Notas de MatLab@

Mario G. Medina Valdez*

Departamento de Matemáticas
Universidad Autónoma
Metropolitana-Iztapalapa

Av San Rafael Atlixco 186
Col. Vicentina
México DF CP 09340
MEXICO

*Correo electrónico mvmg@xanum.uam.mx

MatLab es un medio ambiente interactivo en el que podemos realizar cálculos numéricos, cálculos simbólicos (mediante Maple) y visualizaciones científicas. Su nombre proviene de MATrix LABoratory, el cual fue diseñado por The Mathworks, Inc. (http://www.mathworks.com). También es un lenguaje de programación el cual está basado (principalmente) en arreglos o matrices, como veremos más adelante; además es compacto y sencillo de manejar y la intuición puede ayudar mucho al trabajar con MatLab. Podemos convertirlo a código C a través del compilador de MatLab para una mejor eficiencia.

Una vez abierto MatLab en la computardora aparece el símbolo “>>” y podemos empezar a usarlo. Para salir de Matlab basta escribir **quit** o **exit** después del símbolo anterior.

```
>> quit
```

o

```
>> exit
```

Caracteres especiales

MatLab tiene una cantidad de caracteres especiales los cuales son caracteres reservados de MatLab para distintos fines. Algunos son los correspondientes a las operaciones aritméticas: +,-,*, / y \. Pero otros caracteres tienen otros propósitos:

% -- después de este símbolo todo lo escrito en el mismo renglón es considerado como comentario.

; -- delimita comandos o nos ayuda a establecer dos comandos en una misma línea, al escribirlo también suprime cualquier tipo de salida obtenida al ser ejecutado el comando respectivo.

```
>> x = 1:2:9; y = 5:19; % dos commandos en una misma línea
```

... -- continuación de un comando, por ejemplo

```
>> x = [ 1 3 5 ...
       7 9]; % x = [1 3 5 7 9] escrito en dos líneas
```

: -- determinación de un rango de valores. Por ejemplo

```
>> x = [1:2:9]; % x=[1,3,5,7,9]
```

' -- transposición de matrices o vectores. En caso de una matriz o vector de entradas complejas se obtiene conjugación compleja y transposición de la matriz.

, -- commando para delimitar

```
>> x = [1:2:9], y = [1:9] % dos comandos en una misma línea
```

. -- precede a una operción aritmética para realizar una operación elemental

Operaciones Elementales

+ - * / ^ (definiciones algebraicas y matriciales)
.+ .- .* ./ .^ (operaciones elemento por elemento entre arreglos)

Funciones elementales de MatLab

>> abs(x) % valor absoluto de x>> exp(x) % e to the x-th power
>> fix(x) % redondea x con el entero más cercano a cero
>> log10(x) % logaritmo de x en base 10
>> rem(x,y) % residuo de dividir x entre y, es decir, residuo de x/y
>> sqrt(x) % raíz cuadrada de x
>> sin(x) % seno de x; x medido en radianes
>> acoth(x) % inverso de la cotangente hiperbólica de x
>> help elfun % nos provee de una lista de funciones elementales disponibles

Trabajando con Matrices

Trabajar con matrices en MatLab es bastante simple, para introducir una matriz de tamaño 3x3 es suficiente escribir en el promt de MatLab la matriz

>> B=[2,4,1;-5,3,-1]

Con lo que se obtiene la respuesta

B =

2 4 1
-5 3 -1

Las tres primeras entradas están separadas por comas (,), al igual que las últimas tres, pero las tres primeras están separadas de las tres últimas por un punto y coma (;), éste último signo nos separa los renglones de la matriz. Otra forma de obtener la misma matriz es dejar espacios en blanco entre las tres primeras entradas en lugar de poner una coma. Lo que no debemos olvidar es el punto y coma que divida los renglones.

>> B=[2 4 1;-5 3 -1]

B =

2 4 1
-5 3 -1

IMPORTANTE: Si deseamos que no se imprima la respuesta o salida debemos usar un punto y coma después del comando

```
>> B=[2,4,1;-5,3,-1];
```

Podemos continuar trabajando con matrices, en este caso con una matriz cuadrada, calcular su se determinante para determinar si es una matriz invertible y posteriormente calcular su inversa

```
>> C=[1,2;4,3]


C =

    1    2
    4    3
```

El comando para calcular el determinante de la matriz C está dada por det(C)

```
>> d=det(C)

d =

   -5
```

De esta forma, la matriz C es invertible y existe su matriz inversa y el comando para calcular la inversa de la matriz C está dada por inv(C)

```
>> D=inv(C)

D =

   -0.6000    0.4000
    0.8000   -0.2000
```

Asimismo es posible verificar que la inversa de C es la matriz D mediante el comando producto de matrices:

```
>> I=C*D

I =

    1    0
    0    1
```

Si una matriz C es invertible, el cual es el caso de la matriz entonces podemos resolver el sistema lineal de ecuaciones Cx=B y en tal caso, X = (1/C)B, o, usando la notación de Matlab tenemos que, X = C\B. Intentemos resolver el sistema Cx = B, con la matriz C dada anteriormente y B= [1; 1]. Observemos que el vector B es un vector columna.

```
>> B=[1;1]

B =

     1
     1
```

Ahora procedemos a calcular la solución del sistema de ecuaciones lineales Cx=B

```
>> x=C\B

x =

   -0.2000
    0.6000
```

Para verificar que efectivamente el vector obtenido es solución del sistema de ecuaciones lineales consideremos el producto de Cx y debemos obtener la matriz B.

```
>> C*x
```

Obtenemos la matriz

```
ans =

     1
     1
```

que es efectivamente la matriz B.

Otras instrucciones básicas al trabajar con matrices son las siguientes: transpuesta **A'** de una matriz A, rango **rank(A)** de una matriz, reducción por renglones **rref(A)** de una matriz A de tamaño nxn y comandos para determinar los vectores y valores propios de una matriz cuadrada A, dado por **[w,lambda]=eig(A)**. Mientras los primeros comandos dan resultados inmediatos, el último, referente a los vectores y valores propios, el resultado obtenido se manifiesta a través de dos matrices, la primera matriz **w** corresponde a una matriz cuyos vectores columna corresponden a los vectores propios de la matriz, mientras la matriz lambda corresponde a una matriz cuadrada diagonal donde cada elemento en la diagonal es un valor propio y los elementos fuera de la

diagonal se anulan, como se muestra en los siguientes ejemplos para la matriz A dada.

```
>> A=[1,2;4,3]

A =

    1    2
    4    3



>> A'

ans =

    1    4
    2    3


>> rank(A)

ans =

    2

>> rref(A)

ans =

    1    0
    0    1

>> [w,lambda]=eig(A)

w =

   -0.7071   -0.4472
    0.7071   -0.8944


lambda =

   -1    0
    0    5
```

También podemos referirnos a una entrada en particular de la matriz A usando el comando **A(m,n)**, el cual nos regresa el número que se encuentra en la entrada correspondiente al renglón m y la columna n.

```
>> A(1,2)

ans =

    2
```

Igualmente es posible extraer submatrices de una matriz dada con usando comandos muy parecidos a los usados en el ejemplo anterior.

Consideremos la matriz

```
>> M=[1,2,3;4,5,6;7,8,0]

M =

    1    2    3
    4    5    6
    7    8    0
```

y queremos extaer la submatriz que conste de la j-ésima columna de M, es decir la matriz columna cuyos elementos están dados por las entradas M(1,j), M(2,j) y M(2,j),. En el siguiente jemplo extraemos la segunda columna de la matriz M

```
>> M(:,2)

ans =

    2
    5
    8
```

o extraer el tercer renglón de la matriz M:

```
>> M(3,:)

ans =

    7    8    0
```

Finalmente, si queremos extraer la submatriz de M que conste de los primeros dos renglones y las últimas dos columnas de necesitamos el comando

```
>> M(1:2,2:3)

ans =

     2     3
     5     6
```

Construcción de Matrices (arreglos) a partir de otras.

Si consideramos la matriz M del ejemplo anterior y el vector columna

```
>> v=[-1,pi,-2]'

v =

  -1.0000
   3.1416
  -2.0000
```

y le queremos añadir una cuarta columna, aquella cuyas entradas son los elemntos del vector v, usamos el comando **[M,v]**.

```
>> [M,v]

ans =

   1.0000   2.0000   3.0000  -1.0000
   4.0000   5.0000   6.0000   3.1416
   7.0000   8.0000        0  -2.0000
```

y si le queremos añadir un cuarto renglón y sus lentradas son los elemntos del mismo vector

```
>> [M;v']

ans =

   1.0000   2.0000   3.0000
   4.0000   5.0000   6.0000
   7.0000   8.0000        0
  -1.0000   3.1416  -2.0000
```

Observemos que en el comando hemos cambiado la coma (,) por un punto y coma (;).

También podemos borrar un renglón o columna de una matriz dada

```
>> M(2,:)=[] % borra el segundo renglon de M

M =

    1    2    3
    7    8    0
```

Indexación lógica:

Un tipo distinto de indexación es la indexación lógica. Los índices lógicos se obtienen de una de las relaciones entre dos objetos dadas por: **==**(igual a), **<** (menor que), **<=** (menor o igual), **~=** (diferente a), **>** (mayor que) y **>=** (mayor o igual a). Como resultado de usar una relación lógica obtenemos un arreglo lógico (logical array), cuyos elementos son 0 o 1 dependiendo que la relación sea *verdadera* o *falsa.* Al uasr un arreglo lógico como un índice obtenemos aquellos valoresdonde el índice es 1.

Consideremos un cuadrado mágico 3x3, obtengamos las entradas de la matriz sonde los elementos son mayores a 3

```
>> N=magic(3)

N =

    8    1    6
    3    5    7
    4    9    2

>> N>3

ans =

    1    0    1
    0    1    1
    1    1    0
```

O consideremos otro cuadrado mágico de tamaño 3x3 y borrémosle la segunda columna

```
>> B = magic(3)

ans =

     8     6     1
     3     7     5
     4     2     9

>> B(:,logical([1 0 1]))    % se borra la segunda columna de B

ans =

     8     1
     3     5
     4     9
```

También es posible usar la forma vectorial en expresiones lógicas, por ejemplo, al considerar una matriz mágica A de tamaño 4x4 y una matriz L, también de tamaño 4x4 que consista de ceros

```
>> A = magic(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1



>> L = zeros(4,4)

L =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

```
>> L(A>8) = 1

L =

     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

La matriz **L** de tamaño 4x4 que está arriba indica cuales son las entradas de A que son mayores que 8 a través de la expresión (L(i,j) = 1). Podemos obtener la misma matriz de otra forma,

```
>> K = A>8

K =

     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

Otros tipos de operaciones con arreglos

Al considerar la matriz

```
>> A=[1,2,3,4;5,6,7,8;9,10,11,12;13,14,15,16]

A =

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

podemos construir los arreglos

```
>> sum(A,1)

ans =

    28    32    36    40
```

y obtenemos la suma de los elementos de las cuatro columnas, mientras que con el comando

```
>> sum(A,2)

ans =

   10
   26
   42
   58
```

obtenemos la suma de los renglones de la matriz A.

Herramientas útiles:

El comando **who** nos dirá cuales han sido todas las variables que han sido definidas en la sesión en que se trabaja.

```
>> who

Your variables are:

M    ans  b    v
```

El commando **whos** nos dirá, aparte de las variables definidas sus tamaños y alguna otra información

```
>> whos
  Name      Size         Bytes  Class

  M         2x3             48  double array
  ans       4x3             96  double array
  b         1x3             24  double array
  v         3x1             24  double array

Grand total is 24 elements using 192 bytes
```

El commando **pi** nos regresa el valor de pi

```
>> pi

ans =

   3.1416
```

El comando **eps** es una función que nos regresa el número más pequeño de punto flotante de MatLab. Esto es útil si se tiene un vector que pudiese contener ceros y se usará para como denominador de algo. Si se le añade eps al vector no se le

añade nada significativo al vector, pero no tendremos problemas al dividir por cero.

Los commandos **format long** y **format short**

```
>> format long
```

y

```
>> format short
```

intercambia los formatos cortos y largos cuando un número o conjunto de números son mostrados en la pantalla. El formato corto solo mostrará los primeros cuatro valores decimales después del punto decimal, mientras que el formato largo mostrará los primeros dieciséis números después del punto decimal.

Lazos (loops) en matlab:

Dada una matriz cuadrada A(2x2) y un vector columna x, si queremos calcular x1=Ax, x2=Ax1, x3=Ax2, un número finito de veces usamos lo que conoce como un ***loop***, lo que se muestra con el siguiente ejemplo

Ejemplo: Consideremos la matriz A=[1,0;0,2] y el vector columna n=[1;2] usamos el promp de MatLab para introsucir los datos correspondientes a la matriz y el vector columna

```
>> A=[1 ,0;0,2]

A =

    1    0
    0    2
```

Y a continuación el comando

```
>> n=[1;1]

n =

    1
    1
```

Con lo anterior quedan ene la memoria de la máquina los datos.Para realizar un loop usamos el comando

```
>> for i=1:5 n=A*n, end
```

Obtenemos por repuesta los siguientes vectores columna

```
n =

     1
     2

n =

     1
     4

n =

     1
     8

n =

     1
    16

n =

     1
    32
```

En este caso hemos calculado x1=A*x, x2=A*x1, x3=A*x2, x4=A*x3 y x5=A*x4.
De esta manera hemos visto como escribir un tipo especial de lazo, llamado en inglés un **for loop.** El cual permite realizar un trabajo repetitivo en MatLab.

OBSERVACIÓN: Este mismo tipo de loop lo podemos usar para iterar funciones reales, por ejemplo
Iterar la funcion lineal f(x)=5x-2 para cierto valor de x,

Damos el valor de x

```
>> x=3

x =

     3
```

e introducimos el ***for loop***

```
>> for i=1:5 x=5*x-2, end
```
Obteniendo por repuesta

```
x =

    13


x =

    63


x =

   313


x =

      1563

x =

      7813
```

ARREGLOS

Las matrices que hemos visto anteriormente son ejemplos de los que se llaman arreglos (**arrays** en inglés)

Podemos dar un arreglo simple, como una matriz de un único renglón. Por ejemplo, el arreglo de los números x=0,0.1π, 0.2π,0.3π,0.4π.

```
>> x=[0,0.1*pi,0.2*pi,0.3*pi,0.4*pi]

x =

      0    0.3142    0.6283    0.9425    1.2566
```

Podemos calcular sin(x) para cada uno de los valores del arreglo y obtener un nuevo arreglo y:

>> y=sin(x)

y =

0 0.3090 0.5878 0.8090 0.9511

Podemos realizar otro tipo de operaciones con arreglos, Por ejemplo, usando los arreglos x y y del ejemplo anterior, podemos calcular arreglos z=x+y, w=1.5x+2

>> z=x+y

z =

0 0.6232 1.2161 1.7515 2.2077

>> w=1.5*x+2

w =

2.0000 2.4712 2.9425 3.4137 3.8850

Mientras que para el arreglo z se obtiene de sumar x+y componente a componente, en el caso del cálculo de 1.5x+2, cada componente de x es multiplicada por 1.5 para posteriormente sumarle 2 a cada uno de los resultados obtenidos.

Si queremos obtener un nuevo arreglo v donde cada una de las componentes de este sea el cuadrado de cada una de las componentes de x, es necesario usar el siguiente comando v=x.^2, con un punto después de la x.

>> v=x.^2

v =

0 0.0987 0.3948 0.8883 1.5791

U obtener un nuevo arreglo b donde cada componente de multiplicar los componentes de x por los correspondientes componentes de v:

>> b=x.*v

b =

0 0.0310 0.2481 0.8372 1.9844

También podemos calcular la potencia de un escalar elevado a cada exponente dado por los elementos de un arreglo

```
>> m=2.^[2,3,4]

m =

    4    8   16
```

Observemos que si olvidamos el punto después de escribir el número 2 obtenemos un error:

```
>> 2^[2,3,4]
??? Error using ==> ^
Matrix must be square.
```

Longitud de un arreglo

Para determinar la longitud que tiene un arreglo que se haya obtenido previamente se usa el siguiente comando

```
>> length(m)

ans =

     3
```

donde el arreglo m es el arreglo previo que se obtuvo de elevar 2 a las potencias 2, 3 y 4.

Otra manera de obtener otro arreglo a partir de dos arreglos de la misma longitud es elevar el primer elemento del arreglo v a la potencia dada por el primer elemento del arreglo b, los mismo con los segundas, terceras, cuartas y quintas entradas de los arreglos

```
>> v.^b

ans =

    1.0000    0.9307    0.7941    0.9056    2.4760
```

Generación de Arreglos:

Si queremos generar un arreglo de 6 números que se encuentren distribuidos uniformemente entre los valores 0 y 2π necesitamos el siguiente comando x=linspace(0,pi,6):

```
>> x=linspace(0,pi,7)

x =

         0    0.5236    1.0472    1.5708    2.0944    2.6180    3.1416
```

En caso de que queramos una mayor cantidad de elementos en el arreglo necesitamos cambiar el número 7 por la nueva cantidad, digamos 13.

```
>> x=linspace(0,pi,13)

x =

  Columns 1 through 7

         0    0.2618    0.5236    0.7854    1.0472    1.3090    1.5708

  Columns 8 through 13

    1.8326    2.0944    2.3562    2.6180    2.8798    3.1416
```

Otra manera de construir arreglos es la siguiente **x=0:0.2:pi**

```
>> x=0:0.2:pi
```

Obteniendo por respuesta un arreglo de 16 números dado por:

```
x =

 Columns 1 through 7

      0    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000

 Columns 8 through 14

   1.4000    1.6000    1.8000    2.0000    2.2000    2.4000    2.6000

 Columns 15 through 16

   2.8000    3.0000
```

Más generación de arreglos

Podemos generar un arreglo que conste de los números del 1 al 10:

```
>> a=1:10

a =

    1    2    3    4    5    6    7    8    9   10
```

Asimismo podemos generar series de tiempo que vayan de 0 a 20 de dos en dos

```
>> b=0:2:20

b =

    0    2    4    6    8   10   12   14   16   18   20
```

Indexación o etiquetado en un arreglo

No podemos terminar de referirnos a los arreglos sin hacer notar que MatLab etiqueta todos los elementos de un arreglo, y para conocer cual es el elemento que se encuentra en cierta “entrada” del arreglo o cuales son los elementos de ciertas entradas del mismo se usan los comandos v(1), v(2),...,v(end), v(2:5) o v(4:end) para bloques de elementos del arreglo:

```
>> v(1)

ans =

    0
```

```
>> v(3)

ans =

   0.3948

>> v(2:5)

ans =

   0.0987    0.3948    0.8883    1.5791

>> v(3:end)

ans =

   0.3948    0.8883    1.5791
```

Arreglos de más de dos dimensiones

Existe una manera de construir arreglos de más de dos dimensiones, pues en el caso de arreglos unidimensionales estamos hablando de vectores (renglón o columna), en el caso de arreglos bidimensionales se tienen las matrices; así que también podemos construir arreglos tridimensionales, por ejemplo.

```
>> cat(3,M,M)

ans(:,:,1) =

   1    2    3
   7    8    0


ans(:,:,2) =

   1    2    3
   7    8    0
```

A partir de la matriz M dada por

```
>> M=[1,2,3;4,5,6;7,8,0]

M =

    1    2    3
    4    5    6
    7    8    0
```

construimos el arreglo tridimensional

```
>> cat(3,M,M)

ans(:,:,1) =

    1    2    3
    4    5    6
    7    8    0


ans(:,:,2) =

    1    2    3
    4    5    6
    7    8    0
```

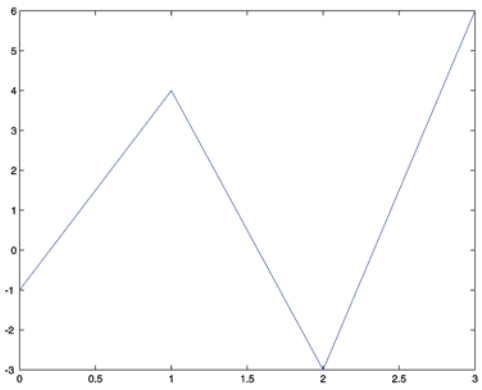
la primer matriz corresponde a la matriz que se encuentra en el frente del arreglo tridimensional, mientras que la segunda matriz se encuentra en la parte posterior del arreglo de tres dimensiones.

Uso de arreglos para construir gráficas

Si queremos construir una gráfica que una los puntos del plano dados por (0,-1), (1,4), (2,.3) y (3,6), los cuales están conectados por segmentos de recta es necesario dar los arreglos de las primeras y segundas coordenadas de los puntos y usar el comando **plot:**

```
>> x=[0,1,2,3];y=[-1,4,-3,6];plot(x,y)
```

Con lo que obtenemos la gráfica deseada:



Uso de arreglos para construir gráficas de funciones de una variable

Para construir la gráfica de f(x)=sin(x) con $x \in [0,2\pi]$, necesitamos generar dos arreglos, el primero se obtiene de dividir el intervalo $[0,2\pi]$ en un número suficientemente grande de puntos

```
>> x=linspace(0,2*pi,100);
```

Hemos usado un punto y coma al final del renglón indicando que no se escriban los 100 elementos del arreglo

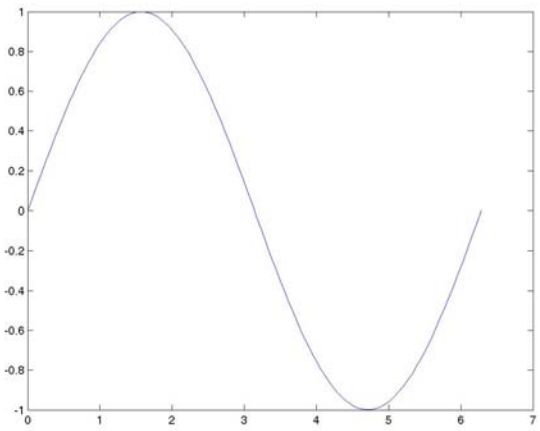
Posteriormente hay que generar otro arreglo, pero este consta del arreglo que se obtiene de evaluar sin(x) para cada uno de los elementos del arrglo x

```
>> y=sin(x);
```

Ahora podemos graficar los puntos correspondientes asociados al par de arreglos x y y usando el comando plot:

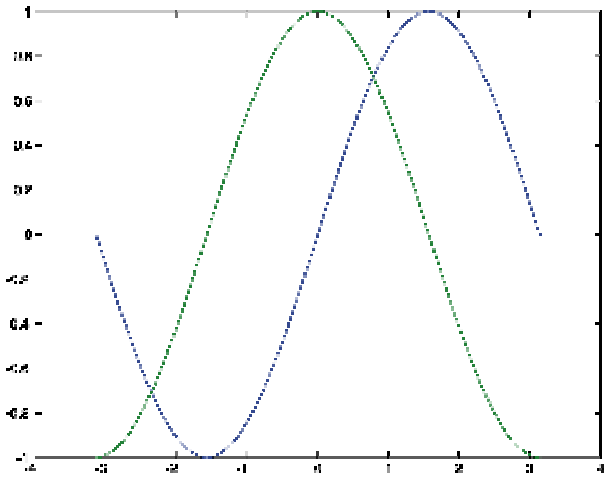
```
>>plot(x,y)
```

Obteniendo la gráfica



Dos bosquejos de funciones de variable real en un mismo gráfico

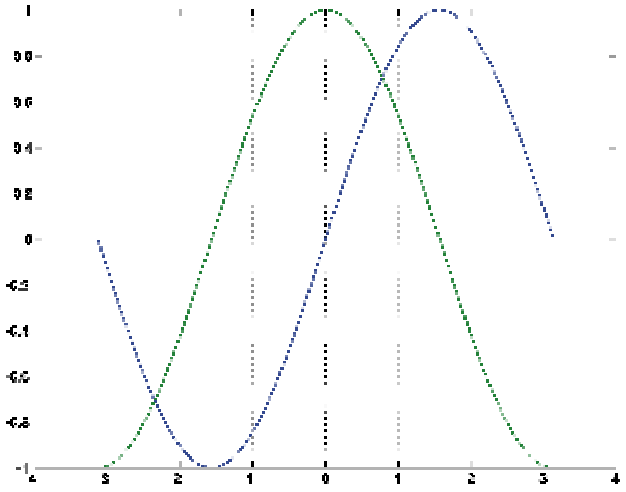
```
>> x=linspace(-pi,pi,100);
>> y=sin(x);
>> z=cos(x);
>> plot(x,y,x,z)
```



La sucesión de comandos

```
>> x=linspace(-pi,pi,100);
>> y=sin(x);
>> z=cos(x);
>> plot(x,y,x,z);grid
```

donde hemos añadido el comando grid en el último renglón le añade una rejilla al gráfico que anteriormente obtuvimos

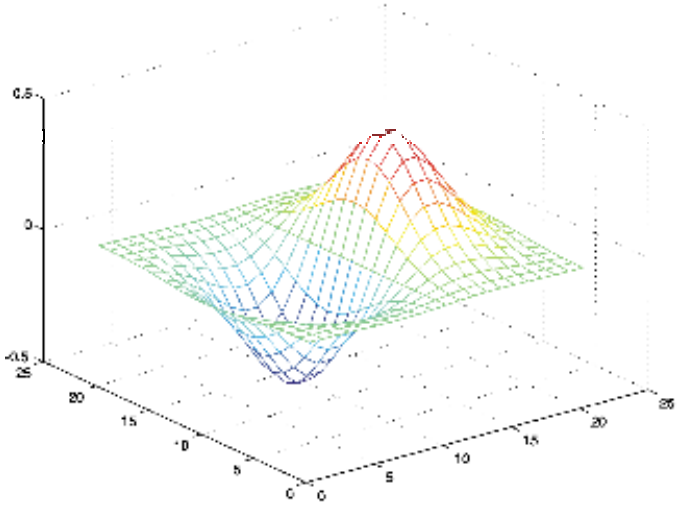


Funciones de dos variables

A continuación se muestra como graficar la función z(x,y) = x exp(- x^2 - y^2):

```
>> [x,y] = meshgrid(-2:.2:2, -2:.2:2);
>>  z = x .* exp(-x.^2 - y.^2);
>>  mesh(z)
```

El primer comando nos permite crear una matriz cuyas entradas son los puntos una celosía o rejilla del cuadrado -2 <= x <= 2, -2 <= y <= 2. Los cuadrados pequeños miden 0.2 unidades de ancho y 0.2 unidades de altura; el segundo comando crea una matriz cuyas entradas son los valores de la función z(x,y) en los puntos de la celosía. Finalmente el tercer comando construye la gráfica usando los dos arreglos obtenidos previamente.

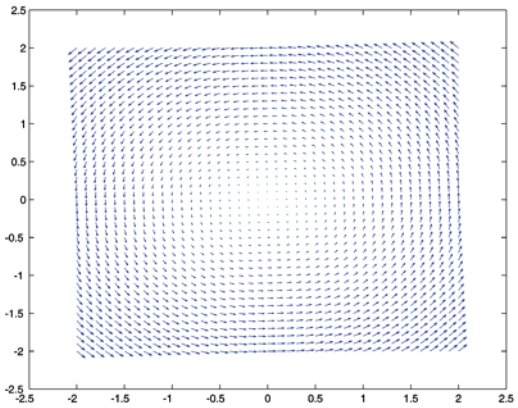


Otro uso de meshgrid (Campo de direcciones de un campo vectorial):

La sucesión de comandos

```
>> [x,y]=meshgrid(-2:0.1:2);
>> u=-y;v=x;
>> quiver(x,y,u,v)
```

Produce el campo de direcciones asociados al campo vectorial
u=-y, v=x



Quiver plot dibuja los vectores velocidad como flechas con componentes (u,v) en los puntos (x,y). Las matrices X,Y,U,V deben tener el mismo tamaño y contener los correspondientes componentes de posición y de velocidad (X y Y pueden ser también vectores que especifiquen una rejilla uniforme). Quiver muestra las flechas a escala para que quepan en la rejilla.

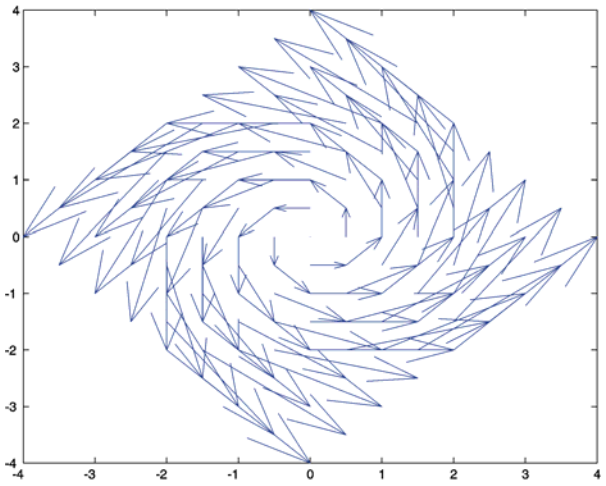
Quiver(U,V) dibuja vectores velocidad en puntos espaciados de igual manera en el plano x-y.

Quiver(U,V,S) o QUIVER(X,Y,U,V,S) reescala automáticamente las flechas para que quepan en la rejilla y posteriormente las encoge/estira por el escalar S. Se debe usar S=0 para dibujar las flechas sin el escalamiento automático.

Usamos

```
   >> [x,y]=meshgrid(-2:0.5:2);
u=-y;v=x;
quiver(x,y,u,v,0)
```

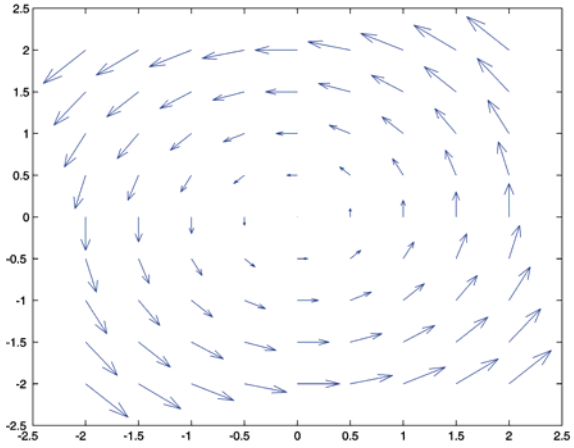
para obtener el campo de diecciones



Y usamos

```
>> [x,y]=meshgrid(-2:0.5:2);
u=-y;v=x;
quiver(x,y,u,v)
```

para obtener el campo de direcciones reescalado



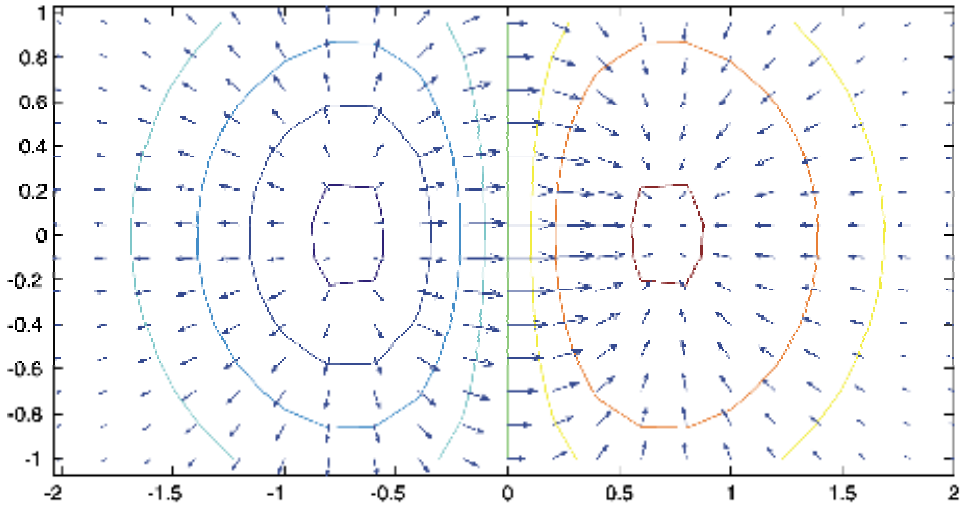
QUIVER(...,LINESPEC) uses the plot linestyle specified for
the velocity vectors. Any marker in LINESPEC is drawn at the base
instead of an arrow on the tip. Use a marker of '.' to specify
no marker at all. See PLOT for other possibilities.

QUIVER(...,'filled') fills any markers specified.

H = QUIVER(...) returns a vector of line handles.

Ejemplo:

```
[x,y] = meshgrid(-2:.2:2,-1:.15:1);
z = x .* exp(-x.^2 - y.^2); [px,py] = gradient(z,.2,.15);
contour(x,y,z), hold on
quiver(x,y,px,py), hold off, axis image
```



Funciones (functions) y escritos (scripts)

Estos son textos que se escriben en **m-archivos (m-files)** que contienen un comando o una sucesión de comandos de Matlab y al guardarse en un archivo, este debe terminar con la extensión **.m.** Los m-archivos se dividen en dos tipos: **script** y **functions.** Al guardar un m-archivo debemos tener cuidado de fijarnos en que directorio guardamos el archivo, pues para ejecutar tal archivo debemos establecer la trayectoria del rachivo. Por ejemplo si queremos ejecutar el archivo *gauss.m* y este tiene trayectoria c:\dir1\dir2\dir3 entonces debemos establecer esta trayectoria en la ventana de comandos por ejemplo, mediante el comando

```
>> cd c:\dir1\dir2\dir3
```

y aparecerá nuevamente el promp

```
>>
```

y procedemos a escribir el nombre del archivo *gauss.m*

```
>> gauss.m
```

Para escribir este tipo de archivos podemos usar el editor de textos que acompaña a MatLab.

Archivos scritp

En el caso del primer tipo de m-archivos, los archivos script, éstos se usan si necesitamos efectuar una tarea que consista de distintos pasos. Cuando se lee un archivo tipo script los comandos escritos en este se interpretan de manera literal como si estuvieran escritos en la ventana de comandos. Los archivos script permiten dar una lista larga de comandos que puede ser revisada en cualquier momento directamente en el archivo script, además podemos realizar la misma tarea posteriormente sin tener que escribir renglón por renglón la sucesión de comandos en la ventana de comandos, entre otros aspectos convenientes.

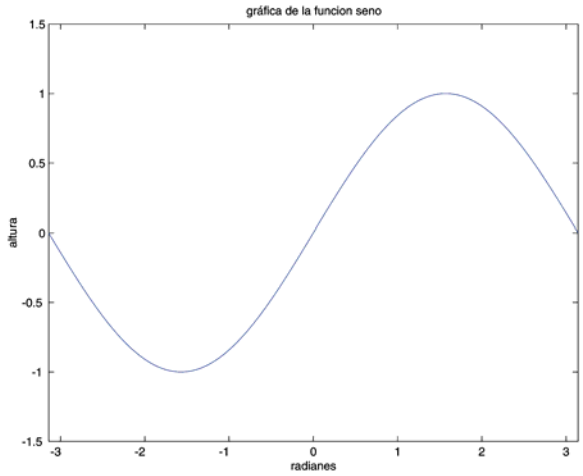
Podemos crear con la ayuda del editor un archivo llamado seno.m y escribir el siguiente código

```
X=linspace(-pi,pi,200);
Y=sin(X);
plot(X,y);
title('gráfica de la funcion seno');
axis([-pi  pi –1.5 1.5]);
```

Como se comentó anteriormente, debemos tener cuidado con la trayectoria del archivo y donde lo guardamos. Finalmente en la ventana de comandos llamamos el archivo

>> seno

con lo que obtenemos la gráfica de la función seno definida en el intervalo $[-\pi, \pi]$ en la ventana dada por el rectángulo $[-\pi, \pi]$x[-1.5,1.5],



Archivos función

Los archivos función permiten usar MatLab a una mayor capacidad que utilizando solamente la ventana de comandos.

Todo archivo-función debe comenzar con una primera línea del tipo siguiente

function [salida1, salida2]=mifuncion(ent1, ent2, ent3)

Las variables **ent1, 1nt2** y **ent3** son argumentos de entrada, mientras que **salida1, salida2** son argumentos de salida. Una función puede tener tantos argumentos de entrada y de salida como se desee y llamarlos cuando éstos se necesiten. Es muy importante observar que **el nombre mifunción de la función debe ser exactamente igual al nombre del archivo**.

A continuación damos algunas características que diferencian un archivo tipo script de un archivo tipo function:

Archivo script: No tiene argumentos de Input (Entrada) o Output (Salida), las variables de trabajo son globales y este tipo de archivos son útiles para automatizar una serie de comandos u órdenes que se quiere repetir.

Archivo function: Puede aceptar atgumentos de Input (Entrada) o Output (Salida), las variables internas son locales por defecto y este tipo de archivos son útiles para agregar a MatLab las funciones que uno desee a.

Por ejemplo podemos crear en el editor un archivo llamado *cubica.m* donde definiremos una función con el mismo nombre *cubica* tecleando en el editor la siguiente lista de comandos y procedemos posteriormente a guardar el archivo con el nombre mencionado

```
function  cubica=f(x)
cubica=x.^3;
```

Finalmente en la ventana de comandos escribimos

>> cubica(1.1)

Y lo que tenemos es evaluar el cubo del valor 1.1

ans =

1.3310

En lugar de cubica(1.1) podemos escribir cubica(nombredevariable) en caso de que a nombredevariable le sea asignado un valor numérico. Observemos hemos escrito x.^3 en lugar de escribir solamente x^3, por lo que podemos evaluar la función cubica tambien en arreglos

>> cubica([1.1 4 pi 0])

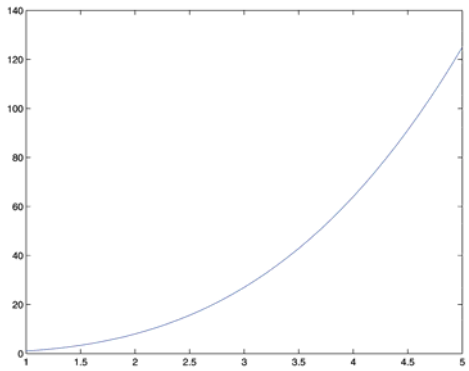
ans =

1.3310 64.0000 31.0063 0

Con los siguientes comandos podemos graficar la función f(x)=x^3, $x \in [1,5]$. Considerando todos los puntos del intervalo [1,5] espaciados por 0.1 unidades de longitud, para construir un nuevo arreglo y, y finalmente graficar ambos arreglos matriciales unidimensionales.

```
>>x=[1:0.1:5];
>>y=cubica(x);
>>plot(x,y)
```

Como lo muestra la siguiente Figura



Continuemos con los ejemplos de cómo construir y usar las m-funciones para realizar cálculos numéricos.

Recordemos que el área de un triángulo de lados a, b y c está dada por $Area = \sqrt{s(s-a)(s-b)(s-c)}$, donde $s = \frac{a+b+c}{2}$. Asimismo para construir una función en MatLab debemos usar un nombre que no sea el de alguna de las funciones que son usadas por default por MatLab, éro que a su vez deje claro el papel que juega la función en nuestros cálculos. En este caso al querer calcular el área de un triángulo, podemos llamar a la función simplemente **area** y al archivo correspondiente **area.m.** La estructura que necesitamos es

function [A]=area(a,b,c)

donde **A** representa un único dato de salida y **a,b** y **c** son los tres datos de entrada. Podemos usar el símbolo de porcentaje % para escribir comentarios, por ejemplo, sobre las características de la función que estamos definiendo lo que le permita a cualquier usuario entender todo el proceso que estemos siguiendo

```
function [A]=area(a,b,c)
%Calculamos al area de un triangulo cuyos lados son a,b y c
% entradas son las longitudes de los lados : a,b,c
% salida es el area A calculada para el triangulo
s=(a+b+c)/2; % s=promedio de las longitudes de los lados del triangulo
A=sqrt(s*(s-a)*(s-b)*(s-c));
```

Si usamos el comando de ayuda en la ventana de comandos obtenemos

```
>> help area

 Calculamos al area de un triangulo cuyos lados son a,b y c
  entradas son las longitudes de los lados : a,b,c
  salida es el area A calculada para el triangulo
```

Si queremos calcular el área de un triángulo de lados a=3, b=4, c=5, solo debemos teclear en la ventana de comandos lo siguiente

```
>> area(3,4,5)

ans =

    6
```

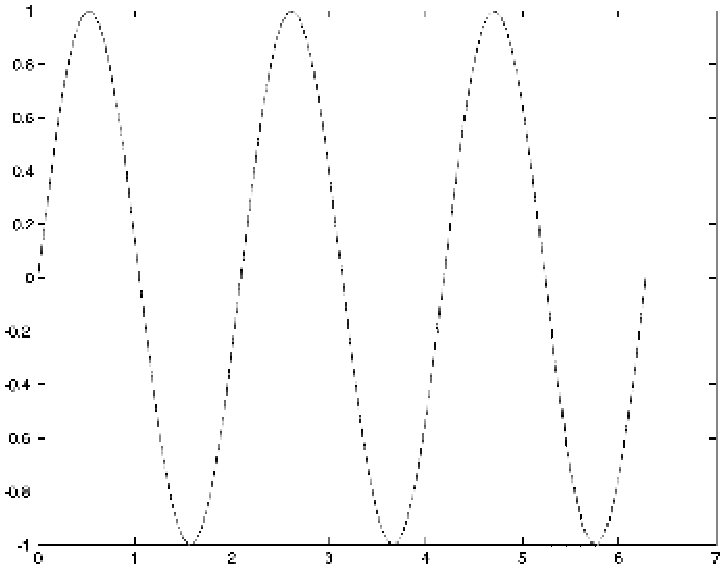
Ahora supongamos que queremos graficar la función seno en el intervalo [a,b], pero no queremos establecer en el m-archivo la cantidad de puntos a considerar al particionar este intervalo, sino queremos tomar en el m-archivo N subintervalos. En tal caso podemos crear un archivo **senito.m** donde el texto está dado por

```
x=a:(b-a)/N:b; %Damos el dominio [a,b] donde queremos graficar la fumcion seno
y=sin(w*x); %definimos la funcion seno
plot(x,y)% graficamos la funcion como grafica de dos arreglos
```

posteriormente, en la ventana de comandos damos los siguientes comandos, primero se determinan los valores de **a,b, N** y **w** para posteriormente proceder a "correr" el programa **senito.m.**

```
>> a=0,b=2*pi,N=100,w=3

a =

    0


b =

   6.2832


N =

   100


w =

    3

>> senito
```

y la gráfica que obtenemos es la siguiente



Otro ejemplo:

Escribamos un m-archivo tipo function, el cual usaremos posteriormente

```
function y=funcion(x); %tenemos la forma clasica de un archivo.m tipo function
y=sin(2.*x); %definimos la funcion que usaremos posteriormente
```

en un nuevo m.archivo tipo script, que llamaremos otrafuncion.m escribimos el programa en el cual usaremos el m.archivo tipo function.

```
x=0:2*pi/N:2*pi;
y=cos(x);
z=funcion(x);
plot(x,y,x,z)
```

observemos que en el segundo renglón hemos mandado llamar el m.archivo tipo function. Ya en la ventana de comandos mandamos llamar este último archivo

```
>> otrafuncion
```

y obtenemos la gráfica siguiente

Asimismo, es posible etiquetar los ejes de la gráfica, así como añadirle un título a la misma o cambiar la ventana asociada a una gráfica. Los comandos adecuados para éstas o tras tareas son los siguientes:

xlabel('etiqueta') etiqueta el eje horizontal,
ylabel('etiqueta') etiqueta el eje vertical,
title('etiqueta') le añade un título a la gráfica,
axis([a b c d]) cambia la ventana de la gráfica al rectángulo $x \in [a,b]$, $y \in [c,d]$,
grid le añade un enrejado rectangular a la gráfica,
hold on mantiene "congelada" una gráfica de tal manera que se puedan plasmar distintas gráficas en la ventana original.
hold off este comando "descongela" la gráfica que se había congelado previamente; la siguiente gráfica borrará la gráfica actual antes de ser dibujada.
subplot se usa para poder dibujar distintas gráficas en una misma ventana.

También podemos usar otros comandos para cambiar el color de la gráfica, la cual de manera automática MatLab dibuja de color azul, asimismo es posible cambiar el estilo de la línea que se usará para graficar, por ejemplo, una función real-valuada.

y	yellow	.	punto
m	magenta	o	círculo
c	cyan	x	cruces
r	rojo	+	signos de suma
g	verde	-	línea sólida
b	azul	*	estrellas
k	negro	:	línea punteada
w	blanco	_.	punto y línea
		--	línea discontinua

Por ejemplo,

```
>> plot(x,y,'r')
```

producirá una gráfica en color rojo,

```
>> plot(x,y,'b:')
```

comando en que hemos combinado dos comandos, lo que producirá una gráfica azul con líneas punteadas.

Si tenemos dos gráficas en una misma y para distinguirlas se requiere que sean producidas en formas distintas, por ejemplo, una punteada y la segunda por líneas discontinuas, necesitamos escribir lo siguiente

```
>> plot(x,y,'--', x z, ':')
```

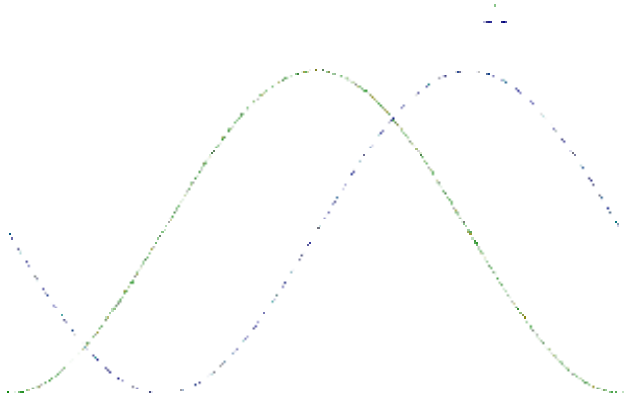
Y si además queremos añadirle una etiqueta a cada una de las gráficas para distinguirlas, podemos escribir lo siguiente

```
>>legend('primera grafica', 'segunda grafica')
```

La sucesión de comandos siguientes nos muestra como usarlos, para obtener una gráfica con dos curvas, asociadas a las funciones seno y coseno:

```
x=-pi:2*pi/100:2*pi;
y=cos(x);
plot(x,y,'g.-',x, sin(x),'b--');
xlabel('eje x');
ylabel('eje y');
title('Las graficas de coseno y seno');
axis([-pi pi -1.5 1.5]);
grid;
legend('grafica coseno','grafica de seno');
```

Obtenemos la gráfica siguiente:



Otro ejemplo de archivos tipo function y su uso:

Editemos un archivo tipo function como sigue, al que llamaremos func.m

```
function Y=func(X)
% Entrada est´a dada por un arreglo X
% Salida esta dada por un arreglo que resulta
% de aplicar la funcion f(x)=sen(x^2)/(1+2x^2) a todos los elementos de X
Y=sin(X.^2)./(1+2*X.^2);
```

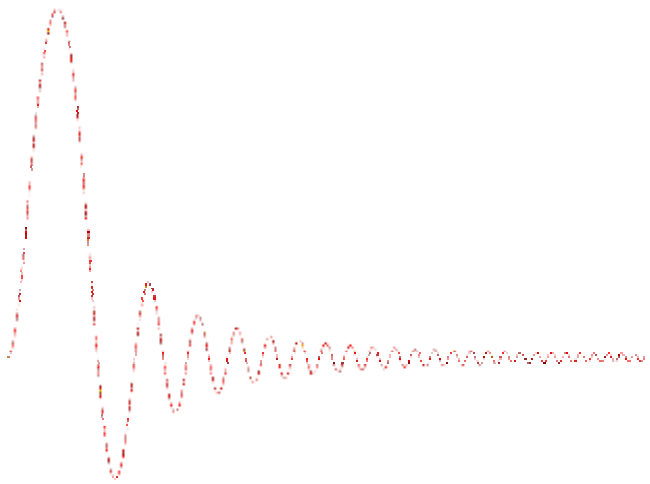
Observemos que debido a que un número real es también una matriz de tamaño 1x1, podemos usar este archivo tipo function para evaluar func(π). A continuación usaremos el archivo func.m para graficar en color rojo la función y=func(x) en el intervalo [0,4π], dando un título a la gráfica, así como añadiendo etiquetas a los ejes coordenados, escribiendo un archivo tipo scritp que llamaremos plotfunc.m y que contiene los siguientes comandos

```
x=[0:0.1:4*pi];
y=func(x);
plot(x,y,'r')
title('Grafica de la funcion f(x)=sin(x^2)/(1+2x^2)')
axis([0 14 –0.15 0.3])
xlabel(‘eje x’)
ylabel(‘eje y’)
```

No olvidemos determinar la trayectoria de los archivos en la ventana de comandos, después de haber hecho esto, procedemos a “correr” el archivo plotfunc.m en la ventana de comandos

>> plotfunc

con lo que obtenemos la gráfica siguiente



Programación en Matlab

Los siguientes operadores de relación, operadores lógicos y valores booleanos son elementos fundamentales para programar con Matlab

Operadores de relación:

==	Igual que
~=	No igual que
<	Menor que
>	Mayor que
>=	Mayor o igual que
<=	Menor o igual que

Operadores lógicos:

~	No	verdadero si y sólo si la proposición es falsa
&	Y	verdadero si las dos proposiciones son ciertas
|	O	verdadero si al menos una de las dos proposiciones es verdadera

Valores Boléanos: 0 Falso; 1 Verdadero

Las instrucciones **for**, **if** y **while** se manejan de manera semejante a su uso en lod distintos lenguaje de programación y su sintaxis es la siguiente:

Sintaxis de for:

for (variable del bucle rizo o loop=rango de bucle)
    instrucciones ejecutables
end

Sintaxis de if:

if (premisa)
    Instrucciones ejecutables
else
    Instrucciones ejecutables
end

Sintaxis de while:

while (premisa)
    Instrucciones ejecutables
end

A continuación damos un ejemplos del uso de estas instrucciones. En un primer ejemplo creemos un archivo sumas.m

```
for i=1:6
   M(i,1)=3;M(1,i)=3;
   %las entradas del primer renglon y la
   %primera columna de una matriz 6x6 son 3's
end
for i=2:6
   for j=2:6
      M(i,j)=M(i,j-1)+M(i-1,j);
   end
end
% a partir de las entradas del primer renglon y la primera columna
% completamos la matriz 6x6 de tal forma que la entrada (i,j) se obtiene de sumar
% las entradas (i,j-1) y (i-1,j)
M
% se pide escribir la matriz final M
```

Ahora en la ventana de comandos tecleemos el nombre del archivo

>>sumas

y obtenemos la matriz M

>> sumas

M =

```
   3    3    3    3    3    3
   3    6    9   12   15   18
   3    9   18   30   45   63
   3   12   30   60  105  168
   3   15   45  105  210  378
   3   18   63  168  378  756
```

Es posible cortar un cálculo de un bucle aún antes de que este se termine de realizar, y para esto usamos la instrucción **break.** Como vemos al escribir un nuevo m-archivo, que llamaremos corte.m

```
for i=1:150
   x=i^(1/3);
   % calcularemos las raices de 1 a 150, pero si en algun momento
   % despues de i igual a 10 se tiene tambien que la raiz cubica del numero menos el
%mayor
   % entero menor o igual a tal raiz se anula, detenemos los calculos
   if((i>10)&(x-floor(x)==0))
      break
   end
end
i
```

Al trabajar en la ventana de comandos y “correr” el archivo corte.m obtenemos la respuesta

>> corte

i =

```
   27
```

Si además es necesario mostrar un texto o un arreglo es indispensable el uso de la instrucción **disp.**

Consideremos el archivo mostrar.m

```
N=20;
i=10;
% a partir de 10 hasta veinte de 1 en 1 calculamos sin(3*pi/i) y se muestra este
%numero,  su cuadrado y su cubo respectivamente
while i<=N
   x=sin(3*pi/i); disp([ 'seno', 'cuadrado', 'cubo']), disp([ x x^2 x^3]), i=i+1;
end
```

El resultado obtenido es

```
>> mostrar
senocuadradocubo
   0.8090    0.6545    0.5295

senocuadradocubo
   0.7557    0.5712    0.4317

senocuadradocubo
   0.7071    0.5000    0.3536

senocuadradocubo
   0.6631    0.4397    0.2916

senocuadradocubo
   0.6235    0.3887    0.2424

senocuadradocubo
   0.5878    0.3455    0.2031

senocuadradocubo
   0.5556    0.3087    0.1715

senocuadradocubo
   0.5264    0.2771    0.1459

senocuadradocubo
   0.5000    0.2500    0.1250

senocuadradocubo
   0.4759    0.2265    0.1078

senocuadradocubo
   0.4540    0.2061    0.0936
```

Uso de rutinas de Matlab para resolver EDOs

Las rutinas de Matlab resuelven problemas de condiciones iniciales para conjunto de ecuaciones diferenciales acopladas . Recordemos que una ecuación diferencial ordinaria de primer orden con condición inicial toma la forma

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

donde **f**(t,y) es una función definida en una región Ω del espacio euclidiano $\mathbb{R}^n \times \mathbb{R}$ y $\mathbf{y}(t_0)=\mathbf{y_0}$ es la condición inicial. Existen cinco tipos de rutinas en Matlab: ode45, ode23, ode113, ode15s, ode23s; cada una de las cuales puede usarse si ciertas condiciones se satisfacen. Además las rutinas pueden cambiar dependiendo de la versión de Matlab en que se trabaje.

`ode45`: Primera rutina que se debe intentar en un problema. Es una rutina explícita de un paso variable con orden 4-5 adecuada para problemas que no son rígidos y que requieren cierta precisión

`ode23`: Rutina explícita de Runge-Kutta de orden 2-3, usado cuando no se requiere gran precisión. O cuando la función **f**(t, **y**) no es una función suave

`ode113`: Rutina de multipaso de Adams-Bashforth-Moulton de órdenes 1-13. Conveniente para problemas que no son rígidos y cuando el nivel de precisión es moderato o alto. Se usa también cuando las función **f**(t, **y**) provoca que el tiempo de cómputo sea alto. No usar cuando la función **f**(t, **y**) no sea suave (discontinuidades o derivadas de órden menor discontinuas.

`ode23s`: Es una rutina implícita de un paso de tipo Rosenbrock modificado de orden 2. Adecuado para problemas rígidos donde una precisón mínima es adecuada o cuando **f**(t, **y**) no es una función continua. Debemos recordar que un problema rígido es un problema donde las constantes de tiempo subyacentes varían en distintos órdenes de magnitud

`ode15s`: Es una rutina implicita de multipaso que usa diferenciación numérica y su porden varía en 1-5. Es conveniente para problemas rígidos que requieren de una precisión moderada. Es la rutina que se debe intentar cuando la rutina ode45 o no funciona o su eficiencia deja mucho que desear.

Como usar las rutinas para resolver EDOs.

Siempre debemos definir un archivo tipo función antes de intentar resolver una o un conjunto de ecuaciones diferenciales, para después escribir un archivo tipo script el cual debemos ejecutar posteriormente en la ventana de comandos.

Consideremos la ecuación diferencial de primer orden con condición inicial

$$y' = 2y + 2/(2+3t^2)\cos(t), \quad y(0) = y_0$$

y queremos graficar la solución de la ecuación diferencial que tiene como condición inicial $y(0) = -2$.

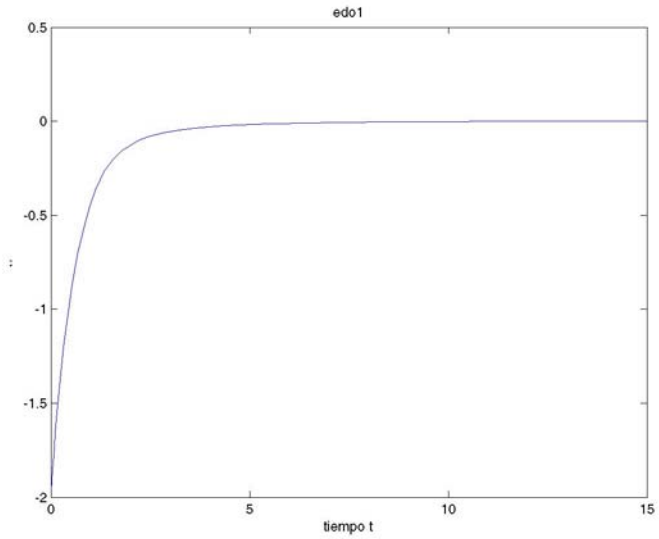
Primero debemos escribir un m-archivo tipo **function** que llamaremos **f1.m**

```
function z=f1(t,y)
z=y^2+2/(2+2*t^3)*cosy;
```

Ahora necesitamos escribir un m-archivo tipo **script** que llamaremos **edo1.m**

```
y0=-2 % establecemos la condicion inicialde la ecuacion diferencial
odeset('RelTol',1e-4,'AbsTol',[1e-5]);
% se establecen los errores relativo y absoluto
[t,y]=ode45(@f1,[0 15],y0);
%se determina la rutina de solucion de matlab
% a usar, se llama la funcion f1 que se encuentra definida en
% el archivo f1.m, el tiempo de solucion, asi como la condicion inicial
plot(t,y) %se grafica la funcion
xlabel('tiempo t')
ylabel('y')
title('edo1')
```

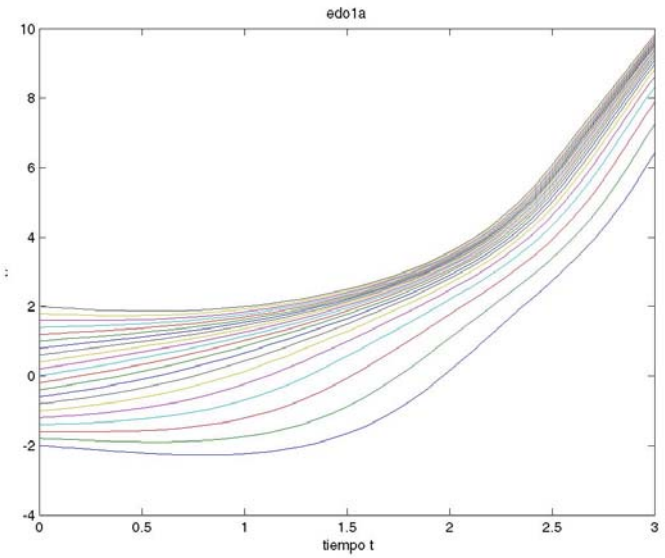
al ejecutar el archivo edo1.m en la ventana de comandos obtenemos la gráfica siguiente



En caso de que la función esté dada por f(t,y)= t.^2+cos(y) debemos escribir un m-archivo tipo function que llamaremos **f1a.m** defina esta función y si además queremos graficar distintas soluciones variando la condición inicial $y(0) = y_0$ con $y_0 \in [-2, 2]$, debemos escribir otro m-archivo tipo **script,** que llamaremos **edo1a.m** y que está dado por

```
y0vect=[-2:0.2:2]' %y0 es ahora un arreglo con las variaciones
%pedidas en la condicion inicial, es decir y0=[-2, -1, 0, 1, 2, 3,...,15]
odeset('RelTol',1e-4,'AbsTol',[1e-5]);
for y0=y0vect
   [t,y]=ode45(@f1a,[0 3],y0);
   plot(t,y);
   xlabel(' tiempo t')
   ylabel('y')
   title('edo1a')
   hold on
end;
hold off
```

Al ejecutar el archivo edo1a en la ventana de comandos obtenemos la gráfica suguiente.



Example: consider the van der Pol equation:

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0$$

Let's choose $y_1 = x$ and $y_2 = dx/dt$

canonical form:

$$\dot{y}_1 = y_2$$

$$\dot{y}_2 = \mu(1 - y_1^2)\, y_2 - y_1$$

```
function ydot = vdpolfun(t,y)
mu = 2;
ydot = [y(2) ; mu*(1-y(1)^2)*y(2)-y(1)];
```

Save this file with the same name as the name of the function ('vdpolfun.m' here).
To solve the ODE, use the following in a new M-file (vdpol.m):

```
tspan = [0 20];     % time span to integrate over
y0 = [2; 0];        % initial conditions (must be a
column)
t, y] = ode45('vdpolfun', tspan, y0);
plot(t, y(:,1), t, y(:,2));
xlabel('time');
ylabel('y1, y2');
legend('y1','y2');
```

You can also identify the specific solution time points desired by adding them to `tspan`.
For example:

```
tspan = linspace(0,20,100);
[t, y] = ode45('vdpolfun', tspan, y0);
```

ODE File Options

To accommodate μ as a parameter, we need to change the ODE file as follows:

```
function ydot = vdpolfun(t,y,flag,mu)
ydot = [y(2) ; mu*(1-y(1)^2)*y(2)-y(1)];
```

The differential equations is then solved using the syntax:

```
tspan = [0 20];     % time span to integrate over
y0 = [2; 0];        % initial conditions (must be a
column)   mu = 10;
ode45('vdpol',tspan,y0,[],mu)
```

For μ = 10, ode45 works fine. Now, solve the van der Pol equation for mu = 100 and for a time span from 0 to 3000. What happens? What do you propose to solve this equation?

Exercises

Exercise 1 (Reactions in series):

Using Matlab, find the concentrations of A, B, and C for the following reactions:

$$A \xrightarrow{k_1} B \xrightarrow{k_2} C$$

Initial conditions: A = 10, B = 0, C = 0

a) Use $k_1 = 5$ and $k_2 = 3$, for the time between 0 and 2
b) k_1 and k_2 to be entered by the user
c) compare graphically with the analytical solution (use symbols to plot the analytical solutions)

Exercise 2 (Reversible reactions and temperature dependence):

Consider the following reaction: $A \underset{k_2}{\overset{k_1}{\rightleftharpoons}} B$, $k_1 = 4$ $k_2 = 1$

$C_A(0) = 100$ $C_B(0) = 0$

a) Determine the differential equations for C_A and C_B.

b) Integrate these differential equations for the time between 0 and 5. Plot the results. Compare with the analytical solution.

c) We decide to add a temperature ramp: $T = 298 + 100 * t$

$k_1 = 100 \exp(-8000/RT)$; $k_2 = 10 \exp(-5700/RT)$

Solve for C_A and C_B. Plot the results.

Sintaxis

- `[T,Y] = rutina(odefun,tspan,y0)`
- `[T,Y] = rutina(odefun,tspan,y0,opciones)`
- `[T,Y] = rutina(odefun,tspan,y0,opciones,p1,p2...)`
- `[T,Y,TE,YE,IE] = rutina(odefun,tspan,y0,opciones)`
- `sol = rutina(odefun,[t0 tf],y0...)`
-

donde rutina puede ser: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

Argumentos

`Odefun`	A function that evaluates the right-hand side of the differential equations. All solvers solve systems of equations in the form $y' = f(t, y)$ or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`Tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0`	A vector of initial conditions.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.
`p1,p2...`	Optional parameters that the solver passes to `odefun` and all the functions specified in `options`..

Descripción

`[T,Y] = rutina(odefun,tspan,y0)` con `tspan = [t0 tf]` integra el sistema de ecuaciones diferenciales $y' = f(t, y)$ del tiempo `t0` al tiempo `tf` con condiciones iniciales `y0`.

function `f = odefun(t,y)`, para un escalar `t` y un vector columna `y`, debe regresar un vector columna `f` correspondiente a $f(t, y)$. Cada renglón en el erreglo solución corresponde aun tiempo enel vector columna `T`. Para obtener soluciones en tiempos específicos `t0`, `t1`, `...`, `tf` (crecientes o decrecientes), se debe usar `tspan = [t0,t1,...,tf]`.

`[T,Y] = rutina(odefun,tspan,y0,opciones)` resuelve de la maneraanterior pero los parámetros de integración son reemplazados por property values especificados en las `opciones`, un argumento creado con la función `odeset`. Las propiedades comunmente usadas incluyen error relativo escalar `RelTol` (`1e-3` por default) y un vector de error absoluto `AbsTol` (todos los componentes son `1e-6` por default). Vea `odeset` para más detalles.

`[T,Y] = solver(odefun,tspan,y0,options,p1,p2...)` solves as above, passing the additional parameters `p1,p2...` to the function `odefun`, whenever it is called. Use `options = []` as a place holder if no options are set.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x` Steps chosen by the solver.

`sol.y` Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.

`sol.solver` Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe` Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.

`sol.ye` Solutions that correspond to events in `sol.xe`.

`sol.ie` Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimnesional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property '`on`' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j)` = 1 if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass`

property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.
- If the mass matrix depends weakly on y, set `MStateDependence` to '`weak`' (the default) and otherwise, to '`strong`'. In either case, the function `MASS` is called with the two arguments (`t`,`y`).

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t, y)$.
- Supply the sparsity pattern of $\partial f / \partial y$ using the `JPattern` property or a sparse $\partial f / \partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t, y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the (`i`,`k`) component of $M(t, y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t, y)y' = f(t, y)$ is a differential algebraic equation. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that

$M(t_0, y_0)yp_0 = f(t_0, y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Rutina	**Tipo de problema**	**Precisión**	**Cuando usarlo**
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	If using crude error tolerances or solving

			moderately stiff problems.
`ode113`	Nonstiff	Low to high	If using stringent error tolerances or solving a computationally intensive ODE file.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	If the problem is only moderately stiff and you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See Algorithms for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and Improving ODE Solver Performance in the "Mathematics" section of the MATLAB documentation.

Parameters	**ode45**	**ode23**	**ode113**	**ode15s**	**ode23s**	**ode23t**	**ode23tb**
`RelTol`, `AbsTol`, `NormControl`	√	√	√	√	√	√	√
`OutputFcn`, `OutputSel`, `Refine`, `Stats`	√	√	√	√	√	√	√
`Events`	√	√	√	√	√	√	√
`MaxStep`, `InitialStep`	√	√	√	√	√	√	√
`Jacobian`, `JPattern`, `Vectorized`	--	--	--	√	√	√	√

`Mass`
`MStateDependence`
`MvPattern`
`MassSingular`
	√
	√
	--
	--
		√
		√
		--
		--
			√
			√
			--
			--
				√
				√
				√
				√
					√
					--
					--
					--
						√
						√
						√
						√
							√
							√
							√
							--

`InitialSlope`	--	--	--	√	--	√	--
`MaxOrder`, `BDF`	--	--	--	√	--	--	--

Un ejemplo de un sistema no rígido está dado por la descripción de un movimiento rígido sin fuerzas externas

$$
\begin{aligned}
y'_1 &= y_2\, y_3 & y_1(0) &= 0 \\
y'_2 &= -y_1\, y_3 & y_2(0) &= 1 \\
y'_3 &= -0.51\, y_1\, y_2 & y_3(0) &= 1
\end{aligned}
$$

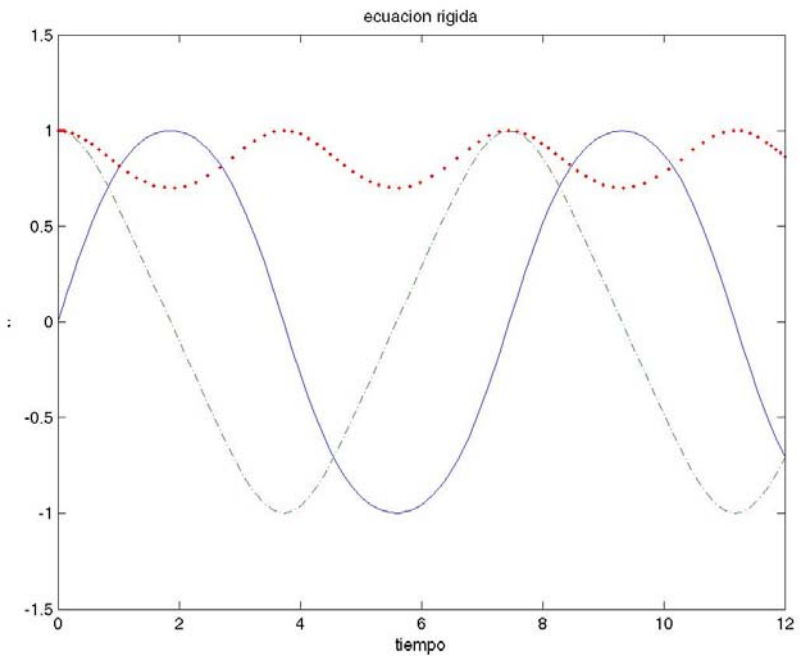
Para ello creamos un archivo tipo función que defina el sistema de ecuaciones diferenciales y llamaremos rigido.m

```
function dy = rigido(t,y)
    dy = zeros(3,1);    % a column vector
    dy(1) = y(2) * y(3);
    dy(2) = -y(1) * y(3);
    dy(3) = -0.51 * y(1) * y(2);
```

Cambiamos la tolerancia al error usando el comando odeset y resolvemos en intervalo de tiempo [0 12] con una condiciones vectoriales [0 1 1] al tiempo 0, usamos un m-archivo tipo script llamado rigidoa.m dado por

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
        [T,Y] = ode45(@rigido,[0 12],[0 1 1],options);
        plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
   xlabel('tiempo')
   ylabel('y')
   title('ecuacion rigida')
```

Obtenemos la gráfica



Un ejemplo de un sistema no rígido está dada por uan función de tipo Van der Pol con oscilaciones con relajación. El ciclo límite tiene partes donde las components de la solución cambian muy despacio y el sistema es rígido, pero hay partes donde hay cambios severos en las componentes en tiempos muy pequeños, donde el sistema no es rígido.

$$y'_1 = y_2 \qquad y_1(0) = 0$$
$$y'_2 = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 1$$

Para poder similar este sistema de ecuaciones debemos definir un m-archivo tipo function que contenga las ecuaciones diferenciales.

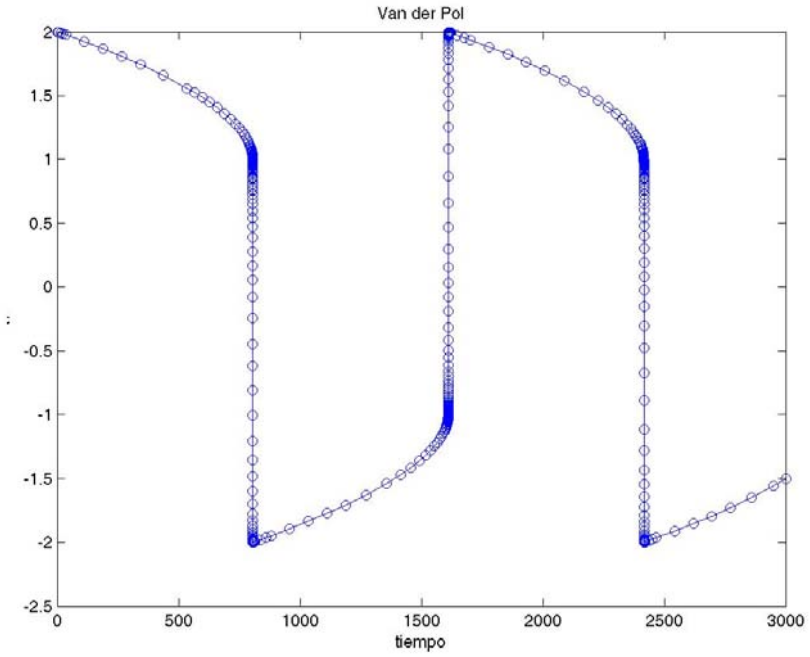
```
function dy = vdp1000(t,y)

dy = zeros(2,1);    % a column vector

dy(1) = y(2);

dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

%For this problem, we will use the default relative and absolute tolerances (1e-3 and 1e-6, %respectively) and solve on a time interval of [0 3000] with initial condition vector [2 0] %al tiempo 0

Y se crea un m-archivo tipo script, que llamamos VderPlo.m

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
    plot(T,Y(:,1),'-o')
xlabel('tiempo')
ylabel('y')
title('Van der Pol')
```

Con lo que obtenemos la gráfica siguiente



deval, odeset, odeget, @ (function handle)

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp 436-451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp 1-9.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp 19-26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp 1-22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp 538-552.

nzmax odefile