UNA INTRODUCCIÓN RÁPIDA A SCILAB

Scilab es un paquete de software libre de código abierto para computación científica, orientado al cálculo numérico, a las operaciones matriciales y especialmente a las aplicaciones científicas y de ingeniería.

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

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

Inicialmente desarrollado por el INRIA (Institut National de Recherche en Informatique et Automatique), actualmente está a cargo de un Consorcio de universidades, empresas y centros de investigación. Website: **http://www.scilab.org**

La versión actual es la 4.1.2. Tanto los binarios para varias plataformas (GNU Linux, Windows 2000/XP/Vista, Hp-UX) como los fuentes están disponibles en **http://www.scilab.org/download**

1. DOCUMENTACIÓN Y AYUDA ON-LINE

Scilab dispone de un manual de usuario que se puede consultar en una ventana de ayuda (**Help Browser**). Esta ayuda se puede invocar desde la barra de herramientas (**? ---> Scilab Help** en Windows, **Help ---> Help Browser** en Linux) o escribiendo mediante el comando `help()`. Se puede acceder fácilmente a la descripción de todas las funciones que, en muchos casos, se acompaña de ejemplos de uso ilustrativos.

También dispone de un conjunto de **Demos** que pueden ayudar mucho al aprendizaje de Scilab ya que se acompañan, en general, del código que genera la demostración.

Sin embargo, Scilab no dispone de un tutorial propiamente dicho que facilite los primeros pasos, que explique la filosofía general del lenguaje o que indique cómo resolver problemas técnicos concretos.

En las páginas **http://www.scilab.org/product** y **http://www.scilab.org/publications** se pueden encontrar documentación adicional y referencias a libros y artículos relativos a Scilab.

2. SCRIPTS Y FUNCIONES. EL EDITOR SCILAB INTEGRADO

2.1 Scripts

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

Así, para llevar a cabo una tarea, en vez de escribir las instrucciones una por una en la línea de comandos de Scilab, se pueden escribir una detrás de otra en un fichero. Para ello se puede utilizar el **Editor integrado** de Scilab: botón "**Editor**" de la barra de menús o bien usando la orden

```
-->scipad()
```

Por convenio, los scripts de Scilab tienen el sufijo `.sce` .

Para ejecutar un script se usa la orden

```
-->exec(nombre_fichero)          // repite todas las instrucciones en pantalla

-->exec(nombre_fichero, -1)      // para que no repita las instrucciones
```

Lógicamente, en el nombre del fichero hay que incluir el **path**, caso de que el fichero no esté en el directorio actual. El nombre del fichero debe ir entre apóstrofes o comillas dobles, puesto que es una cadena de caracteres.

2.2 Funciones

Es posible definir nuevas **funciones** Scilab. La diferencia entre un script y una función es que esta última tiene una "interfase" de comunicación con el exterior mediante **argumentos de entrada y de salida**.

Las funciones Scilab responden al siguiente formato de escritura:

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

Las funciones se pueden definir on-line o bien escribiéndolas en un fichero (ASCII). A los ficheros que contienen funciones Scilab, por convenio, se les pone el sufijo `.sci` . Las funciones definidas on-line están disponibles de modo inmediato. Las funciones guardadas en un fichero hay cargarlas en el espacio de trabajo de una sesión de Scilab mediante el comando **exec**. Se pueden incluir varias funciones en un mismo fichero, una a acontinuación de otra.

También es posible definir funciones on-line mediante el comando **deff**:

```
-->deff('[arg_out]=nombre(arg_in)','instrucciones')
```

EJEMPLO

```
-->function [y]=fun1(x), y=2*x+1, endfunction
-->sqrt(fun1(4))
ans  =
    3.

-->deff('[s]=fun2(x,y)','s=sqrt(x^2+y^2)')
-->fun2(3,4)
ans  =
    5.

-->exec('misfunciones.sci',-1)
-->t=5;
-->max(fun3(t),0)
ans  =
    0.
```

FICHERO misfunciones.sci

```
function [z]=fun3(x)
z=sin(x);
endfunction
```

3. OBJETOS Y SINTAXIS BÁSICOS

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

El objeto básico de trabajo de Scilab es una matriz bidimensional cuyos elementos son números reales o complejos. Escalares y vectores son casos particulares de matrices. También se pueden manipular matrices de cadenas de caracteres, booleanas, enteras y de polinomios. También es posible construir otro tipo de estructuras de datos definidos por el usuario.

Algunas constantes numéricas están predefinidas. Sus nombres comienzan por el símbolo `%`. En particular `%pi` es el número π, `%e` es el número **e**, `%i` es la unidad imaginaria, `%eps` es la precisión de la máquina (mayor número real doble precisión para el que `1+%eps/2` es indistinguible de `1`), `%inf` es el infinito-máquina (overflow: cualquier número que supere al mayor número real representable en doble precisión), `%nan` es el símbolo **NaN** (Not a Number) para una operación inválida (por ejemplo, `0/0` es `%nan`).

El lenguaje de Scilab es interpretado, esto es, las instrucciones se traducen a lenguaje máquina una a una y se ejecutan antes de pasar a la siguiente. Es posible escribir varias instrucciones en la misma línea, separándolas por una **coma** o por **punto y coma**.
Scilab distingue entre mayúsculas y minúsculas: `%nan` **NO ES LO MISMO QUE** `%Nan`

Se pueden recuperar comandos anteriores, usando las teclas de flechas arriba y abajo. Con las flechas izquierda y derecha nos podemos desplazar sobre la línea de comando y modificarlo.

3.1 Constantes y operadores aritméticos

Reales: `8.01` `-5.2` `.056` `1.4e+5` `0.23E-2` `-.567d-21` `8.003D-12`
Complejos: `1+2*%i`
Booleanos: `%t` `%f`
Caracteres (entre apóstrofes o comillas): `'esto es una cadena de caracteres'` `"string"`
Operadores aritméticos: `+` `-` `*` `/` `^`
Operadores de comparación: `==` `~=` (ó `<>`) `<` `>` `<=` `>=`
Operadores lógicos: `&` `|` `~`

3.2 Funciones elementales

Los nombres de las funciones elementales son los habituales. Algunas de ellas:

`sqrt(x)`	raiz cuadrada	`sin(x)`	seno (radianes)
`abs(x)`	módulo	`cos(x)`	coseno (radianes)
`conj(z)`	complejo conjugado	`tan(z)`	tangente (radianes)
`real(z)`	parte real	`cotg(x)`	cotangente (radianes)
`imag(z)`	parte imaginaria	`asin(x)`	arcoseno
`exp(x)`	exponencial	`acos(x)`	arcocoseno
`log(x)`	logaritmo natural	`atan(x)`	arcotangente
`log10(x)`	logaritmo decimal	`cosh(x)`	cos. hiperbólico
`rat(x)`	aprox. racional	`sinh(x)`	seno hiperbólico
`modulo(x,y)`	resto de dividir x por y	`tanh(x)`	tangente hiperbólica
`floor(x)`	n tal que n<=x<(n+1)	`acosh(x)`	arcocoseno hiperb.
`ceil(x)`	n tal que (n-1)<x<=n	`asinh(x)`	arcoseno hiperb.
`int(x)`	parte entera inglesa: floor(x) si x>=0 ceil(x) si x<0	`atanh(x)`	arcotangente hiperb.

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

3.3 Uso como calculadora

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

EJEMPLO

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

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

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

3.4 Variables

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

EJEMPLOS

```
-->a=10
a =
     10.
--> pepito=exp(2.4/3)
pepito  =
    2.2255409
-->pepito=a+pepito*(4-0.5*%i)
pepito  =
    18.902164 - 1.1127705i
```

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

`who`	lista las variables actuales
`whos`	como el anterior, pero más detallado
`clear`	elimina todas las variables que existan en ese momento
`clear a b c`	elimina las variables a, b y c (atención: sin comas!)
`browsevar()`	abre, en ventana aparte, un "ojeador" de la memoria de trabajo de Scilab: permite "ver" el contenido y características de las variables e, incluso, editar su valor.

3.5 Formatos

Por defecto, Scilab muestra los números en formato "variable" con 10 dígitos. Se puede modificar esto mediante el comando **format**:

`format(14)`	14 dígitos
`format('e')`	formato científico o exponencial, coma flotante
`format('v')`	formato variable (por defecto)
`format('v',20)`	formato variable con 20 dígitos
`format('e',15)`	formato científico con 15 dígitos

3.6 Algunos comandos utilitarios

Están disponibles algunos comandos utilitarios, como:

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

4. MATRICES

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

4.1 Operadores elementales y funciones de construcción de matrices

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

EJEMPLOS (construcciones elementales de matrices)

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

Observaciones:

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

Otras órdenes para crear matrices son:

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

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

EJEMPLOS (construcciones elementales de matrices)

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

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

Algunas funciones útiles para generar matrices aparecen en la tabla siguiente:

`diag(v)`	Si **v** es un vector, **diag(v)** es una matriz cuadrada de ceros con diagonal principal = **v**
`diag(A)`	Si **A** es una matriz, **diag(A)** es un vector = diagonal principal de **A**
`diag(A,k)`	Si **A** es una matriz y **k** es un entero, **diag(A,k)** es un vector = **k**-ésima sub o super diagonal de **A** (según sea **k**<0 ó **k**>0)
`zeros(n,m)`	matriz **n**x**m** con todas sus componentes iguales a cero
`ones(n,m)`	matriz **n**x**m** con todas sus componentes iguales a uno
`eye(n,m)`	matriz unidad: matriz **n**x**m** con diagonal principal =1 y el resto de las componentes =0
`matrix(A,n,m)`	Re-dimensiona una matriz: si **A** es una matriz **h**x**k**, **matrix(A,n,m)** es otra matriz con los mismos elementos que **A**, pero de dimensiones **n**x**m** (tiene que ser **h*****k**=**n*****m**)

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

EJEMPLOS (creación de matrices por bloques)

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

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

4.2 Manipulación de matrices: operadores y funciones

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

Sean: A y B matrices de elementos a_{ij} y b_{ij} y k un escalar	
A+B (A-B)	matriz de elementos a_{ij} + b_{ij} (a_{ij} - b_{ij}) (si dimensiones iguales)
A*B	producto matricial de A y B (si dimensiones adecuadas)
A^k	matriz **A** elevada a la potencia **k**
A+k	matriz de elementos a_{ij} + k
A-k	matriz de elementos a_{ij} - k
k*A	matriz de elementos k * a_{ij}
A/k =(1/k)*A	matriz de elementos a_{ij} / k
A^k	matriz A elevada a la potencia k: si k entero>0, A^k=A*A*...*A si k entero <0, A^k= (inv(A))^(-k) si no, A^k se calcula por diagonalización
k./A	matriz de elementos k / a_{ij}
A.^k	matriz de elementos (a_{ij}) ^ k
k.^A	matriz de elementos k ^ (a_{ij})
A.*B	matriz de elementos a_{ij} * b_{ij} (si dimensiones iguales)
A./B	matriz de elementos a_{ij} / b_{ij} (si dimensiones iguales)
A.^B	matriz de elementos a_{ij} ^ b_{ij} (si dimensiones iguales)

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

EJEMPLO (diferencia entre exp y expm)

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

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

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

`sum(A)`	suma de las componentes de la matriz **A**
`sum(A,1),  sum(a;'r')`	es un vector fila (**r**ow) conteniendo la suma de los elementos de cada columna de **A**
`sum(A,2),  sum(a;'c')`	es un vector columna conteniendo la suma de los elementos de cada fila de **A**
`trace(A)`	traza de A : **sum(diag(A))**
`prod(A)`	producto de las componentes de la matriz A
`prod(A,1), prod(A,'r')`	es un vector fila (**r**ow) conteniendo el producto de los elementos de cada columna de **A**
`prod(A,2), prod(A,'c')`	es un vector columa conteniendo el producto de los elementos de cada fila de **A**
`max(A)` `max(A,'r'), max(A,'c')`	máximo de las componentes de la matriz **A** máximos de columnas y filas resp. (como antes)
`mean(A)` `mean(A,'r'), mean(A,'c')`	media de las componentes de la matriz **A** medias de columnas y filas resp. (como antes)
`norm(v)` `norm(v,2)`	norma euclídea del **vector v**
`norm(v,p)`	norma-**p** del **vector v**: **sum(abs(v).^p)^(1/p)**
`norm(v,'inf')` `norm(v,%inf)`	norma infinito del **vector v:** **max(abs(v))**
`norm(A)` `norm(A,2)`	máximo autovalor de la **matriz A**
`norm(A,1)`	norma-**1** del **matriz A**: máximo entre las sumas de sus columnas: **max(sum(abs(A),'r'))**
`norm(A,'inf')` `norm(A,%inf)`	norma infinito de la **matriz A**: máximo entre las sumas de sus filas: **max(sum(abs(A),'c'))**
`size(A)`	devuelve, en un vector fila, las dimensionbes de la matriz A
`size(A,'r')` `size(A,'c')`	número de filas/columnas de la matriz A
`length(A)`	devuelve un escalar con el número de elementos de la matriz A: si A es una vector, length(A) es su longitud; si A es una matriz length(A) es el producto de sus dimensiones

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

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

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

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

Por ejemplo, si **v** es un vector y **h** es un vector de subíndices, `v(h)` hace referencia al subconjunto de componentes de **v** correspondientes a los valores contenidos en **h**. Análogamente con `A(h,k)` si `A` es una matriz bidimensional y **h** y **k** son vectores de subíndices. El símbolo **:** (dos puntos) en el lugar de un índice indica que se toman todos. El símbolo **$** indica el último valor del subíndice.

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

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

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

```
-->A(2:3,1:2)
```

$$\begin{pmatrix} 2.1 & 2.2 \\ 3.1 & 3.2 \end{pmatrix}$$

```
-->A(:,2)        // representa la segunda columna de A
-->A(:,2:$)      // representa las columnas desde 2 hasta la última
-->A(:)          // representa todos los elementos de A, en una sola columna
```

Si `B` es un vector booleano (sus elementos son `%t` para verdadero y `%f` para falso) entonces `A(B)` especifica la submatriz que se obtiene considerando o no cada elemento en función del valor verdadero o falso del vector booleano:

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

```
-->v=linspace(1,5,9)

-->b=[%t,%t,%t %f %f %t %t %f %f]

-->w=v(b)
```

$v = \begin{pmatrix} 1 & 1.5 & 2 & 2.5 & 3 & 3.5 & 4 & 4.5 & 5 \end{pmatrix}$

$w = \begin{pmatrix} 1 & 1.5 & 2 & 3.5 & 4 \end{pmatrix}$

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

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

```
-->A=rand(4,4)     // Se almacena en A una matriz 4x4 de num. aleatorios

-->A(2,2)=0        // Se modifica el segundo elem. diagonal de A

-->A(5,2)=1        // Obsérvese que A(5,2) no existía:: de hecho, se
                   // MODIFICAN LAS DIMENSIONES de la matriz, para AÑADIR
                   // el elemento A(5,2). El resto se llena con ceros.
                   // Ahora A es una matriz 5x4

-->A(2:3,1:2)=1    // La submatriz A(2:3,1:2) se llena con unos

-->A(:,2)=[]       // El símbolo [] representa una "matriz vacía"
                   // Esta instrucción ELIMINA la segunda columna de A
                   // Ahora A tiene dimensión 5x3

-->A=[A,[1:5]']    // Se añade a A una nueva columna al final
```

5. SISTEMAS LINEALES. AUTOVALORES Y AUTOVECTORES

Para la resolución "rápida" de sistemas lineales Scilab dispone del "operador" **** :

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

```
-->A\b
```

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

Este operador debe ser usado con precaución: Si la matriz **A** es singular o mal condicionada, la instrucción **A\b** emite un mensaje de advertencia y devuelve una solución del correspondiente problema de mínimos cuadrados:

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

Si la matriz **A** no es cuadrada (teniendo el mismo número de líneas que el segundo miembro) entonces la instrucción **A\b** devuelve directamente una solución de mínimos cuadrados sin advertir absolutamente nada. En el caso en que la solución de mínimos cuadrados no es única (la matriz **A** no es de rango máximo), la solución que devuelve **NO** es la de norma mínima. Para obtener esta última es mejor utilizar la instrucción siguiente, que utiliza la pseudo-inversa de la matriz A:

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

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

```
-->spec(A)
```

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

```
-->[X,V]=spec(A)
```

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

Otras funciones de interés pueden ser:

det(A)	determinante de la matriz cuadrada A
rank(A)	rango de la matriz
cond(A)	Número de condición de la matriz A en norma 2: $\lambda_{max} / \lambda_{min}$ (Si es muy grande, la matriz está mal condicionada)
rcond(A)	Inverso del número de condición de la matriz A en norma 1: (Si es próximo a 1, la matriz está bien condicionada; si es próximo a cero está mal condicionada)
inv(A)	inversa de la matriz A
lu(A)	factorización LU de la matriz A
chol(A)	factorización de Cholesky de la matriz A
qr(A)	factorizacion QR de la matriz A

Para información más completa sobre estas y otras funciones se recomienda visitar el epígrafe **Linear Algebra** del Help de Scilab.

Casi todas las funciones Scilab relacionadas con el álgebra lineal numérica están basadas en rutinas de la librería LAPACK.

(Ver ejemplos en script **ejemplo5.sce**)

6. POLINOMIOS

Scilab puede manipular objetos de tipo "polinomio". Algunas formas de generar polinomios son las siguientes:

```
-->p=poly(a,"x")
```
siendo **a** una matriz cuadrada **nxn** es el polinomio caracterísitco de la matriz **a**, con variable simbólica **x**

```
-->a=companion(p)
```
donde **p** es un polinomio es la matriz "compañera" de **p**, también, llamada de Frobenius, es decir la matriz cuyo polinomio característico es **p**.

```
-->p=poly(v,"x","roots")
```
siendo **v** un vector numérico es el polinomio, con variable simbólica **x**, cuyas raíces son las componentes de **v**

```
-->p=poly(v,"s","coeff")
```
siendo **v** un vector numérico es el polinomio, con variable simbólica **s**, cuyos coeficientes son las componentes del vector **v** (en orden de menor a mayor grado)

```
-->t=poly(0,"t")
-->p=1+t-4*t^2+%pi*t^5
```
define **t** como un símbolo que permite definir otros polinomios con variable **t** mediante su expresión algebraica.

```
-->roots(p)
```
calcula las raíces del polinomio **p**

EJEMPLOS

```
-->a=[1,2,3;4,5,6;7,8,-9];
-->p=poly(a,"t")
p  =
                  2   3
  - 54 - 126t + 3t + t

-->p=poly([1,2,3,4],"x","coeff")
p  =
               2    3
    1 + 2x + 3x + 4x
-->roots(p)
ans  =
  - 0.0720852 + 0.6383267i
  - 0.0720852 - 0.6383267i
  - 0.6058296
```

7. RESOLUCIÓN DE ECUACIONES NO LINEALES

Para resolver (sistemas de) ecuaciones no lineales

$$f(x) = 0, \quad x \in R^n, \quad f : R^n \to R^n$$

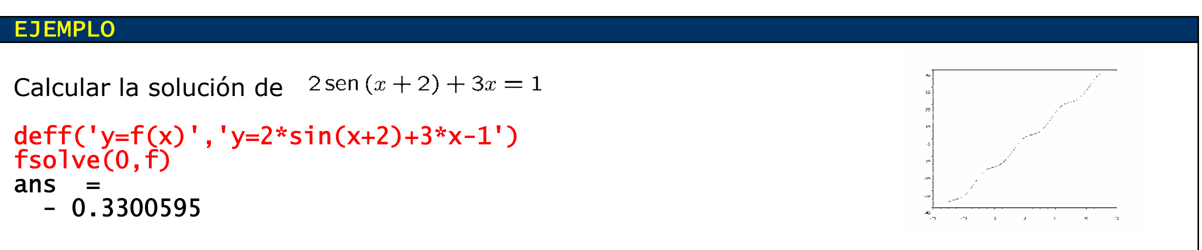
Scilab dispone de la función **fsolve**, cuya utilización más sencilla es:

```
-->x=fsolve(x0,fun)
-->[x,val]=fsolve(x0,fun)
```

donde:

- **fun** es el nombre de la función Scilab que calcula f(x).
  Debe responder a la forma: **[y]=fun(x)**
  (ver en el epígrafe 2 las distintas formas de definir f)
- **x0** es el valor inicial de **x** para comenzar las iteraciones
- **x** es la solución del sistema
- **val** (opcional) es el valor de **f** en la solución

EJEMPLO



Calcular la solución de $2\,\text{sen}\,(x+2) + 3x = 1$

```
deff('y=f(x)','y=2*sin(x+2)+3*x-1')
fsolve(0,f)
ans  =
 - 0.3300595
```

EJEMPLO

Calcular la (única) solución del siguiente sistema de ecuaciones en el primer cuadrante:

$$\begin{cases} x^3 + y^3 = 3 \\ x^2 + y^2 - 2y = 0 \end{cases} \qquad x = (x_1, x_2), \quad f(x) = \begin{pmatrix} x_1^3 + x_2^3 - 3 \\ x_1^2 + x_2^2 - 2x_2 \end{pmatrix} = 0$$

```
deff('[y]=g(x)','y=[x(1)^3+x(2)^3-3,x(1)^2+x(2)^2-2*x(2)]');
[x,v]=fsolve([1,1],g)
 v  =
    0.     0.
 x  =
    0.9587068    1.2843962
```

Si se conoce la derivada de la función **f(x)** se puede proporcionar en la llamada a fsolve:

```
-->fsolve(x0,fun,jacob)
```

donde:

- **jacob** es el nombre de la función Scilab que calcula la matriz jacobiana de f(x) .
  Argumentos: **[dfx]=jacob(x)**

EJEMPLO

Calcular la (única) solución del siguiente sistema de ecuaciones en el primer cuadrante:

$$x = (x_1, x_2), \quad f(x) = \begin{cases} x_1^3 + x_2^3 - 3 \\ x_1^2 + x_2^2 - 2x_2 \end{cases}, \quad \frac{df}{dx}(x) = \begin{pmatrix} 3x_1^2 & 3x_2^2 \\ 2x_1 & 2x_2 - 2 \end{pmatrix}$$

```
deff('[y]=g(x)','y=[x(1)^3+x(2)^3-3,x(1)^2+x(2)^2-2*x(2)]');
deff('[dg]=derg(x)','dg=[3*x(1)^2,3*x(2)^2;2*x(1),2*x(2)-2]')
x=fsolve([1,1],g,derg)
x  =
   0.9587068    1.2843962
```

Para otros argumentos opcionales, véase el Help.

Ver ejemplos en script **ejemplo7.sce**

8. CÁLCULO DE INTEGRALES DEFINIDAS

Para calcular la integral definida de una función de una variable Scilab dispone de la función:

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

```
-->intg(a,b,f)
```

donde **f** debe ser una función de la forma **y=f(t)** y **a** y **b** son los límites de integración

EJEMPLO

$$\int_0^{2\pi} \frac{x \operatorname{sen}(30x)}{\sqrt{1-\left(\frac{x}{2\pi}\right)^2}}\,dx$$

```
deff('[y]=f(x)','y=(x*/sin(30*x)) / sqrt(1-(x/(2*%pi))^2)');
intg(0,2*%pi,f)
ans  =
  - 2.5432596
```

Para la descripción de argumentos opcionales y utilización con funciones Fortran o C véase el Help.

Para calcular

$$\int_R f(x,y)\,dx\,dy$$

siendo Ω una región formada por la unión de **N** triángulos se puede calcular con

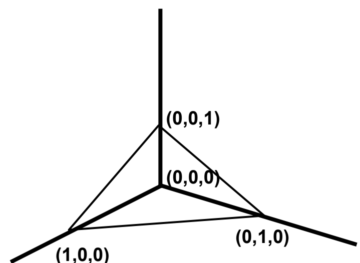
```
-->int2d(X,Y,f)
```

donde **X** e **Y** son matrices **3xN** tales que la j-ésima columna de X (respec. Y) contiene las abscisas (respec. las ordenadas) de los vértices del j-ésimo triángulo y **f** es una función de la forma **v=f(x,y)**

Análogamente,

```
-->int3d(X,Y,Z,f)
```

sirve para calcular la integral de una función de tres variables en una región formada por la unión de tetraedros.
X, **Y**, **Z** son matrices **4xN** que contienen las coordenadas de los vértices de los tetraedros y **f** es una función de la forma **v=f(x,n)** siendo **x** un vector de tres componentes.



EJEMPLO

$$\int_S e^{x^2+y^2+z^2}\,dx\,dy\,dz$$

```
deff('[v]=f(x,nf)','v=exp(sum(x.*x))');
X=[0,1,0,0]';
Y=[0,0,1,0]';
Z=[0,0,0,1]';
int3d(X,Y,Z,f)
ans  =
    0.2278
```

9. RESOLUCIÓN DE ECUACIONES DIFERENCIALES

Para resolver problemas de valor inicial relativos a (sistemas de) ecuaciones diferenciales ordinarias Scilab dispone de la funcion **ode** .

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

El uso más simple es:

```
-->y=ode(y0,t0,t,f)
```

donde:

- **y0** es el valor de la condición inicial y^0
- **t0** es el punto en que se impone la condición inicial t_0
- **t** es el vector que contiene los valores de t para los cuales se quiere calcular la solución
- **f** es el nombre de la función que calcula $f(x,y)$
- **y** es un vector conteniendo los valores de la solución

EJEMPLO: Problema de Cauchy para una ed. diferencial ordinaria

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

RESOLUCIÓN INTERACTIVA

```
function [dydx]=fty(t,y), dydx=0.3*y*(2-y), endfunction
t0=0; y0=0.1; t=linspace(t0,tf);
y=ode(y0,t0,t,fty);
plot2d(t,y)                    // para dibujar la solucion
```

Se podría poner, directamente:

```
plot2d(linspace(0,20),ode(0.1,0,linspace(0,20),fty))
```

Mediante la utilización de otros argumentos opcionales es posible decidir, en la llamada a ode, el método a usar para la resolución del problema así como cotas para el error tolerado. También es posible utilizar una rutina escrita en Fortran o C para calcular la función del segundo miembro, así como pasar a ésta argumentos opcionales extra. Pero queda fuera del ámbito de estos apuntes explicar estas funcionalidades.

EJEMPLO: Problema de Cauchy para una sistema diferencial ordinario

$$\begin{cases} y_1' = 2y_2 - y_1 y_2 \\ y_2' = 0.3 y_1 y_2 - y_2 \\ y_1(0) = 1 \\ y_2(0) = 2 \end{cases} \quad \text{o bien, en notación vectorial,} \quad \begin{cases} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}' = \begin{pmatrix} 2y_2 - y_1 y_2 \\ 0.3 y_1 y_2 - y_2 \end{pmatrix} \\ \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}(0) = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \end{cases}$$

RESOLUCIÓN INTERACTIVA

```
-->function [dydx]=fty(t,y),dydx[(2-y(2))*y(1);(0.3*y(1)-1)*y(2)],endfunction
-->t0=0; y0=[1;2]; tf=10; t=linspace(t0,tf);
-->y=ode(y0,t0,t,fty);
-->plot2d(t',y')                    // para dibujar la solucion
```

Se podría poner, directamente:

```
--> plot2d(linspace(0,10)',ode([1;2],0,linspace(0,10),fty)')
```

Las posibilidades de esta funcion (la más básica para resolver e.d.o.'s) y de otras relacionadas con las ecuaciones y la simulación son imposibles de exponer aquí: hay libros enteros dedicados a ello.

Para algo más de información y algunos ejemplos, véase el Help de esta función y, en general el epígrafe "**Optimization and simulation**". Los ejemplos de estas notas están en el script **ejemplo9.sce**

10. REPRESENTACIONES GRÁFICAS CON SCILAB

Estas notas sólo pretenden exponer algunos de los comandos más básicos de que dispone Scilab para generar gráficos. En principio se exponen, exclusivamente, los comandos de dibujo y posteriormente se explicará cómo modificar los distintos (y numerosos) parámetros que determinan sus características.

9.1 Curvas planas

El comando básico de Scilab para dibujar curvas planas es **plot2d**.

plot2d

Dados dos vectores (VER **(*)**)
x=[x1,x2,. . . ,xn] e y=[y1,y2, . . . ,yn]

```
-->plot2d(x,y)
```

dibuja la curva que pasa por los puntos (x1,y1). . .(xn,yn)

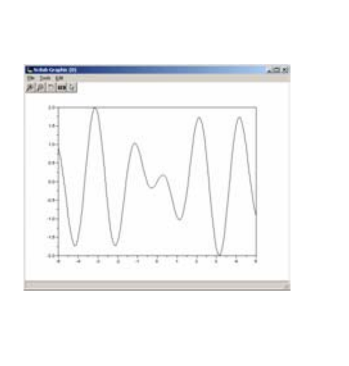
Ejemplo: dibujar, en **[-5,5]**

$$y = 2\,\mathrm{sen}\,(\frac{x}{2})\cos(3x)$$

```
x=linspace(-5,5)';
y=2*sin(x/2).*cos(3*x);    // ( VER (**) )
plot2d(x,y)
```

También se podría poner, directamente:

```
x=linspace(-5,5)';
plot2d(x, 2*sin(x/2).*cos(3*x))
```



(*) Para dibujar una única curva, es indiferente que los vectores **x** e **y** sean filas o columnas. Sin embargo, cuando se desean dibujar varias curvas juntas no lo es. Por ello, usaremos siempre vectores-columna.

(**) Obsérvese que, puesto que se calculan todas las ordenadas "de una sola vez", es preciso "vectorizar" la escritura de la fórmula, para que, al ser el argumento **x** un vector, la fórmula devuelva un vector de las mismas dimensiones calculado elemento e elemento.

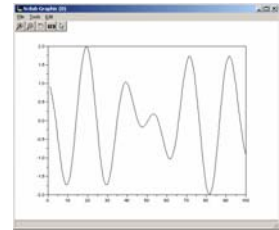
Si se hubiera escrito `y=2*sin(x/2)*cos(3*x)` se hubiera obtenido un error, ya que **x** es un vector columna de longitud 100 (i.e. una matriz 100x1), luego también `sin(x/2)` y `cos(3*x)` son matrices 100x1 y la multiplicación (matricial) `sin(x/2)*cos(3*x)` carece de sentido.

```
-->plot2d(y)
```

dibuja la curva que pasa por los puntos (1,y1). . .(n,yn)

Ejemplo:

```
x=linspace(-5,5)';
y=2*sin(x/2).*cos(3*x);
plot2d(y)
```

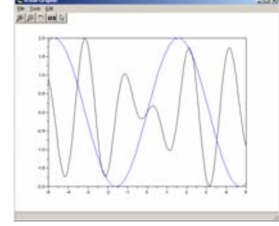


```
-->plot2d(x,y)
```

siendo **x** un vector e **y** una matriz dibuja una curva por cada columna de **y**

Ejemplo:

```
x=linspace(-5,5)';
y=2*sin(x/2).*cos(3*x);
z=2*sin(x);
w=[y,z];
plot2d(x,w)
```



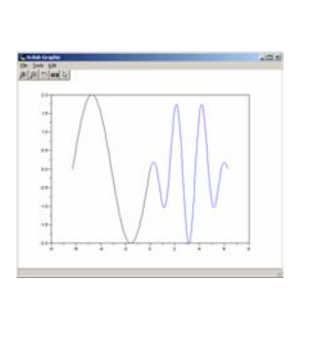
```
-->plot2d(x,y)
```

siendo **x** e **y** matrices de las mismas dimensiones, dibuja una curva por cada par (columna de **x** , columna de **y)**

Ejemplo: dibujar en **[-2 pi , 2 pi]** la gráfica de la función:

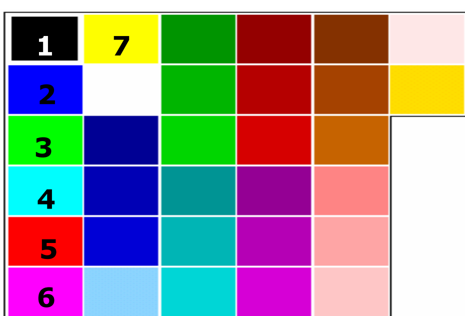
$$y = \begin{cases} 2\,\mathrm{sen}\,(x), & \text{si } x \leq 0 \\ 2\,\mathrm{sen}\,(\frac{x}{2})\cos(3x), & \text{si } x > 0 \end{cases}$$

```
x1=linspace(-2*%pi,0)';
x2=linspace(0,2*%pi)';
y1=2*sin(x1);
y2=2*sin(x2/2).*cos(3*x2);
plot2d([x1,x2],[y1,y2])
```



Observaciones:

- Obsérvese que, por defecto, gráficas sucesivas se superponen. Para evitarlo, hay que borrar la gráfica anterior antes de dibujar de nuevo. Ello puede hacerse ó bien cerrando la ventana gráfica ó bien borrando su contenido desde la barra de menús (**Edit --> Erase Figure**) o mediante un comando (**xbasc()**).
- Cuando se borra el contenido de la ventana gráfica, pero no se cierra, se conservan sus características. Si, por ejemplo, se ha modificado la carta de colores de esa ventana, se seguirá conservando la carta al borrarla, pero no al crerrarla.
- Cuando plot2d dibuja varias curvas, les asigna distintos colores. El orden de los colores asignados viene determinado por la carta de colores activa. Por defecto, es la siguiente:



polarplot

```
-->polarplot(theta,rho)
```

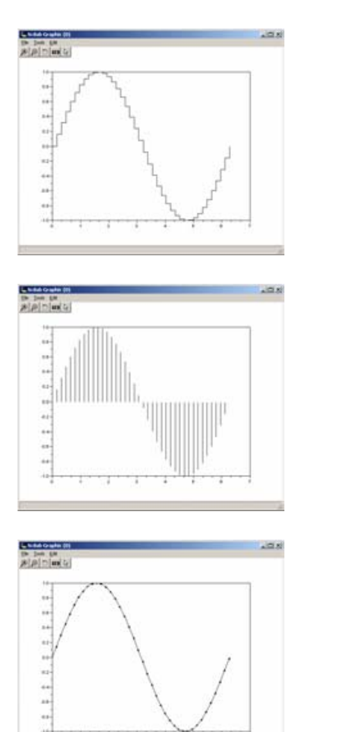
siendo theta y rho vectores de las misma dimensión, dibuja la curva que pasa por los puntos de coordenadas polares
`(theta_1,rho_1). . .(theta_n,rho_n)`

Ejemplo: $\rho = \mathrm{sen}\,(2\theta)\cos(3\theta)$

```
theta=linspace(0,2*%pi)';
rho=sin(2*theta).*cos(3*theta);
polarplot(theta,rho)
```

Esta función se puede utilizar también con otros argumentos, como **plot2d** y su funcionamiento es similar.

plot2d2, plot2d3, plot2d4



```
-->plot2d2(x,y)
```

es similar a plot2d, pero dibuja una función constante a trozos (escalonada)

```
-->plot2d3(x,y)
```

es similar a plot2d, pero dibuja una función utilizando líneas verticales

```
-->plot2d4(x,y)
```

es similar a plot2d, pero dibuja una curva utilizando puntas de flecha (por ejemplo para señalar la dirección de una trayectoria)

fplot2d

```
-->fplot2d(x,f)
```

donde **x** es un vector y **f** es el nombre de una función Scilab, dibuja la curva que pasa por los puntos **(xi,f(xi))**. La función debe responder a la forma **[y]=f(x)**.

Ejemplo:

```
x=linspace(-%pi,%pi);
exec("gf1.sci",-1)
fplot2d(x,gf1)
```

o directamente poniendo

```
fplot2d(-%pi:0.1:%pi,gf1)
```

FICHERO gf1.sci

```
function [y]=gf1(x)
y  = ones(x);
y(x>=0) = cos(x(x>=0));
endfunction
```

Puede resultar interesante observar cómo se "vectoriza" el cálculo de esta función:
1. La orden **y=ones(x)** crea un vector de unos, con las mismas dimensiones que **x** : así **y** será fila o columna (u otra matriz) según lo sea **x**
2. La operación de comparación **x>=0** produce un vector booleano de las mismas dimensiones que **x**. Así, **y(x>=0)** extrae de **y** sólo aquellas componentes tales que la correspondiente componente del vector **x** es mayor o igual que cero.

Otra posibilidad, obviamente, es dar valor a las coponentes de **y** una a una, utilizando un bucle. Pero esta opción llevaría bastante más tiempo de cálculo.

paramfplot2d : animaciones

```
-->paramfplot2d(f,x,t)
```

donde **f** es el nombre de una función Scilab **[y]=f(x,t)**, y **x** y t son vectores, dibuja una animación de la función **f(x,t)**, es decir, dibuja sucesivamente **f(x,t1), f(x,t2), . . . f(x,tn)**

Ejemplo:

```
x=linspace(-2*%pi,2*%pi);
t=linspace(-1,1,60);

paramfplot2d(gfun2,x,t)
```

FICHERO gfun2.sci

```
function [y]=gfun2(x,t)
y=zeros(x);
b=-%pi<=x & x<=%pi;
y(b)=-t*sin(x(b));
endfunction
```

9.2 Curvas en el espacio

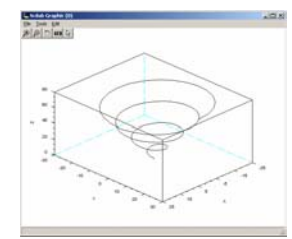
param3d

```
-->param3d(x,y,z)
```

donde **x** , **y** , **z** son tres vectores de la misma dimensión, dibuja la curva que pasa por los puntos **(xi,yi,zi)**

Ejemplo:

```
t=linspace(0,8*%pi);
param3d(t.*sin(t),t.*cos(t),3*t)
```



param3d1

```
-->param3d1(x,y,z)
```

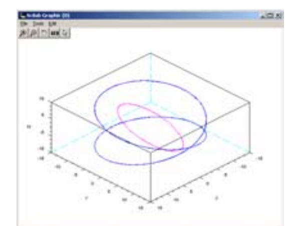
es similar a param3d pero permite dibujar varias curvas. Aquí, **x** , **y** , **z** son tres matrices de la misma dimensión cuyas columnas determinan las distintas curvas.

```
-->param3d1(x,y,list(z,colors))
```

donde **colors** en un vector de números enteros, dibuja la i-ésima curva del color definido por el i-ésimo elemento de **colors**

Ejemplo:

```
t= linspace(0,5*%pi)';
x= [zeros(t),12*sin(2*t)];
y= [10*cos(t),12*cos(2*t)];
z= [10*sin(t),12*sin(t)];
param3d1(x,y,list(z,[6,2]))
```



9.3 Superficies

plot3d, fplot3d, plot3d1, fplot3d1

```
-->plot3d(x,y,z)
```

dibuja la superficie definida por la función **z=f(x,y)**.

x : vector de dimensión **n**
y : vector de dimensión **m**
z : matriz de dimensión **nxm**

x e y contienen las coordenadas de los puntos de la malla rectangular sobre la que se dibuja la función
z contiene los valores de la función en los nodos: **z(i,j)=f(x(i), y(j))**

Para construir la matriz z a partir de los vectores x e y puede ser util la función

```
-->[xm,ym]=ndgrid(x,y)
```

que construye dos matrices xm e ym de dimensión nxm tales que el nodo (i,j) de la malla tiene las coordenadas (xm(i,j),ym(i,j)). Con ayuda de estas matrices los valores de la función f(x,y) pueden calcularse mediante:

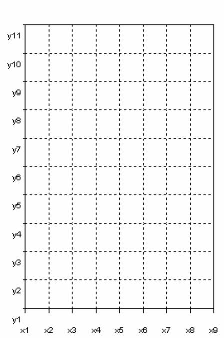
```
-->z=f(xm,ym)
```

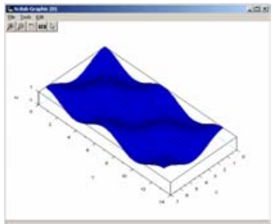
(siempre que la función f esté "vectorizada", claro está).

Ejemplo:

```
x=linspace(0,2*%pi,30);
y=linspace(0,4*%pi,50);
[xm,ym]=ndgrid(x,y);
z=cos(xm).*cos(ym);

plot3d(x,y,z)
```





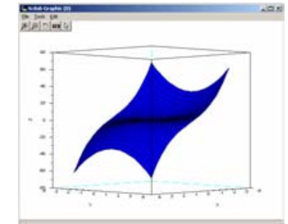
```
-->fplot3d(x,y,f)
```

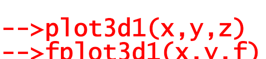
donde **x** e **y** son vectores y **f** es una función, es similar a **plot3d** pero el argumento de entrada es el nombre de la función en lugar de los valores de **z**

Ejemplo:

```
function [z]=f(x,y), z=x.*(-x.^2-y.^2), endfunction
x=linspace(-%pi,%pi,30);

fplot3d(x,x,f)
```



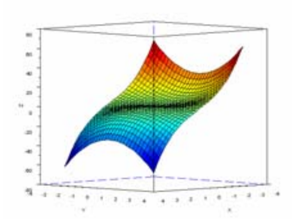


```
-->plot3d1(x,y,z)
-->fplot3d1(x,y,f)
```

Son similares a las anteriores pero producen una superficie formada por facetas coloreadas en función de la altura (valor de **z**)

Ejemplo:

```
function [z]=f(x,y), z=x.*(-x.^2-y.^2), endfunction
x=linspace(-%pi,%pi,30);
fplot3d1(x,x,f)
// La carta de colores por defecto no es "degradada"
// modificación de la carta de colores de la figura
//
fig=gcf();      // apuntador a la figura actual
fig.color_map=jetcolormap(32);
```



grayplot, Sgrayplot, fgrayplot, Sfgrayplot

```
-->grayplot(x,y,z)       // con valores de z
-->fgrayplot(x,y,f)      // con nombre de función
-->Sgrayplot(x,y,z)      // suavizado, con valores
-->Sfgrayplot(x,y,f)     // suavizado, con función
```

representación plana de una función **z=f(x,y)** usando colores.

x : vector de dimensión **n**
y : vector de dimensión **m**
z : matriz de dimensión **nxm** : **z(i,j)=f(x(i),y(j))**
f : función externa del tipo **z=f(x,y)**

Sgrayplot y **Sfgrayplot** hacen lo mismo que las anteriores pero con "suavizado", es decir, el color en cada faceta se interpola, en lugar de ser constante

Ejemplo:

```
function [z]=f(x,y), z=x.^2+y.^2, endfunction
x=linspace(-1,1,30);

fgrayplot(x,x,f)     // produce la primera figura

fig=gcf();
fig.color_map=jetcolormap(30);

Sfgrayplot(x,x,f)    // produce la segunda figura

//
// se añade una "colorbar" para identificar los valores
//
z=f(x,x);
zmin=min(z); zmax=max(z);
colorbar(zmin,zmax);
```

contour, contour2d, contourf

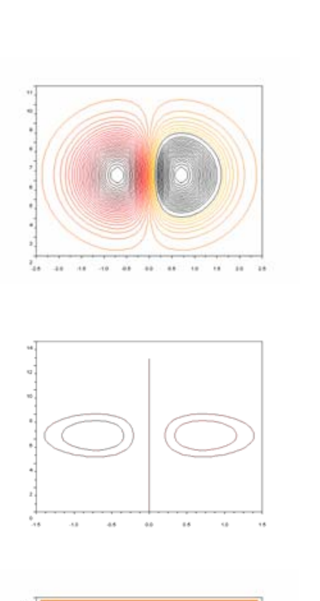
Este grupo de funciones sirve para dibujar curvas de nivel de una superficie

```
-->contour(x,y,z,nz)
-->contour2d(x,y,z,nz)
-->contourf(x,y,z,nz)
```

x : vector de dimensión **n**
y : vector de dimensión **m**
z : matriz de dimensión **nxm** : **z(i,j)=f(x(i),y(j))**
nz : si es un numero entero, se dibujan **nz** curvas de nivel regularmente espaciadas;
si **nz** es un vector numérico se dibujan las curvas de nivel correspondientes a los valores contenidos en **nz**

Ejemplo:

```
function [z]=f(x,y), z=x.*exp(-x.^2-y.^2), endfunction
x=linspace(-%pi,%pi,50);
[xm,ym]=ndgrid(x,x);
zm=f(xm,ym);
contour(x,x,zm,50)                    // primera figura
fig=gcf();
fig.color_map=hotcolormap(50);
//
xbasc()
//
contour(x,x,zm,[-0.3,-0.2,0,0.2,0.3])  // segunda figura
//
xbasc()
//
contourf(x,x,zm,50)                   // tercera figura
colorbar(min(zm),max(zm))
```



champ, champ1

Estas dos funciones sirven para representar un campo de vectores en el plano **XY**

```
-->champ(x,y,fx,fy)     // intensidad = longitud flechas
-->champ1(x,y,fx,fy)    // intensidad = color flechas
```

representación del campo de vectores

$$\vec{v}(x,y) = (fx(x,y), fy(x,y))$$

x : vector de dimensión **n**
y : vector de dimensión **m**
fx : matriz de dimensión **nxm** : primera comp. campo
fy : matriz de dimensión **nxm** : segunda comp. campo

Ejemplo:

$$z = g(x,y) = x\,cos(y), \quad \nabla g(x,y) = \begin{pmatrix} \cos y \\ -x \operatorname{sen} y \end{pmatrix}$$

```
function [z]=f(x,y), z=x.*cos(y), endfunction
//
x=linspace(-%pi/2,%pi/2,30);
[xm,ym]=ndgrid(x,x);
z=f(xm,ym);
plot3d1(x,x,z)                    // primera figura
fig=gcf();fig.color_map=hotcolormap(30);
//
xbasc()
//
gradx=cos(ym);
grady=-xm.*sin(ym);
champ(x,x,gradx,grady)         // segunda figura
//
xbasc()
//
champ1(x,x,gradx,grady)       // tercera figura
//
```