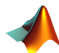
Universidad de Zaragoza
Facultad de Ciencias
Departamento de Matemática Aplicada

Introducción a MATLAB



Luis Rández

© 2007 randez@unizar.es
LaTeX 2ε Actualizado el: 27 de Febrero de 2007

Tabla de Contenido

1. Introducción
2. Básicos
3. Vectores y Matrices
4. Estructuras y Células
5. Órdenes básicas de programación
6. m-files: script y function
7. Gráficos
8. Gráficos Avanzados
9. Optimizaciones
10. Entradas/Salidas
11. Profiling
12. Algunos ejemplos de Análisis Numérico
13. Interfaces Gráficos
14. Matrices Huecas (*Sparse*)
15. Miscelánea

1. Introducción

En la implementación de algoritmos numéricos, usualmente hay que elegir un lenguaje de programación, así como librerías auxiliares:

Futurama

Lenguajes de programación.

- C, C++
- FORTRAN[a], FORTRAN 9X

Librerías numéricas

- LINPACK, LAPACK, FFT, GSL, ...
- NAG, IMSL, ...

Librerías gráficas

- dislin
- GKS

lo que obliga a conocer un amplio abanico de programas.

---

[a] El compilador optimizado de FORTRAN es uno de **The Top Ten Algorithms of the XX Century**

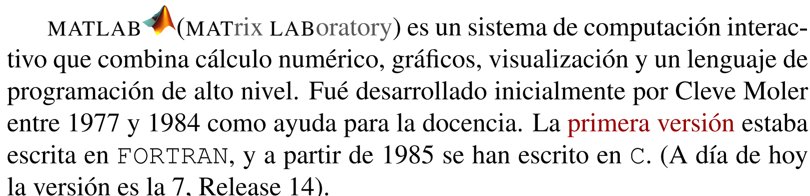
Inconvenientes y ventajas de los lenguajes compilados

- Puede ser un proceso *laborioso*, donde es posible cometer errores en la programación de algoritmos[1] [2], y es conveniente depurar correctamente los códigos realizados.
- Es preciso *linkar* con librerías numéricas y gráficas, y es usual que dependan del lenguaje de programación y del sistema operativo.
- Las herramientas de *debugging* y *profiling* no suelen ser fáciles de utilizar.
- Con adecuadas optimizaciones, son rápidos en ejecución.
- Pueden ser paralelizados (`MPI, PVM, OpenMp`).
- Posibilidad de uso de precisión extendida en los cálculos.

---

[1] En 1962, el Mariner 1 tuvo un fallo debido al código `FORTRAN` `DO I=1.3`

[2] El Mariner 18 se perdió debido a un NOT desaparecido en un programa



MATLAB (MATrix LABoratory) es un sistema de computación interactivo que combina cálculo numérico, gráficos, visualización y un lenguaje de programación de alto nivel. Fué desarrollado inicialmente por Cleve Moler entre 1977 y 1984 como ayuda para la docencia. La primera versión estaba escrita en FORTRAN, y a partir de 1985 se han escrito en C. (A día de hoy la versión es la 7, Release 14).

MATLAB puede realizar operaciones aritméticas reales y complejas con matrices y escalares, resolver sistemas de ecuaciones no lineales, integrar funciones y sistemas de ecuaciones diferenciales y algebraicas, ... Es lo que suele llamarse un **manipulador numérico**.

Hay manipuladores numéricos tipo MATLAB como:

- `Octave`: Casi un clon de MATLAB. **Gratuito**.
- `Scilab`: Muchas prestaciones. **Gratuito**.
- `Euler`: Potente y de tamaño reducido. **Gratuito**.
- `O-Matrix`: Rápido. **Comercial**
- `Interactive Data Language`: **Comercial**
- `Gauss`: **Comercial**

Inconvenientes y ventajas de MATLAB

- Es algo caro.
- Como es un lenguaje interpretado, suele ser más lento que los compilados. [3]
- Permite el desarrollo de códigos de forma rápida utilizando un lenguaje de muy alto nivel.
- Los códigos escritos son transportables entre distintos sistemas operativos.
- Gráficos de muy buena calidad.
- Utilidades de *debugging* y *profiling*.
- Hay Toolboxes específicas para ampliar las prestaciones de MATLAB (Simulink, Optimization, Symbolic, ...).

[3] Puede utilizarse el compilador de MATLAB y/o linkar con códigos FORTRAN o C mediante ficheros MEX para aumentar la velocidad.

Facilidad de uso

Ejemplo.- Consideremos el problema de sumar a una matriz el producto de otras dos, todas ellas de dimensión $n \times n$. El programa en FORTRAN 9X, suprimiendo las líneas relacionadas con el interface de subrutinas, para $n = 1000$ podría ser:

```
program blas
integer, parameter :: d0 = selected_real_kind(14)
integer :: m, n, k, lda, ldb, ldc
character(len=1) :: transa, transb
integer, dimension(1) :: iseed
real(kind = d0),dimension(1000,1000) :: a, b, c
real(kind = d0) :: alpha, beta
real :: s, s1, s2
!============================================
iseed=1
n=1000
lda=n
ldb=n
ldc=n
m=n
k=n
```

```
   call random_seed(put=iseed)
   call random_number(a(1:n,1:n))
   call random_number(b(1:n,1:n))
   call random_number(c(1:n,1:n))
   transa="n"
   transb="n"
   alpha = 1.0_d0
   beta = 1.0_d0
 ! ======= LAPACK ===================
   CALL CPU_TIME(s)
   CALL dgemm(TRANSA, TRANSB, M, N, K, ALPHA, A, &
  & LDA, B, LDB, BETA,C, LDC )
   CALL CPU_TIME(s1)
   print*, s1-s
   print*, c(1,1)
 ! ======= MATMUL (INTRINSECA) ======
   CALL CPU_TIME(s1)
   c = c + matmul(a,b)
   CALL CPU_TIME(s2)
  print*, s2-s1
  print*, c(1,1)

  end program blas
```

mientras que en MATLAB es:

```
>> a=rand(1000);  b=rand(1000);  c=rand(1000);
>> tic; c=c+a*b; toc
```

En la Tabla **1**, podemos ver el resultado de la operación anterior en segundos de CPU[4] con distintas versiones de MATLAB y utilizando un compilador de FORTRAN 9x con la función intrínseca matmul. La diferencia esencial radica en el uso de las librerías ATLAS (Automatic Tuned Linear Algebra Subroutines) que acelera este tipo de operaciones matriciales.

Tabla 1

FORTRAN9X matmul		FORTRAN9X	MATLAB	MATLAB
NO Optim.	Optim.	ATLAS	5.1	6.1 (ATLAS)
39.84	5.81	3.19	29.96	3.32

[4]Tiempo relativo a un ordenador Pentium III 866 Mhz con 512 Mb de memoria y sistema operativo Linux–2.4.18

Ejemplo. Consideremos el problema de calcular los valores y vectores propios de la matrices de Hilbert de dimensiones $n = 300, \ldots, 600$. El programa en FORTRAN 9X usando la subrutina dsyev de la librería LAPACK y suprimiendo las líneas relacionadas con el interface de subrutinas externas, para $n = 300$ podría ser:

```
program euler
 integer, parameter :: d0 = selected_real_kind(14)
 integer :: n, lda, info, lwork
 character(len=1) :: jobz, uplo
 INTEGER :: i, j
 integer, dimension(1) :: iseed
 real(kind = d0),dimension(300,300) :: a
 real(kind = d0),dimension(300) :: w
 real(kind = d0),dimension(2400) :: work
 real :: s, s1
 iseed =1
 n=300
 lda=n
```

```
  do j=1, n
  do i=1, n
   a(i,j) = 1.0_d0/(i+j-1.0_d0)
  end do
  end do
  jobz = "v"
  uplo = "u"
  lwork=2400
  CALL CPU_TIME(s)
  call dsyev(jobz,uplo,n,a,lda,w,work,lwork,info)
  CALL CPU_TIME(s1)
  print*, s1-s
  print*, w(1)
 end program euler
```

y en MATLAB es:

```
>> a=hilb(500);
>> tic; [b,c]=eig(a); toc
```

En la tabla **2** podemos ver las diferencias en tiempo de CPU (segundos), al resolver el problema anterior para diversos valores de la dimensión n. Puede apreciarse que en este caso MATLAB es más rápido que `FORTRAN 9X`.

Tabla 2

n	`FORTRAN 9X`	MATLAB 5.1	MATLAB 6.1
300	1.47	2.11	1.44
400	4.02	5.57	3.54
500	9.42	10.78	6.97
600	15.61	19.76	12.04

Que no es MATLAB

MATLAB está considerado, quizá, el mejor manipulador numérico y con las Toolboxes[5] de que dispone, sus prestaciones se amplían de manera considerable.

Aunque con las Toolboxes:

- `Symbolic`
- `Extend Symbolic`

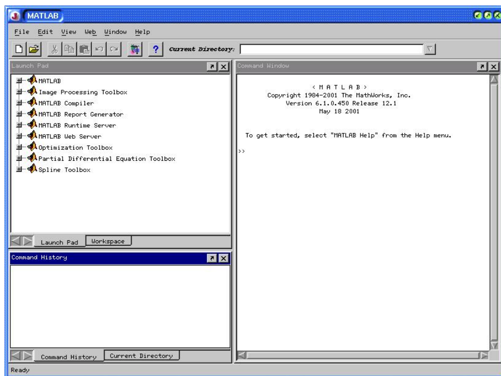
se puede acceder a MAPLE[6], no deberían utilizarse de forma sistemática como sustitución del mismo, ya que suelen ser de una versión anterior y la programación es más complicada que en MAPLE.

[5]Entre otras toolboxes, podemos destacar: Simulink, Stateflow, Real–Time Workshop, Communications Blockset, Control System, Financial Time Series, C/C++ Library, Image Processing, Neural Network, Optimization, Power System Blockset, Wavelet, Spline, ...

[6]Suele ser recíproco

2. Básicos

Al ejecutar MATLAB aparece una pantalla de presentación[7] similar a:



pudiendo introducir datos y órdenes específicas[8].

---

[7]En entorno gráfico, que en MATLAB v6.x, v7.x, depende de JAVA

[8]La primera orden suele ser `bench`, que mide el rendimiento de nuestro ordenador.

Las operaciones aritméticas básicas son: `+ - * / ^` que pueden combinarse con paréntesis. El orden de prioridad en la evaluación de una expresión aritmética es:

- cantidades entre paréntesis
- `^`
- `* /` de izquierda a derecha
- `+ -` de izquierda a derecha

En la práctica, MATLAB trabaja con aritmética compleja, y cuando usa aritmética real, no almacena la parte imaginaria.

Reconoce los siguientes tipos de variables:

reales	3.1415, 1.42e+8
complejas	1.23 + 5.98i, 1 + 1i
caracter	`'entre_comillas simples'`
lógicas	true (1), false (0)

MATLAB trabaja internamente con aritmética IEEE ($\simeq$ 15 cifras significativas). La forma de imprimir los resultados en pantalla se controla con la sentencia `format`

`format short`	3.1416
`format long`	3.14159265358979
`format short g`	La mejor de las dos anteriores con 5 dígitos
`format short e`	3.1416e+00
`format long e`	3.14159265358979e+00
`format long g`	La mejor de las dos anteriores con 15 dígitos
`format hex`	400921fb54442d18
`format bank`	3.14
`format rat`	355/113

En cuanto al espaciado vertical, `format compact` suprime las líneas en blanco extras y `format loose` las repone (defecto).

Los nombres de variables pueden contener cualquier combinación de letras, números y el símbolo _, aunque siempre comenzando por una letra, diferenciando entre mayúsculas y minúsculas.

MATLAB tiene variables predefinidas como:

`ans`	último cálculo no asignado
`pi`	$\pi = 3.141\ldots$
`i`, `j`	$\sqrt{-1}$ unidad imaginaria
`Inf`, `NaN`	infinito y NaN (aritmética IEEE)
`eps`	distancia del 1 hasta el siguiente número en coma flotante $2^{-52} \simeq 2.22 \times 10^{-16}$ (doble de la unidad de redondeo)
`realmax`	$\simeq 1.80 \times 10^{308}$
`realmin`	$\simeq 2.23 \times 10^{-308}$

```
>> i=3
i =
   3
>> a=1+5*i, b=1+5j, c=1+i, d=1+1i
a =
   16
b =
   1.0000 + 5.0000i
c =
   4
d =
   1.0000 + 1.0000i
>> 1+1i
ans =
   1.0000 + 1.0000i
>> a=pi+ans
a =
   4.1416 + 1.0000i
```

Aritmética IEEE

MATLAB ejecuta todas las operaciones en aritmética en doble precisión de acuerdo al estándar IEEE y a partir de la versión 6 la función `isieee` siempre da el valor 1 (cierto).

```
>> isieee
Warning:  ISIEEE is obsolete.  MATLAB only runs on IEEE
machines.
ans =
     1
```

- MATLAB almacena cada número en 8 bytes, y el rango de los números en coma flotante es aproximadamente [`realmin`, `realmax`] con unidad de redondeo $2^{-53} \simeq 1.11 \times 10^{-16}$.
- La función `eps` da como resultado la distancia desde el 1.0 al siguiente número en coma flotante, y es el doble de la unidad de redondeo.
- De acuerdo al estándar IEEE, el resultado de un cálculo es un número en coma flotante o en excepciones `Inf`, `NaN`.
- Si el cálculo es mayor que `realmax`, entonces el resultado es `Inf`, si es menor que `-realmax` es `-Inf` y para cálculos no definidos es `NaN`.

- Notar que muchas de las propiedades usuales de los números reales (asociativa, distributiva, . . .) se pierden en aritmética IEEE.

```
>> -1+(1+eps/2),(-1+1)+eps/2 % NO ASOCIATIVA!!
ans =
   0
ans =
   1.1102e-16
>> format hex; 1+eps, 1+eps/2
ans =
   3ff0000000000001
ans =
   3ff0000000000000
>> 0.6-0.2-0.4
ans =
   -5.5511e-17
>> 1-1/3-1/3-1/3
ans =
   1.1102e-16
```

```
>> 1.1*realmax
ans =
   Inf
>> -4*realmax
ans =
   -Inf
>> 1/0
Warning:  Divide by zero.
ans =
   Inf
>> Inf-Inf
ans =
   NaN
>> Inf/Inf
ans =
   NaN
>> 1^Inf
ans =
   NaN
>> NaN + NaN
ans =
   NaN
```

Notas

- Si queremos suprimir la salida de las variables por pantalla, hay que añadir un “;” al final de cada sentencia. Esto es útil cuando se ejecuta un fichero de órdenes en MATLAB.
- Pueden escribirse varias sentencias en una línea separadas por “;” o “,”.
- La sentencia `clc` borra la ventana de comandos.
- Si no se asigna un resultado a una variable, por defecto lo asocia a la variable `ans`.
- Para continuar una línea hay que usar “...” al final de cada línea que se quiera continuar.
- Los comentarios en MATLAB deben ir precedidos por el símbolo `%`.
- La orden `clear all` borra el contenido de todas las variables, funciones, ... Resulta interesante cuando se depuran programas.

```
>> x=1; y=2; ...
z=3; % esto es un comentario, w=9
>> x+y+z, z^3
   ans = 6
   ans = 27
>> u=ans*9
   u = 243
```

Utilidades

- Emplear `help tópico` o `helpwin tópico`. Nos dará información sobre `tópico` y podrá sacarnos de algún aprieto.

```
>> help atan
   ATAN Inverse tangent.
   ATAN(X) is the arctangent of the elements of X.
   See also ATAN2.
```

- Para abortar una tarea en MATLAB utilizar CTRL + c.
- Para medir el tiempo de cpu consumido por un bloque de sentencias pueden usarse:

```
>> tic; bloque de sentencias; toc
>> ti=cputime; bloque de sentencias; cputime-ti
```

- entrada de datos por teclado. Para este fin está la función `input`, con sintaxis:

```
>> variable= input('texto ');
texto 2.345
>> variable=input('texto ','s');
```

siendo la primera para valores numéricos y la segunda para variables caracter.

- salida de datos formateados. La función adecuada es:

```
fprintf('format',vars)
```

donde `vars` es una lista de variables separadas por comas y algunos de los elementos que pueden escribirse en `format` vienen dados en la tabla siguiente:

`%P.Qe`	notación exponencial
`%P.Qf`	notación decimal
`%P.Qg`	la más corta de las anteriores
`%Pd`, `%Pi`	imprime un entero
`%s`	imprime una variable caracter
`\n`	salto de línea

con P y Q números enteros no negativos. P indica la longitud mínima de caracteres que se usará para escribir el número y Q el número de dígitos de la parte decimal.

```
>> x=1007.46; y=pi; j=78;
>> fprintf( 'x= %8.2e y=%12.8f j=%i\n', x,y,j)
   x= 1.01e+03 y= 3.14159265 j=78
```

- La sentencia disp(X) es una forma simple de imprimir la matriz X sin su nombre o un texto si X es una variable caracter.

```
>> X=hilb(4)     % hilb_ij = 1/(i+j-1)
X =
  1.0000 0.5000 0.3333 0.2500
  0.5000 0.3333 0.2500 0.2000
  0.3333 0.2500 0.2000 0.1667
  0.2500 0.2000 0.1667 0.1429
>> disp(X); disp('imprime esto');
  1.0000 0.5000 0.3333 0.2500
  0.5000 0.3333 0.2500 0.2000
  0.3333 0.2500 0.2000 0.1667
  0.2500 0.2000 0.1667 0.1429

imprime esto
```

En MATLAB están implementadas la mayoría de las funciones matemáticas más comunes[9]:

```
  Trigonometric.
   sin         - Sine.
   sinh        - Hyperbolic sine.
   asin        - Inverse sine.
   asinh       - Inverse hyperbolic sine.
   cos         - Cosine.
   cosh        - Hyperbolic cosine.
   acos        - Inverse cosine.
   acosh       - Inverse hyperbolic cosine.
   tan         - Tangent.
   tanh        - Hyperbolic tangent.
   atan        - Inverse tangent.
   atan2       - Four quadrant inverse tangent.
   atanh       - Inverse hyperbolic tangent.
```

[9] `help elfun, help specfun`

```
 sec        - Secant.
 sech       - Hyperbolic secant.
 asec       - Inverse secant.
 asech      - Inverse hyperbolic secant.
 csc        - Cosecant.
 csch       - Hyperbolic cosecant.
 acsc       - Inverse cosecant.
 acsch      - Inverse hyperbolic cosecant.
 cot        - Cotangent.
 coth       - Hyperbolic cotangent.
 acot       - Inverse cotangent.
 acoth      - Inverse hyperbolic cotangent.

Exponential.
 exp        - Exponential.
 log        - Natural logarithm.
 log10      - Common (base 10) logarithm.
 log2       - Base 2 logarithm and dissect floating
              point number.
 pow2       - Base 2 power and scale floating
              point number.
 sqrt       - Square root.
 nextpow2   - Next higher power of 2.
```

```
Complex.
 abs        - Absolute value.
 angle      - Phase angle.
 complex    - Construct complex data from real
              and imaginary parts.
 conj       - Complex conjugate.
 imag       - Complex imaginary part.
 real       - Complex real part.
 unwrap     - Unwrap phase angle.
 isreal     - True for real array.
 cplxpair   - Sort numbers into complex conjugate pairs.

Rounding and remainder.
 fix        - Round towards zero.
 floor      - Round towards minus infinity.
 ceil       - Round towards plus infinity.
 round      - Round towards nearest integer.
 mod        - Modulus (signed remainder after division).
 rem        - Remainder after division.
 sign       - Signum.
```

```
Specialized math functions.
 airy      - Airy functions.
 besselj   - Bessel function of the first kind.
 bessely   - Bessel function of the second kind.
 besselh   - Bessel functions of the third kind
             (Hankel function).
 besseli   - Modified Bessel function of the first kind.
 besselk   - Modified Bessel function of the second kind.
 beta      - Beta function.
 betainc   - Incomplete beta function.
 betaln    - Logarithm of beta function.
 ellipj    - Jacobi elliptic functions.
 ellipke   - Complete elliptic integral.
 erf       - Error function.
 erfc      - Complementary error function.
 erfcx     - Scaled complementary error function.
 erfinv    - Inverse error function.
 expint    - Exponential integral function.
 gamma     - Gamma function.
 gammainc  - Incomplete gamma function.
 gammaln   - Logarithm of gamma function.
 legendre  - Associated Legendre function.
 cross     - Vector cross product.
 dot       - Vector dot product.
```

```
Number theoretic functions.
 factor    - Prime factors.
 isprime   - True for prime numbers.
 primes    - Generate list of prime numbers.
 gcd       - Greatest common divisor.
 lcm       - Least common multiple.
 rat       - Rational approximation.
 rats      - Rational output.
 perms     - All possible permutations.
 nchoosek  - All combinations of N elements taken K at
             a time.
 factorial - Factorial function.

Coordinate transforms.
 cart2sph  - Transform Cartesian to spherical coordinates.
 cart2pol  - Transform Cartesian to polar coordinates.
 pol2cart  - Transform polar to Cartesian coordinates.
 sph2cart  - Transform spherical to Cartesian coordinates.
 hsv2rgb   - Convert hue-saturation-value colors to
             red-green-blue.
 rgb2hsv   - Convert red-green-blue colors to
             hue-saturation-value.
```

3. Vectores y Matrices

En MATLAB tienen un lugar privilegiado los vectores y matrices. De hecho, podríamos considerar que es el único tipo de variable que existe[10] (un escalar se considera como una matriz 1×1).

```
>> vector_fila = [1,2,3]
vector_fila =
  1 2 3
>> vector_columna = [1;2;3]
vector_columna =
  1
  2
  3
>> matriz = [11,12,13; 21,22,23; 31,32,33]
matriz =
  11 12 13
  21 22 23
  31 32 33
```

Los elementos de una matriz se introducen por filas, separándolas con `;` y en cada fila se separan por comas o espacios en blanco.

[10]Chiste de Cleve Moler

Hay que tener especial precaución con los espacios en blanco al definir un vector o una matriz

```
>> x=[1, 2, 4 +7]
x =
  1 2 4 7
>> x=[1, 2, 4 + 7]
x =
  1 2 11
>> x=[1, 2, (4 +7)]
x =
  1 2 11
>> matriz = [1 ; 2 +7]
???  Error using ==> vertcat
All rows in the bracketed expression must have
the same number of columns.
```

La notación `a:b:c` sirve para construir vectores fila comenzando en `a`, incrementándose por el valor `b` hasta `c`. El comando `whos` lista las variables empleadas, el tamaño, espacio que ocupan y la clase.

```
>> x=1:1:10, y=1:10
x =
  1  2  3  4  5  6  7  8  9  10
y =
  1  2  3  4  5  6  7  8  9  10
>> z=0.1:.05:.3, x1=-1:-2, p=1;
z =
  0.1000 0.1500 0.2000 0.2500 0.3000
x1 =
  Empty matrix:  1-by-0
>> whos
 Name   Size   Bytes   Class

 p      1x1        8   double array
 x      1x10      80   double array
 x1     1x0        0   double array
 y      1x10      80   double array
 z      1x5       40   double array
Grand total is 25 elements using 200 bytes
```

Para convertir un vector fila en un vector columna (o recíprocamente) se utiliza la transposición, que viene denotada por `'`.

```
>> matriz'
ans =
  11 21 31
  12 22 32
  13 23 33
>> matriz_compleja=[ 1+1i, 1+2i; 2+1i, 2+2i]
matriz_compleja =
  1 + 1i 1 + 2i
  2 + 1i 2 + 2i
>> matriz_compleja' % compleja transpuesta conj.
ans =
  1 - 1i 2 - 1i
  1 - 2i 2 - 2i
>> matriz_compleja.'  % compleja transpuesta
ans =
  1 + 1i 2 + 1i
  1 + 2i 2 + 2i
```

Podemos efectuar las siguientes operaciones entre matrices, siempre que estén definidas:

`c=a ± b`	suma (resta) ($c_{ij} = a_{ij} \pm b_{ij}$) si `a` es escalar, ($c_{ij} = a \pm b_{ij}$)
`c=a*b`	producto matricial
`c=a.*b`	producto de matrices elemento a elemento ($c_{ij} = a_{ij} * b_{ij}$)
`c=a/b`	equivalente a `c=(inverse(b')*a')'`
`c=a./b`	cociente elemento a elemento ($c_{ij} = a_{ij}/b_{ij}$) si `a` es escalar, ($c_{ij} = a/b_{ij}$)
`c=a\b`	equivalente a `c=inverse(a)*b`
`c=a^k`	potencia k–ésima de la matriz `a`
`c=a.^b`	potenciación elemento a elemento ($c_{ij} = a_{ij}^{b_{ij}}$)
`c=a'`	matriz compleja conjugada transpuesta (`ctranspose(a)`)
`c=a.'`	matriz transpuesta (`transpose(a)`)

```
>> u=[1,2], v=[1;2] % v=(1:2)'
u =
  1 2
v =
  1
  2
>> u'*v
???  Error using ==> *
Inner matrix dimensions must agree.
>> u*v
ans =
  5
>> v*u
ans =
  1 2
  2 4
>> x=(-1:0.5:1)
x =
  -1.000 -0.500 0.000 0.500 1.000
>> y=x .* sin(pi*x)
y =
  0.0000 0.5000 0 0.5000 0.0000
```

```
>> x=[0.1, 0.01, 0.001, 0.0001]
x =
  0.1000 0.0100 0.0010 0.0001
>> y=sin(pi*x) ./ x
y =
  3.0902 3.1411 3.1416 3.1416
>> format long; y - pi
ans =
  -5.1423e-02 -5.1675e-04 -5.1677e-06 -5.1677e-08
>> format short; a=[1,2; 3,4]
a =
    1   2
    3   4
>> a^pi
ans =
  1.0e+02 *
    0.4696 - 0.0001i  0.6851 + 0.0001i
    1.0276 + 0.0001i  1.4972 - 0.0000i
>> a.^a
ans =
     1     4
    27   256
```

Hay varias funciones que permiten manejar y definir matrices de forma cómoda, como:

ones(nf,nc)	matriz de unos de dimensión nf×nc
zeros(nf,nc)	matriz de ceros de dimensión nf×nc
eye(nf,nc)	matriz identidad de dimensión nf×nc
[nf,nc]=size(A)	número de filas y de columnas de la matriz A
diag(v)	matriz diagonal con vector v en la diagonal
hilb(n)	matriz de Hilbert de dimensión n×n
norm(A,p)	norma p de A. p=1,2,Inf,'fro', ...
rand(nf,nc)	matriz de números aleatorios uniformemente distribuidos en el intervalo (0,1) de dimensión nf×nc
randn(nf,nc)	matriz de números normalmente distribuidos con media nula y varianza uno de dimensión nf×nc

Las funciones ones, zeros, eye, rand, randn con un solo argumento dan una matriz cuadrada de dimensión ese argumento.

`sum(A)`	si `A` es una matriz $\mathtt{nf} \times \mathtt{nc}$, el resultado es un vector fila `s` con $\mathtt{s}_i = \sum_{j=1}^{\mathtt{nf}} A(j,i)$, $i = 1, \ldots, \mathtt{nc}$. Si `A` es un vector, es la suma de las componentes de `A`.
`prod(A)`	análogo a `sum` pero con productos.
`max(A)`	si `A` es una matriz $\mathtt{nf} \times \mathtt{nc}$, el resultado es un vector fila `s` con $\mathtt{s}_i = \max_{1 \leq j \leq \mathtt{nf}} A(j,i)$, $i = 1, \ldots, \mathtt{nc}$. Si `A` es un vector, es el máximo de las componentes de `A`.
`min(A)`	análogo a `max` pero con mínimo. Si `A` es compleja ver `help max`
`linspace(a,b,n)`	genera una red de `n` puntos igualmente espaciados entre `a` y `b`.
`logspace(a,b,n)`	genera una red de `n` puntos logarítmicamente espaciados entre $10^{\mathtt{a}}$ y $10^{\mathtt{b}}$.
`repmat(A,nf,nc)`	genera una matriz por bloques `nf`×`nc` con el valor de `A` en cada bloque.

Una imagen digital se puede considerar una matriz, y las operaciones que hagamos sobre dicha matriz representarán cambios en la imagen (brillo, contraste, ...)[11].

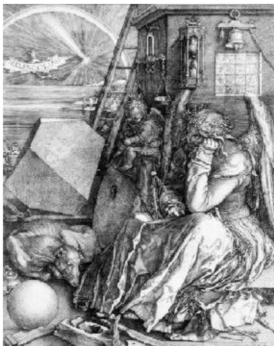
```
>> % imagen 256 × 256 escala de grises.
>> D = imread('lenna.jpg'); % Playmate Noviembre 1972
>> imshow(D); % original (grises).   D_ij ∈ [0,255]
>> imshow(255-D); % negativo  version 7.x
>> imshow(uint8(255-double(D))); % version 6.x
```

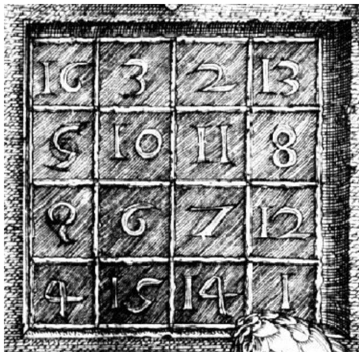
[11]Requiere Image_Toolbox

```
>> A=ones(2,4)
A =
    1 1 1 1
    1 1 1 1
>> B=zeros(size(A))
B =
    0 0 0 0
    0 0 0 0
>> C=rand(2,4)
C =
    0.25150 0.28437 0.56549 0.60927
    0.18432 0.39318 0.32908 0.79883
>> D=eye(2)
D =
    1 0
    0 1
>> % normas 1, 2 e infinito
>> norm(C,1), norm(C,2), norm(C,'inf')
    ans = 1.4081
    ans = 1.3105
    ans = 1.7106
```

```
>> sum(A), sum(A'), sum(sum(A))
ans =
   2 2 2 2
ans =
   4 4
ans =
   8
>> c=magic(4) % help magic
c =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
>> sum(c), sum(c'), sum(diag(c)), sum(sum(c))
ans =
   34 34 34 34
ans =
   34 34 34 34
ans =
   34
ans =
   136
```

Melancolía. Alberto Durero (1471–1528)






```
>> magic(4) % help magic
ans =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
```

Ejemplo.- Definir la matriz $A = \left[\begin{array}{c|c} A_{11} & I_2 \\ \hline 1_{2\times 2} & A_{22} \end{array}\right]$ de bloques 2×2, donde todos sus bloques son matrices 2×2 y A_{ii} son matrices aleatorias.
Construir la matriz $\left[\begin{array}{c|c|c} I_2 & I_2 & I_2 \end{array}\right]$

```
>> c=[rand(2), eye(2); ones(2), rand(2)]
c =
   0.0159   0.8125   1.0000        0
   0.6283   0.2176        0   1.0000
   1.0000   1.0000   0.9501   0.6068
   1.0000   1.0000   0.2311   0.4860
>> repmat(eye(2),1,3)
ans =
    1 0 1 0 1 0
    0 1 0 1 0 1
>> repmat(pi,2,3)
ans =
    3.1416 3.1416 3.1416
    3.1416 3.1416 3.1416
```

Es más rápido usar `repmat(pi,m,n)` que `pi*ones(m,n)` ya que se evitan los productos.

```
>> tic; a=pi*ones(3000,3000); toc
elapsed_time =
   1.5654
>> tic; a=repmat(pi,3000,3000); toc
elapsed_time =
   0.6842
```

Definidos los vectores columna `x` e `y`, puede calcularse su producto escalar como `x'*y` o `dot(x,y)`, así como la matriz de rango uno `x*y'`. La función `cross(x,y)` da el producto vectorial (sólo en $\mathbb{R}^3$).

```
>> x=[1 1 1]'; y=[1 -2 1]';
>> x'*y
ans =
     0
>> x*y'
ans =
   1   -2   1
   1   -2   1
   1   -2   1
>> dot(x,y)
ans =
     0
>> cross(x,y) % Sólo para dimensión 3
ans =
    3
    0
   -3
```

Ejemplo. Comprobar las condiciones de orden 4 del método RK clásico, dado por la tabla de coeficientes en notación de Butcher:

$$\begin{array}{c|cccc} 0 & 0 & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 2/6 & 2/6 & 1/6 \end{array} = \begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

que son:

$$\begin{array}{llll} b^T e = 1, & b^T c = 1/2, & b^T c^2 = 1/3, & b^T c^3 = 1/4, \\ b^T Ac = 1/6, & b^T Ac \cdot c = 1/8, & b^T Ac^2 = 1/12, & b^T A^2 c = 1/24, \end{array}$$

con $e = (1, \ldots, 1)^T$ y $c = Ae$.

```
>> s=4; % número de etapas del RK clásico
>> A=[0,0,0,0; 1/2,0,0,0; 0,1/2,0,0; 0,0,1,0];
>> e=ones(s,1); b=[1;2;2;1]/6;
>> c=A*e; Ac=A*c; % condiciones hasta orden 4
>> Phi=[e,c,c.^2,Ac,c.^3,Ac.*c,A*c.^2,A*Ac];
>> gamma=1./[1,2,3,6,4,8,12,24];
>> resul=b' * Phi - gamma
resul =   0 0 0 0 0 0 0 0
```

Podemos extraer submatrices y subvectores con la notación `:`, e incluso borrar filas o columnas asignándolas a un vector de dimensión nula, por ejemplo:

```
>> A=[11 12 13; 21 22 23; 31 32 33];
>> A(:,1)
ans =
  11
  21
  31
>> A(2,:)
ans =
  21 22 23
>> A(2,:)=[]
A =
  11 12 13
  31 32 33
>> A(:,3)=[]
A =
  11 12
  31 32
```

El manejo de índices de matrices y vectores es muy polivalente. Como ejemplo, damos el conocido *Tony's Trick*, para generar vectores y matrices con copias de un vector.

```
>> x=rand(3,1);
x =
    0.9218
    0.7382
    0.1763
>> % extrae la primera columna 3 veces
>> x(:,[1,1,1])
ans =
   0.9218   0.9218   0.9218
   0.7382   0.7382   0.7382
   0.1763   0.1763   0.1763
>> y=rand(1,3)
y =
    0.4103 0.8936 0.0579
>> y(:,[1:3])
ans =
    0.4103 0.8936 0.0579
>> y(:,[1:3,1:3])
ans =
    0.4103 0.8936 0.0579 0.4103 0.8936 0.0579
```

```
>> z=rand(2,3);
>> % z(:)  es el vector columna formado por las
>> % columnas concatenadas de la matriz z
>> z(:)
```

```
>> x=rand(1,3), ind=1:2;
>> x(ind)
   0.1389 0.2028
>> x([ind,ind])
   0.1389 0.2028 0.1389 0.2028
>> x([ind;ind])
   0.1389 0.2028
   0.1389 0.2028
```

```
>> A=magic(3); b=ones(3,1);
>> A\b, inv(A)*b
>> % pinv es la pseudo-inversa de una matriz (1), (2)
>> b\A, pinv(b)*A
>> x1 = -1:-2; % matriz con una dimensión nula
>> y1 = -1:-4; % matriz con una dimensión nula
>> x1*y1' % ?
>> x1'*y1 % ?
```

$x_i = (b_i - \sum_{j=i+1}^{n} a_{ij} x_j)/a_{ii}, \quad i = n, n-1, \ldots$

Operaciones más especializadas como el producto de Kronecker entre las matrices $A \in \mathbb{R}^{m\times n}$ y $B \in \mathbb{R}^{p\times q}$, dado por:

$$A \otimes B = \left[\begin{array}{c|c|c|c} a_{11}B & a_{12}B & \dots & a_{1n}B \\ \hline a_{21}B & a_{22}B & \dots & a_{2n}B \\ \hline \vdots & & \ddots & \vdots \\ \hline a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{array}\right] \in \mathbb{R}^{m\,p\times n\,q},$$

también están definidas en MATLAB:

```
>> A=[11 12; 21 22]; B=[1 2 3; 4 5 6];
>> C=kron(A,B)
C =
   11    22    33   12    24    36
   44    55    66   48    60    72
   21    42    63   22    44    66
   84   105   126   88   110   132

>> [nf,nc]=size(C)
nf =
    4
nc =
    6
```

Hay diferencia entre las funciones `expm` y `exp`. La primera calcula la exponencial de una matriz cuadrada, mientras que la segunda calcula la exponencial de cada uno de sus elementos.

```
>> format short g;
>> A=[9 4; 0 9];
>> B=expm(A)
B =
   8103.1    32412
        0   8103.1
>> C=exp(A)
C =
   8103.1   54.598
        1   8103.1
>> logm(B) % logaritmo de la matriz B
ans =
            9   4
   4.9304e-31   9
>> A=[10, 1; 1, 10]; A^pi
ans =
   1.4321e+03   4.3703e+02
   4.3703e+02   1.4321e+03
```

```
>> expm(pi*logm(A))
ans =
   1.4321e+03   4.3703e+02
   4.3703e+02   1.4321e+03
```

`funm(A,@fun)`, siendo `A` una matriz cuadrada, evalúa la versión matricial de `fun` (sin, cos, log, exp, sinh, cosh) [12]

```
>> A=[9 4; 0 9];
>> B=sqrtm(A)
B =
   3.0000   0.6667
        0   3.0000
>> B*B-A
ans =
   0   0
   0   0
>> C=funm(rand(2), @sin)
C =
   0.7643   0.4424
   0.1685   0.4259
```

[12]Para funciones generales es necesario suministrar la función `fun(A,k)` para evaluar la derivada `k`–ésima de `fun`

Con la función `diag` podemos obtener las diagonales de una matriz `A`, de acuerdo a:

$$\texttt{diag}(\texttt{A},\texttt{k}) = \begin{cases} \text{k–ésima diagonal por encima de la principal si } \mathrm{k} > 0 \\ \text{diagonal principal si } \mathrm{k} = 0 \\ \text{k–ésima diagonal por debajo de la principal si } \mathrm{k} < 0 \end{cases}$$

```
>> A=[11 12 13; 21 22 23; 31 32 33];
>> diag(A)
ans =
    11
    22
    33
>> diag(A,-1)
ans =
    21
    32
>> diag(A,1)
ans =
    12
    23
```

y también para definir matrices diagonales. La función `reshape(A,nf,nc)` cambia el tamaño de la matriz `A` para que sea `nf`×`nc`.

```
>> A = diag([1,2,1],0) % A=diag([1,2,1])
ans =
    1  0  0
    0  2  0
    0  0  1
>> A = diag([2,2,2],1) + diag([-1,-1,-1],-1)
A =
     0     2     0     0
    -1     0     2     0
     0    -1     0     2
     0     0    -1     0
>> % el número de elementos debe ser igual en A
>> % y en reshape(A,nf,nc)
>> reshape(A,2,8)
ans =
     0    0    2    -1    0     0    0    2
    -1    0    0     0    2    -1    0    0
```

Es posible hallar el máximo (mínimo) de los elementos de una matriz, con la función max (min),

```
>> A=[1.1 1.2 1.3; 2.1 2.2 2.3; 3.1 3.2 3.3];
>> max(A), [m, ind]=max(A), max(max(A)), max(A(:))
ans =
    3.1000 3.2000 3.3000
m =
    3.1000 3.2000 3.3000
ind =
    3 3 3
ans =
    3.3000
ans =
    3.3000
```

así como permutar ($i \to n - i + 1$) los elementos de un vector $x \in \mathbb{R}^n$

```
>> x=1:4;
>> x(end:-1:1)
ans =
    4 3 2 1
```

Ejemplo.- Permutar filas/columnas de una matriz.

```
>> A=[11 12; 21 22];
A =
   11   12
   21   22
>> A([1,2],:)  = A([2,1],:)  % permutar filas
A =
   21   22
   11   12
>> A(:,[1,2]) = A(:,[2,1]) % permutar columnas
A =
   22   21
   12   11
```

Ejemplo.- Localización del máximo de los elementos de una matriz aleatoria de dimensión 100×100.

```
>> A = rand(100);
>> [x, i]= max(A);
>> [y, j]= max(x);
>> i(j), j, A(i(j),j), y % A(11,45)
ans =
   11
j =
   45
ans =
   0.9998
y =
   0.9998
```

Hallaremos una aproximación a la derivada primera de una función diferenciable $f(x)$ en cada punto $a + ih, i = 1, \ldots, n$, con $h = (b - a)/n$ dada por

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h},$$

utilizando la orden `diff`. Si `f` es un vector, `diff(f)` es un vector de una dimensión menos que `f` con componentes

$$(\mathtt{f}(2) - \mathtt{f}(1), \mathtt{f}(3) - \mathtt{f}(2), \ldots, \mathtt{f}(\mathtt{n}) - \mathtt{f}(\mathtt{n}-1)).$$

```
% a=0, b=4, h=1/100, f(x) = sin(x)
>> x=a:h:b; % x=linspace(0,4,401);
>> y=sin(x);
>> deri = diff(y)/h; % aproximación a cos(x)
>> max( abs(deri - cos(x(1:end-1))))
ans =
    0.0050
```

que concuerda con la fórmula del error

$$\max|-\frac{h}{2}f''(\eta)| = \max|-\frac{h}{2}\sin(\eta)| = \frac{h}{2} = 0.005.$$

Ejemplo.- Utilizar la función `diff` para aproximar la derivada segunda de la función anterior.

```
>> deri2 = diff(deri)/h;
>> max( abs( deri2 - (-sin(x(1:end-2))))) % error
ans =
    0.0100
>> max( abs( deri2 - (-sin(x(2:end-1))) )) % error
ans =
    8.3333e-06 % ?
```

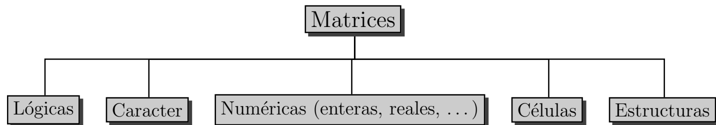
Hay más funciones como:

`tril(A)`	Parte triangular inferior de `A`
`triu(A)`	Parte triangular superior de `A`
`rot90(A)`	Rota 90° la matriz `A`
`sort(A)`	Ordena de menor a mayor cada columna de la matriz `A`
`mean(A)`	Calcula la media de cada columna de la matriz `A`
`var(A)`	Calcula la varianza de cada columna de la matriz `A`
⋮	⋮

4. Estructuras y Células

MATLAB utiliza varios tipos de datos para la generación de matrices[13], entre los que podemos destacar

Tipos de variables en MATLAB



En particular, las estructuras y las células permiten definir matrices heterogéneas, i. e., con elementos de tipos distintos.

Cada elemento de una matriz de estructuras[14] permite almacenar variables de naturaleza distinta. Por ejemplo, para definir los colores RGB de cada pixel de una imagen, podemos usar como campos las coordenadas `x`, `y` y `rgb`.

[13] `help datatypes`

[14] Similar a los tipos derivados en `FORTRAN 9x`

```
>> pixel(1).x=13; pixel(1).y=45; pixel(1).rgb=[.2 .4 .7];
>> pixel(2).x=23; pixel(2).y=43; pixel(2).rgb=[.4 .6 .1];
>> class(pixel)
ans =
   struct
>> pixel
pixel =
   1x2 struct array with fields:
   x
   y
   rgb
>> variable = 'rgb';
>> pixel.(variable)
ans =
   0.2000 0.4000 0.7000
ans =
   0.4000 0.6000 0.1000
>> cat(1,pixel.(variable))
ans =
   0.2000 0.4000 0.7000
   0.4000 0.6000 0.1000
```

La sentencia `cat(DIM,A,B)` concatena las matrices `A` y `B` en la dimensión dada por `DIM`. Por tanto, `cat(2,A,B)=[A,B]` y `cat(1,A,B)=[A;B]`.

La sentencia `struct` permite definir la variable `pixel` en la forma:

```
struct('nombre1',valor1, 'nombre2',valor2, ...)
```

cuyos argumentos son los nombres de los campos seguidos por sus valores entre llaves[15], que deberían ser del mismo tamaño

```
>> pixel=struct('x',{13,23},'y',{45,43}, ...
   'rgb', {[.2 .4 .7], [.4 .6 .1]})
```

Puede emplearse `repmat` para completar los elementos de una estructura

```
>> pixel=repmat(struct('x',{9},'y',{3}, ...
   'rgb',{[.1 .2 .3]}),3,4)
pixel =
   3x4 struct array with fields:
   x
   y
   rgb
>> xvector= cat(2,pixel.x)
xvector =
   9 9 9 9 9 9 9 9 9 9 9 9
```

15 Células (ver página siguiente)

A diferencia de las estructuras, que se accede por campo, en las células se hace por índices, de manera análoga a las matrices. Las células se pueden definir entre llaves,

```
>> cel0={-1:1, 1i, 'hola', rand(3) }   % vector fila
cel0 =
   [1x3 double] [0 + 1.0000i] 'hola' [3x3 double]
>> cel1={-1:1, 1i; 'hola', rand(3) }   % matriz
cel1 =
   [1x3 double] [0 + 1.0000i]
   'hola' [3x3 double]
>> cel1(1,1) % célula
ans =
   [1x3 double]
>> iscell(cel1(1,1))
ans =
   1
>> cel1(1,1)={[pi,pi,pi,1,2]}; % entre llaves
>> cel1{1,1}=[pi,pi,pi,1,2]; % valor de la célula
>> cel1{1,1}
ans =
   3.1416 3.1416 3.1416 1.0000 2.0000
```

Con `celldisplay(C)` se muestran los contenidos de la célula `C`.

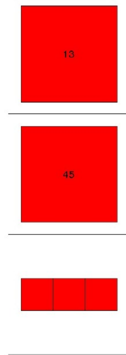
```
>> celldisp(cel)
cel{1,1} =
    3.1416 3.1416 3.1416 1.0000 2.0000
cel{2,1} =
hola
cel{1,2} =
    0 + 1.0000i
cel{2,2} =
  0.6038 0.0153 0.9318
  0.2722 0.7468 0.4660
  0.1988 0.4451 0.4186
```

Hay conversores `mat2cell`, `cellstr`, `struct2cell`, `deal`, ... entre los tipos de variables conocidos por MATLAB. Por ejemplo,

```
>> pixel(1).x=13; pixel(1).y=45; pixel(1).rgb=[.2 .4 .7];
>> cel = struct2cell(pixel)
cel =
  [ 13]
  [ 45]
  [1x3 double]
```

Con objeto de visualizar el contenido de las células disponemos de `cellplot(C)`.

```
>> cellplot(cel)
```



Trabajar con este tipo de variables puede resultar en un **decremento de la velocidad**, aunque MATLAB las emplea en algunas *toolboxes* (database, symbolic).

5. Órdenes básicas de programación

bucle `for` La función `sin(x)` calcula el seno de cada una de las componentes del vector `x`. Esto se podía haber hecho también con un bucle del tipo `for ... end`

```
for i=1:n
  y(i) = sin(x(i));
end;
```

- el uso de este tipo de bucles consume mucho tiempo en MATLAB. Así se recomienda utilizarlos lo mínimo posible[16].
- El índice del bucle puede tener la forma `i=i0:ipaso:ifinal` e incluso `i=[1,3,4,6,7,8]`.

```
>> for i=[1,3,5,8:12]; i, end % imprime i
>> for i=1:10; i, end % imprime i
```

[16]Life is too short to spend writing loops — Getting Started with MATLAB (1998)

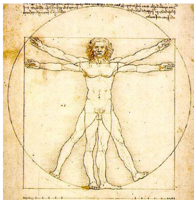
Ejemplo.- Realizar varias iteraciones del esquema iterativo

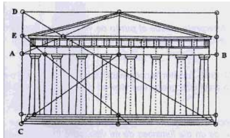
$$x_{n+1} = 1 + 1/x_n, \quad n = 0, 1, 2, \ldots, m,$$

con x_0 dado.

```
>> x=1; m=20; for i=0:m, x=1+1/x; end
>> x
x =
    1.6180 % Número aúreo
```

El número aúreo en distintas manifestaciones:





Los siguientes ejemplos sirven para comprobar que las operaciones vectoriales en MATLAB son bastante más rápidas que los bucles. La vectorización, que puede aplicarse también en FORTRAN 9X, mejora sustancialmente la velocidad de ejecución de los códigos que la usen.

Ejemplo.- Evaluar la función

$$f(x) = \mathrm{e}^{-x} + \sin\left(\frac{\cos(x/100)}{\sqrt{x}} + \log(x)\right),$$

para los enteros positivos $x = 1, 2, \ldots, 5000$.

```
% Código NO vectorizado (tiempo de ejecución 0.57 sg)
% =====================
y=zeros(5000,1);
for i=1:5000;
  y(i)=exp(-i) + sin(cos(i/100)/sqrt(i) + log(i));
end;
```

```
% Código vectorizado (tiempo de ejecución 0.0092 sg)
% ==================
x=(1:5000)';
y=exp(-x) + sin(cos(x/100)./sqrt(x) + log(x));
```

Ejemplo.- Realizar la operación

$$r = \sum_{i=1}^{n} x_i^2,$$

siendo x un vector de números aleatorios de dimensión $n = 10^6$.

```
>> n=10^6; x=rand(n,1); tic; r=sum(x.^2); toc

elapsed_time =
    0.0958

>> n=10^6; x=rand(n,1); ...
tic; r=0; for i=1:n; r=r+x(i)^2; end; toc

elapsed_time =
    6.1481
```

[17]

[17] Ver también `help cumsum`

`relaciones lógicas` MATLAB representa los valores lógicos `true` y `false` por medio de los enteros `true=1`, `false=0`.
Las operaciones lógicas usuales son:

`x == a`	igualdad
`x ~= a`	desigualdad
`x > a`	mayor
`x < a`	menor
`x >= a`	mayor o igual
`x <= a`	menor o igual

mientras que los operadores lógicos son

`&`	`and`
`|`	`or`
`~`	`not`
`xor`	or exclusivo

y que por supuesto pueden aplicarse a matrices.

```
>> A=[-3:-1; 7:9]
A =
   -3   -2   -1
    7    8    9
>> A > 0
ans =
    0  0  0
    1  1  1
>> (A > 0) & (A < 8)
ans =
    0  0  0
    1  0  0
>> (A > 1) | (A < -2) % Es conveniente poner paréntesis
ans =
    1  0  0
    1  1  1
>> xor(A>-2, A<5) % ~((A>-2)&(A<5))
ans =
    1  1  0
    1  1  1
```

También están las funciones `all(x)`, que es cierta si todos los elementos del vector `x` son no nulos, y `any(x)` si algún elemento de `x` es no nulo:

```
>> x=-2:2;
>> any(x), all(x)
ans =
  1
ans =
  0
```

Para matrices, `all` y `any` devuelven los valores aplicados a cada columna de la matriz. Por tanto, lo análogo a vectores sería

`all(all(A))` (`all(A(:))`), `any(any(A))` (`any(A(:))`)

Una utilidad interesante es `find(x)`, que halla los índices del vector `x` que son no nulos:

```
>> x=[ -10 0 -30 0 0 60 70 80 Inf -Inf]
x =
  -10 0 -30 0 0 60 70 80 Inf -Inf
>> find(x)
ans =
  1 3 6 7 8 9 10
>> x( find(x<0))=0 % valores negativos de x sean nulos
x =   0 0 0 0 0 60 70 80 Inf 0
>> % Sería válido también x(x<0) = 0 Masking (mejor así?)
```

```
>> A = rand(3)
A =
   0.7382   0.9355   0.8936
   0.1763   0.9169   0.0579
   0.4057   0.4103   0.3529
>> A(A>0.5) = Inf
A =
      Inf      Inf      Inf
   0.1763      Inf   0.0579
   0.4057   0.4103   0.3529
```

Hay más funciones lógicas como `ischar`, `isnan`, `isreal`, `isinf`, ...

Son interesantes las operaciones de masking cuando se aplican a matrices que representan imágenes digitales[18]

```
>> D = imread('lenna.jpg'); % lenna again
>> imshow(D); % original (grises)
>> D0=D+60; D0(D0>255)=255; % brillante (version 7.x)
>> D0=D+uint8(60); D0(D0>255)=255; % (version 6.x)
>> imshow(D0);
>> D1=D; D1(D1>127)=255; D1(D1<=127)=0; % imagen B/N
>> imshow(D1);
```

[18]Requiere Image_Toolbox

MATLAB admite la construcción de sentencias condicionales `if`, de manera análoga a `FORTRAN 9X`, `C`, `C++`, ...

`if` La sentencia `if` tiene la forma general

```
if condición
  bloque de sentencias if
elseif condición1
  bloque de sentencias elseif condición1
elseif condición2
  bloque de sentencias elseif condición2
else
  bloque de sentencias else
end
```

Por ejemplo, para determinar el signo de un número, podría ser:

```
if x > 0
  si = 1;
elseif x == 0
  si = 0;
else
  si = -1;
end
```

El bloque de sentencias anterior puede escribirse en una línea, pero hay que separar con `,`, `;` cada bloque de sentencias `if`, `elseif`, `else`. Por ejemplo,

```
>> if x>0, si=1; elseif x==0, si=0; else, si=-1; end;
```

```
>> if x>0 si=1, elseif x==0 si=0, else si=-1, end;
```

```
>> if x>0 si=1; elseif x==0 si=0; else si=-1; end;
```

bucle `while` MATLAB también dispone de la sentencia `while` y es:

```
while condición
  bloque de sentencias
end
```

que ejecuta el bloque de sentencias mientras la condición sea cierta.

- `break` termina la ejecución de bucles `for` o `while`. En bucles encajados, termina la ejecución sólo del bucle más interno.
- La sentencia `continue` pasa el control a la iteración siguiente de un bucle `for` o `while`.

Ejemplo.- Cálculo de la unidad de redondeo:

```
>> uround = 1;
>> while 1+uround ~= 1
uround = uround/2;
end;
>> uround
uround =
  1.1102e-16
```

También se puede escribir la sentencia `while` en una línea, separando las órdenes de manera análoga a lo comentado en `if`.

```
>> uround = 1;
>> while 1+uround ~= 1, uround = uround/2; end;
>> uround
uround =
  1.1102e-16
```

`switch` La sentencia `switch` permite ejecutar diferentes bloques de sentencias dependiendo de una expresión. La forma general es

```
switch switch_expr
  case case_expr
    sentencias
  case {case_expr1, case_expr2, case_expr3, ...}
    sentencias
    ...
  otherwise
    sentencias
end
```

Ejemplo.- Supuesto que la variable `metodo` tiene asignado un valor, sería válido el bloque `case` siguiente:

```
switch (metodo)
  case {1,2,3}
    disp('[1,3]')
  case {4,5,6,7}
    disp('[4-7]')
  case {8}
    disp('[8]')
  otherwise
    disp('< 1 o > 8')
end
```

6. m-files: script y function

funciones En FORTRAN 9X hay un programa principal y funciones y subrutinas como subprogramas, mientras que en MATLAB sólo dispone del programa principal **script m-file** y funciones **function m-file**.

Un **script m-file** permite almacenar comandos MATLAB y poder ejecutarlos posteriormente, ... Prácticamente muchos de los ejemplos anteriores se podían haber escrito en este formato.

Una **function m-file** permite escribir funciones estilo MATLAB de manera que acepten argumentos de entrada y de salida, siendo su forma general:

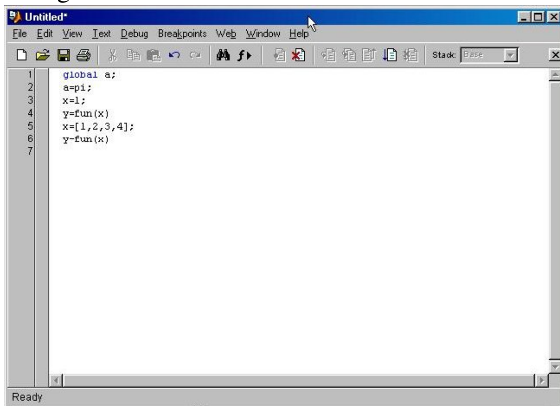
```
function [variables_out] = nombre_funcion(variables_in)
Definición de la función
```

- Todas las variables de la función son locales.
- Para que una variable sea global, i. e. común al programa principal y a una función, hay que hacer uso de la sentencia `global variable` en el programa principal y en dicha función.
- Notar que MATLAB permite recursividad, pero es lenta.
- Puede emplearse `return` para salir de una función.

Para editar m-**files** vale cualquier editor, aunque puede utilizarse el editor de MATLAB con la orden

```
>> edit
```

y aparecerá la siguiente ventana:



en la que podremos empezar a escribir nuestros programas.

Ejemplo Guardar las siguientes órdenes en el fichero esfera.m

```
function [volumen, area] = esfera(r)
% esta funcion calcula el volumen
% y el area de una esfera de radio r
volumen = 4/3*pi*r^3;
area = 4*pi*r^2;
```

y en una sesión MATLAB[19], se trata de ver el comportamiento de diferentes formas de llamar a la función:

```
>> [vol, area]=esfera(4) % dos variables asignadas
  vol = 268.0826
  area = 201.0619
>> vol=esfera(4) % una variable asignada
  vol = 268.0826
>> esfera(4) % ninguna variable asignada
  ans = 268.0826
>> help esfera % escribe los comentarios

esta funcion calcula el volumen
y el area de una esfera de radio r
```

[19]El fichero esfera.m debe estar en el PATH de MATLAB o en el directorio de trabajo. En windows suele ser c:\MATLABR12\WORK.

Además, y para generalizar la función de manera que admita argumentos de entrada vectoriales, cambiamos el exponente `^` por `.^`.

```
function [volumen, area] = esfera(r)
% esta funcion calcula el volumen
% y el area de una esfera de radio r
volumen = 4/3*pi*r.^3;
area = 4*pi*r.^2;
```

```
>> [vol, area]=esfera(1:4)
vol =
   4.1888   33.5103   113.0973   268.0826
area =
   12.5664   50.2655   113.0973   201.0619
>> x=[1;2];
>> [vol, area]=esfera(x) % herencia
vol =
    4.1888
   33.5103
area =
   12.5664
   50.2655
```

Ejemplo.- Escribir una función para calcular la matriz de Hilbert $H \in \mathbb{R}^{n\times n}$ definida por:

$$H_{ij} = \frac{1}{i+j-x},$$

si x está definido o $x = 1$ en caso contrario. Usaremos `nargin`, que es el número de argumentos de entrada de una función para establecer el valor de x.

```
function H = hilbx(n,x)
% Fichero hilbx.m, H=hilbx(n,x)
% esta funcion calcula la matriz de Hilbert
% H(i,j) = 1/(i+j-x), i,j=1,...,n
if nargin < 1
  disp('hilbx(n,x) con n>=1');
  return;
elseif nargin==1
  x=1;
end;
H=zeros(n,n);
for j=1:n;
  H(1:n,j) = 1./((1:n)+j-x)';
end;
```

```
>> a=hilbx(2)
a =
   1.0000   0.5000
   0.5000   0.3333
>> a=hilbx(2,-2)
a =
   0.2500   0.2000
   0.2000   0.1667
>> a=hilbx(2,2)
Warning:  Divide by zero.
> In hilbx.m at line 13
a =
      Inf   1.0000
   1.0000   0.5000
>> help hilbx
  Fichero hilbx.m, H=hilbx(n,x)
  esta funcion calcula la matriz de Hilbert
  H(i,j) = 1/(i+j-x), i,j=1,...,n
```

Ejemplo.- Escribir una función en MATLAB para resolver la ecuación de Kepler:

$$y = x - e\sin(x) \iff f(x) = y - x + e\sin(x) = 0,$$

siendo e la excentricidad, y la anomalía media y x la anomalía excéntrica, por el método de Newton

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \ldots,$$

con x_0 aproximación inicial a la solución.

Usaremos además de `nargin`, `nargout` que es el número de argumentos de salida de la función. La función m-script, `kepler.m`, es:

```
function [x,iter,f,fun] = kepler(y,x0,e,tol,maxiter)
% Resolucion de la ecuacion de Kepler por Newton
% f(x) = y - x + e sin(x)=0
% [x,iter,f,fun] = kepler(y,x0,e,tol,maxiter)
% e excentricidad
% y Anomalía media
% x0 aproximacion inicial
% x Anomalía excéntrica.  solución calculada
% iter iteraciones
% f valor de f en x
% fun valor de f + abs(f)*sin(f) (sólo si se pide)
% Chequear número de argumentos
if nargin < 3
  disp('falta y, x0 y/o la excentricidad e');
  return;
elseif nargin ==3
  tol = 10*eps;
  maxiter = 50;
elseif nargin ==4
  maxiter = 50;
end
```

```
% Chequear valores iniciales de los parámetros
if (e<0) | (e >=1) | (tol<=0) | (maxiter <=1)
  disp('error, e>=1, e<0, tol <=0 o maxiter <=1');
  return;
end;

x = x0; iter = 0; error = Inf;
while error > tol
  x0 = x;
  f = y-x +e*sin(x); fp =-1+e*cos(x);
  x = x - f / fp;
  iter = iter + 1;
  error = abs(x-x0);
  if iter > maxiter,
    disp('maxiter alcanzado'); return;
  end;
end;
% Solo en el caso que se solicite el cuarto argumento
% de salida se calcula
if (nargout==4), fun=f + abs(f)*sin(f); end;
```

```
>> [x,iter,f,fun]=kepler(1,1,.5)
x =
  1.4987
iter =
  6
f =
  -5.5511e-17
fun =
  -0.9640

>> [x,iter,f]=kepler(0.1,2,0.3)
x =
  0.1427
iter =
  6
f =
  -6.9389e-18
```

Mostraremos el comportamiento de una función utilizando variables globales, con argumentos escalares y vectoriales con los dos ficheros siguientes:

```
% fichero main.m (script m-file)
global a;
a=pi;
x=1; y=fun(x)
x=[1,2,3,4]; y=fun(x)
x=[1,2;3,4]; y=fun(x)
```

```
% fichero fun.m (function m-file)
function f = fun(x)
global a;
f=a*x.^2; % .^ vectorización
```

```
>> main
y =
  3.1416
y =
  3.1416 12.5664 28.2743 50.2655
y =
    3.1416   12.5664
   28.2743   50.2655
```

Un tipo de variable similar a la global es la persistente. Éstas son variables locales conocidas solamente en la función en que se declaran, su valor se guarda en memoria entre sucesivas llamadas, y a diferencia del resto, se inicializan como la matriz vacía.

Considerar la siguiente función,

```
function y=funper()
persistent a
if isempty(a)==1, a=0; end
a = a + 1
y = a;
```

```
>> x=zeros(1,10); for i=1:10, x(i)=funper(); end;
>> x
x =
   1  2  3  4  5  6  7  8  9  10
```

En MATLAB es posible definir funciones *simples* en una línea, con la sentencia `inline`

`inline('fun')`	Los argumentos se determinan automáticamente
`inline('fun','var1', ...)`	Los argumentos son var1, ...
`inline('fun',n)`	Los argumentos son x, P1, ..., Pn

```
>> f=inline('sin(x)/(2+cos(y))','x','y');
>> f=inline('sin(x)/(2+cos(y))'); % también así
f =
  Inline function:
  f(x,y) = sin(x)/(2+cos(y))
>> f=argnames(f)
ans =
  'x'
  'y'
>> f=formula(f)
ans =
  sin(x)/(2+cos(y))
>> f(1,2)
ans =
  0.5313
```

Para aplicar la función `f(x,y)` a vectores (matrices) es necesario *vectorizarla* (i. e. insertar un `.` antes de las operaciones `^`, `*` o `/` en la función).

```
>> f=inline('sin(x)./(2+cos(y))'); % vectorizada
>> f=inline('sin(x)/(2+cos(y))');  % NO vectorizada
>> f=vectorize(f)
f =
  Inline function:
  f(x,y) = sin(x)./(2+cos(y))
```

Veamos un ejemplo de la última forma de definición de una función con `inline`

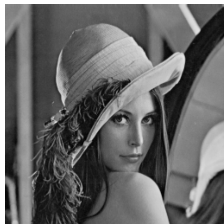
```
>> f=inline('eig(P1*x*inv(P1))',1);
>> U=rand(3); A=hilb(3);
>> eig(A)', f(A,U)'
ans =
  0.0027 0.1223 1.4083
ans =
  1.4083 0.0027 0.1223
```

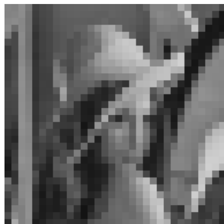
Ejemplo.- Codificación jpeg *muy simplificada* (Requiere Image_Toolbox).

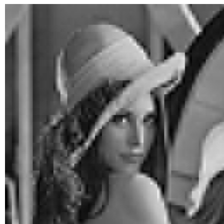
- Dividir la imagen digital en *cuadraditos* 8×8 aplicando en cada uno de ellos la transformada discreta del coseno (TDC) en 2D.
- Enmascarar cada *cuadradito* con $\text{mask}_{ij} = \begin{cases} 1, & i,j \leq k \\ 0, & \text{otro caso} \end{cases}$.
- Finalmente, recomponer la imagen con la inversa de la TDC.

`T=dctmtx(n)`, $T \in \mathbb{R}^{n \times n}$ es la matriz de transformación de la TDC. Si $A \in \mathbb{R}^{n \times n}$, la TDC 2D de A es TAT'. `blkproc(I,[m,n],fun)` procesa la imagen `I` dividiéndola en submatrices `m×n` aplicando en cada una de ellas la función `fun(x)`.

```
function jpeg(k) %     1 ≤ k ≤ 8
im=imread('lenna.jpg'); %   im_ij ∈ [0,255]
im=im2double(im); % coma flotante
T=dctmtx(8); % matriz de la TDC
X=blkproc(im,[8,8],'P1*x*P2',T,T'); % TDC
mask=zeros(8); mask(1:k,1:k)=1;
X=blkproc(X,[8,8],'P1.*x',mask); % máscara
im2=blkproc(X,[8,8],'P1*x*P2',T',T); % ITDC
imshow(im2)
```

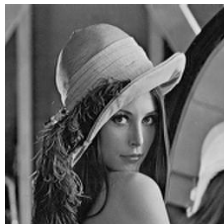

original


$k = 1$


$k = 2$

$k = 3$

$k = 4$


$k = 5$

Algunas funciones especializadas de MATLAB (quad, fsolve, ...) necesitan pasar como argumento una función de una variable,

```
>> quad('cos',0,pi) % ∫_0^π cos(x) dx = 0
ans =
   -1.1102e-16
>> quad('exp(sin(x-1)-x.*x)./(1+x.*x)',0,1) % vectorizada !
ans =
   0.3594
>> fzero('exp(cos(x))-19*x*x',0)
ans =
   -0.3659
```

Es posible asignar en MATLAB un *identificador* único a una función[20]. Es útil para pasar una función (identificador) complicada como argumento. La sintaxis como *función anónima* es:

```
identificador_funcion = @(argumentos) expresión
```

y si está definida en el fichero `fun.m` el identificador es simplemente `@fun`.

[20]function handle

Si la función a la que se llama espera una función argumento de k–variables, y ésta depende de más variables (*función parametrizada*), hay que pasar el identificador y su lista de argumentos previamente definidos. Por ejemplo,

```
f = @(x,a,b) cos(a+b*x.^2)+sin(x*a)./(a^2+x.^2) - a*b;
```

```
function f=fun(x,a,b) % fichero fun.m
f=cos(a+b*x.^2) + sin(x*a)./(a^2+x.^2) - a*b;
```

`quad` y `fzero` esperan funciones de una variable, mientras que `f` y `fun` dependen de tres.

```
>> a=1; b=-1; % definir los parámetros
>> fzero(@(x) f(x,a,b),3)
ans =
    3.2115
>> quad(@(x) fun(x,a,b),0,2)
ans =
    3.5539
```

Esta forma es la más general y es válida siempre.

Mayor simplificación se obtiene cuando la función que se pasa como argumento no está parametrizada.

```
f = @(x) cos(x-sin(x)) % función anónima
```

```
function f=fun(x) % fichero fun.m
f=cos(x-sin(x));
```

y la llamada a `quad`, `fzero` con estas funciones como argumentos sería

```
>> quad(f,0,2)
ans =
    1.8235
>> quad(@fun,0,2)
ans =
    1.8235
```

Ver un ejemplo de integración numérica de P.V.I (Ecuación de Lorentz)

Recursividad Un ejemplo interesante de recursividad es el método de Strassen (1969) para el producto de dos matrices A, $B \in \mathbb{R}^n$, $n = 2^J$. Dicho producto viene dado por:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

donde

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}, \end{aligned}$$

y tiene una complejidad computacional $\mathcal{O}(n^3)$.

El algoritmo de Strassen, está definido por:

$$\begin{array}{ll} P_1 = (A_{11} + A_{22})(B_{11} + B_{22}), & P_2 = (A_{21} + A_{22})B_{11}, \\ P_3 = A_{11}(B_{12} - B_{22}), & P_4 = A_{22}(B_{21} - B_{11}), \\ P_5 = (A_{11} + A_{12})B_{22}, & P_6 = (A_{21} - A_{11})(B_{11} + B_{12}), \\ P_7 = (A_{12} - A_{22})(B_{21} + B_{22}), & \end{array}$$

$$\begin{array}{ll} C_{11} = P_1 + P_4 - P_5 + P_7, & C_{12} = P_3 + P_5, \\ C_{21} = P_2 + P_4, & C_{22} = P_1 - P_2 + P_3 + P_6, \end{array}$$

y tiene una complejidad computacional del orden $\mathcal{O}(n^{2.81})$, así que teóricamente y para n suficientemente grande es más rápido que el *clásico*. Además admite una recursividad directa en los productos de las matrices A_{ij} y B_{ij}.

Una implementación del algoritmo de Strassen viene dada en *The Matrix Computation Toolbox*, y la recursividad se realiza hasta que la dimensión de la matriz sea menor o igual que una cota inferior (`nmin`). Por tanto, cuanto menor sea `nmin`, mayor es el nivel de recursividad empleada.

```
function C = strassen(A, B, nmin)
n = length(A);
if n <= nmin
  C = A*B;
else
  m = n/2; i = 1:m; j = m+1:n;
  P1 = strassen( A(i,i)+A(j,j), B(i,i)+B(j,j), nmin);
  P2 = strassen( A(j,i)+A(j,j), B(i,i), nmin);
  P3 = strassen( A(i,i), B(i,j)-B(j,j), nmin);
  P4 = strassen( A(j,j), B(j,i)-B(i,i), nmin);
  P5 = strassen( A(i,i)+A(i,j), B(j,j), nmin);
  P6 = strassen( A(j,i)-A(i,i), B(i,i)+B(i,j), nmin);
  P7 = strassen( A(i,j)-A(j,j), B(j,i)+B(j,j), nmin);
  C = [ P1+P4-P5+P7 P3+P5; P2+P4 P1+P3-P2+P6 ];
end
```

Tiempos de ejecución del algoritmo `strassen` en función de `nmin` para matrices 256×256.

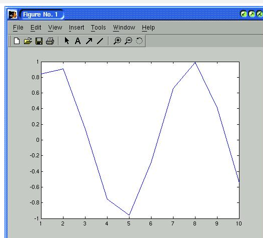
`nmin`	4	8	16	32	64	128	**256**
CPU (sg)	19.70	3.75	1.04	0.487	0.275	0.154	**0.093**

7. Gráficos

La parte gráfica de MATLAB tiene una calidad extraordinaria, puede manejar muchos tipos de figuras (2D, 3D, imágenes) y es relativamente fácil de manejar. En esta sección veremos las órdenes básicas de *alto nivel* para dibujar.

2D El dibujo más sencillo se realiza con `plot(x,y)`, siendo `x` e `y` vectores [21], que dibuja la poligonal que une los puntos (x_i, y_i), $i = 1, \ldots, n$.

```
>> x=1:10; y=sin(x); plot(x,y)
```



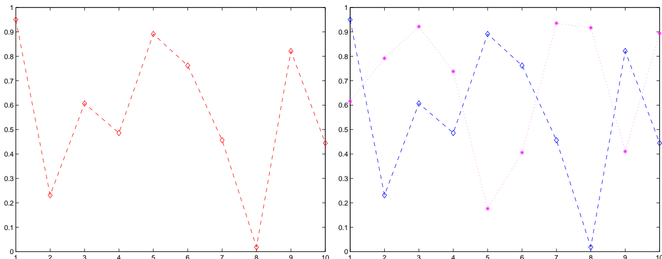
[21] Si `x` o `y` son matrices, o tienen componentes complejas, … ver `help plot`

Esta orden, admite la expresión:

`plot(x,y,'`**c m e**`')`

donde las letras de la expresión caracter '**c** | **m** | **e**' son relativas a **color**, **marca** y **estilo** del dibujo respectivamente, pudiendo aparecer en cualquier orden. Por ejemplo,

```
>> x=1:10; y=rand(size(x)); z=rand(size(x));
>> plot(x,y,'rd--')
>> plot(x,y,'bd--',x,z,'m*:')   % Dos gráficas
```



Pueden añadirse las opciones `'LineWidth',anch`, siendo `anch` la anchura de la línea (defecto 0.5 puntos) y `'MarkerSize',size`, donde `size` es el tamaño del símbolo (defecto 6 puntos), `'Color',[r,g,b]`, siendo `r`, `g`, `b`, valores $\in [0,1]$ relativos a la codificación RGB, ...

```
plot(x,y,'b*-','LineWidth',4,'MarkerSize',9,'Color',[.8,.8,0])
```

Es posible superponer varios gráficos con distintas órdenes `plot`. Para ello utilizar `hold on/off`:

```
>> clf % borra el contenido de la ventana gráfica
>> hold on
>> plot (...)  % varios dibujos en la misma ventana
>> plot (...)
>> hold off
>> grid on % Superpone una rejilla
>> grid off % Borra la rejilla
```

Podemos controlar los ejes, añadir texto, leyendas, ...

- `axis`: Control de ejes:

`axis([xmin xmax ymin ymax])`	Especificar los límites
`axis auto`	MATLAB lo hace automático
`axis off`	Se quitan los ejes
`axis equal`	escala el dibujo
`axis xy`	Ejes cartesianos estándar (defecto)
`axis ij`	Ejes para matrices $\begin{matrix} 11 & \dots & 1n \\ \vdots & \ddots & \vdots \\ n1 & \dots & nn \end{matrix}$

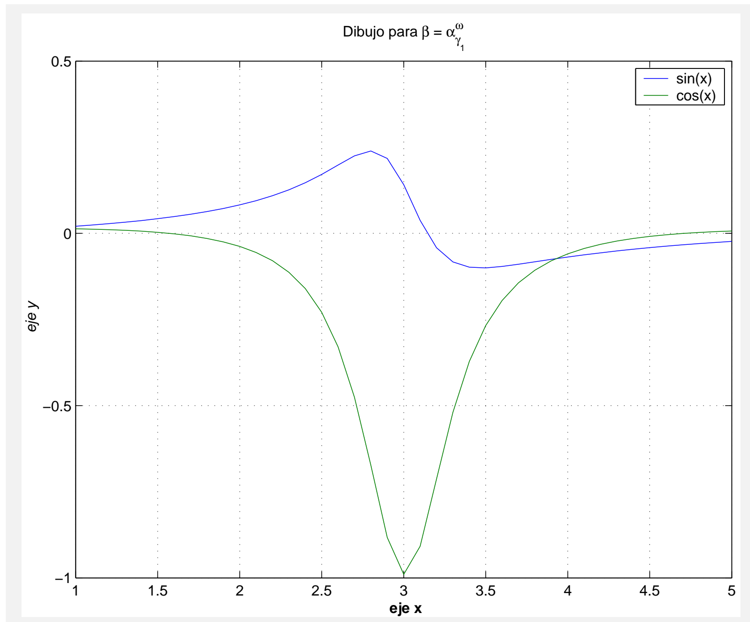
- `text(x,y,'texto')`: Añade `texto` al gráfico en el punto de coordenadas `(x,y)`. Se pueden incluir comandos de TEX.
- `title('titulo')`: Añade `titulo` en la parte superior del dibujo.
- `xlabel('etiqueta_x')`: Añade `etiqueta_x` en el eje de abcisas.
- `ylabel('etiqueta_y')`: Análogo a `xlabel`, pero para el eje de ordenadas.
- `legend('text1','text2',...,pos)`: Añade al gráfico las leyendas `text1`, `text2`, ... como etiquetas. La colocación depende del valor de `pos`:

0	Automático (Defecto).
1	Esquina superior derecha
2	Esquina superior izquierda
3	Esquina inferior derecha
4	Esquina inferior izquierda
-1	Derecha del gráfico

No obstante, el cuadro de la leyenda puede moverse pinchando con el ratón, y con doble click, puede modificarse su contenido.

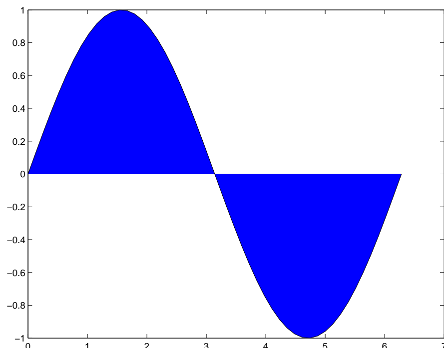
Ejemplo:

```
>> x=1:0.1:5;
>> y = 1./(1 + 10*(x-3).^2).*sin(x);
>> z = 1./(1 + 10*(x-3).^2).*cos(x);
>> clf;
>> plot(x,y,x,z)
>> title('Dibujo para \beta = \alpha_{\gamma_1}^\omega')
>> xlabel('\bf eje x'); ylabel('\it eje y')
>> legend('sin(x)','cos(x)');
>> grid on
```


Dibujo para β = $\alpha_{\gamma_1}^{\omega}$
sin(x)
cos(x)
eje y
eje x
0.5
0
−0.5
−1
1
1.5
2
2.5
3
3.5
4
4.5
5

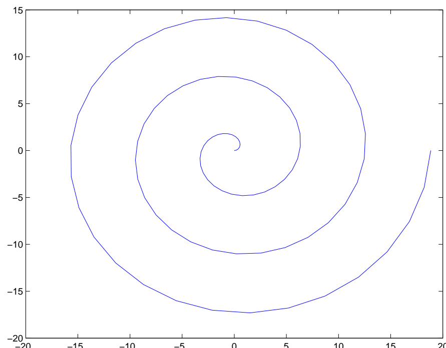
Con la orden `fill(x,y,color)` se rellena con el color especificado por `color` el polígono definido por los vectores `x` e `y`.

```
>> x=linspace(0,2*pi,50);
>> y=sin(x);
>> fill(x,y,'b')
```



Las curvas en paramétricas, $(x(t), y(t))$, $t \in [a, b]$ se dibujan de manera análoga a las estándar:

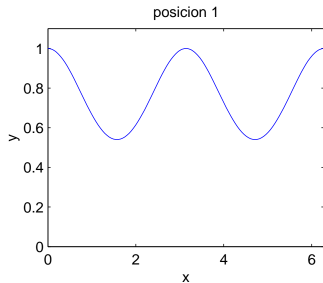
```
>> t=linspace(0,6*pi,90);
>> x= t.*cos(t); y=t.*sin(t);
>> plot(x,y,'b')
```

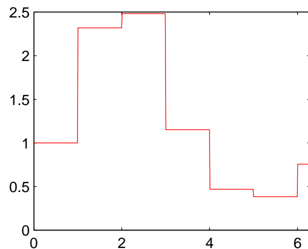


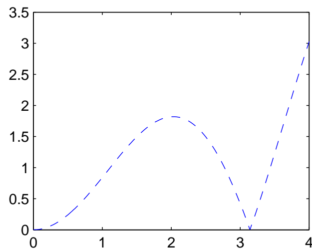
La ventana gráfica de MATLAB puede albergar varios gráficos independientes. Esto se hace con la orden `subplot(mnk)` o `subplot(m,n,k)`. Esta divide la ventana gráfica en una matriz `m×n` de subventanas, realizando el dibujo en la ventana `k`, contando por filas.

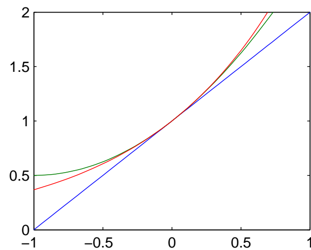
Para dibujar expresiones matemáticas, puede resultar preferible emplear la orden `fplot('funcion', dominio)`, que genera de manera automática el dibujo.

```
>> subplot(221); fplot('cos(sin(x))',[0 2*pi 0 1.1])
>> xlabel('x'); ylabel('y'); title('posicion 1')
>> subplot(222);
>> fplot('exp(sin(floor(x)))',[0 2*pi],'r')
>> subplot(223);
>> fplot('abs(sin(x)*x)',[0 4],'b--')
>> subplot(224);
>> fplot('[1+x,1+x+x^2/2,exp(x)]',[-1 1 0 2])
```


posicion 1
y
x
0
0.2
0.4
0.6
0.8
1
0
2
4
6


0
0.5
1
1.5
2
2.5
0
2
4
6


0
0.5
1
1.5
2
2.5
3
3.5
0
1
2
3
4


0
0.5
1
1.5
2
–1
–0.5
0
0.5
1

Los gráficos generados por MATLAB pueden exportarse a ficheros en varios formatos, entre los que destacamos:

driver	**Descripción**
-dps	PostScript for black and white printers
-dpsc	PostScript for color printers
-deps[2]	Encapsulated PostScript [level 2]
-depsc[2]	Encapsulated Color PostScript [level 2]
-djpeg<nn>	JPEG image, Nivel de calidad nn (sólo figuras) Si se omite, el defecto es 75.
-dpng	Portable Network Graphic 24-bit truecolor

con la orden `print driver fichero`.

Por ejemplo, para generar el fichero `'figura.eps'` cuyo contenido es el dibujo de la ventana gráfica actual, será:

```
>> print -depsc figura.eps
>> print('-depsc', 'figura.eps') % forma de función
```

Esta última forma permite introducir nombres de ficheros en función de una o más variables. Por ejemplo,

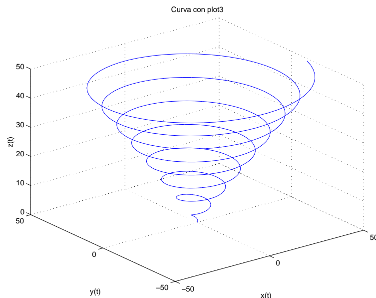
```
>> name=['fichero',num2str(i),'.eps'];
name =
fichero9.eps
>> print('-depsc', name)
>> ls -l *.eps
ans =
-rw-r--r-- 1 user user 8396 jun 24 19:02 fichero9.eps
```

Hay más órdenes gráficas, entre las que destacamos:

`ezplot`	Coordenadas cartesianas (versión fácil)
`loglog`	Doble escala logarítmica
`semilogx`	Escala logarítmica en x
`semilogy`	Escala logarítmica en y
`polar`	Coordenadas polares
`ezpolar`	Coordenadas polares (versión fácil)
`area`	Rellena áreas de gráficos
`bar`	Gráfico de barras
`pie`	Gráficos de tarta

3D En el caso de gráficas tridimensionales, el ejemplo más simple es el dibujo de una curva alabeada $(x(t), y(t), z(t))$, $t \in [t_0, t_f]$, con la orden `plot3(x,y,z)`:

```
>> t=linspace(0,50,500);
>> x=t.*cos(t); y=t.*sin(t); z=t;
>> plot3(x,y,z)
>> xlabel('x(t)'); ylabel('y(t)'); zlabel('z(t)');
>> grid on; title('Curva con plot3');
```



Para dibujar una superficie $z = f(x, y)$ en un dominio $\mathcal{D} = [x_0, x_f] \times [y_0, y_f]$, primero hay que definir una malla con `meshgrid` y luego evaluar $z = f(x, y)$ en dicha malla.

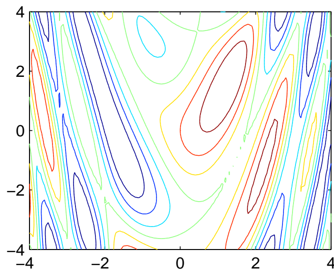
```
>> [X,Y]=meshgrid( x0:xpaso:xf, y0:ypaso:yf);
>> Z=f(X,Y);
```

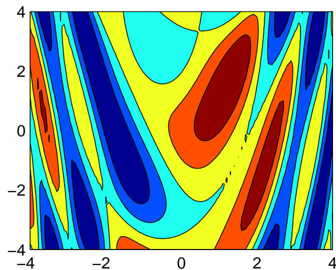
Las órdenes de dibujo 3D más usuales son:

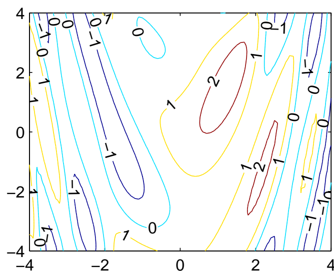
`contour(X,Y,Z,num)`	Dibuja “num” curvas de nivel
`contourf(X,Y,Z,num)`	Dibuja y rellena “num” curvas de nivel
`ezcontour('f',dominio)`	version fácil de contour
`mesh(Z)`	Dibuja la función `Z` (ejes matriciales)
`mesh(X,Y,Z)`	Dibuja la función `Z` (ejes cartesianos)
`meshc(Z)`	mesh + contour (ejes matriciales)
`meshc(X,Y,Z)`	mesh + contour (ejes cartesianos)
`surf(Z)`	Dibujo sólido (ejes matriciales)
`surf(X,Y,Z)`	Dibujo sólido (ejes cartesianos)
`surfc(...)`	contour + surf

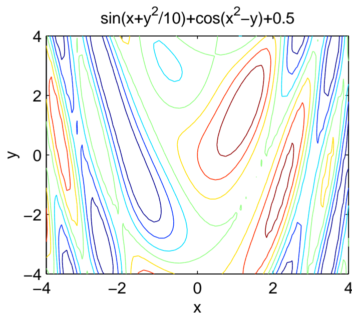
Ejemplo.- Veamos un ejemplo de dibujo de curvas de nivel con la orden `contour`:

```
>> [X,Y]=meshgrid( -4:0.1:4, -4:0.1:4);
>> Z= sin(X + Y.^2/10) + cos(X.^2-Y) + 0.5;
>> % Dibujo por defecto.  (Ejes cartesianos)
>> subplot(221)
>> contour(X,Y,Z)
>> % Dibujar 5 curvas de nivel con relleno.
>> subplot(222)
>> contourf(X,Y,Z,5)
>> % Dibujar valores determinados de las curvas de
>> % nivel y etiquetarlos
>> subplot(223)
>> val = [-1 0 1.  2.];
>> [C,h] = contour(X,Y,Z,val);
>> clabel(C,h,val);
>> % Forma sencilla de dibujar curvas de nivel.
>> subplot(224)
>> ezcontour('sin(x+y^2/10)+cos(x^2-y)+.5',[-4,4,-4,4]);
```



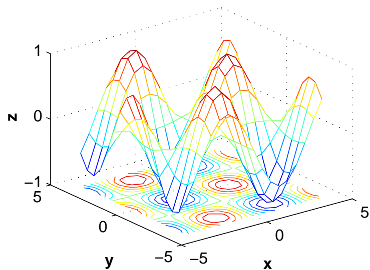


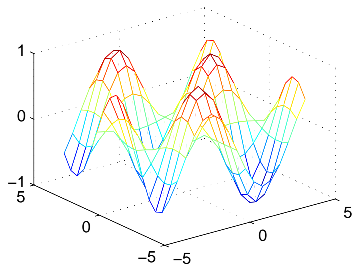


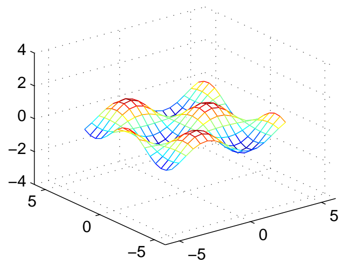

sin(x+y^2/10)+cos(x^2−y)+0.5
y
x

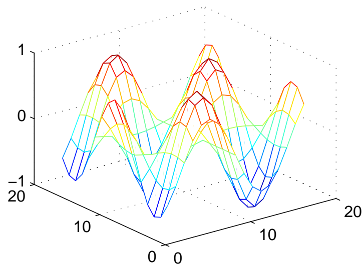
Ejemplo.- La orden `mesh`, dibuja la superficie como si fuera una malla:

```
>> [X,Y]=meshgrid( -4:0.5:4, -4:0.5:4);
>> Z= sin(X).*cos(Y);
>> subplot(221)
>> meshc(X,Y,Z)
>> xlabel('\bf x'); ylabel('\bf y'); zlabel('\bf z');
>> subplot(222)
>> mesh(X,Y,Z)
>> subplot(223)
>> mesh(X,Y,Z) % ejes cartesianos
>> axis([-6 6 -6 6 -4 4]);
>> subplot(224)
>> mesh(Z) % ejes matriciales
```


1
0
−1
z
5
0
−5
y
−5
0
5
x


1
0
−1
5
0
−5
−5
0
5


4
2
0
−2
−4
5
0
−5
−5
0
5

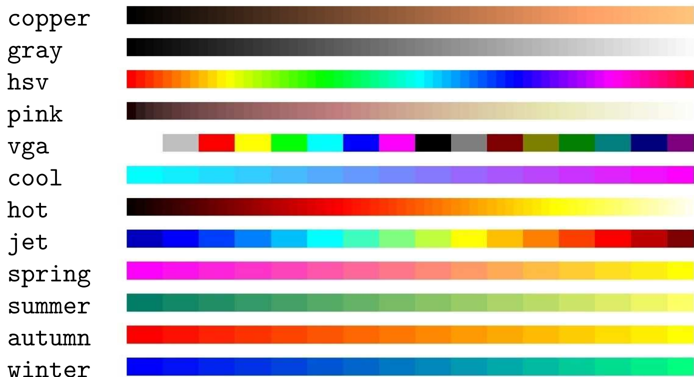

1
0
−1
20
10
0
0
10
20

Ejemplo.- La orden `surf` dibuja las superficies con colores sólidos. Se han introducido en la generación de las gráficas algunas especificaciones como:

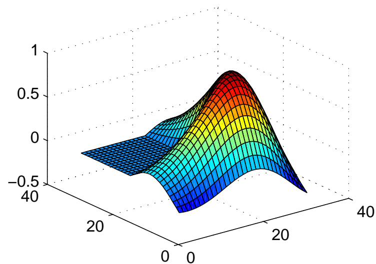
`colormap(C)`	Tabla de color usada para los gráficos. `C` debe ser una matriz `m`×3, donde cada fila codifica un color RGB. El defecto es `'jet'` (`'copper'`,`'gray'`,... (ver ejemplos))
`colorbar`	Dibuja una barra de colores con los valores del actual `colormap` Por defecto es la altura

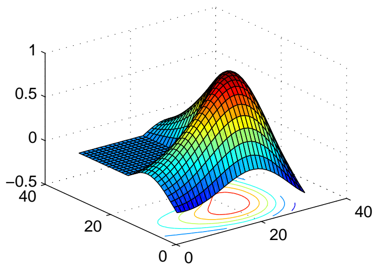
`shading flat`
`shading interp`
 	
`shading faceted`
	Quita la red del dibujo
	Varía el color en cada segmento
	por interpolación
	Defecto

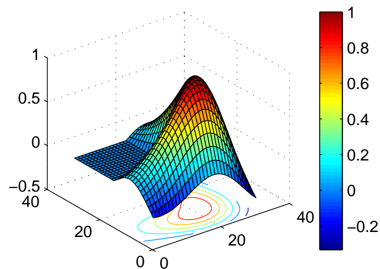
Diversos `colormap` de MATLAB

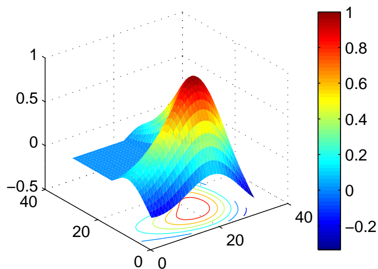


```
>> Z=membrane;
>> subplot(221), surf(Z)
>> subplot(222), surfc(Z)
>> subplot(223), surfc(Z), colorbar
>> subplot(224), surfc(Z), shading flat, colorbar
```


1
0.5
0
−0.5
40
20
0
0
20
40


1
0.5
0
−0.5
40
20
0
0
20
40

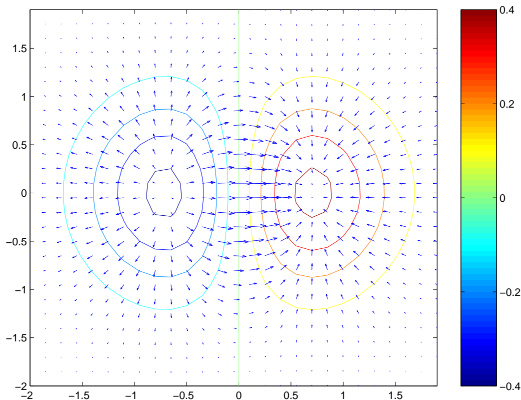

1
0.5
0
−0.5
40
20
0
0
20
40
1
0.8
0.6
0.4
0.2
0
−0.2


1
0.5
0
−0.5
40
20
0
0
20
40
1
0.8
0.6
0.4
0.2
0
−0.2

Para el dibujo de campos vectoriales gradientes, conviene utilizar la orden `[px,py]=gradient(f,hx,hy)`, que calcula una aproximación por diferencias al gradiente de `f`, utilizando `hx`, `hy` como los desplazamientos en las direcciones de abcisas y ordenadas respectivamente, según las fórmulas

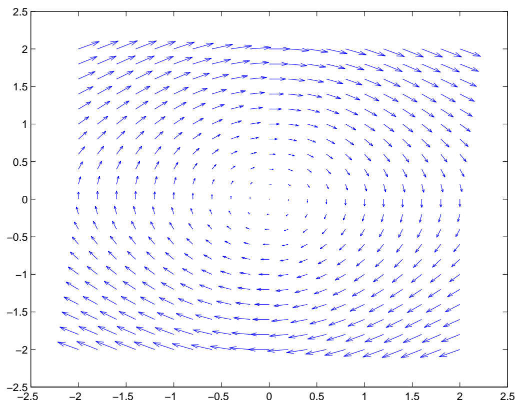
$$\nabla f(x,y) \simeq \left(\frac{f(x+hx,y)-f(x,y)}{hx}, \frac{f(x,y+hy)-f(x,y)}{hy} \right).$$

```
>> [X,Y] = meshgrid(-2:.15:2,-2:.15:2);
>> Z = X.*exp(-X.^2-Y.^2);
>> [px,py] = gradient(Z,.15,.15);
>> contour(X,Y,Z), hold on, quiver(X,Y,px,py), colorbar
>> hold off
```



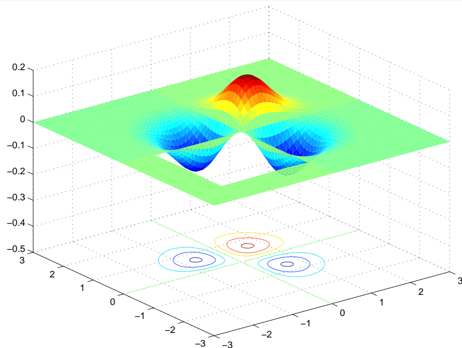
En el caso de campos vectoriales más generales $(x, y) \to (f_1, f_2)$, simplemente hay que evaluar cada componente, por ejemplo, para el campo vectorial $F : \mathbb{R}^2 \to \mathbb{R}^2$, $(x, y) \mapsto (f_1, f_2) = (y, -\sin(x))$, sería:

```
>> [X,Y] = meshgrid(-2:.2:2,-2:.15:2);
>> f1 = Y; f2 = -sin(X);
>> quiver(X,Y,f1,f2)
```

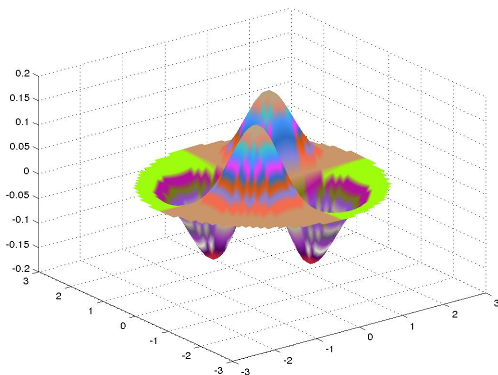


Una propiedad interesante de NaN (Not a Number), es que en las representaciones gráficas no se dibuja, por lo que es factible su uso para dibujar gráficos con agujeros:

```
>> [X,Y] = meshgrid(-3:.1:3,-3:.1:3);
>> Z = sin(X).*sin(Y).* exp(-X.^2-Y.^2);
>> Z( 10:30, 10:30) = NaN;
>> surfc( X, Y, Z), shading flat
```

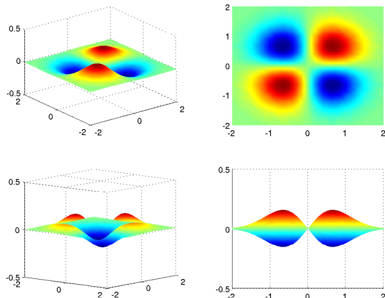


```
>> [X,Y] = meshgrid(-3:.1:3,-3:.1:3);
>> Z = sin(X).*sin(Y).* exp(-X.^2-Y.^2);
>> mask = X.^2 + Y.^2 > 6;
>> Z(mask)=NaN;
>> surf(X,Y,Z),shading interp,colormap(rand(20,3))
```



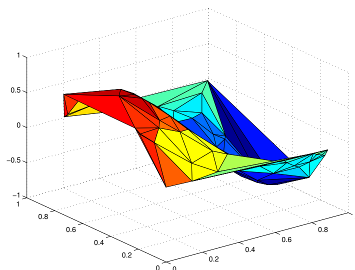
El ángulo de observación de la gráfica se cambia con `view(`θ_1`,`θ_2`)`, donde θ_1 es el azimut o rotación horizontal (defecto -37.5°) respecto del eje OY^- y θ_2 es la latitud o elevación vertical (defecto 30°). `view(2)` y `view(3)` son las vistas por defecto de dibujos 2D y 3D respectivamente.

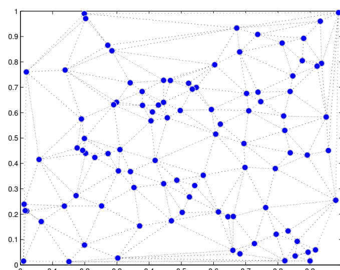
```
>> [X,Y] = meshgrid(-2:.1:2,-2:.1:2);
>> Z = sin(X).*sin(Y).*exp(-X.^2-Y.^2);
>> subplot(221), surf(X,Y,Z),shading interp
>> subplot(222), surf(X,Y,Z),shading interp,view(0,90)
>> subplot(223), surf(X,Y,Z),shading interp,view(50,-10)
>> subplot(224), surf(X,Y,Z),shading interp,view(90,0)
```



Para reconstruir (o dibujar) una función $z = f(x, y)$ a partir de valores en un conjunto de puntos $\{(x_i, y_i, z_i)\}_{i=1}^{N}$ en el espacio, una posible forma es el método de triangulación de Delaunay.

```
>> n=100; x=rand(n,1); y=rand(n,1);
>> z=cos(pi*x).*sin(pi*y);
>> tri=delaunay(x,y);
>> trisurf(tri,x,y,z) % dibujo de la gráfica
>> triplot(tri,x,y) % dibujo de la triangulación
>> % Equivalentes a surf y plot para triangulaciones
```

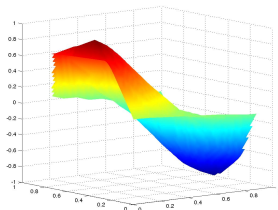


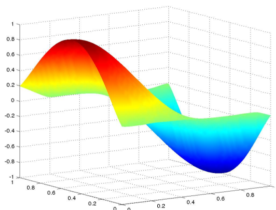


Dibujo de la gráfica y de la triangulación. Pueden superponerse con `hold on`

Otra solución es usar `griddata`[22] para interpolar los valores de una función en una malla rectangular dada por `meshgrid`

```
>> [XI,YI]=meshgrid(0:1/50:1,0:1/50:1);
>> ZI=griddata(x,y,z,XI,YI,'linear');
>> % Usa Delaunay y pueden aparecer NaN en ZI
>> % ZI(isnan(ZI))=0; % Eliminar los NaN de ZI ?
>> surf(XI,YI,ZI); shading interp
>> ZI=griddata(x,y,z,XI,YI,'v4'); % No usa Delaunay
>> surf(XI,YI,ZI); shading interp
```





'linear' *vs* 'v4'

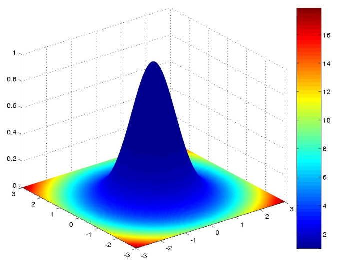
[22] ver también `tri2grid`

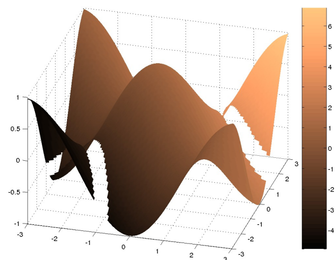
La sentencia `surf` puede utilizarse con cuatro parámetros `(X,Y,Z,C)`, donde `C` representa la escala de color. MATLAB escala `C` para obtener los colores del actual `colormap`. Si sólo hay tres parámetros, entonces `C=Z`, es decir el color es proporcional a la altura. Esto puede aprovecharse para representar una función $g(x, y, z)$ sobre la superficie definida por (x, y, z).

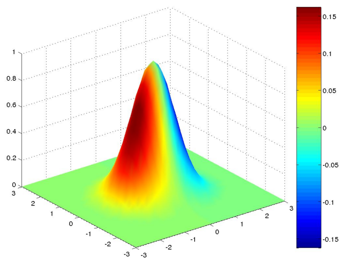
```
>> [X,Y]=meshgrid(-3:0.05:3,-3:0.05:3);
>> Z=exp(-X.^2-Y.^2);
>> surf(X,Y,Z, X.^2+Y.^2+Z.^2), shading interp
>> colorbar
```

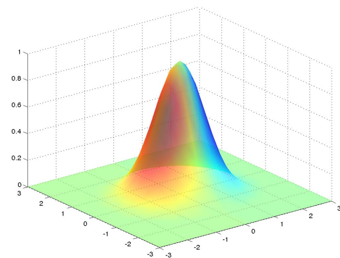
```
>> [X,Y]=meshgrid(-3:0.05:3,-3:0.05:3);
>> Z=cos(X).*cos(Y); Z(X.*Y > 2 & X.*Y<3)=NaN;
>> surf(X,Y,Z, X+Y+Z), shading interp
>> colormap('copper'); colorbar
```

```
>> [X,Y]=meshgrid(-3:0.2:3); Z=exp(-X.^2-Y.^2);
>> surf(X,Y,Z, gradient(Z)), shading interp, colorbar
>> % Transparencia alpha(t), t ∈ [0,1]
>> % alpha(0) transparente total, alpha(1) opaco
>> surf(X,Y,Z,gradient(Z)), shading interp, alpha(0.6)
```


1
0.8
0.6
0.4
0.2
0
3 2 1 0 -1 -2 -3
-3 -2 -1 0 1 2 3
16
14
12
10
8
6
4
2


1
0.5
0
-0.5
-1
-3 -2 -1 0 1 2 3
3 2 1 0 -1 -2 -3
6
5
4
3
2
1
0
-1
-2
-3
-4


1
0.8
0.6
0.4
0.2
0
3 2 1 0 -1 -2 -3
-3 -2 -1 0 1 2 3
0.15
0.1
0.05
0
-0.05
-0.1
-0.15


1
0.8
0.6
0.4
0.2
0
3 2 1 0 -1 -2 -3
-3 -2 -1 0 1 2 3

Al resolver la ecuación de onda de Schrödinger se obtienen una serie de funciones de onda (o probabilidades de distribución de los electrones) para los distintos niveles energéticos que se llaman *orbitales atómicos*.

En general, el orbital de números cuánticos (l, m) está dado por el armónico esférico

$$Y_{l,m}(\theta, \phi) = S_{l,|m|}(\theta) T_m(\phi),$$

$$T_m(\phi) = \tfrac{1}{\sqrt{2\pi}} \mathrm{e}^{im\phi},$$

$$S_{l,|m|}(\theta) = \left(\tfrac{(2l+1)(l-|m|)!}{2(l+|m|)!}\right)^{1/2} P_l^{|m|}(\cos\theta),$$

donde $\theta \in [0, \pi)$, $\phi \in [0, 2\pi)$ son las coordenadadas polar y azimutal respectivamente, y

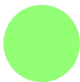
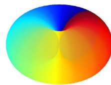
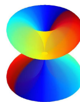
$$P_l^m(x) = (-1)^m (1 - x^2)^{m/2} \frac{\mathrm{d}^m}{\mathrm{d}x^m} P_l(x)$$

siendo $P_l(x)$ el polinomio de Legendre de grado l.

En el fichero `orb.m` está implementado el cálculo de los orbitales de números cuánticos (l, m), $m = 0, \pm 1, \ldots, \pm l$.

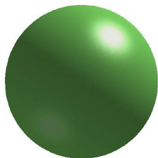
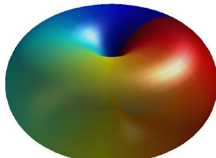
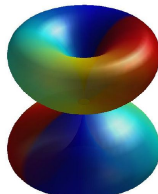
```
function [XO,YO,ZO,A0] = orb(X,Y,l,mm,ind)
% representacion del valor absoluto de los armonicos esfericos
% |Y(l,m)|.  El color representa la fase de la funcion compleja
% en cada punto.  Los armonicos Y(l,-m) dan la misma figura que
% Y(l,m) pero el mapa de colores rota en sentido inverso
% si ind=-1 dibuja la parte real, ind=1 la imaginaria e ind=0
% calcula el valor absoluto.
[nf, nc]=size(Y); m = abs(mm);
cosy = cos(Y); siny = sin(Y);
v = legendre(l,cosy(:)); v = v(m+1,:);
if l==0, v = ones(nf,nc);
else, v = reshape(var,nf,nc);
end
cc=sqrt(factorial(l-m)/factorial(l+m)*(2*l+1)/(4*pi))*v;
if ind == -1, r= cos(mm*X);
elseif ind == 0, r= complex(cos(mm*X),sin(mm*X));
else, r= sin(mm*X);
end
r = r .* cc; A0 = angle(r); r = abs(r);
% paso de esféricas a cartesianas.
XO = r .* cos(X).*siny;
YO = r .* sin(X).*siny;
ZO = r .* cosy;
```

```
>> [X,Y]=meshgrid(linspace(0,2*pi,50), linspace(0,pi,50));
>> [XO,YO,ZO,A0] = orb(X,Y,2,-1,0); surf(XO,YO,ZO,A0)
>> shading interp, axis equal, axis off, alpha(0.85)
```

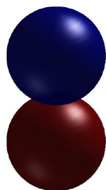




$\mathrm{orb}(\mathrm{X}, \mathrm{Y}, 0, 0, -1)$ $\mathrm{orb}(\mathrm{X}, \mathrm{Y}, 1, 1, 0)$ $\mathrm{orb}(\mathrm{X}, \mathrm{Y}, 2, 1, 0)$

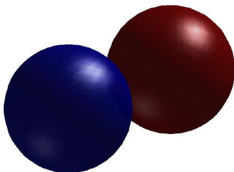
Si se emplean las sentencias `light`, `lighting phong` que controlan la iluminación, los dibujos adoptan un aspecto curioso.

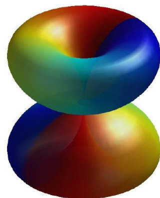
$\mathrm{orb}(\mathrm{X}, \mathrm{Y}, 0, 0, -1)$ $\mathrm{orb}(\mathrm{X}, \mathrm{Y}, 1, 1, 0)$ $\mathrm{orb}(\mathrm{X}, \mathrm{Y}, 2, 1, 0)$



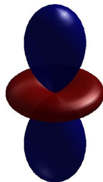
$\text{orb}(X, Y, 1, 0, -1)$



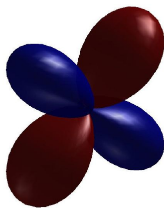
$\text{orb}(X, Y, 1, 1, -1)$



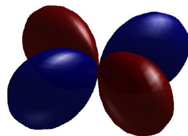
$\text{orb}(X, Y, 2, -1, 0)$



$\text{orb}(X, Y, 2, 0, -1)$



$\text{orb}(X, Y, 2, 1, -1)$

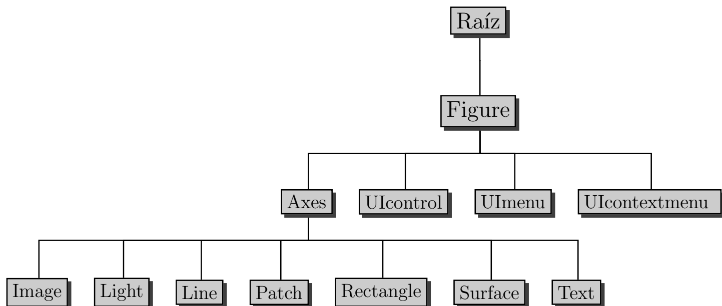


$\text{orb}(X, Y, 2, 2, -1)$

8. Gráficos Avanzados

En esta sección se trata de acceder a la parte gráfica de MATLAB pero a *bajo nivel*. Todas las funciones que hemos visto forman parte de un sistema de objetos gráficos, identificadores gráficos (*handle graphics*), en los que se almacenan las propiedades de cada objeto gráfico. Éstos dan control total de todas las propiedades de los gráficos.

Los gráficos en MATLAB están formados por objetos de distinta naturaleza y con la siguiente estructura jerárquica:

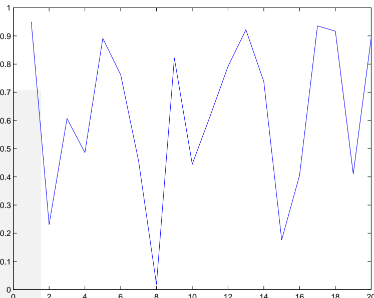


Estos objetos son los elementos básicos para la construcción de interfaces de usuario (*user interface*) (UI) y gráficos en MATLAB.

`Raíz`	corresponde a la pantalla gráfica (única)
`Figure`	ventana donde se dibujan los ejes (`Axes`) y objetos UI (`uicontrols, uimenus y uicontextmenus`)
`Axes`	ejes cartesianos para dibujar gráficos en una figura
`Uicontrol`	control de UI para interaccionar con el usuario. botones, sliders, ...
`Uimenu`	menú de UI en la barra superior de la figura.
`Uicontextmenu`	menú contextual que se despliega al realizar alguna acción (p. e. pinchar con el ratón)
`Image`	imagen 2D basada en pixeles
`Light`	añade fuentes de luz (afectan a polígonos y superficies)
`Line`	dibuja líneas (`plot, plot3`, ...)
`Patch`	polígonos 2D (`fill`, ...)
`Rectangle`	rectángulos, rectángulos redondeados y elipses
`Surface`	Genera una superficie en 3D de una matriz (`surf`, ...)
`Text`	Texto

Para la generación y manipulación de objetos gráficos hay que emplear las siguientes funciones:

`get(id)`	examina los valores de las propiedades del objeto con identificador `id`
`get(id,'P1')`	examina el valor de la propiedad `P1` del objeto con identificador `id`
`set(id)`	muestra los posibles valores del objeto con identificador `id`
`set(id,'P1')`	muestra los posibles valores de la propiedad `P1`. del objeto con identificador `id`
`set(id,'P1',V1)`	modifica el valor de `P1` con `V1` del objeto con identificador `id`
`findobj`	muestra los valores de los identificadores del objeto raíz e hijos
`findobj('P1',V1)`	busca objetos con propiedades concretas del objeto raíz y de sus hijos
`findobj(id,'P1',V1)`	busca objetos con propiedades concretas en el objeto con identificador `id`



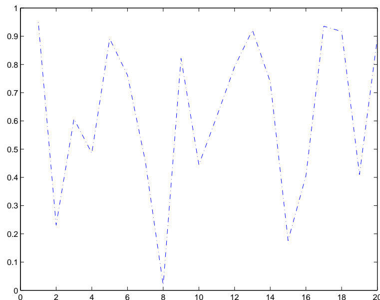

```
>> plot(rand(20,1))
>> id = findobj
id =
          0
     1.0000
   151.0018
   152.0023
>> get(id,'type')
ans =
   'root'
   'figure'
   'axes'
   'line'
```

get(id(4))		set(id(4))	
Color:	[0 0 1]	Color:	{}
EraseMode:	'normal'	EraseMode:	{4x1 cell}
LineStyle:	'-'	LineStyle:	{5x1 cell}
LineWidth:	0.5000	LineWidth:	{}
Marker:	'none'	Marker:	{14x1 cell}
MarkerSize:	6	MarkerSize:	{}
⋮		⋮	
Parent:	151.0018	Parent:	{}
⋮		⋮	

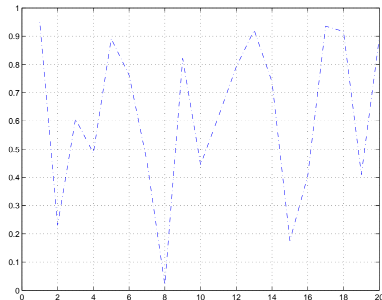
Notar que el padre de `id(4)` es `151.0018`(`axes`), y la poligonal tiene color azul ([0 0 1]), ... Por ejemplo, para ver los posibles valores de la propiedad `LineStyle`, escribimos

```
>> set(id(4),'linestyle')  % posibles valores de linestyle
[ {-} | -- | :  | -.  | none ]
```

y en la figuras siguientes vemos los efectos de modificaciones realizadas con `set`.



`set(id(4),'linestyle','-.')`



`set(id(3),'xgrid','on','ygrid','on')`

Podemos sustituir los identificadores con la siguiente notación simplificada,

`gcf`	`get(0,'CurrentFigure')`
`gca`	`get(gcf,'CurrentAxes')`
`gco`	`get(gcf,'CurrentObject')`. Es el último objeto *pinchado* con el ratón

Las funciones para borrar figuras, ejes, identificadores son:

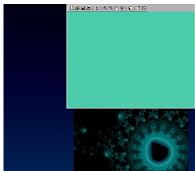
`clf`	Borra la figura actual
`clf(id)`	Borra la figura cuyo identificador es `id`
`cla`	Borra los ejes actuales
`cla(id)`	Borra los ejes cuyo identificador es `id`
`delete(id)`	Borra el objeto gráfico con identificador `id`

Notar que `delete('file')` borra el fichero de nombre `file`.

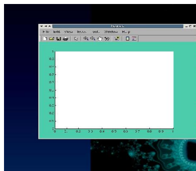
Algunos ejemplos de las funciones anteriores.

```
>> h = figure('color',[0.3 0.8 .67], 'papertype',...
  'a4','pointer','crosshair');                       % fig(1,1)
>> h1 = axes('Position',[.1 .1 .75 .8]);             % fig(1,2)
>> set(h1,'visible','off')                           % fig(1,3)
>> set(h1,'visible','on')                            % fig(2,1)
>> set(h,'position',[0,0,50,100])                    % fig(2,2)
>> set(h,'Position',[10 400 50 50],'resize','off',...
  'menubar','none')                                  % fig(2,3)
>> plot(rand(10,1))                                  % fig(3,1)
>> delete(h1)                                        % fig(3,2)
>> delete(h)                                         % fig(3,3)
```

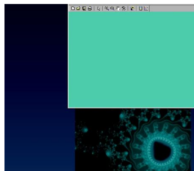
Mostramos la secuencia de imágenes relativas a estos sentencias.



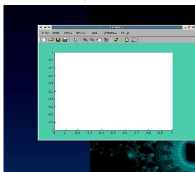
fig(1,1)



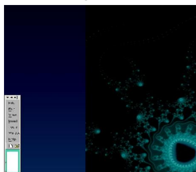
fig(1,2)



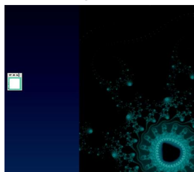
fig(1,3)



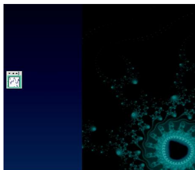
fig(2,1)



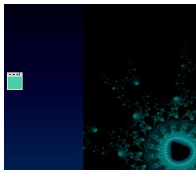
fig(2,2)



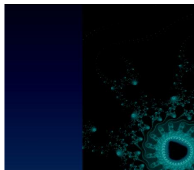
fig(2,3)



fig(3,1)



fig(3,2)



fig(3,3)

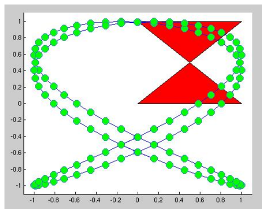
En este ejemplo, se utiliza `findobj` para hallar todos los objetos y la orden `delete` para borrarlos. Notar que el objeto raíz (pantalla) no se puede borrar.

```
>> plot(rand(10));
>> id = findobj, get(id,'type'),
>> for i=13:-1:1, pause(0.3), delete(id(i)); end
```

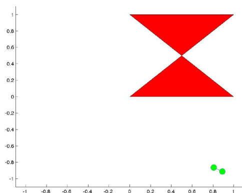
Ejemplo.- Se pueden realizar animaciones en MATLAB controlando la variable `'erasemode'`, que define la forma en que se trata el dibujo y el borrado de objetos de tipo `line`.

`normal`	Es el defecto. Redibuja la región de la pantalla que se modifica (ocultación de líneas, ...). Es lenta pero precisa. Al realizar animaciones, los ejes pueden parpadear cuando el objeto se redibuja o se borra.
`none`	No borra el objeto anterior cuando se actualiza. En este caso, el dibujo sólo es visible en pantalla.
`background`	El objeto se borra a sí mismo cambiando el color del fondo de la figura (background). El problema es que puede dañar a otros objetos dibujados con anterioridad (grid, ...)
`xor`	Sólo se borra el objeto y es útil para animación

```
clf, axis([-1.1, 1.1, -1.1, 1.1])
h = line([NaN],[NaN],'marker','o','linesty','-', ...
   'erasemode','none');
patch([0 1 0 1],[0 1 1 0],'r');
t = (0:pi/60:2*pi);
for n = 2:length(t),
  set(h,'xdata',cos(3*t((n-1):n)+pi/5),'ydata',...
    sin(2*t((n-1):n)-pi/3)),
  set(h,'markerfacecolor',[0,1,0],'markeredgecolor',...
    [0.2 0.5 0.8],'markersize',12);
  pause(0.05),
end
```

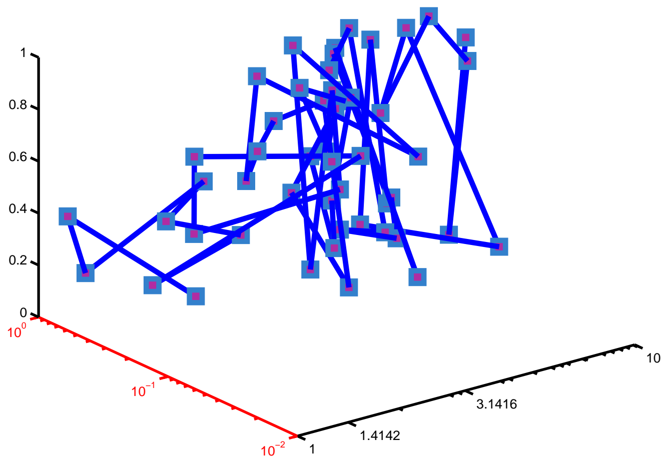


pantalla

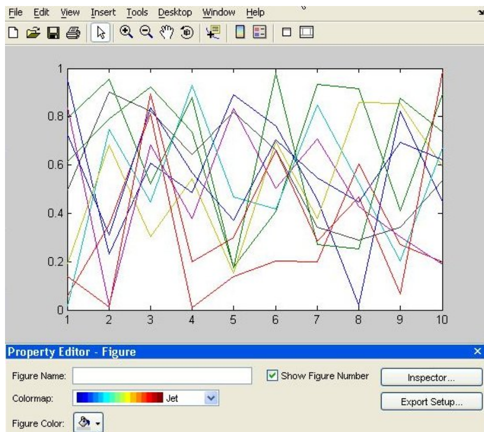


impresora

```
f=axes();
x=linspace(1,10,50); y=exp(x).*x.^2;
h=plot(x, y);
set(f,'box','off')
set(f,'xscale','log','yscale','log')
set(h,'ydata',rand(size(x)))
set(f,'Xtick',[0 1 sqrt(2) pi 10])
set(f,'Ycolor','red')
set(f,'LineWidth',2)
set(h,'LineWidth',4)
set(h,'marker','square')
set(f,'TickDir','out')
set(h,'markerfacecolor',rand(1,3),'markeredgecolor',...
  [0.2 0.5 0.8],'markersize',12);
set(h,'zdata',rand(size(x)))
view(3)
```


1
0.8
0.6
0.4
0.2
0
10^0
10^{-1}
10^{-2}
1
1.4142
3.1416
10

Existe la posibilidad[23] de usar la sentencia `propertyeditor` para acceder de modo visual al contenido de las variables de los objetos gráficos.



[23] Sólo en plataformas windows versión 7.x

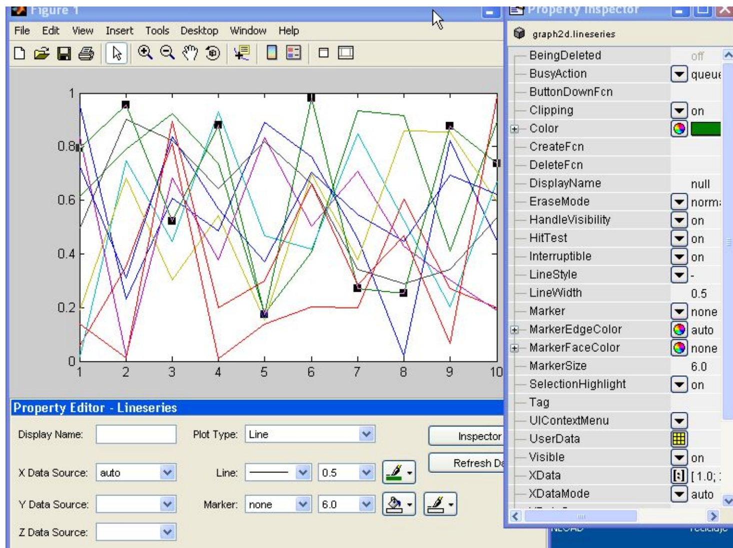


Figure 1
File Edit View Insert Tools Desktop Window Help
1
0.8
0.6
0.4
0.2
0
1 2 3 4 5 6 7 8 9 10
Property Editor - Lineseries
Display Name:
Plot Type: Line
Inspector
X Data Source: auto
Line: 0.5
Refresh Da
Y Data Source:
Marker: none 6.0
Z Data Source:
Property Inspector
graph2d.lineseries
BeingDeleted off
BusyAction queue
ButtonDownFcn
Clipping on
Color
CreateFcn
DeleteFcn
DisplayName null
EraseMode norm
HandleVisibility on
HitTest on
Interruptible on
LineStyle -
LineWidth 0.5
Marker none
MarkerEdgeColor auto
MarkerFaceColor none
MarkerSize 6.0
SelectionHighlight on
Tag
UIContextMenu
UserData
Visible on
XData [1.0;
XDataMode auto

9. Optimizaciones

Para incrementar la velocidad de ejecución de MATLAB, es conveniente seguir un conjunto de reglas básicas:

- Dimensionar los vectores y/o matrices que utilizamos en el código. Aunque MATLAB permite dimensionar dinámicamente, es **muy lento**.

Ejemplo.- Cálculo de la matriz de Hilbert para $n = 1000$.

```
% No se define a=zeros(n) (tiempo de ejec.  39.418 sg)
function a = hilb0(n)
for j=1:n;
  a(1:n,j) = 1./((0:n-1)+j)';
end;
```

```
% Se define a=zeros(n) (tiempo de ejec.  0.1990 sg)
function a = hilb00(n)
a = zeros(n);
for j=1:n;
  a(1:n,j) = 1./((0:n-1)+j)';
end;
```

● Otra optimización muy importante, como se ha comentado anteriormente, es la vectorización. Veamos distintas formas de definir la matriz de Hilbert:

Ejemplo.- Definición de la matriz de Hilbert de dimensión 2000.

```
% Código NO vectorizado (tiempo de ejecución 48.825 sg)
function a = hilb1(n)
a=zeros(n);
for i=1:n;
  for j=1:n;
    a(i,j) = 1/(i+j-1);
  end;
end;
```

```
% Código vectorizado (tiempo de ejecución 0.672 sg)
% Por columnas
function a = hilb2(n)
a=zeros(n);
for j=1:n;
  a(1:n,j) = 1./((0:n-1)+j)';
end;
```

```
% Código vectorizado (tiempo de ejecución 1.0535 sg)
% Por filas
function a = hilb3(n)
a=zeros(n);
for j=1:n;
  a(j,1:n) = 1./((0:n-1)+j);
end;
```

```
% Función MATLAB (tiempo de ejecución 2.396 sg)
function a = hilb4(n)
J = 1:n;
J = J(ones(n,1),:); % "Tony's trick"
I = J';
E = ones(n,n);
a = E./(I+J-1);
```

```
% Código OCTAVE (tiempo de ejecución 1.096 sg)
function a = hilb5(n)
a = [1:n]'*ones(1,n)+ones(n,1)*[0:n-1];
a = 1 ./ a;
```

Ejemplo.- Definición de la matriz de la transformada de Fourier de dimensión $n = 500$,

$$F = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w & w^2 & \dots & w^{n-1} \\ 1 & w^2 & w^4 & \dots & w^{2(n-1)} \\ \vdots & & & \vdots & \\ 1 & w^{n-1} & w^{2(n-1)} & \dots & w^{(n-1)^2} \end{bmatrix}$$

siendo $w = \mathrm{e}^{2\pi i/n}$.

```
% Código NO vectorizado (tiempo de ejecución 6.96 sg)
function a = fou1(n)
w = exp(2*pi*j/n); a = zeros(n);
for k = 1:n; for l = 1:n
  a(k,l)= w^((k-1)*(l-1));
end; end;
```

```
% Código vectorizado (tiempo de ejecución 0.805 sg)
function a = fou2(n)
w = exp(2*pi*j/n); a = zeros(n);
v = w.^(0:n-1);
for i=1:n; a(i,:)= v.^(i-1); end;
```

- Utilizar condiciones lógicas (*masking*) para evitar bucles. Supongamos que queremos evaluar la siguiente función

$$f(x) = \begin{cases} 1 + \sin(2\pi x), & \text{si } |x| > 0.5, \\ 0, & \text{si } |x| \leq 0.5. \end{cases}$$

Podemos definir las siguientes funciones MATLAB:

```
% Código sin masking (tiempo de ejecución 0.8218 sg)
% x=rand(1000,1); tic; mask1(x); toc
function f = mask1(x)
f=zeros(size(x));
for i = 1:length(x)
  if abs( x(i)) > 0.5
    f(i) = 1 + sin(2*pi*x(i));
  end;
end;
```

```
% Código con masking (tiempo de ejecución 0.0451 sg)
% x=rand(1000,1); tic; mask2(x); toc
function f = mask2(x)
f=zeros(size(x));
mask = abs(x) > 0.5;
f(mask) = 1+ sin(x(mask)*2*pi);
```

10. Entradas/Salidas

Hay bastantes funciones para lectura/escritura de ficheros[24]. Usualmente, interesa la escritura de variables a un fichero o la lectura de ellas desde un fichero.

En MATLAB, antes de leer/escribir en un fichero es preciso utilizar la orden `fid=fopen('fichero','permiso')`, que nos da un identificador para su posterior uso. `'fichero'` es el nombre del fichero para lectura/escritura y `permiso` puede tener diversos valores, entre los que destacamos:

`'w'`	escritura
`'r'`	lectura
`'a'`	añadir a un fichero

Para la escritura utilizaremos la orden `fprintf(fid, 'formato')`.

[24] `help iofun`

Consideremos el código siguiente para la escritura a un fichero de una matriz aleatoria 3×5:

```
>> A = rand(3,5)
A =
  0.0153 0.9318 0.8462 0.6721 0.6813
  0.7468 0.4660 0.5252 0.8381 0.3795
  0.4451 0.4186 0.2026 0.0196 0.8318
>> fid = fopen('file.out','w');
>> fprintf(fid,'%f %f %f %f %12.8f \n',A);
>> fclose(fid); type file.out
  0.015274 0.746786 0.445096 0.931815 0.46599434
  0.418649 0.846221 0.525152 0.202647 0.67213747
  0.838118 0.019640 0.681277 0.379481 0.83179602
```

Notar que MATLAB escribe la matriz A por columnas,[25] siguiendo el formato dado por `fprintf`.

En el caso de lectura de un fichero `'datos.in'`, usaremos la orden `fscanf(fid, 'formato', tamaño)`:

[25] herencia de FORTRAN

También puede usarse la sentencia `save fichero variables formato` para guardar una o varias variables en un fichero.

```
>> A = rand(2,2), x = pi;
A =
0.6822 0.5417
0.3028 0.1509
>> save datos.dat A x -ASCII -DOUBLE
>> type datos.dat
6.8222322359138354e-01 5.4167385389808764e-01
3.0276440077660899e-01 1.5087297614976464e-01
3.1415926535897931e+00
```

```
>> load -ASCII resul.res
>> % carga el fichero resul.res en la variable resul.
>> % resul.res debe tener el mismo número de columnas
>> % en todas las filas
```

Si se sustituye `ASCII` por `MAT` (formato binario), se pueden leer todas las variables, independientemente de su dimensión.

```
>> A = rand(2,2); x= pi;
>> save datos.dat A x -MAT % no es necesario -DOUBLE
>> clear all; load -MAT datos.dat A x
>> % carga las variables A y x de datos.dat
```

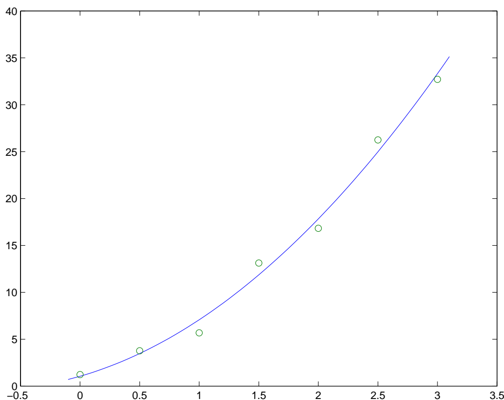
```
>> type datos.in
1.12 1.38383 -2.383
1.112 2.47474 3.393
2.1212e2 3.464 -34.345
4.454e-2 5.75 .9876
6.44 7.98e2 -9.0292
fid = fopen('datos.in','r');
>> % MATLAB lee el fichero rellenando por
>> % columnas la matriz dimensionada en fscanf.
>> % El número de columnas es Inf para permitir
>> % cualquier número de líneas en el fichero.
>> A = fscanf(fid,'%f',[3,Inf])
A =
   1.1200   1.1120   212.1200   0.0445     6.4400
   1.3838   2.4747     3.4640   5.7500   798.0000
  -2.3830   3.3930   -34.3450   0.9876    -9.0292

>> A = A' % transponer y queda igual que en el fichero.
```

Si hubiera alguna línea incompleta en el fichero, MATLAB toma elementos nulos hasta rellenar la matriz.

Ejemplo.- Ajustar por mínimos cuadrados una parábola al conjunto de datos que se encuentran en el fichero `datos1.in`.

```
>> type datos1.in
      x         y
       0    1.2381
  0.5000    3.7696
       :         :
  3.0000   32.7095
>> fid = fopen('datos1.in','r');
>> rot=fscanf(fid,'%s',[1,2]) % quitar el rótulo
rot =
   xy
>> A=fscanf(fid,'%g',[2,Inf]);
>> A=A'; disp(A)
       0    1.2381
  0.5000    3.7696
  1.0000    5.6813
  1.5000   13.1295
  2.0000   16.8318
  2.5000   26.2528
  3.0000   32.7095
>> fclose(fid);
```

```
>> p = polyfit(A(:,1),A(:,2),2) % ajuste de parábola
p =
   2.3657 3.6550 1.0592
>> % y = 2.3657x^2 + 3.6550x + 1.0592
>> xx=-0.1:0.05:3.1;
>> yy=polyval(p,xx); % evaluar el polinomio
>> plot(xx,yy, A(:,1), A(:,2),'o')
```



11. Profiling

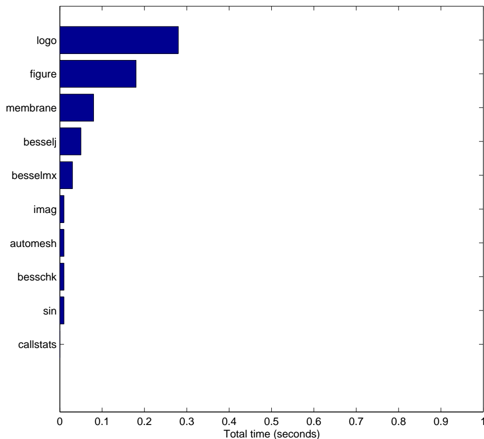
MATLAB tiene un `profiler` sencillo de usar y que nos puede ayudar a investigar que partes de nuestro código dominan el tiempo total de cómputo, líneas que nunca se ejecutan, ...

Una estructura típica de profiling de un fichero **funcion** sería:

```
>> profile on % Inicializa el profiler en forma básica
>> profile on -detail level
>> % El profiling se hace de todas las m-funciones,
>> % m-ficheros y mex-funciones que se llamen (defecto).
>> profile on -detail builtin
>> % Se añaden más funciones como eig
>> profile on -detail operator
>> % Se añaden los operadores como (+,-,*, ...)
>> funcion
>> profile report
>> % Hace un report en html del profiler y lo termina.
>> profile plot % Gráfico de tiempos
>> profile off % Se termina el profiler
```

Por ejemplo,

```
>> profile on
>> logo % logo de MATLAB
>> profile report; profile plot; profile off
```



Summary | Function Details

besselj	0.04	80.0%	4	0.010
rot90	0.00	0.0%	1	0.000

100% of the total time in this function was spent on the following lines:

```
               70:   t = sqrt(lambda)*r;
0.01  20%      71:   b1 = besselj(alf1,t);
0.01  20%      72:   b2 = besselj(alf2,t);
               73:   A = [b1(:,k1) b2(:,k2)];

               98:   for j = 1:np
0.02  40%      99:      S = S + c(j) * besselj(alfa(j),r) .* sin(alfa(j)*theta);
              100:   end

              106:   if sym == 3, L = L + L' - diag(diag(L)); end
0.01  20%     107:   L = rot90(L/max(max(abs(L))),-1);
              108:
```

12. Algunos ejemplos de Análisis Numérico

Inicialmente MATLAB se diseñó para el Análisis Numérico, por lo que no es¡de extrañar que tenga gran cantidad de funciones relativas a este campo de la¡Matemática Aplicada (Álgebra lineal, no lineal, interpolación, mínimos cuadrados, cuadraturas, ecuaciones diferenciales, . . .)

Entre las funciones del Álgebra lineal podemos mencionar:

[L,U,P]=lu(A)	Factorización LU de A con pivote parcial P*A=L*U
cond(A,p)	Estimación del condicionamiento de la matriz A en la norma p
[Q,R]=qr(A)	Factorización QR de A
[Q,LAMBDA]=eig(A)	Valores y vectores propios de A.
[U,S,V] = svd(A)	Descomposición singular de A

Ejemplo.- Resolución de sistemas lineales.

```
>> b=ones(14,1); A = hilb(14);
>> x = A\b;
Warning:  Matrix is close to singular or badly scaled.
Results may be inaccurate.  RCOND = 1.464356e-18.
>> norm( A*x-b,Inf), norm( A*x-b,1)
ans =
  3.1316e-09
ans =
  7.4625e-09
>> % Con la factorización lu, es:
>> [l,u,p]=lu(A);
>> x=u\(l\(p*b));
>> cond(A,1)
ans =
  6.8289e+17
>> 1/ans
ans =
  1.4644e-18
```

Ejemplo.- Entre las aplicaciones (académicas) de la descomposición en valores singulares (SVD), está la de comprimir imágenes[26].

La SVD de una matriz $A \in \mathbb{R}^{m\times n}$, es

$$A = U\Sigma V^T,$$

donde U y V son matrices ortogonales con $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{n\times n}$ y $\Sigma \in \mathbb{R}^{m\times n}$ conteniendo los valores singulares de A

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0,$$

en su diagonal principal.

La idea es elegir $k << n$ tal que $A_k := \sum_{i=1}^{k} \sigma_i u_i v_i^T$ sea una buena aproximación a la matriz A. Puesto que MATLAB permite almacenar una imagen en una matriz, puede considerarse que se comprime la imagen.

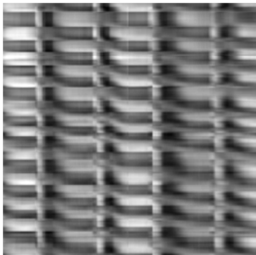
[26] Requiere Image_Toolbox

```
A=imread('basket.jpg');
% matriz 256 x 256, escala de grises
% Pasar la matriz a coma flotante para la SVD
A = im2double(A);
imshow(A)
[U,S,V] = svd(A);
k=20; Ak = U(:,1:k) * S(1:k,1:k) * V(:,1:k)';
imshow(Ak)
```

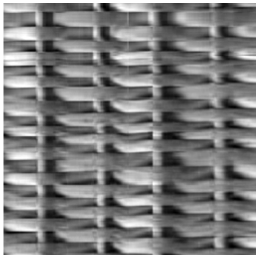
Con imágenes en color, `imread` da una matriz `M×N×3`, (o 3 matrices `M×N` correspondientes a la codificación RGB respectivamente). En este caso, el código anterior modificado es:

```
A = imread('imagen_color.jpg');
[m,n,p] = size(A); A = im2double(A);
A = reshape(A,m,n*p); [U,S,V] = svd(A);
k=20; Ak = U(:,1:k) * S(1:k,1:k) * V(:,1:k)';
Ak = reshape(Ak,m,n,p);
imshow(Ak)
```

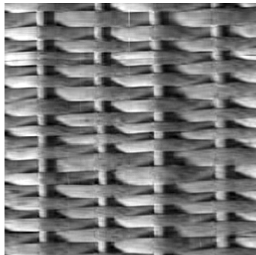
5 valores singulares



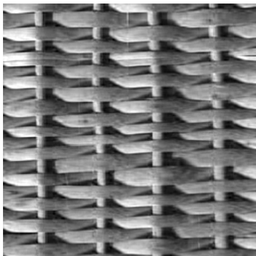
10 valores singulares



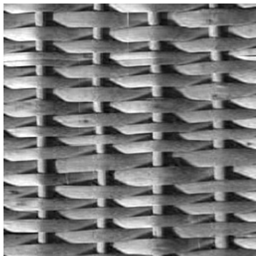
20 valores singulares

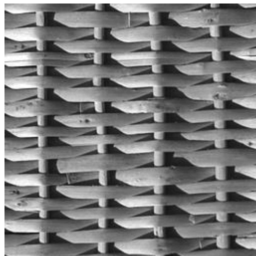


30 valores singulares



40 valores singulares





En línea con lo anterior, podría generarse una película cuyos fotogramas fueran las imágenes (matrices) almacenadas en A_k, $k = 1, \ldots,$ `n_frames`. Para este fin, `getframe(gcf)`[27] captura la ventana gráfica almacenándola en la variable `M(k)` de tipo `struct`.

```
A=imread('basket.jpg'); A=im2double(A); imshow(A)
[U,S,V] = svd(A);
n_frames=10;
for k=1:n_frames;
 Ak = U(:,1:k) * S(1:k,1:k) * V(:,1:k)';
 imshow(Ak), xlabel(k);
 M(k) = getframe(gcf);
end
clf; axis off; movie(M,1,1) % 1 frame por segundo
```

Guardar la película en formato `.avi`, con una calidad del 90% y un fotograma por segundo. A más calidad, mayor es el tamaño del fichero.[28]

```
movie2avi(M,'file.avi','quality',90,'fps',1)
```

[27] Notar la diferencia con `gca` (help gcf, help gca)
[28] En windows admite varios codecs de compresión

Ejemplo.- Dibujo de los conjuntos de Mandelbrot $\mathcal{M}$ y Julia $\mathcal{J}$, definidos por:

$$\mathcal{M} = \left\{ c \in \mathcal{C} \,|\, \lim_{n\to\infty} |z_n| < \infty \right\}, \quad z_{n+1} = z_n^2 + c,$$

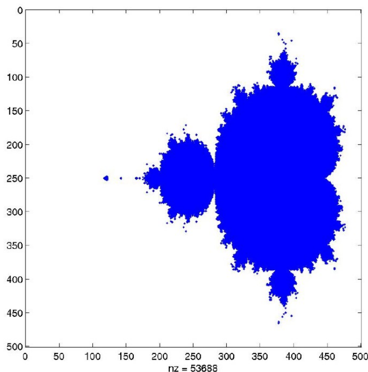
$$\mathcal{J} = \left\{ c \in \mathcal{C} \,|\, \lim_{n\to\infty} |z_n| < \infty \right\}, \quad z_{n+1} = z_n^2 + k,$$

con $z_0 = c$, $z_n,\ c,\ k \in \mathcal{C}$.

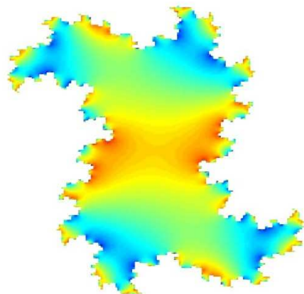
```
function Z=mandel(n,itemax)
x=linspace(-2.5,0.6,n); y=linspace(-1.2,1.2,n);
[X,Y] = meshgrid(x,y); C = complex(X,Y);
Zmax = 1e10; Z=C;
for k=1:itemax,  Z = Z.^2 + C; end
spy( Z < Zmax);
% spy(A) dibuja los elementos no nulos de A.
% Suele usarse para matrices ``huecas"
```

```
function Z=julia(n,itemax)
x=linspace(-1.5,1.5,n); y=linspace(-1.2,1.2,n);
[X,Y] = meshgrid(x,y); Z = complex(X,Y); Zmax = 1e10;
for k=1:itemax,  Z = Z.^2 + 0.353+0.288i; end
Z=abs(Z); Z(Z>Zmax)=NaN; surf(X,Y,Z), shading interp
view(2), axis off;
```

Notar que si `zn=complex(Inf,Inf)`, entonces `zn+1=complex(NaN,Inf)` y `zn+2=complex(NaN,NaN)`.



$\mathtt{Z} = \mathtt{mandel}(500, 50);$



$\mathtt{Z} = \mathtt{julia}(200, 50);$
$\mathtt{k} = 0.353 + 0.288\mathtt{i}$

Ejemplo.- MATLAB dispone de un conjunto bastante bueno de integradores numéricos de Problemas de valor inicial (P.V.I.)

$$\begin{aligned} &y' = f(t, y), \quad t \in [t_0, t_f], \\ &y(t_0) = y_0 \end{aligned}$$

basados en métodos de un paso y multipaso (ver tabla). Son capaces de integrar un amplio abanico de problemas y su implementación está documentada en ode suite.

nombre	tipo	orden	algoritmo
`ode45`	No *Stiff*	5(4)	Par encajado Explícito Dormand & Prince
`ode23`	No *Stiff*	3(2)	Par encajado Explícito Bogacki & Shampine
`ode113`	No *Stiff*	1-11	Multipaso de orden variable basado en las fórmulas de Adams-Bashforth-Moulton PECE
`ode15s`	*Stiff*	1-5	Multipaso de orden variable basado en las fórmulas de diferenciación numérica (opcionalmente puede usar BDF)
`ode23s`	*Stiff*	3(2)	Par modificado de Rosenbrock
`ode23t`	Semi *Stiff*	3(2)	Regla del Trapecio implícita con un interpolante.
`ode23tb`	*Stiff*	3(2)	Implementación *tipo* RK de la regla del trapecio y BDF2.

Ejemplo.- Integración del P.V.I. de Lorenz[29], dado por:

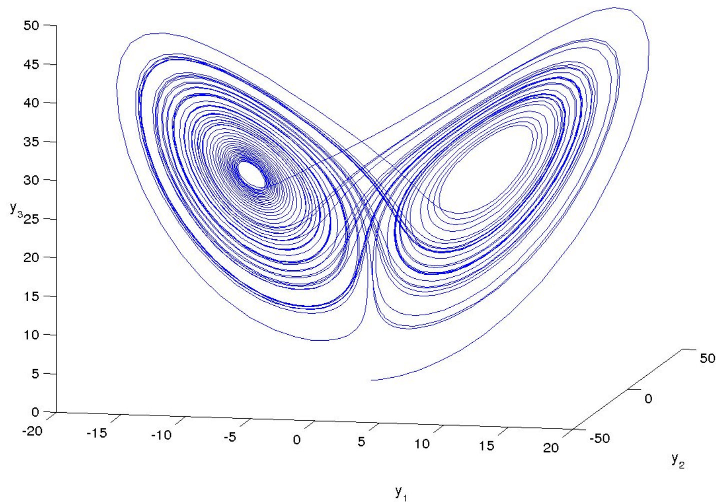
$$\begin{aligned} y_1' &= 10(y_2 - y_1), & y_1(0) &= 0, \\ y_2' &= 28y_1 - y_2 - y_1 y_3, & y_2(0) &= 1, \\ y_3' &= y_1 y_2 - 8y_3/3, & y_3(0) &= 0, \end{aligned}$$

en el intervalo $[0, 60]$, con la función `ode45`, que usa el par encajado Runge–Kutta explícito de Dormand & Prince de órdenes DP5(4)7FM.

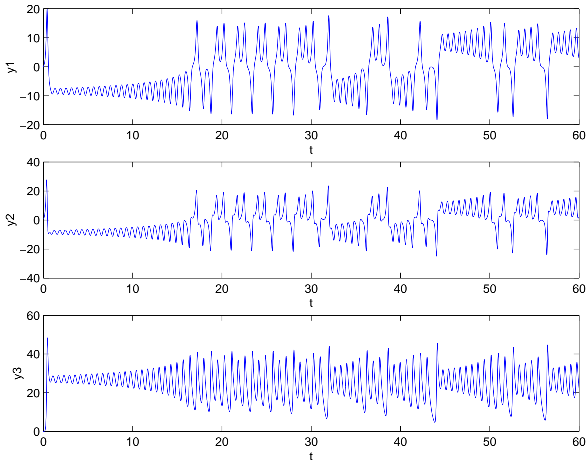
```
function f = lorenzz(t,y)
% Definición de la función derivada en lorenzz.m
% No dejar espacios en blanco !!!
f = [10*(y(2)-y(1)) % filas en líneas separadas
  28*y(1)-y(2)-y(1)*y(3)
  y(1)*y(2)-8/3*y(3)];
```

```
t0=[0 60]; y0 = [0;1;0]; % valores iniciales
[t,y] = ode45(@lorenzz,t0,y0);
plot3( y(:,1), y(:,2), y(:,3))
xlabel('y_1'); ylabel('y_2'); zlabel('y_3','Rotation',0)
view(12,12), title('Mariposa de Lorenz')
```

[29]Ejecutar `lorenz` en una sesión de MATLAB


Mariposa de Lorenz
50
45
40
35
30
25
20
15
10
5
0
y3
-20
-15
-10
-5
0
5
10
15
20
y1
50
0
-50
y2

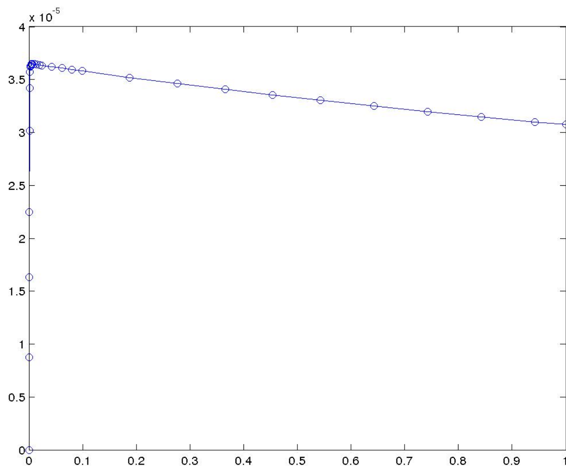
```
>> clf; % Borra la ventana gráfica
>> subplot(311),plot(t,y(:,1)), xlabel('t'),ylabel('y1')
>> subplot(312),plot(t,y(:,2)), xlabel('t'),ylabel('y2')
>> subplot(313),plot(t,y(:,3)), xlabel('t'),ylabel('y3')
```



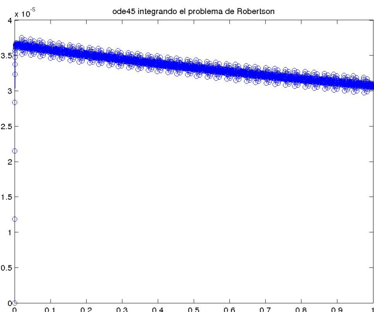
Problema de Robertson implementado en el fichero `chem.m`

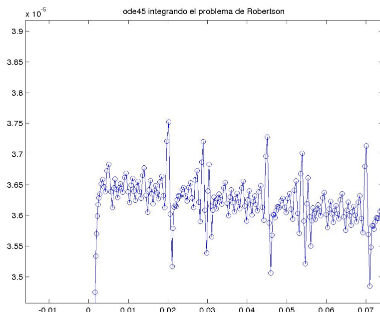
```
function f=chem(t,y)
f=[ -0.04*y(1)+10^4*y(2)*y(3)
  0.04*y(1)-10^4*y(2)*y(3)-3*10^7*y(2)^2
  3*10^7*y(2)^2 ];
```

```
>> ta=[0 1]; ya=[1;0;0];
>> opts=odeset('RelTol',1e-6,'AbsTol',1e-6,'Stats','on');
>> [T,Y] = ode15s(@chem,ta,ya,opts);
30 successful steps
5 failed attempts
71 function evaluations
2 partial derivatives
13 LU decompositions
61 solutions of linear systems
>> plot(T,Y(:,2),'-o')
```


x 10^-5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

```
>> opts=odeset('RelTol',1e-6,'AbsTol',1e-6,'Stats','on');
>> ta=[0 1]; ya=[1;0;0]; % valores iniciales
>> [T,Y] = ode45(@chem,ta,ya,opts);
679 successful steps
170 failed attempts
5095 function evaluations
>> plot(T,Y(:,2),'-o')
```





Hay más ejemplos de resolución numérica de ecuaciones diferenciales, como:

- problemas de contorno (`help bvp4c`)
- problemas con matrices de masas (`help fem1ode, fem2ode, batonode`)
- D.A.E, (`help amp1dae`)

13. Interfaces Gráficos

TODO

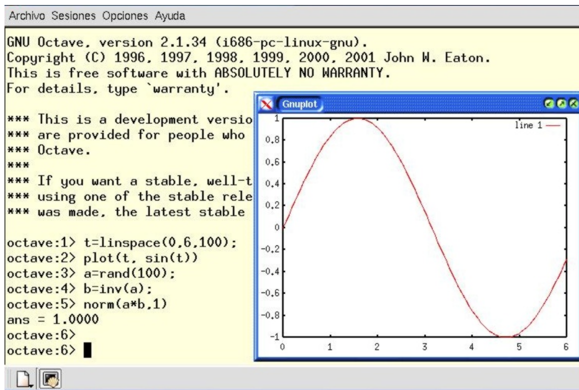
14. Matrices Huecas (*Sparse*)

TODO

15. Miscelánea

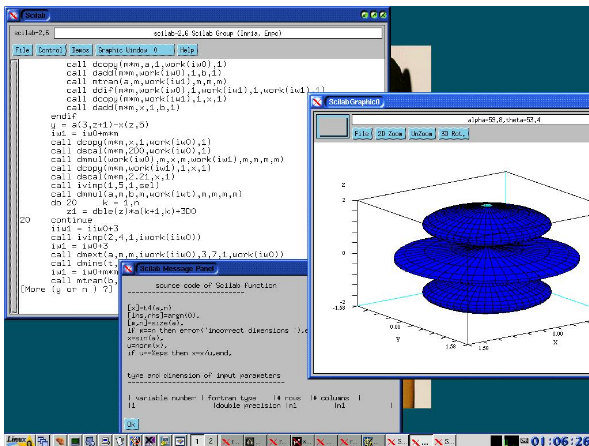
La gran mayoría de las órdenes empleadas en este tutorial son compatibles con OCTAVE. Las principales excepciones son las leyendas en los gráficos y exportar gráficos en formato postscript.

octave

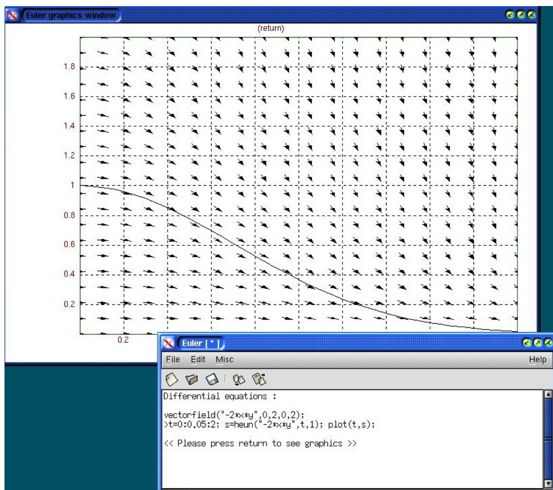


Las siguientes figuras muestran diferentes capturas de pantalla de algunos manipuladores numéricos similares a MATLAB:

Scilab



Euler



Bibliografía

[1] D. F. Griffiths. An Introduction to MATLAB version 2.1. (1997)

[2] John W. Eaton. GNU OCTAVE. (1997).

[3] D. J. Higham, N. J. Higham. Matlab Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. (2000).

[4] Getting Started with MATLAB. The MathWorks, Inc., Natick, MA, USA. Versión online.