

BREVES APUNTES DE MATLAB

UNA INTRODUCCIÓN RÁPIDA PERO NO TRIVIAL



MATLAB es un potente paquete de software para computación científica, orientado al cálculo numérico, a las operaciones matriciales y especialmente a las aplicaciones científicas y de ingeniería.

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

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

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

1. DOCUMENTACIÓN Y AYUDA ON-LINE

- Ayuda on-line en la ventana de comandos:
`>> help nombre_de_comando`
- Ayuda on-line en la ventana del navegador: "Help" en la barra de menús, ó bien:
`>> helpwin`

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

2. SCRIPTS Y FUNCIONES. EL EDITOR INTEGRADO

2.1 Scripts

Un **script** es un conjunto de instrucciones (de cualquier lenguaje) guardadas en un fichero (usualmente de texto) que son ejecutadas normalmente mediante un intérprete. Son útiles para automatizar pequeñas tareas. También puede hacer las veces de un "programa principal" para ejecutar una aplicación.

Así, para llevar a cabo una tarea, en vez de escribir las instrucciones una por una en la línea de comandos de MATLAB, se pueden escribir una detrás de otra en un fichero. Para ello se puede utilizar el **Editor integrado**: icono "**hoja en blanco**" del menú de herramientas, opción "**New M-file**" del Menú "**File**" o bien usando la orden

`>> edit`

Los scripts de MATLAB deben guardarse en un fichero con sufijo **.m** para ser reconocidos.

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

2.2 Funciones

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

Las funciones MATLAB responden al siguiente formato de escritura (la cláusula **end** del final no es obligatoria, excepto en el caso de funciones anidadas, que se explica más adelante):

```
function [argumentos de salida] = nombre(argumentos de entrada)
% comentarios
%
-----
instrucciones (normalmente terminadas por ; para evitar eco en
pantalla)
-----
end (opcional salvo en las funciones anidadas)
```

Las funciones deben guardarse en un fichero con el mismo nombre que la función y sufijo **.m**. Se pueden incluir en el mismo fichero otras funciones, denominadas **subfunciones**, a continuación de la primera, pero sólo serán “visibles” para las funciones del mismo fichero.

También es posible definir funciones anidadas, esto es, funciones “insertadas” dentro del código de otras funciones. (Se exponen aquí para conocer su existencia. Su utilización es delicada.)

EJEMPLO : fichero tresxcuadrado.m

```
function z=tresxcuadrado(x)
% Ejemplo de una funcion anidada
%
%%%%%%%%%
function [y]=cuadrado(x)
y=x*x;
end
%%%%%%%%%
z=3*cuadrado(x);
end
```

2.3 Funciones on-line (anónimas)

Algunas funciones “sencillas”, que devuelvan el resultado de una expresión, se pueden definir mediante una sola instrucción, en mitad de un programa (script o función) o en la línea de comandos. Se llaman **funciones anónimas**:

handle = @(argumentos) expresion

EJEMPLO

```
>> mifun = @(x) sin(2*x);
>> mifun(pi/4)
ans =
1
```

Pueden tener varias variables y hacer uso de variables previamente definidas:

EJEMPLO

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

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

2.4 Workspace y ámbito de las variables

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

Las variables creadas desde la línea de comandos de MATLAB pertenecen al **base workspace** (espacio de trabajo base). Lo mismo sucede con las variables creadas por **scripts** que se ejecutan desde la línea de comandos. Estas variables permanecen en el **base workspace** cuando se termina la ejecución del **script** y se mantienen allí durante toda la sesión de trabajo o hasta que se borren.

Sin embargo, las variables creadas por una **función** pertenecen al **espacio de trabajo de dicha función**, que es independiente del espacio de trabajo base. Es decir, las variables de las funciones son **LOCALES**: MATLAB reserva una zona de memoria cuando comienza a ejecutar una función, almacena en esa zona las variables creadas por esa función, y “borra” dicha zona cuando termina la ejecución de la función.

Para hacer que una variable de una función pertenezca al **base workspace**, hay que declararla **GLOBAL**: la orden

global a suma error

en una función hace que las variables **a** , **suma** y **error** pertenezcan al **base workspace**.

Sin embargo, las variables que se crean en una función son locales y temporales, es decir, no son accesibles desde otros programas, y no “viven” en el *Workspace*: MATLAB reserva una zona de memoria cuando comienza a ejecutar una función, almacena en esa zona las variables creadas por esa función, y “borra” dicha zona cuando termina la ejecución de la función.

Esta es una de las principales diferencias entre los scripts y las funciones: cuando finaliza la ejecución de un script se puede “ver” y utilizar el valor de todas las variables que ha creado el script en el *Workspace*; en cambio, cuando finaliza una función no hay rastro de sus variables en el *Workspace* (salvo las que se hayan declarado globales).

Otras diferencias son que, como ya se ha dicho, los scripts no pueden tener argumentos de entrada ni de salida, y que tampoco pueden contener subfunciones.

3. OBJETOS Y SINTAXIS BÁSICOS

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

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

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

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

Algunas **constantes** numéricas están **predefinidas**:

i , j	unidad imaginaria : 2+3i -1-2j
pi	número π
Inf	"Infinito", número mayor que el más grande que se puede almacenar. Se produce con operaciones como x/0 , con x\neq 0
NaN	"Not a Number" : magnitud no numérica resultado de cálculos indefinidos. Se produce con cálculos del tipo 0/0 o Inf/Inf . (0+2i)/0 da como resultado NaN + Inf i

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

MATLAB distingue entre mayúsculas y minúsculas: **Log** no es lo mismo que **log**.

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

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

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

3.1 Constantes y operadores aritméticos

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

3.2 Funciones elementales

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

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

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

Help → MATLAB → Functions-Categorical List → Mathematics → Elementary Math

Algunas de ellas son:

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

3.3 Uso como calculadora

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

EJEMPLO

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

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

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

3.4 Variables

En MATLAB las variables no son nunca declaradas: su tipo y su tamaño cambian de forma dinámica de acuerdo con los valores que le son asignados. Así, una misma variable puede ser utilizada, por ejemplo, para almacenar un número complejo, a continuación una matriz 25x40 de números enteros y luego para almacenar un texto. Las variables se crean automáticamente al asignarles un contenido. Asimismo, es posible eliminar una variable de la memoria si ya no se utiliza.

EJEMPLOS

```
>> a=10
a =
    10.

>> pepito=exp(2.4/3)
pepito =
    2.2255

>> pepito=a+pepito*(4-0.5i)
pepito =
    18.9022 - 1.1128i

>> clear pepito
```

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

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

Algunos comandos relacionados con la inspección y eliminación de variables son:

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

3.5 Formatos

Por defecto, MATLAB muestra los números en formato de punto fijo con 5 dígitos. Se puede modificar esto mediante el comando **format**:

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

3.6 Algunos comandos utilitarios

Están disponibles algunos comandos utilitarios, como:

ls	Lista de ficheros del directorio actual (como Unix)
dir	Lista de ficheros del directorio (de otra forma)
pwd	Devuelve el nombre y path del directorio actual
cd	Para cambiar de directorio
clc	"Limpia" la ventana de comandos
date	Fecha actual

3.7 Algunas observaciones sobre las funciones de MATLAB

1. Las explicaciones sobre las funciones/comandos que se presentan en estas notas están muy resumidas y sólo incluyen las funcionalidades que, según el parecer subjetivo de la autora, pueden despertar más interés. La mayoría de las funciones tienen mas y/o distintas funcionalidades que las que se exponen aquí. Para una descripción exacta y exhaustiva es preciso consultar la Ayuda on-line.

4. MATRICES

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

4.1 Construcción de matrices

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

EJEMPLOS (construcciones elementales de matrices)

```
>> v=[1,-1,0,sin(2.88)]           % vector fila longitud 4
>> w=[0;1.003;2;3;4;5*pi]         % vector columna longitud 6
>> a=[1,2,3,4;5,6,7,8;9,10,11,12] % matriz 3x4
>> mat=['Hola','Mari';'¿Como','estas?'] % matriz 2x2 de caracteres
```

Observaciones:

- Lo que se escribe en cualquier línea detrás de % es considerado como comentario.
- El hecho de que, al introducir las matrices, se escriban las matrices por filas no significa que internamente, en la memoria del ordenador, estén así organizadas: en la memoria las matrices se almacenan como un vector unidimensional ordenadas por columnas, **como siempre**.

Otras órdenes para crear matrices son:

```
>> v1=a:h:b % crea un vector fila de números desde a hasta un número c <= b
              % tal que c+h > b, con incrementos de h
>> v2=a:b    % como el anterior, con paso h=1
>> v3=v2'    % matriz traspuesta (conjugada si es compleja)
>> v4=v2.'   % matriz traspuesta sin conjugar
```

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

EJEMPLOS (construcciones elementales de matrices)

```
>> v1=1:4
>> v2=[v1,5;0.1:0.1:0.5]
>> v3=[v2',[11,12,13,14,15]']
```

$$v1 = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \quad v2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 0.1 & 0.2 & 0.3 & 0.4 & 0.5 \end{pmatrix} \quad v3 = \begin{pmatrix} 1 & 0.1 & 11 \\ 2 & 0.2 & 12 \\ 3 & 0.3 & 13 \\ 4 & 0.4 & 14 \\ 5 & 0.5 & 15 \end{pmatrix}$$

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

Help → MATLAB → Functions-Categorical List → Mathematics → Arrays and Matrices

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

linspace(a,b,n)	Si a y b son números reales y n un número entero, genera una partición regular del intervalo [a,b] con n nodos (n-1 subintervalos)
linspace(a,b)	Como el anterior, pero se asume n=100
logspace(e,f,n)	Vector con n elementos logarítmicamente espaciados desde 10^e hasta 10^f , es decir, cuyos logaritmos están regularmente espaciados. (logspace(e,f,n) = 10.^linspace(e,f,n))
logspace(e,f)	Como el anterior, pero se asume n=50

diag(v)	Si v es un vector, diag(v) es una matriz cuadrada de ceros con diagonal principal = v
diag(A)	Si A es una matriz, diag(A) es un vector = diagonal principal de A
diag(A,k)	Si A es una matriz y k es un entero, diag(A,k) es un vector = k -ésima sub o super diagonal de A (según sea k<0 ó k>0)
blkdiag(A,...,K)	Construye una matriz diagonal por bloques con las matrices A, ... K
triu(A) tril(A)	Extrae la parte triangular superior (inferior) de la matriz A
triu(A,k) tril(A,k)	Extrae la parte superior (inferior) de la matriz A , desde la k -ésima diagonal hacia arriba (abajo)
zeros(n,m)	matriz nxm con todas sus componentes iguales a cero.
ones(n,m)	matriz nxm con todas sus componentes iguales a uno
eye(n,m)	matriz unidad: matriz nxm con diagonal principal =1 y el resto de las componentes =0
reshape(A,n,m)	Re-dimensiona una matriz: si A es una matriz h x k , reshape(A,n,m) es otra matriz con los mismos elementos que A , pero de dimensiones nxm (tiene que ser h*k=n*m)

rand(n) rand(n,m)	Construye una matriz nxn ó nxm con números aleatorios con distribución uniforme
randn(n) randn(n,m)	Ídem con distribución normal
compan(p)	Es la matriz compañera (ó de Frobenius) del vector p , es decir, que tiene como autovalores las raíces de p
magic(n)	Devuelve una matriz nxn que es un cuadrado mágico
pascal(n)	matriz de Pascal de dimensión nxn
vander(v)	matriz de Vandermonde asociada al vector v : a(i,j) = v(i)^(n-j)

EJEMPLOS

```
>> A=[eye(2,2),ones(2,3);linspace(1,2,5);zeros(1,5)]
>> w=diag(A)
>> B=reshape(A,5,4)
>> C=diag(diag(A,1))+diag(diag(B,-2),1)
```

$$A = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1.25 & 1.5 & 1.75 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad w = \begin{pmatrix} 1 \\ 1 \\ 1.5 \\ 0 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 1 & 1.5 & 0 \\ 0 & 1.25 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 1.75 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1.75 & 1.75 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

4.2 Operadores y funciones

Los operadores aritméticos representan las correspondientes operaciones matriciales siempre que tengan sentido. Cuando van precedidos de un punto deben entenderse en el sentido de que la operación se efectúa "elemento a elemento".

Sean: A y B matrices de elementos a_{ij} y b_{ij} y k un escalar	
A+B (A-B)	matriz de elementos a_{ij} + b_{ij} (a_{ij} - b_{ij}) (si las dimensiones son iguales)
A*B	producto matricial de A y B (si las dimensiones son adecuadas)
A+k	matriz de elementos a_{ij} + k
A-k	matriz de elementos a_{ij} - k
k*A	matriz de elementos k * a_{ij}
A/k =(1/k)*A	matriz de elementos a_{ij} / k

A^k	matriz A elevada a la potencia k : si k entero > 0 , $A^k = A * A * \dots * A$ si k entero < 0 , $A^k = (\text{inv}(A))^{(-k)}$ si no, A^k se calcula por diagonalización: $A^k = V * A.^k * \text{inv}(V)$, donde $[V, D] = \text{eig}(A)$
$k ./ A$	matriz de elementos k / a_{ij}
$A.^k$	matriz de elementos $(a_{ij})^k$
$k.^A$	matriz de elementos $k^{a_{ij}}$
$A.*B$	matriz de elementos $a_{ij} * b_{ij}$ (si dimensiones iguales)
$A./B$	matriz de elementos a_{ij} / b_{ij} (si dimensiones iguales)
$A.^B$	matriz de elementos $a_{ij}^{b_{ij}}$ (si dimensiones iguales)

La mayoría de las funciones MATLAB están hechas de forma que admiten matrices como argumentos. Esto se aplica en particular a las funciones matemáticas elementales y su utilización debe entenderse en el sentido de "elemento a elemento": si **A** es una matriz de elementos a_{ij} , **exp(A)** es otra matriz cuyos elementos son **exp(a_{ij})**. No debe confundirse con la función exponencial matricial que, a una matriz cuadrada **A**, asocia la suma de la serie exponencial matricial, y que en MATLAB se calcula mediante la función **expm**.

EJEMPLO (diferencia entre exp y expm)

```
>> A=[1,0;0,2]
>> B=exp(A)
>> C=expm(A)
```

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \quad B = \begin{pmatrix} 2.7182818 & 1 \\ 1 & 7.3890561 \end{pmatrix} \quad C = \begin{pmatrix} 2.7182818 & 0 \\ 0 & 7.3890561 \end{pmatrix}$$

Por otra parte, algunas funciones útiles en cálculos matriciales son:

sum(A)	suma de las componentes de la matriz A
sum(A,1)	es un vector fila conteniendo la suma de los elementos de cada columna de A
sum(A,2)	es un vector columna conteniendo la suma de los elementos de cada fila de A
trace(A)	traza de A : sum(diag(A))
prod(A)	producto de las componentes de la matriz A
prod(A,1), prod(A,2)	es un vector fila / columna conteniendo el producto de los elementos de cada columna / fila de A
max(v)	si v es un vector, máximo de sus componentes
max(A)	si A es una matriz, es un vector fila conteniendo el máximo elemento de cada columna.
mean(v), mean(A)	Como max, pero para la media
norm(v)	norma euclídea del vector v
norm(v,2)	
norm(v,p)	norma- p del vector v : sum(abs(v).^p)^(1/p)
norm(v,inf)	max(abs(v))
norm(v,-inf)	min(abs(v))
norm(A)	máximo autovalor de la matriz A
norm(A,2)	
norm(A,1)	norma-1 de la matriz A : máximo entre las sumas de sus columnas: max(sum(abs(A)))
norm(A,inf)	norma infinito de la matriz A : máximo entre las sumas de sus filas: max(sum(abs(A')))
size(A)	devuelve, en un vector fila, las dimensiones de la matriz A
length(A)	longitud: length(A) = max(size(A))

4.3 Manipulación de los elementos de una matriz. Extracción, inserción y eliminación

Para especificar los elementos de una matriz se usa la sintaxis:

$v(i)$	Si v es un vector es v_i
$A(i,j)$	Si A es una matriz, es a_{ij}
$A(k)$	Si A es una matriz, es el k -ésimo elemento de A , en el orden en que está almacenada en la memoria (por columnas)

Los subíndices, en MATLAB, siempre comienzan en 1.

Pero MATLAB posee un buen número de facilidades para designar globalmente un conjunto de elementos de una matriz o vector, consecutivos o no.

Por ejemplo, si v es un vector y h es un vector de subíndices, $v(h)$ hace referencia al subconjunto de componentes de v correspondientes a los valores contenidos en h . Análogamente con $A(h,k)$ si A es una matriz bidimensional y h y k son vectores de subíndices.

EJEMPLOS (referencias a subconjuntos de elementos de una matriz mediante vectores de subíndices)

```

>> v = [11,12,13,14,15,16,17,18,19,20]
>> ind = [2,5,1,8]
>> v(ind)
ans =
    12    15    11    18

>> A=[1.1,1.2,1.3;2.1,2.2,2.3;3.1,3.2,3.3]

>> A(2:3,1:2)      % submatriz de A formada por la intersección de las
                    % filas 2 y 3 con las columnas 1 y 2

```

$$A = \begin{pmatrix} 1.1 & 1.2 & 1.3 \\ 2.1 & 2.2 & 2.3 \\ 3.1 & 3.2 & 3.3 \end{pmatrix}$$

Recuérdese que $n:m$ es el vector $[n,n+1,n+2,\dots,m]$

El símbolo `:` (dos puntos) en el lugar de un índice indica que se toman todos. El símbolo `end` indica el último valor del subíndice.

EJEMPLOS

```

>> A(:,2)          % la segunda columna de A
>> A(:,2:end)      % las columnas de A desde la 2 hasta la última
>> A(:)            % todos los elementos de A, en una sola columna

```

Si B es un vector booleano (sus elementos son `true` y `false`, que MATLAB muestra como `1` y `0`) entonces $A(B)$ es la submatriz que se obtiene considerando o no cada elemento en función del valor verdadero o falso del vector booleano:

EJEMPLOS (referencias a subconjuntos de elementos de una matriz mediante vectores booleanos)

```

>> v=linspace(1,5,9);
>> b=[true,true,true,false,false,true,true,false,false]
b =
    1    1    1    0    0    1    1    0    0
>> w=v(b)

```

$$v = (1 \ 1.5 \ 2 \ 2.5 \ 3 \ 3.5 \ 4 \ 4.5 \ 5)$$

$$w = (1 \ 1.5 \ 2 \ 3.5 \ 4)$$

Esta sintaxis para designar conjuntos de elementos de una matriz puede usarse tanto para recuperar los valores que contienen (para, por ejemplo, utilizarlos en una expresión), como para asignarles valores. Cuando estas expresiones aparecen a la izquierda de un signo igual (es decir, en una instrucción de asignación) pueden tener distintos significados:

EJEMPLOS (asignación de valores a partes de una matriz y modificación de su dimensión) (se recomienda ejecutarlos para comprender sus efectos)

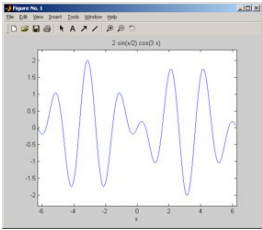
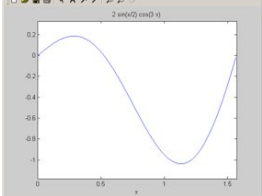
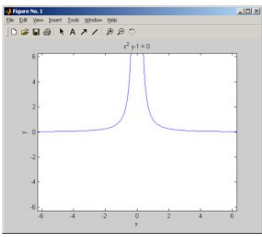
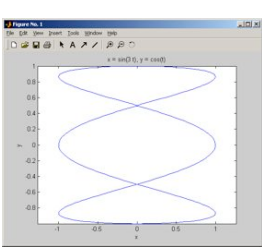
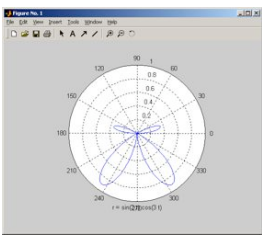
```
>> A=rand(4,4)    % Se almacena en A una matriz 4x4 de num. aleatorios
>> A(2,2)=0       % Se modifica el segundo elem. diagonal de A
>> A(5,2)=1       % Obsérvese que A(5,2) no existía:: de hecho, se
                  % MODIFICAN LAS DIMENSIONES de la matriz, para AÑADIR
                  % el elemento A(5,2). El resto se llena con ceros.
                  % Ahora A es una matriz 5x4
>> A(2:3,1:2)=1    % La submatriz A(2:3,1:2) se llena con unos
>> A(:,2)=[]       % El símbolo [] representa una "matriz vacía"
                  % Esta instrucción ELIMINA la segunda columna de A
                  % Ahora A tiene dimensión 5x3
>> A=[A,[1:5]']    % Se añade a A una nueva columna al final
```

5. REPRESENTACIÓN GRÁFICA DE FUNCIONES DEFINIDAS POR UNA FÓRMULA

Los comandos que se presentan en este apartado son funciones MATLAB "fáciles de usar" (easy-to-use) para representar gráficamente, de forma rápida, funciones definidas por una expresión matemática. Tienen sólo un pequeño número de parámetros que se pueden especificar. Todas ellas hacen uso de otras funciones MATLAB que disponen de un número superior de opciones y parámetros que podemos modificar. Cuando se necesite hacer gráficas de funciones que no vienen definidas por una fórmula (definidas a trozos, definidas a través de programas o por sus valores en un número finito de puntos, ...) habrá que recurrir a dichas funciones más generales.

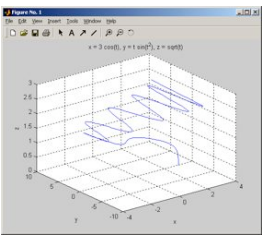
5.1 Curvas planas

El comando más fácil de usar de que dispone MATLAB para dibujar curvas planas definidas por una fórmula matemática (no por un conjunto de valores) es el comando **ezplot**, que puede ser usado de varias formas.

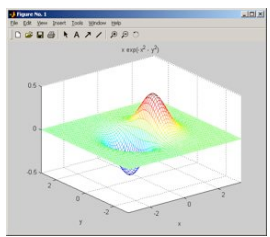
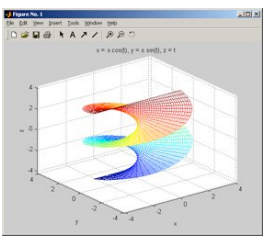
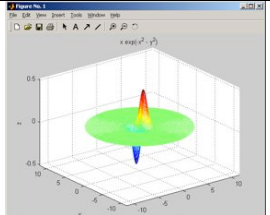
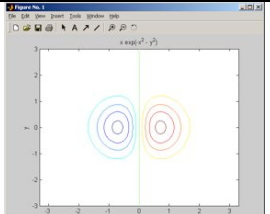
<p>ezplot(f)</p>	<p>donde</p> <ul style="list-style-type: none"> f es una cadena de caracteres conteniendo la expresión de una función y=f(x) <p>dibuja la función y=f(x) para x en el intervalo [-2π,2π]</p> <p>Ejemplo:</p> <pre>>> ezplot('2*sin(x/2)*cos(3*x)')</pre>	
<p>ezplot(f,[a,b])</p>	<p>lo mismo que la anterior para x variando en el intervalo [a,b]</p> <p>Ejemplo:</p> <pre>>>ezplot('2*sin(x/2)*cos(3*x)',[0,pi/2])</pre>	
<p>ezplot(f) ezplot(f,[a,b])</p>	<p>si f es una expresión de (x,y), dibuja la curva implícitamente definida por f(x,y)=0, para x e y variando en el intervalo [-2π,2π] en el primer caso y para x e y variando en el intervalo [a,b] en el segundo caso.</p> <p>Ejemplo:</p> <pre>>> ezplot('x^2*y-1')</pre>	
<p>ezplot(x,y) ezplot(x,y,[a,b])</p>	<p>donde</p> <ul style="list-style-type: none"> x e y son dos cadenas de caracteres conteniendo las expresiones de dos funciones x(t) e y(t) <p>dibuja la curva de ecuaciones paramétricas x=x(t) y=y(t) para t en el intervalo [0,2π], en el primer caso y para t en el intervalo [a,b] en el segundo</p> <p>Ejemplo:</p> <pre>>> ezplot('sin(3*t)','cos(t)')</pre>	
<p>ezpolar(f) ezpolar(f,[a,b])</p>	<p>donde</p> <ul style="list-style-type: none"> f es una cadena de caracteres conteniendo la expresión de una función f(θ) <p>dibuja la curva definida en coordenadas polares por ρ= f(θ) para θ variando en el intervalo [0,2π], en el primer caso y en el intervalo [a,b] en el segundo</p> <p>Ejemplo:</p> <pre>>> ezpolar('sin(2*t)*cos(3*t)',[0,pi])</pre>	

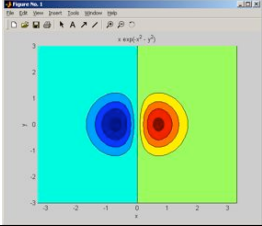
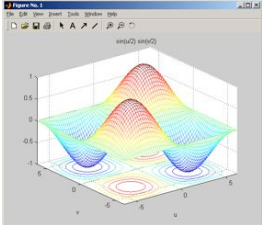
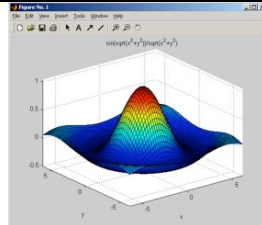
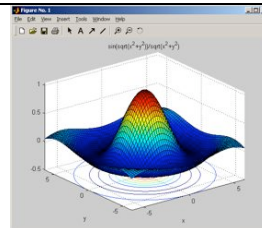
5.2 Curvas en el espacio

Para dibujar curvas en el espacio tridimensional, MATLAB dispone del comando **ezplot3**:

<pre>ezplot3(x,y,z) ezplot3(x,y,z,[a,b])</pre>	<p>donde</p> <ul style="list-style-type: none"> x, y, z son tres cadenas de caracteres conteniendo las expresiones de tres funciones $x(t)$, $y(t)$, $z(t)$ <p>dibuja la curva de ecuaciones paramétricas $x=x(t)$, $y=y(t)$, $z=z(t)$ para t en el intervalo $[0, 2\pi]$, en el primer caso y para t en el intervalo $[a, b]$ en el segundo</p> <p>Ejemplo:</p> <pre>>>ezplot3('3*cos(t)','t*sin(t^2)','sqrt(t)')</pre>	
--	---	---

5.3 Superficies

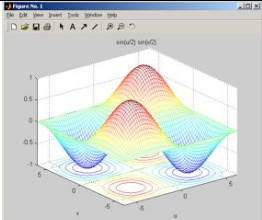
<pre>ezmesh(f) ezmesh(f,[a,b]) ezmesh(f,[a,b,c,d])</pre>	<p>donde</p> <ul style="list-style-type: none"> f es una expresión de dos variables <p>dibuja la superficie $z=f(x,y)$ para (x,y) variando en el cuadrado $[-\pi,\pi] \times [-\pi,\pi]$ en el primer caso, en el cuadrado $[a,b] \times [a,b]$ en el segundo, y en el rectángulo $[a,b] \times [c,d]$ en el tercer caso. El método de dibujo es una malla con segmentos coloreados, en función de los valores en los extremos.</p> <p>Ejemplo:</p> <pre>>> ezmesh('x*exp(-x^2 - y^2)')</pre>	
<pre>ezmesh(x,y,z) ezmesh(x,y,z,[a,b]) ezmesh(x,y,z,[a,b,c,d])</pre>	<p>donde</p> <ul style="list-style-type: none"> x, y, z son expresiones de funciones de dos variables <p>dibuja la superficie de coordenadas paramétricas $x=x(s,t)$, $y=y(s,t)$, $z=z(s,t)$ para (s,t) variando en el cuadrado $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$ en el primer caso, en el cuadrado $[a,b] \times [a,b]$ en el segundo, y en el rectángulo $[a,b] \times [c,d]$ en el tercer caso</p> <p>Ejemplo:</p> <pre>>> ezmesh('s*cos(t)','s*sin(t)','t',[-pi,pi])</pre>	
<pre>ezmesh(..., 'circ')</pre>	<p>en cualquiera de los usos anteriores, dibuja la función correspondiente sobre un círculo centrado en el origen</p> <p>Ejemplo:</p> <pre>>> ezmesh('x*exp(-x^2 - y^2)', 'circ')</pre>	
<pre>ezcontour(f) ezcontour(f,[a,b]) ezcontour(f,[a,b,c,d])</pre>	<p>dibuja las líneas de nivel (isovalores) de la función $z=f(x,y)$</p> <p>Ejemplo:</p> <pre>>> ezcontour('x*exp(-x^2 - y^2)')</pre>	

ezcontourf(. . .)	<p>hace lo mismo que ezcontour, pero rellenando con un color sólido las distintas zonas determinadas por las líneas de nivel</p> <p>Ejemplo:</p> <pre>>> ezcontourf('x*exp(-x^2 - y^2)')</pre>	
ezmeshc(f)	<p>con los mismos argumentos que ezmesh, dibuja simultáneamente las líneas de nivel y la superficie</p> <p>Ejemplo:</p> <pre>>> ezmeshc('sin(u/2)*sin(v/2)')</pre>	
ezsurf(f)	<p>dibuja una superficie coloreada z=f(x,y). Sus argumentos son como en ezmesh</p> <p>Ejemplo:</p> <pre>>> ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)')</pre>	
ezsurfc(f)	<p>como la anterior pero, además, dibuja las líneas de nivel</p> <p>Ejemplo:</p> <pre>>>ezsurfc('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)')</pre>	

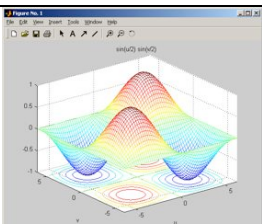
En los comandos expuestos en este apartado aparecen los argumentos **f**, **x**, **y** ó **z**, que representan las funciones a dibujar. En todos los ejemplos se ha escrito directamente la expresión en la llamada a la función **ez***** correspondiente, como una cadena de caracteres.

No obstante, también se pueden suministrar dichas funciones de otras dos formas:

1. Mediante un "handle" (manejador) de una función anónima (véase [apartado 2.3](#))

ezmeshc(f)	<p>en vez de</p> <pre>>> ezmeshc('sin(u/2)*sin(v/2)')</pre> <p>se podría haber escrito:</p> <pre>>> fun=@(u,v) sin(u/2)*sin(v/2) >> ezmeshc(fun)</pre>	
-------------------	--	---

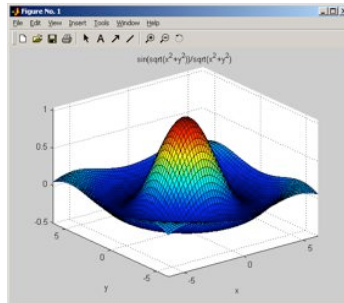
2. Mediante un "handle" de una M-función escrita en un fichero (véase [apartado 2.2](#))

ezmeshc(f)	<pre>>> ezmeshc(@fun)</pre> <p>siendo fun la M-función, almacenada en el fichero fun.m, siguiente:</p> <pre>function [z]=fun(u,v) z= sin(u/2)*sin(v/2);</pre>	
-------------------	---	---

5.4 Algunos comandos gráficos auxiliares

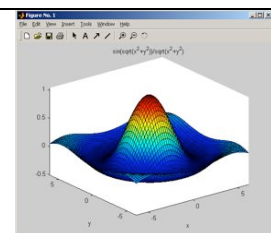
Se ven aquí algunos comandos que modifican el aspecto de un gráfico. En estas notas no se exponen todas las posibilidades de estos comandos. Utilizando el **help** de MATLAB, se pueden ver el resto de las opciones. Para mostrar el efecto de los comandos se utilizará la siguiente figura, creada con la orden

```
>> ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)')
```



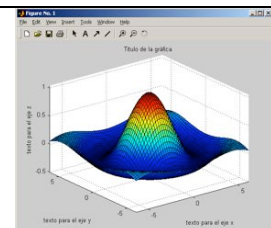
```
grid on
grid off
```

muestra – esconde la cuadrícula

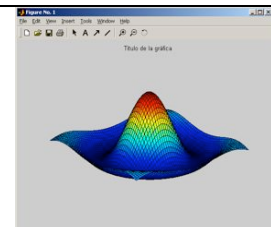


```
xlabel('Etiqueta para el eje x')
ylabel('Etiqueta para el eje y')
zlabel('Etiqueta para el eje z')

title('Título de la gráfica')
```



```
axis on
axis off
```

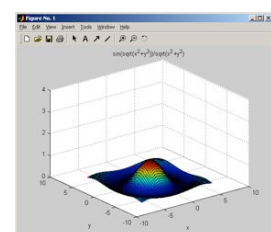


```
axis([x1,x2,y1,y2,z1,z2])
```

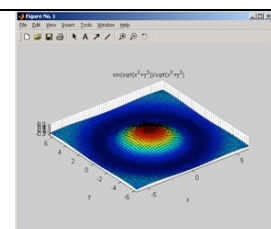
Determina los límites de la gráfica. En gráficos bi-dimensionales no se incluyen $z1$, $z2$.

axis auto
impone los límites establecidos por defecto por MATLAB, determinados en función de los límites de la gráfica actual

Ejemplo:
`axis([-10,10,-10,10,0,4])`



axis equal
determina los mismos factores de escala para todos los ejes (esto haría, por ejemplo, que una circunferencia se viera redonda en vez de elíptica...)

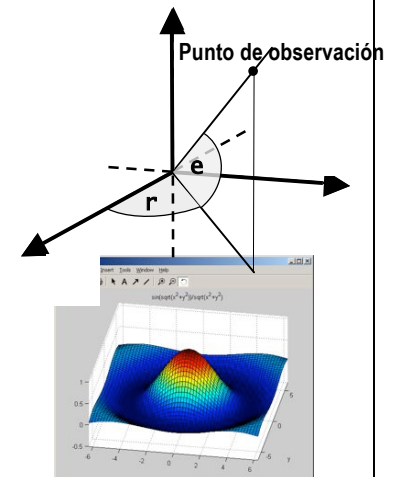


view(r,e)

en las gráficas tridimensionales permite cambiar el punto de observación del objeto representado.

Los valores por defecto son $r=-37.5$, $e=30$

>> view(10,50)



shading

determina la forma de utilizar los colores de las superficies coloreadas:

shading flat

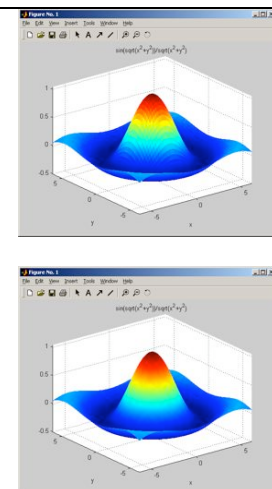
la función color es constante a trozos (en cada segmento o panel)

shading interp

la función color es bilineal a trozos (en cada segmento o panel)

shading faceted (es la opción por defecto)

la función color es constante a trozos (como en flat), pero además se superpone el dibujo de la malla con los segmentos en negro



hold on hold off	Todas las órdenes de dibujo entre ambos comandos se ejecutan sobre la misma ventana gráfica, sin borrar lo anterior.
figure figure(h)	Crea una nueva ventana gráfica, la activa y la trae al frente, delante de todas las ventanas abiertas. En el primer caso le asigna un número de forma automática. En el segundo caso le asigna el número h, es decir, el nombre "Figure No. h"
gcf	Devuelve el número de la ventana gráfica activa en ese momento.
shg	Trae la ventana gráfica activa al frente de todas.
clf	Borra la figura de la ventana gráfica activa. No cierra la ventana; sólo borra su contenido.
close close(h)	Cierra la ventana gráfica activa, en el primer caso, o la de número h, en el segundo.

subplot(m,n,p)

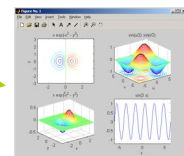
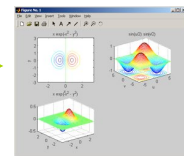
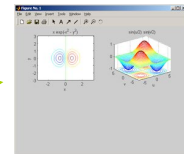
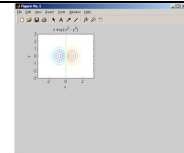
Este comando permite dividir la ventana gráfica en una matriz $m \times n$ de sub-ventanas gráficas, activando para dibujar la p -ésima de ellas. Ver ejemplo siguiente:

```
>> subplot(2,2,1); ezcontour('x*exp(-x^2 - y^2)');
```

```
>> subplot(2,2,2); ezmeshc('sin(u/2)*sin(v/2)');
```

```
>> subplot(2,2,3); ezmesh('x*exp(-x^2 - y^2)');
```

```
>> subplot(2,2,4); ezplot('sin(3*x)');
```



6. SISTEMAS LINEALES. AUTOVALORES Y AUTOVECTORES

Para la resolución "rápida" de sistemas lineales MATLAB dispone del "operador" `\` (backslash):

Si **A** es una matriz cuadrada **nxn** no singular y **b** es un vector columna de longitud **n**, entonces

```
>> x = A\b;
```

calcula la solución del sistema lineal de ecuaciones **Ax=b**. Para recordar la sintaxis, debe asimilarse el operador `\` con una "división por la izquierda", es decir **A\b** es como **A⁻¹b**.

Este operador debe ser usado con precaución: Si la matriz **A** es singular o mal condicionada, la instrucción **A\b** emite un mensaje de advertencia, pero devuelve en cualquier caso un resultado, que puede ser disparatado.

Si la matriz **A** no es cuadrada (teniendo el mismo número de líneas que el segundo miembro) entonces la instrucción **A\b** devuelve, sin advertir absolutamente nada, una solución de mínimos cuadrados:

$$\|Ax - b\| = \min_{y \in \mathbb{R}^n} \|Ay - b\|_2$$

En el caso en que la solución de mínimos cuadrados no es única (la matriz **A** no es de rango máximo), la solución que devuelve **NO** es la de norma mínima. Para obtener esta última es mejor utilizar la instrucción siguiente, que utiliza la pseudo-inversa de la matriz **A**:

```
>> pinv(A)*b
```

Los autovalores de una matriz se pueden calcular mediante la función:

```
>> eig(A)
```

Si se desean obtener, además, los vectores propios, hay que llamar a la función **spec** de la siguiente forma:

```
>> [V,D]=eig(A)
```

mediante la cual se obtienen **D**: matriz cuadrada diagonal con los autovalores y **V**: matriz cuadrada invertible cuya *i*-ésima columna es un vector propio asociado al *i*-ésimo autovalor.

Otras funciones de interés pueden ser:

A/B	El operador / (slash) representa la "división por la derecha": A/B es como A*B⁻¹ . Luego X= A*B⁻¹ es la solución de X*B=A . Por esta razón, si k es un escalar y A es una matriz, A/k es A*k⁻¹ = A*(1/k)
det(A)	determinante de la matriz cuadrada A Atención: este cálculo se lleva a cabo a partir de los factores triangulares LU que se obtienen por eliminación gaussiana. Por ello, si la matriz es "mal condicionada", los errores de redondeo pueden producir respuestas patológicas.
rank(A)	rango de la matriz A (método basado en descomposición SVD)
cond(A)	Número de condición de la matriz A en norma 2: λ_{max} / λ_{min} (Si es muy grande, la matriz está mal condicionada)
rcond(A)	Inverso del número de condición de la matriz A en norma 1: (Si es próximo a 1, la matriz está bien condicionada; si es próximo a cero está mal condicionada)
null(A)	Base ortonormal del núcleo de la matriz A (subespacio { x : Ax=0 })
inv(A)	inversa de la matriz A
lu(A)	factorización LU de la matriz A
chol(A)	factorización de Cholesky de la matriz A
qr(A)	factorización QR de la matriz A
Casi todas las funciones MATLAB relacionadas con el álgebra lineal numérica están basadas en rutinas de la librería LAPACK.	

Para explicaciones más completas sobre estos y otros temas relacionados con el álgebra lineal ver:

Help → MATLAB → Mathematics → Matrices and Linear Algebra

Para referencias completas sobre estas y otras funciones relacionadas ver:

Help → MATLAB → Functions-Categorical List → Mathematics → Linear Algebra

7. POLINOMIOS, INTERPOLACIÓN Y GEOMETRÍA COMPUTACIONAL

7.1 Polinomios

MATLAB representa los polinomios como vectores fila que contienen los coeficientes ordenados de mayor a menor potencia.

```
>> p=[c1, c2, c3,..., cn]
```

representa el polinomio $p(x) = c_1x^{n-1} + c_2x^{n-2} + \dots + c_{n-1}x + c_1$

A continuación se muestran algunas funciones relacionadas con polinomios:

roots(p)	Calcula las raíces del polinomio de coeficientes p
poly(r)	Coeficientes del polinomio cuyas raíces son r
poly(A)	Coeficientes del polinomio de la matriz A
polyval(p,x)	Devuelve el valor en x del polinomio cuyos coeficientes son p
conv(p,q)	Producto de los polinomios p y q
[c,r]=deconv(p,q)	Cociente y resto de la división p/q : p = conv(c,q) + r
[r,p,k]=residue(p,q) [p,q]=residue(r,p,k)	Calcula los coeficientes de la descomposición en fracciones simples de la expresión racional p/q y viceversa Ver HELP para detalles
polyder(p)	derivada del polinomio p
p=polyfit(x,y,n)	Calcula los coeficientes del polinomio de grado n que se ajusta a un conjunto de puntos, de coordenadas (x_i , y_i), en el sentido de los mínimos cuadrados.

Más información en:

Help → MATLAB → Mathematics → Polinomials and Interpolation → Polinomials
Help → MATLAB → Functions-Categorical List → Mathematics → Polinomials

7.2 Interpolación

MATLAB dispone de una gran cantidad de funciones relacionadas con la interpolación. Para una descripción completa véase

Help → MATLAB → Mathematics → Polinomials and Interpolation → Polinomials
Help → MATLAB → Functions-Categorical List → Mathematics → Polinomials

Algunas de ellas son:

yi=interp1(X,Y,xi)	Interpolación en dimensión 1 Devuelve, en el vector yi , los valores en los puntos contenidos en el vector xi , de la función que interpola los valores X , Y El método, por defecto, es lineal a trozos
interp2 interp3 interp n	Son funciones análogas, pero en dimensión 2 , 3 y n respectivamente Ver HELP
griddata(x,y,z,xi,yi)	Similar a interp2 pero los datos que se interpolan (x , y , z) no necesitan estar ordenados Ver HELP
griddata3 griddata n	Similares a griddata , en dimensiones 3 y n respectivamente Ver HELP
yi=spline(X,Y,xi)	Devuelve el valor en xi del spline cúbico que interpola los valores (X , Y)
pp=spline(X,Y)	Devuelve los coeficientes del spline cúbico que interpola los valores (X , Y) (para ser usados con la función ppval)
y = ppval(x,pp)	Calcula el valor en x del spline cúbico determinado por pp (previamente calculado con la función spline) x puede ser un vector, en cuyo caso también lo es y
polyder(p)	Derivada del polinomio p

p=polyfit(x,y,n)	Calcula los coeficientes del polinomio de grado n que se ajusta a un conjunto de puntos, de coordenadas (x_i , y_i), en el sentido de los mínimos cuadrados
-------------------------	--

7.3 Geometría computacional

Para una descripción detallada, véase:

Help → MATLAB → Mathematics → Polinomials and Interpolation → Interpolation → Triangulation and Interpolation of Scattered Data

Help → MATLAB → Functions-Categorical List → Mathematics → Interpolation and Computational Geometry

k=convhull(x,y)	Envolvente convexa de los puntos dados: k son los índices de los vértices del polígono
[k,a]=convhull(x,y)	Devuelve, también, el área del polígono
tri=delaunay(x,y)	Triangulación de Delaunay (2D) Ver HELP
tri=delaunay3(x,y,z)	Teselación de Delaunay 3D (tetraedros, por defecto) Ver HELP
in=inpolygon(x,y, xp, yp)	Devuelve true si el punto (x,y) está en el polígono de vértices (xp(i),yp(i)) x e y pueden ser vectores
voronoi(x,y) voronoi(x,y, ,tri)	Dibuja el diagrama de Voronoi asociado a los puntos (x(i),y(i)) Ver HELP
a=polyarea(xp,yp)	Área del polígono de vértices (xp(i),yp(i))
a=rectint(A,B)	Área de la intersección de dos rectángulos: A y B A=[x,y,anchura,altura] ; (x,y)=vértice inf. izq. B análogo
patch(xp,yp,c)	Dibuja el polígono de vértices (xp(i),yp(i)), relleno con el color c Ver en HELP más detalles (muchos)
patch(xp,yp,zp,c)	Dibuja un polígono en el espacio 3D
triplot(tri,x,y)	Dibuja la triangulación tri de vértices x e y
trisurf(tri,x,y,z)	Dibuja la superficie definida por la triangulación tri , de vértices x e y , y la función z : "altura" en el punto (x,y)
trimesh(tri,x,y,z)	Igual que trisurf , salvo que trimesh no da color a las facetas

8. RESOLUCIÓN DE ECUACIONES NO LINEALES

Para resolver ecuaciones no lineales

$$f(x) = 0, \quad x \in \mathbb{R}, \quad f : \mathbb{R} \rightarrow \mathbb{R}$$

MATLAB dispone de la función **fzero**, cuya utilización más sencilla es:

```
>> x = fzero(fun,x0)
>> [x,fval] = fzero(fun,x0)
```

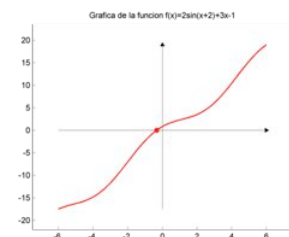
donde:

- **fun** es un "handle" de la función que calcula $f(x)$ (ver apartados 2.2 y 2.3 y ejemplos) Debe responder a la forma: **[y]=fun(x)**
- **x0** es el valor inicial de **x**, a partir del cual se comienza a "buscar" la solución. En general, debe ser un valor próximo a la solución buscada.
- **x** es la solución de la ecuación
- **fval** (opcional) es el valor de **f** en la solución

EJEMPLO

Calcular la solución de $2\sin(x+2) + 3x = 1$

```
>> fun = @(x) 2*sin(x+2)+3*x-1
>> sol = fzero(fun , 0 )
sol =
- 0.3301
```

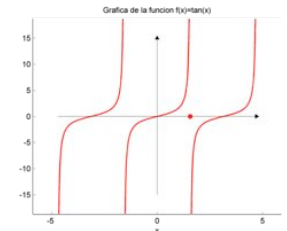


El buen funcionamiento de la función **fzero** requiere que la función sea continua y que "cruce" el eje X. Si la función no es continua el algoritmo puede converger a un punto de discontinuidad.

EJEMPLO : utilización fallida de fzero con una función discontinua

La función $y=\tan(x)$ tiene discontinuidades en todos los múltiplos de $\pi/2$

```
>> fzero(@tan , 1)
ans =
1.5708
```



Si la función tiene un cero en el que no cambia de signo, el algoritmo no convergerá.

EJEMPLO : utilización fallida de fzero con una función que no cambia de signo

La función $y=x^2$ tiene un cero en $x=0$ en el que no cambia de signo

```
>> fzero(@(x) x^2, 0.5)
Exiting fzero: etc.
ans =
NaN
```

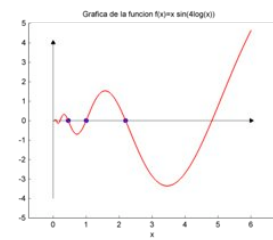


En el mejor de los casos, **fzero** sólo calcula UNA solución de la ecuación. En caso de múltiples soluciones, hay que utilizarla una vez para cada solución que interese encontrar y cobra más importancia el hecho de que el punto inicial, **x0**, esté próximo a la solución.

EJEMPLO : utilización de fzero con multiplicidad de soluciones

La función $y = x \sin(4 \ln(x))$ tiene infinitas soluciones positivas.
Aquí se calculan algunas:

```
>> fun=@(x) x.*sin(4*log(x));
>> sol1 = fzero(fun , 0.5)
sol1 =
    0.4559
>> sol2 = fzero(fun , 1)
sol2 =
    1
>> sol3 = fzero(fun, 2)
sol3 =
    2.1933
```



La función **fzero**, puede también ser usada con un argumento opcional

```
>> x = fzero(fun,x0,options)
```

que permite determinar varios parámetros relativos al modo de funcionamiento de la función **fzero**, como tolerancia a los errores admitida, máximo de número de iteraciones y/o de evaluaciones de la función permitidas, etc. (consultar HELP).

A veces es preciso calcular ceros de funciones que dependen, además de la variable independiente, de uno o varios parámetros. En el ejemplo siguiente se muestra cómo se pueden usar las funciones anónimas para calcular dichos ceros utilizando **fzero**, sin tener que re-escribir la función coste para cada valor de los parámetros.

Se desean calcular las coordenadas del punto de la curva $y = \sin(x)$ más próximo a un punto dado, (a,b) , que se determina en tiempo de ejecución. Lo que hay que hacer para ello es minimizar la función

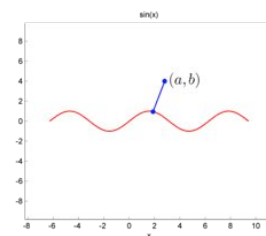
$$f(x) = (x - a)^2 + (\sin(x) - b)^2$$

o bien hallar una raíz de la ecuación

$$x - a + (\sin(x) - b) \cos(x) = 0$$

EJEMPLO : function PtoProx.m

```
function [xs,ys]=PtoProx(a,b)
%
% [xs,ys] = punto de la curva y=sin(x)
% mas proximo al punto (a,b)
%
fun=@(x) x-a+(sin(x)-b)*cos(x);
xs= fzero(fun,a);
ys=sin(xs);
```



La función **fzero** admite también ser utilizada con uno o varios argumentos de entrada más, en la forma:

```
>> x = fzero(fun,x0,options,p1,p2,p3,...)
```

Esto sirve para pasar parámetros auxiliares a la función **fun**: **fzero** no hace nada con ellos, simplemente los añade a lista de argumentos que enviará a **fun** cuando la llame. El ejemplo anterior también se podría haber hecho como sigue:

EJEMPLO : function PtoProxBis.m

```
function [xs,ys]=PtoProxBis(a,b)
%
% [xs,ys] = punto de la curva y=sin(x)
% mas proximo al punto (a,b)
%
options=[];
xs= fzero(@fun,a,options,a,b);
ys=sin(xs);
%
%
function [y]=fun(x,a,b)
y=x-a+(sin(x)-b)*cos(x);
```

9. CÁLCULO DE MÍNIMOS DE FUNCIONES

Para resolver el problema

$$\text{Hallar } x \in [a, b] \text{ tal que } f(x) \leq f(y) \forall y \in [a, b]$$

MATLAB dispone de la función **fminbnd**, cuya utilización más sencilla es:

```
>> x = fminbnd(fun,a,b)
>> [x,fval] = fminbnd(fun,a,b)
```

donde:

- **fun** es un "handle" de la función que calcula $f(x)$ (ver apartados 2.2 y 2.3 y ejemplos)
Debe responder a la forma: **[y]=fun(x)**
- **a,b** son los extremos del intervalo
- **x** es una aproximación del punto que produce el mínimo
- **fval** (opcional) es el valor de **f** en la solución

Otras posibles utilizaciones de **fminbnd** son

```
>> x = fminbnd(fun,a,b,options)
>> [x,fval] = fminbnd(fun,a,b,options,p1,p2,...)
```

El significado de los argumentos y la utilización de esta función debe resultar obvio después de la lectura del epígrafe anterior sobre la función **fzero**.

MATLAB dispone de la función **fminsearch** para calcular mínimos de funciones escalares de varias variables:

```
>> x = fminsearch(fun,x0)
>> [x,fval] = fminsearch(fun,x0)
>> x = fminbnd(fun,x0,options)
>> x = fminbnd(fun,x0,options,p1,p2,...)
```

fminsearch lleva a cabo una búsqueda del mínimo, partiendo de la aproximación inicial **x0**. La utilización de esta función es similar a las dos anteriores.

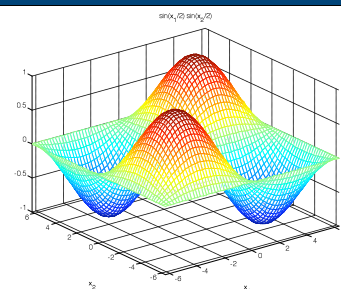
Si la función a minimizar depende de **N** variables, entonces **fun** debe responder a la forma **[y]=fun(x)**, siendo **x** un vector de **N** componentes. Ver el ejemplo siguiente:

EJEMPLO :

Calcular, comenzando en (1,1), un mínimo de la función

$$f : \mathbb{R}^2 \longrightarrow \mathbb{R}, \quad f(x) = \sin\left(\frac{x_1}{2}\right)\sin\left(\frac{x_2}{2}\right)$$

```
>> fun = @(x) sin(x(1)/2)*sin(x(2)/2) ;
>> xsol = fminsearch(fun,[1,1])
xsol =
    3.1416   -3.1416
```



El algoritmo utilizado por **fminsearch** no utiliza información alguna sobre el gradiente de la función. En consecuencia no se puede esperar gran cosa de él en según qué casos ...

El punto encontrado puede ser "sólo" un mínimo local.

10. CÁLCULO DE INTEGRALES DEFINIDAS

La función MATLAB quad sirve para calcular integrales definidas:

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

Su utilización más sencilla es:

```
>> v = quad(fun,a,b)
```

donde:

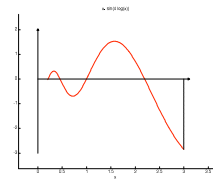
- **fun** es un "handle" de la función que calcula el integrando $f(x)$. Debe responder a la forma: **[y]=fun(x)** y su código **debe estar vectorizado**, esto es, debe poder admitir como argumento **x** un vector y devolver un vector.
- **a,b** son los límites de integración

EJEMPLO

Calcular

$$\int_{0.2}^3 x \operatorname{sen}(4 \ln(x)) dx$$

```
>> fun = @(x) x.*sin(4*log(x));
>> area = quad(fun,0.2,3)
area =
-0.2837
```



Otros argumentos opcionales de la función quad son:

```
>> v = quad(fun,a,b,tol,trace)
```

que permiten determinar el nivel de tolerancia del error admitido, así como el nivel de parloteo durante los cálculos (ver HELP).

La función quad permite también el paso de parámetros a la función integrando, mediante

```
>> v = quad(fun,a,b,tol,trace,p1,p2,...)
```

EJEMPLO

Calcular

$$\int_{0.2}^3 x \operatorname{sen}(4 \ln(kx)) dx$$

```
>> fun = @(x,k) x.*sin(4*log(k*x));
>> k=3.33;
>> area = quad(fun,0.2,3,[],[],3.33)
area =
1.9558
>> k=0.22;
>> area = quad(fun,0.2,3,[],[],0.22)
area =
-0.7245
```


Para calcular integrales dobles

$$w = \int_a^b \int_c^d f(x, y) dy dx$$

se puede usar la función **dblquad**

```
>> w = dblquad(fun,a,b,c,d)
```

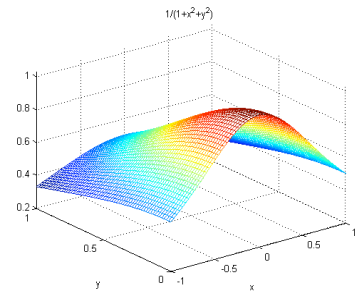
La función **fun** debe responder a la forma **[v]=fun(x,y)** y admitir un vector como argumento **x** (y devolver un vector de su misma dimensión).

EJEMPLO

Calcular

$$\int_{-1}^1 \int_0^1 \frac{1}{1+x^2+y^2} dy dx$$

```
>> fun = @(x,y) 1./(1+x.^2+y.^2);
>> dblquad(fun,-1,1,0,1)
ans =
    1.2790
```



La función

```
>> w = triplequad(fun,a,b,c,d,e,f)
```

permite calcular una integral triple:

$$w = \int_a^b \int_c^d \int_e^f g(x, y, z) dz dy dx$$

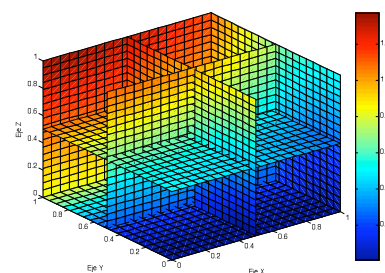
La función integrando debe responder a la forma **[w]=fun(x,y,z)** y admitir un vector como argumento **x** (y devolver un vector de su misma dimensión).

EJEMPLO

Calcular

$$\int_0^1 \int_0^1 \int_0^1 x \sin(x) + z \cos(y) \cos(x) dz dy dx$$

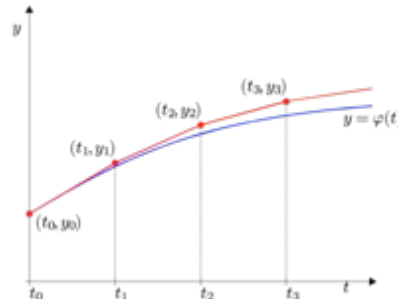
```
>> fun=@(x,y,z) x.*sin(x)+z*cos(y)*cos(x);
>> triplequad(fun,0,1,0,1,0,1)
ans =
    0.6552
```



11. RESOLUCIÓN DE ECUACIONES DIFERENCIALES

Para resolver problemas relativos a (sistemas de) ecuaciones diferenciales ordinarias MATLAB dispone de una buena cantidad de funciones.

La resolución numérica de (un problema relativo a) una ecuación diferencial consiste en hallar valores numéricos que aproximen los valores en determinados puntos de la solución exacta del problema en cuestión.



11.1 Problemas de valor inicial para ecuaciones diferenciales ordinarias

Para resolver problemas de valor inicial como

$$\begin{cases} y' = f(x, y) & \text{en } [t_0, t_f] \\ y(t_0) = y^0 \end{cases}$$

MATLAB dispone de la familia de funciones **ode*****. Cada una de ellas implementa un método numérico distinto de resolución de ecuaciones diferenciales ordinarias, siendo adecuado usar unas u otras en función de las dificultades de cada problema concreto. La utilización de todas las funciones de la familia **ode**** es similar, al menos en lo más básico.

En primera instancia, para resolver problemas no demasiado "difíciles", se deben usar las funciones **ode45** y **ode23**. Su utilización más sencilla es:

```
>> ode45(fun, [t0, tf], y0)
```

donde:

- **fun** es un "handle" de la función que calcula el segundo miembro **f(t,y)**. Debe responder a la forma: **[dy]=fun(t,y)** y puede ser una función anónima o una M-función
- **[t0,tf]** es el intervalo en el que se desea resolver la ecuación
- **y0** es el valor de la condición inicial

EJEMPLO: Problema de valor inicial para una ed. diferencial ordinaria

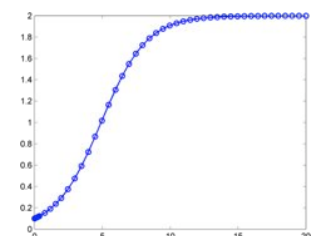
$$\begin{cases} y' = 0.3(2 - y)y & \text{en } [0, 20] \\ y(0) = 0.1 \end{cases}$$

RESOLUCIÓN INTERACTIVA

En este caso **f(t,y) = 0.3(2-y)y**, **[t0,tf] = [0,20]**
e **y0 = 0.1**

```
>> fun = @(t,y) 0.3*y*(2-y) ;  
>> ode45(fun, [0,20], 0.1)
```

Como resultado de la ejecución de esta orden se obtiene la gráfica de la solución numérica



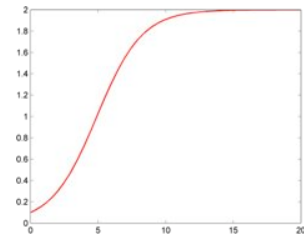
Si lo que se desea es recuperar los valores de la solución numérica se debe usar la función **ode45** en la forma:

```
>> [t,y] = ode45(fun, [t0, tf], y0)
```

De esta manera se obtienen dos vectores, **t** e **y**, conteniendo, respectivamente, las abscisas y las ordenadas de la solución numérica, que luego se podrán utilizar a conveniencia.

Para el mismo problema del ejemplo anterior, se desea calcular el valor en **t=20** de la solución:

```
>> fun = @(t,y) 0.3*y*(2-y);
>> [t,y] = ode45(fun,[0,20],0.1);
>> plot(t,y,'Color','red')
>> y(end)
ans =
    1.9998
```



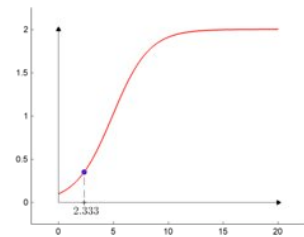
Si se desea conocer el valor de la solución numérica en puntos intermedios del intervalo **[t0,tf]**, se puede utilizar la función **ode45** en combinación con la función **deval** de la siguiente forma:

```
>> sol = ode45(fun,[t0,tf],y0);
>> yint = deval(xint,sol);
```

donde **xint** es el punto del intervalo en el que se desea evaluar la solución. **xint** puede ser un vector, en cuyo caso **yint** será un vector de la misma dimensión.

Para el mismo problema del ejemplo anterior, se desea calcular el valor en **t=2.333** de la solución:

```
>> fun = @(t,y) 0.3*y*(2-y);
>> sol = ode45(fun,[0,20],0.1);
>> yint = deval(2.333,sol);
yint =
    0.3517
```



Las mismas funciones de la familia ode*** sirven para resolver numéricamente sistemas de ecuaciones diferenciales ordinarias. Para ello basta escribir dichos sistemas en notación vectorial, es decir, como una ecuación diferencial de la forma

$$Y' = \frac{dY}{dt} = F(t, Y), \quad F : R^{N+1} \longrightarrow R^N$$

cuyas soluciones son funciones dependientes de t con valores vectoriales

$$Y : I \subset R \longrightarrow R^N.$$

El ejemplo siguiente ayudará a comprender esto y a explicar cómo utilizar la función **ode45** (u otra de la familia) para resolver numéricamente un problema de valores iniciales asociado a un sistema diferencial.

El problema de valores iniciales siguiente

$$\begin{cases} y_1' = 0.5 y_1 - 0.2 y_1 y_2 \\ y_2' = -0.5 y_2 + 0.1 y_1 y_2 \\ y_1(0) = 10 \\ y_2(0) = 4 \end{cases}$$

se escribe en notación vectorial

$$\begin{cases} \begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} 0.5 y_1 - 0.2 y_1 y_2 \\ -0.5 y_2 + 0.1 y_1 y_2 \end{pmatrix} \\ \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 10 \\ 4 \end{pmatrix} \end{cases}$$

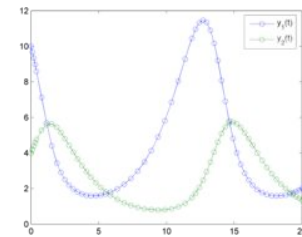
$$Y(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix}, \quad Y^0 = \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 10 \\ 4 \end{pmatrix}, \quad F(t, Y) = \begin{pmatrix} 0.5 y_1 - 0.2 y_1 y_2 \\ -0.5 y_2 + 0.1 y_1 y_2 \end{pmatrix}$$

Las órdenes siguientes permiten calcular la solución de este problema: obsérvese que **fun(t,y)** es un vector columna con dos componentes, igual que la condición inicial.

```
>> fun = @(t,y) [ 0.5*y(1)-0.2*y(1)*y(2) ; -0.5*y(2)+0.1*y(1)*y(2) ];
>> ode45(fun,[0,20],[ 10 ; 4 ])
```

Estas órdenes dibujarán las gráficas de las dos componentes de la solución. Las leyendas se han insertado con la orden

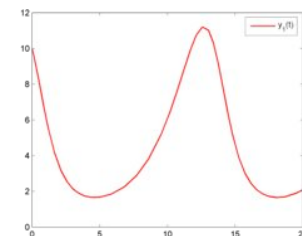
```
>> legend('y_1(t)', 'y_2(t)')
```



De la misma manera que antes, se pueden usar las funciones **ode**** para recuperar los valores de la solución numérica:

Para el mismo problema del ejemplo anterior, dibujar la primera componente de la solución:

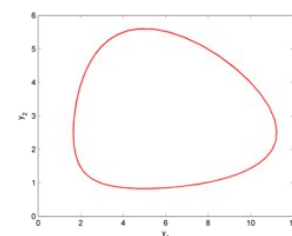
```
>> [t,y]= ode23(fun,[0,20],[ 10 ; 4 ]);
>> plot(t,y(:,1),'Color',[1,0,0],'LineWidth',1.5)
>> legend('y_1(t)')
```



También se puede evaluar la solución numérica con ayuda de la función **deval**, como antes.

Dibujar la gráfica de la solución en el plano de fases (gráfica de y_2 frente a y_1)

```
>> sol = ode23(fun,[0,20],[ 10 ; 4 ]);
>> tt = linspace(0,20,100) ;
>> yy = deval(tt,sol) ;
>> plot(yy(1, :),yy(2, :))
```



11.2 Problemas de valor inicial para ecuaciones diferenciales ordinarias de orden superior a 1

Las mismas funciones son también válidas para resolver problemas de valor inicial relativos a ecuaciones diferenciales ordinarias de orden superior a 1, como:

$$\begin{cases} y^{(n)} = f(t, y, y', y'', \dots, y^{(n-1)}) \\ y(t_0) = y_0 \\ y'(t_0) = y_1 \\ y''(t_0) = y_2 \\ \dots \\ y^{(n-1)}(t_0) = y_{n-1} \end{cases}$$

Un problema como este siempre se puede, mediante un sencillo cambio de variable, escribir como un sistema diferencial de primer orden equivalente. En efecto, el cambio de variable:

$$z_1(t) = y(t), \quad z_2(t) = y'(t), \quad \dots \quad z_n(t) = y^{(n-1)}(t)$$

transforma el problema anterior en el siguiente sistema, cuya expresión vectorial se da al lado:

$$\begin{cases} z_1' = z_2 \\ z_2' = z_3 \\ \dots \\ z_n' = f(t, z_1, z_2, z_3, \dots, z_{n-1}) \\ z_1(t_0) = y_0 \\ z_2(t_0) = y_1 \\ \dots \\ z_n(t_0) = y_{n-1} \end{cases} \quad \begin{cases} Z' = F(t, Z) = \begin{pmatrix} z_2 \\ z_3 \\ \vdots \\ f(t, z_1, z_2, z_3, \dots, z_{n-1}) \end{pmatrix} \\ Z(t_0) = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \end{cases}$$

EJEMPLO: ecuación de van der Pol

La ecuación de segundo orden llamada de van der Pol,

$$y'' - \alpha(1 - y^2)y' + y = 0$$

es equivalente al sistema diferencial de primer orden

$$\begin{cases} z_1' = z_2 \\ z_2' = \alpha(1 - z_1^2)z_2 - z_1 \end{cases}$$

en el sentido de que, si se conoce una solución $(z_1(t), z_2(t))$ del sistema, entonces $y(t) = z_1(t)$ es una solución de la ecuación.

El problema de valor inicial

$$\begin{cases} y'' - \alpha(1 - y^2)y' + y = 0 \\ y(0) = y_{00} \\ y'(0) = y_{01} \end{cases}$$

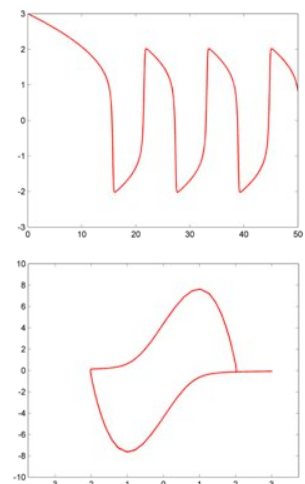
es así equivalente a

$$\begin{cases} z_1' = z_2 \\ z_2' = \alpha(1 - z_1^2)z_2 - z_1 \\ z_1(0) = y_{00} \\ z_2(0) = y_{01} \end{cases}$$

RESOLUCIÓN

```
>> alpha = 5;
>> fun=@(t,z) [z(2) ; alpha*(1-z(1).^2).*z(2)-z(1) ];
>> [t,y] = ode23(fun,[0,50],[2;0]);
>> plot(t,y(:,1)) % 1a. componente
```

```
>> plot(y(:,1),y(:,2)) % órbita en el plano de fases
```



11.3 Ecuaciones diferenciales ordinarias con retardo

En construcción.

11.4 Problemas de contorno para sistemas diferenciales de primer orden

Consideramos aquí problemas diferenciales del tipo siguiente:

$$\begin{cases} \text{Hallar } Z : [a, b] \longrightarrow \mathbb{R}^n \text{ tal que} \\ Z' = F(t, Z), \quad t \in [a, b] \\ BC(Z(a), Z(b)) = 0 \end{cases}$$

donde $BC(Z(a), Z(b)) = 0$ son las condiciones de contorno, esto es, condiciones que involucran los valores de la solución buscada en ambos extremos del intervalo.

Por ejemplo, el problema diferencial (con condiciones de contorno)

$$y'' + |y| = 0, \quad y(0) = 0, \quad y(4) = -2$$

(que tiene dos soluciones) se puede escribir en la forma anterior:

$$\begin{cases} z_1' &= z_2 \\ z_2' &= -|z_1| \\ z_1(0) &= 0 \\ z_2(4) + 2 &= 0 \end{cases}$$

La función MATLAB **bvp4c** permite resolver este tipo de problemas. Su uso es el siguiente:

```
>> sol = bvp4c(odefun,bcfun,solinit)
```

donde

- **odefun** es la función que evalúa el segundo miembro del sistema diferencial, $F(t, Z)$. Debe ser de la forma
[dzdt] = odefun(t,z)
- **bcfun** es la función que describe la condición de contorno $BC(Z(a), Z(b)) = 0$. Debe ser de la forma
[res] = bcfun(za,zb)
- **solinit** es una estructura que proporciona un ensayo inicial para la solución. Puede ser construida con ayuda de la función **bvpinit**, cuya forma más sencilla de uso es
solinit = bvpinit(tmesh,zinit)
donde **tmesh** es una partición inicial del intervalo **[a,b]** y **zinit** es un valor inicial para la solución, por ejemplo un vector constante (también puede ser una función; ver HELP).
- **sol** es una estructura que permite recuperar los valores de la solución numérica.
sol.x es la partición final del intervalo **[a,b]** (vector fila 1 x N)
sol.z son los valores de las componentes de la solución en los puntos de la partición anterior (matriz n x N) (n = número es sistema)
También permite, con ayuda de la función **deval**, evaluar la solución en cualquier punto del intervalo **[a,b]**:
z = deval(t,sol)

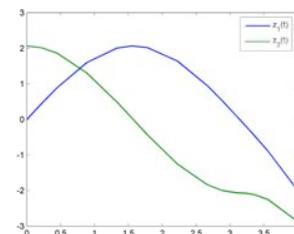
EJEMPLO: problema de contorno para una e.d.o. de 2o. orden

Calcular una solución de

$$\begin{cases} z_1' &= z_2 \\ z_2' &= -|z_1| \\ z_1(0) &= 0 \\ z_2(4) &= -2 \end{cases}$$

RESOLUCIÓN

```
>> ftz = @(t,z) [ z(2) ; -abs(z(1)) ] ;
>> fbc = @(za,zb) [ za(1) ; zb(1)+2 ] ;
>> tt = linspace(0,4,10) ;
>> zinit = [ 1 ; 0 ] ;
>> solinit = bvpinit(tt,zinit) ;
>> sol = bvp4c(ftz,fbc,solinit) ;
>> t = sol.x ;
>> z = sol.y ;
>> plot(t,z)
>> legend('z_1(t)', 'z_2(t)')
```



(La otra solución se puede encontrar eligiendo **zinit = [-1 ; 0]**)

11.5 Problemas de contorno para sistemas diferenciales de primer orden con parámetros

La función **bvp4c** también puede resolver problemas de contorno con parámetros desconocidos:

$$\begin{cases} \text{Hallar } Z : [a, b] \rightarrow R^n \text{ y } p \in R^k \text{ tales que} \\ Z' = F(t, Z, p), \quad t \in [a, b] \\ BC(Z(a), Z(b), p) = 0 \end{cases}$$

En ese caso **bvp4c** se utiliza en la misma forma, pero cambia la definición de sus argumentos:

```
>> sol = bvp4c(odefun,bcfun,solinit)
```

donde

- **odefun** es la función que evalúa el segundo miembro del sistema diferencial, $F(t, Z, p)$
Debe ser de la forma
[dzdt] = odefun(t,z,p)
- **bcfun** es la función que describe la condición de contorno $BC(Z(a), Z(b), p)$
Debe ser de la forma
[res] = bcfun(za,zb,p)
- **solinit** es una estructura que proporciona un ensayo inicial para la solución.
Puede ser construida con ayuda de la función **bvpinit**, cuya forma más sencilla de uso es
solinit = bvpinit(tmesh,zinit,pinit)
donde **tmesh** es una partición inicial del intervalo **[a,b]**, **zinit** es un valor inicial para la solución, por ejemplo un vector constante (también puede ser una función; ver HELP), y **p** es el valor inicial para el ó los parámetros
- **sol** es una estructura que permite recuperar los valores de la solución numérica.
sol.x es la partición final del intervalo **[a,b]** (vector fila 1 x N)
sol.z son los valores de las componentes de la solución en los puntos de la partición anterior (matriz n x N) (n= número de ecuaciones)
sol.parameters es el valor encontrado para la incógnita p
También permite, con ayuda de la función **deval**, evaluar la solución en cualquier punto del intervalo **[a,b]**:
z = deval(t,sol)

EJEMPLO: cálculo de un autovalor y una autofunción para una e.d.o. de 2o. orden

Se considera el siguiente problema de autovalores

$$\begin{cases} t^2 y'' + t y' = -\lambda y \\ y(1) = 0 \\ y(e) = 0 \end{cases}$$

Este problema tiene infinitas soluciones :

$$\lambda_n = (n\pi)^2, \quad \varphi_n(t) = c_n \sin(n\pi \ln(t)), \quad n = 1, 2, \dots$$

Es equivalente al siguiente problema de contorno para un sistema diferencial de primer orden

$$\begin{cases} z'_1 = z_2 \\ z'_2 = -\frac{1}{t} z_2 - \frac{\lambda}{t^2} z_1 \\ z_1(1) = 0 \\ z_1(e) = 0 \end{cases}$$

Al haber ahora una incógnita más (el parámetro λ) se necesita imponer una incógnita más (que fije una entre todas las autofunciones). Imponemos, por ejemplo, $y'(1) = 1$ o lo que es lo mismo $z_2(1) = 1$. Con ello, el sistema a resolver queda

$$\begin{cases} z'_1 = z_2 \\ z'_2 = -\frac{1}{t} z_2 - \frac{\lambda}{t^2} z_1 \\ z_1(1) = 0 \\ z_1(e) = 0 \\ z_2(1) = 1 \end{cases}$$

RESOLUCIÓN

```

>> funode = @(t,z,lambda) [ z(2) ; -z(2)/t - lambda*z(1)/(t*t) ];
>> funbc = @(za,zb,lambda) [ za(1) ; zb(1) ; za(2)-1 ];
>> solinit = bvpinit(linspace(1,exp(1),10),[1;1],100);
>> sol = bvp4c(funode,funbc,solinit);
>> tt = linspace(1,exp(1));
>> zz = deval(tt,sol);
>> plot(tt,zz)
>> lambda = sol.parameters ;
>> title(['Autofuncion asoc. al autovalor ',num2str(lambda)])

```



11.6 Ecuaciones y sistemas de ecuaciones en derivadas parciales parabólicas y elípticas dependientes de una variable espacial y del tiempo

En construcción.

12. INSTRUMENTOS BÁSICOS DE PROGRAMACIÓN

12.1. Decisiones y bucles

Las sentencias MATLAB para implementar las tomas de decisión (bifurcaciones) y los bucles (repeticiones) son las siguientes:

- **Sentencia IF**

En su forma más simple se escribe en la forma siguiente:

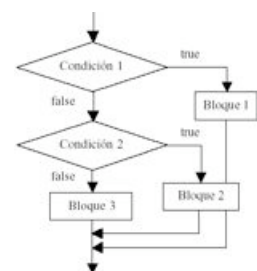
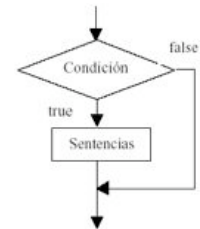
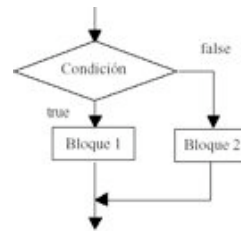
```
if condicion
    sentencias
end
```

También se puede utilizar en la forma:

```
if condicion
    sentencias-1
else
    sentencias-2
end
```

Su forma más compleja es:

```
if condicion-1
    sentencias-1
elseif condicion-2
    sentencias-2
elseif condicion-3
    sentencias-3
else
    sentencias-4
end
```



En estas instrucciones, **condicion** es una expresión con valor lógico (true o false). Si el valor de la expresión es una matriz, sólo se ejecuta el conjunto de sentencias si todos los elementos son **true** (o, igualmente, distintos de cero).

- **Sentencia SWITCH**

Es análoga a un conjunto de **if ... elseif** concatenados. Su forma general es:

```
switch expresion
case caso-1
    sentencias-1
case caso-2
    sentencias-2
case {caso-3, caso-4, caso-5, ...}
    sentencias-3
otherwise
    sentencias-4
end
```

Aquí, **expresion** debe tener un valor numérico o de cadena de caracteres.

Ejemplo tonto de uso de SWITCH

```
%
% Ver más adelante el uso de las instrucciones input y disp
%
resp=input('escribe un numero del 0 al 9 :: ');
%
switch resp
case 0
    disp('Has escrito 0 (cero)')
case 1
    disp('Has escrito 1')
case {2,3,4,5}
    disp('Has escrito un numero entre 2 y 5')
otherwise
    disp('has escrito un numero mayor que 5')
end;
```

- **Sentencia FOR**

Repite un conjunto de sentencias un numero predeterminado de veces. Su forma general es la siguiente:

```
for variable=vector-fila
    sentencias
end
```

Se ejecuta el conjunto de **sentencias** tantas veces como elementos tenga el **vector-fila**, y en cada repetición, la **variable** toma el valor del elemento correspondiente. Por ejemplo, se puede usar en la forma:

```
for i=1:n
    sentencias
end
```

que ejecutará el conjunto de sentencias **n** veces, para $i=1,2,3,\dots,n$.
En la forma:

```
for i=n:-1:1
    sentencias
end
```

se ejecuta el conjunto de sentencias también **n** veces, pero en orden inverso: $i=n,n-1,\dots,1$.

- **Sentencia WHILE**

Repite la ejecución de un conjunto de sentencias mientras que una determinada condición se verifique. Su forma general es:

```
while condicion
    sentencias
end
```

Se ejecutará el bloque de sentencias mientras que **condicion** tome el valor **true**, o mientras que **condicion** sea distinto de cero.

- **Sentencia CONTINUE**

Detiene la ejecución de la iteración actual en un bucle **for** o **while** y pasa el control a la siguiente iteración.

- **Sentencia BREAK**

Detiene completamente la ejecución de un bucle **for** o **while**. Si hay bucles anidados, se detiene la ejecución del más interno.

12.2. Operadores de relación y lógicos

- **Operadores de relación o comparación**

Los siguientes operadores producen como resultado un valor lógico:

true (cualquier valor distinto de cero)

false (cero)

Cuando estos operadores se utilizan para comparar dos matrices de las mismas dimensiones, el resultado es otra matriz de la misma dimensión y la comparación se realiza elemento a elemento. Cuando se utilizan para comparar un escalar con una matriz, el resultado es una matriz, y se compara el escalar con cada uno de los elementos.

== Igual a

~= No igual a

< Menor que

> Mayor que

<= Menor o igual que

>= Mayor o igual que

- **Operadores lógicos**

Son operadores que actúan entre valores lógicos:

- **& Y lógico**

A & B produce los resultados que se reflejan en la tabla
A & B es equivalente a **and(A, B)**

- **| O lógico**

A | B produce los resultados que se reflejan en la tabla
A | B es equivalente a **or(A, B)**

- **~ Negación lógica**

~A produce los resultados que se reflejan en la tabla
~A es equivalente a **not(A)**

		A	
		T	F
B	T	T	F
	F	F	F

		A	
		T	F
B	T	T	T
	F	F	F

A	T	F
NOT(A)	F	T

MATLAB dispone, además, de los operadores **&&** y **||** llamados respectivamente **short-circuit and** y **short-circuit or**, que sólo son válidos entre operandos escalares y que se diferencian de los anteriores en que, si no es necesario, no calculan el segundo operando. Por ejemplo, suponiendo que **A=1** y que **B=2**, la operación

(A==2) & (B==2)

evaluaría en primer lugar **A==2** (resultado **false**), evaluaría en segundo lugar **B==2** (resultado **true**) y evaluaría por último **false & true** (resultado **false**). Sin embargo, la operación

(A==2) && (B==2)

evaluaría **A==2** (resultado **false**) y daría directamente **false** como resultado, sin evaluar el segundo operando.

12.3. Instrucciones sencillas de entrada/salida

- **Función INPUT**

Permite imprimir un mensaje en la línea de comandos y leer datos desde el teclado. La instrucción

```
>> resp=input('Mensaje que se imprime')
```

imprime **Mensaje que se imprime** en una línea de la ventana de comandos y se queda esperando a que el usuario teclee un valor o una expresión.

Si se teclea un valor (escalar o matricial), MATLAB lo almacena en la variable **resp**.

Si se teclea una expresión, MATLAB la evalúa con los valores actuales de las variables del Workspace, y el resultado se almacena en la variable **resp**.

Si en el mensaje se incluye el comando **\n**, se produce un salto de línea.

- **Función DISP**

Permite imprimir en la pantalla un mensaje de texto:

```
>> disp('Mensaje que se imprime')
```

También permite imprimir el valor de una variable, sin imprimir su nombre:

```
>> a=[1,2;3,4];
>> disp(a)
     1     2
     3     4
```

Para mezclar texto con el valor numérico de una variable o expresión, hay que transformar el valor numérico en la cadena de texto equivalente y concatenar las cadenas. Por ejemplo:

```
>> disp([ 'El numero pi vale : ' , num2str(pi) ])
El numero pi vale : 3.1416
```

12.4. Instrucciones detalladas de entrada/salida

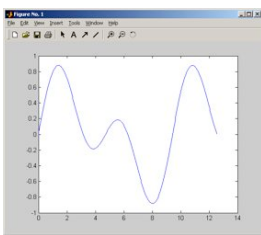
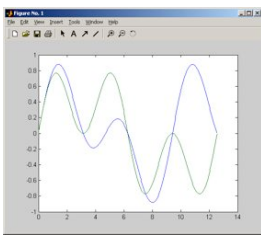
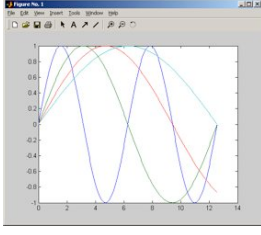
En construcción.

12.4. Lectura y escritura en/desde ficheros

En construcción.

13. COMANDOS DE DIBUJO 2D Y 3D

Vemos aquí los comandos más completos de que dispone MATLAB para dibujar curvas planas y en el espacio, superficies, líneas de nivel, etc. La diferencia con los que se vieron en la Sección 5 es que aquí hay que proporcionar a las funciones los datos numéricos para construir la gráfica.

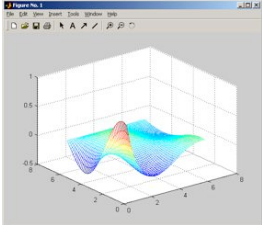
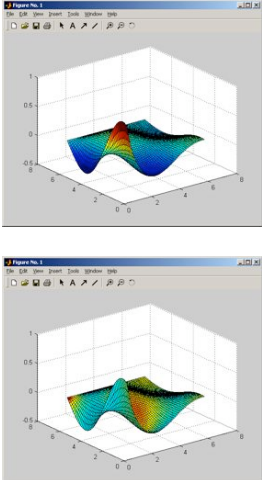
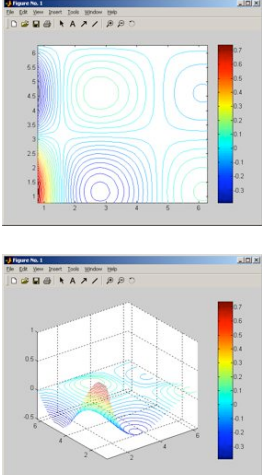
<code>plot(x,y)</code>	<p>Si x e y son dos vectores de la misma dimensión, n, dibuja una curva (lineal a trozos) que pasa por los puntos (x_i , y_i), i=1,... n</p> <p>Ejemplo: <code>>> x=0:pi/100:4*pi;</code> <code>>>y=sin(x) .*cos(x/3);</code> <code>>> plot(x,y)</code></p>			
<code>plot(x1,y1,x2,y2)</code>	<p>Dibuja las dos curvas (x1_i , y1_i), i=1,... n1 y (x2_i , y2_i), i=1,... n2 en la misma ventana y en los mismos ejes</p> <p>Ejemplo: <code>>> x=0:pi/100:4*pi;</code> <code>>> y=sin(x) .*cos(x/3);</code> <code>>> z=sin(x) .*cos(x/2);</code> <code>>> plot(x,y,x,z)</code></p>			
<code>plot(x,A)</code>	<p>donde x es un vector-columna (resp. fila) y A es una matriz, dibuja tantas curvas (x_i , A_{ij}), i=1,... n (como columnas (resp. filas) tenga la matriz A</p> <p>Ejemplo: <code>>> x=0:pi/100:4*pi;</code> <code>>>A=[sin(x);sin(x/2);sin(x/3);sin(x/4)];</code> <code>>> plot(x,A)</code></p>			
<p>El comando plot asigna, por defecto, determinados colores a las gráficas. Estos colores, así como el tipo de línea a utilizar para dibujar (continua, de puntos, con marcadores, etc.) se pueden modificar.</p>				
<code>plot(x,y,param)</code>	<p>donde param es una cadena de caracteres como máximo, cada uno a elegir de una de las columnas siguientes:</p> <table><tr><td>b amarillo</td><td>.</td> asteriscos</tr></table>	b amarillo	.	-
b amarillo	.			
b azul	.	-		
g verde	o	:		
r rojo	x	-.		
c cyan	+	--		
m magenta	*			
y amarillos	s			
k negro	d			
	v			

etc. (consultar help plot)

Ejemplo:
`>> x=0:pi/100:4*pi;`
`>> y=sin(x) .*cos(x/3);`
`>> plot(x,y,'r*')`

 || `plot3(x,y,z)` `plot3(x,y,z,param)` | Si **x** , **y** , **z** son tres vectores de la misma dimensión, **n**, dibuja una curva tridimensional (lineal a trozos) que pasa por los puntos **(x_i , y_i , z_i)**, **i=1,... n** **Ejemplo:** `>> alpha=0:pi/80:8*pi;` `>> z=alpha/8;` `>> x=z/(8*pi)+z.*cos(alpha);` `>> y=z/(8*pi)+z.*sin(alpha);` `>> plot3(x,y,z)` | |

El comando plot asigna, por defecto, determinados colores a las gráficas. Estos colores, así como el tipo de línea a utilizar para dibujar (continua, de puntos, con marcadores, etc.) se pueden modificar.

<p>meshgrid(xp,yp) mesh(x,y,z) meshc(x,y,z)</p>	<p>Representa una superficie $z=f(x,y)$ sobre una malla rectangular. Los argumentos x, y, z son matrices de la misma dimensión conteniendo, respectivamente, las coordenadas x, y, z de los nodos de la malla. Los segmentos de la malla se colorean según los valores de la función (coordenada z)</p> <p>La función meshc hace lo mismo, pero dibujando además las líneas de nivel en el plano XY</p> <p>La función meshgrid sirve para construir la malla de base, en el plano XY: Si xp es una partición del intervalo $[x_0, x_1]$ e yp es una partición del intervalo $[y_0, y_1]$, entonces</p> <pre>>> [x,y]=meshgrid(xp,yp)</pre> <p>construye dos matrices, x e y, que definen una malla del rectángulo $[x_0, x_1] \times [y_0, y_1]$. La matriz x contiene las coordenadas X de los puntos de la malla y la matriz y sus coordenadas Y</p> <p>Ejemplo:</p> <pre>>> xp=linspace(pi/4,2*pi,50); >> [x,y]=meshgrid(xp,xp); >> z=(cos(x)./x).*(sin(y)./sqrt(y)); >> mesh(x,y,z)</pre>	
<p>surf(x,y,z) surfc(x,y,z) surf1(x,y,z)</p>	<p>La función surf hace lo mismo que mesh, pero dibujando los segmentos de la malla en color negro y rellenando los "rectángulos" de la malla de color, según los valores de la función</p> <p>La función surfc hace lo mismo, pero dibujando además las líneas de nivel en el plano XY</p> <p>La función surf1 hace lo mismo que surf, pero además añade una fuente de luz lateral</p>	
<p>contour(x,y,z,n) contour3(x,y,z,n)</p>	<p>La función contour dibuja las proyecciones sobre el plano XY de las líneas de nivel (isovalores)</p> <p>La función contour3 dibuja las líneas de nivel sobre la misma superficie</p> <p>En ambos casos n es el número de líneas a dibujar.</p> <p>Se puede usar la función colorbar para añadir a la gráfica una barra con la graduación de colores y la correspondencia con los valores de la función representada. (Esta función puede ser usada con cualquier otra función gráfica que utilice colores).</p>	

colormap(m)

permite cambiar el mapa de colores que se utiliza en la representación. En general, m es una matriz con tres columnas, de modo que la i -ésima fila determina las proporciones, en la escala RGB, del i -ésimo color utilizado.

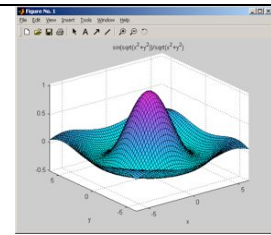
Para más comodidad, MATLAB dispone de una serie de mapas de colores pre-definidos, que se pueden imponer mediante

colormap(mapa)

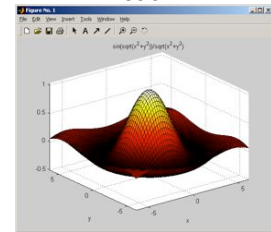
mapa puede tomar uno de los valores siguientes:

hsv	- la escala por defecto (hue-saturation-value).
cool	- Gama de colores entre cyan y magenta.
hot	- Gama de colores entre rojo oscuro y amarillo.
gray	- Gama de colores grises.
copper	- Gama de colores cobrizos.
pink	- Gama de colores rosados.
flag	- Alterna rojo - blanco - azul - negro.
colorcube	- Contraste de colores.
autumn	- Colores entre el rojo y el amarillo.
spring	- Colores entre el magenta y el amarillo.
winter	- Colores entre el azul y el verde.
summer	- Colores entre el verde y el amarillo.

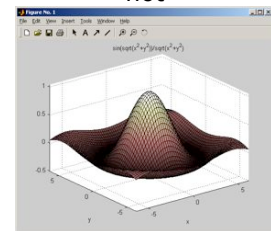
para más mapas ver `help graph3d`



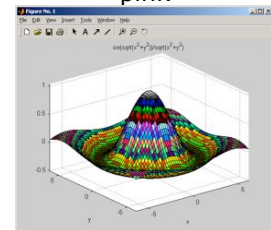
cool



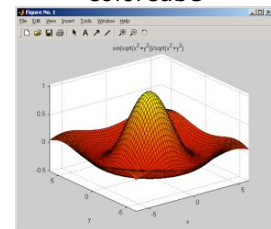
hot



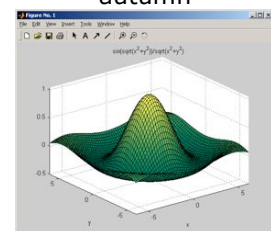
pink



colorcube



autumn



summer