvando y ejecutando el *script*.

6. A continuación leemos el fichero de `triangulos.dat`, también con `load`, y recuperamos las dimensiones de la matriz `tri`, para saber el número de triángulos que hay:

```
tri = load('triangulos.dat');
[Nt,nc]=size(tri);             % Nt es el número de triángulos
```

7. Mediante un bucle desde `k=1` hasta `n=Nt`, recorremos los triángulos y para cada uno de ellos recuperamos los números de sus tres vértices, las coordenadas de sus tres vértices y construimos una poligonal que los una

```
for k=1:Nt

 p1=tri(k,1);      %
 p2=tri(k,2);      %  los números de los vértices del triángulo
 p3=tri(k,3);      %

 x1 = coor(p1,1);  %  las coordenadas del vértice número p1
 y1 = coor(p1,2);  %

 x2 = coor(p2,1);  %  las coordenadas del vértice número p2
 y2 = coor(p2,2);  %

 x3 = coor(p3,1);  %  las coordenadas del vértice número p3
 y3 = coor(p3,2);  %
```

```
x=[x1,x2,x3,x1];  %  abscisas de la linea a dibujar

y=[y1,y2,y3,y1];  %  ordenadas de la linea a dibujar

plot(x,y,'b')

end
```

4 Resolución de sistemas lineales

4.1 Los operadores de división matricial

Sea A una matriz cualquiera y sea B otra matriz con el mismo número de filas que A[1]. Entonces, la "solución" del "sistema" (en realidad un sistema lineal por cada columna de B)

$$A\,X = B$$

se calcula en MATLAB mediante el operador no estándar "\" denominado *backward slash*, (barra inversa en español)

```
X = A \ B
X = mldivide(A, B)      (equivalente a lo anterior)
```

Hay que pensar en él como el operador de "división matricial por la izquierda".

De forma similar, si C es una matriz con el mismo número de columnas que A, entonces la solución del sistema

$$X\,A = C$$

se obtiene en MATLAB mediante el operador"/" ("división matricial por la derecha")

```
X = C / A
X = mrdivide(C, A)      (equivalente a lo anterior)
```

Estos operadores se aplican incluso si A no es una matriz cuadrada. Lo que se obtiene de estas operaciones es, lógicamente, distinto según sea el caso.

De forma resumida, si `A` es una matriz cuadrada e `y` es un vector columna, `c = A\y` es la solución del sistema lineal `Ac=y`, obtenida por diferentes algoritmos, en función de las características de la matriz `A` (diagonal, triangular, simétrica, etc.). Si `A` es singular o mal condicionada, se obtendrá un mensaje de alerta (*warning*). Si `A` es una matriz rectangular, `A\y` devuelve una solución de mínimos cuadrados del sistema `Ac=y`.

Para una descripción completa del funcionamiento de estos operadores, así como de la elección del algoritmo aplicado, véase la documentación de MATLAB correspondiente, tecleando en la ventana de comandos

```
doc mldivide
```

[1] A menos que A sea un escalar, en cuyo caso `A\B` es la operación de división elemento a elemento, es decir `A./B`.

Ejemplo 4.1 (Uso del operador \)
Calcular la solución del sistema (compatible determinado)

$$\begin{cases} 2x_1 + x_2 - x_3 &= -1 \\ 2x_1 - x_2 + 3x_3 &= -2 \\ 3x_1 - 2x_2 &= 1 \end{cases}$$

```
>> A = [2, 1, -1; 2, -1, 3; 3, -2, 0];
>> b = [-1; -2; 1];
>> x = A\b
x =
   -0.3636
   -1.0455
   -0.7727
```

Como comprobación calculamos el resíduo que, como es de esperar, no es exactamente nulo, debido a los errores de redondeo:

```
>> A*x - b
ans =
   1.0e-15 *
   -0.4441
         0
         0
```

Ejemplo 4.2 (Uso del operador \)
Calcular la solución del sistema (compatible indeterminado)

$$\begin{cases} x_1 + x_2 + x_3 &= 1 \\ 2x_1 - x_2 + x_3 &= 2 \\ x_1 - 2x_2 &= 1 \end{cases}$$

```
>> A = [1, 1, 1; 2, -1, 1; 1, -2, 0];
>> b = [ 1; 2; 1];
>> x = A\b
Warning: Matrix is singular to working precision.
x =
   NaN
   NaN
   NaN
```

Ejemplo 4.3 (Uso del operador \)
Calcular la solución del sistema (incompatible)

$$\begin{cases} 2x+2y+t &= 1 \\ 2x-2y+z &= -2 \\ x-z+t &= 0 \\ -4x+4y-2z &= 1 \end{cases}$$

```
>> A = [2, 2, 0, 1; 2, -2, 1, 0; 1, 0, -1, 1; -4, 4, -2, 0];
>> b = [ 1; -2; 0; 1];
>> x = A\b
Warning: Matrix is singular to working precision.
x =
   NaN
   NaN
  -Inf
  -Inf
```

4.2 Determinante. ¿Cómo decidir si una matriz es singular?

El determinante de una matriz cuadrada A se calcula con la orden

```
det(A)
```

Este cálculo se hace a partir de la factorización LU de la matriz A, ya que se tiene $\det(L) = 1$, y $\det(A) = \det(U)$, que es el producto de sus elementos diagonales.

El uso de `det(A) == 0` para testar si la matriz A es singular sólo es aconsejable para matrices de pequeño tamaño y elementos enteros también de pequeña magnitud.

El uso de `abs(det(A)) < epsilon` tampoco es recomendable ya que es muy difícil elegir el `epsilon` adecuado (véase el Ejemplo 4.4).

Lo aconsejable para testar la singularidad de una matriz es usar su **número de condición** `cond(A)`.

Ejemplo 4.4 (Matriz no singular, con determinante muy pequeño, bien condicionada)
Se considera la matriz 10×10 siguiente, que no es singular, ya que es múltiplo de la identidad,

```
>> A = 0.0001 * eye(10);
>> det(A)
ans =
   1.0000e-40
```

Vemos que su determinante es muy pequeño. De hecho, el test `abs(det(A)) < epsilon` etiquetaría esta matriz como singular, a menos que se eligiera `epsilon` extremadamente pequeño. Sin embargo, esta matriz no está mal condicionada:

```
>> cond(A)
ans =
     1
```

Ejemplo 4.5 (Matriz singular, con `det(A)` muy grande)
Se considera la matriz 13×13 construida como sigue, que es singular y de diagonal dominante:

```
>> A = diag([24, 46, 64, 78, 88, 94, 96, 94, 88, 78, 64, 46, 24]);
>> S = diag([-13, -24, -33, -40, -45, -48, -49, -48, -45, -40, -33, -24], 1);
>> A = A + S + rot90(S,2);
```

La matriz que se obtiene es

```
    24   -13     0     0     0     0     0     0     0     0     0     0     0
   -24    46   -24     0     0     0     0     0     0     0     0     0     0
     0   -33    64   -33     0     0     0     0     0     0     0     0     0
     0     0   -40    78   -40     0     0     0     0     0     0     0     0
     0     0     0   -45    88   -45     0     0     0     0     0     0     0
     0     0     0     0   -48    94   -48     0     0     0     0     0     0
     0     0     0     0     0   -49    96   -49     0     0     0     0     0
     0     0     0     0     0     0   -48    94   -48     0     0     0     0
     0     0     0     0     0     0     0   -45    88   -45     0     0     0
     0     0     0     0     0     0     0     0   -40    78   -40     0     0
     0     0     0     0     0     0     0     0     0   -33    64   -33     0
     0     0     0     0     0     0     0     0     0     0   -24    46   -24
     0     0     0     0     0     0     0     0     0     0     0   -13    24
```

A es singular (la suma de todas sus filas da el vector nulo). Sin embargo, el cálculo de su determinante con la función `det` da

```
>> det(A)
ans =
   1.0597e+05
```

cuando debería dar como resultado cero! Esta (enorme) falta de precisión es debida a los errores de redondeo que se cometen en la implementación del método LU, que es el que MATLAB usa para calcular el determinante. De hecho, vemos que el número de condición de A es muy grande:

```
>> cond(A)
ans =
   2.5703e+16
```

4.3 La factorización LU

La factorización LU de una matriz se calcula con MATLAB con la orden

```
[L, U, P] = lu(A)
```

El significado de los distintos argumentos es el siguiente:

`L` es una matriz triangular inferior con unos en la diagonal.
`U` es una matriz triangular superior.
`P` es una matriz de permutaciones, que refleja los intercambios de filas realizados durante el proceso de eliminación gaussiana sobre las filas de la matriz A, al aplicar la técina del pivot.
`LU=PA` es la factorización obtenida.

Entonces, para calcular la solución del sistema lineal de ecuaciones

$$A\,x = b$$

utilizando la factorización anterior sólo hay que plantear el sistema, equivalente al anterior,

$$P\,A\,x = P\,b \iff L\,U\,x = P\,b \iff \begin{cases} L\,v = P\,b \\ U\,x = v \end{cases}$$

El primero de estos dos sistemas se resuelve fácilmente mediante un algoritmo de bajada, ya que la matriz del mismo es triangular inferior. Una vez calculada su solución, v, se calcula x como la solución del sistema $U\,x = v$, que se puede calcular mediante un algoritmo de subida, al ser su matriz triangular superior.

Ejercicio 4.1 Escribir una M-función `function [x] = Bajada(A, b)` para calcular la solución `x` del sistema $Ax = b$, siendo A una matriz cuadrada triangular inferior.

Algoritmo de bajada
$n =$ dimensión de A
Para cada $i = 1, 2, \ldots n$,

$$x_i = \frac{1}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j\right)$$

Fin

Para comprobar el funcionamiento del programa, construir una matriz 20×20 (por ejemplo) y un vector columna b de números generados aleatoriamente (con la función `rand` o bien con `randi`) y luego extraer su parte triangular inferior con la función `tril`(*). (Consultar en el help de MATLAB la utilización de estas funciones).

(*) Aunque, en realidad esto no es necesario. Obsérvese que en el programa `Bajada` que sigue, no se utiliza la parte superior de la matriz `A`.

```
function [x] = Bajada(A, b)
%
%  Bajada(A, b) es la solucion del sistema lineal de
%       matriz triangular inferior Ax = b
%
%------ tolerancia para el test de singularidad
tol = 1.e-10;
%------ inicializaciones
n = length(b);
x = zeros(n,1);
for i = 1:n
    Aii = A(i,i);
    if  abs(Aii) < tol
        warning(' La matriz A es singular ')
    end
    suma = 0;                          % en version vectorial, sin usar for,
    for j = 1:i-1                      % se puede calcular la suma con:
        suma = suma + A(i,j)*x(j);     %
    end                                % suma = A(i,1:i-1)*x(1:i-1)
        x(i) = (b(i) - suma)/Aii;
end
```

Ejercicio 4.2 Escribir una M-función `function [x] = Subida(A, b)` para calcular la solución `x` del sistema $Ax = b$, siendo A una matriz cuadrada triangular superior.

Algoritmo de subida
$n =$ dimensión de A
Para cada $i = n, \ldots, 2, 1$

$$x_i = \frac{1}{A_{ii}}\Big(b_i - \sum_{j=i+1}^{n} A_{ij}x_j\Big)$$

Fin

Para comprobar el funcionamiento del programa, construir una matriz A y un vector b de números generados aleatoriamente (como en el ejercicio anterior) y luego extraer su parte triangular inferior con la función `triu`.

Ejercicio 4.3 Escribir una M-función `function [x, res] = LU(A, b)` que calcule la solución `x` del sistema $Ax = b$ y el residuo `res`$=Ax - b$ siguiendo los pasos siguientes:
— Calcular la factorización LU mediante la función `lu` de MATLAB
— Calcular la solución del sistema $Lv = Pb$ mediante la M-función `Bajada`
— Calcular la solución del sistema $Ux = v$ mediante la M-función `Subida`

Utilizar la M-función `LU` para calcular la solución del sistema

$$\begin{pmatrix} 1 & 1 & 0 & 3 \\ 2 & 1 & -1 & 2 \\ 3 & -1 & -1 & 2 \\ -1 & 2 & 3 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

Observación: Llamamos `LU` (con mayúsculas) a esta función para que no se confunda con la función `lu` (minúsculas) de MATLAB.

Ejercicio 4.4 Las matrices de Hilbert definidas por $H_{ij} = \dfrac{1}{i+j-1}$ son un ejemplo notable de matrices mal condicionadas, incluso para dimensión pequeña.

Utilizar la M-función `LU` para resolver el sistema $Hx = b$, donde H es la matriz de Hilbert de dimensión $n = 15$ (por ejemplo) y b es un vector de números generados aleatoriamente. Comprobar que el residuo es grande. Comprobar que la matriz H está mal condicionada calculando su número de condición.

La función `hilb(n)` construye la matriz de Hilbert de dimensión n.

4.4 La factorización de Cholesky

La factorización de Cholesky de una matriz simétrica se calcula en MATLAB con alguna de las órdenes

```
U = chol(A)              % se obtiene una matriz triang. superior
L = chol(A, 'lower')     % se obtiene una matriz triang. inferior
```

de manera que se tiene $U^t\,U = A$ con la primera opción y $L\,L^t = A$ con la segunda.

Cuando se usa en la forma `chol(A)`, la función `chol` sólo usa la parte triangular superior de la matriz A para sus cálculos, y asume que la parte inferior es la simétrica. Es decir, que si se le pasa una matriz A que no sea simétrica no se obtendrá ningún error. Por el contrario, si se usa en la forma `chol(A,'lower')`, sólo se usará la parte triangular inferior de A, asumiendo que la parte superior es la simétrica.

La matriz A tiene que ser definida positiva. Si no lo es, se obtendrá un error.

Una vez calculada la matriz L, para calcular la solución del sistema lineal de ecuaciones

$$A\,x = b$$

utilizando la factorización anterior sólo hay que plantear el sistema, equivalente al anterior,

$$A\,x = b \iff L\,L^t\,x = b \iff \begin{cases} L\,v = b \\ L^t\,x = v \end{cases}$$

que, de nuevo, se resuelven fácilmente utilizando sendos algoritmos de bajada + subida.

Si, por el contrario, se calcula la factorización en la forma $U^tU = A$, entonces se tiene

$$A\,x = b \iff U^t\,U x = b \iff \begin{cases} U^t\,v = b \\ U\,x = v \end{cases}$$

que se resuelve aplicando en primer lugar el algoritmo de subida y a continuación el de bajada.

Ejercicio 4.5 Escribir una M-función `function [x, res] = CHOL(A, b)` que calcule la solución `x` del sistema con matriz simétrica definida positiva $Ax = b$ y el residuo `res`$=Ax - b$ siguiendo los pasos siguientes:
— Calcular la matriz L de la factorización de Cholesky LL^t de A mediante la función `chol` de MATLAB
— Calcular la solución del sistema $Lv = b$ mediante la M-función `Bajada`
— Calcular la solución del sistema $L^tx = v$ mediante la M-función `Subida`

Para comprobar el funcionamiento del programa, se puede generar alguna de las matrices definida positivas siguientes:
— `M = gallery('moler', n)`
— `T = gallery('toeppd', n)`
— `P = pascal(n)`

Observación: Por la misma razón que en el ejercicio 4.3, llamamos `CHOL` (mayúsculas) a esta función para que no se confunda con la función `chol` (minúsculas) de MATLAB.

4.5 Resolución de sistemas con características específicas

En la resolución numérica de sistemas lineales, sobre todo si son de grandes dimensiones, resulta imprescindible aprovechar propiedades concretas que pueda tener la matriz del sistema para utilizar un método *ad hoc* que reduzca el número de operaciones a realizar y, en consecuenia, el tiempo de cálculo y los errores de redondeo.

MATLAB dispone de una gran cantidad de funciones dedicadas a este problema. Para detalles se debe consultar la documentación.

Sin entrar en muchos detalles, la función `linsolve` proporciona un poco más de control que el operador `\` sobre el método de resolución a utilizar. La orden

```
x = linsolve(A, b)
```

resuelve el sistema lineal `Ax = b` por factorización LU si A es una matriz cuadrada y regular, y por factorización QR si no.

La orden

```
x = linsolve(A, b, opts)
```

resuelve el sistema lineal `Ax = b` por el método que resulte más apropiado, dadas las propiedades de la matriz `A`, que se pueden especificar mediante el argumento opcional `opts` (véase la documentación).

4.6 Matrices huecas (*sparse*)

En muchas aplicaciones prácticas, aparecen matrices que sólo tienen un pocos elementos distintos de cero. Estas matrices se denominan *huecas* (*sparse* en la terminología en inglés, *creuse* en la terminología francesa). Por ejemplo, en simulación de circuitos y en implementación de métodos de elementos finitos, aparecen matrices que tienen menos de un 1 % de elementos no nulos.

Para tales matrices, es un enorme desperdicio de memoria almacenar todos sus ceros, y es un enorme despercio de tiempo realizar operaciones aritméticas con ceros. Por esta razón se han desarrollado un buen número de estrategias de almacenamiento que evitan el uso de memoria para guardar ceros, y se han adaptado los algoritmos para no realizar operaciones con elementos nulos.

MATLAB ofrece una estrategia general para este tipo de matrices: el almacenamiento como matriz `sparse`, consistente en almacenar sólo los elementos no nulos, junto con su posición en la matriz, esto es, su fila y su columna.

Ejemplo 4.6 (Almacenamiento *sparse*)
En forma *sparse*, de la matriz 5×5

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{matrix}$$

se almacenarían sólo los siguientes datos

$$\begin{matrix} 2 & 1 & 1 \\ 3 & 5 & -1 \\ 5 & 3 & 3 \end{matrix}$$

Para crear matrices *sparse*, MATLAB tiene la función `sparse`, que se puede utilizar de varias formas.

La orden

```
AS = sparse(A)
```

convierte la matriz *llena* `A` en la matriz *sparse* `AS`.

La orden

```
A = full(AS)
```

hace la operación contraria, i.e., convierte la matriz *sparse* `AS` en la matriz *llena* `A`.

Ejemplo 4.7

```
>> A = [0, 0, 1; 0, 2, 0; -1, 0, 0];
>> AS = sparse(A)
AS =
   (3,1)       -1
   (2,2)        2
   (1,3)        1

>> A = full(AS)
A =
     0     0     1
     0     2     0
    -1     0     0
```

La orden

```
AS = sparse(i, j, s)
```

crea la matriz *sparse* `AS` cuyos elementos no nulos son `AS(i(k), j(k)) = s(k)`

Ejemplo 4.8

```
>> i = [2, 3, 5]
>> j = [1, 5, 3]
>> s = [1, -1, 3]
>> AS = sparse(i, j, s)
AS =
   (2,1)        1
   (5,3)        3
   (3,5)       -1
```

Las operaciones con matrices sparse funcionan, con MATLAB, con la misma sintaxis que para las matrices llenas. En general, las operaciones con matrices llenas producen matrices llenas y las operaciones con matrices *sparse* producen matrices *sparse*. Las operaciones mixtas generalmente producen matrices sparse, a menos que el resultado sea una matriz con alta densidad de elementos distintos de cero. Por ejemplo, con la misma matriz `AS` del ejemplo anterior:

Ejemplo 4.9

```
>> AS(5,3)
ans =
   (1,2)        1
   (5,3)       -1
   (3,5)        3

>> AS'
ans =
   (1,2)        1
   (5,3)       -1
   (3,5)        3

>> 10*AS
ans =
   (2,1)       10
   (5,3)       30
   (3,5)      -10

>> AS(3,5) = 7
AS =
   (2,1)        1
   (5,3)        3
   (3,5)        7

>> AS(1,4) = pi
AS =
   (2,1)       1.0000
   (5,3)       3.0000
   (1,4)       3.1416
   (3,5)       7.0000
```

5 Interpolación y ajuste de datos

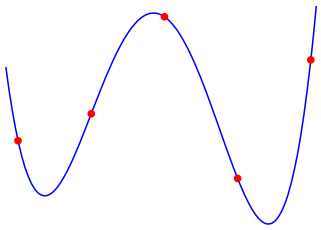
5.1 Introducción

En Física y otras ciencias con frecuencia es necesario trabajar con conjuntos discretos de valores de alguna magnitud que depende de otra variable. Pueden proceder de muestreos, de experimentos o incluso de cálculos numéricos previos.

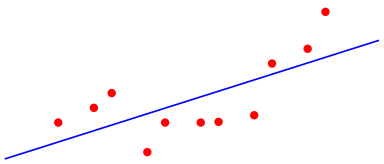
En ocasiones, para utilizar estos valores en cálculos posteriores es preciso «darles forma» de función, es decir: es preciso disponer de una función dada por una expresión matemática que «coincida» con dichos valores.

Existen básicamente dos enfoques para conseguir esto:

Interpolación es el proceso de determinar una función que tome exactamente los valores dados para los valores adecuados de la variable independiente, es decir que pase exactamente por unos puntos dados. Por ejemplo, determinar un polinomio de grado 4 que pase por 5 puntos dados, como en la figura de la derecha.



Ajuste de datos es el proceso de determinar la función, de un tipo determinado, que mejor se aproxime a los datos («mejor se ajuste»), es decir tal que la distancia a los puntos (medida de alguna manera) sea lo menor posible. Esta función no pasará necesariamente por los puntos dados. Por ejemplo, determinar un polinomio de grado 1 que aproxime lo mejor posible unos datos, como se muestra en la figura adjunta.



5.2 Interpolación polinómica global

Si la función a construir es un polinomio de un determinado grado, se habla de interpolación polinómica.

Interpolación lineal. Por dos puntos dados del plano pasa una sola línea recta.

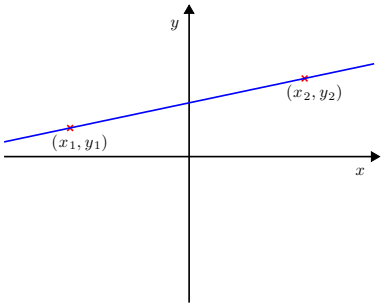
Más concretamente, dados dos puntos en el plano (x_1, y_1) y (x_2, y_2), con $x_1 \neq x_2$ se trata de determinar una función polinómica de grado 1

$$y = ax + b$$

que tome el valor y_1 para $x = x_1$ y el valor y_2 para $x = x_2$, es decir

$$\begin{cases} y_1 = ax_1 + b \\ y_2 = ax_2 + b \end{cases}$$

La solución de este sistema lineal de dos ecuaciones con dos incógnitas proporciona los valores adecuados de los coeficientes a y b.



Interpolación cuadrática. En general, tres puntos del plano determinan una única parábola (polinomio de grado 2).

Dados (x_1, y_1), (x_2, y_2) y (x_3, y_3) con x_1, x_2 y x_3 distintos dos a dos, se trata de determinar una función de la forma

$$y = ax^2 + bx + c$$

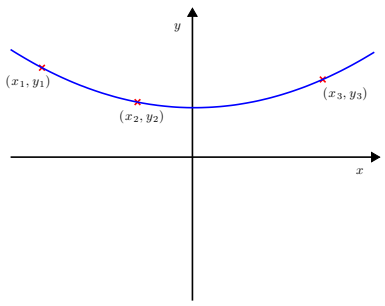
que pase por dichos puntos, es decir tal que

$$\begin{cases} y_1 = ax_1^2 + bx_1 + c \\ y_2 = ax_2^2 + bx_2 + c \\ y_3 = ax_3^2 + bx_3 + c \end{cases}$$

Este sistema lineal se escribe en forma matricial

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

y su solución (única) proporciona los coeficientes que determinan la función interpolante.



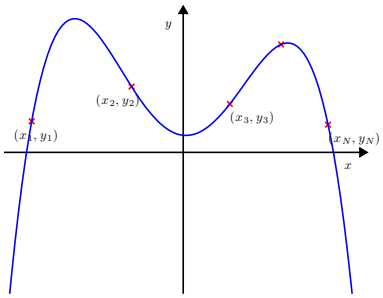
Interpolación global. En general, dados N puntos (x_k, y_k), $k = 1, \ldots, N$, con x_k todos distintos, existe un único polinomio de grado $N-1$ que pasa exactamente por estos puntos. Este polinomio se puede expresar de la forma

$$p(x) = c_1 x^{N-1} + c_2 x^{N-2} + \cdots + c_{N-1} x + c_N$$

y verifica que $p(x_k) = y_k$ para $k = 1, \ldots, N$, es decir:

$$\begin{cases} y_1 &= c_1 x_1^{N-1} + c_2 x_1^{N-2} + \cdots + c_{N-1} x_1 + c_N \\ y_2 &= c_1 x_2^{N-1} + c_2 x_2^{N-2} + \cdots + c_{N-1} x_2 + c_N \\ \cdots & \\ y_N &= c_1 x_N^{N-1} + c_2 x_N^{N-2} + \cdots + c_{N-1} x_N + c_N \end{cases}$$

Este procedimiento se conoce como interpolación global de Lagrange. [1]



Interpolación con MATLAB. Dados N puntos (x_k, y_k), $k = 1, 2, \ldots N$, con todos los x_k distintos, la instrucción

```
c=polyfit(x,y,N-1)
```

calcula los coeficientes del polinomio de grado `N-1` que pasa por los `N` puntos.

`x, y` son dos vectores de longitud `N` conteniendo respectivamente las abscisas y las ordenadas de los puntos.

El vector `c` devuelto por `polyfit` contiene los coeficientes del polinomio de interpolación:

$$p(x) = c_1 x^{N-1} + c_2 x^{N-2} + \cdots + c_{N-1} x + c_N.$$

Ejercicio 5.1 Calcular el polinomio de interpolación de grado 2 que pasa por los puntos

$$(1, -3),\ (2, 1),\ (3, 3)$$

Representar su gráfica en el intervalo $[0, 7]$, señalando con marcadores los puntos interpolados y dibujando también los ejes coordenados.

1. Crea dos vectores `x` e `y` conteniendo respectivamente las abscisas y las ordenadas de los puntos:

[1] Joseph Louis Lagrange (1736–1813), fue un matemático, físico y astrónomo italiano nacido en Turín.

```
x = [1,2,3];
y = [-3,1,3];
```

2. Calcula con la orden `polyfit` los coeficientes de la parábola que pasa por estos puntos:

```
c = polyfit(x,y,2);
```

   Obtendrás el resultado `c=[-1,7,-9]`, que significa que el polinomio de interpolación de grado 2 que buscamos es $p(x) = -x^2 + 7x - 9$.

3. Para dibujar la gráfica de esta función comenzamos por crear un vector `z` de puntos en el intervalo $[0, 9]$:

```
z = linspace(0,7);
```

4. Necesitamos ahora calcular los valores del polinomio $p(x)$ en todos estos puntos. Para ello usamos la función `polyval`:

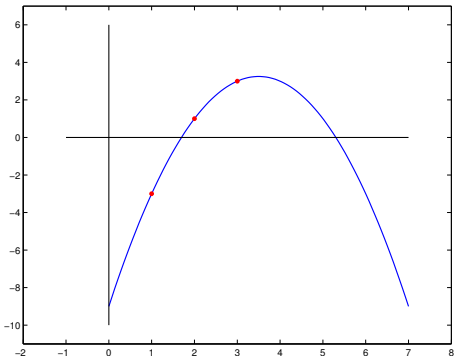
```
p = polyval(c,z);
```

5. Dibuja la parábola:

```
plot(z,p)
```

6. Añade ahora los ejes de coordenadas y los marcadores de los puntos del soporte de interpolación:

```
hold on
plot([-1,7],[0,0], 'k','LinWidth',1.1)
plot([0,0],[-10,6],'k','LineWidth',1.1)
plot(x,y,'r.','MarkerSize',15)
axis([-2,8,-11,7])
hold off
```



Ejercicio 5.2 La temperatura del aire cerca de la tierra depende de la concentración K del ácido carbónico (H_2CO_3) en él. En la tabla siguiente se recoge, para diferentes latitudes L sobre la tierra y para el valor de $K = 0.67$, la variación δ_K de la temperatura con respecto a una cierta temperatura de referencia:

L	65	35	5	-25	-55
δ_K	-3.1	-3.32	-3.02	-3.2	-3.25

Calcular y dibujar el polinomio de interpolación global de estos datos. Usando el polinomio de interpolación construido, calcular la variación de la temperatura para $L = 10$ (valor de la latitud que no está entre las mediciones de la tabla de datos).

1. Crea dos vectores con los datos de la tabla:

```
x = [-55,-25,5,35,65];
y = [-3.25,-3.2,-3.02,-3.32,-3.1];
```

2. Calcula y dibuja el polinomio de interpolación:

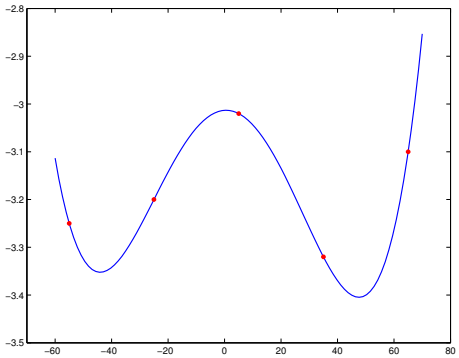
```
c = polyfit(x,y,4);
z = linspace(-60,70);
p = polyval(c,z);
plot(z,p,'Color',[0,0,1],'LineWidth',1.2)
```

3. Dibuja también marcadores de los puntos:

```
hold on
plot(x,y,'r.','MarkerSize',15)
```

4. Usa el polinomio que acabas de obtener para calcular δ_K para $L = 10$ e imprime su valor:

```
L = 10;
delta=polyval(c,10);
fprintf('Para L =%3i delta = %6.3f \n ',L, delta)
```

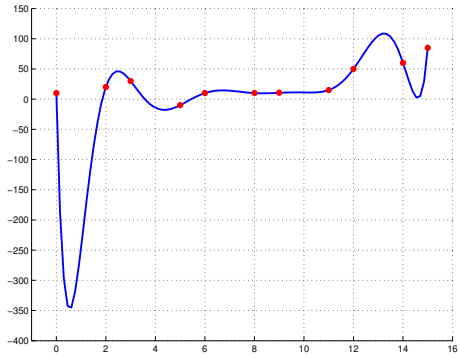


Ejercicio 5.3 (Propuesto) Calcula el polinomio de grado 10 que interpola los valores:

$$x = (0,\ 2,\ 3,\ 5,\ 6,\ 8,\ 9,\ 11,\ 12,\ 14,\ 15),$$

$$y = (10,\ 20,\ 30,\ -10,\ 10,\ 10,\ 10.5,\ 15,\ 50,\ 60,\ 85)$$

Dibuja su gráfica y observa las inestabilidades cerca de los extremos:



Este ejercicio pretende mostrar que el procedimiento de **interpolación global** es, en general inestable, ya que los polinomios tienden a hacerse oscilantes al aumentar su grado y eso puede producir grandes desviaciones sobre los datos.

Ejercicio 5.4 Calcula el polinomio de grado 5 que interpola los valores:

```
x=[0.01,2.35,1.67,3.04,2.35,1.53];
y=[6,5,4,3,2,3];
```

Recibirás un Warning y unos resultados.

Intenta comprender lo que dice el mensaje del Warning y analizar los resultados. ¿Ves algo raro? ¿Llegas a alguna conclusión?

1. El mensaje del Warning dice que el polinomio está mal condicionado. Que intentes añadir puntos con distintos valores de x, que reduzcas el grado del polinomio,... En resumidas cuentas, que los datos no son correctos.

2. Observa los valores que has obtenido para los coeficientes:

   ```
   c =
      1.0e+15 *
      -0.0588    0.5060   -1.5906    2.1713   -1.0956    0.0107
   ```

   Como verás son desmesurados (observa que están todos multiplicados por `1.0e+15`).

3. El motivo de todo esto es que, entre los datos que hemos usado, hay dos abscisas iguales a las que asignamos dos valores distintos: dos de los puntos que usamos están en la misma vertical. Lógicamente es **imposible** construir una **función** regular que pase por esos puntos.

Observación: El hecho de disponer de una herramiento tan potente como lo es MATLAB no nos exime de saber lo que estamos haciendo. Resulta imprescindible realizar un análisis previo que dé validez a lo que MATLAB calcula.

5.3 Interpolación lineal a trozos

Como se ha visto en el ejercicio 5.3, la interpolación polinómica global es inestable cuando el número de puntos es elevado. En el ejercicio se usan procedimientos de **interpolación a trozos**, que se explica en lo que sigue.

Consideramos N puntos (x_k, y_k), $k = 1, \ldots, N$, con los valores de x_k todos diferentes y ordenados en orden creciente o decreciente. Se llama **interpolante lineal a trozos** a la poligonal que sobre cada intervalo formado por dos valores de x consecutivos: $[x_k, x_{k+1}]$, $k = 1, \ldots, N-1$ está definida por el segmento que une los puntos (x_k, y_k) y (x_{k+1}, y_{k+1}), como en la Figura 5.1.

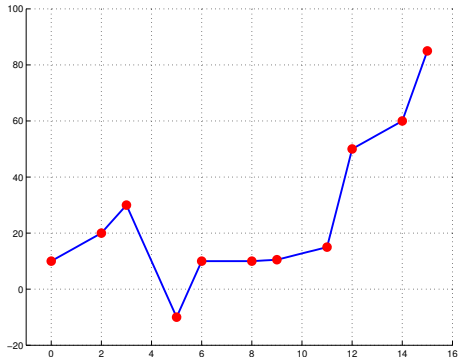


Figura 5.1: Interpolante lineal a trozos.

La instrucción MATLAB:

```
s1=interp1(x,y,z)
```

calcula el valor en el/los punto(s) `z` de la interpolante lineal a trozos que pasa por los puntos `(x,y)`.

`x y` son dos vectores de la misma dimensión que contienen las abscisas y las ordenadas de los puntos dados.

`z` son las abscisas de los puntos a interpolar, es decir, los puntos en los cuales queremos evaluar la interpolante. Puede ser una matriz.

`s1` son los valores calculados, es decir, los valores de la función interpolante en `z`. `s1` tendrá las mismas dimensiones que `z`

Ejercicio 5.5 Se consideran los mismos valores del ejercicio 5.3:

$$x = (0,\ 2,\ 3,\ 5,\ 6,\ 8,\ 9,\ 11,\ 12,\ 14,\ 15),$$
$$y = (10,\ 20,\ 30,\ -10,\ 10,\ 10,\ 10.5,\ 15,\ 50,\ 60,\ 85)$$

Calcula, a partir del polinomio de interpolación lineal a trozos, el valor interpolado para $x = 1$ y compáralo con el obtenido mediante un polinomio de interpolación global de grado 10 (el del ejercicio 5.3)

Dibuja juntas las gráficas del polinomio de interpolación lineal a trozos y del polinomio de interpolación de grado 10.

1. Crea dos variables `x` e `y` con las abscisas y las ordenadas que vas a interpolar:

```
x = [0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y = [10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Calcula a partir del polinomio de interpolación lineal a trozos el valor interpolado para $x = 1$:

```
s1 = interp1(x,y,1)
```

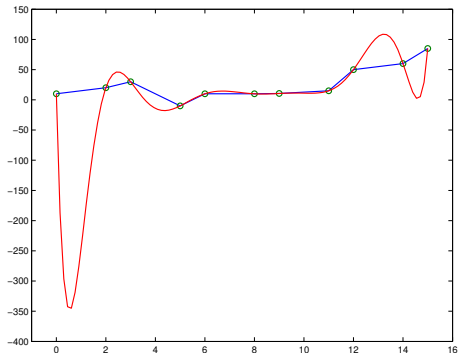
   Obtendrás el valor `s1=15`, mientras que mediante el polinomio de interpolación de grado 10 del ejercicio 3 se obtendría:

```
c = polyfit(x,y,10);
s = polyval(c,1)      % = -245.7853
```

3. Dibuja el polinomio de interpolación lineal a trozos y el polinomio de grado 10 junto con los puntos a interpolar:

```
t = linspace(0,15);
p = polyval(c,t);
plot(x,y,x,y,'o',t,p)
axis([-1,16,-400,150])
```

   Observa la gráfica y compara los valores los valores `s1` y `c1` ¿Qué puedes decir? ¿Cuál de los dos valores da una aproximación *a priori* más acertada?



Propuesta de ejercicio para valientes

Escribe tu propia M-función

```
function [yi] = interpol(xs,ys,xi)
```

que haga lo mismo que `interp1(xs,ys,xi)`.

5.4 Interpolación por funciones *spline*

Con frecuencia se necesita interpolar un conjunto de datos con funciones «suaves» (sin picos), como por ejemplo en la creación de gráficos por ordenador.

Obsérvese que la función de interpolación lineal a trozos es sólo continua y para obtener funciones «suaves» necesitamos que tengan al menos una derivada continua. Esto se consigue usando, por ejemplo, interpolación a trozos como la que hemos explicado antes, pero con polinomios cúbicos en vez de líneas rectas. Las funciones así construidas se conocen como **splines cúbicos**

En MATLAB podemos usar la instrucción

```
s = spline(x,y,z)
```

Los argumentos de entrada `x`, `y` y `z` tienen el mismo significado que en el comando `interp1` (ver la Sección 5.3), el argumento de salida almacena en el vector `s` los valores de la función spline en los puntos arbitrarios guardados en el vector `z`.

Ejercicio 5.6 Calcula el spline cúbico que interpola los datos del ejercicio 5.3. Dibuja en la misma ventana el spline calculado junto con el polinomo de interpolación lineal a trozos.

1. Crea dos variables `x` e `y` con las abscisas y las ordenadas que vas a interpolar:

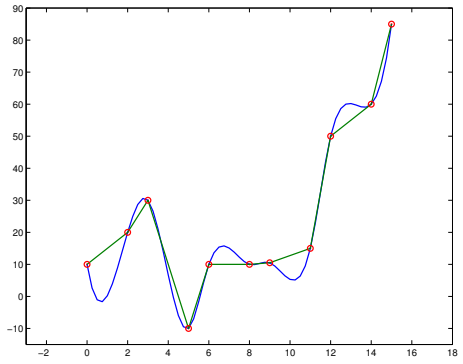
```
x=[0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y=[10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Calcula el spline cúbico

```
z=linspace(0,15);
s=spline(x,y,z);
```

3. Dibuja el spline cúbico y la interpolante lineal a trozos junto con los datos:

```
plot(x,y,'or',z,s,x,y,'LineWidth',2)
```



Ejercicio 5.7 El fichero `Datos7.dat` contiene una matriz con dos columnas, que corresponden a las abscisas y las ordenadas de una serie de datos.

Hay que leer los datos del fichero, y calcular y dibujar juntos el polinomio de interpolación global y el spline cúbico que interpolan dichos valores, en un intervalo que contenga todos los puntos del soporte.

1. Puesto que el fichero `Datos7.dat` es de texto, se puede editar. Ábrelo para ver su estructura. Cierra el fichero.

2. Puesto que todas las líneas del fichero tienen el mismo número de datos, se puede leer con la orden `load`:

   ```
   datos = load('Datos7.dat');
   ```

   que guarda en la variable `datos` una matriz 13×2 cuya primera columna contiene los valores de las abscisas y la segunda los de las ordenadas. Compruébalo.

3. Crea dos vectores `x` e `y` con 13 componentes para usarlos como abscisas y ordenadas respectivamente de los puntos de la tabla a interpolar:

   ```
   x=datos(:,1);
   y=datos(:,2);
   ```

4. Vamos a dibujar el spline en un intervalo que contenga todos los nodos:

   ```
   z = linspace(min(x),max(x));
   ```

   Calculamos los valores del spline en `z`:

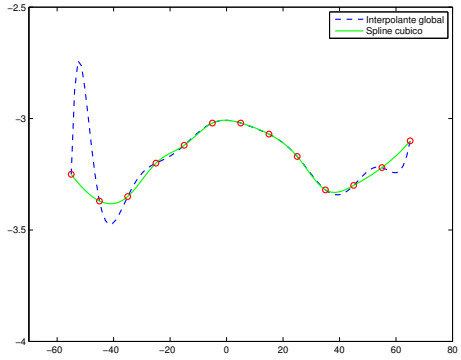
   ```
   s=spline(x,y,z);
   ```

5. Calculamos los coeficientes del polinomio de interpolación global y lo evaluamos en `z`:

   ```
   c=polyfit(x,y,length(x)-1);
   p=polyval(c,z);
   ```

6. Dibujamos las dos funciones y los puntos del soporte:

   ```
   plot(z,p,'b--',z,s,'g','LineWidth',2)
   legend('Interpolante global','Spline cubico')
   hold on
   plot(x,y,'or','LineWidth',1.2)
   axis([-70,80,-4,-2.5])
   hold off
   ```



Existen distintos tipos de splines que se diferencian en la forma que toman en los extremos. Para más información consulta en el Help de MATLAB.

Ejercicio 5.8 (Para ampliar conocimientos) (Para ampliar conocimientos) Cuando se calcula un spline cúbico con la función `spline` es posible cambiar la forma en que éste se comporta en los extremos. Para ello hay que añadir al vector `y` dos valores extra, uno al principio y otro al final. Estos valores sirven para imponer el valor de la pendiente del spline en el primer punto y en el último. El spline así construido se denomina *sujeto*.

Naturalmente, todos los procedimientos de interpolación antes explicados permiten aproximar funciones dadas: basta con interpolar un soporte de puntos construido con los valores exactos de una función.

En este ejrcicio se trata de calcular y dibujar una aproximación de la función sen(x) en el intervalo $[0, 10]$ mediante la interpolación con dos tipos distintos de spline cúbico y comparar estos resultados con la propia función. Hay por lo tanto que dibujar tres curvas en $[0, 10]$:

1. La curva $y = sen(x)$.
2. El spline que calcula MATLAB por defecto (denominado *not-a-knot*).
3. El spline *sujeto* con pendiente $= -1$ en $x = 0$ y pendiente $= 5$ en $x = 10$.

Observación: Este ejercicio no es imprescindible.

1. Construimos un conjunto de nodos para la interpolación en $[0, 10]$ y calculamos los valores en estos nodos de la función $sen(x)$. Por ejemplo:

```
x = 0:10;
y = sin(x);
```

   Estos vectores `x` e `y` van a ser los puntos soporte para la construcción de los splines.

2. Lo que queremos es dibujar los dos splines y la función. Para ello construimos un vector de puntos en el intervalo $[0, 10]$ que utilizaremos para las gráficas, calculando en ellos los valores de las tres funciones:

```
z=linspace(0,10);
```

3. Calculamos el valor en estos puntos z del spline *not-a-knot*:

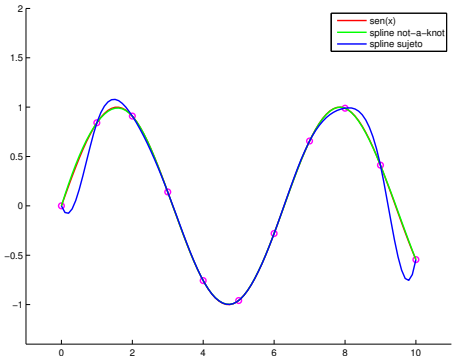
```
s1=spline(x,y,z);
```

4. Calculamos ahora el valor del spline *sujeto*. Para ello añadimos al vector y los valores -1 y 5 al principio y al final respectivamente:

```
ys=[-1, y, 5];          %  o tambien simplemente
s2=spline(x,ys,z);      %  s2 = spline(x,[-1,y,5],z);
```

5. Ahora dibujamos las tres curvas y los puntos del soporte

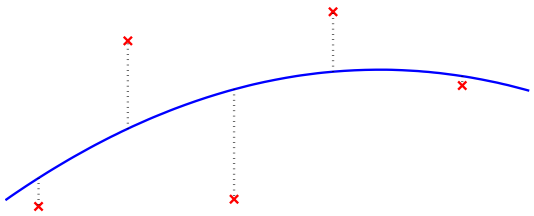
```
axis([-1,11,-1.4,2]);
hold on
plot(x,y,'mo')
h1 = plot(z,sin(z),'Color',[1,0,0]);
h2 = plot(z,s1,    'Color',[0,1,0]);
h3 = plot(z,s2,    'Color',[0,0,1]);
legend([h1,h2,h3],'sen(x)','spline not-a-knot','spline sujeto')
hold off
```



5.5 Ajuste de datos

La técnica de interpolación que hemos explicado antes requiere que la función que interpola los datos pase exactamente por los mismos. En ocasiones esto no da resultados muy satisfactorios, por ejemplo si se trata de muchos datos. También sucede con frecuencia que los datos vienen afectados de algún error, por ejemplo porque provienen de mediciones. No tiene mucho sentido, pues, obligar a la función que se quiere construir a «pasar» por unos puntos que ya de por sí no son exactos.

Otro enfoque diferente es construir una función que no toma exactamente los valores dados, sino que «se les parece» lo más posible, por ejemplo minimizando el error, medido éste de alguna manera.



Cuando lo que se minimiza es la suma de las distancias de los puntos a la curva (medidas como se muestra en la figura) hablamos de **ajuste por mínimos cuadrados**. La descripción detallada de este método se escapa de los objetivos de este tema. Veremos solamente cómo se puede hacer esto con MATLAB en algunos casos sencillos.

5.5.1 Ajuste por polinomios

La función `polyfit` usada ya para calcular el polinomio de interpolación global sirve también para ajustar unos datos por un polinomio de grado dado:

```
c=polyfit(x,y,m)
```

`x y` son dos vectores de la misma dimensión que contienen respectivamente las abscisas y las ordenadas de los N puntos.

`m` es el grado del polinomio de ajuste deseado

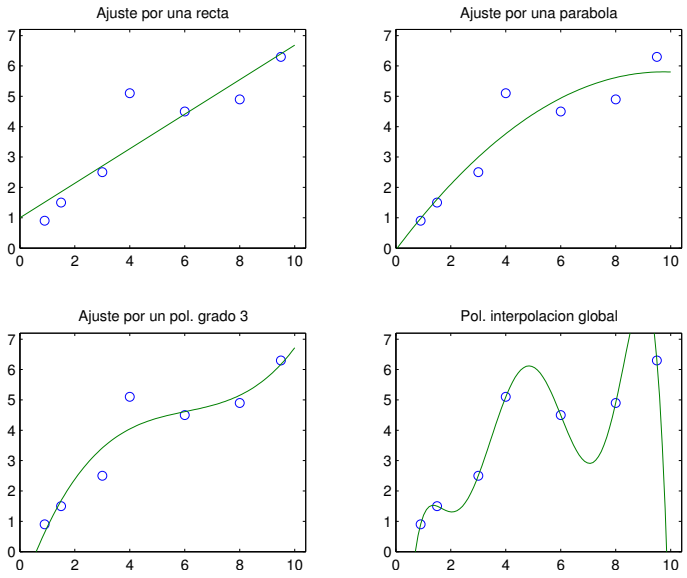
`c` es el vector con los coeficientes del polinomio de ajuste

Si $m = 1$, el polinomio resultante es una recta, conocida con el nombre de **recta de regresión**, si $m = 2$ es una parábola, y así sucesivamente. Naturalmente, cuando $m = N - 1$ el polinomio calculado es el polinomio de interpolación global de grado $N - 1$ que pasa por todos los puntos.

Ejercicio 5.9 Calcula y dibuja los polinomios de ajuste de grado 1, 2, 3 y 6 para los siguientes datos:

$$(0.9, 0.9) \quad (1.5, 1.5) \quad (3, 2.5) \quad (4, 5.1) \quad (6, 4.5) \quad (8, 4.9) \quad (9.5, 6.3)$$

Una vez calculados, escribe las expresiones analíticas de los polinomios que has obtenido.



1. Construye los vectores `x` e `y` a partir de los datos:

```
x=[0.9, 1.5, 3,   4,   6,   8,   9.5];
y=[0.9, 1.5, 2.5, 5.1, 4.5, 4.9, 6.3];
```

2. Calcula la recta de regresión e imprime los coeficientes calculados:

```
p1=polyfit(x,y,1);
```

   Obtendrás los valores: `0.57` y `1.00`. La recta de regresión es $y = 0.57x + 1$

3. Dibuja la recta de regresión (recuerda que para dibujar una recta bastan dos puntos):

```
subplot(2,2,1)
plot(x,y,'o',[0,10],[polyval(p1,0),polyval(p1,10)])
title('Ajuste por una recta')
```

4. Calcula ahora la parábola que ajusta los datos:

```
p2=polyfit(x,y,2);
```

   Obtendrás los valores `-0.0617`, `1.2030` y `-0.0580`. La parábola es por tanto: $y = -0.062\,x^2 + 1.2\,x - 0.06$ (redondeando a dos decimales).

5. Dibuja la parábola

```
xp=linspace(0,10);
subplot(2,2,2)
plot(x,y,'o',xp,polyval(p2,xp));
title('Ajuste por una parabola')
```

6. Termina el ejercicio construyendo y dibujando los polinomimios de grado 3 y 6. Observa que el polinomio de grado 6 es el polinomio de interpolación que pasa por todos los puntos.

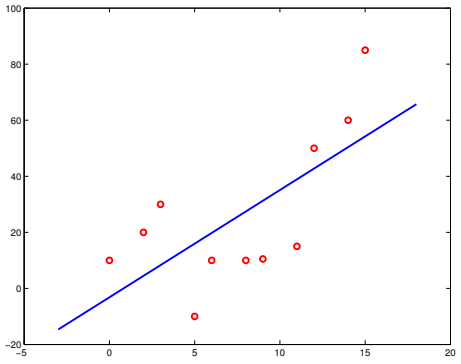
Al realizar este ejercicio dibujando cada curva en un cuadro distinto, como se hace aquí, debes tener cuidado de fijar en cada `subplot` los mismos ejes con el comando `axis`, para poder comparar bien. Si no lo haces los resultados te pueden resultar confusos.

Ejercicio 5.10 Calcula y representa gráficamente la recta de regresión asociada a los siguientes datos:

$$x = (0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15)$$

$$y = (10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85)$$

```
x = [0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y = [10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
axis([-5,20,-20,100]);
hold on
plot(x,y,'ro')
c = polyfit(x,y,1);
z = [-5,20];
yz = polyval(c,z);
plot(z,yz)
hold off
```



Ejercicio 5.11 En el fichero `Finanzas.dat` está recogido el precio de una determinada acción de la Bolsa española a lo largo de 88 días, tomado al cierre de cada día.

Queremos ajustar estos datos por una función que nos permita predecir el precio de la acción para un corto intervalo de tiempo más allá de la última cotización.

Lee los datos del fichero y represéntalos. Calcula los polinomios de ajuste de grados 1, 2 y 4 y represéntalos también.

1. El fichero `Finanzas.dat` contiene una sola columna con los 88 datos.
2. Comienza por cargar en la memoria los datos del fichero y luego crea un vector con los números de los 88 días:

```
A  = load('Finanzas.dat');
dd = (1:88)';
```

   ¿Porqué transponemos el vector `dd`?

3. Dibujamos los datos directamente (recuerda que lo que resulta así es la interpolante lineal a trozos):

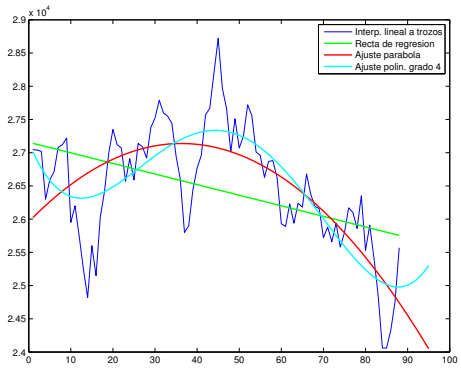
```
h0 = plot(dd,A)
hold on
```

4. Calcula los coeficientes de los polinomios de ajuste de grado 1, 2 y 4:

```
c1 = polyfit(dd,A,1);
c2 = polyfit(dd,A,2);
c4 = polyfit(dd,A,4);
```

5. Para dibujar todos los polinomios vamos a construir un soporte un poco más amplio, que nos permita observar la predicción:

```
d  = 1:95;
s1 = [polyval(c1,1),polyval(c1,95)]];
s2 = polyval(c2,d);
s4 = polyval(c4,d);

h1  = plot([1,88],s1,'r');
h2  = plot(d,s2,'g');
h3  = plot(d,s4,'c');
```



5.5.2 Otras curvas de ajuste mediante cambios de variable

En ocasiones hace falta usar funciones distintas a las polinómicas para ajustar datos. Desde el punto de vista teórico se puede utilizar cualquier función. En general, el planteamiento teórico y la resolución del problema es estos casos es muy distinta de lo presentado aquí.

Sin embargo, en algunos casos sencillos, se puede reducir a un problema lineal mediante un cambio de variables. Entonces se calcula la recta de regresión para las nuevas variables y luego se deshace el cambio. Las que se utilizan habitualmente son las siguientes:

Función potencial: $y = b\,x^m$ Se trata de encontrar b y m de forma que la función $y = b\,x^m$ se ajuste lo mejor posible a unos datos. Observando que

$$y = bx^m \quad \Leftrightarrow \quad \ln(y) = \ln(b) + m\ln(x)$$

se ve que se pueden encontrar $\ln(b)$ y m ajustando los datos $\ln(x)$, $\ln(y)$ mediante un polinomio de grado 1.

Función exponencial: $y = be^{mx}$ Nuevamente, tomando logaritmos se tiene:

$$y = be^{mx} \quad \Leftrightarrow \quad \ln(y) = \ln(b) + mx$$

de donde se pueden encontrar $\ln(b)$ y m ajustando los datos x y $\ln(y)$ mediante una recta.

Función logarítmica: $y = m\ln(x) + b$ Está claro que hay que ajustar los datos $\ln(x)$ e y mediante una recta.

Función hiperbólica $y = \dfrac{1}{mx+b}$ La anterior relación se puede escribir también:

$$y = \frac{1}{mx+b} \quad \Leftrightarrow \quad mx + b = \frac{1}{y}$$

lo que muestra que x y $\dfrac{1}{y}$ se relacionan linealmente. Por lo tanto se pueden calcular m y b ajustando x y $\dfrac{1}{y}$ mediante una recta de regresión.

Ejercicio 5.12 Ajustar los datos

x	2.0	2.6	3.2	3.8	4.4	5.0
y	0.357	0.400	0.591	0.609	0.633	0.580

mediante una función potencial $y = bx^m$ y dibujar la función obtenida así como los datos.

Puesto que, si los x e y son positivos,

$$y = b\,x^m \quad \Leftrightarrow \quad \ln(y) = \ln(b) + m\ln(x)$$

podemos calcular la recta de regresión $y = \alpha x + \beta$ para los datos $(\ln(x), \ln(y))$ y luego tomar $b = e^\beta$ y $m = \alpha$.

1. Creamos los dos vectores:

```
x = [2.0, 2.6, 3.2, 3.8, 4.4, 5.0];
y = [0.357 , 0.400 , 0.591 , 0.609 , 0.633 , 0.580];
```

2. Calculamos los coeficientes de la recta de regresión:

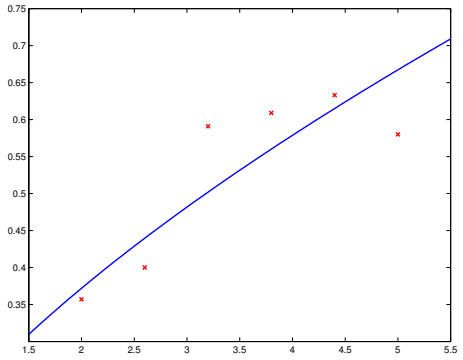
```
c = polyfit(log(x),log(y),1);
```

Obtendrás `c = [0.6377, -1.4310]`, lo que significa, por lo dicho antes, que la función que buscamos es:

$$y = e^{-1.4310} \cdot x^{0.6377} = 0.2391 \cdot x^{0.6377}$$

3. Dibujamos ahora la función obtenida y los datos:

```
plot(x,y,'rx','LineWidth',1.5)
hold on

xs = linspace(1.5,5.5);
ys = exp(c(2)) * xs.^(c(1));
plot(xs,ys,'LineWidth',1.5)
hold off
```



Ejercicio 5.13 Ajustar los datos contenidos en el fichero `datos13.dat` mediante una función exponencial $y = b\,e^{mx}$ y dibujar la función obtenida así como los datos.

1. Leemos los datos del fichero

```
data = load('datos13.dat');     % data = matriz 50 x 2
```

2. Recuperamos `x` e `y`:

```
x = data(:,1);
y = data(:,2);
```

3. Calculamos los coeficientes de la recta de regresión: $\ln(y) = \ln(b) + mx$

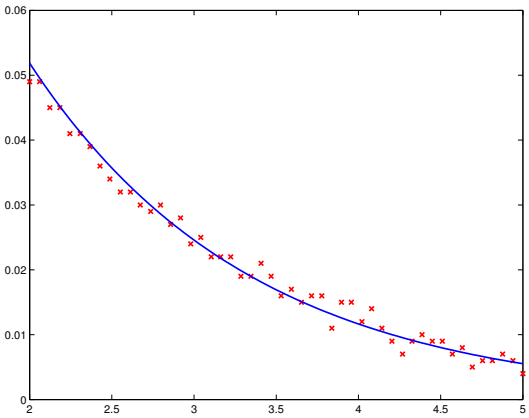
```
c = polyfit(x,log(y),1);
```

Obtendrás `c = [-0.7464, -1.4665]`, lo que significa, por lo dicho antes, que la función que buscamos es:

$$y = e^{-1.4665}\, e^{-0.7464\, x} = 0.2307\, e^{-0.7464\, x}$$

4. Dibujamos ahora la función obtenida y los datos:

```
plot(x,y,'rx','LineWidth',1.5)
hold on

xs = linspace(min(x),max(x));
m  = c(1);
b  = exp(c(2));
ys = b * exp(m*xs);
plot(xs,ys,'LineWidth',1.5)
hold off
```



Ejercicio 5.14 Determinar una función, de las indicadas antes, que se ajuste lo mejor posible a los datos de la siguiente tabla:

x	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
y	6	4.83	3.7	3.15	2.41	1.83	1.49	1.21	0.96	0.73	0.64

Escribe la expresión de la función que has calculado.

1. En primer lugar representamos los datos que se quieren ajustar.

```
x=0:0.5:5;
y=[6, 4.83, 3.7, 3.15, 2.41, 1.83, 1.49, 1.21, 0.96, 0.73, 0.64];
plot(x,y,'rx','LineWidth',1.2)
```

Observa, mirando la gráfica, que una función lineal no proporcionaría el mejor ajuste porque los puntos claramente no siguen una línea recta. De las restantes funciones, la logarítmica también se excluye, ya que el primer punto es $x = 0$. Lo mismo sucede con la función potencial ya que ésta se anula en $x = 0$ y nuestros datos no.

Vamos a realizar el ajuste con las funciones exponencial $y = be^{mx}$ e hiperbólica $y = \dfrac{1}{mx+b}$. A la vista de la gráfica que vamos a obtener veremos cuál de ellas se ajusta mejor a los datos.

2. Comienza por calcular la función exponencial $y = be^{mx}$. Utiliza la función `polyfit` con `x` y `log(y)` para calcular los coeficientes b y m:

```
c=polyfit(x,log(y),1);
m1=c(1);
b1=exp(c(2));
```

Escribe la expresión de la función que has calculado.

3. Determina ahora la función hiperbólica $y = 1/(mx+b)$, utilizando de nuevo `polyfit` con `x` y `1./y`:

```
p=polyfit(x,1./y,1);
m2=p(1);
b2=p(2);
```

Escribe la expresión de la función que has calculado.

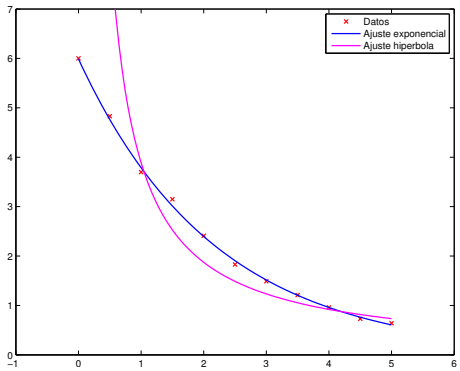
4. Para dibujar las dos funciones escribimos:

```
t1=linspace(0,5);
t2=linspace(0.1,5)
y1=b1*exp(m1*t1);
y2=1./(m2*t2+b2);
```

¿Porqué se ha creado el vector `t2`? ¿Es necesario hacerlo?

5. Realizamos la representación gráfica de ambas funciones

```
axis([-1,6,0,7])
plot(t1,y1,t2,y2)
legend('Datos','Ajuste por exponencial','Ajuste por hipérbola')
```



A la vista de la gráfica obtenida, ¿cuál de las dos funciones se ajusta mejor a los datos?

6 Resolución de ecuaciones no lineales

6.1 Introducción

Dada $f : [a, b] \subset \mathbb{R} \mapsto \mathbb{R}$, continua, se plantea el problema de encontrar soluciones de la ecuación

$$f(x) = 0. \tag{6.1}$$

A las soluciones de esta ecuación, también se les suele llamar **ceros** de la función f, por ser los puntos en los que $f = 0$.

Desde el punto de vista geométrico, las soluciones de la ecuación 6.1 son los puntos en los que la gráfica de la función f «toca» al eje OX, aunque no necesariamente lo atraviesa (véanse las figuras)

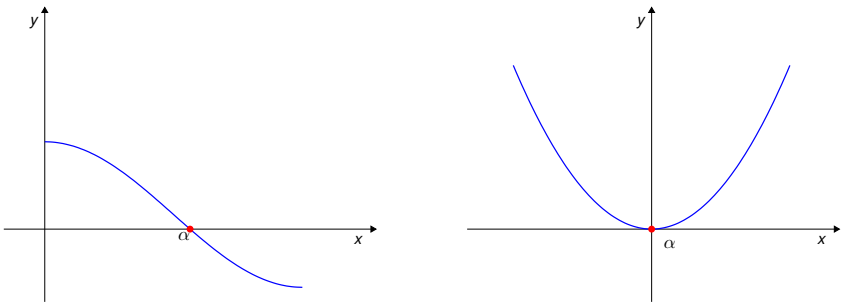


Figura 6.1: La gráfica corta al eje de abscisas en el punto α, lo que significa que α es un cero de la función f, aunque no en ambos casos la función cambia de signo en α.

En Física surgen frecuentemente problemas que conducen a ecuaciones del tipo (6.1) cuyas soluciones no se pueden calcular explícitamente.

Por ejemplo, la ecuación

$$e^x + e^{-x} = \frac{2}{\cos x}$$

que aparece, por ejemplo, cuando se quieren determinar las frecuencias de las oscilaciones transversales de una viga con extremos empotrados y sometida a un golpe. Las soluciones de esta ecuación no se pueden calcular por métodos analíticos.

Incluso para ecuaciones tan aparentemente sencillas como las polinómicas

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0 \tag{6.2}$$

es bien conocido que para $n \geq 5$, no existe una fórmula explícita de sus soluciones.

6.2 Resolución de ecuaciones polinómicas

En el caso de ecuaciones polinómicas, es posible calcular, de una vez, todas sus soluciones, gracias al comando, ya conocido,

```
s = roots(p)
```

que calcula todas las raíces (reales y complejas) de un polinomio y puede ser utilizado para calcular las soluciones reales de la ecuación 6.2.

Ejercicio 6.1 Calcular las soluciones de la ecuación polinómica:

$$x^3 - 9x^2 - x + 5 = 0.$$

1. Comenzamos introduciendo el polinomio p que aparece en el primer miembro de la ecuación homogénea anterior. Recuerda que se introduce como un vector fila cuyas componentes son los coeficientes del polinomio, ordenados de mayor a menor grado. En este caso

   ```
   p=[1,-9,-1,5];
   ```

   Recuerda que hay que incluir los coeficientes nulos, si los hay.

2. Calculamos las raíces

   ```
   roots(p)
   ```

   obtendras las raíces: `9.0494`, `-0.7685` y `0.7190` que, puesto que son todas reales, son las soluciones de la ecuación.

Recuerda también, siempre, que estamos realizando cálculos numéricos con el ordenador y que, por consiguiente, **todo** lo que calculemos es **aproximado**.

Ejercicio 6.2 Calcular las soluciones de la ecuación polinómica:

$$2x^2(x+2) = -1.$$

1. Comenzamos por escribir la ecuación en forma homogénea (con segundo miembro cero) y desarrollar el polinomio, para disponer de todos sus coeficientes:

   $$2x^2(x+2) = -1 \iff 2x^3 + 4x^2 + 1 = 0$$

2. Los coeficientes del polinomio son:

   ```
   p=[2,4,0,1];
   ```

3. Calculamos las raíces

```
roots(p)
```

obtendras las raíces: `-2.1121`, `0.0560 + 0.4833i` y `0.0560 - 0.4833i`. Por consiguiente, en el campo real, la única solución de la ecuación es $x = -2.1121$.

Ejercicio 6.3 Se quiere determinar el volumen V ocupado por un gas a temperatura T y presión p a partir de la ecuación de estado

$$\left[p + a\left(\frac{N}{V}\right)^2\right](V - Nb) = kNT,$$

siendo N el número de moleculas, k la constante de Boltzmann y a y b constantes que dependen del gas.

Escribir una M-función

```
function [V]=volumen(T,p,N,k,a,b)
```

que, a partir de los valores de T, p, N, k, a y b, calcule la solución de la ecuación polinómica

$$f(V) = pV^3 - (pbN + kNT)V^2 + aN^2V - abN^3 = 0$$

(obtenida de la ecuación de estado multiplicando por V^2) y realice una gráfica de la función $f(V)$ en un intervalo adecuado, marcando en ella la solución.

Datos para comprobación: Para el dióxido de carbono (CO_2) se tiene $k = 1.3806503 \cdot 10^{-23}$ Joule/K, $a = 0.401$ Pa m^6 y $b = 42.7 \cdot 10^{-6}$ m^3.

El volumen ocupado por 1000 moléculas de CO_2 a temperatura $T = 300$ K y presión $p = 3.5 \cdot 10^7$ Pa debe ser $V = 0.0427$.

```
function [V]=volumen(T,p,N,k,a,b)
%-------------------------------------------------------------
%   Calculo del Volumen (V) ocupado por
%   N  moleculas de un gas a
%   T  temperatura y
%   p  presion
%-------------------------------------------------------------
%   V es la solucion de la ecuacion polinomica
%   c1*V^3 + c2*V^2 + c3*V + c4 = 0
%   donde
%   c1 = p;
%   c2 = -(p*b*N+k*N*T);
%   c3 = a*N*N;
%   c4 = -c3*b*N;
%-------------------------------------------------------------
%   Argumentos de entrada
```

```
%    T = temperatura
%    p = presion
%    N = numero de moleculas
%    k = constante de Boltzmann
%    a y b = constantes del gas
%-------------------------------------------------------------
%   Datos para comprobacion
%    T = 300 K
%    p = 3.5e7 Pa
%    N = 1000
%    k = 1.3806503e-23 Joule/K
%    a = 0.401 Pa m6
%    b = 42.7e-6 m3
%-------------------------------------------------------------
% Coeficientes del polinomio f(V)=c1*V^3 + c2*V^2 + c3*V + c4
c1 = p;
c2 = -(p*b+k*T)*N;
c3 = a*N*N;
c4 = -a*b*N*N*N;
c  = [c1,c2,c3,c4];

%  raices del polinomio
sol = roots(c);

%  seleccion de las raices reales
V = [];
for j = 1:3
    if(isreal(sol(j)))
        V = [V,sol(j)];
    end
end

% grafica de la función f(V)
z = linspace(-0.1,0.1);
vz = polyval(c,z);
plot(z,vz,'LineWidth',1.2)
hold on

%  solucion
plot(V,polyval(c,V),'ro','LineWidth',1.2)

%  ejes coordenados
plot([-0.1,0.1],[0,0],'k')
plot([0,0],[-1.e5,1.e5],'k')

%  parte visible
axis([-0.1,0.1,-1.e5,1.e5])
hold off
shg
```

6.3 El comando fzero

Los algoritmos numéricos para resolver el problema

$$\text{Hallar } x \in [a,b] \text{ tal que } f(x) = 0 \tag{6.3}$$

cuando la ecuación $f(x) = 0$ es no lineal (en el caso lineal su resolución es inmediata) son en general **algoritmos iterados**.

Esto significa que, a partir de un punto o intervalo iniciales (dependiendo del algoritmo), se construye una sucesión de aproximaciones (calculadas una a partir de la anterior) cuyo límite es la solución buscada. Estos algoritmos, cuando convergen, lo hacen a **una** de las soluciones de la ecuación. Si ésta tuviera más de una solución, habría que utilizar el algoritmo una vez para cada una, cambiando el punto inicial.

En la práctica, lógicamente, sólo se efectúan un número finito de iteraciones y el algoritmo se detiene cuando se verifica algún criterio, previamente establecido.

Para resolver el Problema 6.3, MATLAB dispone de la función

```
solucion=fzero(funcion,xcero)
```

donde

`funcion` es un manejador de la función que define la ecuación, f. Puede ser el nombre de una **función anónima** dependiente de una sola variable, o también un manejador de una **M-función**, en cuyo caso se escribiría `@funcion`. Ver los ejemplos a continuación.

`xcero` es un valor «cercano» a la solución, a partir del cual el algoritmo iterado de búsqueda de la solución comenzará a trabajar.

`solucion` es el valor (aproximado) de la solución encontrado por el algoritmo.

Ejercicio 6.4 La ecuación:

$$x + \ln\left(\frac{x}{3}\right) = 0$$

tiene una solución cerca de $x = 1$. Calcularla.

1. Comienza por definir una función anónima que evalúe la expresión del primer miembro:

   ```
   fun = @(x)  x + log(x/3);
   ```

2. A continuación usa el comando `fzero` tomando `x=1` como valor inicial:

   ```
   fzero(fun,1)
   ```

   obtendrás el resultado `1.0499`.

Tambien podríamos haber usado una M-función para definir la función, en lugar de una función anónima. Mostramos a continuación cómo se haría.

1. Escribimos, en un fichero de nombre `mifuncion.m`, las órdenes siguientes:

```
function [y] = mifuncion(x)
y = x + log(x/3);
```

   Salvamos el fichero y lo cerramos. Recuerda que el fichero tiene que llamarse igual que la M-función.

2. En la ventana de comandos, para calcular el cero de la función escribimos:

```
fzero(@mifuncion,1)
```

Como se ha dicho antes, los algoritmos de aproximación de raíces de ecuaciones no lineales necesitan que el punto inicial que se tome esté «cerca» de la solución. ¿Cómo de cerca? Esta pregunta no tiene una respuesta fácil.

Hay condiciones matemáticas que garantizan la convergencia del algoritmo si el punto inicial se toma en un entorno adecuado de la solución. Estas condiciones no son inmediatas de comprobar y requieren un análisis de cada caso.

El ejemplo siguiente pone de manifiesto la necesidad de elegir un punto cercano a la solución.

Ejercicio 6.5 Analizar la influencia del punto inicial en la convergencia de `fzero` al aproximar la solución de

$$x + \ln\left(\frac{x}{3}\right) = 0$$

1. Ya en la práctica anterior se ha definido la función anónima y se ha comprobado la eficacia de `fzero` eligiendo $x = 1$ como punto inicial:

```
fun = @(x)  x + log(x/3);
fzero(fun,1)
```

2. Prueba ahora eligiendo un punto inicial más alejado de la solución, por ejemplo:

```
fzero(fun,15)
```

   Recibirás un mensaje diciendo que se aborta la búsqueda de un cero porque durante la búsqueda se han encontrado valores complejos de la función (la función logaritmo no está definida para argumentos negativos en el campo real, pero sí lo está en el campo complejo). En la búsqueda de un intervalo que contenga un cambio de signo de la función, el algoritmo se ha topado con valores de x negativos, en los que la función `log` toma valores complejos.

El algoritmo utilizado por `fzero` comienza por «localizar» un intervalo en el que la función cambie de signo. No funcionará, pues, con ceros en los que no suceda esto, como pasa en el ejemplo siguiente:

Ejercicio 6.6 Utiliza el comando `fzero` para aproximar la solución de la ecuación

$$x^2 = 0$$

Comprobarás que no funciona, sea cual sea el punto inicial.

6.4 Gráficas para localizar las raíces y elegir el punto inicial

La determinación teórica de un punto inicial cercano a la solución puede ser una tarea difícil. Sin embargo, en muchos casos, un estudio gráfico previo puede resultar de gran ayuda.

Ejercicio 6.7 Calcular, si existe, una solución positiva de la ecuación

$$\operatorname{sen}(x) - 2\cos(2x) = 2 - x^2$$

determinando un punto inicial a partir de la gráfica de la función.

1. Comienza por escribir la ecuación en forma homogénea:

$$\operatorname{sen}(x) - 2\cos(2x) + x^2 - 2 = 0$$

2. A continuación, representa gráficamente la función.

```
fun = @(x)  sin(x) -2*cos(2*x) + x.^2 - 2;
x = linspace(-5,5);
plot(x,fun(x));
```

   La gráfica te mostrará que esta función tiene dos ceros: uno positivo cerca de $x = 1$ y otro negativo cerca de $x = -1$.

3. Utiliza ahora `fzero` para intentar aproximar el cero positivo:

```
fzero(fun,1)
```

   Obtendrás la solución `x = 0.8924`

Como se ha dicho antes, en los casos en que la ecuación tenga varias soluciones, habrá que utilizar `fzero` una vez para cada solución que interese calcular.

Ejercicio 6.8 Calcular las soluciones de la ecuación

$$\operatorname{sen}\left(\frac{x}{2}\right)\cos\left(\sqrt{x}\right) = \frac{1}{5} \quad \text{en } [0, \pi].$$

1. El primer paso es escribir la ecuación en forma homogénea ($f(x) = 0$). Por ejemplo

$$f(x) = \text{sen}\left(\frac{x}{2}\right)\cos\left(\sqrt{x}\right) - \frac{1}{5} = 0$$

2. Comienza por definir una función anónima y hacer la gráfica:

```
fun = @(x)  sin(x/2).*cos(sqrt(x))- 0.2;
x = linspace(0,pi);
plot(x,fun(x));
grid on
```

   Comprobarás que hay una solución cerca de $x = 0.5$ y otra cerca de $x = 1.5$.

3. Utiliza ahora `fzero` para intentar aproximar cada una de ellas:

```
fzero(fun,0.5)          %  sol. x = 0.5490
fzero(fun,1.5)          %  sol. x = 1.6904
```

Ejercicio 6.9 Estudiar el número de soluciones de la ecuación

$$(x^2 - 1)e^{x/2} = \frac{(x+2)^2}{10} - 1$$

en el intervalo $[-10, 2]$ y calcularlas.

1. Comienza por escribir la ecuación en forma homogénea:

$$(x^2 - 1)e^{x/2} - \frac{(x+2)^2}{10} + 1 = 0 \iff 10\,(x^2 - 1)e^{x/2} - (x+2)^2 + 10 = 0$$

2. Define ahora una función anónima para el primer miembro de la última ecuación y dibuja su gráfica en el intervalo $[-10, 2]$:

```
fun = @(x) 10*(x.^2-1).*exp(x/2)-(x+2).^2+10;
x = linspace(-10,2);
plot(x,fun(x))
grid on
```

   Podrás observar que la curva corta al eje OX en tres puntos, cuyas abscisas son la soluciones de la ecuación. Uno de ellos está en el intervalo $[-8, -6]$, otro en el intervalo $[-2, 0]$ y otro en $[0, 2]$.

3. Calcula ahora cada una de las raíces utilizando `fzero` con un punto inicial adecuado:

```
x1 = fzero(fun,-6);          %  x1 = -6.9611
x2 = fzero(fun,-2);          %  x2 = -0.3569
x3 = fzero(fun, 2);          %  x3 =  0.9612
```

Cuando se utiliza la gráfica por ordenador para «localizar» las soluciones de una ecuación es preciso prestar especial atención a los factores de escala de los ejes en el dibujo y hacer sucesivos dibujos «acercándose» a determinadas zonas de la gráfica para comprender la situación.

También será necesario un análisis teórico para no caer en equivocaciones.

Ejercicio 6.10 Calcular todas las soluciones de la ecuación

$$\frac{\ln(x+1)}{x^2+1} = x^2 - 8x + 6.$$

1. Comienza por escribir la ecuación en forma homogénea:

$$\frac{\ln(x+1)}{x^2+1} = x^2 - 8x + 6 \iff f(x) = \ln(x+1) - (x^2 - 8x + 6)(x^2+1) = 0$$

   Puesto que no tenemos ninguna indicación del intervalo en el que pueden estar las soluciones, lo primero que hay que observar es que el dominio de definición de la función $f(x)$ es $(-1, +\infty)$.

   También conviene analizar cuál es el comportamiento de f en los extremos de este intervalo:

$$\lim_{x\to(-1)^+} f(x) = -\infty \qquad \lim_{x\to+\infty} f(x) = -\infty$$

   Parece adecuado, pues, comenzar dibujando la gráfica de la función en un intervalo de la forma $[-1, M]$ con M «grande».

   Observación: la función $ln(x+1)$ no está definida para $x = -1$. El cálculo, con MATLAB, de `log(0)` devuelve el valor `-Inf`. La función `plot` ignora los valores `Inf` y `NaN` que aparezcan en el vector `y` de las ordenadas. Por lo tanto, la inclusión del valor $x = -1$ en el intervalo de dibujo no producirá error. Simplemente MATLAB lo ignorará.

2. Define una función anónima para f y dibuja su gráfica en el intervalo $[-1, 20]$ (por ejemplo)

```
fun = @(x)   log(x+1) - (x.^2-8*x+6).*(x.^2+1);
x = linspace(-1,20);
plot(x,fun(x))
grid on
```

   Resulta obvio, a la vista de la gráfica y del análisis del apartado anterior, que no hay ningún cero de f a la derecha de $x = 10$. Sin embargo, entre $x = -1$ y $x = 20$ la situación resulta confusa, ya que la escala del eje OY es demasiado pequeña: observa que la parte visible es $[-1, 20] \times [-10000, 2000]$. Vuelve a dibujar la gráfica, pero ahora en el intervalo (por ejemplo) $[0, 10]$.

3. Con esta nueva gráfica parece claro que hay un cero en el intervalo $[0, 2]$ y otro en el intervalo $[6, 8]$. Tratamos de aproximarlos con `fzero`.

```
x1 = fzero(fun,0);          %  x1 = 0.7815
x2 = fzero(fun,6);          %  x2 = 7.1686
```

Ejercicio 6.11 Las frecuencias naturales de la vibración de una viga homogénea sujeta por un extremo son las soluciones de :

$$f(x) = \cos(x)\,\cosh(x) + 1 = 0$$

Se desea saber qué raíces tiene f en el intervalo $[0, 15]$. Calcular dichas raíces utilizando la función MATLAB `fzero`.

Realizar las gráficas necesarias para «localizar» los ceros de la función y luego calcularlos, uno a uno, partiendo de un punto suficientemente próximo.

1. Define una función anónima para f y dibuja su gráfica en el intervalo $[0, 15]$:

```
fun = @(x)   cos(x).*cosh(x)+1;
x = linspace(1,15);
plot(x,fun(x))
grid on
```

   Comprobarás que, salvo que hay una raíz cerca de $x = 15$, la gráfica no clarifica si hay o no más raíces ni dónde están.

   Tendrás que hacer «acercamientos» sucesivos en intervalos más pequeños para localizarlas.

2. Las soluciones son:

$$x1 = 1.8751, \quad x2 = 4.6941, \quad x3 = 7.8548, \quad x4 = 10.9955, \quad x5 = 14.1372$$

Ejercicio 6.12 Se desea saber si un tanque con desperdicios tóxicos que se deja caer en el océano alcanza una determinada velocidad crítica al tocar fondo, ya que, de ser así, podría romperse, liberando su carga y contaminando el océano.

Aplicando las leyes de Newton, se obtiene la siguiente ecuación para la velocidad de caída, v:

$$\frac{gd}{p} = -\frac{v}{c} - \frac{p-b}{c^2}\ln\left(\frac{p-b-cv}{p-b}\right)$$

donde
- g: es la constante de aceleración de la gravedad
- d: es la distancia (altura) que recorre el tanque
- p: es el peso del tanque
- b: es la fuerza hidrotástica
- c: es el coeficiente de proporcionalidad de la fuerza hidrodinámica

Escribir una M-función

```
function [v] = caida(p,d)
```

que, a partir de los valores de `p` (peso) y `d` (altura), calcule la velocidad `v` con la que el tanque llega al fondo, considerando los siguientes valores del resto de los parámetros:

$g = 32.174$ ft/s^2, $\quad b = 470.327$ lb, $\quad c = 0.08$.

Determinar si tiene o no peligro de rotura un tanque de 527.436 lb de peso que se deja caer en un punto con 300 ft de profundidad, teniendo en cuenta que la velocidad crítica es $v_c = 40$ ft/s.

Vamos a comenzar por re-escribir la ecuación de una forma más conveniente. Multiplicando toda la ecuación por $c^2\, p$ y re-ordenando los términos se tiene:

$$f(v) = p(p-b)\, \ln\left(1 - \frac{cv}{p-b}\right) + pcv + gdc^2 = 0$$

Vamos a dibujar la función $f(v)$ para v en el intervalo $[0, 50]$ ya que, por un lado, debe ser $v > 0$ y por otro queremos averiguar si la solución de la ecuación alcanza el valor de la velocidad crítica $v_c = 40$. Este intervalo parece, pues, razonable.

El uso de la M-función con `p=527.436` y `d=300`,

```
v = caida(527.436, 300)
```

dará como resultado v = 44.7481, indicando que, efectivamente, el tanque se rompería al tocar fondo.

```
function [v] = caida(p,d)
%-----------------------------------------------------------------
% v = caida(p,d) es la velocidad con la que llega al fondo
%          del océano un tanque de peso p en un punto de
%          profundidad d
%-----------------------------------------------------------------
% v  es la solución de la ecuacion
%  f(v) =  p(p-b) log( 1-(cv)/(p-b) ) + pcv + gdc^2 = 0
% donde
% b   es la fuerza hidrostatica
% c   es el coef. proporcionalidad de fuerza hidrodinamica
% g   es la aceleración de la gravedad
%-----------------------------------------------------------------
%

%   parametros del problema
g = 32.174;
c = 0.08;
b = 470.327;

%   coeficientes de la función
gdc  = g*d*c*c;
pmb  = p-b;
cpb  = c/pmb;
pc   = p*c;
ppmb = p*pmb;

%   funcion anonima f(v)
fun = @(v) ppmb*log(1-cpb * v) + pc*v + gdc;
```

```
%   representacion grafica para v en [0,50]
v  = linspace(0,50);
fv = fun(v);
plot(v,fv)
grid on
shg

%   busqueda del cero a partir de v=40
v = fzero(fun,40);
```

6.5 Ceros de funciones definidas por un conjunto discreto de valores

En ocasiones, como ya se ha visto antes, es preciso trabajar con funciones de las que sólo se conocen sus valores en un conjunto finito de puntos.

Para calcular (de forma aproximada) en qué punto(s) se anula una tal función se pueden interpolar sus valores por alguno de los procedimientos estudiados en el Tema anterior, definir una función anónima con la interpolante y calcular con `fzero` el cero de esta función.

Ejercicio 6.13 El fichero `Datos.dat` contiene, en dos columnas, las abscisas y las ordenadas de un conjunto de puntos correspondientes a los valores de una función.

Se desea interpolar dichos valores mediante una interpolante lineal a trozos y calcular el valor para el que dicha interpolante toma el valor 0.56.

1. Comienza por leer los datos del fichero:

   ```
   mat = load('Datos.dat');
   xs = mat(:,1);
   ys = mat(:,2);
   ```

2. Si denotamos por $g(x)$ a la interpolante, lo que tenemos que hacer es resolver la ecuación

   $$g(x) = 0.56 \iff g(x) - 0.56 = 0$$

   Necesitamos definir una función anónima que evalúe el primer miembro de la anterior ecuación

   ```
   fun = @(x)  interp1(xs,ys,x)-0.56;
   ```

3. Ahora utilizamos `fzero` para calcular un cero de esta función partiendo, por ejemplo, del punto medio del intervalo ocupado por los `xs` (se podría elegir cualquier otro).

   ```
   fzero(fun,mean(x))
   ```

   Obtendrás el valor $x = 0.6042$.

Ejercicio 6.14 Repite la práctica anterior, pero interpolando mediante un spline cúbico.

1. Igual que antes, lee los datos del fichero:

```
mat = load('Datos.dat');
xs = mat(:,1);
ys = mat(:,2);
```

2. La función anónima ahora se podría escribir

```
fun = @(x)  spline(xs,ys,x)-0.56;
```

   Sin embargo, sería mas adecuado lo siguiente:

```
coef = spline(xs,ys);           % calcula coeficientes del spline
fun = @(x) ppval(coef,x)-0.56;  % evalúa el spline en x
```

   Esta segunda opción es mejor porque con ella los coeficientes se calculan una sola vez.

3. Ahora utiliza `fzero` igual que antes

```
fzero(fun,mean(x))
```

   Obtendrás el valor $x = 0.6044$.

7 Integración numérica

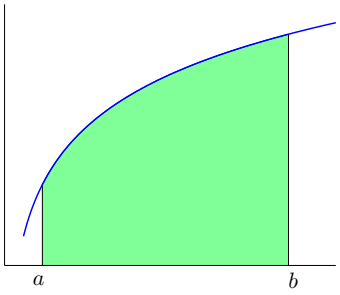
7.1 Introducción

La integral definida de una función continua

$$f : [a, b] \subset \mathbb{R} \mapsto \mathbb{R},$$

$$I(f) = \int_a^b f(x)\, dx \tag{7.1}$$

es, si $f(x) > 0$ en $[a, b]$, el área de la región del plano delimitada por la gráfica de la función, el eje de abscisas y las rectas verticales $x = a$ y $x = b$ como se muestra en la Figura.



Si se conoce una primitiva, F, de la función f, es bien sabido que el valor de la integral definida se puede calcular mediante la *Regla de Barrow*:

$$\int_a^b f(x)\, dx = F(b) - F(a).$$

En la mayoría de los casos, sin embargo, no se puede utilizar esta fórmula, ya que no se conoce dicha primitiva. Es posible, por ejemplo, que no se conozca la expresión matemática de la función f, sino sólo sus valores en determinados puntos. Pero también hay funciones (de apariencia sencilla) para las que se puede demostrar que no tienen ninguna primitiva que pueda escribirse en términos de funciones elementales, como por ejemplo $f(x) = e^{-x^2}$.

La **integración numérica** es una herramienta de las matemáticas que proporciona **fórmulas** y **técnicas** para calcular aproximaciones de integrales definidas. Gracias a ella se pueden calcular, aunque sea de forma aproximada, valores de integrales definidas que no pueden calcularse analíticamente y, sobre todo, se puede realizar ese cálculo en un ordenador.

7.2 La función integral (función quad en versiones anteriores)

MATLAB dispone de la función `integral` para calcular integrales definidas[1]

```
integral(fun,a,b);
```

`a, b` son los límites de integración

`fun` es la función a integrar y puede ser especificada de dos formas:

`integral(fun,a,b)` mediante una función anónima

`integral(@fun,a,b)` mediante una M-función

Ejercicio 7.1 Calcula la integral definida

$$\int_{0.2}^{3} x\ \mathrm{sen}(4\ln(x))\,dx$$

1. Comienza por definir la función a integrar como una función anónima:

   ```
   f = @(x) x.*sin(4*log(x));
   ```

   Observa que la expresión de la función debe escribirse en forma vectorizada (de forma que si el argumento es un vector, devuelva un vector).

2. Calcula la integral usando `integral`

   ```
   q = integral(f, 0.2, 3)
   ```

   Debes obtener el valor `-0.2837`.

3. También se podría escribir, directamente,

   ```
   q = integral(@(x) x.*sin(4*log(x)), 0.2, 3)
   ```

Ejercicio 7.2 Calcula la integral definida

$$\int_{0}^{8} (x\,e^{-x^{0.8}} + 0.2)\,dx$$

[1] **Observación:** en antiguas versiones de MATLAB, la función básica para cálculo numérico de integrales definidas era `quad`, que ha sido sustituida por `integral`, que, aunque tiene más funcionalidades, funciona igual que `quad` en los casos en que esta última funcionaba. MATLAb avisa de que la función `quad` desaparecerá en futuras versiones.

Para definir la función a integrar usa una M-función de nombre `mifun.m`. Usando la función `area`, representa gráficamente el área por debajo de la curva de la función a integrar, las rectas $x = 0$ y $x = 8$ y el eje OX. Consulta para ello el help de MATLAB.

1. Escribe, en un fichero de nombre `mifun.m`, una M-función que evalúe la función a integrar:

```
function [y] = mifun(x)
y = x.*exp(-x.^0.8)+0.2;
```

   Observa nuevamente que la expresión de la función se escribe utilizando operaciones elemento a elemento, como si el argumento `x` fuera un vector.

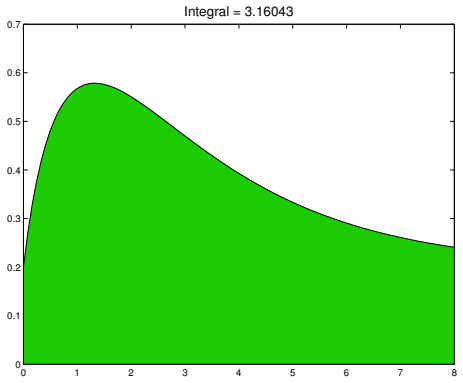
2. Crea un fichero de nombre `practica2.m` que va a contener el cálculo de la integral definida y escribe en él las órdenes.

```
q = integral(@mifun, 0, 8)
```

   Debes obtener el valor `3.1604`

3. La gráfica pedida se puede hacer con las órdenes:

```
x = linspace(0,8);
y = mifun(x);
area(x,y, 'FaceColor', [0.1,0.8,0])
title(['Integral=',num2str(q,'%12.5f')], 'FontSize',14);
```



Ejercicio 7.3 Calcular la integral definida de la Práctica 7.1, usando una M-función

$$\int_{0.2}^{3} x \,\mathrm{sen}(4\ln(x))\,dx$$

1. Crea una M-función de nombre `mifun3.m` para evaluar la función a integrar:

```
function [y] = mifun3(x)
y = x.*sin(4*log(x));
```

   Recuerda vectorizar las operaciones.

2. Calcula la integral:

```
q = integral(@mifun3, 0.2, 3)
```

Ejercicio 7.4 Calcula la integral definida

$$\int_{0.5}^{7} x \,\mathrm{sen}(4\ln(kx))\,dx$$

para $k = 3.33$. Para definir la función a integrar usa una función anónima. Cambia el valor del parámetro k, tomando otro valor y observa cómo cambia el valor de la integral.

1. Crea un fichero de nombre `practica4.m` que contendrá el código para realizar este ejercicio. Define una función anónima que depende del parámetro k:

```
k = 3.33;
f = @(x) x.*sin(4*log(k*x));
```

2. Calcula la integral definida:

```
q = integral(f, 0.5, 7);
```

   Comprueba que has obtenido el valor `-9.6979`.

3. Cambia el valor de k, tomando por ejemplo $k = 10$ y vuelve a ejecutar el programa. Debes obtener el valor de la integral igual a `-1.9038`.

4. Otra posibilidad sería la siguiente:

```
f = @(x,k) x.*sin(4*log(k*x));
integral( @(x) f(x,3.33), 0.5, 7);
integral( @(x) f(x,10), 0.5, 7);
```

7.3 La función trapz

La función MATLAB `trapz` permite calcular la integral definida de una función definida por un conjunto discreto de datos, es decir, una función de la que sólo se conoce un número finito de puntos de su gráfica. Lo que se hace es calcular la integral definida del interpolante lineal a trozos correspondiente.

```
q = trapz(x,y);
```

`x,y` son dos vectores de la misma dimensión y representan las coordenadas de los puntos.

`q` es el valor de la integral

Ejercicio 7.5 Calcula la integral definida de una función que viene dada a través del siguiente conjunto de puntos:

$$x = (0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15])$$
$$y = (10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85)$$

1. Define dos vectores que contienen las abscisas y las ordenadas de los puntos:

```
x = [ 0,  2,  3,   5,  6,  8,  9,   11, 12, 14, 15];
y = [10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Calcula la integral usando `trapz`

```
q = trapz(x,y)
```

   Debes obtener el valor `345.7500`.

7.4 Cálculo de áreas

Como es bien sabido, si $f : [a, b] \mapsto \mathbb{R}$, es continua y $f > 0$ en $[a, b]$, entonces el área A de la región delimitada por la curva de ecuación $y = f(x)$, el eje OX y las rectas verticales $x = a$ y $x = b$ viene dada por

$$A = \int_a^b f(x)\, dx$$

Ejercicio 7.6 Calcular el área de la región plana delimitada por la curva de ecuación $y = \text{sen}(4\ln(x))$, el eje OX y las rectas verticales $x = 1$ y $x = 2$.

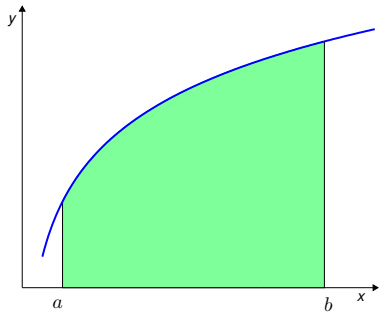
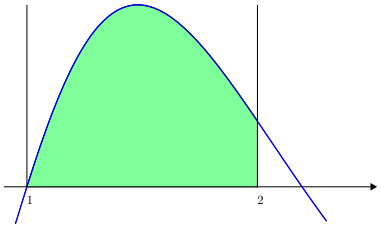


Figura 7.1: Región delimitada por la gráfica de la función $y = f(x)$, el eje de abscisas y las rectas $x = a$ y $x = b$.

1. Comenzamos analizando y representando gráficamente la función $y = \text{sen}(4\ln(x))$ para asegurarnos de que es positiva en el intervalo $[1, 2]$:

```
f = @(x) sin(4*log(x));
x1 = linspace(0.95,2.3);
y1 = f(x1);
plot(x1,y1)
grid on
```



2. Puesto que f es positiva en $[1, 2]$, el área de la región mencionada es:

$$A = \int_1^2 f(x)\,dx$$

y se calcula con la orden

```
A = integral(f, 1, 2)
```

Obtendras el valor: `A = 0.7166`.

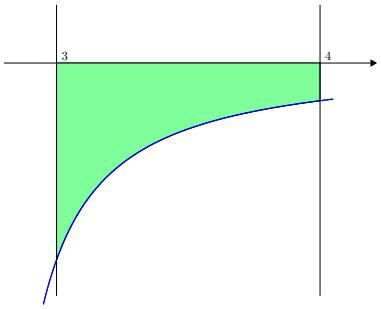
Si la función f es negativa en $[a, b]$, entonces el área de la región delimitada por la curva $y = f(x)$, el eje OX y las rectas verticales $x = 1$ y $x = b$ es

$$A = -\int_a^b f(x)\,dx$$

Ejercicio 7.7 Calcular el área de la región plana delimitada por la curva de ecuación $y = \dfrac{1}{x(1-\ln(x))}$, el eje OX y las rectas verticales $x = 3$ y $x = 4$.

1. De nuevo comenzamos representando gráficamente la función para analizar su signo:

```
f = @(x)  1./(x.*(1-log(x)));
x1 = linspace(3,4);
y1 = f(x1);
plot(x1,y1)
grid on
```



2. Puesto que f es negativa en $[3, 4]$, el área de la región mencionada es:

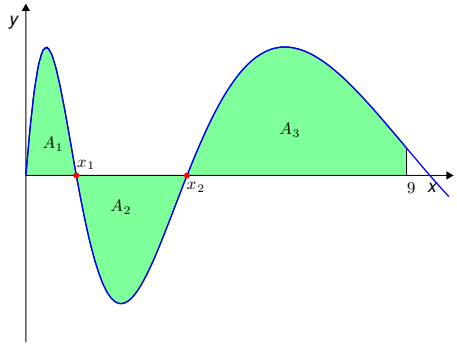
$$A = -\int_3^4 f(x)\,dx$$

y se calcula con la orden

```
A = - integral(f,3,4)
```

Obtendras el valor: `area = 1.3654`.

Ejercicio 7.8 Calcular el área total de la región delimitada por la curva de ecuación $y = \operatorname{sen}(4\ln(x+1))$ y el eje OX, entre los puntos de abscisas $x = 0$ y $x = 9$.



1. Comenzamos por representar gráficamente la función para analizar su signo:

```
f = @(x) sin(4*log(x+1));
x1 = linspace(0,10);
y1 = f(x1);
plot(x1,y1)
grid on
```

2. La observación de la gráfica nos muestra que la función f cambia de signo en dos puntos del intervalo $[0, 9]$: x_1 y x_2 y que es positiva en $[0, x_1]$, negativa en $[x_1, x_2]$ y de nuevo positiva en $[x_2, 9]$.

   Tenemos que calcular las tres áreas por separado y para ello lo primero es calcular los valores x_1 y x_2:

```
xcero1 = fzero(f, 1);                % xcero1 = 1.1933
xcero2 = fzero(f, 4);                % xcero2 = 3.8105
```

3. Calculamos ahora las tres áreas:

$$A_1 = \int_0^{x_1} f(x)\,dx \qquad A_2 = -\int_{x_1}^{x_2} f(x)\,dx \qquad A_3 = \int_{x_2}^{9} f(x)\,dx$$

```
A1 =   integral(f,0,xcero1);          % A1 = 0.7514
A2 = - integral(f,xcero1,xcero2);     % A2 = 1.6479
A3 =   integral(f,xcero2,9);          % A3 = 3.5561
```

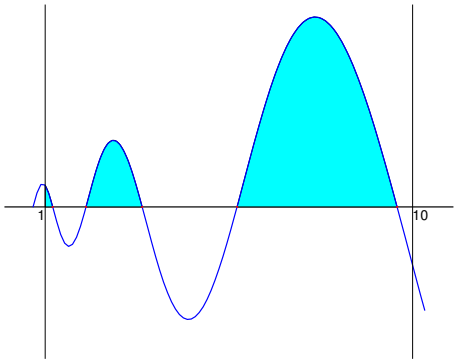
4. El área total buscada es finalmente:

```
A = A1 + A2 + A3;                     % A  = 5.9554
```

Ejercicio 7.9 Calcular el área de la región del primer cuadrante delimitada por la curva $y = x \operatorname{sen}\left(6 \log\left(\frac{x}{2}\right)\right)$, el eje OX y las rectas verticales $x = 1$ y $x = 10$.

1. Comenzamos por representar gráficamente la función para analizar su signo:

```
f = @(x) x.*sin(6*log(x/2));
x = linspace(0.7,10.3);
y = f(x);
plot(x,y)
grid on
```



2. La observación de la gráfica nos muestra que la función f cambia de signo en cinco puntos del intervalo $[1, 10]$: x_1, x_2, x_3, x_4 y x_5 y delimita tres regiones en el primer cuadrante. Comenzamos por calcular los puntos de cambio de signo:

```
xcero1 = fzero(f, 1);
xcero2 = fzero(f, 2);
xcero3 = fzero(f, 3);
xcero4 = fzero(f, 6);
xcero5 = fzero(f, 10);
```

3. Calculamos ahora las tres áreas:

$$A_1 = \int_1^{x_1} f(x)\,dx \qquad A_2 = \int_{x_2}^{x_3} f(x)\,dx \qquad A_3 = \int_{x_4}^{x_5} f(x)\,dx$$

```
A1 = integral(f,1,xcero1);              %  A1 =  0.0892
A2 = integral(f,xcero2,xcero3);         %  A2 =  2.3098
A3 = integral(f,xcero4,xcero5);         %  A3 = 18.7567
```

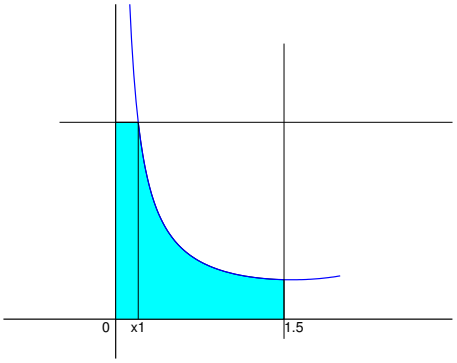
4. El área total buscada es finalmente:

```
A = A1 + A2 + A3;                       %  A  = 21.1557
```

Ejercicio 7.10 (Propuesta) Calcular el área de la región delimitada por la curva $y = \dfrac{1}{\operatorname{sen} x}$, el eje OX, el eje OY, la recta horizontal $y = 5$ y la recta vertical $x = 1.5$.

1. Comenzamos por dibujar las curvas que delimitan la región:

```
f = @(x)  1./sin(x);
x = linspace(0,1.5);
y = f(x);
plot(x,y)
grid on
axis([0,1.5,0,5])
```



2. La observación de la gráfica nos muestra que la región cuya área hay que calcular es la unión de dos regiones: un rectángulo de base x_1 y altura 5 y la zona bajo la curva $y = 1/\operatorname{sen}(x)$ entre x_1 y 1.5.

   x_1 es la abscisa del punto en que la curva $y = 1/\operatorname{sen}(x)$ corta a la recta horizontal $y = 5$, es decir, $x_1 = \operatorname{arc\,sen}(1/5)$.

3. Calculamos ahora las áreas:

$$A_1 = x_1 \times 5, \qquad A_2 = \int_{x_1}^{1.5} f(x)\,dx$$

```
x1 = asin(1/5);
A1 = 5*x1;                      %  A1 = 1.0068
A2 = integral(f, x1, 1.5);      %  A2 = 2.2216
A  = A1 + A2;                   %  3.2284
```

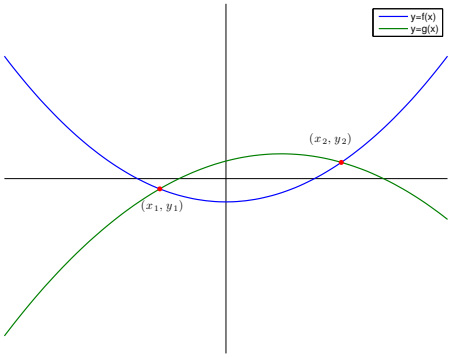
Ejercicio 7.11 Calcular los puntos de corte de las curvas siguientes, así como el área de la región plana encerrada entre ellas

$$y = x^2 - 4 = f(x) \qquad \text{y} \qquad y = 2x - 0.8\,x^2 + 3 = g(x)$$

1. Tenemos que comenzar por identificar la zona en cuestión y hacernos una idea de la localización de los puntos de corte. Para ello vamos a dibujar ambas curvas, por ejemplo, entre $x = -5$ y $x = 5$ (podrían ser otros valores):

```
f  = @(x) x.^2-4;
g  = @(x) 2*x - 0.8*x.^2 +3;
```

```
x  = linspace(-5,5);
yf = f(x);
yg = g(x);
plot(x,yf,x,yg)
grid on
```



2. La observación de la gráfica nos muestra que ambas curvas se cortan en dos puntos, (x_1, y_1) y (x_2, y_2).
   Para calcularlos comenzamos por calcular sus abscisas x_1 y x_2, que son las soluciones de la ecuación

$$f(x) = g(x) \quad \text{equivalente a} \quad f(x) - g(x) = 0$$

```
fg = @(x) f(x) - g(x);
x1 = fzero(fg, -1);                    %  x1 = -1.4932
x2 = fzero(fg,  3);                    %  x2 =  2.6043
```

   Las ordenadas de los puntos de corte son los valores en estos puntos de la función f (o g, ya que toman el mismo valor en ellos):

```
y1 = f(x1);                 % o bien y1 = g(x1);    y1 = -1.7703
y2 = f(x2);                 % o bien y2 = g(x2);    y2 =  2.7826
```

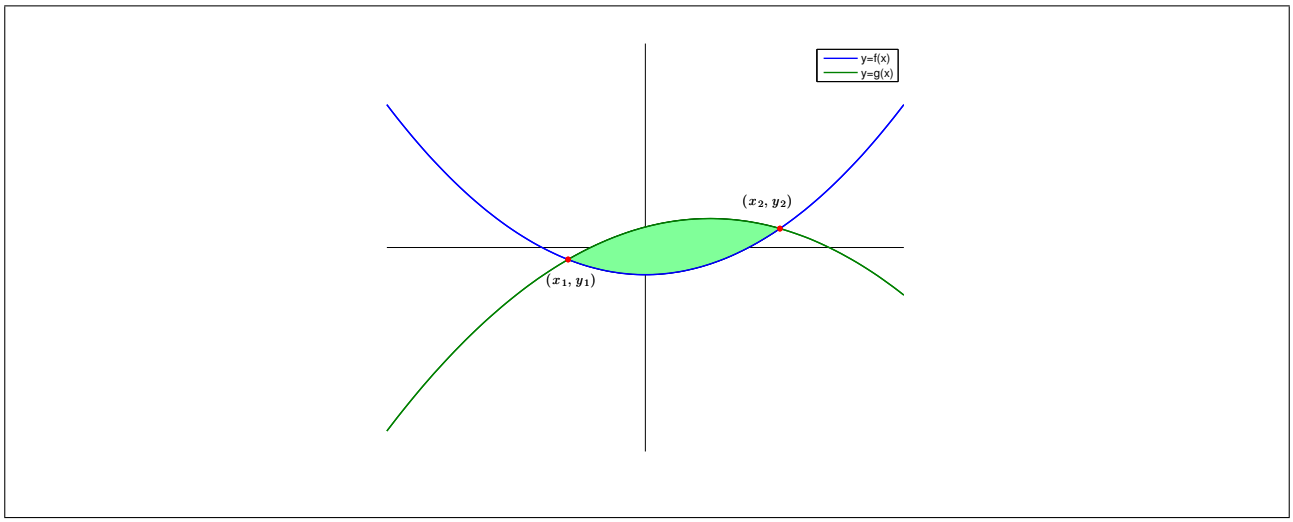
   Así pues, las dos curvas se cortan en los puntos $(-1.4932, -1.7703)$ y $(2.6043, 2.7826)$.

3. El área de la región que queda encerrada entre las dos curvas es, ya que en $[x_1, x_2]$ se tiene $g(x) \geq f(x)$:

$$A = \int_{x_1}^{x_2} (g(x) - f(x))\, dx = -\int_{x_1}^{x_2} (f(x) - g(x))\, dx$$

   y se puede calcular con la orden:

```
A = - integral(fg,x1,x2);                %  A = 20.6396
```


y=f(x)
y=g(x)
(x_2, y_2)
(x_1, y_1)

7.5 Cálculo de integrales de funciones definidas por un conjunto discreto de valores

Como hemos mencionado antes, muchas veces es preciso trabajar con funciones de las que sólo se conocen sus valores en un conjunto de puntos. Para calcular (de forma aproximada) la integral definida de una tal función se pueden interpolar sus valores por alguno de los procedimientos vistos anteriormente, definir una función anónima con la interpolante y calcular con `quad` la integral definida.

Ejercicio 7.12 Se considera el siguiente conjunto de puntos correspondientes a los valores de una función:

```
x=[0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y=[0, 20, 30, -10, 10, 10, 10.5, 15, 50 , 60 , 85];
```

Se desea interpolar dichos valores mediante un *spline* cúbico $s = s(x)$, representarlo gráficamente y calcular la integral definida:

$$\int_{10}^{14} s(x)\,dx$$

1. Crea dos variables `x` e `y` con las abscisas y las ordenadas que vas a interpolar:

   ```
   x=[0,  2,  3,   5,  6,  8,  9,   11, 12, 14, 15];
   y=[0, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
   ```

2. Calcula con la orden `spline` los coeficientes del *spline* cúbico que pasa por estos puntos:

   ```
   c = spline(x,y);
   ```

3. Define una función anónima $s = s(x)$ que evalúe el *spline* en los puntos x:

   ```
   s = @(x) ppval(c, x);
   ```

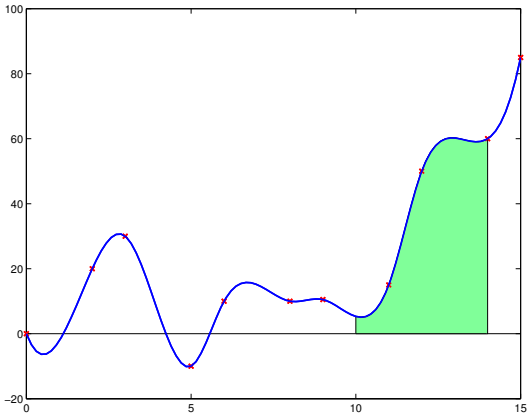
4. Calcula (usando la función `integral`) la integral definida requerida

   ```
   a = quad(s, 10, 14)
   ```

   Obtendrás el valor $a = 157.0806$.

5. Dibuja los puntos y el *spline*:

   ```
   plot(x,y,'rx', 'LineWidth',1.5)
   hold on
   xs = linspace(min(x),max(x));
   ys = s(xs);
   plot(xs,ys, 'LineWidth',1.5)
   hold off
   ```



Ejercicio 7.13 Se considera el siguiente conjunto de datos:

```
x = [0, 1,     2,     3,    4,    5,    6,     7,     8,    9,   10];
y = [0, 0.28, -1.67, -2.27, 1.63, 4.95, 0.92, -6.32, -5.33, 4.72, 9.64]
```

Se desea interpolar dichos valores mediante una interpolante lineal a trozos $s(x)$, calcular la integral definida

$$\int_{\pi/2}^{2\pi} s(x)\,dx$$

y realizar una representación gráfica de los datos junto con la interpolante lineal a trozos.

1. Crea dos variables `xs` e `ys` con las abscisas y las ordenadas que vas a interpolar:

   ```
   xs = 0:10;
   ys = [0, 0.28, -1.67, -2.27, 1.63, 4.95, 0.92, -6.32, -5.33, 4.72, 9.64]
   ```

2. Define una función anónima que evalúe $s(x)$:

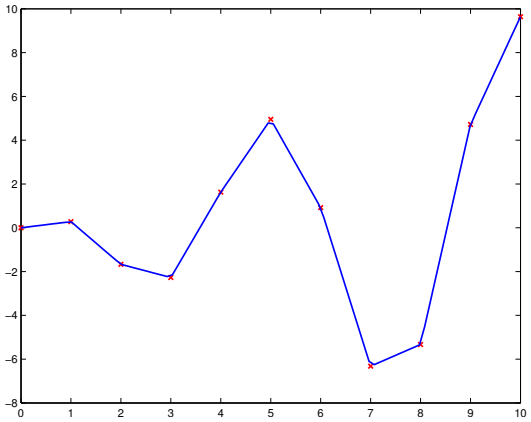
   ```
   s1 = @(x) interp1(xs,ys,x);
   ```

3. Ahora utiliza `integral` para calcular la integral definida:

   ```
   a = integral(s, pi/2, 2*pi)            % a= 3.3681
   ```

4. Representa ahora los puntos y la intepolante lineal a trozos:

   ```
   plot(xs,ys,'rx', 'LineWidth',1.5)
   hold on
   x1 = linspace(min(xs),max(xs));
   y1 = s1(x1);
   plot(x1,y1, 'LineWidth',1.5)
   ```



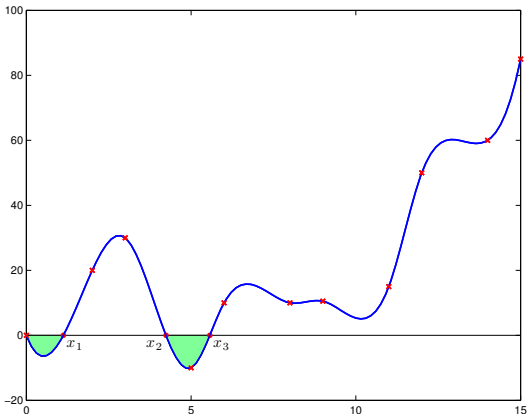
Ejercicio 7.14 Usando los datos del ejercicio 7.12, calcular el area de la región del cuarto cuadrante ($x \geq 0,\ y \leq 0$) delimitada por el *spline* cúbico que interpola estos datos y el eje OX.

1. Crea dos variables `x` e `y` con las abscisas y las ordenadas de los puntos a interpolar:

```
x = [0,  2,  3,   5,  6,  8,  9,   11, 12, 14, 15];
y = [0, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Comienza por representar gráficamente el *spline* para analizar su signo:

```
c = spline(x, y);
s = @(x) ppval(c, x);
xs = linspace(0, 15);
ys = s(xs);
plot(xs,ys)
grid on
```



3. La observación de la gráfica muestra que el *spline* cúbico cambia de signo en tres puntos del intervalo $[0, 15]$: x_1, x_2 y x_3 y que las regiones que nos interesan son las que quedan entre $x = 0$ y $x = x_1$ por un lado, y entre $x = x_2$ y $x = x_3$ por otro. Lo primero es calcular los valores x_1, x_2 y x_3:

```
x1 = fzero(s,1)                    %  x1 = 1.1240
x2 = fzero(s,4)                    %  x2 = 4.2379
x3 = fzero(s,6)                    %  x3 = 5.5685
```

4. Calcula ahora las dos áreas:

$$A_1 = -\int_0^{x_1} s(x)\,dx, \qquad A_2 = -\int_{x_2}^{x_3} s(x)\,dx,$$

```
A1 = -integral(s,0,x1)             %  A1 = 4.7719
A2 = -integral(s,x2,x3);           %  A2 = 8.6921
```

5. El área total buscada es finalmente:

```
A = A1 + A2                        %  A =  13.4640
```

Ejercicio 7.15 (Propuesta) Se considera el siguiente conjunto de puntos correspondientes a los valores de una función:

```
x = [ 12,    13,   14,    15,    16,     17,    18,    19,    20]
y = [-11.43, -7.30, 8.86, 13.82, -1.76, -16.73, -8.06, 13.87, 17.24]
```

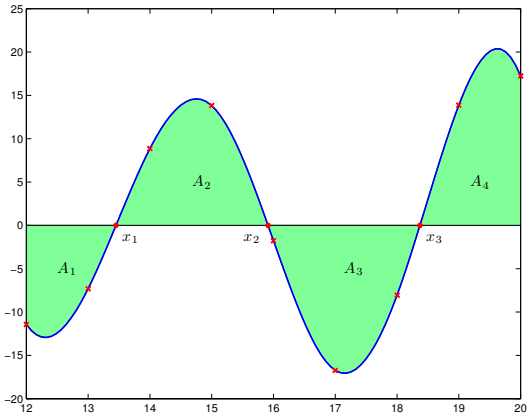
Se pide calcular el area de la región delimitada por el *spline* cúbico que interpola estos datos, el eje OX y las rectas verticales $x = 12$ y $x = 20$.

1. Creamos dos variables `x` e `y` con las abscisas y las ordenadas de los puntos dados:

```
x = 12:20
y = [-11.43, -7.30, 8.86, 13.82, -1.76, -16.73, -8.06, 13.87, 17.24]
```

2. Comenzamos por representar gráficamente el *spline* para analizar su signo:

```
c  = spline(x,y);
s  = @(x) ppval(c,x);
xs = linspace(12,20);
ys = s(xs);
plot(xs,ys)
grid on
```



3. La observación de la gráfica nos muestra que el *spline* cúbico cambia de signo en tres puntos del intervalo $[0, 15]$: x_1, x_2 y x_3 y que es negativo en $[12, x_1]$, positivo en $[x_1, x_2]$, es negativo en $[x_2, x_3]$ y de nuevo positivo en $[x_3, 20]$.

   Tenemos que calcular las cuatro áreas por separado y para ello lo primero es calcular los valores x_1, x_2 y x_3:

```
x1 = fzero(s,13)                    %  x1 = 13.4514
x2 = fzero(s,16)                    %  x2 = 15.9126
x3 = fzero(s,19)                    %  x3 = 18.3707
```

4. Calculamos ahora las cuatro áreas:

$$A_1 = -\int_{12}^{x_1} s(x)\,dx, \quad A_2 = \int_{x_1}^{x_2} s(x)\,dx, \quad A_3 = -\int_{x_2}^{x_3} s(x)\,dx, \quad A_4 = \int_{x_3}^{20} s(x)\,dx$$

```
A1 = -integral(s, 12, x1)           %  A1 = 13.1036
A2 =  integral(s, x1, x2)           %  A2 = 22.7972
A3 = -integral(s, x2, x3)           %  A3 = 26.5567
A4 =  integral(s, x3, 20)           %  A4 = 23.0904
```

5. El área total buscada es finalmente:

```
A = A1 + A2 + A3 + A4               %  A =   85.5480
```

Ejercicio 7.16 El fichero `datos21.dat` contiene una matriz con dos columnas, que corresponden a las abscisas y las ordenadas de una serie de datos.

Se pide leer los datos del fichero, y calcular y dibujar el *spline* cúbico que interpola dichos valores, en un intervalo que contenga todos los puntos del soporte. También se desea calcular el área de la región del primer cuadrante delimitada por el *spline* cúbico.

1. Puesto que el fichero `datos21.dat` es de texto, se puede editar. Ábrelo para ver su estructura y luego ciérralo.

2. Dado que todas las líneas del fichero tienen el mismo número de datos, se puede leer con la orden `load`:

   ```
   datos = load('datos21.dat');
   ```

   que guarda en la variable `datos` una matriz 50×2 cuya primera columna contiene los valores de las abscisas y la segunda los de las ordenadas. Compruébalo.

3. Extrae la primera columna de la matriz `datos` a un vector `x` y la segunda columna a un vector `y`:

   ```
   x = datos(:,1);
   y = datos(:,2);
   ```

4. Calcula los coeficientes del *spline* y guárdalos en una variable `c`

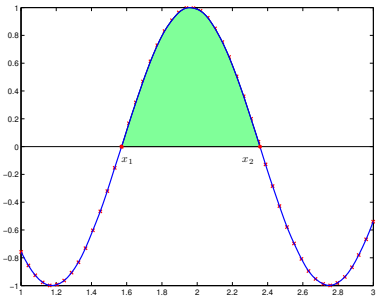
   ```
   c = spline(x,y);
   ```

5. Define una función anónima que evalúe el *spline* en x:

   ```
   f = @(x) ppval(c,x);
   ```

6. Dibuja el *spline* en un intervalo que contenga todos los nodos:

   ```
   xs = linspace(min(x),max(x));
   ys = f(xs);
   plot(xs,ys)
   grid on
   ```

7. En la gráfica se observa que el *spline* cúbico cambia de signo en tres puntos del intervalo $[1, 3]$: x_1 y x_2. La región del primer cuadrante delimitada por el *spline* es la que queda entre $x = x_1$ y $x = x_2$.

   Comienza por calcular los valores de x_1 y x_2:

   ```
   x1 = fzero(f,1.5)                % x1 = 1.5713
   x2 = fzero(f,2.5)                % x2 = 2.3566
   ```

8. Calcula ahora el área:

$$A_1 = \int_{x1}^{x_2} f(x)\,dx$$

```
A = integral(f, x1, x2)                    %  A =  0.5002
```

Ejercicio 7.17 El fichero `datos22.dat` contiene una matriz con dos columnas, que corresponden a las abscisas y las ordenadas de una serie de datos.

Se pide leer los datos del fichero, y calcular la función logarítmica $f(x) = m\ln(x) + b$ que mejor se ajusta a estos valores, y dibujarla en un intervalo que contenga todos los puntos del soporte.

También se pide calcular el área de la región del plano delimitada por la gráfica de la función $f(x)$, el eje OX y las rectas verticales $x = 1.5$ y $x = 3.5$.

1. Puesto que el fichero `datos22.dat` es de texto, se puede editar. Ábrelo para ver su estructura. Cierra el fichero.

2. Dado que todas las líneas del fichero tienen el mismo número de datos, se puede leer con la orden `load`:

   ```
   datos = load('datos22.dat');
   ```

   que guarda en la variable `datos` una matriz 50×2 cuya primera columna contiene los valores de las abscisas y la segunda los de las ordenadas. Compruébalo.

3. Extrae la primera columna de la matriz `datos` a un vector `x` y la segunda columna a un vector `y`:

   ```
   x = datos(:,1);
   y = datos(:,2);
   ```

4. Puesto que hay que usar una función logarítmica de la forma $y = m\ln(x) + b$ para ajustar los datos, está claro que hay que ajustar los datos $\ln(x)$ e y mediante una recta. Calcula los coeficientes de la recta de regresión:

   ```
   c = polyfit(log(x),y,1);
   ```

   Obtendrás `c = [0.9145, 0.0646]` lo que significa, por lo dicho antes, que la función buscada es

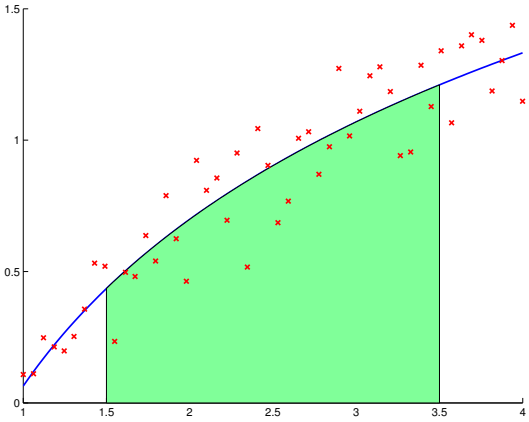
   $$y = 0.9145\ln(x) + 0.0646$$

5. Define ahora la función anónima correspondiente que evalúa en x la función logarítmica obtenida:

```
f = @(x) c(1)*log(x)+c(2);
```

6. Dibuja la función de ajuste obtenida en un intervalo que contenga todos los nodos y muestra también los datos originales:

```
xs = linspace(min(x),max(x));
ys = f(xs);
plot(xs,ys, 'LineWidth',1.5)
hold on
plot(x,y,'rx', 'LineWidth',1.5)
```



7. La gráfica muestra que la función de ajuste es positiva en el intervalo $[1.5, 3.5]$. Calcula ahora el área:

$$A = \int_{1.5}^{3.5} f(x)\,dx$$

```
A = integral(f, 1.5, 3.5)                    % A =  1.7538
```

7.6 Funciones definidas a trozos

Los siguientes ejemplos muestran cómo vectorizar el cálculo de una función que no viene definida mediante una fórmula; por ejemplo, que está definida a trozos.

Ejercicio 7.18 Escribir una M-función para calcular la siguiente función definida a trozos:

$$f(x) = \begin{cases} 0.5x^2 & \text{si } x \geq 0 \\ x^3 & \text{si no} \end{cases}$$

y calcular el área de la región delimitada por la curva $y = f(x)$, el eje OX y las rectas verticales $x = -2$ y $x = 2.5$.

1. En una primera aproximación, escribiríamos probablemente algo como lo que sigue para calcular la función f:

```
function [y] = mifun(x)
if (x>=0)
   y = 0.5*x^2;
else
   y = x^3;
end
```

   en un fichero de nombre `mifun.m`.

   Esto sería correcto si sólo fuéramos a utilizarla siendo el argumento x un escalar. Pero no funcionaría si x fuera un vector, ya que el `if` **no se lleva a cabo componente a componente**.

   Intenta comprender, mediante algún ejemplo y leyendo el help, cuál sería el funcionamiento del `if` anterior si `x` fuese un vector.

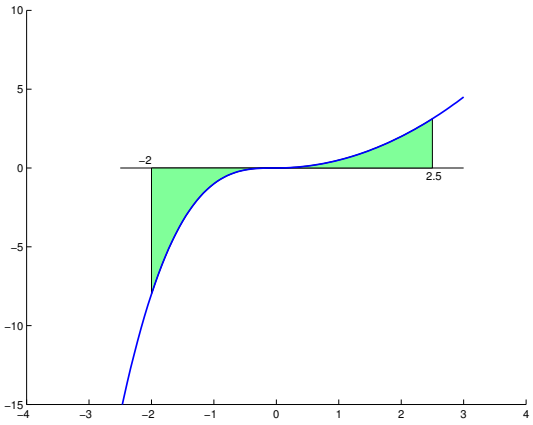
2. Para que una M-función como la anterior sea «vectorial» hay que calcular los valores de `y` componente a componente, como se muestra a continuación:

```
function [y] = mifun(x)
y = zeros(size(x));
for k = 1:length(x)
    if (x(k)>=0)
       y(k) = 0.5*x(k)^2;
    else
       y(k) = x(k)^3;
    end
end
```

   Escribe el código anterior en un fichero y sálvalo con el nombre `mifun.m`. Comprueba, desde la ventana de comandos, que el programa funciona correctamente, dando distintos valores a la variable de entrada.

3. Dibuja la función en el intervalo $[-3, 3]$ (por ejemplo):

```
x = linspace(-3,3);
y = mifun(x);                    % calculo vectorial
plot(x,y, 'LineWidth',1.5)
```



4. Calcula ahora las dos áreas:

$$A_1 = -\int_{-2}^{0} f(x)\,dx, \qquad A_2 = \int_{0}^{2.5} f(x)\,dx,$$

```
A1 = -integral(@mifun, -2, 0)                    %  A1 =  4
A2 =  integral(@mifun, 0, 2.5)                   %  A2 =  2.6042
```

5. Luego el área buscada es:

```
A = A1+A2                                        %  A  =  6.6042
```

Observación

Las órdenes para realizar este ejercicio se pueden escribir todas en un solo fichero de nombre, por ejemplo, `practica23.m`, de la siguiente manera (para ejecutarlo hay que invocar el programa principal, es decir, el que lleva el nombre del fichero):

```
function practica23
%-------------------------------------------------
% Practica numero 23 : funcion "principal"
%
x = linspace(-2.5,3);
y = mifun(x);
plot(x,y, 'LineWidth',1.5)
axis([-2.5,3,-15,10])
grid on
%
A1 = - integral(@mifun,-2,0);
A2 =   integral(@mifun,0,2.5);
A  = A1+A2;
fprintf(' El area calculada es: %12.6f \n',A)
```

```
%-------------------------------------------------
%  mifun : funcion auxiliar
%
function [y] = mifun(x)
y = zeros(size(x));
for k =1:length(x)
    if (x(k)>=0)
       y(k) = 0.5*x(k)^2;
    else
       y(k) = x(k)^3;
    end
end
```

Ejercicio 7.19 Escribir una M-función para evaluar:

$$f(x) = \begin{cases} x\cos(x) & \text{si } x \geq 0 \\ \text{sen}(x) & \text{si no} \end{cases}$$

y calcular el área de la región delimitada por la curva $y = f(x)$, el eje OX y las rectas verticales $x = -5$ y $x = 4$.

8 Resolución de ecuaciones y sistemas diferenciales ordinarios

Una **ecuación diferencial** es una ecuación en que la incógnita es una función y que, además, involucra también las derivadas de la función hasta un cierto orden. La incógnita no es el valor de la función en uno o varios puntos, sino la función en sí misma.

Cuando la incógnita es una función de una sola variable se dice que la ecuación es ordinaria, debido a que la o las derivadas que aparecen en ella son derivadas ordinarias (por contraposición a las derivadas parciales de las funciones de varias variables).

Comenzamos con la resolución de ecuaciones diferenciales ordinarias (edo) de primer orden, llamadas así porque en ellas sólo aparecen las derivadas de primer orden.

Más adelante trataremos la resolución de sistemas de ecuaciones diferenciales ordinarias (sdo) de primer orden. Con ello podremos, en particular, abordar la resolución de ecuaciones diferenciales de orden superior a uno, ya que éstas siempre se pueden reducir a la resolución de un sistema de primer orden.

8.1 Ecuaciones diferenciales ordinarias de primer orden

Una ecuación diferencial ordinaria

$$y' = f(t, y), \quad f\colon \Omega \subset \mathbb{R}^2 \to \mathbb{R}$$

admite, en general, infinitas soluciones. Si, por ejemplo, $f \in C^1(\Omega; \mathbb{R})$[1], por cada punto $(t_0, y_0) \in \Omega$ pasa una única solución $\varphi\colon I \to \mathbb{R}$, definida en un cierto intervalo $I \subset \mathbb{R}$, que contiene a t_0.

Se denomina Problema de Cauchy (PC) o Problema de Valor Inicial (PVI)

$$\begin{cases} y' = f(t, y), \\ y(t_0) = y_0 \end{cases} \tag{8.1}$$

al problema de determinar, de entre todas las soluciones de la ecuación diferencial $y' = f(t, y)$, aquélla que pasa por el punto (t_0, y_0), es decir, que verifica $y(t_0) = y_0$.

Sólo para algunos (pocos) tipos muy especiales de ecuaciones diferenciales es posible encontrar la expresión de sus soluciones en términos de funciones elementales. En la inmensa mayoría de los casos prácticos sólo es posible encontrar aproximaciones numéricas de los valores de una solución en algunos puntos.

Así, una **aproximación numérica** de la solución del problema de Cauchy

$$\begin{cases} y' = f(t, y), \quad t \in [t_0, t_f] \\ y(t_0) = y_0 \end{cases} \tag{8.2}$$

[1] En realidad basta con que f sea continua y localmente Lipschitziana con respecto de y en Ω.

consiste en una sucesión de valores de la variable independiente:

$$t_0 < t_1 < \ldots < t_n = t_f$$

y una sucesión de valores $y_0, y_1, \ldots, y_n$ tales que

$$y_k \approx y(t_k), \quad k = 0, 1, \ldots, n,$$

es decir, y_k es una aproximación del valor en t_k de la solución del problema (8.2).

$$y_k \approx y(t_k), \quad k = 0, 1, \ldots$$

8.2 Resolución de problemas de Cauchy para ecuaciones diferenciales ordinarias de primer orden

MATLAB dispone de toda una familia de funciones para resolver (numéricamente) ecuaciones diferenciales:

```
ode45, ode23, ode113
ode15s, ode23s, ode23t, . . .
```

Cada una de ellas implementa un método numérico diferente, siendo adecuado usar unas u otras en función de las dificultades de cada problema en concreto. Para problemas no demasiado «difíciles» será suficiente con la función `ode45` ó bien `ode23`.

Exponemos aquí la utilización de la función `ode45`, aunque la utilización de todas ellas es similar, al menos en lo más básico. Para más detalles, consúltese la documentación.

Para dibujar la gráfica de la solución (numérica) de (8.2) se usará la orden:

```
ode45(odefun, [t0,tf], y0)
```

donde:

`odefun` es un manejador de la función que evalúa el segundo miembro de la ecuación, $f(t, y)$. Puede ser el nombre de una función anónima dependiente de dos variables, siendo t la primera de ellas e y la segunda, o también una M-función, en cuyo caso se escribiría `@odefun`. Ver los ejemplos a continuación.

`[t0,tf]` es el intervalo en el que se quiere resolver la ecuación, i.e. `t0`$= t_0$, `tf`$= t_f$.

`y0` es el valor de la condición inicial, `y0`$= y_0$.

Ejercicio 8.1 Calcular la solución del Problema de Cauchy:

$$\begin{cases} y' = 5y \quad \text{en} \quad [0, 1] \\ y(0) = 1 \end{cases}$$

Comparar (gráficamente) con la solución exacta de este problema, $y = e^{5t}$

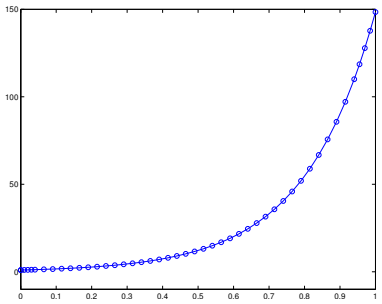
1. Comenzamos por definir una función anónima que represente el segundo miembro de la ecuación $f(t, y)$:

```
f = @(t,y) 5*y;
```

2. Calculamos y dibujamos la solución numérica:

```
ode45(f, [0,1], 1)
```

Obsérvese que no se obtiene ningún resultado numérico en salida, únicamente la gráfica de la solución.



3. Para comparar con la gráfica de la solución exacta, dibujamos en la misma ventana, sin borrar la anterior:

```
hold on
t = linspace(0,1);
plot(t, exp(5*t), 'r')
shg
```

4. Para hacer lo mismo utilizando una M-función en vez de una función anónima, tendríamos que escribir, en un fichero (el nombre `odefun` es sólo un ejemplo):

```
function [dy] = odefun(t,y)
%  Segundo miembro de la ecuacion diferencial
dy = 5*y;
```

y guardar este texto en un fichero de nombre `odefun.m` (el mismo nombre que la función). Este fichero debe estar en la carpeta de trabajo de MATLAB.

Después, para resolver la ecuación, usaríamos `ode45` en la forma:

```
ode45(@odefun, [0,1], 1)
```

Si se desean conseguir los valores t_k e y_k de la aproximación numérica para usos posteriores, se debe usar la función `ode45` en la forma:

```
[t, y] = ode45(odefun, [t0,tf], y0);
```

obteniéndose así los vectores `t` e `y` de la misma longitud que proporcionan la aproximación de la solución

del problema:

$$y(k) \approx y(t(k)), \quad k = 1, 2, \ldots, \texttt{length(y)}.$$

Si se utiliza la función `ode45` en esta forma, **no se obtiene ninguna gráfica**.

Ejercicio 8.2 Calcular el valor en $t = 0.632$ de la solución del problema

$$\begin{cases} y' = t\, e^{t/y} \\ y(0) = 1 \end{cases}$$

Calculamos la solución en el intervalo $[0, 1]$ y luego calculamos el valor en $t = 0.632$ utilizando una interpolación lineal a trozos:

```
f     = @(t,y) t.*exp(t/y);
[t,y] = ode45(f, [0,1], 1);
v     = interp1(t, y, 0.632)
```

Obsérvese que `t` e `y`, las variables de salida de `ode45`, son dos vectores columna de la misma longitud.

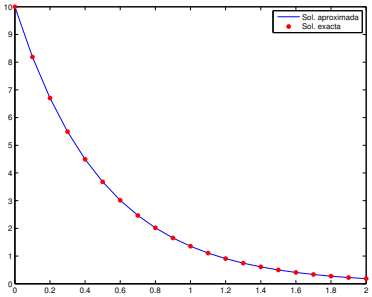
En ocasiones puede ser necesario calcular las aproximaciones y_k en unos puntos t_k pre-determinados. Para ello, en lugar de proporcionar a `ode45` el intervalo `[t0,tf]`, se le puede proporcionar un vector con dichos valores, como en el ejemplo siguiente:

Ejercicio 8.3 Calcular (aproximaciones de) los valores de la solución del problema

$$\begin{cases} y' = -2y \\ y(0) = 10 \end{cases}$$

en los puntos: 0, 0.1, 0.2, ..., 1.9, 2. Comparar (gráficamente) con la solución exacta $y = 10\, e^{-2t}$.

```
f = @(t,y) -2*y;
t = 0:0.1:2;
[t,y] = ode45(f, t, 10);
plot(t, y, 'LineWidth', 1.1)
hold on
plot(t, 10*exp(-2*t), 'r.')
legend('Sol. aproximada', 'Sol. exacta')
shg
```



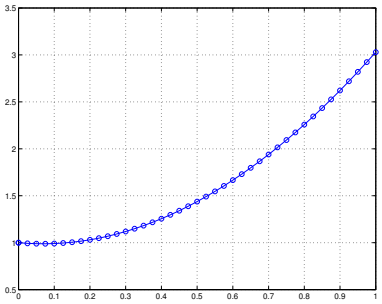
Ejercicio 8.4 Calcular el instante en que la solución del problema

$$\begin{cases} y' = 0.5\big(10t - \ln(y+1)\big) \\ y(0) = 1 \end{cases}$$

alcanza el valor $y = 1.5$.

1. Comenzamos por visualizar la gráfica de la solución, para obtener, por inspección, una primera aproximación:

```
f = @(t,y) 0.5*(10*t-log(y+1));
ode45(f, [0,1], 1)
grid on
shg
```



Vemos así que el valor $y = 1.5$ se produce para un valor de t en el intervalo $[0.5, 0.6]$.

2. Usaremos ahora la función `fzero` para calcular la solución de la ecuación $y(t) = 1.5$ escrita en forma homogénea, es decir, $y(t) - 1.5 = 0$, dando como aproximación inicial (por ejemplo) $t = 0.5$. Para ello, necesitamos construir una función que interpole los valores que nos devuelva `ode45`. Usamos interpolación lineal a trozos:

```
[ts,ys] = ode45(f, [0,1], 1);
fun   = @(t) interp1(ts, ys, t) - 1.5;
fzero(fun, 0.5)
```

Obtendremos el valor $t \approx 0.5292$.

8.3 Resolución de problemas de Cauchy para sistemas diferenciales ordinarios de primer orden

Nos planteamos ahora la resolución de sistemas diferenciales ordinarios:

$$\begin{cases} y_1' = f_1(t, y_1, y_2, \ldots, y_n), \\ y_2' = f_2(t, y_1, y_2, \ldots, y_n), \\ \ldots \\ y_n' = f_n(t, y_1, y_2, \ldots, y_n), \end{cases} \tag{8.3}$$

y, concretamente, de Problemas de Cauchy asociados a ellos, es decir, problemas consistentes en calcular la solución de (8.3) que verifica las condiciones iniciales:

$$\begin{cases} y_1(t_0) = y_1^0, \\ y_2(t_0) = y_2^0, \\ \ldots \\ y_n(t_0) = y_n^0. \end{cases} \tag{8.4}$$

Gracias a las características vectoriales del lenguaje de MATLAB, estos problemas se resuelven con las mismas funciones (`ode45`, `ode23`, etc.) que los problemas escalares. Basta escribir el problema en forma vectorial. Para ello, denotamos:

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{pmatrix}, \quad F(t, Y) = \begin{pmatrix} f_1(t, y_1, y_2, \ldots, y_n) \\ f_2(t, y_1, y_2, \ldots, y_n) \\ \ldots \\ f_n(t, y_1, y_2, \ldots, y_n) \end{pmatrix}, \quad Y_0 = \begin{pmatrix} y_1^0 \\ y_2^0 \\ \ldots \\ y_n^0 \end{pmatrix}, \tag{8.5}$$

Con esta notación el problema planteado se escribe:

$$\begin{cases} Y' = F(t, Y) \\ Y(t_0) = Y_0 \end{cases} \tag{8.6}$$

y se puede resolver (numéricamente) con la función `ode45`, de forma similar al caso escalar, con las adaptaciones oportunas, como en el ejemplo siguiente.

Ejercicio 8.5 Calcular (una aproximación de) la solución del problema

$$\begin{cases} y_1' = y_2\, y_3, \\ y_2' = -0.7\, y_1\, y_3, \qquad t \in [0, 5\pi] \\ y_3' = -0.51\, y_1\, y_2, \\ y_1(0) = 0 \\ y_2(0) = 1 \\ y_3(0) = 1 \end{cases}$$

1. Comenzamos por escribir el sistema en notación vectorial:

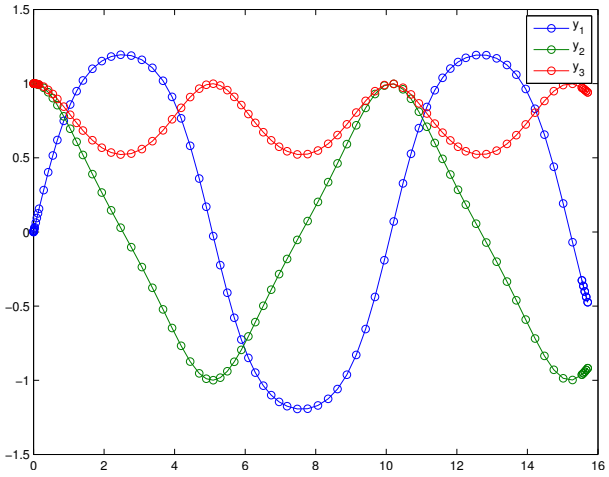
$$Y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \quad F(t, Y) = \begin{pmatrix} y_2\, y_3 \\ -0.7\, y_1\, y_3 \\ -0.51\, y_1\, y_2 \end{pmatrix}, \quad Y_0 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \tag{8.7}$$

2. Definimos una función anónima para $F(t, Y)$ y el dato inicial (ambos son vectores columna con tres componentes):

```
f = @(t,y) [ y(2)*y(3); -0.7*y(1)*y(3); -0.51*y(1)*y(2)];
y0 = [ 0; 1; 1];
```

3. Utilizamos `ode45` para calcular y dibujar la solución. Añadimos una leyenda para identificar correctamente la curva correspondiente a cada componente: $y_1(t)$, $y_2(t)$, $y_3(t)$.

```
ode45(f, [0, 5*pi], y0)
legend('y_1', 'y_2', 'y_3')
```



4. Vamos ahora a utilizar `ode45` recuperando las variables de salida de la aproximación numérica:

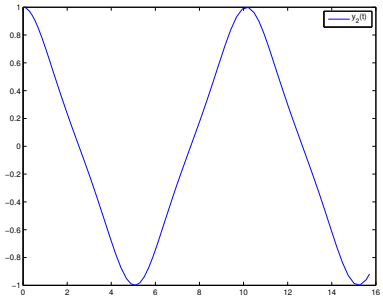
```
[t, y] = ode45(f, [0, 5*pi], y0);
```

Obsérvese que `t` es un vector columna y que `y` es una matriz con tantas filas como `t` y tres columnas: cada columna es una de las componentes de la solución Y. Es decir:

$$\texttt{y(k,i)} \approx y_i(t_k)$$

5. Si, por ejemplo, sólo quisiéramos dibujar la gráfica de la segunda componente $y_2(t)$ usaríamos la orden `plot` con la segunda columna de `y`:

```
plot(t, y(:,2))
legend('y_2(t)')
shg
```



En el análisis de las soluciones de sistemas diferenciales autónomos (que no dependen explícitamente del tiempo) con dos ecuaciones, interesa con frecuencia dibujar las soluciones en el *plano de fases*, es decir, en el plano (y_1, y_2), como en el ejemplo siguiente.

Ejercicio 8.6 Calcular (una aproximación de) la solución del sistema siguiente para $t \in [0, 15]$ y dibujar la solución en el plano de fases:

$$\begin{cases} y_1' = 0.5y_1 - 0.2y_1y_2 \\ y_2' = -0.5y_2 + 0.1y_1y_2 \\ y_1(0) = 4 \\ y_2(0) = 4 \end{cases}$$

1. Comenzamos por escribir el sistema en notación vectorial:

$$Y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad F(t, Y) = \begin{pmatrix} 0.5y_1 - 0.2y_1y_2 \\ -0.5y_2 + 0.1y_1y_2 \end{pmatrix}, \quad Y_0 = \begin{pmatrix} 4 \\ 4 \end{pmatrix},$$

2. Para ver cómo se haría, vamos a escribir la función del segundo miembro como una M-función. También se podría hacer como una función anónima, como en el ejercicio anterior.

```
function [dy] = odefun(t,y)
dy = zeros(2,1);
dy(1) =  0.5*y(1) - 0.2*y(1)*y(2);
dy(2) = -0.5*y(2) + 0.1*y(1)*y(2);
```

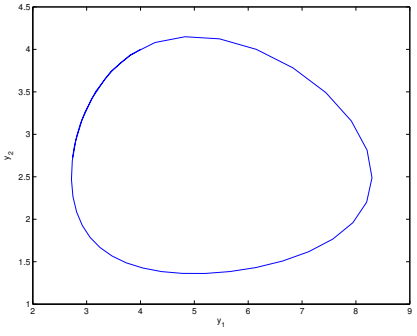
Observaciones:
a) Esta función debe ser guardada en un fichero de nombre `odefun.m`, que debe estar en la carpeta de trabajo de MATLAB para poder ser encontrada.
b) El nombre `odefun` es sólo un ejemplo.
c) La orden `dy = zeros(2,1);` en la función `odefun` tiene el objetivo de crear la variable `dy` desde un principio con las dimensiones adecuadas: un vector columna con dos componentes.

3. Calculamos la solución:

```
[t, y] = ode45(@odefun, [0, 15], [4; 4]);
```

4. Dibujamos la órbita de la solución en el plano de fases:

```
plot(y(:,1), y(:,2))
xlabel('y_1')
ylabel('y_2')
```



8.4 Resolución de problemas de Cauchy para ecuaciones diferenciales ordinarias de orden superior

En esta sección nos planteamos la resolución numérica de problemas de Cauchy para ecuaciones diferenciales ordinarias de orden superior a uno, es decir, problemas como:

$$\begin{cases} y^{(n)} = f(t, y, y', \ldots, y^{(n-1)}) \\ y(t_0) = y_0 \\ y'(t_0) = y_1 \\ \ldots \\ y^{(n-1)}(t_0) = y_{n-1} \end{cases} \tag{8.8}$$

La resolución de este problema se puede reducir a la resolución de un problema de Cauchy para un sistema diferencial de primer orden mediante el cambio de variables:

$$z_1(t) = y(t), \quad z_2(t) = y'(t), \quad \ldots, \quad z_n(t) = y^{(n-1)}.$$

En efecto, se tiene:

$$\begin{cases} z_1' = y' = z_2 \\ z_2' = y'' = z_3 \\ \ldots \\ z_n' = y^{(n)} = f(t, y, y', \ldots, y^{(n-1)}) = f(t, z_1, z_2, \ldots, z_{n-1}) \end{cases} \qquad \begin{cases} z_1(t_0) = y(t_0) = y_0 \\ z_2(t_0) = y'(t_0) = y_1 \\ \ldots \\ z_n(t_0) = y^{(n-1)}(t_0) = y_{n-1} \end{cases}$$

Para escribir este sistema en notación vectorial, denotamos:

$$Z = \begin{pmatrix} z_1 \\ z_2 \\ \ldots \\ z_n \end{pmatrix}, \quad F(t, Z) = \begin{pmatrix} z_2 \\ z_3 \\ \ldots \\ f_n(t, z_1, z_2, \ldots, z_{n-1}) \end{pmatrix}, \quad Z_0 = \begin{pmatrix} y_0 \\ y_1 \\ \ldots \\ y_{n-1} \end{pmatrix},$$

Con esta notación el problema planteado se escribe:

$$\begin{cases} Z' = F(t, Z) \\ Z(t_0) = Z_0 \end{cases} \tag{8.9}$$

y puede ser resuelto usando `ode45` como se ha mostrado en la sección anterior.

Ejercicio 8.7 Calcular y dibujar la solución del problema

$$\begin{cases} y'' + 7 \operatorname{sen} y + 0.1 \cos t = 0, \quad t \in [0, 5] \\ y(0) = 0 \\ y'(0) = 1 \end{cases}$$

Siguiendo los pasos antes indicados, este problema se puede reducir al siguiente:

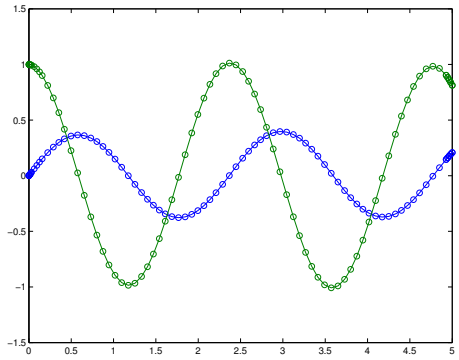
$$\begin{cases} Z' = F(t, Z) \\ Z(0) = Z_0 \end{cases}$$

con

$$F(t, Z) = \begin{pmatrix} z_2 \\ -7 \operatorname{sen} z_1 - 0.1 \cos t \end{pmatrix}, \quad Z_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

1. Comenzamos por usar `ode45` para dibujar la solución del sistema (2 componentes)

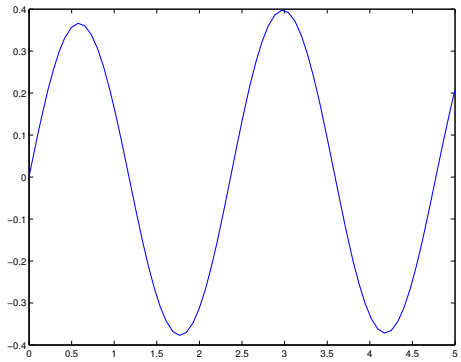
```
f = @(t,z) [z(2); -7*sin(z(1))-0.1*cos(t)];
ode45(f, [0,5], [0;1])
```



En esta gráfica están representadas las dos componentes de la solución $Z(t) = \begin{pmatrix} z_1(t) \\ z_2(t) \end{pmatrix}$

2. A nosotros sólo nos interesa la primera de las componentes de $Z(t)$, ya que es $y(t) = z_1(t)$. Por lo tanto usaremos `ode45` para recuperar los valores de la solución y dibujaremos únicamente la primera componente:

```
[t, z] = ode45(f, [0,5], [0;1]);
y = z(:, 1);
plot(t, y)    % o bien, directamente, plot(t, z(:, 1))
```



9 Resolución de problemas de contorno para sistemas diferenciales ordinarios

Exponemos aquí la forma de utilizar las herramientas de MATLAB para resolver **problemas de contorno** relativos a sistemas diferenciales ordinarios de primer orden. En estos problemas se busca encontrar una solución del sistema, definida en un intervalo $[a, b]$, que verifique unas condiciones adicionales que involucran los valores de la solución en ambos extremos del intervalo.

Concretamente, queremos calcular una solución del problema

$$\begin{cases} y' = f(x, y), & x \in [a, b] \\ g\big(y(a), y(b)\big) = 0. \end{cases} \tag{9.1}$$

Como es conocido, el problema (9.1) puede no tener solución, tener un número finito de soluciones o incluso tener infinitas soluciones.

Las mismas herramientas que permiten calcular (cuando la haya) una solución de (9.1) permiten también resolver problemas de contorno relativos a ecuaciones diferenciales ordinarias de orden superior, ya que, mediante un cambio de variable adecuado, estos problemas se pueden reducir al problema (9.1).

9.1 La función bvp4c

La función `bvp4c` utiliza, para resolver (9.1), un método de colocación: Se busca una solución aproximada $S(x)$ que es continua (de hecho, es de clase C^1) y cuya restricción a cada subintervalo $[x_i, x_{i+1}]$ de la malla determinada por la partición $a = x_0 < x_1 < \ldots < x_N = b$ es un polinomio cúbico.

A esta función $S(x)$ se le impone que verifique las condiciones de contorno

$$g\big(S(a), S(b)\big) = 0,$$

y, también, la ecuación diferencial en los extremos y el punto medio de cada subintervalo $[x_i, x_{i+1}]$. Esto da lugar a un sistema (no lineal) de ecuaciones que es resuelto por métodos iterados.

Por esta razón, y también porque el problema (9.1) puede tener más de una solución, es necesario proporcionar ciertas aproximaciones iniciales.

El uso básico de la función `bvp4c` es el siguiente:

```
sol = bvp4c(odefun, ccfun, initsol)
```

donde:

`odefun` es un manejador de la función que evalúa el segundo miembro de la ecuación, $f(x, y)$. Como en otros casos, puede ser el nombre de una función anónima o un manejador de una M-función.

`ccfun` es un manejador de la función que evalúa las condiciones de contorno $g(y_a, y_b)$

`initsol` es una estructura que contiene las aproximaciones iniciales. Esta estructura se crea mediante la función `bvpinit`, cuya descripción está más adelante.

`sol` es una estructura[1] que contiene la aproximación numérica calculada, así como algunos datos informativos sobre la resolución. Los campos más importantes de la estructura `sol` son:

- `sol.x` contiene la malla final del intervalo $[a, b]$.
- `sol.y` contiene la aproximación numérica de la solución $y(x)$ en los puntos `sol.x`.
- `sol.yp` contiene la aproximación numérica de la derivada de la solución $y'(x)$ en los puntos `sol.x`.
- `sol.stats` contiene información sobre las iteraciones realizadas por `bvp4c`.

```
initsol = bvpinit(xinit, yinit)
```

`xinit` es un vector que describe una malla inicial, i.e., una partición inicial $a = x_0 < x_1 < \ldots < x_N = b$ del intervalo $[a, b]$. Normalmente bastará con una partición con pocos puntos creada, por ejemplo, con `xinit = linspace(a,b,5)`. La función `bvp4c` refinará esta malla inicial, en caso necesario. Sin embargo, en casos difíciles, puede ser necesario proporcionar una malla que sea más fina cerca de los puntos en los que se prevea que la solución cambia rápidamente.

`yinit` es una aproximación inicial de la solución. Puede ser, o bien un vector, o bien una función. Se usará la primera opción cuando se desee proporcionar una aproximación inicial de la solución que sea constante. Si se desea proporcionar, como aproximación inicial, una función $y = y_0(x)$, se suministrará el nombre de una función que evalúe $y_0(x)$.

Ejercicio 9.1 Calcular una aproximación numérica de la solución de:

$$\begin{cases} y_1' = 3y_1 - 2y_2 \\ y_2' = -y_1 + \dfrac{1}{2}y_2 \\ y_1(0) = 0, \quad y_2(1) = \pi \end{cases}$$

1. Comenzamos por definir una función anónima que represente el segundo miembro de la ecuación $f(t, y)$:

```
dydx = @(x,y) [ 3*y(1) - 2*y(2); -y(1) + 0.5*y(2)];
```

[1] En MATLAB, una **estructura** es una *colección* de datos, organizados en *campos*, que se identifican mediante un *nombre de campo*. Ello permite manipular de forma compacta conjuntos de datos de distintos tipos, a diferencia de las matrices, en que todas las componentes deben ser del mismo tipo.

2. Definimos también una función anónima que evalúe la función que define las condiciones de contorno

```
condcon = @(ya, yb) [ ya(1); yb(2)-pi];
```

3. Construimos a continuación la estructura para proporcionar la aproximación inicial:

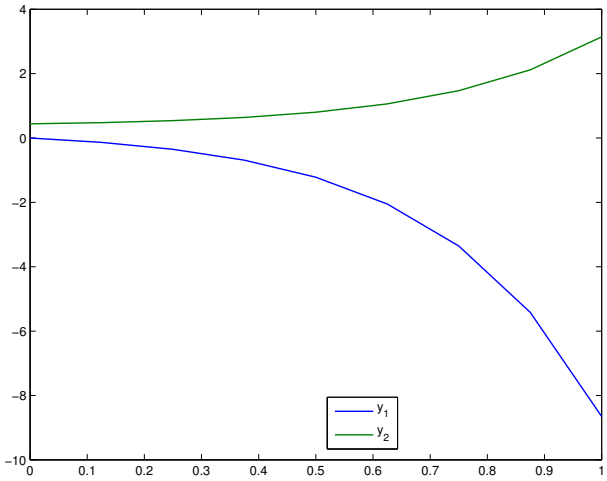
```
xinit  = linspace(0,1,5);
yinit  = [0; pi];
solinit = bvpinit(xinit, yinit);
```

4. Utilizamos ahora la función `bvp4c` para calcular la solución

```
sol = bvp4c(dydx, condcon, solinit);
```

Ahora, para dibujar las curvas de la solución $(y_1(x), y_2(x))$, utilizamos la estructura `sol`:
— `sol.x` es un vector fila que contiene los puntos $a = x_0 < x_1 < \ldots < x_N = b$ del soporte de la solución numérica
— `sol.y` es una matriz con dos filas; la primera fila `sol.y(1,:)` son los valores de $y_1(x)$ en los puntos `sol.x`; la segunda fila `sol.y(2,:)` son los valores de $y_2(x)$ en los puntos `sol.x`.
En consecuencia, para obtener la gráfica de la solución ponemos:

```
plot(sol.x, sol.y)
legend('y_1(x)', 'y_2(x)', 'Location', 'Best')
```



Observación: la forma en que se ha construido la gráfica anterior sólo utiliza los valores de la solución numérica en los puntos del soporte `sol.x`. **No hace uso del hecho de que la aproximación obtenida es un polinomio cúbico a trozos**. Para tener esto en cuenta, habría que usar la función `deval`, que se explica a continuación.

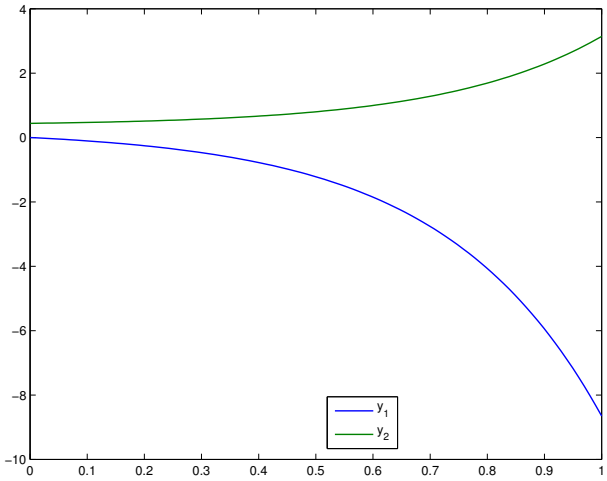
Para evaluar la solución obtenida con `bvp4c` en un punto o vector de puntos `xp`, se debe utilizar la función `deval`:

```
yp = deval(sol, xp);
```

donde `sol` es la estructura devuelta por `bvp4c`.

Ejercicio 9.1 (sigue...) Repetimos el ejercicio, utilizando una partición más fina del intervalo $[0,1]$ para realizar la gráfica, de forma que las curvas se vean más «suaves». Utilizamos la función `deval` para calcular los valores de la solución aproximada en los puntos de la partición.

```
function ejercicio1
%
dydx = @(x,y) [ 3*y(1) - 2*y(2); -y(1) + 0.5*y(2)];
condcon = @(ya, yb) [ ya(1); yb(2)-pi];
xinit  = linspace(0,1,5);
yinit  = [0; pi];
solinit = bvpinit(xinit, yinit);

sol = bvp4c(dydx, condcon, solinit);

xx = linspace(0,1);
yy = deval(sol, xx);
plot(xx,yy)
legend('y_1', 'y_2', 'Location', 'Best')

shg
```



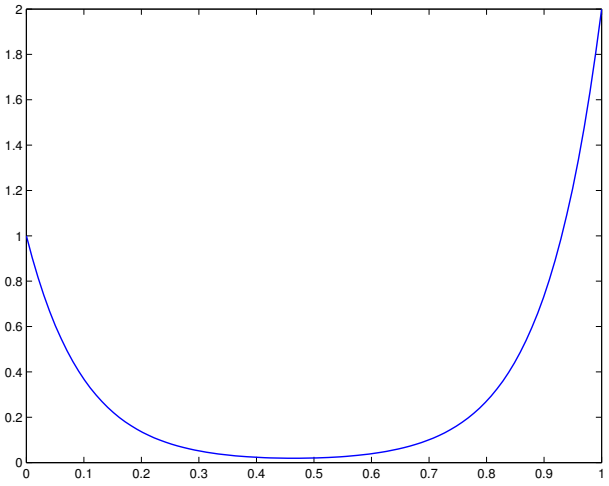
Ejercicio 9.2 Calcular la solución del problema

$$\begin{cases} -y'' + 100y = 0, & x \in [0,1] \\ y(0) = 1, \quad y(1) = 2 \end{cases} \tag{9.2}$$

Comenzamos por escribir el problema en forma de un problema de contorno para un SDO de primer orden. Hacemos el cambio de variable $z_1 = y, \quad z_2 = y'$ y el problema (9.2) se transforma en:

$$\begin{cases} z_1' = z_2 \\ z_2' = 100 z_1 \\ z_1(0) = 1, \quad z_1(1) = 2 \end{cases} \tag{9.3}$$

```
function ejercicio2
%
dydx = @(t,y) [ y(2); 100*y(1)];
condcon = @(ya, yb) [ ya(1)-1; yb(1)-2];
xinit  = linspace(0,1,5);
yinit  = [1.5; 0];
solinit = bvpinit(xinit, yinit);

sol = bvp4c(dydx, condcon, solinit);

xx = linspace(0,1);
yy = deval(sol, xx);
plot(xx, yy(1,:), 'LineWidth', 1.1)

shg
```



La función `bvp4c` permite, mediante un argumento opcional, modificar algunas de los parámetros que utiliza el algoritmo por defecto. Para ello hay que usar la función en la forma

```
sol = bvp4c(odefun, ccfun, initsol, options)
```

donde

`options` es una estructura que contiene los valores que se quieren dar a los parámetros opcionales. Esta estructura se crea mediante la función `bvpset` que se explica a continuación.

```
options = bvpset('nombre1', valor1, 'nombre2', valor2, ...)
```

`nombre1, nombre2` son los nombres de las propiedades cuyos valores por defecto se quieren modificar. Las propiedades no mencionadas conservan su valor por defecto.

`valor1, valor2` son los valores que se quieren dar a las propiedades anteriores.

Para una lista de las propiedades disponibles, véase el help de MATLAB. Los valores actuales se pueden obtener usando la función `bvpset` sin argumentos. Mencionamos aquí la propiedad que vamos a usar en el ejercicio siguiente, que nos permite determinar el nivel de precisión que `bvp4c` utilizará para detener las iteraciones y dar por buena una aproximación:

`RelTol` es el nivel de tolerancia de error relativo que se aplica a todas las componentes del vector residuo $r_i = S'(x_i) - f(x_i, S(x_i))$. Esencialmente, la aproximación se dará por buena cuando

$$\left\| \frac{S'(x_i) - f(x_i, S(x_i))}{\max\{|f(x_i, S(x_i))|\}} \right\| \leq RelTol$$

El valor por defecto de `RelTol` es 1.e-3.

`Stats` si tiene el valor `'on'` se muestran, tras resolver el problema, estadísticas e información sobre la resolución. El valor por defecto es `'off'`.

Ejercicio 9.3 La ecuación diferencial

$$y'' + \frac{3py}{\left(p + t^2\right)^2} = 0$$

tiene la solución analítica

$$\varphi(t) = \frac{t}{\sqrt{p + t^2}}.$$

Para $p = 10^{-5}$, resolver el problema de contorno

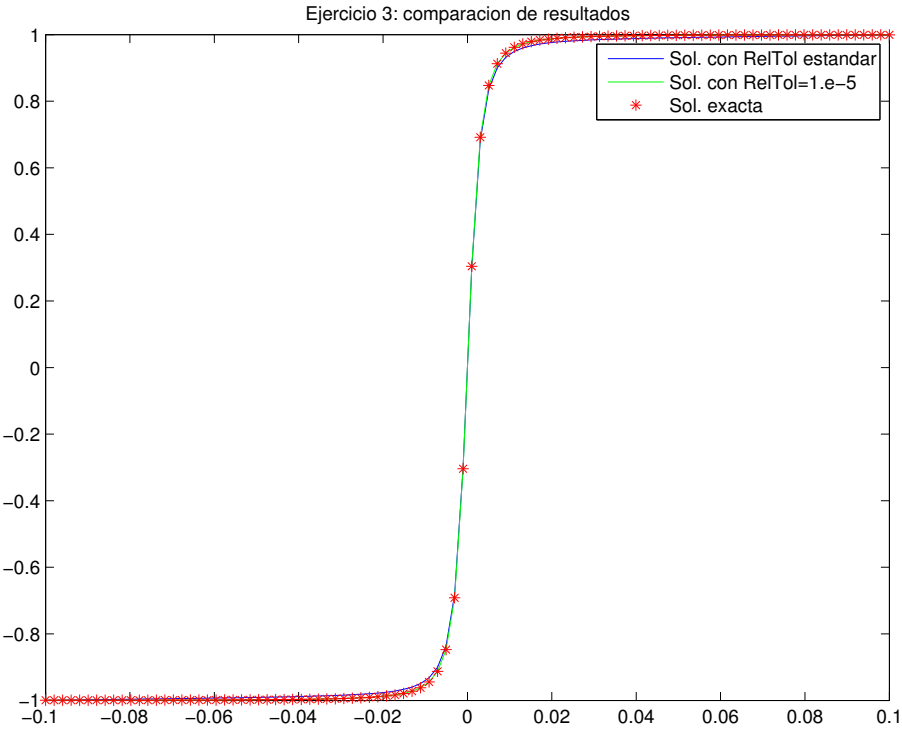
$$\begin{cases} y'' + \dfrac{3py}{\left(p + t^2\right)^2} = 0 \\ y(-0.1) = \varphi(-0.1), \quad y(0.1) = \varphi(0.1) \end{cases}$$

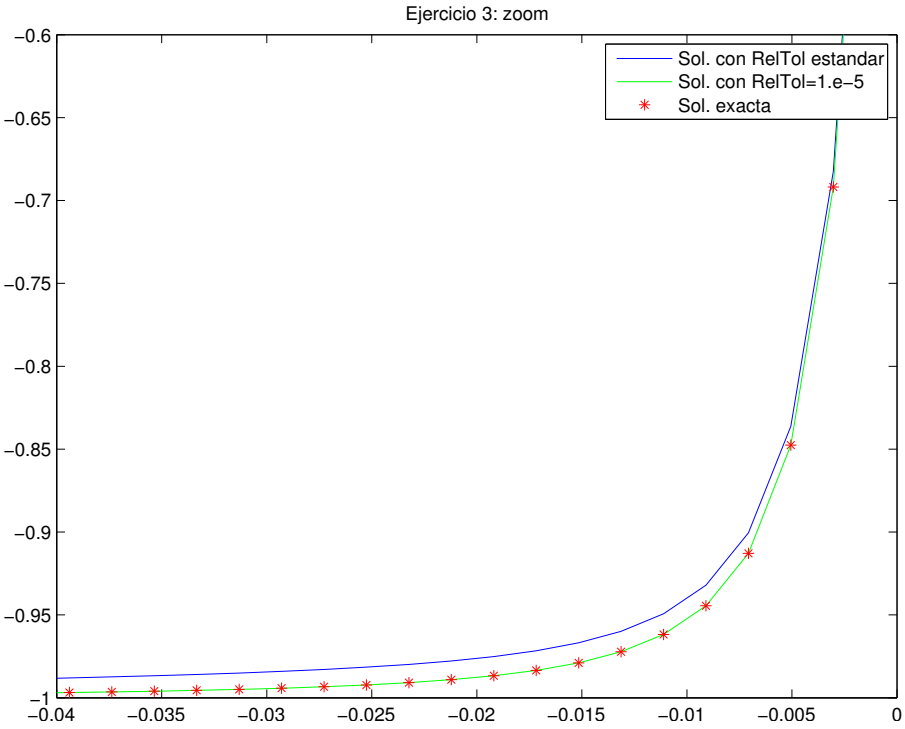
para el valor por defecto de `RelTol` y para `RelTol=1.e-5` y comparar con la solución exacta.

Comenzamos por escribir el problema en forma de un problema de contorno para un SDO de primer orden. Hacemos el cambio de variable $z_1 = y$, $z_2 = y'$ y el problema (9.2) se transforma en:

$$\begin{cases} z_1' = z_2 \\ z_2' = -\dfrac{3pz_1}{\left(p + t^2\right)^2} \\ z_1(-0.1) = \dfrac{-0.1}{\sqrt{p + 0.01}}, \quad z_1(0.1) = \dfrac{0.1}{\sqrt{p + 0.01}} \end{cases} \tag{9.4}$$

```
function ejercicio3

%   Parametros y constantes
%
p    = 1.e-5;
yenb = 0.1/sqrt(p+0.01);

%   Funciones anonimas
%
dydt    = @(t,y) [ y(2); -3*p*y(1)/(p+t^2)^2];
condcon = @(ya, yb) [ya(1)+yenb; yb(1)-yenb ];
exacta  = @(t) t./sqrt(p+t.^2);

%   Aproximaciones iniciales
%   La aproximacion inicial se ha obtenido promediando
%   las condiciones de contorno
%
tguess  = linspace(-0.1, 0.1, 10);
yguess  = [0; 10];
solinit = bvpinit(tguess, yguess);

%   Resolucion con valor estandar de RelTol
%
options = bvpset('Stats','on');
fprintf('\n')
fprintf('==> Resolucion con valor estandar de RelTol \n')
sol1 = bvp4c(dydt, condcon, solinit,options);

%   Resolucion con RelTol = 1.e-5
%
options = bvpset(options, 'RelTol', 1.e-5);
fprintf('\n')
fprintf('==> Resolucion con RelTol =%15.10f\n',bvpget(options,'RelTol'))
sol2 = bvp4c(dydt, condcon, solinit, options);

%   Graficas
%   Dibujamos solo la primera componente de la solucion del sistema
%
xx = linspace(-0.1, 0.1);
yy1 = deval(sol1, xx, 1);
yy2 = deval(sol2, xx, 1);
yye = exacta(xx);
plot(xx, yy1,'b', xx, yy2,'g', xx, yye,'r*')
shg
```


Ejercicio 3: comparacion de resultados
Sol. con RelTol estandar
Sol. con RelTol=1.e−5
Sol. exacta
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−0.1
−0.08
−0.06
−0.04
−0.02
0
0.02
0.04
0.06
0.08
0.1


Ejercicio 3: zoom
Sol. con RelTol estandar
Sol. con RelTol=1.e−5
Sol. exacta
−0.6
−0.65
−0.7
−0.75
−0.8
−0.85
−0.9
−0.95
−1
−0.04
−0.035
−0.03
−0.025
−0.02
−0.015
−0.01
−0.005
0

10 Resolución de problemas minimización

Exponemos aquí brevemente la forma de utilizar las herramientas de MATLAB para resolver **problemas de minimización con restricciones**.

Concretamente, queremos calcular una solución del problema

$$\begin{cases} \text{Minimizar } J(x) \\ \text{sujeto a } x \in \mathbb{R}^n \\ G_i(x) = 0 \quad i = 1, \ldots, m_e \\ G_i(x) \leq 0 \quad i = m_e + 1, \ldots, m \end{cases} \tag{10.1}$$

donde J es una función con valores reales.

10.1 La función fmincon

La función `fmincon`, del Optimization Toolbox de MATLAB, se puede utilizar para resolver el problema (10.1) en el caso en que tanto la función a minimizar como las restricciones (o algunas de ellas) son no lineales.

Para resolver el problema (10.1) con `fmincon` lo primero que hay que hacer es clasificar el conjunto de las restricciones ($G_i(x) = 0$, $i = 1, \ldots, m_e$, $G_i(x) \leq 0$, $i = m_e + 1, \ldots, m$) según los tipos siguientes:

- Cotas: $c_i \leq x \leq c_s$
- Igualdades lineales: $Ax = b$, donde A es una matriz
- Desigualdades lineales: $Cx \leq d$, donde C es una matriz
- Igualdades nolineales: $f(x) = 0$
- Desigualdades no lineales: $g(x) \leq 0$

Cada tipo de restrición habrá que incorporarla de manera distinta a la función `fmincon`.

Los usos más sencillos de la función `fmincon` son los siguientes:

```
x = fmincon(funJ, x0, C, d)
x = fmincon(funJ, x0, C, d, A, b)
x = fmincon(funJ, x0, C, d, A, b, ci, cs)
x = fmincon(funJ, x0, C, d, A, b, ci, cs, resnolin)
x = fmincon(funJ, x0, C, d, A, b, ci, cs, resnolin, options)
```

donde:

`funJ` es un manejador de la función a minimizar $J(x)$. Como en otros casos, puede ser el nombre de una función anónima o un manejador de una M-función.

`x0` es una aproximación inicial de la solución.

`C, d` son, respectivamente, la matriz y el segundo miembro que definen las restricciones lineales de desigualdad $Cx \leq d$.

`A, b` son, respectivamente, la matriz y el segundo miembro que definen las restricciones lineales de igualdad $Ax = b$. Si hay que utilizar `A` y `b` y nuestro problema no tiene restricciones con desigualdades lineales, se pone `C=[]` y `b=[]`.

`ci, cs` son, respectivamente, las cotas inferior y superior $c_i \leq x \leq c_s$. Si no hay cota inferior para alguna de las variables se especificará `-Inf`. Análogamente, si no hay cota superior, se pondrá `Inf`. Como antes, si alguno de los argumentos anteriores (`C`, `d`, `A`, `b`) no se usa, se utilizarán matrices vacías (`[]`) en su lugar.

`resnolin` es un manejador de la función que define las restricciones no lineales, tanto de igualdad $f(x) = 0$ como de desigualdad $g(x) \leq 0$. Puede ser el nombre de una función anónima o un manejador de una M-función. Esta función debe aceptar como argumento de entrada un vector `x` y devolver dos vectores: `c` = $g(x)$, `ceq` = $f(x)$:

```
function [c, ceq] = resnolin(x)    %  o bien
resnolin = @(x) [c, ceq]
```

`options` es una estructura que permite modificar algunos de los parámetros internos de la función `fmincon`. Esta estructura se crea mediante la función:

```
options = optimset('Opcion1','valor1','Opcion2','valor2', ...)
```

Aunque son numerosas las opciones que se pueden modificar, mencionamos aquí sólo dos:

`Algorithm` permite elegir entre cuatro algoritmos, en función de las caracterísitcas y/o dificultad del problema a resolver. Los posibles valores son: `'interior-point'`, `'sqp'`, `'active-set'` y `'trust-region-reflective'`.

`Display` permite elegir el nivel de *verbosity*, es decir, de información suministrada durante la ejecución. Posibles valores son: `'off'`, `'iter'`, `'final'` y `'notify'`.

Ejercicio 10.1 Calcular una aproximación de la solución de:

$$\begin{cases} \text{Minimizar } f(x) = -x_1\, x_2\, x_3 \\ \text{sujeto a} \\ 0 \leq x_1 + 2x_2 + 3x_3 \leq 72 \end{cases}$$

partiendo de la aproximación inicial $x_0 = (10, 10, 10)$.

Re-escribimos las restricciones.

$$\begin{cases} -x_1 - 2x_2 - 3x_3 \leq 0 \\ x_1 + 2x_2 + 3x_3 \leq 72 \end{cases}$$

Se trata en ambos casos de restricciones lineales de desigualdad, que se pueden expresar en la forma

$$Cx \leq d \quad \text{con } C = \begin{pmatrix} -1 & -2 & -3 \\ 1 & 2 & 3 \end{pmatrix}, \quad d = \begin{pmatrix} 0 \\ 72 \end{pmatrix}$$

```
funJ = @(x) - prod(x);

C = [-1, -2, -3; 1, 2, 3];
d = [0; 72];
ci = [-Inf; -Inf; -Inf];
cs = [ Inf;  Inf;  Inf];
x0 = [10; 10; 10];
options = optimset('Algorithm', 'interior-point', 'Display', 'iter');

x = fmincon(funJ, x0, C, d, [], [], ci, cs,[],options)
```

Ejercicio 10.2 Calcular una aproximación de la solución de:

$$\begin{cases} \text{Minimizar } f(x) = -x_1\, x_2\, x_3 \\ \text{sujeto a} \\ 0 \leq x_1 + 2x_2 + 3x_3 \leq 72 \\ x_1^2 + x_2^2 + x_3^2 \leq 500 \end{cases}$$

partiendo de la aproximación inicial $x_0 = (10, 10, 10)$.

Las restricciones lineales son las mismas del ejercicio anterior. La restricción no lineal hay que escribirla en forma homogénea

$$x_1^2 + x_2^2 + x_3^2 - 500 \leq 0$$

y escribir una M-función para definir dichas restricciones:

```
function [c, ceq] = resnolin(x)
%
%  funcion definiendo las restricciones no lineales
%
c = sum(x.^2)-500;
ceq = 0;
```

Para resolver el problema,

```
funJ = @(x) - prod(x);
C = [-1, -2, -3; 1, 2, 3];
d = [0; 72];
```

```
ci = [-Inf; -Inf; -Inf];
cs = [ Inf;  Inf;  Inf];
x0 = [10; 10; 10];
options = optimset('Algorithm', 'interior-point', 'Display', 'iter');
x = fmincon(funJ, x0, C, d, [], [], ci, cs, @resnolin, options)
```