1. Práctica 1

Las dos primeras prácticas son, en realidad, una introducción a MATLAB y, en esta primera, empezaremos por ver cómo hacer cálculos básicos, tanto con escalares como con matrices o vectores, para finalizar con el manejo de funciones.

1.1. Operaciones básicas

En MATLAB, los cálculos como sumas, restas, multiplicaciones y divisiones se hacen, como con cualquier calculadora, utilizando `+`, `-`, `*` y `/`, mientras que, para hacer potencias, el exponente se escribe tras el símbolo `^`. Debe tenerse en cuenta que la jerarquía de las operaciones se estructura por niveles en el orden siguiente: en primer lugar las operaciones entre paréntesis, luego exponentes, después productos y cocientes, y, finalmente, sumas y restas. Dentro de un mismo nivel, las operaciones se realizan en el orden en el que aparecen escritas de izquierda a derecha:

```
>> 2^2+2+4*3/2
ans =
    12
>> 2^(2+2)+4*3/2
ans =
    22
```

Como se puede observar, el programa almacena el último resultado en la variable `ans` (abreviatura de *answer*), si bien ese nombre se puede cambiar, haciendo, por ejemplo:

```
>> a=2+3
a =
     5
```

El símbolo % permite introducir comentarios en una instrucción, ignorándose todo lo que se escribe a su derecha:

```
>> b=2*a % Cálculo del doble de a
b =
    10
```

Debe tenerse en cuenta que los nombres de variables son sensibles a las mayúsculas, deben comenzar siempre con una letra, no pueden contener espacios en blanco y pueden nombrarse hasta con 63 caracteres, que deben ser letras, números o el símbolo `_` de subrayado. Si se nombra una variable con más de 63 caracteres se truncará el nombre de dicha variable.

Aunque es posible, no es conveniente usar como nombres de variables las que MATLAB ya tiene predefinidas como, por ejemplo, las siguientes:

pi: Da como resultado el valor del número π.

inf: Es la abreviatura de infinito y aparece como resultado cuando se intenta representar un número demasiado grande.

NaN: Es la abreviatura de *Not a Number* y aparece cuando un cálculo da lugar a una indeterminación.

i o **j**: Representa $\sqrt{-1}$, es decir, la unidad imaginaria.

Es posible hacer varios cálculos en una misma línea, separándolos por comas (`,`) que hace que se visualice el resultado, o puntos y comas (`;`) que omiten el resultado de la operación, pese a que ésta se realiza:

```
>> a=2+1, b=3*2; c=3^2
a =
     3
c =
     9
>> b+1
ans =
     7
```

Cuando se tienen varias variables almacenadas, podemos pedir información al programa sobre una o más variables, haciendo

```
>> whos a b
  Name      Size            Bytes  Class     Attributes
  a         1x1                 8  double
  b         1x1                 8  double
```

o pedir información sobre todas las variables, con

```
>> whos
  Name      Size            Bytes  Class     Attributes
  a         1x1                 8  double
  ans       1x1                 8  double
  b         1x1                 8  double
  c         1x1                 8  double
```

Para borrar variables, se utiliza **clear**. Si se ejecuta `clear variable1 variable2 ...`, se eliminan las variables especificadas, mientras que, si ejecutamos `clear`, se borran todas las variables:

```
>> clear a b
>> a+1
??? Undefined function or variable 'a'.
>> c+2
ans =
    11

>> clear
>> 2*c
??? Undefined function or variable 'c'.
```

No debe confundirse **clear** con **clc**, que borra la pantalla, pero no las variables almacenadas:

```
>> a=3
a =
     3
>> clc
>> a+1
ans =
     4
```

Para finalizar esta sección, indiquemos que el signo igual (=), que se utiliza para asignar un valor a una variable, no tiene el significado de igualdad matemática. Así, por ejemplo, `n=n+1` sería incoherente matemáticamente, pero tiene sentido para MATLAB, si la variable `n` tiene previamente asignado algún valor:

```
>> n=1
n =
     1
>> n=n+1
n =
     2
```

1.2. Vectores y matrices

Para crear un vector introducimos, entre corchetes, los valores deseados separados por espacios o comas:

```
>> v=[2 4 6 8 10 12 14]
v =
     2     4     6     8    10    12    14
```

Si ahora queremos acceder a un elemento del vector, basta ejecutar `v(n)`, donde `n` indica el número de coordenada que deseamos:

```
>> v(3)
ans =
     6
```

Si ejecutamos la orden `v([n1 n2 ...])`, obtenemos los elementos de `v` que ocupan las posiciones `n1`, `n2`, etc:

```
>> v([4 5 1])
ans =
     8    10     2
```

Para definir la posición deseada, también se puede utilizar `end`, `end-1`, ..., como podemos ver, al hacer:

```
>> v([1 end-2 end])
ans =
     2    10    14
```

Las órdenes anteriores, además de para obtener los elementos deseados del vector, sirven para modificarlos, si se les asigna un valor,

```
>> v([1 end])=[0 -1]
v =
     0     4     6     8    10    12     -1
```

y también para eliminarlos, haciendo:

```
>> v([1 end])=[]
v =
     4     6     8    10    12
```

Se pueden generar vectores fila con las órdenes:

`linspace(a,b,n)`: construye un vector de `n` elementos igualmente espaciados, siendo `a` el primer elemento y `b` el último. Si no se especifica `n`, su valor por defecto es 100.

`a:h:b`: construye el vector de mayor tamaño posible, de elementos `a`, `a+h`, `a+2h`, ..., `a+kh`, de forma que el último elemento, `a+kh`, esté comprendido entre `a` y `b`. Si no se especifica `h`, su valor por defecto es 1.

A continuación, vemos algunos ejemplos de uso de estas órdenes:

```
>> linspace(4,14,6)
ans =
     4     6     8    10    12    14
>> 9:-2:3
ans =
     9     7     5     3
>> 3:2:10
ans =
     3     5     7     9
>> 3:-2:10
ans =
   Empty matrix: 1-by-0
>> 2:5
ans =
     2     3     4     5
```

Veamos ahora cómo construir matrices: la forma es análoga a la de construcción de vectores, sin más que separar por punto y coma (;) los vectores que forman las distintas filas de la matriz. Por ejemplo, podemos hacer

```
>> A=[0 1 2;2 3 5;3 4 6]
A =
     0     1     2
     2     3     5
     3     4     6
>> B=[7 7;8 8;9 9]
B =
     7     7
     8     8
     9     9
>> C=[A B]
C =
     0     1     2     7     7
     2     3     5     8     8
     3     4     6     9     9
```

y, como se puede ver en el ejemplo anterior, es posible unir matrices, siempre que sus dimensiones sean coherentes.

El método para acceder a los elementos de una matriz es similar al usado con vectores, salvo que ahora la posición de los elementos deseados se fija con dos índices separados por una coma, antes de la cual, se deben indicar las filas a las que se quiere acceder y, tras la coma, se deben indicar las columnas:

```
>> C(2,3)
ans =
     5
>> C([1 3],2)
ans =
     1
     4
>> C(1:2,3:5)
ans =
     2     7     7
     5     8     8
```

Si se usa el signo dos puntos (:) como indicador de una línea, se obtienen todos los elementos de esa línea:

```
>> C(1:2,:)
ans =
     0     1     2     7     7
     2     3     5     8     8
>> C(:,3)
ans =
     2
     5
     6
```

Al igual que en el caso de vectores, las órdenes anteriores sirven para modificar o eliminar elementos de la matriz:

```
>> C(1,[4 5])=[0 0]
C =
     0     1     2     0     0
     2     3     5     8     8
     3     4     6     9     9
>> C(:,5)=[]
C =
     0     1     2     0
     2     3     5     8
     3     4     6     9
```

MATLAB permite generar matrices con distintas órdenes, como por ejemplo:

`zeros(m,n)`: construye una matriz de ceros, de `m` filas y `n` columnas; si no se especifica `m`, se toma `m=n`.

`ones(m,n)`: es el equivalente a la orden anterior para construir una matriz de unos.

`eye(n)`: construye la matriz unidad, de `n` filas y `n` columnas.

Así, podemos hacer:

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

Las operaciones que se realicen con vectores y matrices se hacen en el sentido matricial, para lo cual será necesario que las dimensiones sean coherentes, si bien hay una excepción: Si a una matriz (o a un vector, como caso particular de matriz) se le suma o resta un escalar (cosa que, matemáticamente, no tiene sentido), MATLAB suma o resta el escalar a todos los elementos de la matriz:

```
>> A=[1 2;1 1]
A =
     1     2
     1     1
>> v=[1;-1]
v =
     1
    -1
>> A*v
ans =
    -1
     0
```

```
>> v*A
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>> A^2
ans =
     3     4
     2     3
>> A^(-1)
ans =
    -1     2
     1    -1
>> A+2
ans =
     3     4
     3     3
```

Para finalizar este apartado, se puede hacer lo que se denomina *operar elemento a elemento* con una matriz. Para ello, las operaciones como `*`, `/` o `^` deben ir precedidas de un punto, como se puede ver en los siguientes ejemplos:

```
>> A=[1 2;3 4], B=[0 1;2 3]
A =
     1     2
     3     4
B =
     0     1
     2     3
>> A.*B
ans =
     0     2
     6    12
>> A.^B
ans =
     1     2
     9    64
>> A.^2
ans =
     1     4
     9    16
>> 4./[1 2 4]
ans =
     4     2     1
```

1.3. Funciones en MATLAB

En esta sección, se estudia la utilización de funciones en MATLAB. Describiremos, en primer lugar, algunas de las funciones más habituales, para, a continuación, centrarnos en las formas de definir nuestras propias funciones y de hacer cálculos con las mismas.

1.3.1. Funciones predefinidas

MATLAB posee una enorme lista de funciones predefinidas de las que, a continuación, vamos a destacar las que utilizaremos más frecuentemente.

Funciones que operan con vectores y matrices:

`length(v)`: Halla el número de elementos del vector `v`.

`[m n]=size(A)`: Halla el número de filas (`m`) y de columnas (`n`) de la matriz `A`.

`sum(v)`: Halla la suma de los elementos del vector `v`.

`sum(A)`: Halla el vector de sumas de los elementos de cada columna de la matriz `A`.

`prod(v)`: Halla el producto de los elementos del vector `v`.

`prod(A)`: Halla el vector de productos de los elementos de cada columna de la matriz `A`.

`max(v)`: Halla el máximo de los elementos del vector `v`.

`max(A)`: Halla el vector de máximos de los elementos de cada columna de la matriz `A`.

`min(v)`: Halla el mínimo de los elementos del vector `v`.

`min(A)`: Halla el vector de mínimos de los elementos de cada columna de la matriz `A`.

`inv(A)`: Halla la inversa de la matriz regular `A`.

`A'`: Halla la traspuesta conjugada de la matriz `A`.

`A.'`: Halla la traspuesta de la matriz `A`.

Funciones matemáticas:

`abs(x)`: Halla el valor absoluto de `x`, si `x` es real, y el módulo de `x`, si `x` es complejo.

`exp(x)`: Halla e^x.

`log(x)`: Halla el logaritmo neperiano de `x`.

`log10(x)`: Halla el logaritmo decimal de `x`.

`log2(x)`: Halla el logaritmo en base 2 de `x`.

`sqrt(x)`: Halla la raíz cuadrada de `x`.

`nthroot(x,n)`: Halla la raíz `n`-ésima de `x`.

`sin(x)`, `cos(x)`, `tan(x)`, `cot(x)`, `sec(x)` y `csc(x)`: Hallan, respectivamente, seno, coseno, tangente, cotangente, secante y cosecante del ángulo (en radianes) `x`.

`asin(x)`, `acos(x)`, `atan(x)`, ...: Funciones trigonométricas inversas de las anteriores, con resultado en radianes.

1.3.2. Definición de funciones

Además de las funciones anteriores, en MATLAB, el usuario puede definir sus propias funciones, que pueden ser de tipo *numérico* y de tipo *simbólico*. La forma numérica está pensada para evaluar la función en uno o más puntos, lo más rápidamente posible, mientras que la forma simbólica es útil para hacer cálculos simbólicos, como derivadas, integrales, etc.

Comencemos estudiando cómo definir funciones de forma simbólica: Una forma es declarar previamente como simbólicas las variables a utilizar y, seguidamente, definir la función por su relación con dichas variables. Para declarar variables simbólicas, la forma más sencilla es ejecutar la orden **syms**, escribiendo, a continuación, y separadas por espacios en blanco, las variables que se deseen. Así, podemos construir $f(x) = x^4$ y $g(x, y) = x^2 - y^3$, sin más que hacer

```
>> syms x y
>> f=x^4
f =
x^4
>> g=x^2-y^3
g =
x^2 - y^3
```

y ahora podemos hacer operaciones simbólicas con f y g, como derivadas e integrales, por ejemplo.

Para derivar, se utiliza el comando **diff**. Cuando se trabaja con funciones de una variable, basta utilizar la orden `diff(función)` para hallar la primera derivada y `diff(función,n)` para calcular la derivada enésima:

```
>> diff(f)
ans =
4*x^3
>> diff(f,3)
ans =
24*x
```

Si la función es de varias variables, las derivadas se hacen respecto de la variable principal, que es la más cercana a x, en el orden alfabético, salvo que especifiquemos otra, en cuyo caso, el formato pasa a ser `diff(función,variable)` para calcular la parcial primera de una función y `diff(función,variable,n)` para hallar la parcial enésima:

```
>> diff(g)
ans =
2*x
>> diff(g,y,2)
ans =
-6*y
```

Para integrar, se usa la orden **int**. En el caso de funciones de una variable, con `int(función)`, hallamos una primitiva de una función y, para calcular la integral definida de una función, la orden pasa a ser `int(función,límite_inferior,límite_superior)`:

```
>> int(f)
ans =
x^5/5
>> int(f,0,1)
ans =
1/5
```

Si la función a integrar es de varias variables, debemos especificar la variable de integración, por lo que el formato de las órdenes anteriores pasa a ser `int(función,variable)` e `int(función,variable,límite_inferior,límite_superior)`. Al igual que en la orden anterior, si no se especifica dicha variable de integración, MATLAB calculará la integral respecto de la variable principal:

```
>> int(g)
ans =
x^3/3 - x*y^3
>> int(g,y)
ans =
x^2*y - y^4/4
>> int(g,y,0,1)
ans =
x^2 - 1/4
```

Calcular el valor de una función simbólica en un punto requiere la orden **subs**, con el formato `subs(z,t,a)`, que ordena al programa obtener el resultado de sustituir, en `z`, la variable `t` por el valor `a`. Así, siguiendo con $f(x) = x^4$ y $g(x, y) = x^2 - y^3$, para hallar $f(2)$ o $g(3, 2)$ no es válido ejecutar `f(2)` ni `g(3,2)`, sino que debemos hacer:

```
>> subs(f,x,2)
ans =
    16
```

```
>> subs(g,[x y],[3 2])
ans =
    1
```

Con la orden **subs**, cuando se ordena calcular el valor de la función en una matriz o en un vector, como caso particular de matriz, las operaciones se hacen elemento a elemento, calculando el valor de la función en cada elemento de la matriz, como se puede observar en el siguiente ejemplo:

```
>> subs(f,x,[1 2;3 4])
ans =
     1    16
    81   256
```

Seguidamente, pasamos a estudiar las funciones de tipo numérico: Además de definirlas por medio de un archivo, método que estudiaremos más adelante, una forma de construir una función, directamente en la ventana de órdenes, es utilizar las llamadas *funciones anónimas*. Para ello, la función se debe definir como `@(variables) expresión`, donde las variables deben ir separadas por comas y `expresión` ha de ser la definición, en función de las variables de las que dependa. Por ejemplo, para construir la función $f(x) = x^2$, bastaría con

```
>> f=@(x) x^2
f =
    @(x)x^2
```

y hallar el valor de la función en un punto, ahora es más sencillo ya que basta con, por ejemplo:

```
>> f(2)
ans =
     4
```

Con este tipo de función no es posible hacer operaciones simbólicas y, si intentamos aplicarles **diff** o **int**, obtendremos un mensaje de error. Sin embargo, pasar de un formato a otro es sencillo; por ejemplo, si queremos pasar $f(x) = x^2$ a la forma simbólica, podríamos hacer

```
>> syms x
>> f_sim=f(x)
f_sim =
x^2
```

y ahora ya podemos derivar o integrar, sin más que aplicar:

```
>> diff(f_sim)
ans =
2*x
>> int(f_sim)
ans =
x^3/3
```

Supongamos ahora que construimos la función simbólica $g(x) = x^3$, por medio de

```
>> syms x, g=x^3
g =
x^3
```

y que deseamos pasarla a formato numérico. Una forma de hacerlo sería utilizar el comando **matlabFunction**, en el que debe respetarse la `F` en mayúsculas. Así, con

```
>> g_num=matlabFunction(g)
g_num =
    @(x)x.^3
```

queda definida como función anónima, con la ventaja de que queda lista para operar elemento a elemento y calcular, por ejemplo, el valor de la función en 2, 3 y 5, haciendo

```
>> g_num([2 3 5])
ans =
     8    27   125
```

Debe tenerse en cuenta que si hubiéramos definido directamente

```
>> g=@(x) x^3
g =
    @(x)x^3
```

al hacer

```
>> g([2 3 5])
??? Error using ==> mpower
Matrix must be square.
Error in ==> @(x)x^3
```

obtenemos un mensaje de error, puesto que el vector `[2 3 5]` se interpreta como una matriz, y MATLAB nos advierte de que, para hallar el cubo de una matriz, ésta debe ser cuadrada.

Recordemos que, si queremos aplicar una función numérica a un vector y hallar el valor de la función en cada coordenada del vector, al definirla, las operaciones como `^`, `*` o `/` deben ir precedidas de un punto. En el ejemplo que estamos considerando, que es muy sencillo, bastaría con ejecutar la orden `g=@(x) x.^3`, pero, en casos más complicados, puede ser útil definirla en forma simbólica, en primer lugar, y con el método que acabamos de estudiar, pasarla a formato numérico.

Para finalizar este apartado, indicar que también se pueden definir funciones anónimas de varias variables. Por ejemplo, para definir $h(x, y) = x^2 + y^3$ y calcular $h(2, -1)$, basta hacer:

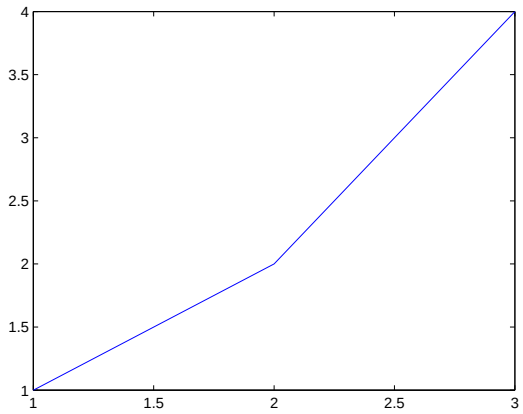
```
>> h=@(x,y) x^2+y^3
h =
    @(x,y)x^2+y^3
>> h(2,-1)
ans =
     3
```

1.3.3. Representaciones gráficas

Para terminar con esta sección dedicada al manejo de funciones, veamos cómo representar gráficamente una función, utilizando la orden **plot**:

Si se ejecuta la orden `plot([x1,x2,x3,...],[y1,y2,y3,...])` el programa representa los segmentos que unen los puntos, (`x1`,`y1`), (`x2`,`y2`), (`x3`,`y3`),..., como se puede ver en el siguiente ejemplo:

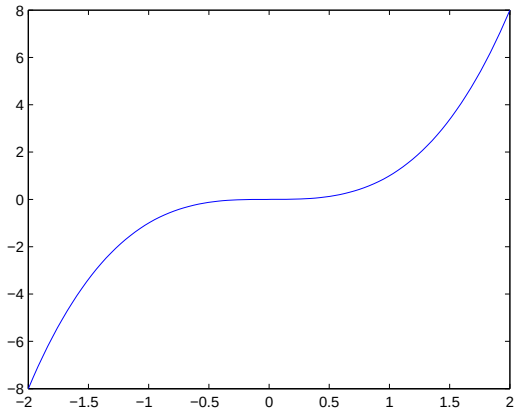
```
>> plot([1 2 3],[1 2 4])
```



Así, para representar, por ejemplo, $f(x) = x^3$, en el intervalo $[-2, 2]$, basta hacer:

```
>> f=@(x) x.^3
f =
    @(x)x.^3
```

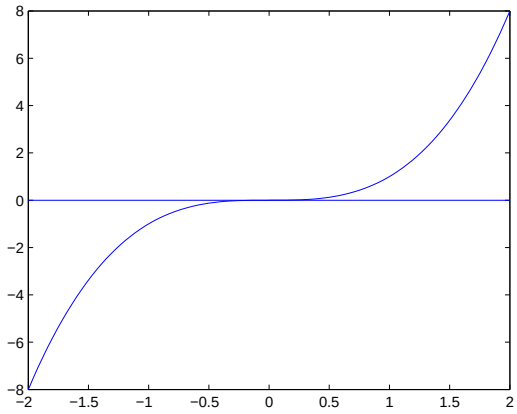
```
>> xx=linspace(-2,2);
>> plot(xx,f(xx))
```



Si tenemos abierta una ventana gráfica, al ejecutar una nueva orden **plot**, la gráfica antigua se sustituye por la nueva, salvo que apliquemos el comando **hold on**, en cuyo caso, a partir de ahí, todas las gráficas se harán en la misma ventana, hasta que se cierre la misma o se ejecute la orden **hold off**.

Así, si a la gráfica anterior le queremos añadir el eje horizontal, podemos hacer:

```
>> plot(xx,f(xx))
>> hold on, plot([-2 2],[0 0])
```



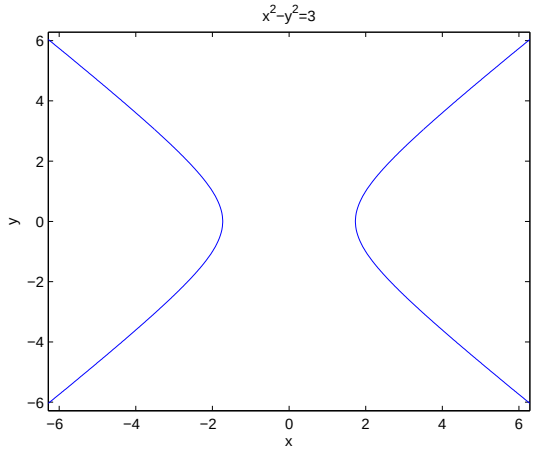
Por supuesto, es posible, aunque no se verá en este momento, modificar características de las gráficas, tales como el color o el tipo de trazo, añadir un título, etc.

Para acabar con esta breve introducción a las gráficas de funciones, indiquemos cómo representar funciones implícitas.

Una forma es utilizar el comando **ezplot**. Para ello, escribimos la expresión de dos variables que define la función implícita como una cadena de texto, es decir, con los caracteres que la componen entrecomillados con la comilla simple (') y ejecutamos la orden `ezplot(función)`, con lo que se genera la gráfica, considerando la variable principal como variable independiente, con ambas variables en el intervalo $[-2\pi, 2\pi]$.

Por ejemplo, para representar la hipérbola $x^2 - y^2 = 3$, basta con ejecutar:

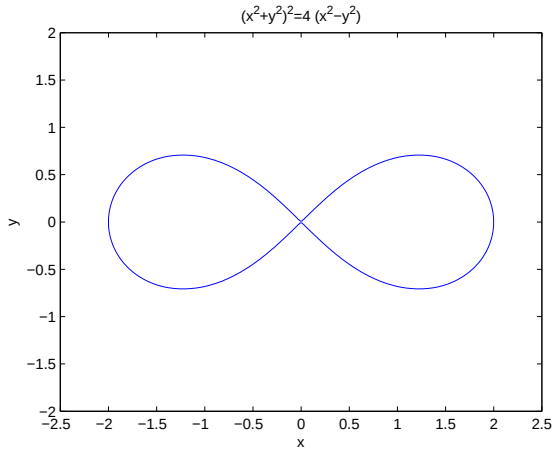
```
>> ezplot('x^2-y^2=3')
```



Si ejecutamos `ezplot(función,[a b])`, el intervalo de variación de la variable independiente pasa a ser `[a,b]` y se pueden especificar distintos intervalos para las variables, utilizando la orden `ezplot(función,[a b c d])`, que representa los puntos de la gráfica para los que la variable independiente pertenece al intervalo `[a,b]` y la dependiente al intervalo `[c,d]`.

Veamos, como ejemplo, la gráfica de la lemniscata de Bernoulli $(x^2 + y^2)^2 = 4(x^2 - y^2)$:

```
>> ezplot('(x^2+y^2)^2=4*(x^2-y^2)',[-2.5 2.5 -2 2])
```



2. Práctica 2

Siguiendo con la introducción a MATLAB, en esta práctica, estudiaremos, en primer lugar, las funciones definidas en ficheros, con diversos ejemplos en los que introduciremos las órdenes básicas de programación y, a continuación, la precisión de los cálculos y sus distintas formas de presentación.

2.1. Ficheros de función

Podemos crear dos tipos de archivo con extensión m: uno es el fichero de comandos y el otro el fichero de función.

El de comandos contiene simplemente un conjunto de órdenes, que se ejecutan sucesivamente al teclear el nombre del fichero en la línea de comandos.

Así, si creamos un fichero datos.m con las órdenes

```
A=[1 2;3 4]
B=[1 2; 1 3]
C=A+B^2
```

al escribir `datos` en la ventana de órdenes y pulsar Enter, resulta:

```
>> datos
A =
     1     2
     3     4
B =
     1     2
     1     3
C =
     4    10
     7    15
```

Los archivos de función permiten definir funciones numéricas análogas a las de MATLAB, con su nombre, argumentos y valores de salida. La primera línea que no sea comentario debe empezar por la palabra function, seguida por los valores de salida (entre corchetes), el signo igual (=) y el nombre de la función seguido de los argumentos de entrada, entre paréntesis y separados por comas, con lo que esa línea sería de la forma

```
function [y_1 y_2 ... y_m]=nombre(x_1,x_2, ... ,x_n)
```

siendo `x_1,x_2, ... ,x_n` los argumentos de entrada de la función e `y_1,y_2, ... ,y_m` las variables de salida; `nombre` es el nombre que asignemos a la función que, aunque no es imprescindible, es muy aconsejable que coincida con el nombre del fichero en el que se va a guardar

la función; es decir, sería recomendable que el fichero se guardase como nombre.m. En las siguientes líneas, deben definirse las variables de salida por sus relaciones con las variables de entrada.

Una vez creado el fichero, ejecutando `[y_1 y_2 ... y_m]=nombre(x_1,x_2, ... ,x_n)`, donde a las variables de entrada se les habrán asignado los valores que se deseen, MATLAB responderá con los valores calculados para las variables de salida.

Ejemplo 2.1 *Constrúyase un fichero de función, de nombre circulo.m, que tenga el radio de un círculo como variable de entrada y su área como variable de salida y, a continuación, utilícese para hallar el área de un círculo de radio* 2*.*

Solución: Creamos el fichero circulo.m, con las órdenes

```
function a=circulo(r)
% Función a=circulo(r) para hallar el área
% de un círculo, a partir de su radio r
a=pi*r^2;
```

donde debe observarse que la línea en la que se define `a` termina con"`;`"para que, al hacer ese cálculo, no se muestre por pantalla el resultado del mismo.

También se ha escrito un comentario (tras el símbolo `%`), que sirva como ayuda cuando se ejecuta `help circulo`:

```
>> help circulo
  Función a=circulo(r) para hallar el área
  de un círculo, a partir de su radio r
```

Por último, para hallar el área de un círculo de radio 2, ejecutamos la orden:

```
>> area=circulo(2)
area =
   12.5664
```

Nota: Como sólo hay una variable de salida, no es imprescindible declarar la función en la forma `[a]=circulo(r)` y hemos utilizado, simplemente, `a=circulo(r)`.

Ejemplo 2.2 *Créese un fichero de función, de nombre rectangulo.m, que calcule el área y el perímetro de un rectángulo, a partir de su base y de su altura; seguidamente, utilícese para hallar área y perímetro de un rectángulo de base* 2 *y altura* 3*.*

Solución: Construimos el fichero rectangulo.m, con las órdenes

```
function [area perimetro]=rectangulo(base,altura)
% Función [area perimetro]=rectangulo(base,altura) para
% hallar el área y el perímetro de un rectángulo, en este orden
area=base*altura;
perimetro=2*base+2*altura;
```

y hacemos:

```
>> [a p]=rectangulo(2,3)
a =
     6
p =
    10
```

Nota: Obsérvese que con

```
>> x=rectangulo(2,3)
x =
     6
```

o bien

```
>> rectangulo(2,3)
ans =
     6
```

sólo obtenemos la primera variable de salida, es decir, el área. En general, si una función tiene varias variables de salida y, al aplicarla, sólo pedimos que calcule un valor de salida, obtendremos como resultado la primera variable de salida; si pedimos que calcule dos valores de salida, obtendremos como resultado las dos primeras variables de salida (en ese orden) y así sucesivamente.

Supongamos ahora que queremos prevenir que, por error, introduzcamos un valor negativo en alguno de los argumentos de entrada. Entonces, podemos usar una estructura de la forma

```
if condición
    órdenes_1
else órdenes_2
end
```

Esta estructura hace que, si `condición` es cierta, se ejecutan `órdenes_1` y, en el resto de casos, se ejecutan `órdenes_2`.

Teniendo esto en cuenta, creamos el fichero rectangulo2.m con las órdenes

```
function [area perimetro]=rectangulo2(base,altura)
% Función [area perimetro]=rectangulo2(base,altura) para
% hallar el área y el perímetro de un rectángulo, en este orden
if base<=0 || altura<=0
    disp('Los argumentos de entrada deben ser positivos')
    area=[]; perimetro=[];
else
    area=base*altura;
    perimetro=2*base+2*altura;
end
```

donde se ha usado el operador `||`, que es la forma de expresar la "o"disyuntiva y el comando `disp('texto')` para que `texto` se muestre por pantalla. A continuación, vemos ejemplos de uso del programa:

```
>> [a p]=rectangulo2(2,-3)
Los argumentos de entrada deben ser positivos
a =
     []
p =
     []
>> [a p]=rectangulo2(-2,3)
Los argumentos de entrada deben ser positivos
a =
     []
p =
     []
>> [a p]=rectangulo2(2,3)
a =
     6
p =
    10
```

En el ejemplo anterior, hemos utilizado una estructura condicional que admitía únicamente dos posibilidades: `condicion` sólo puede ser cierta o falsa. En el caso de que haya más de dos opciones, podemos utilizar una estructura del tipo

```
if condición_1
    órdenes_1
elseif condición_2
    órdenes_2
..................
..................
elseif condición_n
    órdenes_n
else órdenes_n+1
end
```

que hace que el programa estudie si es cierta `condición_1`, en cuyo caso ejecuta `órdenes_1`; si es falsa, estudia `condición_2` y, si ésta es cierta, ejecuta `órdenes_2`; si también es falsa, seguiría estudiando las condiciones establecidas hasta llegar a `condición_n`; si es cierta, ejecutaría `órdenes_n` y, en caso contrario, ejecutaría `órdenes_n+1`.

Ejemplo 2.3 *El precio base de una determinada modalidad de seguro, para cierto modelo de coche, es de* 300 *euros y la compañía, para calcular el precio final, tiene en cuenta lo siguiente:*

Si el conductor tiene menos de 26 *años, el precio se incrementa un* 25 *%; si tiene entre* 26 *y* 30 *años se incrementa un* 10 *%; si tiene entre* 31 *y* 65 *años el precio no se modifica; si tiene más de* 65 *años el precio se incrementa un* 15 *%.*

Además, si el conductor tiene permiso de conducir con menos de 2 *años de antigüedad, el precio resultante se incrementa un* 25 *%.*

Con estos datos, constrúyase la función `seguro(e,a)` *que calcule el precio final del seguro, en función de la edad* `e` *y de la antigüedad* `a` *del permiso de conducir.*

Solución: Creamos el fichero seguro.m con las órdenes

```
function x=seguro(e,a)
% Funcion x=seguro(e,a) que calcula el precio final del seguro
% ARGUMENTOS DE ENTRADA:
%     e ..... Edad del conductor
%     a ..... Antigüedad del permiso de conducir
% ARGUMENTO DE SALIDA:
%     x ..... Precio final del seguro
e=floor(e); base=300;
if e<26
    x=base*1.25;
elseif e<=30
    x=base*1.10;
elseif e<=65
    x=base;
else x=base*1.15;
end
if a<2
    x=x*1.25;
end
if a<0
    disp('La antigüedad del permiso no puede ser un número negativo')
    x=[];
end
if e<18
    disp('La edad no puede ser inferior a 18 años')
    x=[];
end
```

donde hemos utilizado la orden `floor(e)` para redondear `e` al entero inferior.

A continuación, vemos algunos ejemplos de uso de la función anterior:

```
>> precio=seguro(25,5)
precio =
   375
>> precio=seguro(25,1)
precio =
  468.7500
>> precio=seguro(35,10)
precio =
   300
>> precio=seguro(17,-1)
La antigüedad del permiso debe ser un número positivo
La edad no puede ser inferior a 18 años
precio =
     []
```

Ejemplo 2.4 *Constrúyase un fichero de función, de nombre factoriales.m, que calcule factoriales, estudiando previamente el dato de entrada, de forma que, si no es un número natural, se obtenga un mensaje de advertencia.*

Solución: Utilizaremos la estructura

```
for k=vector
órdenes
end
```

que hace que, si `vector` tiene unas coordenadas `x_1`, `x_2`, ... , `x_n`, entonces `órdenes` se ejecutan, en primer lugar, para `k=x_1`, a continuación, para `k=x_2` y así sucesivamente.

De esta manera, podríamos crear factoriales.m con las órdenes

```
function f=factoriales(n)
% Función f=factoriales(n) para calcular el factorial de n
if n==round(n) && n>=0
    f=1;
    for k=2:n
        f=f*k;
    end
else disp('ATENCIÓN: El dato de entrada debe ser un número natural')
    f=[];
end
```

Aquí, hemos utilizado la orden `round(n)` que redondea `n` al entero más próximo y, como `n` y `round(n)` deben coincidir, hemos pedido que MATLAB los compare, utilizando un doble `=`, para distinguirlo del `=` simple, que se utiliza para asignar un valor a una variable.

Podemos comprobar el funcionamiento del fichero con:

```
>> factoriales(-2)
ATENCIÓN: El dato de entrada debe ser un número natural
>> factoriales(pi)
ATENCIÓN: El dato de entrada debe ser un número natural
>> factoriales(5)
ans =
   120
```

Ejemplo 2.5 *Dada una sucesión a_n, créese el fichero suma_parcial.m, de forma que calcule $S_k = a_1 + a_2 + \cdots + a_k$; los datos de entrada deben ser una función numérica a, dependiente de n, que represente el término enésimo de la sucesión, y que ha de ser definida previamente, y el número de sumandos k de la suma parcial.*

Solución: Escribimos, en el fichero suma_parcial.m, las órdenes

```
function s=suma_parcial(a,k)
% Función s=suma_parcial(a,k) para hallar a_1+a_2+ ... +a_k
% ARGUMENTOS DE ENTRADA:
%     a ..... Término enésimo definido previamente, como función
%             numérica, en función de n
%     k ..... Número de sumandos de la suma parcial
% ARGUMENTO DE SALIDA:
%     s ..... Valor de la suma parcial
s=0;
for n=1:k
    s=s+a(n);
end
```

y vamos a comprobar su correcto funcionamiento, hallando sumas parciales de la serie:

$$\sum_{n=1}^{\infty} \frac{4 \cdot (-1)^{n-1}}{2n-1} = \pi$$

Para ello, podemos hacer:

```
>> x=@(n) 4*(-1)^(n-1)/(2*n-1)
x =
    @(n)4*(-1)^(n-1)/(2*n-1)
>> suma=suma_parcial(x,300)
suma =
    3.1383
>> suma=suma_parcial(x,3000)
suma =
    3.1413
>> suma=suma_parcial(x,30000)
suma =
    3.1416
```

Nota: Cuando una función tiene como argumento de entrada otra función, como en este caso, en el que la función `x` era argumento de entrada de la función `suma_parcial`, MATLAB espera que la función `x` esté ya memorizada como función anónima. En el caso de que la función esté definida a través de un fichero, hay que hacer un pequeño retoque en la orden `suma_parcial(función,k)` que deberá pasar a ser `suma_parcial(@función,k)`.

Como ejemplo, volvemos a la serie anterior, pero definimos su término enésimo en el fichero a.m, con las órdenes

```
function y=a(n)
y=4*(-1)^(n-1)/(2*n-1);
```

y vemos que se obtiene un error, al ejecutar

```
>> suma=suma_parcial(a,30000)
??? Input argument "n" is undefined.
Error in ==> a at 2
y=4*(-1)^(n-1)/(2*n-1);
```

sin embargo, con

```
>> suma=suma_parcial(@a,30000)
ans =
    3.1416
```

obtenemos el mismo resultado que cuando aplicábamos `suma=suma_parcial(x,30000)`.

Para hallar de forma aproximada $\lim_{n\to\infty} x_n$, un procedimiento muy habitual consiste en ir calculando elementos de la sucesión y utilizar como test de parada la condición

$$\frac{|x_{n+1} - x_n|}{|x_n|} \leq T$$

siendo T una medida del error relativo con el que se desea trabajar. Eso sí, para prevenir posibles divisiones por cero, dicha condición se suele plantear como:

$$|x_{n+1} - x_n| \leq T \cdot |x_n|$$

Ejemplo 2.6 *Constrúyase el fichero limite.m, de forma que calcule* $\lim_{n\to\infty} x_n$.

Los datos de entrada deben ser una función numérica x, dependiente de n, que represente el término enésimo de la sucesión, y que ha de ser definida previamente, y la tolerancia relativa T con la que se desea trabajar.

Solución: Para resolver este problema, usaremos la estructura

```
while condición
    órdenes
end
```

que hace que se ejecuten las `órdenes` que se hayan establecido, mientras `condición` sea cierta.

Así, podemos crear el fichero limite.m, con las órdenes

```
function [l n]=limite(x,T)
% Función [l n]=limite(x,T) para hallar el límite de x_1,x_2,x_3,...
% ARGUMENTOS DE ENTRADA:
%     x ..... Término enésimo definido previamente, como función
%             numérica, en función de n
%     T ..... Tolerancia relativa permitida
% ARGUMENTOS DE SALIDA:
%     l ..... Valor aproximado del límite
%     n ..... Número de orden del elemento que aproxima el límite
n=1; l=x(1);
test_parada=0; % n es el contador de elementos
while test_parada==0
    n=n+1;
    test_parada=abs(x(n)-l) <= T*abs(l);
    l=x(n);
end
```

y comprobar su funcionamiento, haciendo:

```
>> x=@(n) (n^2+1)/(2*n^2+3*n+5)
x =
    @(n)(n^2+1)/(2*n^2+3*n+5)
>> [lim n]=limite(x,1e-9)
lim =
    0.5000
n =
       38731
```

Para finalizar este apartado, indicar que para prevenir que no haya convergencia, o que ésta sea muy lenta, se suele añadir una *condición de salvaguarda* que impida que el programa haga un número excesivo de cálculos o los haga de forma indefinida. En el ejemplo anterior, podemos indicar que no se calculen más de un número máximo de elementos, construyendo el fichero limite2.m, a partir del fichero limite.n, de la siguiente forma

```
function [l n]=limite2(x,T,N)
% Función [l n]=limite2(x,T) para hallar el límite de x_1,x_2,x_3,...
% ARGUMENTOS DE ENTRADA:
%     x ..... Término enésimo definido previamente, como función
%             numérica, en función de n
%     T ..... Tolerancia relativa permitida
%     N ..... Número máximo de elementos que se calcularán
```

```
% ARGUMENTOS DE SALIDA:
%     l ..... Valor aproximado del límite
%     n ..... Número de orden del elemento que aproxima el límite
n=1; l=x(1);
test_parada=0; % n es el contador de elementos
while test_parada==0 && n<N
    n=n+1;
    test_parada=abs(x(n)-l) <= T*abs(l);
    l=x(n);
end
if n == N
    disp('No converge con la precisión pedida.')
    disp('El resultado no es el límite: es el último elemento hallado.')
end
```

donde hemos utilizado el operador `&&`, que equivale a la conjunción "y".

Así, si intentamos hallar el límite de una sucesión no convergente, como $\frac{n^2}{n+1}$:

```
>> x=@(n) n^2/(n+1)
x =
    @(n)n^2/(n+1)
>> limite2(x,1e-6,20000)
No converge con la precisión pedida.
El resultado no es el límite: es el último elemento hallado.
ans =
  1.9999e+004
```

2.2. Precisión de los cálculos

En MATLAB, la precisión con la que se representan los números reales internamente es siempre la misma y está entre 15 y 16 dígitos. Esta precisión se puede ver con la orden `eps(x)`, que representa la distancia entre `x` y el siguiente número del ordenador. Por ejemplo, si ejecutamos

```
>> eps(10)
ans =
  1.7764e-015
```

esto indica que, para el programa, el siguiente número a `10` es `10+1.7764e-015`, por lo que, si hacemos,

```
>> 10+10^(-16)
ans =
     10
```

vemos que se obtiene un resultado erróneo.

Con las siguientes órdenes, podemos ver la evolución de `eps(x)` y la pérdida de precisión que resulta, a medida que crece `x`:

```
>> eps(0)
ans =
  4.9407e-324
>> eps(1)
ans =
  2.2204e-016
>> eps % obsérvese que equivale a eps(1)
ans =
  2.2204e-016
>> eps(1000)
ans =
  1.1369e-013
>> eps(10^16)
ans =
     2
```

Otro problema que se produce al hacer operaciones con números grandes, cosa que se debe evitar, siempre que sea posible, es que hay un valor concreto que es el número más grande que puede manejar MATLAB. Dicho valor se puede obtener ejecutando

```
>> realmax
ans =
  1.7977e+308
```

y esta limitación hace que, al hacer cálculos con números grandes, puedan producirse errores o que propiedades matemáticas, de sobra conocidas, no sean válidas:

```
>> 10^308*2*0.1
ans =
   Inf
>> 10^308*(2*0.1)
ans =
  2.0000e+307
```

Para finalizar, indicar que, por defecto, MATLAB muestra 5 cifras significativas, pero esto se puede cambiar con la orden **format** (ejecutando `help format`, se obtiene información de las opciones del comando **format**), como se puede observar, haciendo

```
>> format long
>> pi
ans =
    3.141592653589793
```

y para volver al formato por defecto:

```
>> format
```

Con independencia del formato que utilicemos, siempre podemos hacer que se muestren el número de decimales que deseemos, utilizando la orden **fprintf**.

Un primer uso de dicha orden es presentar texto (que irá siempre entrecomillado, con la comilla simple ') como se puede ver en el siguiente ejemplo

```
>> fprintf('Texto de ejemplo\n')
Texto de ejemplo
```

donde se ha utilizado `\n` para forzar un cambio de línea.

Otra forma de utilizar este comando es para presentar texto y valores de variables: MATLAB, al encontrar una orden del tipo `fprintf('... % ... % ... % ...',x1,x2,x3, ...)`, sustituye el primer símbolo de `%` por el valor de la variable `x1`, el segundo por el valor de la variable `x2` y así sucesivamente, presentando dichos valores con el formato que se indique a continuación de los símbolos de `%`.

Como ejemplo, vemos una forma de presentar la longitud de una circunferencia de radio $\sqrt{2}$:

```
>> r=sqrt(2); l=2*pi*r
l =
    8.8858
>> fprintf('Una circunferencia de radio %.2f tiene longitud %.6f.\n',r,l)
Una circunferencia de radio 1.41 tiene longitud 8.885766.
```

Nótese que con `%.2f` y `%.6f` conseguimos que las variables `r` y `l` sean presentadas con 2 y 6 decimales, respectivamente.

3. Práctica 3

En esta práctica, se inicia el estudio de las ecuaciones de una variable, con dos métodos de resolución: el método de bisección y el del punto fijo.

3.1. Método de bisección

Dada una función real $f(x)$, de variable $x \in \mathbb{R}$, nos planteamos el problema de calcular sus ceros o raíces, es decir, resolver $f(x) = 0$.

El método de bisección, basado en el teorema de Bolzano, consiste en buscar dos valores, a y b, en los que la función f cambie de signo. Entonces, la función tiene que anularse en algún punto entre a y b, con lo que una forma de resolver el problema sería denotar x al punto medio del intervalo $[a, b]$ y estudiar si $f(x) = 0$. En caso de serlo, habríamos resuelto el problema y, en caso contrario, habrá que estudiar si $f(a)f(x) < 0$ o si, por el contrario es $f(x)f(b) < 0$. Si $f(a)f(x) < 0$, la solución estaría en el intervalo $[a, x]$, mientras que, si fuera $f(x)f(b) < 0$, estaría en el intervalo $[x, b]$.

De esta forma, la longitud del nuevo intervalo al que pertenece la solución es la mitad del intervalo original, con lo que, repitiendo el procedimiento el número suficiente de veces, podemos llegar a localizar la solución en un intervalo de longitud tan pequeña como se quiera y, en definitiva, aproximar la solución con la precisión que se desee.

Debemos tener en cuenta que, si se sabe que la solución está en $[a, b]$, al aproximarla por el punto medio del intervalo, el error máximo cometido es

$$\frac{b-a}{2}$$

con lo cual, si el método se aplica n veces, la cota del error sería

$$\frac{b-a}{2^n}$$

por lo que una forma de garantizar que el error máximo fuese una cantidad E prefijada, sería hallar n de forma que

$$\frac{b-a}{2^n} = E$$

y tras redondearlo al entero superior, si el resultado no es exacto, aplicar el método de bisección n veces.

Así, una forma de programar todo lo anterior, como fichero de función, es crear el fichero biseccion.m con las órdenes

```
function x=biseccion(f,a,b,E)
%     Función x=biseccion(f,a,b,E)
%     Método de Bisección para resolver f(x) = 0
% ARGUMENTOS DE ENTRADA:
%   f .....  Función numérica de la que se buscan los ceros
%   a,b ...  Extremos del intervalo de búsqueda de la solución
%   E .....  Cota del error absoluto: |x-r| <= E, siendo r la raíz exacta
% ARGUMENTO DE SALIDA:
%   x  ....  Valor aproximado de la raíz
fa=f(a); fb=f(b);
if fa*fb<0
    N=ceil(log2(abs(b-a)/E)); % Nº de iteraciones para que |x-r| <= E
    N=max(N,1); % Se hace al menos una iteración
    for n=1:N
        x=(a+b)/2; fx=f(x);
        if fx==0
            disp('Solución exacta'), return
        elseif fa*fx<0
            b=x;
        else
            a=x;
        end
    end
else  % Casos especiales
    if fa==0
        disp('Solución exacta'), x=a; return
    elseif fb==0
        disp('Solución exacta'), x=b; return
    else
        disp('MÉTODO NO APLICABLE:')
        disp('La función no cambia de signo en los extremos')
        x=[];
    end
end
```

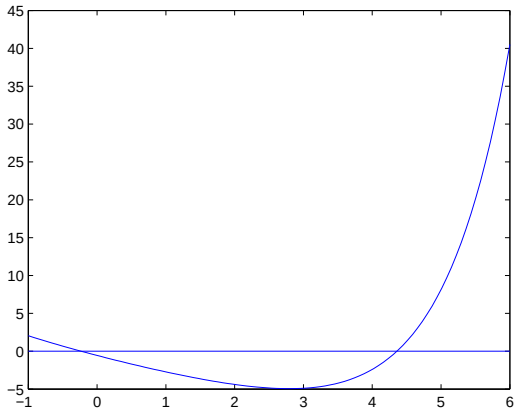
donde se han utilizado los comandos **ceil** (`ceil(x)` redondea `x` al entero superior, si `x` no es entero) y **return**, que hace que los cálculos se interrumpan en el punto en el que aparece dicho comando.

Ejemplo 3.1 *Obténganse, en el intervalo* $[-1,6]$*, con un error máximo de* 10^{-12}*, los ceros de la función* $f(x) = e^{x-2} - \ln(x+2) - 2x$.

Solución: En primer lugar, definimos la función y la representamos gráficamente, ejecutando las órdenes

```
>> f=@(x) exp(x-2)-log(x+2)-2*x
f =
    @(x)exp(x-2)-log(x+2)-2*x
>> xx=linspace(-1,6);
```

```
>> plot(xx,f(xx))
>> hold on, plot([-1 6],[0 0])
```



y vemos que la función tiene una raíz entre -1 y 0 y otra entre 4 y 5, por lo que, para calcularlas con la precisión requerida, basta hacer:

```
>> x1=biseccion(f,-1,0,1e-12)
x1 =
   -0.2314
>> x2=biseccion(f,4,5,1e-12)
x2 =
    4.3575
```

3.2. Método del punto fijo

Este método se utiliza para resolver ecuaciones de la forma $f(x) = x$, es decir, para calcular un *punto fijo de* f que, gráficamente, sería hallar la intersección de la curva $y = f(x)$ con la bisectriz del primer cuadrante $y = x$.

La idea es, partiendo de un punto x_0, construir la sucesión $x_{n+1} = f(x_n),\ n \in \mathbb{N}$; si se verifican las hipótesis del teorema del punto fijo, dicha sucesión converge a la solución buscada, independientemente de la elección que se haga del punto de partida x_0.

Así, construiremos la sucesión y utilizaremos como test de parada la condición

$$\frac{|f(x_n) - x_n|}{|x_n|} = \frac{|x_{n+1} - x_n|}{|x_n|} \leq T$$

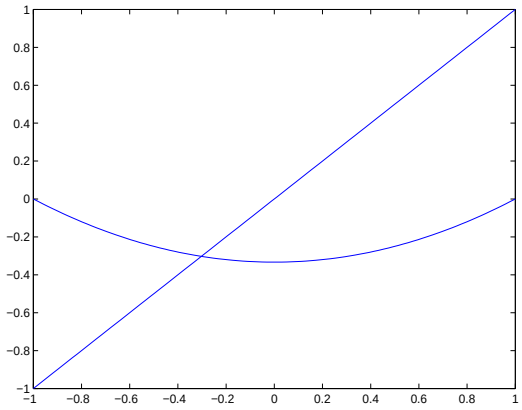
siendo T una medida del error relativo con el que se desea trabajar. Para prevenir posibles divisiones por números próximos a cero, la plantearemos como $|x_{n+1} - x_n| \leq T \cdot |x_n|$ y, de este modo, una forma de programar este método, como fichero de función, sería crear el fichero ptofijo.m con las órdenes

```
function [x n]=ptofijo(f,x0,T,N)
%    Función [x n]=ptofijo(f,x0,T,N)
% Método de punto fijo para resolver f(x) = x
% ARGUMENTOS DE ENTRADA:
%   f .......  Función numérica
%   x0 ......  Punto de partida de las iteraciones
%   T .......  Tolerancia relativa permitida
%   N .......  Número máximo de Iteraciones
% ARGUMENTOS DE SALIDA:
%   x .......  Solución
%   n .......  Número de iteraciones efectuadas
test_parada=0; n=0; % n es el contador de iteraciones
while test_parada==0 && n<N
    n=n+1;
    x=f(x0);
    test_parada=abs(x-x0)<=T*abs(x0);
    x0=x;
end
if n==N
    disp('No converge con la precisión pedida.')
    disp('El valor hallado no es la solución, sino la última iteración:')
end
```

Ejemplo 3.2 *Compruébese gráficamente que la función* $f(x) = \dfrac{x^2-1}{3}$ *posee un punto fijo en* $[-1,1]$ *y calcúlese el mismo, partiendo de* $x_0 = 0$*, con tolerancia relativa de* 10^{-6}*.*

Solución: Definimos la función y representamos la curva $y = f(x)$, junto con la recta $y = x$, haciendo

```
>> f=@(x) (x.^2-1)/3
f =
    @(x)(x.^2-1)/3
>> xx=linspace(-1,1);
>> plot(xx,f(xx))
>> hold on, plot(xx,xx)
```



y se observa que hay un único punto de corte, que es el punto fijo, que podemos hallar, ejecutando:

```
>> x=ptofijo(f,0,1e-6,100)
x =
   -0.3028
```

Ejemplo 3.3 *Constrúyase la función* `ptofijo2`*, modificando la función* `ptofijo`*, de manera que la variable de salida contenga los valores de las iteraciones del método del punto fijo. A continuación, aplíquense* `ptofijo` *y* `ptofijo2`*, con* $x_0 = 1$*, tolerancia relativa* 10^{-6} *y* 100 *como máximo de iteraciones, a la función:*

$$g(x) = \sqrt{\frac{x + 3 - x^4}{2}}$$

Solución: Creamos ptofijo2.m con las órdenes

```
function x=ptofijo2(f,x0,T,N)
%    Función x=ptofijo2(f,x0,T,N)
% Método de punto fijo para resolver f(x) = x
% ARGUMENTOS DE ENTRADA:
%   f .......  Función numérica
%   x0 ......  Punto de partida de las iteraciones
%   T .......  Tolerancia relativa permitida
%   N .......  Número máximo de iteraciones
% ARGUMENTO DE SALIDA:
%   x  ......  Vector cuya última coordenada es la solución
%              y las demás, las iteraciones anteriores
test_parada=0; n=0; % n es el contador de iteraciones
x(1)=x0;
while test_parada==0 && n<N
     n=n+1;
     x(n+1)=f(x(n));
    test_parada=abs(x(n+1)-x(n))<=T*abs(x(n));
end
x(1)=[]; % Eliminamos el punto x0 de la sucesión
if n==N
    disp('No converge con la precisión pedida.')
end
```

y comprobamos su funcionamiento con la función del ejemplo anterior:

```
>> x=ptofijo2(f,0,1e-6,100)
x =
  Columns 1 through 7
   -0.3333   -0.2963   -0.3041   -0.3025   -0.3028   -0.3028   -0.3028
  Columns 8 through 10
   -0.3028   -0.3028   -0.3028
```

A continuación, construimos la función $g(x)$ con

```
>> g=@(x) sqrt((x+3-x^4)/2)
g =
    @(x)sqrt((x+3-x^4)/2)
```

y, al aplicar,

```
>> x=ptofijo(g,1,1e-6,100)
No converge con la precisión pedida.
El valor hallado no es la solución, sino la última iteración:
x =
    0.9306
```

vemos que con 100 iteraciones no se encuentra el punto fijo, por lo que podríamos plantearnos si el motivo es que hemos hecho pocas iteraciones. Sin embargo, al aplicar

```
>> x=ptofijo2(g,1,1e-6,100);
No converge con la precisión pedida.
```

si observamos los últimos elementos de la sucesión, haciendo

```
>> x(95:end)
ans =
    1.2611    0.9306    1.2611    0.9306    1.2611    0.9306
```

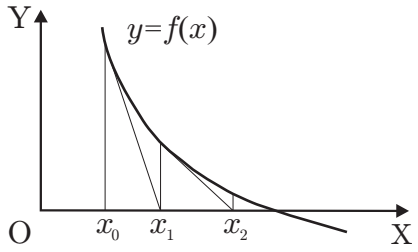
es claro que no merece la pena aumentar las iteraciones, ya que la no convergencia se debe a que la subsucesión de elementos pares tiene distinto límite que la de elementos impares.

4. Práctica 4

Siguiendo con la resolución de ecuaciones de una variable, en esta práctica estudiaremos los métodos de Newton-Raphson y de la secante, así como la utilización de la orden **fzero**.

4.1. Método de Newton-Raphson

Este método resuelve la ecuación $f(x) = 0$, como el límite de una sucesión construida de la siguiente forma: partiendo de un punto x_0, se traza la tangente a la curva $y = f(x)$, en el punto $(x_0, f(x_0))$, y se corta con el eje OX, resultando un punto x_1. A continuación, se repite lo anterior con x_1 para obtener x_2 y así sucesivamente. En el siguiente gráfico, se puede observar lo anteriormente expuesto.



Así, si calculamos la recta tangente en $(x_0, f(x_0))$, por medio de la fórmula $y - y_0 = m(x - x_0)$,

$$y - f(x_0) = f'(x_0)(x - x_0)$$

al cortar con la recta $y = 0$,

$$-f(x_0) = f'(x_0)(x - x_0) \implies -\frac{f(x_0)}{f'(x_0)} = x - x_0 \implies x_0 - \frac{f(x_0)}{f'(x_0)} = x$$

con lo que resulta,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

y en general:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Utilizaremos como test de parada la condición

$$\frac{|x_{n+1} - x_n|}{|x_n|} \leq T$$

que plantearemos como $|x_{n+1} - x_n| \leq T \cdot |x_n|$, siendo T una medida del error relativo con el que se desea trabajar.

Así, una forma de programar el anterior algoritmo es crear el fichero de función newton.m con las órdenes:

```
function [x n]=newton(f,f_der,x0,T,N)
%     Función [x n]=newton(f,f_der,x0,T,N)
% Método de Newton-Raphson para resolver f(x) = 0
% ARGUMENTOS DE ENTRADA:
%   f ..........  Función numérica
%   f_der ......  Función derivada de f en formato numérico
%   x0 .........  Punto inicial del algoritmo
%   T ..........  Tolerancia relativa permitida
%   N ..........  Número máximo de iteraciones
% ARGUMENTOS DE SALIDA:
%   x ..........  Solución
%   n ..........  Número de iteraciones efectuadas
test_parada=0; n=0; % n es el contador de iteraciones
while test_parada==0 && n<N
    n=n+1;
    x=x0-f(x0)/f_der(x0);
    test_parada=abs(x-x0)<=T*abs(x0);
    x0=x;
end
if n==N
    disp('No converge con la precisión pedida.')
    disp('El valor hallado no es la solución, sino la última iteración:')
end
```

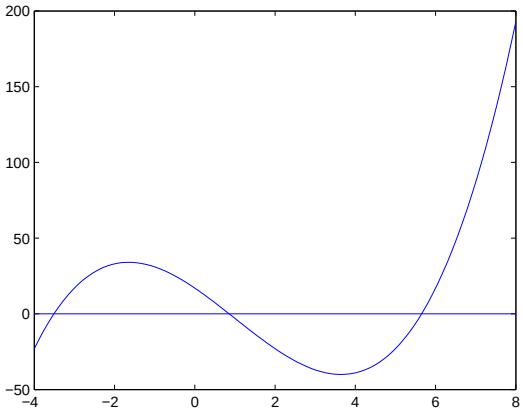
Ejemplo 4.1 *Compruébese gráficamente que la ecuación* $x^3 - 3x^2 - 18x + 17 = 0$ *tiene sus tres soluciones en el intervalo* $[-4, 8]$ *y calcúlense las mismas, aplicando el método de Newton-Raphson, con tolerancia* 10^{-6}.

Solución: Definimos $f(x) = x^3 - 3x^2 - 18x + 17$ de forma simbólica, para que MATLAB pueda calcular su derivada y hallamos ésta, haciendo

```
>> syms x, f=x^3-3*x^2-18*x+17
f =
x^3 - 3*x^2 - 18*x + 17
>> f_der=diff(f)
f_der =
3*x^2 - 6*x - 18
```

y, a continuación, las pasamos a formato numérico y representamos $y = f(x)$:

```
>> f_num=matlabFunction(f)
f_num =
    @(x)x.*-1.8e1-x.^2.*3.0+x.^3+1.7e1
>> f_der_num=matlabFunction(f_der)
f_der_num =
    @(x)x.*-6.0+x.^2.*3.0-1.8e1
>> xx=linspace(-4,8); plot(xx,f_num(xx))
>> hold on, plot([-4 8],[0 0])
```



Por último, hallamos las soluciones:

```
>> x1=newton(f_num,f_der_num,-3,1e-6,200)
x1 =
   -3.5094
>> x2=newton(f_num,f_der_num,2,1e-6,200)
x2 =
    0.8570
>> x3=newton(f_num,f_der_num,6,1e-6,200)
x3 =
    5.6524
```

Ejemplo 4.2 *Constrúyase la función* `newton2`*, modificando la función* `newton`*, de manera que la variable de salida contenga los valores de las iteraciones del método de Newton-Raphson. A continuación, aplíquese* `newton2`*, a la resolución de la ecuación del ejemplo anterior.*

Solución: Construimos el fichero newton2.m con las instrucciones

```
function x=newton2(f,f_der,x0,T,N)
%     Función x=newton2(f,f_der,x0,T,N)
% Método de Newton-Raphson para resolver f(x) = 0
% ARGUMENTOS DE ENTRADA:
%   f ..........  Función numérica
%   f_der ......  Función derivada de f en formato numérico
%   x0 .........  Punto inicial del algoritmo
%   T ..........  Tolerancia relativa permitida
%   N ..........  Número máximo de iteraciones
% ARGUMENTO DE SALIDA:
%   x  ......     Vector cuya última coordenada es la solución
%                 y las demás, las iteraciones anteriores
test_parada=0; n=0; % n es el contador de iteraciones
x(1)=x0;
```

```
    while test_parada==0 && n<N
        n=n+1;
        x(n+1)=x(n)-f(x(n))/f_der(x(n));
        test_parada=abs(x(n+1)-x(n))<=T*abs(x(n));
    end
    x(1)=[]; % Eliminamos el punto x0 de la sucesión
    if n==N
        disp('No converge con la precisión pedida.')
    end
```

y hallamos las soluciones del ejemplo anterior:

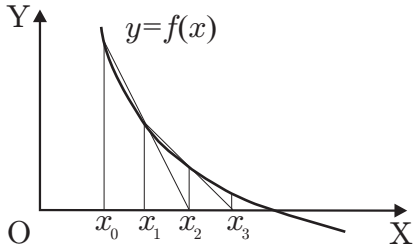
```
>> y1=newton2(f_num,f_der_num,-3,1e-6,200)
y1 =
   -3.6296   -3.5140   -3.5094   -3.5094   -3.5094
>> y2=newton2(f_num,f_der_num,2,1e-6,200)
y2 =
    0.7222    0.8576    0.8570    0.8570

>> y3=newton2(f_num,f_der_num,6,1e-6,200)
y3 =
    5.6852    5.6527    5.6524    5.6524
```

4.2. Método de la secante

Este método es una variación del de Newton, en el que la recta tangente se sustituye por la secante a la curva que pasa por dos puntos de la misma, con la ventaja de que, a diferencia del método anterior, no requiere el cálculo de la derivada.

Así, resolveremos $f(x) = 0$, hallando el límite de una sucesión construida de la siguiente manera: partiendo de dos puntos x_0, x_1, trazamos la recta que pasa por los puntos $(x_0, f(x_0))$ y $(x_1, f(x_1))$, y se corta con el eje OX, resultando un punto x_2. A continuación, se repite lo anterior con x_1 y x_2 para obtener x_3 y así sucesivamente. En el siguiente gráfico, se resume lo anterior:



Teniendo en cuenta que la recta que pasa por los puntos (x_0, y_0), (x_1, y_1) es

$$\frac{y - y_1}{x - x_1} = \frac{y_1 - y_0}{x_1 - x_0}$$

la recta que une $(x_0, f(x_0))$ con $(x_1, f(x_1))$ tiene de ecuación

$$\frac{y - f(x_1)}{x - x_1} = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

con lo que, si hallamos su intersección con $y = 0$, resulta $-\dfrac{f(x_1)}{x - x_1} = \dfrac{f(x_1) - f(x_0)}{x_1 - x_0} \implies$

$$\implies -f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} = x - x_1 \implies x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} = x$$

con lo cual,

$$x_2 = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

y en general:

$$x_{n+2} = x_{n+1} - f(x_{n+1})\frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)}$$

Dejaremos de hacer iteraciones, cuando $|x_{n+2} - x_{n+1}| \leq T \cdot |x_{n+1}|$, donde T es una medida del error relativo deseado, con lo que el método se puede programar, como fichero de función, de la siguiente forma:

```
function [x n]=secante(f,x0,x1,T,N)
%    Función [x n]=secante(f,x0,x1,T,N)
%    Método de la Secante para resolver la ecuación f(x) = 0
% ARGUMENTOS DE ENTRADA:
%   f ..........  Función numérica
%   x0,x1 ......  Puntos iniciales del algoritmo
%   T ..........  Tolerancia relativa permitida
%   N ..........  Número máximo de iteraciones
% ARGUMENTOS DE SALIDA:
%   x ..........  Solución
%   n ..........  Número de iteraciones efectuadas
y0=f(x0); y1=f(x1); test_parada=0; n=0;
% n es el contador de iteraciones
while test_parada==0 && n<N
    n=n+1;
    x=x1-y1*(x1-x0)/(y1-y0);
    test_parada=abs(x-x1)<=T*abs(x1);
    x0=x1; y0=y1; x1=x; y1=f(x1);
end
if n==N
    disp('No converge con la precisión pedida.')
    disp('El valor hallado no es la solución, sino la última iteración:')
end
```

Ejemplo 4.3 *Resuélvase nuevamente la ecuación $x^3 - 3x^2 - 18x + 17 = 0$, con la misma tolerancia de los ejemplos anteriores, aplicando el método de la secante.*

Solución: Teniendo en cuenta la gráfica que habíamos obtenido, podemos hacer:

```
>> z1=secante(f_num,-4,-2,1e-6,200)
z1 =
   -3.5094
>> z2=secante(f_num,0,2,1e-6,200)
z2 =
    0.8570
>> z3=secante(f_num,4,6,1e-6,200)
z3 =
    5.6524
```

4.3. La orden fzero

Esta orden combina los métodos de bisección y de la secante, para resolver la ecuación $f(x) = 0$. Para buscar una solución cercana a `x0`, se ejecuta `fzero(f,x0)`, si la función está definida como anónima, o bien `fzero(@f,x0)`, si la función está definida en un fichero.

Si en un intervalo `[a,b]` la función cambia de signo, se puede ejecutar `fzero(f,[a,b])` o `fzero(@f,[a,b])`, dependiendo, al igual que antes, de que la función se haya construido de forma anónima o en un fichero.

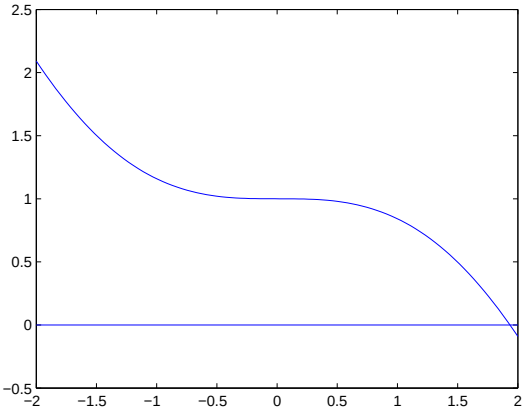
Ejemplo 4.4 *Resuélvase la ecuación* $\operatorname{sen} x - x + 1 = 0$*, utilizando* **fzero**.

Solución: Para que se verifique la ecuación, debe ser

$$x = \operatorname{sen} x + 1 \implies |x| \leq |\operatorname{sen} x| + 1 \leq 2$$

con lo que, si hay soluciones, deben pertenecer al intervalo $[-2, 2]$. Por ello, hacemos

```
>> f=@(x) sin(x)-x+1
f =
    @(x)sin(x)-x+1
>> xx=linspace(-2,2); plot(xx,f(xx))
>> hold on, plot([-2 2],[0 0])
```



y vemos que existe una solución cercana a 2, que podemos hallar ejecutando

```
>> fzero(f,2)
ans =
    1.9346
```

o también se podría resolver por medio de:

```
>> fzero(f,[1 2])
ans =
    1.9346
```

5. Práctica 5

En esta práctica, iniciamos la resolución de sistemas de ecuaciones con los métodos directos de resolución de sistemas lineales, que plantearemos de la forma $A\,\mathbf{x} = \mathbf{b}$, donde A es la matriz cuadrada de coeficientes del sistema, $\mathbf{x}$ el vector columna de las incógnitas, $\mathbf{b}$ el vector columna de los términos independientes y supondremos que el sistema tiene una única solución.

5.1. Eliminación gaussiana

La eliminación gaussiana consiste, mediante operaciones elementales en las matrices A y $\mathbf{b}$, en transformar el sistema en otro equivalente con matriz de coeficientes triangular. Por ello, empezaremos por programar dos algoritmos de resolución para sistemas triangulares, aunque, antes, vamos a ver el funcionamiento de las órdenes **diag**, **triu** y **tril**, que utilizaremos más adelante, por medio de algunos ejemplos:

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

```
>> diag(A)
ans =
     1
     2
     3
     4
>> diag(A,-1)
ans =
     1
     3
     5
>> diag(A,2)
ans =
     1
     3
>> triu(A)
ans =
     1     2     1     2
     0     2     4     3
     0     0     3     6
     0     0     0     4
>> tril(A)
ans =
     1     0     0     0
     1     2     0     0
     2     3     3     0
     3     4     5     4

>> triu(A,-1)
ans =
     1     2     1     2
     1     2     4     3
     0     3     3     6
     0     0     5     4
>> triu(A,1)
ans =
     0     2     1     2
     0     0     4     3
     0     0     0     6
     0     0     0     0
```

```
>> tril(A,2)
ans =
     1     2     1     0
     1     2     4     3
     2     3     3     6
     3     4     5     4
```

Sistema triangular superior:

Dado el sistema

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1\,n-1} & a_{1n} \\ 0 & a_{22} & \cdots & a_{2\,n-1} & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1\,n-1} & a_{n-1\,n} \\ 0 & 0 & \cdots & 0 & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}$$

es sencillo obtener su solución como

$$x_n = \frac{b_n}{a_{nn}} \; ; \; x_k = \frac{1}{a_{kk}} \left(b_k - \sum_{j=k+1}^{n} a_{kj} x_j \right), \; k = n-1, \; n-2, \; \ldots, \; 1$$

que podemos programar, con el siguiente fichero de función:

```
function x=sustitucion_regresiva(A,b)
% Función x=sustitucion_regresiva(A,b)
% Resolución de Ax=b, cuando A es triangular superior
x=[]; [m n]=size(A);
if m~=n, disp('ERROR: Matriz del sistema NO Cuadrada'), return, end
if m~=length(b), disp('ERROR: Sistema NO Coherente'), return, end
if isequal(A,triu(A))==0
    disp('ERROR: Matriz NO Triangular Superior'), return
end
x=zeros(n,1); x(n)=b(n)/A(n,n);
for k=n-1:-1:1
    x(k)=(b(k)-A(k,k+1:n)*x(k+1:n))/A(k,k);
end
```

Nota: `isequal(A,B)` devuelve el valor `1`, si `A`=`B`, y es igual a `0`, en caso contrario.

Sistema triangular inferior:

Si el sistema es de la forma

$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 & 0 \\ a_{21} & a_{22} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-1\,1} & a_{n-1\,2} & \cdots & a_{n-1\,n-1} & 0 \\ a_{n1} & a_{n2} & \cdots & a_{n\,n-1} & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}$$

su solución resulta

$$x_1 = \frac{b_1}{a_{11}} \; ; \; x_k = \frac{1}{a_{kk}} \left(b_k - \sum_{j=1}^{k-1} a_{kj} x_j \right), \; k = 2, \; 3, \; \ldots, \; n$$

que se puede programar de la siguiente manera:

```
function x=sustitucion_progresiva(A,b)
% Función x=sustitucion_progresiva(A,b)
% Resolución de Ax=b, cuando A es triangular inferior
x=[]; [m n]=size(A);
if m~=n, disp('ERROR: Matriz del sistema NO Cuadrada'), return, end
if m~=length(b), disp('ERROR: Sistema NO Coherente'), return, end
if isequal(A,tril(A))==0
    disp('ERROR: Matriz NO Triangular Inferior'), return
end
x=zeros(n,1); x(1)=b(1)/A(1,1);
for k=2:n
    x(k)=(b(k)-A(k,1:k-1)*x(1:k-1))/A(k,k);
end
```

Tras esto, pasamos a programar el método de Gauss. Como se sabe, consiste en hacer ceros por debajo de la diagonal principal, usando como pivotes los elementos de dicha diagonal, por lo que, de ser alguno nulo, debe buscarse algún elemento no nulo por debajo del mismo y, una vez encontrado, proceder al intercambio de filas.

De esta forma, el método podría programarse como sigue:

```
function x=gauss(A,b)
% Función x=gauss(A,b)
% Resolución de Ax=b, por el método de Gauss
x=[]; [m n]=size(A);
if m~=n, disp('ERROR: Matriz del sistema NO Cuadrada'),return, end
if m~=length(b), disp('ERROR: Sistema NO Coherente'), return, end
for k=1:n
    if A(k,k)==0
        j=find(A(k+1:n,k),1);
        if isempty(j)
            disp('ERROR:Matriz de coeficientes singular'), return
        end
        j=j+k; A([k j],:)=A([j k],:); b([k j])=b([j k]);
    end
    for i=k+1:n
        z=A(i,k);
        A(i,k:n)=A(i,k:n)-A(k,k:n)/A(k,k)*z;
        b(i)=b(i)-b(k)/A(k,k)*z;
    end
end
x=sustitucion_regresiva(A,b);
```

En este fichero se han utilizado las órdenes **find** e **isempty**, que funcionan de la siguiente forma:

`j=find(z,m)`: construye el vector `j` con las posiciones que ocupan, dentro del vector `z`, sus `m` primeros elementos no nulos.

`isempty(z)`: estudia si la variable `z=[]`.

Si trabajásemos con aritmética exacta, el método que acabamos de programar sería suficiente para resolver un sistema, pero hay que tener en cuenta que, al hacer ceros, se necesita dividir por los pivotes y que, en MATLAB, si se divide por una cantidad próxima a cero, los errores de redondeo pueden producir grandes errores en el cociente, por lo que una estrategia para minimizar este problema consiste en elegir los pivotes como los elementos de mayor valor absoluto posible. De esta forma, surge el llamado *método de Gauss con pivoteo*, que podemos programar de la siguiente manera:

```
function x=gausspivoteo(A,b)
% Función x=gausspivoteo(A,b)
% Resolución de Ax=b, por el método de Gauss con pivoteo
x=[]; [m n]=size(A);
if m~=n, disp('ERROR: Matriz del sistema NO Cuadrada'), return, end
if m~=length(b), disp('ERROR: Sistema NO Coherente'), return, end
for k=1:n
    [p j]=max(abs(A(k:n,k))); j=j+k-1;
    if p==0
        disp('ERROR:Matriz de coeficientes singular'), return
    end
    if k~=j
        A([k j],:)=A([j k],:); b([k j])=b([j k]);
    end
    for i=k+1:n
        z=A(i,k);
        A(i,k:n)=A(i,k:n)- A(k,k:n)/A(k,k)*z;
        b(i)=b(i)- b(k)/A(k,k)*z;
    end
end
x=sustitucion_regresiva(A,b);
```

En el archivo anterior, hemos utilizado la orden **max** con dos argumentos de salida, cuyo funcionamiento es como sigue:

`[max_v j]=max(v)`: da como resultados el mayor elemento de `v`, `max_v`, y la posición `j` que ocupa dicho elemento, dentro del vector `v`.

Ejemplo 5.1 *Dado el sistema* $\begin{cases} 10^{-15}x + y = 1 \\ x + y = 2 \end{cases}$ *, se pide:*

a) Denotar $\mathbf{s}$ *y* $\mathbf{s_1}$ *a las soluciones obtenidas por el método de Gauss, con y sin pivoteo, respectivamente.*

b) Hallar, en porcentaje, el error relativo cometido al hacer la aproximación $\mathbf{s_1} \approx \mathbf{s}$*, así como los errores absolutos cometidos en cada incógnita.*

Solución:

Apartado a): Introducimos los datos y resolvemos, con

```
>> A=[1e-15 1;1 1], b=[1;2]
A =
    0.0000    1.0000
    1.0000    1.0000
b =
     1
     2
>> s=gausspivoteo(A,b)
s =
    1.0000
    1.0000
>> s1=gauss(A,b)
s1 =
    0.9992
    1.0000
```

Apartado b): Para hallar los errores, basta hacer

```
>> error_porcentual=norm(s1-s)/norm(s)*100
error_porcentual =
    0.0565
>> errores_absolutos=abs(s1-s)
errores_absolutos =
  1.0e-003 *
    0.7993
         0
```

5.2. Factorización LU

Como se sabe, si la matriz A es regular, existe una permutación de las filas de A, tal que la matriz resultante se puede descomponer en la forma LU, donde L es una matriz triangular inferior, con todos los elementos de la diagonal principal iguales a 1, y U es una matriz triangular superior.

En otras palabras, si A es regular, existe una matriz de permutación P, tal que $PA = LU$, siendo L una matriz triangular inferior, con todos los elementos de la diagonal principal iguales a 1, y U una matriz triangular superior. Así, si se conocen L, U y P, el problema se reduce a resolver dos sistemas triangulares.

Pues bien, en MATLAB, las matrices L, U y P se calculan ejecutando `[L U P]=lu(A)`.

Ejemplo 5.2 *Resuélvase el sistema*

$$\begin{cases} x_2 + x_3 = 5 \\ x_1 + x_3 = 4 \\ x_1 + x_2 = 3 \end{cases}$$

utilizando la descomposición LU.

Solución: Introducimos los datos y hallamos L, U y P, haciendo

```
>> A=[0 1 1;1 0 1;1 1 0]; b=[5;4;3];
>> [L U P]=lu(A)
L =
     1     0     0
     0     1     0
     1     1     1
U =
     1     0     1
     0     1     1
     0     0    -2
P =
     0     1     0
     1     0     0
     0     0     1
```

con lo que, si en el sistema de partida $A\,\mathbf{x} = \mathbf{b}$, multiplicamos por P, resulta

$$PA\,\mathbf{x} = P\,\mathbf{b} \implies LU\,\mathbf{x} = \mathbf{b}_1, \text{ con } \mathbf{b}_1 = P\mathbf{b}$$

y, haciendo $U\,\mathbf{x} = \mathbf{y}$, el problema se reduce a resolver $L\,\mathbf{y} = \mathbf{b}_1$, que es un sistema triangular inferior, y, a continuación, el sistema $U\,\mathbf{x} = \mathbf{y}$, que es triangular superior.

Por tanto, hacemos

```
>> b1=P*b
b1 =
     4
     5
     3
>> y=sustitucion_progresiva(L,b1)
y =
     4
     5
    -6
```

```
>> x=sustitucion_regresiva(U,y)
x =
     1
     2
     3
```

5.3. Factorización de Cholesky

Si A es una matriz simétrica definida positiva, es posible descomponer $A = LU$, con L triangular inferior y U triangular superior, de forma que L y U sean traspuestas una de la otra, con lo que la factorización sería:

$$A = {}^tUU$$

Pues bien, dicha matriz U, en MATLAB, se calcula ejecutando `chol(A)`.

Ejemplo 5.3 *Resuélvase el sistema*

$$\begin{cases} x_1 + x_2 + x_3 = 0 \\ x_1 + 2x_2 + 2x_3 = -1 \\ x_1 + 2x_2 + 6x_3 = 1 \end{cases}$$

utilizando la factorización de Cholesky, comprobando previamente que dicha factorización es posible.

Solución: Introducimos los datos

```
>> clear, A=[1 1 1;1 2 2;1 2 6], b=[0;-1;1]
A =
     1     1     1
     1     2     2
     1     2     6
b =
     0
    -1
     1
```

y se observa, a simple vista, que la matriz es simétrica, si bien se puede comprobar por medio de:

```
>> isequal(A,A.')
ans =
     1
```

Para ver que A es definida positiva, calculamos sus menores principales, con

```
>> for i=1:3, D(i)=det(A(1:i,1:i)); end
>> D
D =
     1     1     4
```

y, como son todos estrictamente positivos, efectivamente, A es simétrica definida positiva.

Nota: Si A fuera de grandes dimensiones, convendría hacer:

```
>> min(D)
ans =
     1
```

Así, podemos factorizar $A = {}^tUU$, con U triangular superior, con lo que $A\,\mathbf{x} = \mathbf{b}$ pasa a ser ${}^tUU\,\mathbf{x} = \mathbf{b}$ y, como en el ejemplo anterior, denotando $U\,\mathbf{x} = \mathbf{y}$, el problema se reduce a resolver el sistema triangular inferior ${}^tU\,\mathbf{y} = \mathbf{b}$ y, a continuación, el sistema triangular superior $U\,\mathbf{x} = \mathbf{y}$, con lo cual:

```
>> U=chol(A)
U =
     1     1     1
     0     1     1
     0     0     2
>> y=sustitucion_progresiva(U.',b)
y =
     0
    -1
     1
>> x=sustitucion_regresiva(U,y)
x =
    1.0000
   -1.5000
    0.5000
```

5.4. La operación \

En MATLAB, la solución de $A\,\mathbf{x} = \mathbf{b}$, se puede obtener sin más que ejecutar `A\b`. Además, con esta operación, el programa resuelve el sistema con algoritmos que se adaptan a las características de la matriz de coeficientes. Por ejemplo: si la matriz es triangular, según sea superior o inferior, utiliza sustitución regresiva o progresiva, respectivamente; si es una matriz simétrica definida positiva, utiliza la factorización de Cholesky, etc.

Ejemplo 5.4 *Resuélvase el ejemplo anterior, utilizando la operación* \ .

Solución: Basta hacer

```
>> A\b
ans =
    1.0000
   -1.5000
    0.5000
```

6. Práctica 6

Para finalizar con los sistemas de ecuaciones, en esta práctica, estudiaremos brevemente los problemas de condicionamiento de sistemas lineales, para, a continuación, estudiar la resolución de sistemas lineales, utilizando métodos iterativos, y terminaremos con el método de Newton-Raphson de resolución de sistemas no lineales.

6.1. Condicionamiento de sistemas lineales

El *número de condición* de una matriz A, que denotaremos $cond(A)$, permite saber si la solución de un sistema $A\,\mathbf{x} = \mathbf{b}$ se verá afectada de forma importante, al realizar una pequeña modificación en los coeficientes de A o $\mathbf{b}$. Esto es importante, por ejemplo, cuando los datos provienen de medidas experimentales, por lo que pueden contener errores.

Dicho número de condición es siempre mayor o igual que 1 y el hecho de que $cond(A) \gg 1$, significa que pequeños cambios en los datos pueden alterar considerablemente la solución del problema.

Con MATLAB, el número de condición de una matriz A se calcula ejecutando `cond(A)`.

Ejemplo 6.1 *Se considera el sistema* $A\,\mathbf{x} = \mathbf{b}$*, donde*

$$A = \begin{pmatrix} -4 & 1 \\ 1 & -4 \end{pmatrix},\ \mathbf{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

y se pide:

a) Calcular el número de condición de A.

b) Denotando $\mathbf{x}$ *a la solución del sistema y* $\mathbf{x_1}$ *a la del sistema resultante al sumar una milésima al último elemento de A, hallar, en porcentaje, el error relativo cometido al hacer* $\mathbf{x} \approx \mathbf{x_1}$*, así como el mayor error absoluto cometido en las incógnitas, determinando en cuál de ellas se produce dicho error máximo.*

Solución:

Apartado a): Basta hacer

```
>> A=[-4 1;1 -4]
A =
    -4     1
     1    -4
>> cond(A)
ans =
    1.6667
```

Apartado b): Para hallar $\mathbf{x}$, $\mathbf{x_1}$ y los errores cometidos al hacer $\mathbf{x} \approx \mathbf{x_1}$, ejecutamos

```
>> b=[1;1], x=A\b
b =
     1
     1
x =
   -0.3333
   -0.3333
>> A(2,2)=A(2,2)+0.001
A =
   -4.0000    1.0000
    1.0000   -3.9990
>> x1=A\b
x1 =
   -0.3334
   -0.3334
>> error_porcentual=norm(x1-x)/norm(x)*100
error_porcentual =
    0.0194
>> [error_absoluto j]=max(abs(x1-x))
error_absoluto =
  8.8913e-005
j =
     2
```

de lo que se deduce que el error relativo es del 0.0194 % y el mayor error absoluto (8.8913e-005) se produce en la segunda incógnita.

Ejemplo 6.2 *Repítase el ejemplo anterior con el sistema* $A\,\mathbf{x} = \mathbf{b}$*, donde:*

$$A = \begin{pmatrix} 3.021 & 2.714 & 6.913 \\ 1.031 & -4.273 & 1.121 \\ 5.084 & -5.832 & 9.155 \end{pmatrix}, \ \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Solución: Basta seguir los pasos del ejercicio anterior, haciendo

```
>> clear, A=[3.021 2.714 6.913;1.031 -4.273 1.121;5.084 -5.832 9.155]
A =
    3.0210    2.7140    6.9130
    1.0310   -4.2730    1.1210
    5.0840   -5.8320    9.1550
>> cond(A)
ans =
  3.7269e+004
```

```
>> b=[1;1;1], x=A\b
b =
     1
     1
     1
x =
  1.0e+003 *
   -2.0000
   -0.2298
    0.9644
>> A(3,3)=A(3,3)+0.001
A =
    3.0210    2.7140    6.9130
    1.0310   -4.2730    1.1210
    5.0840   -5.8320    9.1560
>> x1=A\b
x1 =
  1.0e+003 *
   -3.8620
   -0.4436
    1.8620
>> error_porcentual=norm(x1-x)/norm(x)*100
error_porcentual =
   93.0950
>> [error_absoluto j]=max(abs(x1-x))
error_absoluto =
  1.8620e+003
j =
     1
```

6.2. Métodos iterativos para sistemas lineales

A continuación, pasamos a los métodos iterativos, comenzando por el del punto fijo para, como consecuencia, obtener el método de Jacobi.

Método del punto fijo

Sea B una matriz cuadrada de orden n, $\mathbf{c} \in \mathbb{R}^n$, g la función vectorial que, a cada $\mathbf{x} \in \mathbb{R}^n$, le hace corresponder

$$g(\mathbf{x}) = B\mathbf{x} + \mathbf{c}$$

y sea $\rho(B)$ el radio espectral de B, es decir, el máximo de los módulos de los autovalores de B.

Entonces, la condición $\rho(B) < 1$ es necesaria y suficiente para que la función $g(\mathbf{x})$ posea un único punto fijo, es decir, exista un único $\mathbf{x} \in \mathbb{R}^n$, tal que $g(\mathbf{x}) = \mathbf{x}$.

Además, si fijamos un punto cualquiera $\mathbf{x}_0 \in \mathbb{R}^n$ y definimos la sucesión

$$\mathbf{x}_1 = g(\mathbf{x}_0),\ \mathbf{x}_2 = g(\mathbf{x}_1),\ \ldots\ ,\ \mathbf{x}_{k+1} = g(\mathbf{x}_k),\ \ldots$$

dicha sucesión converge al punto fijo de g.

Método de Jacobi

Consideremos el sistema $A\,\mathbf{x} = \mathbf{b}$, siendo A una matriz cuadrada de orden n, descompuesta en la forma $A = D + L + U$, donde D es una matriz diagonal, L triangular inferior, con diagonal principal nula, y U triangular superior, también con diagonal principal nula.

Suponemos que los elementos de la diagonal de A son todos no nulos, con lo que D es regular, y definimos:

$$g(\mathbf{x}) = B\mathbf{x} + \mathbf{c},\ \text{con}\ B = -D^{-1}(L+U),\ \mathbf{c} = D^{-1}\mathbf{b}$$

Si desarrollamos $g(\mathbf{x}) = \mathbf{x}$, resulta

$$-D^{-1}(L+U)\mathbf{x} + D^{-1}\mathbf{b} = \mathbf{x} \implies -(L+U)\mathbf{x} + \mathbf{b} = D\mathbf{x} \implies \mathbf{b} = (D+L+U)\mathbf{x} \implies \mathbf{b} = A\mathbf{x}$$

y, por tanto, resolver $A\mathbf{x} = \mathbf{b}$ equivale a resolver $g(\mathbf{x}) = \mathbf{x}$.

De esta forma, si $\rho(B) < 1$, a partir de cualquier $\mathbf{x}_0 \in \mathbb{R}^n$, podemos hallar la solución de $A\,\mathbf{x} = \mathbf{b}$ como el límite de la sucesión:

$$\mathbf{x}_{k+1} = g(\mathbf{x}_k) = B\,\mathbf{x}_k + \mathbf{c} \implies \mathbf{x}_{k+1} = -D^{-1}(L+U)\mathbf{x}_k + D^{-1}\mathbf{b},\ k = 0, 1, 2, \ldots$$

Planteando la ecuación anterior como $D\mathbf{x}_{k+1} = \mathbf{b} - (L+U)\mathbf{x}_k$ y denotando a_{ij} a los elementos de A, b_i a los de $\mathbf{b}$ y $x_i^{(k)}$ a los de $\mathbf{x}_k$, resulta

$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} - \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix} \begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{pmatrix}$$

con lo que el elemento i-ésimo de la anterior igualdad es

$$a_{ii}x_i^{(k+1)} = b_i - (a_{i1}x_1^{(k)} + a_{i2}x_2^{(k)} + \cdots + a_{i\,i-1}x_{i-1}^{(k)} + a_{i\,i+1}x_{i+1}^{(k)} + \cdots + a_{in}x_n^{(k)}) \implies$$

$$\implies x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}x_j^{(k)} \right),\ i = 1, 2, \ldots, n$$

y podemos programar el método como sigue:

```
function [x k]=jacobi(A,b,T,N)
% Función [x k]=jacobi(A,b,T,N)
% Resolución de Ax=b, por el método de Jacobi
% ARGUMENTOS DE ENTRADA:
%   A    ..... Matriz cuadrada
%   b    ..... Vector columna
%   T  ........ Tolerancia relativa permitida
%   N    ..... Número máximo de iteraciones
% ARGUMENTOS DE SALIDA:
%   x .......  Solución
%   k .......  Número de iteraciones efectuadas
x=[]; k=0; % k es el contador de iteraciones
[m n]=size(A);
if m~=n, disp('ERROR: Matriz del sistema NO Cuadrada'), return, end
if m~=length(b), disp('ERROR: Sistema NO Coherente'), return, end
if min(abs(diag(A)))==0
   disp('ERROR: Método no válido por existir elemento diagonal nulo')
   return
end
x0=zeros(n,1); x=zeros(n,1);
test_parada=0;
while test_parada==0 && k<N
    k=k+1;
    for i=1:n
        x(i)=(b(i)-A(i,[1:i-1 i+1:n])*x0([1:i-1 i+1:n]))/A(i,i);
    end
    test_parada=norm(x-x0)<=T*norm(x0);
    x0=x;
end
if k==N
    disp('No converge con la precisión pedida.')
    disp('El valor hallado no es la solución, sino la última iteración:')
end
```

donde, como siempre, hemos utilizado

$$\frac{\|\mathbf{x}_{n+1}-\mathbf{x}_n\|}{\|\mathbf{x}_n\|}\leq T$$

como condición de parada, planteada en la forma $\|\mathbf{x}_{n+1}-\mathbf{x}_n\|\leq T\cdot|\mathbf{x}_n\|$.

Ejemplo 6.3 *Resuélvase el sistema*

$$\begin{cases} 4x_1+x_2-x_3+2x_4=1\\ x_1+3x_2+x_3=1\\ x_1+2x_2+5x_3+x_4=1\\ x_1-x_2+2x_3+6x_4=1 \end{cases}$$

por el método de Jacobi, con tolerancia relativa 10^{-6}*, comprobando previamente que es posible aplicar dicho método.*

Solución: Introducimos los datos, haciendo

```
>> clear, A=[4 1 -1 2;1 3 1 0;1 2 5 1;1 -1 2 6], b=[1;1;1;1]
A =
     4     1    -1     2
     1     3     1     0
     1     2     5     1
     1    -1     2     6
b =
     1
     1
     1
     1
```

y, a simple vista, se observa que la diagonal principal de A la forman elementos no nulos, si bien (en el caso de que A fuera de gran tamaño) se puede corroborar comprobando que

```
>> min(abs(diag(A)))
ans =
   3
```

es distinto de cero.

A continuación, descomponemos $A = D + L + U$

```
>> D=diag(diag(A))
D =
     4     0     0     0
     0     3     0     0
     0     0     5     0
     0     0     0     6
>> L=tril(A,-1)
L =
     0     0     0     0
     1     0     0     0
     1     2     0     0
     1    -1     2     0
>> U=triu(A,1)
U =
     0     1    -1     2
     0     0     1     0
     0     0     0     1
     0     0     0     0
```

construimos la matriz de iteración de Jacobi y hallamos su radio espectral, utilizando la orden `eig(B)`, que construye un vector, cuyos elementos son los valores propios de `B`,

```
>> B=-inv(D)*(L+U)
B =
         0   -0.2500    0.2500   -0.5000
   -0.3333         0   -0.3333         0
   -0.2000   -0.4000         0   -0.2000
   -0.1667    0.1667   -0.3333         0
>> radio_espectral=max(abs(eig(B)))
radio_espectral =
    0.6171
```

y, al ser éste menor que 1, podemos resolver por el método de Jacobi, haciendo:

```
>> x=jacobi(A,b,1e-6,100)
x =
    0.0858
    0.2961
    0.0258
    0.1931
```

6.3. Sistemas no lineales

Para finalizar, estudiemos un sistema no lineal de la forma:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{cases}$$

En primer lugar, se plantea en forma vectorial, denotando $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ y

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{pmatrix}$$

con lo que el sistema de partida se reduce a $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

De los distintos métodos de resolución que se pueden utilizar para un sistema de este tipo, nos centraremos en el *método de Newton-Raphson*, basado en construir una sucesión $\mathbf{x}_k$ convergente a la solución buscada.

Dicha sucesión se define partiendo de una estimación inicial $\mathbf{x}_0$ y definiendo el resto de elementos como

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [d\mathbf{f}(\mathbf{x}_k)]^{-1}\mathbf{f}(\mathbf{x}_k)$$

donde se ha utilizado la notación:

$$d\mathbf{f} = \begin{pmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} & \cdots & \dfrac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \dfrac{\partial f_n}{\partial x_2} & \cdots & \dfrac{\partial f_n}{\partial x_n} \end{pmatrix}$$

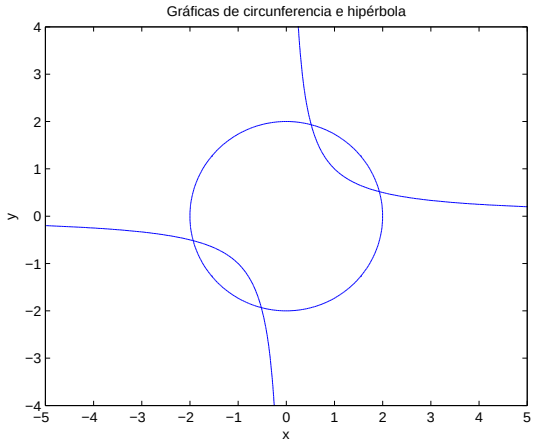
Una forma de programar este método, como fichero de función, es la siguiente:

```
function [x k]=newton_n(f,df,x0,T,N)
% Función [x k]=newton_n(f,df,x0,T,N)
% Resolución del sistema f(x)=0, por el método de Newton-Raphson
% ARGUMENTOS DE ENTRADA:
%   f    ..... Función numérica vectorial (DIMENSIÓN n x 1)
%   df  ..... Función diferencial de f, en forma numérica vectorial
%   x0 ..... Estimación inicial (DIMENSIÓN n x 1)
%   T  ..... Tolerancia relativa permitida
%   N    ..... Número máximo de iteraciones
% ARGUMENTOS DE SALIDA:
%   x .....  Solución
%   k .....  Número de iteraciones efectuadas
test_parada=0; k=0; % k es el contador de iteraciones
while test_parada==0 && k<N
    k=k+1;
    h=-df(x0)\f(x0);
    x=x0+h;
    test_parada=norm(h)<=T*norm(x0);
    x0=x;
end
if k==N
    disp('No converge con la precisión pedida.')
    disp('El valor hallado no es la solución, sino la última iteración:')
end
```

Ejemplo 6.4 *Hállense los puntos de corte de la circunferencia* $x^2 + y^2 = 4$ *y la hipérbola* $xy = 1$.

Solución: En primer lugar, hacemos las representaciones gráficas, con

```
>> ezplot('x^2+y^2=4',[-5 5 -4 4])
>> hold on, ezplot('x*y=1'), title('Gráficas de circunferencia e hipérbola')
```



y construimos la función vectorial **f** que define el sistema, planteándola en forma simbólica, para que MATLAB calcule las derivadas, por medio de

```
>> syms x y, f=[x^2+y^2-4;x*y-1]
f =
 x^2 + y^2 - 4
       x*y - 1
```

para, a continuación, hallar su matriz jacobiana con la orden:

```
>> df=jacobian(f)
df =
[ 2*x, 2*y]
[   y,   x]
```

Tras esto, convertimos ambas funciones a numéricas vectoriales, ejecutando

```
>> f_num=matlabFunction(f,'vars',[x y])
f_num =
    @(x,y)[x.^2+y.^2-4.0;x.*y-1.0]
>> f_vectorial=@(z) f_num(z(1),z(2))
f_vectorial =
    @(z)f_num(z(1),z(2))
>> df_num=matlabFunction(df,'vars',[x y])
df_num =
    @(x,y)reshape([x.*2.0,y,y.*2.0,x],[2,2])
>> df_vectorial=@(z) df_num(z(1),z(2))
df_vectorial =
    @(z)df_num(z(1),z(2))
```

y, por último, calculamos los puntos de corte, aplicando el método de Newton:

```
>> p1=newton_n(f_vectorial,df_vectorial,[-2;0],1e-6,100)
p1 =
   -1.9319
   -0.5176
>> p2=newton_n(f_vectorial,df_vectorial,[0;-2],1e-6,100)
p2 =
   -0.5176
   -1.9319
>> p3=newton_n(f_vectorial,df_vectorial,[1;2],1e-6,100)
p3 =
    0.5176
    1.9319
>> p4=newton_n(f_vectorial,df_vectorial,[2;1],1e-6,100)
p4 =
    1.9319
    0.5176
```

7. Práctica 7

Iniciamos aquí el estudio de la aproximación de funciones, con el que pretendemos resolver el siguiente problema:

Dado un conjunto de puntos (x_i, y_i), $i = 1, 2, \ldots, n+1$, donde, generalmente, los y_i serán los valores $f(x_i)$ que toma una cierta función f en los x_i, se plantea el problema de calcular una función, de manejo relativamente sencillo, que aproxime la función f. La técnica que se utilizará será la de exigir que la función aproximante pase por los puntos de partida o, en su defecto, "cerca"de ellos y, según la técnica de aproximación que utilicemos, cambiará la forma de construir la función.

En esta práctica, nos centraremos en la *interpolación polinómica*, que consiste en calcular un polinomio que pase por todos los puntos (x_i, y_i).

Como quiera que MATLAB tiene órdenes específicas para manejar polinomios, empezamos por hacer un breve resumen de ellas.

7.1. Polinomios en MATLAB

El polinomio $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ se puede representar por medio del vector `p=[`$\mathtt{a_n\ a_{n-1}\ \cdots\ a_1\ a_0}$`]` y, para manejar un polinomio escrito en formato vectorial, disponemos de diversas órdenes, de entre las que utilizaremos las siguientes:

`polyval(p,t)`: Evalúa el polinomio `p` en `t`.

`polyder(p)`: Deriva el polinomio `p`.

`roots(p)`: Halla las raíces del polinomio `p`.

`conv(p,q)`: Multiplica los polinomios `p` y `q`.

A continuación, vemos algunos ejemplos en los que se utilizan estas órdenes:

```
>> p=[1 1 1] % (Polinomio x^2+x+1)
p =
     1     1     1
>> polyval(p,2)
ans =
     7
>> p1=polyder(p)
p1 =
     2     1
>> r=roots([1 0 -9]) % (Raíces de x^2-9)
r =
     3
    -3
```

```
>> conv([1 1],[1 -1]) % (Producto de x+1 por x-1)
ans =
     1     0    -1
```

7.2. Interpolación de Lagrange

Dados $n+1$ puntos $(x_1, y_1), (x_2, y_2), \ldots, (x_{n+1}, y_{n+1})$, con $x_i \neq x_j$, para $i \neq j$, existe un único polinomio de grado menor o igual que n que pasa por todos ellos, denominado polinomio interpolador. Dicho polinomio se puede calcular, utilizando la fórmula de Lagrange, como

$$p(x) = y_1 l_1(x) + y_2 l_2(x) + \cdots + y_{n+1} l_{n+1}(x)$$

donde los $l_j(x)$ son los polinomios de Lagrange, que vienen definidos por:

$$l_j(x) = \prod_{\substack{i=1 \\ i \neq j}}^{n+1} \frac{x - x_i}{x_j - x_i}$$

Una forma de programar la construcción de los polinomios de Lagrange, como fichero de función, es crear el fichero lagrange.m con las órdenes

```
function L=lagrange(x)
%   Función L=lagrange(x) que calcula los polinomios de lagrange
%   correspondientes a los nodos del vector x
% ARGUMENTO DE ENTRADA:
% x ......... Vector que contiene los nodos
% ARGUMENTO DE SALIDA:
% L ......... Matriz cuya fila j contiene el polinomio de Lagrange l_j(x)
m=length(x); for j=1:m
    l=1;
    for i=[1:j-1 j+1:m]
        l=conv(l,[1 -x(i)])/(x(j)-x(i));
    end
    L(j,:)=l;
end
```

con lo cual, una vez construida la matriz de polinomios de Lagrange como

$$\begin{pmatrix} l_1(x) \\ l_2(x) \\ \vdots \\ l_{n+1}(x) \end{pmatrix}$$

para calcular el polinomio interpolador, bastará hacer el producto matricial:

$$(y_1 \ y_2 \ \ldots \ y_{n+1}) \begin{pmatrix} l_1(x) \\ l_2(x) \\ \vdots \\ l_{n+1}(x) \end{pmatrix} = y_1 l_1(x) + y_2 l_2(x) + \cdots + y_{n+1} l_{n+1}(x)$$

Ejemplo 7.1 *Dada la tabla de valores*

x	0	1	2	3	4	5
y	1.1	1.5	2.4	2	3	1

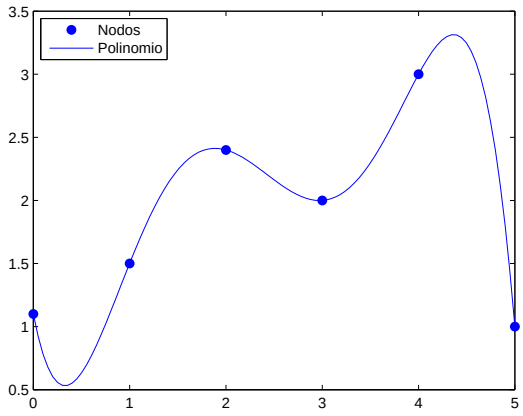
hállese el correspondiente polinomio de interpolación y represéntese su gráfica, junto con los nodos de interpolación.

Solución: Introducimos los datos y calculamos el polinomio interpolador, haciendo

```
>> x=0:5, y=[1.1 1.5 2.4 2 3 1]
x =
     0     1     2     3     4     5
y =
    1.1000    1.5000    2.4000    2.0000    3.0000    1.0000
>> L=lagrange(x);
>> p=y*L
p =
   -0.0967    1.1542   -4.8083    8.0458   -3.8950    1.1000
```

y, por último, representamos gráficamente:

```
>> plot(x,y,'.','markersize',20)
>> xx=linspace(0,5);
>> yy=polyval(p,xx);
>> hold on, plot(xx,yy)
>> legend('Nodos','Polinomio','location','northwest')
```



El polinomio interpolador como aproximación de una función

En muchos casos, los datos y_i son los valores de una función $f(x)$ que se desea aproximar, es decir, $y_i = f(x_i)$. Evidentemente, al hacer $f(x) \approx p(x)$, estaremos cometiendo un error, salvo que x sea uno de los nodos. En realidad, se tendrá

$$f(x) = p(x) + E(x),$$

siendo $E(x) = f(x) - p(x)$ el error cometido.

Pues bien, si $f \in \mathcal{C}^{n+1}[a,b]$, y $x_1, x_2, \ldots, x_{n+1} \in [a,b]$, para cualquier $x \in [a,b]$, existe $c \in [a,b]$, tal que:

$$E(x) = \frac{1}{(n+1)!} \prod_{i=1}^{n+1} (x - x_i)\, f^{(n+1)}(c)$$

Ejemplo 7.2 *Aproxímese la función $f(x) = x \operatorname{sen} x$, en el intervalo $[0,3]$, por el siguiente método:*

Constrúyase un vector de 5 elementos x_i, igualmente espaciados en dicho intervalo, y calcúlese el polinomio interpolador $p(x)$ que pasa por los puntos $(x_i, f(x_i))$. A continuación, hállese el error máximo cometido al aproximar $f(x)$ por $p(x)$ y represéntense, para $0 \leq x \leq 3$, tanto el polinomio hallado como la función $f(x)$, observando el fenómeno de extrapolación que se produce si dichas gráficas se hacen en el intervalo $[0,4]$.

Solución: Definimos $f(x)$ y calculamos el polinomio interpolador, con

```
>> f=@(x) x.*sin(x)
f =
    @(x)x.*sin(x)
>> x=linspace(0,3,5), y=f(x)
x =
         0    0.7500    1.5000    2.2500    3.0000
y =
         0    0.5112    1.4962    1.7507    0.4234
>> L=lagrange(x)
L =
    0.1317   -0.9877    2.5926   -2.7778    1.0000
   -0.5267    3.5556   -7.7037    5.3333         0
    0.7901   -4.7407    8.4444   -4.0000         0
   -0.5267    2.7654   -4.1481    1.7778         0
    0.1317   -0.5926    0.8148   -0.3333         0
>> p=y*L
p =
    0.0465   -0.6851    1.7795   -0.2872         0
```

y, al hacer $f(x) \approx p(x)$, el error viene dado por

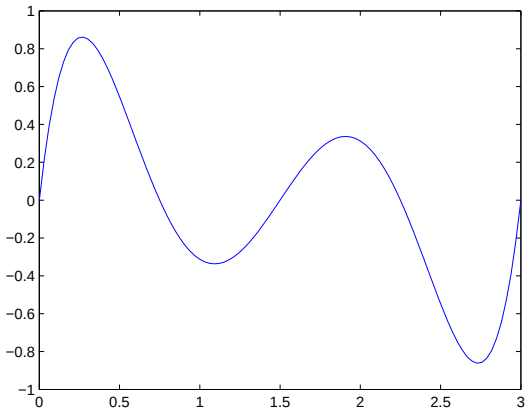
$$E(x) = \frac{1}{5!} \prod_{i=1}^{5} (x - x_i)\, f^{(v)}(c) = \frac{1}{5!} q(x) f^{(v)}(c), \text{ con } q(x) = \prod_{i=1}^{5} (x - x_i)$$

con lo cual, el error máximo que podría darse sería $\frac{1}{5!} K \cdot M$, siendo:

$$K = \text{máx}\{|q(x)| :\ 0 \leq x \leq 3\},\ M = \text{máx}\{|f^{(v)}(x)| :\ 0 \leq x \leq 3\}$$

Para hallar K, empezamos por construir el polinomio $q(x)$ y representarlo gráficamente:

```
>> q=1; for i=1:5, q=conv(q,[1 -x(i)]); end; q
q =
    1.0000   -7.5000   19.6875  -21.0938    7.5938         0
>> xx=linspace(0,3); yy=polyval(q,xx); plot(xx,yy)
```



A continuación, vamos a calcular los valores de x en los que $q(x)$ alcanza los máximos y mínimos relativos que se observan en la gráfica anterior. Como, en dichos puntos, $q'(x) = 0$, construimos $q'(x)$ con la orden **polyder**,

```
>> q1=polyder(q)
q1 =
    5.0000  -30.0000   59.0625  -42.1875    7.5938
```

con lo que $q'(x)$ se anula para los valores de x dados por

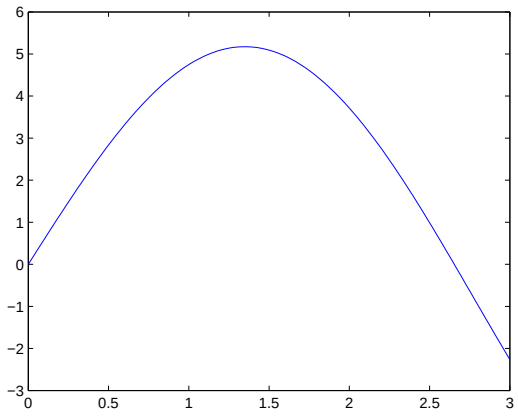
```
>> r=roots(q1)
r =
    2.7333
    1.9079
    1.0921
    0.2667
```

y, por tanto, el máximo de $|q(x)|$, en $[0, 3]$, es:

```
>> K=max(abs(polyval(q,r)))
K =
    0.8618
```

Para calcular el máximo de $|f^{(v)}(x)|$, vamos a denotar $g(x) = f^{(v)}(x)$ y la representamos, con

```
>> syms x, g=diff(f(x),5)
g =
5*sin(x) + x*cos(x)
>> yy=subs(g,x,xx); plot(xx,yy)
```



y, para obtener el máximo relativo que observamos en la figura, debemos estudiar el punto en que $g'(x) = 0$. Esto lo haremos, utilizando la orden **fzero** de la siguiente forma:

```
>> g1=diff(g)
g1 =
6*cos(x) - x*sin(x)
>> g1_num=matlabFunction(g1)
g1_num =
    @(x)cos(x).*6.0-x.*sin(x)
>> c=fzero(g1_num,1)
c =
    1.3496
```

Por tanto, el máximo de $|f^{(v)}(x)|$ es

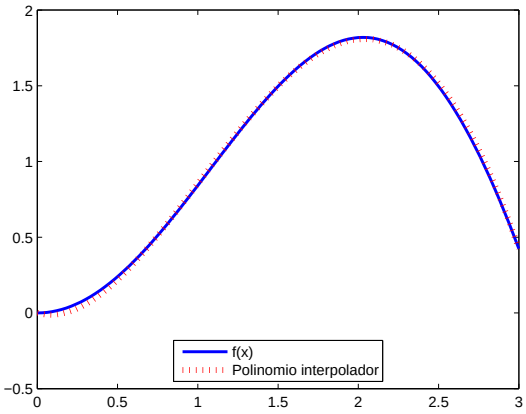
```
>> M=subs(g,x,c)
M =
    5.1743
```

y el error máximo viene dado por

```
>> error_maximo=K*M/factorial(5)
error_maximo =
    0.0372
```

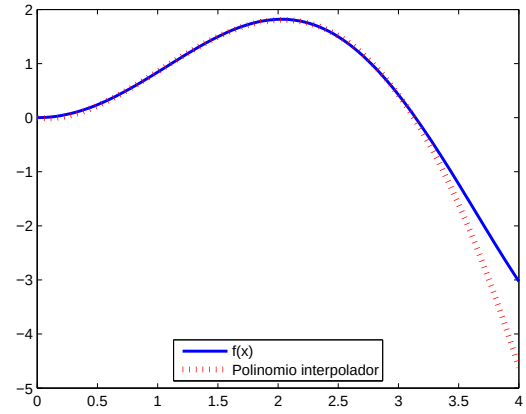
con lo que se puede considerar que la aproximación es razonablemente buena, como podemos ver a continuación:

```
>> plot(xx,f(xx),'linewidth',2)
>> yy=polyval(p,xx); hold on, plot(xx,yy,'r:','linewidth',4)
>> legend('f(x)','Polinomio interpolador','location','south')
```



Lógicamente, si se amplía el intervalo, esa buena aproximación se pierde:

```
>> xx=linspace(0,4); plot(xx,f(xx),'linewidth',2)
>> yy=polyval(p,xx); hold on, plot(xx,yy,'r:','linewidth',4)
>> legend('f(x)','Polinomio interpolador','location','south')
```



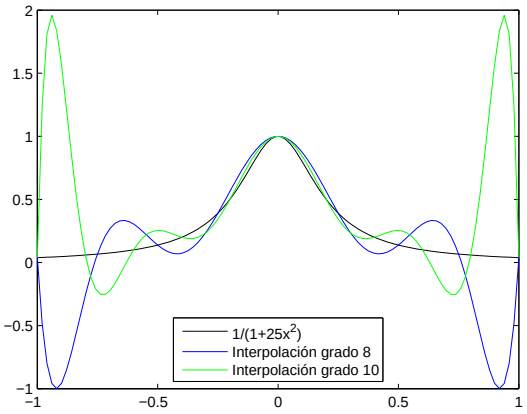
Aunque parece razonable construir el polinomio de interpolación a partir de nodos equiespaciados, no siempre es la mejor solución para obtener buenas aproximaciones. Por ejemplo, si intentamos aproximar, en $[-1, 1]$, la función

$$f(x) = \frac{1}{1 + 25x^2}$$

utilizando polinomios interpoladores de grados 8 y 10, resulta

```
>> f=@(x) 1./(1+25*x.^2); x=linspace(-1,1,9); y=f(x); L=lagrange(x); p_8=y*L;
>> x=linspace(-1,1,11); y=f(x); L=lagrange(x); p_10=y*L;
>> xx=linspace(-1,1); plot(xx,f(xx),'k')
```

```
>> hold on, yy8=polyval(p_8,xx); plot(xx,yy8,'b')
>> yy10=polyval(p_10,xx); plot(xx,yy10,'g')
>> legend('1/(1+25x^2)','Interpolación grado 8',...
    'Interpolación grado 10','location','south')
```



y se observa que, en los extremos del intervalo, la aproximación empeora, al aumentar el número de nodos.

Una forma de corregir esto es utilizar los nodos de Chebyshev, que son los que minimizan el error cometido, al hacer este tipo de aproximaciones en el intervalo $[-1,1]$. Así, en el caso de utilizar n puntos, en $[-1,1]$, en lugar de tomarlos equiespaciados, se deben tomar los nodos de Chebyshev, que se pueden hallar como:

$$x_i^{(n)} = \cos\frac{(2i-1)\pi}{2n}\ ,\ i = 1, 2, \ldots, n$$

En el caso de que se desee aproximar una función definida en el intervalo $[a,b]$, basta tener en cuenta que el cambio

$$x = \frac{a+b}{2} + \frac{b-a}{2}t$$

hace corresponder cada $t \in [-1,1]$ con $x \in [a,b]$, con lo que la forma de elegir n nodos, para interpolar una función en $[a,b]$, pasaría a ser:

$$x_i^{(n)} = \frac{a+b}{2} + \frac{b-a}{2}\cos\frac{(2i-1)\pi}{2n}\ ,\ i = 1, 2, \ldots, n$$

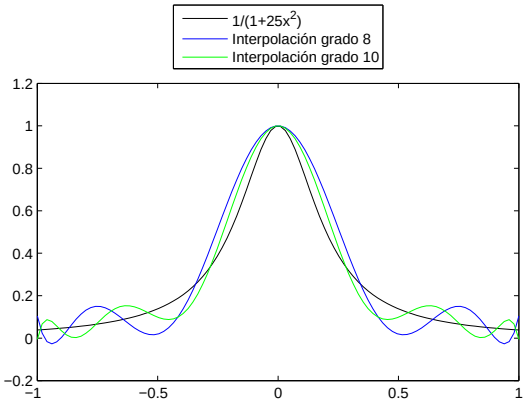
Ejemplo 7.3 *Aproxímese, en* $[-1,1]$*, la función*

$$f(x) = \frac{1}{1+25x^2}$$

utilizando polinomios interpoladores de grados 8 y 10, construidos a partir de los nodos de Chebyshev, y represéntense gráficamente los resultados obtenidos.

Solución: Teniendo en cuenta que debemos utilizar 9 y 11 nodos, respectivamente, hacemos

```
>> f=@(x) 1./(1+25*x.^2)
f =
    @(x)1./(1+25*x.^2)
>> n=9; i=1:n; x=cos((2*i-1)*pi/(2*n)), y=f(x)
x =
    0.9848    0.8660    0.6428    0.3420    0.0000   -0.3420   -0.6428
   -0.8660   -0.9848
y =
    0.0396    0.0506    0.0883    0.2548    1.0000    0.2548    0.0883
    0.0506    0.0396
>> L=lagrange(x); p_8=y*L
p_8 =
   17.6203   -0.0000  -40.3504    0.0000   31.3482   -0.0000   -9.5134
    0.0000    1.0000
>> n=11; i=1:n; x=cos((2*i-1)*pi/(2*n)), y=f(x)
x =
    0.9898    0.9096    0.7557    0.5406    0.2817    0.0000   -0.2817
   -0.5406   -0.7557   -0.9096   -0.9898
y =
    0.0392    0.0461    0.0654    0.1204    0.3351    1.0000    0.3351
    0.1204    0.0654    0.0461    0.0392
>> L=lagrange(x); p_10=y*L
p_10 =
  -46.6329    0.0000  130.1058   -0.0000 -133.4448   -0.0000   61.4430
   -0.0000  -12.4765    0.0000    1.0000
>> xx=linspace(-1,1); plot(xx,f(xx),'k')
>> hold on, yy8=polyval(p_8,xx); plot(xx,yy8,'b')
>> yy10=polyval(p_10,xx); plot(xx,yy10,'g')
>> legend('1/(1+25x^2)','Interpolación grado 8',...
    'Interpolación grado 10','location','northoutside')
```



8. Práctica 8

Para finalizar con la aproximación de funciones, en esta práctica, estudiaremos la *interpolación polinómica a trozos* y el *ajuste de datos por mínimos cuadrados*.

8.1. Interpolación polinómica a trozos

Consiste en, dados los datos

x	x_1	x_2	$\dots$	x_{n+1}
y	y_1	y_2	$\cdots$	y_{n+1}

donde supondremos $x_1 < x_2 < \cdots < x_{n+1}$, calcular una función que pase por los puntos (x_i, y_i), de manera que, en cada subintervalo $[x_i, x_{i+1}]$, la función sea un polinomio de grado m.

Las aproximaciones más habituales son la interpolación segmentada lineal, también llamada interpolación lineal a trozos, caso en el que $m = 1$, y el spline cúbico, que resulta al elegir $m = 3$. En MATLAB, esto se se resuelve con las órdenes que se describen a continuación.

`interp1(x,y,a)`: Halla el valor que toma en el punto `a`, la interpolación lineal a trozos correspondiente a los datos definidos por los vectores `x` e `y`.

`spline(x,y,a)`: Halla el valor que toma en el punto `a`, el spline cúbico correspondiente a los datos definidos por los vectores `x` e `y`.

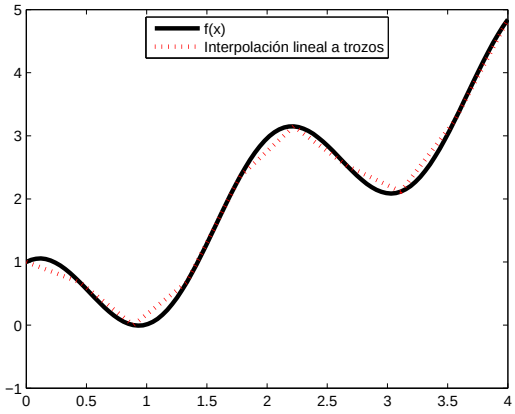
Ejemplo 8.1 *Dada la función $f(x) = x + \cos 3x$, constrúyanse 10 puntos x_i igualmente espaciados en el intervalo $[0, 4]$ y represéntense, en dicho intervalo, la función $f(x)$ junto con las aproximación obtenida por interpolación lineal a trozos correspondiente a los datos $x = (x_i)$, $y = f(x_i)$. A continuación, hállense los errores que se cometen con la aproximación anterior en $x = 1$, $x = 2$, y $x = 3$.*

Solución: Introducimos los datos con

```
>> f=@(x) x+cos(3*x)
f =
    @(x)x+cos(3*x)
>> x=linspace(0,4,10)
x =
         0    0.4444    0.8889    1.3333    1.7778    2.2222    2.6667
    3.1111    3.5556    4.0000
>> y=f(x)
y =
    1.0000    0.6797   -0.0004    0.6797    2.3596    3.1496    2.5212
    2.1153    3.2325    4.8439
```

y representamos gráficamente con las órdenes:

```
>> xx=linspace(0,4); plot(xx,f(xx),'k','linewidth',3)
>> hold on, plot(x,y,'r:','linewidth',3)
>> legend('f(x)','Interpolación lineal a trozos','location','north')
```



Por último, calculamos los errores pedidos:

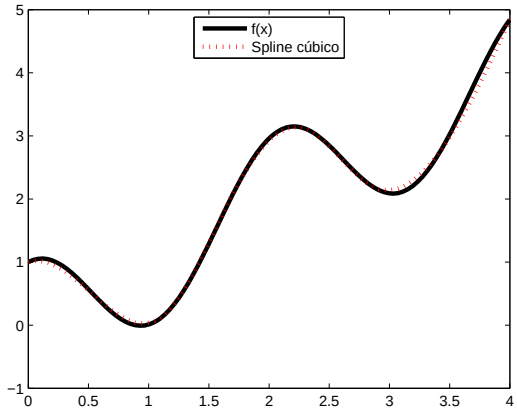
```
>> xxx=1:3
xxx =
     1     2     3
>> yyy=interp1(x,y,xxx)
yyy =
    0.1696    2.7546    2.2168
>> errores=abs(yyy-f(xxx))
errores =
    0.1596    0.2056    0.1279
```

Ejemplo 8.2 *Repítase el ejercicio anterior, dividiendo el intervalo en* 8 *puntos equiespaciados y utilizando el spline cúbico para aproximar la función, en lugar de la interpolación segmentada lineal.*

Solución:

```
>> x=linspace(0,4,8), y=f(x)
x =
         0    0.5714    1.1429    1.7143    2.2857    2.8571    3.4286
    4.0000
y =
    1.0000    0.4284    0.1838    2.1316    3.1255    2.1997    2.7768
    4.8439
```

```
>> plot(xx,f(xx),'k','linewidth',3), hold on
>> yy=spline(x,y,xx); plot(xx,yy,'r:','linewidth',3)
>> legend('f(x)','Spline cúbico','location','north')
```




```
>> yyy=spline(x,y,xxx)
yyy =
    0.0357    2.9224    2.1396
>> errores_spline=abs(yyy-f(xxx))
errores_spline =
    0.0257    0.0378    0.0507
```

8.2. Ajuste de datos por mínimos cuadrados

Dado un número finito de puntos $(x_1, y_1),\ (x_2, y_2), \ldots,\ (x_m, y_m)$, se pretende hallar la función $y = f(x)$, dentro de un conjunto de funciones prefijado, de forma que, al hacer la aproximación $y_i \approx f(x_i)$, se cometa el menor error posible. El método de mínimos cuadrados se basa en elegir f de manera que

$$\sum_{i=1}^{m} [y_i - f(x_i)]^2$$

tome el valor mínimo.

Cuando la función se elige entre los polinomios de grado k, se obtiene el llamado *polinomio de regresión de grado k*.

Con MATLAB, este problema se resuelve con la orden que se describe seguidamente.

`polyfit(x,y,k)`: Halla el polinomio de regresión de grado `k`, que ajusta, en el sentido de mínimos cuadrados, los datos definidos por los vectores `x` e `y`.

Ejemplo 8.3 *Ajústense, mediante recta de regresión, los datos:*

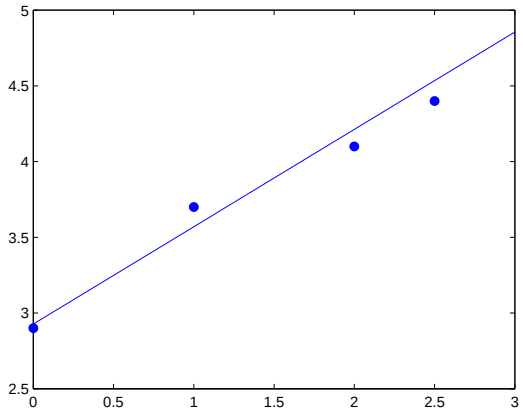
x	0	1	2	2.5	3
y	2.9	3.7	4.1	4.4	5

Solución: Basta hacer

```
>> x=[0 1 2 2.5 3]; y=[2.9 3.7 4.1 4.4 5.0]; p=polyfit(x,y,1)
p =
    0.6431    2.9267
```

y podemos comprobar el ajuste con la gráfica:

```
>> plot(x,y,'.','markersize',20), hold on
>> xx=linspace(0,3); yy=polyval(p,xx); plot(xx,yy)
```



Ejemplo 8.4 *Determínense a y b, para ajustar, por medio de $y = ae^{bx}$, los datos:*

x	1.2	2.8	4.3	5.4	6.8	7.9
y	7.5	16.1	38.9	67	146.6	266.2

Solución: Tomando neperianos en $y = ae^{bx}$, resulta $\ln y = \ln a + bx$, con lo que, denotando $Y = \ln y$, $A = b$ y $B = \ln a$, resulta $Y = Ax + B$ y el problema se convierte en un ajuste por una recta. Por tanto, hacemos

```
>> x=[1.2 2.8 4.3 5.4 6.8 7.9]; y=[7.5 16.1 38.9 67 146.6 266.2]; Y=log(y);
>> p=polyfit(x,Y,1)
p =
    0.5366    1.3321
```

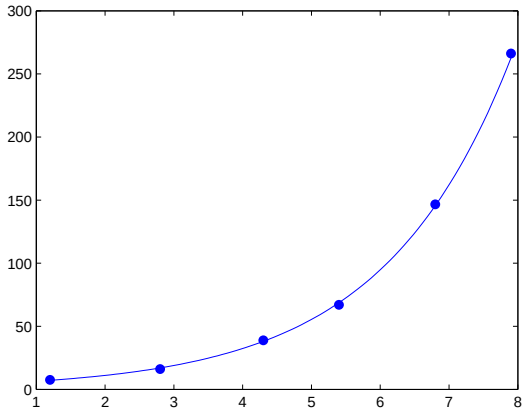
con lo que la función f que ajusta los datos se construye definiendo:

```
>> A=p(1), B=p(2), a=exp(B), b=A, f=@(x) a*exp(b*x)
A =
    0.5366
B =
    1.3321
```

```
a =
    3.7889
b =
    0.5366
f =
    @(x)a*exp(b*x)
```

Por último, comprobamos el ajuste obtenido:

```
>> plot(x,y,'.','markersize',20), hold on
>> xx=linspace(1.2,7.9); plot(xx,f(xx))
```



Ejemplo 8.5 *La densidad relativa d del aire depende de la altura h, habiéndose obtenido las siguientes mediciones:*

h (en km)	0	1.525	3.05	4.575	6.1	7.625	9.15
d (en kg/m^3)	1	0.8617	0.7385	0.6292	0.5328	0.4481	0.3741

Ajústense los datos anteriores por una parábola de regresión y estímese la densidad del aire a 5 kilómetros de altitud.

Solución: Introducimos los datos con

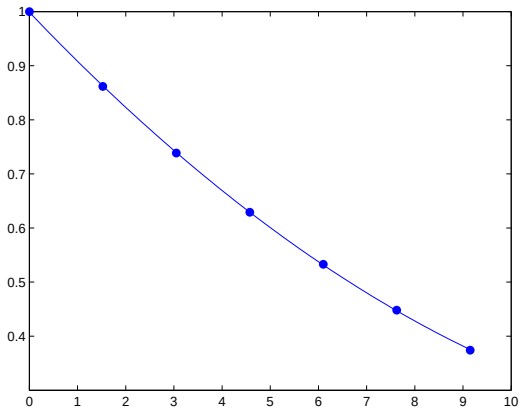
```
>> h=[0 1.525 3.05 4.575 6.1 7.625 9.15];
>> d=[1 0.8617 0.7385 0.6292 0.5328 0.4481 0.3741];
```

calculamos la parábola de regresión, ejecutando

```
>> p=polyfit(h,d,2)
p =
    0.0028   -0.0934    0.9989
```

comprobamos el ajuste, mediante la gráfica

```
>> plot(h,d,'.','markersize',20), hold on
>> hh=linspace(0,9.15); dd=polyval(p,hh); plot(hh,dd)
```



y, por último, estimamos la densidad para $h = 5$:

```
>> polyval(p,5)
ans =
    0.6007
```

Nota: Si se desea ajustar, utilizando mínimos cuadrados, un conjunto de $n + 1$ puntos por un polinomio de grado n, obtendremos el polinomio de interpolación, ya que, como éste pasa por todos los puntos, la suma de errores al cuadrado será nula y, evidentemente, ése es el valor mínimo posible.

Comprobémoslo, volviendo sobre el primer ejemplo de la práctica anterior. Allí, habíamos hecho

```
>> x=0:5; y=[1.1 1.5 2.4 2 3 1];
>> L=lagrange(x);
```

y obtenido el polinomio interpolador como:

```
>> p=y*L
p =
   -0.0967    1.1542   -4.8083    8.0458   -3.8950    1.1000
```

Pues bien, ejecutando

```
>> q=polyfit(x,y,5)
q =
   -0.0967    1.1542   -4.8083    8.0458   -3.8950    1.1000
```

se observa que obtenemos idéntico resultado.

9. Práctica 9

En esta práctica, tratamos de calcular la integral definida de una función, aproximándola por la integral de un polinomio que interpola a la función, en el intervalo de integración, en un número determinado de puntos. En función del número de puntos que se elijan y de la forma de elegirlos, iremos obteniendo distintas reglas de cuadratura, como veremos a continuación.

9.1. Regla del trapecio

Una de las reglas más simples se obtiene al hacer

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx$$

siendo $p(x)$ el polinomio que interpola a la función en los extremos a y b del intervalo de integración.

Gráficamente, esto supone calcular el área del trapecio formado por la recta que une los puntos $(a, f(a))$ y $(b, f(b))$, junto con el eje OX y las rectas $x = a$ y $x = b$, con lo cual:

$$\int_a^b f(x)dx \approx \frac{b-a}{2}[f(a) + f(b)]$$

Además, denotando E a la diferencia entre el valor exacto de la integral y el valor aproximado, se demuestra que existe $c \in (a, b)$, tal que

$$E = -\frac{(b-a)^3}{12}f''(c) \implies |E| \leq \frac{(b-a)^3}{12}M, \text{ con } M = \text{máx}\{|f''(x)| : \ a < x < b\}$$

con lo que la fórmula es de grado de precisión 1, es decir, que es exacta para los polinomios de grado menor o igual que 1.

Para programarla, basta hacer:

```
function r=trapecio(f,a,b)
% Función r=trapecio(f,a,b) que aproxima la integral de la función
% numérica f, en el intervalo [a,b], utilizando la regla del trapecio
r=(b-a)*(f(a)+f(b))/2;
```

Ejemplo 9.1 *Calcúlense, utilizando la regla del trapecio,* $\int_0^2 (3x+4)dx$, $\int_0^\pi \text{sen}\,\frac{x}{4}dx$ *y los errores cometidos.*

Solución:

```
>> f=@(x) 3*x+4
f =
    @(x)3*x+4
>> integral_trapecio=trapecio(f,0,2)
integral_trapecio =
    14
```

```
>> syms x, f_sim=f(x)
f_sim =
3*x + 4
>> integral_exacta=int(f_sim,0,2)
integral_exacta =
14
>> error=double(abs(integral_exacta-integral_trapecio))
error =
     0
>> f=@(x) sin(x/4)
f =
    @(x)sin(x/4)
>> integral_trapecio=trapecio(f,0,pi)
integral_trapecio =
    1.1107
>> syms x, f_sim=f(x)
f_sim =
sin(x/4)
>> integral_exacta=int(f_sim,0,pi)
integral_exacta =
4 - 2*2^(1/2)
>> error=double(abs(integral_exacta-integral_trapecio))
error =
    0.0609
```

9.2. Regla de Simpson

En este caso, aproximamos la integral por la del polinomio que interpola a la función en los extremos a, b y el punto medio de ambos. Denotando por x_1 a dicho punto, planteamos el polinomio como $p(x) = \alpha(x - x_1)^2 + \beta(x - x_1) + \gamma$, y resulta

$$\int p(x)dx = \alpha\frac{(x - x_1)^3}{3} + \beta\frac{(x - x_1)^2}{2} + \gamma x$$

con lo que, si hacemos $h = \dfrac{b-a}{2}$, teniendo en cuenta que $b - x_1 = h$ y que $a - x_1 = -h$, hallamos:

$$\int_a^b p(x)dx = \frac{\alpha}{3}[h^3 - (-h)^3] + \frac{\beta}{2}[h^2 - (-h)^2] + \gamma(b - a) = \frac{2\alpha h^3}{3} + 2\gamma h = \frac{h}{3}(2\alpha h^2 + 6\gamma)$$

Por otra parte, de $p(a) = f(a),\ p(x_1) = f(x_1),\ p(b) = f(b)$, se deduce

$$\alpha h^2 - \beta h + \gamma = f(a),\ \gamma = f(x_1),\ \alpha h^2 + \beta h + \gamma = f(b)$$

luego, sumando la primera igualdad y la última,

$$2\alpha h^2 + 2\gamma = f(a) + f(b) \implies 2\alpha h^2 = f(a) + f(b) - 2f(x_1)$$

con lo cual:

$$\int_a^b p(x)dx = \frac{h}{3}[f(a)+f(b)-2f(x_1)+6f(x_1)] \implies$$
$$\implies \int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right]$$

Además, si E es la diferencia entre el valor exacto de la integral y el valor aproximado, es posible probar que existe $c \in (a,b)$, tal que

$$E = -\frac{(b-a)^5}{2880}f^{(iv)}(c) \implies |E| \leq \frac{(b-a)^5}{2880}M, \text{ con } M = \max\{|f^{(iv)}(x)| : \ a < x < b\}$$

luego, la fórmula es de grado de precisión 3, con lo que es exacta para los polinomios de grado menor o igual que 3, y podemos programarla de la siguiente forma:

```
function r=simpson(f,a,b)
% Función r=simpson(f,a,b) que aproxima la integral de la función
% numérica f, en el intervalo [a,b], utilizando la regla de Simpson
r=(b-a)*(f(a)+4*f((a+b)/2)+f(b))/6;
```

Ejemplo 9.2 *Calcúlense, utilizando la regla de Simpson,* $\int_0^2 (x^3+x^2+2x-1)dx$, $\int_0^\pi \text{sen}\,\frac{x}{4}dx$ *y los errores cometidos.*

Solución:

```
>> f=@(x) x^3+x^2+2*x-1
f =
    @(x)x^3+x^2+2*x-1
>> integral_simpson=simpson(f,0,2)
integral_simpson =
    8.6667
>> syms x, f_sim=f(x)
f_sim =
x^3 + x^2 + 2*x - 1
>> integral_exacta=int(f_sim,0,2)
integral_exacta =
26/3
>> error=double(abs(integral_exacta-integral_simpson))
error =
     0
>> f=@(x) sin(x/4)
f =
    @(x)sin(x/4)
>> integral_simpson=simpson(f,0,pi)
integral_simpson =
    1.1717
```

```
>> syms x, f_sim=f(x)
f_sim =
sin(x/4)
>> integral_exacta=int(f_sim,0,pi)
integral_exacta =
4 - 2*2^(1/2)
>> error=double(abs(integral_exacta-integral_simpson))
error =
  1.5768e-004
```

9.3. Regla del trapecio compuesta

Se basa en dividir el intervalo de integración $[a, b]$ en n subintervalos de igual longitud. Para ello, se denota

$$x_0 = a,\ x_1 = a + h,\ x_2 = a + 2h,\ \ldots\ ,\ x_n = a + nh = b,\ \text{con } h = \frac{b-a}{n}$$

y se hace

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx$$

resolviendo las integrales anteriores, aplicando la regla del trapecio.

Además, denotando E a la diferencia entre el valor exacto de la integral y el valor aproximado, se demuestra que existe $c \in (a, b)$, tal que:

$$E = -\frac{(b-a)^3}{12n^2} f''(c) \implies |E| \leq \frac{(b-a)^3}{12n^2} M,\ \text{con } M = \text{máx}\{|f''(x)| :\ a < x < b\}$$

La programación de este método se puede hacer de la siguiente forma:

```
function r=trapecio_compuesta(f,a,b,n)
% Función r=trapecio_compuesta(f,a,b,n) que aproxima la integral de la
% función numérica f, en [a,b], utilizando la regla del trapecio
% compuesta, dividiendo [a,b] en n subintervalos de igual longitud
x=linspace(a,b,n+1); s=zeros(1,n);
for i=1:n
    s(i)=trapecio(f,x(i),x(i+1));
end
r=sum(s);
```

Ejemplo 9.3 *Dada* $\displaystyle\int_0^3 x^2 \cos 2x dx$,

a) Calcúlese por la regla del trapecio compuesta, dividiendo el intervalo de integración en 20 *subintervalos, y hállese el error cometido.*

b) Obténgase el número de subintervalos que deberían utilizarse para que el error sea menor que una milésima y compruébese el resultado obtenido.

Solución:

Apartado a): Calculamos la integral y el error cometido, haciendo

```
>> f=@(x) x^2*cos(2*x)
f =
    @(x)x^2*cos(2*x)
>> integral_trapecio_compuesta=trapecio_compuesta(f,0,3,20)
integral_trapecio_compuesta =
    0.2730
>> syms x, f_sim=f(x)
f_sim =
x^2*cos(2*x)
>> integral_exacta=int(f_sim,0,3)
integral_exacta =
(3*cos(6))/2 + (17*sin(6))/4
>> error=double(abs(integral_exacta-integral_trapecio_compuesta))
error =
    0.0203
```

Apartado b): El error está acotado por $\dfrac{(b-a)^3}{12n^2}M$, con $M = \text{máx}\{|f''(x)| : \ a < x < b\}$, con lo cual, si calculamos n de forma que

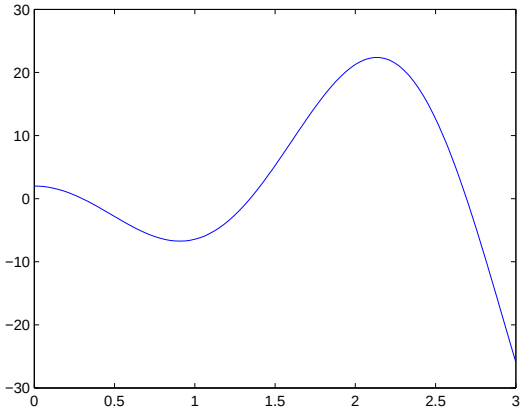
$$\frac{(b-a)^3}{12n^2}M = \varepsilon$$

tendremos garantizado que el error sea menor que ε, por lo que n deberá cumplir:

$$\frac{(b-a)^3 M}{12\varepsilon} = n^2 \implies n = \sqrt{\frac{(b-a)^3 M}{12\varepsilon}}$$

Por tanto, empezaremos por calcular M, para lo que vamos a denotar $f_2(x) = f''(x)$, con lo que deberemos hallar el máximo del valor absoluto de $f_2(x)$ en el intervalo $[0, 3]$. Así, hacemos

```
>> f2=diff(f_sim,2), xx=linspace(0,3); yy=subs(f2,x,xx); plot(xx,yy)
f2 =
2*cos(2*x) - 8*x*sin(2*x) - 4*x^2*cos(2*x)
```



y, a continuación, calculamos el máximo relativo x_0 que se puede observar en la gráfica. Como $f_2'(x)$ debe anularse en ese punto, podemos hallarlo por medio de

```
>> f2_der=diff(f2)
f2_der =
8*x^2*sin(2*x) - 24*x*cos(2*x) - 12*sin(2*x)
>> f2_der_num=matlabFunction(f2_der)
f2_der_num =
    @(x)sin(x.*2.0).*-1.2e1-x.*cos(x.*2.0).*2.4e1+x.^2.*sin(x.*2.0).*8.0
>> x0=fzero(f2_der_num,2)
x0 =
    2.1337
```

y como, a la vista de la gráfica, se plantea la duda de si $|f_2(x)|$ alcanza su máximo en x_0 o en 3, calculamos

```
>> abs(subs(f2,x,[x0,3]))
ans =
   22.3851   25.9398
```

con lo que queda claro que

```
>> M=abs(subs(f2,x,3))
M =
   25.9398
```

con lo cual:

```
>> a=0; b=3; e=1e-3; n=ceil(sqrt((b-a)^3*M/12/e))
n =
   242
```

En definitiva, debemos dividir el intervalo de integración en 242 subintervalos, hallando la integral, como

```
>> integral_trapecio_compuesta=trapecio_compuesta(f,0,3,n)
integral_trapecio_compuesta =
    0.2529
```

y, finalmente, comprobamos que el error es menor que 10^{-3}, ejecutando:

```
>> error=double(abs(integral_exacta-integral_trapecio_compuesta))
error =
  1.3819e-004
```

9.4. Regla de Simpson compuesta

Consiste en dividir el intervalo de integración $[a, b]$ en n subintervalos de igual longitud. Para ello, al igual que en la regla del trapecio compuesta, se denota

$$x_0 = a,\ x_1 = a + h,\ x_2 = a + 2h,\ \ldots\ ,\ x_n = a + nh = b,\ \text{con}\ h = \frac{b-a}{n}$$

y se hace

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx$$

resolviendo las integrales anteriores, aplicando la regla de Simpson.

Además, si E es la diferencia entre el valor exacto de la integral y el valor aproximado, es posible probar que existe $c \in (a, b)$, tal que:

$$E = -\frac{(b-a)^5}{2880n^4} f^{(iv)}(c) \implies |E| \leq \frac{(b-a)^5}{2880n^4} M,\ \text{con}\ M = \text{máx}\{|f^{(iv)}(x)| :\ a < x < b\}$$

Este método se puede programar de la siguiente manera:

```
function r=simpson_compuesta(f,a,b,n)
% Función r=simpson_compuesta(f,a,b,n) que aproxima la integral de la
% función numérica f, en [a,b], utilizando la regla de Simpson compuesta,
% dividiendo [a,b] en n subintervalos de igual longitud
x=linspace(a,b,n+1); s=zeros(1,n);
for i=1:n
    s(i)=simpson(f,x(i),x(i+1));
end
r=sum(s);
```

Ejemplo 9.4 *Calcúlese* $\int_0^{\pi} x^2 \operatorname{sen} 5x dx$*, por la regla de Simpson compuesta, dividiendo el intervalo de integración en* 10 *subintervalos, y hállese el error cometido.*

Solución: Hallamos la integral y el error cometido, haciendo

```
>> f=@(x) x^2*sin(5*x)
f =
    @(x)x^2*sin(5*x)
>> integral_simpson_compuesta=simpson_compuesta(f,0,pi,10)
integral_simpson_compuesta =
    1.9462
>> syms x, f_sim=f(x)
f_sim =
x^2*sin(5*x)
>> integral_exacta=int(f_sim,0,pi)
integral_exacta =
pi^2/5 - 4/125
```

```
>> error=double(abs(integral_exacta-integral_simpson_compuesta))
error =
    0.0042
```

9.5. Cuadratura gaussiana

Los polinomios de Legendre se pueden construir como

$$L_0(x) = 1,\ L_1(x) = x,\ L_n(x) = \frac{2n-1}{n} x L_{n-1}(x) - \frac{n-1}{n} L_{n-2}(x),\ n = 2, 3, \ldots$$

o también por medio de la siguiente expresión:

$$L_n(x) = \frac{1}{2^n \cdot n!} \frac{d^n}{dx^n}[(x^2-1)^n]$$

Pues bien, al hacer la aproximación

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} p(x)dx$$

donde $p(x)$ es un polinomio que interpola a $f(x)$ en n nodos elegidos en el intervalo $[-1, 1]$, para que el error sea el menor posible, los nodos, en lugar de equiespaciados, deben elegirse como las raíces del polinomio de Legendre de grado n.

Cambio de intervalo:

En el caso de que se desee hallar $\displaystyle\int_a^b f(x)dx$, basta tener en cuenta que, al hacer el cambio de variable

$$x = \frac{a+b}{2} + \frac{b-a}{2} t,\ \text{o equivalentemente, } t = \frac{2x-a-b}{b-a}$$

resulta

$$\int_a^b f(x)dx = \int_{-1}^{1} f\left(\frac{a+b}{2} + \frac{b-a}{2}t\right)\frac{b-a}{2}dt = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{a+b}{2} + \frac{b-a}{2}x\right)dx$$

con lo que basta definir

$$g(x) = f\left(\frac{a+b}{2} + \frac{b-a}{2}x\right)$$

para obtener

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} g(x)dx$$

que ya es un problema en el intervalo $[-1, 1]$, al que podemos aplicar la técnica anterior.

A continuación, vamos a ver un ejemplo de utilización de cuadratura gaussiana; en él, utilizaremos dos comandos relativos a polinomios, cuya descripción es la siguiente:

`poly2sym(p)`: Convierte el polinomio `p`, escrito en formato vectorial, a función simbólica.

`sym2poly(p)`: Convierte el polinomio `p`, escrito como función simbólica, al formato vectorial.

Así, podemos hacer:

```
>> p=[1 2 3]
p =
     1     2     3
>> p_sim=poly2sym(p)
p_sim =
x^2 + 2*x + 3
>> syms x, q_sim=x^3-x+2
q_sim =
x^3 - x + 2
>> q=sym2poly(q_sim)
q =
     1     0    -1     2
```

Ejemplo 9.5 *Aproxímese* $\int_{-1}^{1} xe^x dx$*, utilizando cuadratura gaussiana con* 4 *nodos, y hállese el error cometido.*

Solución:

```
>> f=@(x) x.*exp(x)
f =
    @(x)x.*exp(x)
>> syms x, L4_sim=diff((x^2-1)^4,4)/2^4/factorial(4)
L4_sim =
3*x^2*(x^2 - 1) + (3*(x^2 - 1)^2)/8 + x^4
>> L4=sym2poly(L4_sim)
L4 =
    4.3750         0   -3.7500         0    0.3750
>> xx=roots(L4)
xx =
   -0.8611
    0.8611
   -0.3400
    0.3400
>> yy=f(xx)
yy =
   -0.3640
    2.0373
   -0.2420
    0.4776
```

```
>> p=polyfit(xx,yy,3)
p =
    0.5366    1.1484    0.9963   -0.0149
>> integral_gaussiana=int(poly2sym(p),-1,1)
integral_gaussiana =
106033687349875207/144115188075855872
>> f_sim=f(x)
f_sim =
x*exp(x)
>> integral_exacta=int(f_sim,-1,1)
integral_exacta =
2/exp(1)
>> error=double(abs(integral_exacta-integral_gaussiana))
error =
  2.3756e-006
```

9.6. Comandos de MATLAB

MATLAB dispone de funciones propias para el cálculo numérico de integrales, de entre las que destacaremos **quad** y **trapz**. Ambas órdenes sirven para aproximar

$$\int_a^b f(x)dx$$

y su funcionamiento es el siguiente:

`quad(f,a,b)`: Aproxima la integral, mediante una variante de la regla de Simpson compuesta, siendo `f` una función numérica, construida de forma que opere elemento a elemento.

`trapz(x,y)`: Aproxima la integral, utilizando la regla del trapecio compuesta. La variable `x` debe contener los nodos a utilizar y la variable `y`, los valores de la función en dichos nodos.

Ejemplo 9.6 *Aproxímese* $\int_0^3 x^2 \cos 2x dx$*, utilizando la orden* **quad** *y la regla del trapecio compuesta, por medio de* **trapz**, *dividiendo el intervalo de integración en* 20 *subintervalos de igual longitud, y hállense los errores cometidos.*

Solución:

```
>> f=@(x) x.^2.*cos(2*x)
f =
    @(x)x.^2.*cos(2*x)
>> integral_quad=quad(f,0,3)
integral_quad =
    0.2527
```

```
>> xx=linspace(0,3,21);
>> yy=f(xx);
>> integral_trapz=trapz(xx,yy)
integral_trapz =
    0.2730
>> syms x, f_sim=f(x)
f_sim =
x^2*cos(2*x)
>> integral_exacta=int(f_sim,0,3)
integral_exacta =
(3*cos(6))/2 + (17*sin(6))/4
>> error_quad=double(abs(integral_exacta-integral_quad))
error_quad =
  1.1378e-008
>> error_trapz=double(abs(integral_exacta-integral_trapz))
error_trapz =
    0.0203
```

10. Práctica 10

Esta práctica está dedicada a la resolución numérica de problemas de valores iniciales relativos a ecuaciones diferenciales, aunque, antes de abordar el problema en sí, para poder comparar la solución aproximada obtenida, con la solución exacta del problema, comenzaremos por ver cómo se resuelve, en MATLAB, de forma simbólica, un sistema de ecuaciones diferenciales.

10.1. Resolución simbólica de ecuaciones diferenciales

Para resolver un sistema de ecuaciones diferenciales, se utiliza el comando **dsolve**:

Con `dsolve(ecuación1,ecuación2,...,condición1,condición2,...,variable)`, se calculan las incógnitas (en orden alfabético) que verifican las ecuaciones y las condiciones especificadas, en función de la variable independiente que se haya establecido en el último argumento de **dsolve**, teniendo en cuenta lo siguiente:

- Todos los argumentos deben ir expresados como cadenas de texto.
- Tanto en las ecuaciones como en las condiciones, y' se expresa como `Dy`, y'' como `D2y`, etc.
- MATLAB, por defecto, considera que la variable independiente es `t`, salvo que se especifique otra.

Ejemplo 10.1 *Resuélvase el problema de valores iniciales:*

$$y'' - 4y' + 3y = 9x^2 + 4,\ y(0) = 6,\ y'(0) = 8$$

Solución: Hacemos

```
>> y=dsolve('D2y-4*Dy+3*y=9*x^2+4','y(0)=6','Dy(0)=8','x')
y =
8*x + 2*exp(3*x) - 6*exp(x) + 3*x^2 + 10
```

y obtenemos la solución del problema, de forma simbólica, que siempre podríamos convertir a la forma anónima, utilizando el comando **matlabFunction**.

Ejemplo 10.2 *Obténgase la solución del sistema*

$$\begin{cases} y_1' = 2y_1 - y_2 - e^x \\ y_2' = 2y_1 - y_2 + e^x \end{cases}$$

que verifica las condiciones iniciales $y_1(0) = 1$, $y_2(0) = 0$.

Solución:

```
>> ed1='Dy1=2*y1-y2-exp(x)', ed2='Dy2=2*y1-y2+exp(x)'
ed1 =
Dy1=2*y1-y2-exp(x)
ed2 =
Dy2=2*y1-y2+exp(x)
>> [y1 y2]=dsolve(ed1,ed2,'y1(0)=1','y2(0)=0','x')
y1 =
4*exp(x) - 3*x*exp(x) - 3
y2 =
6*exp(x) - 3*x*exp(x) - 6
```

10.2. Resolución numérica de problemas de valores iniciales

En esta sección, vamos a calcular soluciones aproximadas de problemas de condiciones iniciales relativos, tanto a sistemas de ecuaciones diferenciales de primer orden como a ecuaciones diferenciales de orden superior. Comencemos por plantear el problema en el caso de sistemas:

Se trata de hallar las funciones $y_k(t)$, $k = 1, 2, \ldots n$, que verifican las ecuaciones diferenciales

$$\begin{cases} y_1' = f_1(t, y_1, y_2, \ldots, y_n) \\ y_2' = f_2(t, y_1, y_2, \ldots, y_n) \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ y_n' = f_n(t, y_1, y_2, \ldots, y_n) \end{cases}$$

y las condiciones iniciales:

$$y_1(t_0) = c_1,\ y_2(t_0) = c_2,\ \ldots,\ y_n(t_0) = c_n$$

Plantearemos el problema en forma vectorial, para lo que denotaremos

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix},\ \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} f_1(t, \mathbf{y}) \\ f_2(t, \mathbf{y}) \\ \vdots \\ f_n(t, \mathbf{y}) \end{pmatrix},\ \mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$

con lo que el problema pasa a ser: $\mathbf{y}' = \mathbf{f}(t, \mathbf{y}),\ \mathbf{y}(t_0) = \mathbf{c}$.

En el caso de que el problema se refiera a una ecuación de orden n, tendremos que hallar $y(t)$, de forma que se verifique la ecuación diferencial

$$y^{(n)} = f(t, y, y', \ldots, y^{(n-1)})$$

junto con las condiciones iniciales:

$$y(t_0) = c_0,\ y'(t_0) = c_1,\ \ldots,\ y^{(n-1)}(t_0) = c_{n-1}$$

Ahora bien, este problema siempre se puede transformar en otro relativo a un sistema, si denotamos $y_1 = y,\ y_2 = y',\ y_3 = y'', \ldots, y_n = y^{(n-1)}$, con lo que resulta el sistema equivalente

$$\begin{cases} y_1' &= y_2 \\ y_2' &= y_3 \\ & \vdots \\ y_{n-1}' &= y_n \\ y_n' &= f(t, y_1, y_2, \ldots, y_n) \end{cases}$$

con las condiciones iniciales

$$y_1(t_0) = c_0,\ y_2(t_0) = c_1,\ \ldots,\ y_n(t_0) = c_{n-1}$$

y, si resolvemos este problema, bastará tener en cuenta que $y = y_1$ para obtener la solución buscada.

Una vez visto que basta resolver $\mathbf{y}' = \mathbf{f}(t, \mathbf{y}),\ \mathbf{y}(t_0) = \mathbf{c}$, vamos a hacer el planteamiento numérico:

Partiendo del valor de $\mathbf{y}(t_0)$, pretendemos calcular el valor aproximado de la solución, en un número finito de puntos, que denotaremos $t_0, t_1, t_2 \ldots, t_N$ y supondremos en orden creciente, a los que se les suele llamar *nodos*.

Aunque no es imprescindible, por comodidad, supondremos que los nodos son equidistantes entre sí, con lo cual,

$$t_k = t_0 + kh,\ k = 0, 1, 2, \ldots, N, \text{ siendo } h = \frac{t_N - t_0}{N}$$

y utilizaremos la notación:

- $\mathbf{y}(t_k)$ es el valor de la solución exacta en t_k.
- $\mathbf{y}_k$ es la aproximación del valor de la solución en t_k.

10.2.1. Método de Euler

En este método, partiendo de $\mathbf{y}(t_0) = \mathbf{c}$, con lo que $\mathbf{y}_0 = \mathbf{c}$, se calculan $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$, por medio de:

$$\mathbf{y}_{j+1} = \mathbf{y}_j + h\mathbf{f}(t_j, \mathbf{y}_j),\ j = 0, 1, 2, \ldots, N-1$$

Una manera de programar el anterior algoritmo es crear el fichero de función euler.m con las órdenes:

```
function [t y]=euler(f,intervalo,c,h)
%      Función [t y]=euler(f,intervalo,c,h)
%      Método de Euler para resolver y'=f(t,y), y(t0)=c
% ARGUMENTOS DE ENTRADA:
%   f ......... Función numérica, en forma de vector columna,
%               que define el segundo miembro del sistema
%   intervalo .. Intervalo de cálculo, de la forma [t0 tN] o [t0,tN]
%   c ......... Vector columna de condiciones iniciales
%   h ......... Distancia entre los nodos
% ARGUMENTOS DE SALIDA:
%   t ......... Vector columna que contiene los nodos t0,t1,t2,...,tN
%   y ......... Solución aproximada en t: Es una matriz, cuyas columnas
%               corresponden a cada una de las funciones solución
t0=intervalo(1); tN=intervalo(2); t=(t0:h:tN).'; N=length(t);
y=zeros(N,length(c));
y(1,:)=c.';
for j=1:N-1
    y(j+1,:)=y(j,:)+h*f(t(j),y(j,:)).';
end
```

Ejemplo 10.3 *Aproxímese la solución del problema de valores iniciales* $y' = 2xy$, $y(0) = 1$, *en el intervalo* $[0, 1]$, *aplicando el método de Euler con distancia entre nodos de* 0.01 *y, a continuación, hállese la solución exacta del problema y calcúlese el error máximo cometido en los nodos utilizados. Finalmente, represéntense gráficamente ambas soluciones.*

Solución: Construimos la función que define la ecuación diferencial, haciendo

```
>> f=@(x,y) 2*x*y
f =
    @(x,y)2*x*y
```

y hallamos la solución aproximada, con:

```
>> [x y_euler]=euler(f,[0 1],1,0.01);
```

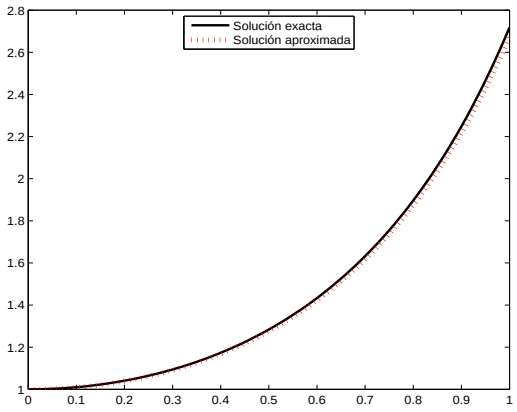
Seguidamente, hallamos la solución exacta y el error máximo

```
>> y=dsolve('Dy=2*x*y','y(0)=1','x')
y =
exp(x^2)
>> y_exacta=matlabFunction(y)
y_exacta =
    @(x)exp(x.^2)

>> error=max(abs(y_exacta(x)-y_euler))
error =
    0.0445
```

y, por último, hacemos las gráficas:

```
>> plot(x,y_exacta(x),'k','linewidth',2)
>> hold on, plot(x,y_euler,'r:','linewidth',2)
>> legend('Solución exacta','Solución aproximada','location','north')
```



Ejemplo 10.4 *Hállense las soluciones aproximadas del problema*

$$\begin{cases} y_1' = y_1 + 2y_2 \\ y_2' = -2y_1 + y_2 + 2e^t \end{cases} \qquad y_1(0) = y_2(0) = 1$$

en el intervalo $[0, 2]$*, utilizando el método de Euler con distancia entre nodos de* 0.01 *y, a continuación, calcúlense las soluciones exactas y los errores máximos cometidos en los nodos. Finalmente, represéntense gráficamente las soluciones exactas y las aproximadas.*

Solución: Hallamos, en primer lugar, las soluciones aproximadas, con

```
>> f=@(t,y) [y(1)+2*y(2);-2*y(1)+y(2)+2*exp(t)]
f =
    @(t,y)[y(1)+2*y(2);-2*y(1)+y(2)+2*exp(t)]
>> [t y_euler]=euler(f,[0 2],[1;1],0.01);
>> y1_euler=y_euler(:,1);
>> y2_euler=y_euler(:,2);
```

y, para calcular las soluciones exactas, hacemos

```
>> [y1 y2]=dsolve('Dy1=y1+2*y2','Dy2=-2*y1+y2+2*exp(t)','y1(0)=1','y2(0)=1')
y1 =
exp(t) + sin(2*t)*exp(t)
y2 =
cos(2*t)*exp(t)
>> y1_exacta=matlabFunction(y1)
y1_exacta =
    @(t)exp(t)+sin(t.*2.0).*exp(t)
```

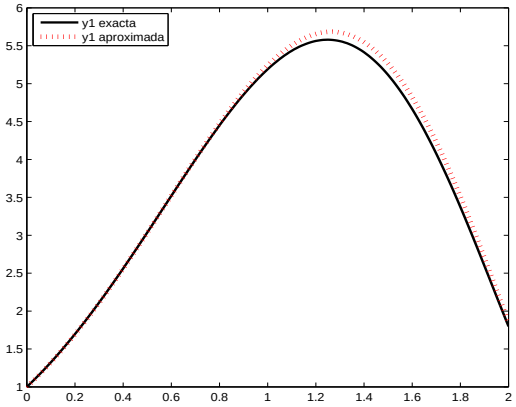
```
>> y2_exacta=matlabFunction(y2)
y2_exacta =
    @(t)cos(t.*2.0).*exp(t)
```

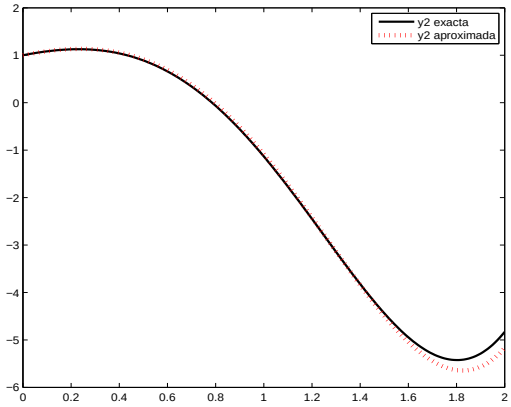
con lo que los errores son:

```
>> error_y1=max(abs(y1_exacta(t)-y1_euler))
error_y1 =
    0.1565
>> error_y2=max(abs(y2_exacta(t)-y2_euler))
error_y2 =
    0.3388
```

Por último, las gráficas resultan:

```
>> plot(t,y1_exacta(t),'k','linewidth',2)
>> hold on, plot(t,y1_euler,'r:','linewidth',2)
>> legend('y1 exacta','y1 aproximada','location','northwest')
>> plot(t,y2_exacta(t),'k','linewidth',2)
>> hold on, plot(t,y2_euler,'r:','linewidth',2)
>> legend('y2 exacta','y2 aproximada','location','northeast')
```





Ejemplo 10.5 *El siguiente problema describe las oscilaciones de una masa, actuando una fuerza periódica sobre ella, pendiente de un muelle colocado en posición vertical:*

$$\frac{d^2x}{dt^2} + 4\frac{dx}{dt} + 20x = 2\cos 2t,\ \ x(0) = 0.2,\ \ x'(0) = -1.2$$

Obténgase su solución aproximada, en el intervalo $[0, 6]$*, por el método de Euler, con distancia entre nodos de* 0.02 *y su solución exacta, así como el error máximo cometido en los nodos. Por último, represéntense ambas soluciones.*

Solución: Expresamos la ecuación como $x'' = -20x - 4x' + 2\cos 2t$ y denotamos $x_1 = x,\ x_2 = x'$, con lo que el problema se transforma en

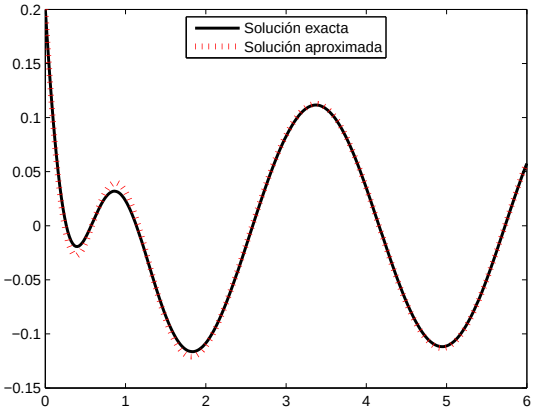
$$\begin{cases} x_1' = x_2 \\ x_2' = -20x_1 - 4x_2 + 2\cos 2t \end{cases} \qquad x_1(0) = 0.2,\ x_2(0) = -1.2$$

y hallamos su solución aproximada, ejecutando:

```
>> f=@(t,X) [X(2);-20*X(1)-4*X(2)+2*cos(2*t)]
f =
    @(t,X)[X(2);-20*X(1)-4*X(2)+2*cos(2*t)]
>> [t X_euler]=euler(f,[0 6],[0.2;-1.2],0.02);
>> x_euler=X_euler(:,1);
```

Para hallar la solución exacta, el error y las gráficas, hacemos:

```
>> x=simplify(dsolve('D2x+4*Dx+20*x=2*cos(2*t)','x(0)=0.2','Dx(0)=-1.2'))
x =
cos(2*t)/10+sin(2*t)/20+cos(4*t)/(10*exp(2*t))-(11*sin(4*t))/(40*exp(2*t))
>> x_exacta=matlabFunction(x);
>> error=max(abs(x_exacta(t)-x_euler))
error =
    0.0088
>> plot(t,x_exacta(t),'k','linewidth',2)
>> hold on, plot(t,x_euler,'r:','linewidth',2)
>> legend('Solución exacta','Solución aproximada','location','north')
```



10.2.2. Métodos Runge-Kutta

Aunque el método de Euler es muy sencillo, requiere que el paso (distancia entre nodos) sea pequeño para obtener una buena aproximación de la solución. Los *métodos Runge-Kutta* mejoran

la convergencia, sin que el número de operaciones crezca excesivamente, en relación al método de Euler, por lo que son ampliamente utilizados. A continuación, se describe el de orden dos.

Método Runge-Kutta de orden dos:

Con las mismas hipótesis sobre los nodos y utilizando la misma notación que en el método de Euler, partiendo de $\mathbf{y}_0$, se hallan $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$, por medio de:

$$\mathbf{y}_{j+1} = \mathbf{y}_j + \frac{h}{2}(K_1 + K_2),\ j = 0, 1, 2, \ldots, N-1, \text{ siendo}$$

$$K_1 = \mathbf{f}(t_j, \mathbf{y}_j)$$
$$K_2 = \mathbf{f}(t_j + h, \mathbf{y}_j + hK_1)$$

Una forma de programar este método es crear el fichero de función rungek2.m con las órdenes:

```
function [t y]=rungek2(f,intervalo,c,h)
%      Función [t y]=rungek2(f,intervalo,c,h)
%      Método Runge-Kutta de orden 2 para resolver y'=f(t,y), y(t0)=c
% ARGUMENTOS DE ENTRADA:
%   f ......... Función numérica, en forma de vector columna,
%               que define el segundo miembro del sistema
%   intervalo .. Intervalo de cálculo, de la forma [t0 tN] o [t0,tN]
%   c ......... Vector columna de condiciones iniciales
%   h ......... Distancia entre los nodos
% ARGUMENTOS DE SALIDA:
%   t ......... Vector columna que contiene los nodos t0,t1,t2,...,tN
%   y ......... Solución aproximada en t: Es una matriz, cuyas columnas
%               corresponden a cada una de las funciones solución
t0=intervalo(1); tN=intervalo(2); t=(t0:h:tN).'; N=length(t);
y=zeros(N,length(c));
y(1,:)=c.';
for j=1:N-1
    K1=f(t(j),y(j,:)).';
    K2=f(t(j)+h,y(j,:)+h*K1).';
    y(j+1,:)=y(j,:)+h/2*(K1+K2);
end
```

Ejemplo 10.6 *Resuélvase el problema planteado en el ejemplo* 10.4*, utilizando el método de Runge-Kutta de orden dos.*

Solución: Creamos el fichero `rungek2.m` con las órdenes y resolvemos el ejemplo 10.4, ejecutando

```
>> f=@(t,y) [y(1)+2*y(2);-2*y(1)+y(2)+2*exp(t)];
>> [t y_rungek2]=rungek2(f,[0 2],[1;1],0.01);
>> y1_rungek2=Y_rungek2(:,1);
>> y2_rungek2=Y_rungek2(:,2);
```

```
>> [y1 y2]=dsolve('Dy1=y1+2*y2','Dy2=-2*y1+y2+2*exp(t)','y1(0)=1','y2(0)=1');
>> y1_exacta=matlabFunction(y1);
>> y2_exacta=matlabFunction(y2);
>> error_y1=max(abs(y1_exacta(t)-y1_rungek2))
error_y1 =
    0.0020
>> error_y2=max(abs(y2_exacta(t)-y2_rungek2))
error_y2 =
    0.0016
```

con lo que se puede observar la considerable disminución del error, en relación al método de Euler.

10.2.3. Comandos de MATLAB

MATLAB dispone de varios comandos para resolver $\mathbf{y}' = \mathbf{f}(t, \mathbf{y}),\ \mathbf{y}(t_0) = \mathbf{c}$. Uno de ellos, que describimos a continuación, es **ode45**.

`[t y]=ode45(f,[t0 tN],c)`: Resuelve el problema, utilizando un método Runge-Kutta de orden 4, en el intervalo `[t0,tN]`. La variable de salida `t` es un vector columna formado por los nodos que se han utilizado para dividir el intervalo, mientras que `y` contiene, por columnas, los valores de las incógnitas en dichos nodos.

Nota: Para que se utilicen `t0 t1 t2...tN` como nodos, debemos reemplazar el argumento de entrada `[t0 tN]` por `[t0 t1 t2...tN]`.

Ejemplo 10.7 *Dado el problema de valores iniciales*

$$\frac{d^3y}{dt^3} + \frac{d^2y}{dt^2} - 2y = 0,\ y(0) = -1,\ y'(0) = 2,\ y''(0) = -2$$

hállese su solución aproximada, en el intervalo $[0, 4]$*, utilizando* `ode45`*, su solución exacta, así como el error máximo cometido en los nodos, y represéntense ambas soluciones.*

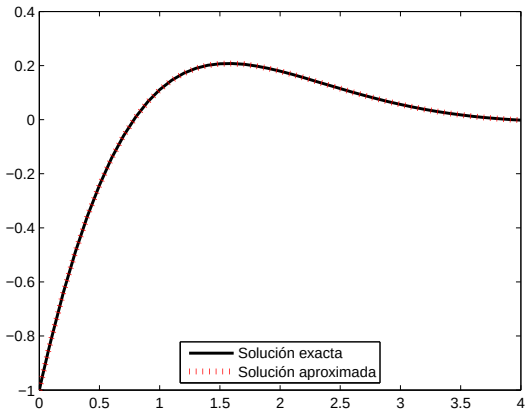
Solución: Escribimos la ecuación como $y''' = 2y - y''$ y denotamos $y_1 = y,\ y_2 = y',\ y_3 = y''$, con lo que el problema pasa a ser

$$\begin{cases} y_1' = y_2 \\ y_2' = y_3 \\ y_3' = 2y_1 - y_3 \end{cases} \qquad y_1(0) = -1,\ y_2(0) = 2,\ y_3(0) = -2$$

y lo resolvemos, ejecutando:

```
>> F=@(t,Y) [Y(2);Y(3);2*Y(1)-Y(3)]
F =
    @(t,Y)[Y(2);Y(3);2*Y(1)-Y(3)]
```

```
>> [t Y_ode45]=ode45(F,[0 4],[-1;2;-2]);
>> y_ode45=Y_ode45(:,1);
>> y=dsolve('D3y+D2y-2*y=0','y(0)=-1','Dy(0)=2','D2y(0)=-2')
y =
sin(t)/exp(t) - cos(t)/exp(t)
>> y_exacta=matlabFunction(y)
y_exacta =
    @(t)-exp(-t).*cos(t)+exp(-t).*sin(t)
>> error=max(abs(y_exacta(t)-y_ode45))
error =
  5.4430e-005
>> plot(t,y_exacta(t),'k','linewidth',2)
>> hold on, plot(t,y_ode45,'r:','linewidth',4)
>> legend('Solución exacta','Solución aproximada','location','south')
```



Nota: Ejecutando `length(t)`, se observa que el intervalo se ha dividido en 45 puntos. Si quisiéramos, por ejemplo, resolver con una distancia entre nodos de 2 centésimas, tendríamos que hacer:

```
>> [t Y_ode45]=ode45(F,0:0.02:4,[-1;2;-2]);
```