Manual

de

Scilab

1. Operaciones Básicas

1.1. Operaciones con matrices y vectores

Como se comentó en la introducción que hemos visto en el punto anterior, Scilab es un programa creado para trabajar con matrices, por lo tanto, este punto es probablemente el más importante y en el que mejor tenemos que aclararnos para empezar a trabajar. Tenemos muchas opciones para trabajar con ellas, podemos intercambiar matrices, permutarlas, invertirlas; Scilab es una herramienta de cálculo muy potente en lo que a matrices se refiere.

1.1.1. Introducción de matrices desde el teclado

Las matrices y vectores son variables del programa cuyos nombres podemos definir, siempre y cuando no utilicemos los caracteres que el programa tiene como caracteres prohibidos.

Para definir Scilab, se determinan el número de filas y de columnas en función del número de elementos que se proporcionan (o se utilizan). ***Las matrices se definen por filas***; los elementos de una misma fila están separados por ***blancos*** o ***comas***, mientras que las filas están separadas por pulsaciones ***intro*** o por caracteres ***punto y coma*** (;). Tomemos como ejemplo:

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

Cuya salida será:

```
a  =

!   1.0E+00    2.0E+00    1.0E+00 !
!   3.0E+00    4.0E+00    2.0E+00 !
!   5.0E+00    3.0E+00    1.0E+00 !
```

A partir de este momento la matriz **a** está disponible para hacer cualquier tipo de operación con ella (además de valores numéricos, en la definición de una matriz o vector se pueden utilizar expresiones y funciones matemáticas). Por ejemplo, una sencilla operación con **a** es hallar su ***matriz transpuesta***. En Scilab, el apóstrofo (') es el símbolo de *transposición matricial*. Para calcular **a'** (transpuesta de **a**) basta teclear lo siguiente (se añade a continuación la respuesta del programa):

```
-->a'
 ans  =

!   1.    3.    5. !
!   2.    4.    3. !
!   1.    2.    1. !
```

Como el resultado de la operación no ha sido asignado a ninguna otra matriz, Scilab utiliza un nombre de variable por defecto (***ans***, de *answer*), que contiene el resultado de la última operación. La variable ***ans*** puede ser utilizada como operando en la siguiente expresión que se introduzca. También podría haberse asignado el resultado a otra matriz llamada **b**.

Ahora vamos a definir una matriz **b** conjugada para hacer operaciones básicas con estas 2 matrices:

```
-->b=a'
 b  =

!   1.    3.    5. !
!   2.    4.    3. !
!   1.    2.    1. !
```

Comenzamos con las operaciones más básicas que podemos encontrar, la suma y la resta de matrices:

```
-->a+b
 ans  =

!   2.    5.    6. !
!   5.    8.    5. !
!   6.    5.    2. !

-->a-b
 ans  =

!   0.  - 1.  - 4. !
!   1.    0.  - 1. !
!   4.    1.    0. !
```

Si realizamos la multiplicación de matrices con el operando '*****' tendremos que tener cuidado con que el número de columnas de la primera matriz debe coincidir con el número de filas de la segunda:

```
-->a*b
 ans  =

!   6.     13.    12. !
!   13.    29.    29. !
!   12.    29.    35. !
```

También podemos utilizar una multiplicación elemento a elemento, que aunque no tiene demasiado sentido como multiplicación de matrices, si que es muy utilizable en el caso de que la matriz no sea más que un conjunto ordenado de valores.

```
-->a.*b
 ans  =

!   1.    6.    5. !
!   6.    16.   6. !
!   5.    6.    1. !
```

A continuación vamos a definir una nueva matriz **a** a partir de una función que genera valores aleatorios entre 0 y 1.

```
-->rand(2,2)
 ans  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !
```

A partir de esta matriz **a** calculamos su inversa con el comando **inv(a):**

```
-->d=inv(a) //es necesario que la matriz sea cuadrada y que posea
determinante
 d  =

! - 2.     1.   - 4.9E-16 !
!   7.    - 4.     1.     !
! - 11.    7.   - 2.     !
```

Podemos comprobar multiplicando una por la otra que el cálculo es correcto:

```
-->a*d
 ans  =

!   1.    0.    0. !
!   0.    1.    0. !
!   0.    0.    1. !
```

Si los valores no son exactos podemos utilizar el comando **round()** ya que debido a los errores de aproximación en los cálculos podemos encontrar valores como `-4.9E-16` que representa un valor extremadamente pequeño.

```
-->d=round(d)
 d  =

! - 2.     1.    0. !
!   7.   - 4.    1. !
! - 11.    7.  - 2. !
```

Si queremos comentar las líneas de código que ejecutamos, a continuación de la operación podemos poner un comentario anteponiendo el carácter //

```
-->d=inv(a) //es necesario que la matriz sea cuadrada
```

De igual manera que se define una matriz podemos definir un vector:

```
-->x=[10 15 20] //vector fila
 x  =

!   10.    15.    20. !

-->y=[10;15;20]; //vector columna

-->z=[10,15,20]' //vector columna
 z  =

!   10. !
!   15. !
!   20. !
```

Podemos observar, podemos definir vectores fila y vectores columna, con sólo hacer la traspuesta del vector en la definición. También podemos definir un vector columna como si hicieramos una matriz de *1xn*

Como podemos ver, no hemos obtenido resultado tras realizar la operación; esto es debido a que hemos puesto un "**;**" al final de la línea de comando, esto hace que no salga por pantalla lo que hemos ejecutado, cosa que resulta muy útil cuando las matrices/vectores son de un número muy grande (100, 1000, …) y por lo tanto, difíciles de manejar visualmente.

En Scilab se accede a los elementos de un vector poniendo el índice entre paréntesis (por ejemplo *x(3)* ó *x(i)*). Los elementos de las matrices se acceden poniendo los dos índices entre paréntesis, separados por una coma (por ejemplo *A(1,2)* ó *A(i,j)*).

Las matrices ***se almacenan por columnas*** (aunque ***se introduzcan por filas***, como se ha dicho antes), y teniendo en cuenta esto puede accederse a cualquier elemento de una matriz con un sólo subíndice. Por ejemplo, si **A** es una matriz (3x3) se obtiene el mismo valor escribiendo *A(1,2)* que escribiendo *A(4)*.

1.1.2. Operaciones con matrices

Scilab puede operar con matrices por medio de ***operadores*** y por medio de ***funciones***. Se han visto ya los operadores *suma* (+), *producto* (*) y *traspuesta* ('), así como la función *invertir* ***inv( )***. Los operadores matriciales de SCILAB son los siguientes:

+	adición o suma
–	sustracción o resta
*	multiplicación
'	traspuesta
^	potenciación
\	división-izquierda
/	división-derecha
.*	producto elemento a elemento
./ y .\	división elemento a elemento
.^	elevar a una potencia elemento a elemento

Estos operadores se aplican también a las variables o valores escalares, aunque con algunas diferencias. Todos estos operadores son coherentes con las correspondientes operaciones matriciales: *no se puede por ejemplo sumar matrices que no sean del mismo tamaño*. Si los operadores no se usan de modo correcto se obtiene un mensaje de error.
Véase el siguiente ejemplo de tres ecuaciones formadas por una recta que no pasa por el origen y los dos ejes de coordenadas:

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

!   1.    2. !
!   1.    0. !
!   0.    1. !

-->x=A\b, resto=A*x-b
 x  =

!   0.33333 !
!   0.66667 !
 resto  =

! - 0.33333 !
!   0.33333 !
!   0.66667 !
```

Vamos a ver como funcionan una serie de operadores:

```
-->a
 a  =

!   1.    2.    1. !
!   3.    4.    2. !
!   5.    3.    1. !

-->b
 b  =

!   1.    3.    5. !
!   2.    4.    3. !
!   1.    2.    1. !

-->a/b
 ans  =

!   0.    0.     1.  !
! - 2.    7.   - 9.  !
! - 7.    24.  - 36. !

-->a\b
 ans  =

!   0.  - 2.  - 7.  !
!   0.    7.    24. !
!   1.  - 9.  - 36. !
```

Si los operadores « / » y « \ » van precedidos de un “**.**” La operación se realiza elemento a elemento:

```
-->a./b
 ans  =

!   1.     0.66667    0.2      !
!   1.5    1.         0.66667  !
!   5.     1.5        1.       !

-->a.\b
 ans  =

!   1.         1.5        5.   !
!   0.66667    1.         1.5  !
!   0.2        0.66667    1.   !
```

1.1.3. Tipos de datos

Scilab trabaja siempre con el tipo Real de doble precisión, también tenemos la posibilidad de trabajar con *reales de simple precisión* (no hay operaciones implementadas), *enteros* y *booleanos*, pero no tienen buena parte de las operaciones matemáticas aunque sí lógicas. Este tipo de dato se guarda con un tamaño de 8Bytes, que tiene un tamaño de 15 cifras exactas. Además del tipo Real, podremos trabajar con strings, matrices, hipermatrices y estructuras más avanzadas.

1.1.3.1. Números reales de doble precisión

Los elementos constitutivos de vectores y matrices son números reales almacenados en 8 bytes (53 bytes para la mantisa y 11 para el exponente de 2; entre 15 y 16 cifras decimales equivalentes). Es importante saber cómo trabaja Scilab con estos números y los casos especiales que presentan. Scilab mantiene una forma especial para los *números muy grandes* (más grandes que los que es capaz de representar), que son considerados como ***infinito***. Por ejemplo, obsérvese cómo responde el programa al ejecutar el siguiente comando:

```
-->%inf
 %inf  =

   Inf
```

Así pues, para Scilab el ***infinito*** se representa como %***inf***. Scilab tiene también una representación especial para los resultados que no están definidos como números, que se representa como **Not a Number** (**%nan**):

```
-->%nan
 ans  =

   Nan
```

Los ***NaN*** se propagan al realizar con ellos cualquier operación aritmética, en el sentido de que, por ejemplo, cualquier número sumado a un ***NaN*** da otro ***NaN***. Scilab tiene esto en cuenta. Algo parecido sucede con los ***Inf***.

```
-->%nan*1
 ans  =

   Nan

-->%nan*%nan
 ans  =

   Nan

-->%nan*%inf
 ans  =

   Nan
```

Podemos encontrar 3 variables predefinidas por Scilab que nos dan los valores máximos y mínimos de este tipo de datos:

- ***eps*** devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. Da una idea de la precisión o número de cifras almacenadas. En un PC, ***eps*** vale 2.2204e-016.
- ***realmin*** devuelve el número más pequeño con que se puede trabajar (2.2251e-308)
- ***realmax*** devuelve el número más grande con que se puede trabajar (1.7977e+308)

1.1.3.2. Números complejos (complex)

Muchas veces nos vamos a encontrar que el cálculo que necesitamos ejecutar nos lleva a tener que definir el cuerpo de los números complejos, dado que el cuerpo de los números reales no es suficiente, por ejemplo, para realizar transformadas de Fourier o Laplace. Para ello se define la variable compleja %**i**:

```
-->%i
 %i  =

    i
```

Como podemos ver, la definición se hace con el % como en todas las constantes prefijadas por el sistema , y que es necesario el uso del operador **"*"** para multiplicarlo por un escalar.

```
-->// j no se considera como variable imaginaria

-->a=[1+%i,2+3*%i;%i,%i + 4]
 a  =

!   1. + i      2. + 3.i !
!   i           4. + i   !

-->b=[1,2;0,4]+[1 3;1 1]*%i
 b  =

!   1. + i      2. + 3.i !
!   i           4. + i   !
```

Es importante advertir que el *operador de matriz transpuesta* (’), aplicado a matrices complejas, produce la matriz transpuesta conjugada. El operador punto y apóstrofo (**.’**) que calcula simplemente la matriz transpuesta.

```
-->a'
 ans  =

!   1. - i    - i        !
!   2. - 3.i    4. - i   !
-->a.'
 ans  =

!   1. + i      i        !
!   2. + 3.i    4. + i   !
```

1.1.3.3. Cadenas de caracteres

Para crear una cadena de caracteres (string) en Scilab podemos hacerlo de estos dos modos:

```
-->a='cadena de caracteres'
 a  =

 cadena de caracteres
-->a="cadena de caracteres"
 a  =

 cadena de caracteres
```

Debido a que para su uso es necesario un conocimiento previo de las funciones orientadas a matrices, postpondré otras explicaciones hasta que se hayan explicado estas, al igual que con los **strings**, se postpone la explicación de **hipermatrices, structs** y **cell arrays**.

Ya han aparecido algunos ejemplos de ***variables*** y ***expresiones*** matriciales. Ahora se va a tratar de generalizar un poco lo visto hasta ahora. Una ***variable*** es un nombre que se da a una entidad numérica, que puede ser una matriz, un vector o un escalar. El valor de esa variable, e incluso el tipo de entidad numérica que representa, puede cambiar a lo largo de una sesión de SCILAB o a lo largo de la ejecución de un programa. La forma más normal de cambiar el valor de una variable es colocándola a la izquierda del ***operador de asignación*** (=).

Una expresión de SCILAB puede tener las dos formas siguientes: primero, asignando su resultado a una variable,

```
variable = expresión
```

y segundo evaluando simplemente el resultado del siguiente modo, `expresión` en cuyo caso el resultado se asigna automáticamente a una variable interna de SCILAB llamada ***ans*** (de *answer*) que almacena el último resultado obtenido. Se considera por

defecto que una expresión termina cuando se pulsa ***intro***. Si se desea que una expresión continúe en la línea siguiente, hay que introducir ***tres puntos*** (...) antes de pulsar ***intro***. También se pueden incluir varias expresiones en una misma línea separándolas por *comas* (,) o *puntos y comas* (;). Si una expresión ***termina en punto y coma*** (;) su resultado se calcula, pero no se escribe en pantalla. Esta posibilidad es muy interesante, tanto para evitar la escritura de resultados intermedios, como para evitar la impresión de grandes cantidades de números cuando se trabaja con matrices de gran tamaño.

A semejanza de **Fortran**, ***SCILAB distingue entre mayúsculas y minúsculas*** en los nombres de variables; además no hace falta declarar las variables que se vayan a utilizar. Esto hace que se deba tener especial cuidado con no utilizar nombres erróneos en las variables, porque no se recibirá ningún aviso del ordenador. Cuando se quiere tener una *relación de las variables* que se han utilizado en una sesión de trabajo se puede utilizar el comando ***who***. Existe otro comando llamado ***whos*** que proporciona además información sobre el tamaño, la cantidad de memoria ocupada y el carácter real o complejo de cada variable.

El comando ***clear*** tiene varias formas posibles:

- **clear** sin argumentos, ***clear*** elimina todas las variables creadas previamente (excepto las variables globales).
- **clear A, b** borra las variables indicadas.

1.1.3.4. Otras formas de definir matrices

SCILAB dispone de varias formas de definir matrices. El introducirlas por teclado sólo es práctico en casos de pequeño tamaño y cuando no hay que repetir esa operación muchas veces. Recuérdese que en SCILAB no hace falta definir el tamaño de una matriz. Las matrices toman tamaño al ser definidas y este tamaño puede ser modificado por el usuario mediante adición y/o borrado de filas y columnas. A continuación se van a ver otras formas más potentes y generales de definir y/o modificar matrices.

1.1.3.4.1 Tipos de matrices predefinidos

Existen en SCILAB varias funciones orientadas a definir con gran facilidad matrices de tipos particulares. Algunas de estas funciones son las siguientes:

- eye(2,2) forma la matriz unidad de tamaño (2x2)

```
-->eye(2,2)
 ans  =

!   1.    0. !
!   0.    1. !
```

- zeros(3,5) forma una matriz de *ceros* de tamaño (3x5)

```
-->zeros(3,5)
 ans  =

!   0.    0.    0.    0.    0. !
!   0.    0.    0.    0.    0. !
!   0.    0.    0.    0.    0. !
```

- ones(3) forma una matriz de *unos* de tamaño (3x3)

- ones(3,4) idem de tamaño (3x4)

```
-->ones(3,4)
 ans  =

!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
```

- linspace(x1,x2,n) genera un vector con **n** valores igualmente espaciados entre **x1** y **x2**

```
-->linspace(1,2,10)
 ans  =


         column 1 to 5

!   1.    1.1111111    1.2222222    1.3333333    1.4444444 !

         column  6 to 10

!   1.5555556    1.6666667    1.7777778    1.8888889    2. !

--> //es semejante a la definicion 1:1.125:10
```

- logspace(d1,d2,n) genera un vector con **n** valores espaciados logarítmicamente entre 10^d1 y 10^d2. Si d2 es **pi**, los puntos se generan entre 10^d1 y **pi.**
- rand(3,3) forma una matriz de números aleatorios entre 0 y 1, podemos dar diferentes posibilidades para cambiar la distribución estadística de la que partirá.
- Poly(a,x,"flag") : construye un polinomio que puede ser el conjunto de coeficientes de a, si por **flag** introducimos " coeff" y las raices del polinomio definido por a si introducimos "roots".

```
-->poly([1 2 1],'s',"roots")
 ans  =


               2   3
  - 2 + 5s - 4s + s
-->poly([1 2 1],'s',"coeff")
 ans  =


              2
   1 + 2s + s
```

1.1.3.4.2 Formación de una matriz a partir de otras

Scilab ofrece también la posibilidad de crear una matriz a partir de matrices previas ya definidas, por varios posibles caminos:

– recibiendo alguna de sus propiedades (como por ejemplo el tamaño),

– por composición de varias submatrices más pequeñas,

– modificándola de alguna forma.

A continuación se describen algunas de las funciones que crean una nueva matriz a partir de otra o de otras, comenzando por dos funciones auxiliares:

- [m,n]=size(A) devuelve el número de filas y de columnas de la matriz **A**. Si la matriz es cuadrada basta recoger el primer valor de retorno

```
-->a=[1 2 1;3 2 5;6 7 5];

-->[m,n]=size(a)

-->m = 3
 m  =

    3.
```

```
-->n = 3
 n  =

    3.
```

- n=length(x) calcula el número de elementos de un vector **x**

```
-->x=1:0.00001:2;

-->n=length(x)
 n  =

    100001.
```

- zeros(size(A)) forma una matriz de *ceros* del mismo tamaño que una matriz **A** previamente creada.

```
-->size(a)
 ans  =

!   3.    3. !

-->zeros(size(a))
 ans  =

!   0.    0. !
```

ones(size(A)) ídem con *unos*

- A=diag(x) forma una matriz diagonal **A** cuyos elementos diagonales son los elementos de un vector ya existente **x.**
- x=diag(A) forma un vector **x** a partir de los elementos de la diagonal de una matriz ya existente **A.**

```
-->diag(a)
 ans  =

!   1. !
!   2. !
!   5. !
```

- diag(diag(A)) crea una matriz diagonal a partir de la diagonal de la matriz **A.**

```
-->diag(diag(a))
 ans  =

!   1.    0.    0. !
!   0.    2.    0. !
!   0.    0.    5. !
```

- triu(A) forma una matriz triangular superior a partir de una matriz **A** (no tiene por qué ser cuadrada).

```
-->s=poly(0,'s');

-->triu([s,s;s,1])
 ans  =

!   s     s  !
!            !
!   0     1  !

-->triu([1/s,1/s;1/s,1])
 ans  =

!   1     1  !
!   -     -  !
!   z     z  !
!            !
!   0     1  !
!   -     -  !
!   1     1  !
```

- tril(A) ídem con una matriz triangular inferior.

Un caso especialmente interesante es el de crear una nueva matriz ***componiendo como submatrices*** otras matrices definidas previamente. A modo de ejemplo, vamos a realizar la matriz generadora de un código ortogonal con M=2:

```
-->a=[%f %f;%f %t];

-->a
 a  =

! F F !
! F T !
-->b=~a
 b  =

! T T !
! T F !
-->h2=[a a;a b]
 h2  =

! F F F F !
! F T F T !
! F F T T !
! F T T F !
```

1.1.3.4.3 Direccionamiento de vectores y matrices a partir de vectores

Los elementos de una matriz **a** pueden direccionarse a partir de los elementos de vectores:

```
-->a=rand(3,4)
 a  =

!   0.2113249    0.3303271    0.8497452    0.0683740 !
!   0.7560439    0.6653811    0.6857310    0.5608486 !
!   0.0002211    0.6283918    0.8782165    0.6623569 !

-->b=[1 3 6 11]
 b  =

!   1.    3.    6.    11. !

-->a(b)
 ans  =

!   0.2113249 !
!   0.0002211 !
!   0.6283918 !
!   0.5608486 !
```

Podemos ver que hemos obtenido las posiciones 1, 3, 6 y 11 de la matriz a, que debemos contar teniendo en cuenta que la matriz se recorre por columnas y no por filas.

Si queremos ver un elemento concreto de la matriz, podemos ejecutar lo siguiente:

```
-->a(3,2)
 ans  =

    0.6283918

-->a(6)
 ans  =

    0.6283918
```

Creamos esta nueva matriz **a** para que sea más fácil seguir los valores:

```
-->a=rand(4,4)*10
 a  =

!   7.2635068    2.3122372    3.0760907    3.616361  !
!   1.9851438    2.1646326    9.3296162    2.9222666 !
!   5.4425732    8.8338878    2.1460079    5.6642488 !
!   2.3207479    6.5251349    3.12642      4.826472  !
```

Ahora podemos ver las columnas 1, 2 y 3 de la 4ª fila

```
-->a(4,1:3)
 ans  =

!   2.3207479    6.5251349    3.12642 !
```

Calculamos la tercera fila

```
-->a(3,:)
 ans  =

!   5.4425732    8.8338878    2.1460079    5.6642488 !
```

Calculamos la tercera columna

```
-->a(:,3)
 ans  =

!   3.0760907 !
!   9.3296162 !
!   2.1460079 !
!   3.12642   !
```

1.1.3.4.4 Operador <<Dos Puntos>> (:)

Se trata de una de las formas de definir vectores y matrices más usado y más fácil de utilizar, dada la rápida visualización de la salida sin necesidad de ver el resultado:

```
-->x=1:1:10
 x  =

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

En cierta forma se podría decir que el operador (:) representa un ***rango***: en este caso, los números enteros entre el 1 y el 10. Por defecto el incremento es 1, pero este operador puede también utilizarse con otros valores enteros y reales, positivos o negativos. En este caso el incremento va entre el valor inferior y el superior, pero podemos hacer que el incremento sea negativo, o que se haga con un incremento mayor o menor:

```
-->x=x(5:-1:1)
 x  =

!   6.    7.    8.    9.    10. !
```

1.1.3.4.5 Definición de matrices y vectores desde fichero

Scilab acepta el uso de scripts desde los que crear matrices, vectores, variables, etc; como si estuviéramos ejecutándolo desde la propia línea de comandos. Por ejemplo, si creamos el fichero **matriz_a.m** donde creamos una matriz a cualquiera, al ejecutarla en Scilab podremos ver que se crea como si estuvieramos generándola en el propio programa:

```
      //primer script
      a=randn(4)*10
      //fin del script
-->exec("/home/jose/eval_scilab/matriz")
a =

  -22.94025    6.55306   -2.80857   -1.33060
    5.12081    8.98726    1.81236    3.29938
   -0.69713    1.40557   25.56103    5.62650
  -17.50361   -0.63861    4.61985    1.21845
```

1.1.3.5. Operadores Relacionales

El lenguaje de programación de Scilab dispone de los siguientes operadores relacionales:

- < menor que
- > mayor que
- <= menor o igual que
- >= mayor o igual que
- == igual que
- ~= distinto que

Obsérvese que, salvo el último de ellos, coinciden con los correspondientes operadores relacionales de C. Sin embargo, ésta es una coincidencia más bien formal. En Scilab los operadores relacionales pueden aplicarse a vectores y matrices, y eso hace que

tengan un significado especial. Al igual que en C, si una comparación se cumple el resultado es *T* (***true***), mientras que si no se cumple es *F* (***false***). La diferencia con C está en que cuando los operadores relacionales de Scilab se aplican a dos matrices o vectores del mismo tamaño, *la comparación se realiza elemento a elemento*, y el resultado es otra matriz de unos y ceros del mismo tamaño, que recoge el resultado de cada comparación entre elementos.

```
-->a=1;b=2;

-->a<b
 ans  =

  T

-->a~=b
 ans  =

  T

-->a>=b
 ans  =

  F
```

1.1.3.6. Operadores Lógicos

Los operadores lógicos de Scilab son los siguientes:

- & and
- | or
- ~ negación lógica

Obsérvese que estos operadores lógicos tienen distinta notación que los correspondientes operadores de C (&&, || y !). Los operadores lógicos se combinan con los relacionales para poder comprobar el cumplimiento de condiciones múltiples.

```
-->a=%t;b=%f;

-->c=a&b
 c  =

  F

-->d=a|b
 d  =

  T
```

1.2. Funciones de Librería

Scilab posee un gran número de funciones integradas y de funciones definidas por el usuario, las primeras son funciones optimizadas para Scilab, las segundas, con extensión *.m son funciones definidas en ficheros, que pueden ser:

- Definidas por Scilab
- Definidas por grupos/usuarios desinteresados que ofrecen su código a los demás usuarios de Scilab
- Definidas por el propio usuario, para su uso y/o compartición con otros usuarios.

1.2.1 Características Generales de las funciones de librería

El concepto de función en SCILAB es semejante al de C y al de otros lenguajes de programación, aunque con algunas diferencias importantes. Al igual que en C, una función tiene ***nombre***, ***valor de retorno*** y ***argumentos***. Una función *se llama* utilizando su nombre en una expresión o utilizándolo como un comando más.

```
//funcion de prueba para evaluar

function y=prueba(x)

  y=x+3;

endfunction

-->getf("prueba.sci")

-->prueba(1)
 ans  =

    4.
```

Podemos ver que esta función es solamente la función de una recta, cuya pendiente es de 45º y desplazada 3 unidades.

y : es el valor de retorno.

prueba : es el nombre de la función.

x : es el argumento de entrada.

Una característica de SCILAB es que las funciones que no tienen argumentos no llevan paréntesis, por lo que a simple vista no siempre son fáciles de distinguir de las simples variables. La manera de cargar una función es mediante el comando **getf** ,este comando tiene como entrada el nombre de la función, con la dirección que posee en el sistema de archivos. Ejemplo:

```
//function sin argumentos de entrada

function hello

printf("hello, world \n")

endfunction
```

-->getf("f:\matriz.sci")
-->hello()
hello, world

Otra forma de definir una función es hacerlo en la propia línea de comandos del programa, de manera que podemos definir rápidamente una función que puntualmente vamos a utilizar es con el comando **deff**:

```
-->deff('[x]=mifuncion(y,z)','x=y+z')
-->a=mifuncion(23,43)
 a  =
   66.
```

De este modo, hemos definido la función $x = y + z$ y podemos utilizarla mientras no cerremos el programa (**Nota:** las funciones pueden ser redefinidas, así que debemos tener cuidado con el nombre que le ponemos a la función definida con **deff**).

Podemos encontrar gran variedad de tipos de función según lo que resuelvan:

1.- Funciones matemáticas elementales.

2.- Funciones especiales.

3.- Funciones matriciales elementales.

4.- Funciones matriciales específicas.

5.- Funciones para la descomposición y/o factorización de matrices.

6.- Funciones para análisis estadístico de datos.

7.- Funciones para análisis de polinomios.

8.- Funciones para integración de ecuaciones diferenciales ordinarias.

9.- Resolución de ecuaciones no-lineales y optimización.

10.- Integración numérica.

11.- Funciones para procesamiento de señal.

Las características principales de estas funciones son:

- Los *argumentos actuales* de estas funciones pueden ser expresiones y también llamadas a otra función.
- Admite valores de retorno matriciales múltiples. Por ejemplo, en el comando:

```
-->a=[1 1; 2 3];

-->[autovectores,autovalores]=spec(a)
 autovalores  =

!   3.7320508    0          !
!   0            0.2679492  !
 autovectores  =

!   0.3437238  - 0.8068982  !
!   0.9390708    0.5906905  !
```

la función ***spec()*** calcula los valores y vectores propios de la matriz cuadrada **A**. Los vectores propios se devuelven como columnas de la matriz **autovectores**, mientras que los valores propios son los elementos de la matriz diagonal **autovalores**.

- Las operaciones de suma y/o resta de una matriz con un escalar consisten en sumar y/o restar el escalar a todos los elementos de la matriz.
- Recuérdese que tecleando ***help nombre_funcion*** se obtiene de inmediato información sobre la función de ese nombre.

1.2.2. Funciones matemáticas elementales que operan de modo escalar

Estas funciones, que comprenden las funciones matemáticas trascendentales y otras funciones básicas, actúan sobre cada elemento de la matriz como si se tratase de un escalar. Se aplican de la misma forma a escalares, vectores y matrices. Algunas de las funciones de este grupo son las siguientes:

- sin(x) : seno
- cos(x) : coseno
- tan(x) : tangente
- asin(x) : arco seno
- acos(x) : arco coseno
- atan(x) : arco tangente (devuelve un ángulo entre $-\pi/2$ y $+\pi/2$)
- sinh(x) : seno hiperbólico
- cosh(x) : coseno hiperbólico
- tanh(x) : tangente hiperbólica
- asinh(x) : arco seno hiperbólico
- acosh(x) : arco coseno hiperbólico
- atanh(x) : arco tangente hiperbólica
- log(x) : logaritmo natural
- log10(x) : logaritmo decimal
- exp(x) : función exponencial
- sqrt(x) : raíz cuadrada
- round(x) : redondeo hacia el entero más próximo
- fix(x) : redondea hacia el entero más próximo a 0
- floor(x) : valor entero más próximo hacia $-\infty$
- ceil(x) : valor entero más próximo hacia $+\infty$
- gcd(x) : máximo común divisor

- lcm(x) : mínimo común múltiplo
- real(x) : partes reales
- imag(x) : partes imaginarias
- abs(x) : valores absolutos

1.2.3. Funciones que actúan sobre matrices

Las siguientes funciones exigen que el/los argumento/s sean matrices. En este grupo aparecen algunas de las funciones más útiles y potentes de SCILAB. Se clasificarán en varios subgrupos.

1.2.3.1 Funciones elementales

- B = A' calcula la traspuesta (conjugada) de la matriz **A**

```
a  =

!   1. + i      2. + 2.i    1. + 4.i !
!   2. + 2.i    3. + 3.i    4. + 3.i !
!   4. + i      3. + 4.i    1. + i   !

-->b=a'
 b  =

!   1. - i      2. - 2.i    4. - i   !
!   2. - 2.i    3. - 3.i    3. - 4.i !
!   1. - 4.i    4. - 3.i    1. - i   !
```

- B = A.' calcula la traspuesta (sin conjugar) de la matriz **A**

```
-->c=a.'
 c  =

!   1. + i      2. + 2.i    4. + i   !
!   2. + 2.i    3. + 3.i    3. + 4.i !
!   1. + 4.i    4. + 3.i    1. + i   !
```

- v = poly(A) devuelve un vector **v** con los coeficientes del polinomio característico de la matriz cuadrada **A**

```
-->x=[1 2 3];
-->v=poly(x,'v')
 v  =


               2   3
  - 6 + 11v - 6v + v
-->v=poly(a,'v')
 v  =

real part


                      2   3
  - 14 - 1.199D-14v - 5v + v
imaginary part


                 2
    14 - 36v - 5v
```

- t = trace(A) devuelve la traza **t** (suma de los elementos de la diagonal) de una matriz cuadrada **A**

```
-->t=trace(c)
 t  =

    5. + 5.i
```

- [m,n] = size(A) devuelve el número de filas **m** y de columnas **n** de una matriz rectangular **A**

```
-->[m,n]=size(c)
 n  =

    3.
 m  =

    3.
```

- n = size(A) devuelve el tamaño de una matriz cuadrada **A**

```
-->n=size(c)
 n  =

!   3.    3. !
```

1.2.3.2 Funciones Especiales

Las funciones ***exp()***, ***sqrt()*** y ***log()*** se aplican elemento a elemento a las matrices y/o vectores que se les pasan como argumentos. Existen otras funciones similares que tienen también sentido cuando se aplican a una matriz como una única entidad. Estas funciones son las siguientes (se distinguen porque llevan una "m" adicional en el nombre):

- expm(A) : si A=XDX', expm(A) = X*diag(exp(diag(D)))*X'
- sqrtm(A) : devuelve una matriz que multiplicada por sí misma da la matriz **A**
- logm(A) : es la función inversa de expm(A)

```
-->logm(c)
 ans  =


         column 1 to 2

!   0.5181834 - 0.0850081i    0.2659114 - 0.7788088i !
!   1.6086851 + 0.6476739i    1.0165639 + 0.2625855i !
! - 0.6899399 + 0.3500422i    1.1527666 + 1.1291921i !

         column 3

!   1.2775146 + 1.1386837i !
!   0.1526474 + 0.5005469i !
!   1.4508836 - 0.9629756i !
```

1.2.3.3 Funciones de Factorización y/o Descomposición Matricial

A su vez este grupo de funciones se puede subdividir en 4 subgrupos:

– Funciones basadas en la factorización triangular (eliminación de Gauss):

- [L,U] = lu(A) descomposición Gaussiana (A = LU) de una matriz. La matriz **L** es una permutación de una matriz triangular inferior (dicha permutación es consecuencia del pivotamiento por columnas utilizado en la factorización).

```
-->a=rand(3,3)
 a  =

!   0.7560439    0.6653811    0.6857310 !
!   0.0002211    0.6283918    0.8782165 !
!   0.3303271    0.8497452    0.0683740 !

-->[l,u]=lu(a)
 u  =

!   0.7560439    0.6653811    0.6857310 !
!   0.           0.6281972    0.8780159 !
!   0.           0.         - 1.0125751 !
 l  =

!   1.           0.           0. !
!   0.0002925    1.           0. !
!   0.4369153    0.8898959    1. !
```

- B = inv(A) calcula la inversa de **A**. Equivale a B=inv(U)*inv(L)

```
-->b=inv(a)
 b  =

!   1.4624031  - 1.1170364  - 0.3190590 !
! - 0.6031884    0.3635175    1.3803181 !
!   0.4312322    0.8788444  - 0.9875811 !
```

- d = det(A) devuelve el determinante **d** de la matriz cuadrada **A**. Equivale a d=det(L)*det(U)

```
-->d=det(a)
 d  =

 - 0.4809171
-->e=det(l),f=det(u),g=e*f
 e  =

   1.
 f  =

 - 0.4809171
 g  =

 - 0.4809171
```

- E = rref(A) reducción a forma de escalón (mediante la eliminación de Gauss con pivotamiento por columnas) de una matriz rectangular **A.** Sólo se utiliza la diagonal y la parte triangular superior de **A**. El resultado es una matriz triangular superior tal que A = U'*U.
- c = rcond(A) devuelve una estimación del recíproco de la condición numérica de la matriz **A** basada en la norma sub-1. Si el resultado es próximo a 1 la matriz **A** está bien condicionada; si es próximo a 0 no lo está.
  - Funciones basadas en el cálculo de valores y vectores propios:
- [X,D] = spec(A) valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de una matriz cuadrada **A**. Con frecuencia el resultado es complejo (si **A** no es simétrica).
- [X,D] = spec(A,B) valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de dos matrices cuadradas **A** y **B** (**Ax** = λ**Bx**).

– Funciones basadas en la descomposición QR:

- [Q,R] = qr(A) descomposición QR de una matriz rectangular. Se utiliza para sistemas con más ecuaciones que incógnitas.

```
-->a=[1 1 1 1;1 1 2 4; 1 2 1 2; 1 2 2 3];

-->[q,r]=qr(a)
 r  =

! - 2.  - 3.  - 3.         - 5.        !
!   0.    1.    1.110D-16    8.882D-16 !
!   0.    0.  - 1.         - 2.        !
!   0.    0.    0.         - 1.        !
 q  =

! - 0.5  - 0.5    0.5    0.5 !
! - 0.5  - 0.5  - 0.5  - 0.5 !
! - 0.5    0.5    0.5  - 0.5 !
! - 0.5    0.5  - 0.5    0.5 !
```

- B = null(A) devuelve una base ortonormal del subespacio nulo (kernel, o conjunto de vectores **x** tales que **Ax** = 0) de la matriz rectangular **A.**
- Q = orth(A) las columnas de **Q** son una base ortonormal del espacio vectorial de columnas de A. El número de columnas de **Q** es el rango de **A .**

```
-->q=orth(a)
 q  =

! - 0.2603607    0.3653579  - 0.8684147    0.2111441 !
! - 0.6376881  - 0.7063492  - 0.0316730    0.3056493 !
! - 0.4226092    0.5756938    0.490361     0.4995239 !
! - 0.5890395    0.1901606    0.0663240  - 0.7826062 !
```

– Funciones basadas en la descomposición de valor singular

- B = pinv(A) calcula la pseudo-inversa de una matriz rectangular **A**

```
-->a=[1 1;2 2];
```

```
-->b=pinv(a)
 b  =

!   0.1    0.2 !
!   0.1    0.2 !
```

- r = rank(A) calcula el rango **r** de una matriz rectangular **A**
- nor = norm(A) calcula la norma sub-2 de una matriz (el mayor valor singular)
- nor = norm(A,2) lo mismo que la anterior, podemos hacer la norma sub-i de la matriz o vector

– Cálculo del rango, normas y condición numérica:

Existen varias formas de realizar estos cálculos, con distintos niveles de esfuerzo de cálculo y de precisión en el resultado. El rango se calcula implícitamente (sin que el usuario lo pida) al ejecutar las funciones ***rref(A)***, ***orth(A)***, ***null(A)*** y ***pinv(A)***.

Normas de vectores:

- norm(x,p) norma sub-p, es decir ***sum(abs(x)^p)^(1/p)***.
- norm(x) norma euclídea; equivale al módulo o ***norm(x,2)***.
- norm(x,%inf) norma sub-∞, es decir ***max(abs(x))***.
- norm(x,1) norma sub-1, es decir ***sum(abs(x))***.

```
-->x=rand(1,4)
 x  =

!   0.5608486    0.6623569    0.7263507    0.1985144 !

-->norm(x,2)
 ans  =

    1.1490262
```

```
-->norm(x,3)
 ans  =

    0.9502447
```

1.3. Más sobre operadores relacionales con vectores y matrices

Cuando alguno de los operadores relacionales vistos previamente (<, >, <=, >=, == y ~=) actúa entre dos matrices (vectores) del mismo tamaño, el resultado es otra matriz (vector) de ese mismo tamaño conteniendo *T* o *F*, según los resultados de cada comparación entre elementos hayan sido ***true*** o ***false***, respectivamente.

De ordinario, las matrices "binarias" que se obtienen de la aplicación de los operadores relacionales no se almacenan en memoria ni se asignan a variables, sino que se procesan sobre la marcha. SCILAB dispone de varias funciones para ello. Recuérdese que cualquier valor distinto de cero equivale a ***true***, mientras que un valor cero equivale a ***false***. Algunas de estas funciones son:

- mtlb_any(x): es una emulación de la función *any(x)* de MatLab; función vectorial; chequea si *alguno* de los elementos del vector **x** cumple una determinada condición (en este caso ser distinto de cero). Devuelve un uno ó un cero,
- mtlb_all(A) se aplica por separado a cada columna de la matriz **A**. El resultado es un vector de unos y ceros.
- mtlb_find(x) busca índices correspondientes a elementos de vectores que cumplen una determinada condición. El resultado es un vector con los índices de los elementos que cumplen la condición.
- mtlb_find(A) cuando esta función se aplica a una matriz la considera como un vector con una columna detrás de otra, de la 1ª a la última.

1.4. Otras funciones que actúan sobre vectores y matrices

Las siguientes funciones pueden actuar sobre vectores y matrices, y sirven para chequear ciertas condiciones:

- exist(var) comprueba si la variable **var** existe
- isnan() chequea si hay valores ***NaN***, devolviendo una matriz de unos y ceros

```
-->x=[1 2 3 4 %nan %inf 6]
 x  =

!   1.    2.    3.    4.   Nan   Inf    6. !

-->isnan(x)
 ans  =

! F F F F T F F !
```

- isinf() chequea si hay valores ***Inf***, devolviendo una matriz de unos y ceros

```
-->isinf(x)
 ans  =

! F F F F F T F !
```

- isempty() chequea si un vector o matriz está vacío

- isglobal() chequea si una variable es global

```
Scilab:97> isglobal(y)
ans = 0
```

- issparse() chequea si una matriz es dispersa (*sparse*, es decir, con un gran número de elementos cero)

2. Otros tipos de datos de SCILAB

En los Capítulos precedentes se ha visto la "especialidad" de SCILAB: trabajar con vectores y matrices. En este Capítulo se va a ver que SCILAB puede también trabajar con otros tipos de datos:

1. Conjuntos o cadenas de caracteres, fundamentales en cualquier lenguaje de programación.
2. Hipermatrices, o matrices de más de dos dimensiones.
3. Estructuras, o agrupaciones bajo un mismo nombre de datos de naturaleza diferente.
4. Vectores o matrices de celdas (cell arrays), que son vectores o matrices cuyos elementos pueden ser cualquier otro tipo de dato.

2.1 Cadenas de caracteres

SCILAB trabaja también con ***cadenas de caracteres***, con ciertas semejanzas y también diferencias respecto a C/C++ y Fortran. A continuación se explica lo más importante del manejo de cadenas de caracteres en SCILAB. Los caracteres de una cadena se almacenan en un vector, con un carácter por elemento. Cada carácter ocupa dos bytes. Las cadenas de caracteres van entre ***apóstrofos*** o ***comillas simples***, como por ejemplo: 'cadena'. Una ***matriz de caracteres*** es una matriz cuyos elementos son caracteres, o bien una matriz cuyas filas son cadenas de caracteres. Todas las filas de una ***matriz de caracteres*** deben tener el ***mismo número de elementos***. Si es preciso, las cadenas (filas) más cortas se completan con blancos.

Las funciones más importantes para manejo de cadenas de caracteres son las siguientes:

- code2str – convierte valores integer a caracteres.

```
-->code2str([-28 12 18 21 10 11])
 ans  =

 Scilab
```

- convstr – convierte los caracteres de mayúscula a minúscula o viceversa
- emptystr – devuelve una string vacía
- grep – busca un string dentro de un vector de strings.
- justify – justifica el array
- length – nos da la longitud del objeto
- part - extraction de una parte del string
- str2code – convierte valores tipo caracter en integer
- strcat – Concatenación de strings
- strindex – busca la posición de un caracter o string en una string.
- string - conversion a string de una matriz

```
-->string(rand(2,2))
 ans  =

!0.2113249  0.0002211  !
!                      !
!0.7560439  0.3303271  !

-->deff('y=mymacro(x)','y=x+1')
```

```
-->[out,in,text]=string(mymacro)
 text  =

    []
 in  =

 x
 out  =

 y

-->x=123.356; 'Result is '+string(x)
 ans  =

 Result is 123.356
```

- strsubst - sustituye un caracter por otro en un string.

2.2 Hipermatrices (arrays de más de dos dimensiones)

SCILAB permite trabajar con ***hipermatrices***, es decir con matrices de más de dos dimensiones. Una posible aplicación es almacenar con un único nombre distintas matrices del mismo tamaño (resulta una hipermatriz de 3 dimensiones). Los elementos de una hipermatriz pueden ser números, caracteres, estructuras, y vectores o matrices de celdas.

El tercer subíndice representa la tercera dimensión la “profundidad” de la hipermatriz.

2.2.1 Definición de Hipermatrices

Las funciones que operan con matrices de más de dos dimensiones son análogas a las funciones vistas previamente, aunque con algunas diferencias. Por ejemplo, las siguientes sentencias generan, en dos pasos, una matriz de 2x2x2:

```
--> rand(2,2,2)
ans =

ans(:,:,1) =

  0.31106  0.70197
  0.48694  0.45170

ans(:,:,2) =

  0.017063  0.261818
  0.234682  0.340703
```

2.3. Estructuras

Una estructura (***struct***) es una agrupación de datos de tipo diferente bajo un mismo nombre. Estos datos se llaman ***miembros*** (*members*) o ***campos*** (*fields*). Una estructura es un nuevo tipo de dato, del que luego se pueden crear muchas variables (***objetos*** o *instances*). Por ejemplo, la estructura ***alumno*** puede contener los campos ***nombre*** (una cadena de caracteres) y ***carnet*** (un número).

2.3.1. Creación de Estructuras

En SCILAB la estructura ***alumno*** se crea creando un objeto de dicha estructura. A diferencia de otros lenguajes de programación, no hace falta definir previamente el modelo o patrón de la estructura. Una posible forma de hacerlo es crear uno a uno los distintos campos, como en el ejemplo siguiente:

```
-->alumno.nombre='jose';
```

```
-->alumno.apellido='escucha';

-->alumno
 alumno  =

   nombre: "jose"
   apellido: "escucha"
```

También puede crearse la estructura por medio de la función ***struct()***, como por ejemplo,

```
-->alumno=struct('nombre','jose maria','dni',77335559)
 alumno  =

   nombre: "jose maria"
   dni: 77335559
```

Scilab permite, además, añadir un nuevo campo a una estructura en cualquier momento. La siguiente sentencia añade el campo ***edad*** a todos los elementos del vector ***alumno.***

```
-->alumno.edad=23
 alumno  =

   nombre: "jose maria"
   dni: 77335559
   edad: 23
```

Como hemos visto desde un inicio, Scilab trabaja con matrices, por lo tanto, todo lo que veamos para un elemento es extrapolable a una matriz, vector o conjunto de estos, por lo tanto, podemos hacer matrices de **structs** al igual que hemos hecho **structs** de vectores/matrices.

2.3.2. Funciones para operar Estructuras

Las estructuras de SCILAB disponen de funciones que facilitan su uso. Algunas de estas funciones son las siguientes:

- isstruct(ST) permite saber si **ST** es o no una estructura
- fieldnames(*struct*) devuelve un array de celdas con el número y nombre de los elementos de la estructura.
- mtlb_isfield (*expr, nombre*) nos dice si la structura posee un campo con el nombre indicado.

```
-->isstruct(alumno)
 ans  =

  T
```

2.4. Vectores o matrices de celdas (*Cell Array*)

Un vector (matriz o hipermatriz) de celdas es un vector (matriz o hipermatriz) cuyos elementos son cada uno de ellos una variable de tipo cualquiera. En un array ordinario todos sus elementos son números o cadenas de caracteres. Sin embargo, en un ***array de celdas***, el primer elemento puede ser un número; el segundo una matriz; el tercero una cadena de caracteres; el cuarto una estructura, etc.

2.4.1. Creación de vectores y matrices de celdas

Obsérvese por ejemplo cómo se crea, utilizando ***llaves*** {}, el siguiente vector de celdas,

```
-->vc(1,1)={'primero'};

-->vc(1,2)={'segundo'};

-->vc(2,1)={'tercero'};

-->vc(2,2)={'cuarto'}
 vc  =

!primero  segundo  !
!                  !
!tercero  cuarto   !
```

Otra nomenclatura alternativa y similar, que también utiliza llaves se trata de crear el *cell array* antes y luego irlo llenando.

```
-->v=cell(2,2)
 v  =


!{}  {}  !
!        !
!{}  {}  !
```

```
-->v(1).entries={'uno'}
 v  =


!"uno"  !
!       !
!{}     !
!       !
!{}     !
!       !
!{}     !
```

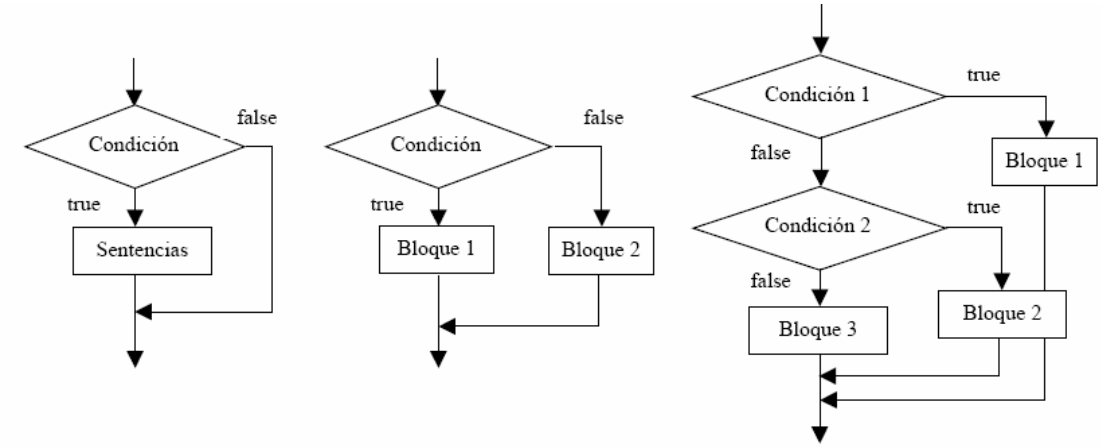
El gran problema de esta estructura es que sólo está implementada para strings y no para datos.

3. Programación en Scilab

Este es uno de los puntos flojos de todo este tipo de herramientas de cálculo numérico, ya que no existen grandes posibilidades de programación, aunque sí las formas de programación básicas.

3.1. Bifurcaciones y bucles

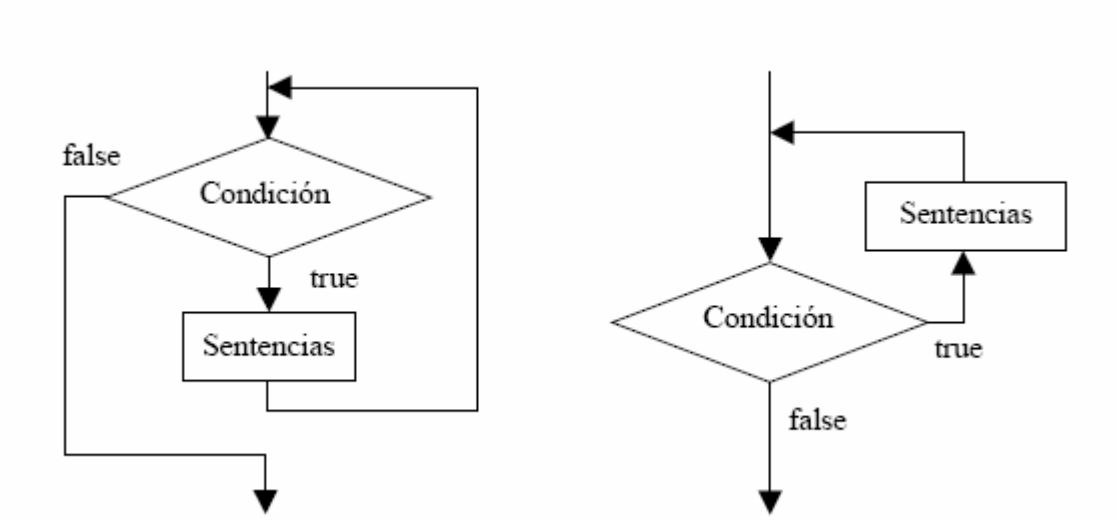
Como todo lenguaje de programación, podemos encontrar bifurcaciones y bucles. Las bifurcaciones sirven para realizar una u otra operación:



Los bucles nos permiten realizar varias iteraciones de un mismo proceso, o sea, realizar una misma operación sobre distintos elementos. Podemos encontrar varios tipos de bucles:

- while
- do-until
- for

Además de las sentencias **break** y **continue** utilizadas dentro de los bucles para salir del proceso.



3.1.1. Sentencia IF

Esta sentencia nos sirve para hacer bifurcaciones, podemos hacer 3 usos diferentes de ella:

- Una sola sentencia que utilizamos si es verdadera y sino no hacemos nada:

```
if (condition)
        then-body
end
```

- Utilizando la expresión *else* con la que conseguiremos hacer uso de una expresión u otra si es consecuentemente *true o false*.

```
if (condition)
        then-body
else
       else-body
end
```

- Utilizando la expression *elseif* con la que se pueden anidar bifurcaciones (aunque es mejor usar la sentencia switch)

```
if (condition)
       then-body
elseif (condition)
       elseif-body
else    #es la opción por defecto cuando no se cumple ninguna condición
       else-body
end
```

Una observación muy importante: la condición del ***if*** puede ser una *condición matricial*, del tipo **A**==**B**, donde **A** y **B** son matrices del mismo tamaño. Para que se considere que la *condición* se cumple, es necesario que sean *iguales dos a dos* ***todos*** *los elementos* de las matrices **A** y **B**. Basta que haya dos elementos diferentes para que las matrices no sean iguales, y por tanto las sentencias del ***if*** no se ejecuten. Análogamente, una condición en la forma A~=B exige que todos los elementos sean diferentes dos a dos. Bastaría que hubiera dos elementos iguales para que la condición no se cumpliese. En resumen:

- if A==B exige que ***todos*** los elementos sean ***iguales*** dos a dos
- if A~=B exige que ***todos*** los elementos sean ***diferentes*** dos a dos

3.1.2. Sentencia SELECT

Se trata de una sentencia con la que podemos hacer una función similar a la concatenación de sentencias *elseif*, de manera que simplifiquemos el modo de programar:

```
select expr0,
 case expr1 then instructions1,
 case expr2 then instructions2,
 ...
 case exprn then instructionsn,
 [else instructions],
end
```

Al principio se evalúa la ***expr0***, cuyo resultado debe ser un número escalar o una cadena de caracteres. Este resultado se compara con las ***expri***, y se ejecuta el bloque de sentencias que corresponda con ese resultado. Si ninguno es igual a ***expr0*** se ejecutan las sentencias correspondientes a ***else***.

3.1.3. Sentencia FOR

Repite una serie de sentencias un número determinado de veces, sin importar los procesos que ocurran dentro, por lo que la única manera de salir del bucle es esperar que acabe (más adelante veremos la sentencia *break*).

```
for variable=expression do instruction, ,instruction,end
```

cuando **var** llega al valor **expression** el bucle se detiene.

3.1.4. Sentencia WHILE

Similar a DO-UNTIL salvo que la comprobación de la condición se hace antes de la ejecución de la iteración.

```
while (condition)
       body
endwhile
```

3.1.5. Sentencia BREAK y ABORT

Al igual que en C/C++, la sentencia ***break*** hace que se termine la ejecución del bucle más interno de los que comprenden a dicha sentencia. La sentencia **abort** hace que automáticamente se pare la ejecución de la iteración actual, por lo que vuelve al principio del bucle (sólo sirve para el bucle FOR).

3.2. Ficheros *.sci

Los ficheros con extensión (***.sci***) son ficheros de texto sin formato (ficheros ASCII) que constituyen el centro de la programación en SCILAB. Ya se han utilizado en varias ocasiones. Estos ficheros se crean y modifican con un editor de textos cualquiera.

Existen dos tipos de ficheros ****.sci***, los ***ficheros de comandos*** (llamados *scripts* en inglés) y las ***funciones***. Los primeros contienen simplemente un conjunto de comandos que se ejecutan sucesivamente cuando se teclea el nombre del fichero en la línea de comandos de SCILAB. Un fichero de comandos puede llamar a otros ficheros de comandos.

Las ***funciones*** permiten definir funciones enteramente análogas a las de SCILAB, con su ***nombre***, sus ***argumentos*** y sus ***valores de retorno***. Los ficheros ****.sci*** que definen

funciones permiten extender las posibilidades de SCILAB. Las funciones definidas en ficheros ***.sci*** se caracterizan porque la primera línea (que no sea un comentario) comienza por la palabra ***function***, seguida por los *valores de retorno* (entre corchetes [ ] y separados por comas, si hay más de uno), el signo igual (=) y el *nombre de la función*, seguido de los *argumentos* (entre paréntesis y separados por comas).

Recuérdese que un fichero ***.sci*** puede llamar a otros ficheros ***.sci***, e incluso puede llamarse a sí mismo de forma recursiva. Los ficheros de comandos se pueden llamar también desde funciones, en cuyo caso las variables que se crean pertenecen a espacio de trabajo de la función. El espacio de trabajo de una función es independiente del espacio de trabajo base y del espacio de trabajo de las demás funciones. Esto implica por ejemplo que no puede haber colisiones entre nombres de variables aunque varias funciones tengan una variable llamada A, en realidad se trata de variables completamente distintas (a no ser que A haya sido declarada como variable ***global***).

3.2.1. Ficheros de Comandos (*SCRIPTS*)

Los ficheros de comandos o *scripts* son ficheros con un nombre tal como ***file1.sci*** que contienen una sucesión de comandos análoga a la que se teclearía en el uso interactivo del programa. Dichos comandos se ejecutan sucesivamente cuando se teclea el nombre del fichero que los contiene (sin la extensión), es decir cuando se teclea ***file1*** con el ejemplo considerado.

Cuando se ejecuta desde la línea de comandos, las variables creadas por ***file1*** pertenecen al espacio de trabajo base de SCILAB. Por el contrario, si se ejecuta desde una función, las variables que crea pertenecen al espacio de trabajo de la función.

En los ficheros de comandos conviene poner los puntos y coma (;) al final de cada sentencia, para evitar una salida de resultados demasiado cuantiosa. Un fichero ***.sci*** puede llamar a otros ficheros ***.sci***, e incluso se puede llamar a sí mismo de modo recursivo.

El comando **mtlb_*echo*** hace que se impriman los comandos que están en un *script* a medida que van siendo ejecutados. Este comando tiene varias formas:

- mtlb_echo on activa el ***echo*** en todos los ficheros script
- mtlb_echo off desactiva el ***echo***
- mtlb_echo file on donde 'file' es el nombre de un fichero de función, activa el ***echo*** en esa función
- mtlb_echo file off desactiva el ***echo*** en la función
- mtlb_echo file pasa de **on** a **off** y viceversa
- mtlb_echo on all activa el ***echo*** en todas las funciones
- mtlb_echo off all desactiva el ***echo*** de todas las funciones

3.3.2. Definición de Funciones

La ***primera línea*** de un fichero llamado ***name.sci*** que define una función tiene la forma:

```
function [lista de valores de retorno] = name(lista de argumentos)
```

donde ***name*** es el nombre de la función. Entre corchetes y separados por comas van los ***valores de retorno*** (siempre que haya más de uno), y entre paréntesis también separados por comas los ***argumentos***. Puede haber funciones sin valor de retorno y también sin argumentos. Recuérdese que los ***argumentos*** son los ***datos*** de la función y los ***valores de retorno*** sus ***resultados***, decir que los argumentos de entrada son por valor y no pueden ser modificados, por lo que si queremos modificarlos debemos utilizarlos a su vez como salida. Si no hay valores de retorno se omiten los corchetes y el signo igual (=); si sólo hay un valor de retorno no hace falta poner corchetes. Tampoco hace falta poner paréntesis si no hay argumentos.

Las variables definidas dentro de una función son ***variables locales***, en el sentido de que son inaccesibles desde otras partes del programa y en el de que no interfieren con

variables del mismo nombre definidas en otras funciones o partes del programa. Se puede decir que pertenecen al propio espacio de trabajo de la función y no son vistas desde otros espacios de trabajo. Para que la función tenga acceso a variables que no han sido pasadas como argumentos es necesario declarar dichas variables como ***variables globales***, tanto en el programa principal como en las distintas funciones que deben acceder a su valor.

Dentro de la función, los valores de retorno deben ser calculados en algún sitio.De todas formas, no hace falta calcular siempre todos los posibles valores de retorno de la función, sino sólo los que el usuario espera obtener en la sentencia de llamada a la función. En cualquier función existen dos variables definidas de modo automático, llamadas ***varargin*** y ***varargout***, que representan respectivamente el número de argumentos y el número de valores de retorno con los que la función ha sido llamada. Dentro de la función, estas variables pueden ser utilizadas como el programador desee.

La ejecución de una función termina cuando se llega a su última sentencia ejecutable. Si se quiere forzar el que una función termine de ejecutarse se puede utilizar la sentencia ***return***, que devuelve inmediatamente el control al entorno de llamada.

4. Gráficos bidimensionales

A estas alturas, después de ver cómo funciona este programa, a nadie le puede resultar extraño que los gráficos 2-D de Scilab estén fundamentalmente orientados a la representación gráfica de vectores (y matrices). En el caso más sencillo los argumentos básicos de la función plot van a ser vectores. Cuando una matriz aparezca como argumento, se considerará como un conjunto de vectores columna (en algunos casos también de vectores fila).

Scilab utiliza un tipo especial de ventanas para realizar las operaciones gráficas. Ciertos comandos abren una ventana nueva y otros dibujan sobre la ventana activa, bien sustituyendo lo que hubiera en ella, bien añadiendo nuevos elementos gráficos a un dibujo anterior. Todo esto se verá con más detalle en las siguientes secciones.

4.1. Funciones gráficas 2D elementales

Scilab dispone de dos funciones básicas para crear gráficos 2-D. Estas funciones se diferencian principalmente por el tipo de implementación que utilizan en los ejes de abscisas y de ordenadas. Estas cuatro funciones son las siguientes:

- plot() crea un gráfico a partir de vectores y/o columnas de matrices, con escalas lineales sobre ambos ejes, está orientado a la compatibilidad con MatLab.
- plot2d() crea un gráfico con escala lineal o logarítmica, según las opciones que la opción que le demos.

En lo sucesivo se hará referencia casi exclusiva a la primera de estas funciones (plot). Plot2d se utiliza de un modo similar.

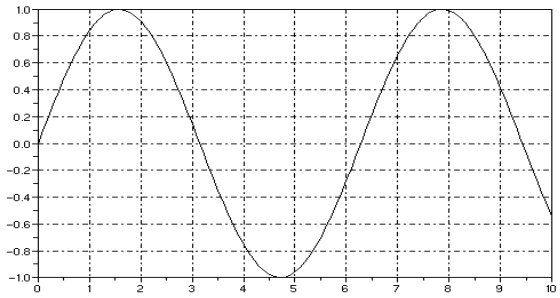
Existen además otras funciones orientadas a añadir títulos al gráfico, a cada uno de los ejes, a dibujar una cuadrícula auxiliar, a introducir texto, etc. Estas funciones son las siguientes:

- title('título') añade un título al dibujo
- legend() define rótulos para las distintas líneas o ejes utilizados en la figura. Para más detalle, consultar el Help
- xgrid activa la inclusión de una cuadrícula en el dibujo. Con xgrid('off') desaparece la cuadrícula

Los dos grupos de funciones anteriores no actúan de la misma forma. Así, la función *plot* dibuja una nueva figura en la ventana activa (en todo momento Scilab tiene una ventana activa de entre todos las ventanas gráficas abiertas), o abre una nueva figura si no hay ninguna abierta, sustituyendo cualquier cosa que hubiera dibujada anteriormente en esa ventana.

Ahora se deben ejecutar los comandos siguientes:

```
->xgrid

-->x=0:0.1:10;

-->y=sin(x);

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

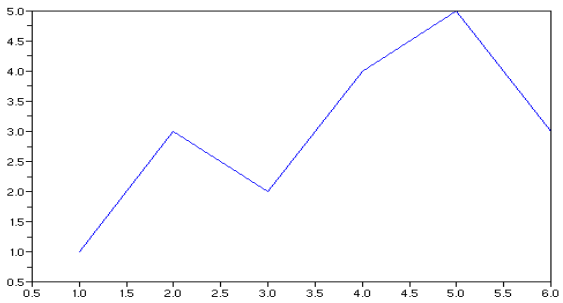


Más adelante se verá que con la función hold pueden añadirse gráficos a una figura ya existente respetando su contenido.

4.1.1. Función PLOT

Esta es la función clave de todos los gráficos 2-D en Scilab. Ya se ha dicho que el elemento básico de los gráficos bidimensionales es el vector. Se utilizan también cadenas de 1, 2 ó 3 caracteres para indicar colores y tipos de línea. La función plot(), en sus diversas variantes, no hace otra cosa que dibujar vectores. Un ejemplo muy sencillo de esta función, en el que se le pasa un único vector como argumento, es el siguiente:

```
-->x=[1 3 2 4 5 3],plot(x)
```



El resultado de este comando es que se abre una ventana. Por defecto, los distintos puntos del gráfico se unen con una línea continua. También por defecto, el color que se utiliza para la primera línea es el azul.

Cuando a la función plot() se le pasa un único vector –real– como argumento, dicha función dibuja en ordenadas el valor de los n elementos del vector frente a los índices 1, 2, ... n del mismo en abscisas. Más adelante se verá que si el vector es complejo, el funcionamiento es bastante diferente.

En la pantalla de su ordenador se habrá visto que Scilab utiliza por defecto color blanco para el fondo de la pantalla y otros colores más oscuros para los ejes y las gráficas.

Una segunda forma de utilizar la función plot() es con dos vectores como argumentos. En este caso los elementos del segundo vector se representan en ordenadas frente a los valores del primero, que se representan en abscisas. Véase por ejemplo cómo se puede dibujar un cuadrilátero de esta forma (obsérvese que para dibujar un polígono cerrado el último punto debe coincidir con el primero):

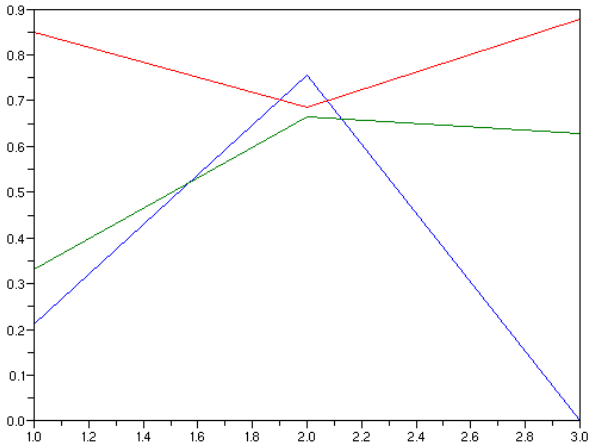
```
-->x=[1 2 3 1 1];y=[2 1 2 1 3];plot(x,y)
```

La función *plot()* permite también dibujar múltiples curvas introduciendo varias parejas de vectores como argumentos. En este caso, cada uno de los segundos vectores se dibujan en ordenadas como función de los valores del primer vector de la pareja, que se representan en abscisas. Obsérvese bien cómo se dibujan el seno y el coseno en el siguiente ejemplo:

```
-->x=0:%pi/25:6*%pi;

-->y=sin(x); z=cos(x);

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

Ahora se va a ver lo que pasa con los vectores complejos. Si se pasan a *plot()* varios vectores complejos como argumentos, Scilab simplemente representa las partes reales y desprecia las partes imaginarias.

```
-->a=rand(3,3)+rand(3,3)*%i;
-->plot(a)
```



Como ya se ha dicho, si se incluye más de un vector complejo como argumento, se ignoran las partes imaginarias.

El comando plot puede utilizarse también con matrices como argumentos. Véanse algunos ejemplos sencillos:

- plot(A) dibuja una línea por cada columna de A en ordenadas, frente al índice de los elementos en abscisas
- plot(x,A) dibuja las columnas (o filas) de A en ordenadas frente al vector x en abscisas. Las dimensiones de A y x deben ser coherentes: si la matriz A es cuadrada se dibujan las columnas, pero si no lo es y la dimensión de las filas coincide con la de x, se dibujan las filas
- plot(A,x) análogo al anterior, pero dibujando las columnas (o filas) de A en abscisas, frente al valor de x en ordenadas
- plot(A,B) dibuja las columnas de B en ordenadas frente a las columnas de A en abscisas, dos a dos. Las dimensiones deben coincidir.

Se puede obtener una excelente y breve descripción de la función *plot()* con el comando *help plot*. La descripción que se acaba de presentar se completará en la siguiente sección, en donde se verá cómo elegir los colores y los tipos de línea.

4.1.2. Estilos de línea y marcadores en la función PLOT

En la sección anterior se ha visto cómo la tarea fundamental de la función *plot()* era dibujar los valores de un vector en ordenadas, frente a los valores de otro vector en abscisas. En el caso general esto exige que se pasen como argumentos un par de vectores. En realidad, el conjunto básico de argumentos de esta función es una tripleta formada por dos vectores y una cadena de 1, 2 ó 3 caracteres que indica el color y el tipo de línea o de marker. En la tabla siguiente se pueden observar las distintas posibilidades.

Símbolo	Color	Símbolo	Marcadores
y	yellow	.	puntos
m	magenta	o	círculos
c	cyan	x	marcas en x
r	r ed	+	marcas en +
g	green	*	marcas en *
b	blue	s	marcas cuadradas (square)
w	white	d	marcas en diamante (diamond)
k	black	^	triángulo apuntando arriba

4.1.3. Función PLOT2D

Tenemos además de la función *plot* podemos usar una función específica de Scilab, **plot2d**, cuyo funcionamiento es idéntico, pero incluye algunas funciones nuevas, sobretodo en las propiedades de los ejes, pues podremos definir tanto en el *eje X* como en el *eje Y* escala logarítmica y lineal.

- Plot2d(x,y,logflag=’nl’) donde “n” hace que el eje X sea lineal y “l” hace que el eje Y sea logarítmico; tenemos las siguientes opciones: “nn”,”nl”,”ln”,”ll”.
- Plot2d(x,y,rect=[xmin,xmax,ymin,ymax])
- Plot2d(x,y,<opt_arg>) donde las opciones son *axesflag, frameflag, leg, logflag, nax, rect, strf, y style*.

4.1.4. Comando SUBPLOT

Una ventana gráfica se puede dividir en m particiones horizontales y n verticales, con objeto de representar múltiples gráficos en ella. Cada una de estas subventanas tiene sus propios ejes, aunque otras propiedades son comunes a toda la figura. La forma general de este comando es:

subplot(m,n,i)

donde m y n son el número de subdivisiones en filas y columnas, e i es la subdivisión que se convierte en activa. Las subdivisiones se numeran consecutivamente empezando por las de la primera fila, siguiendo por las de la segunda, etc. Por ejemplo, la siguiente secuencia de comandos genera cuatro gráficos en la misma ventana:

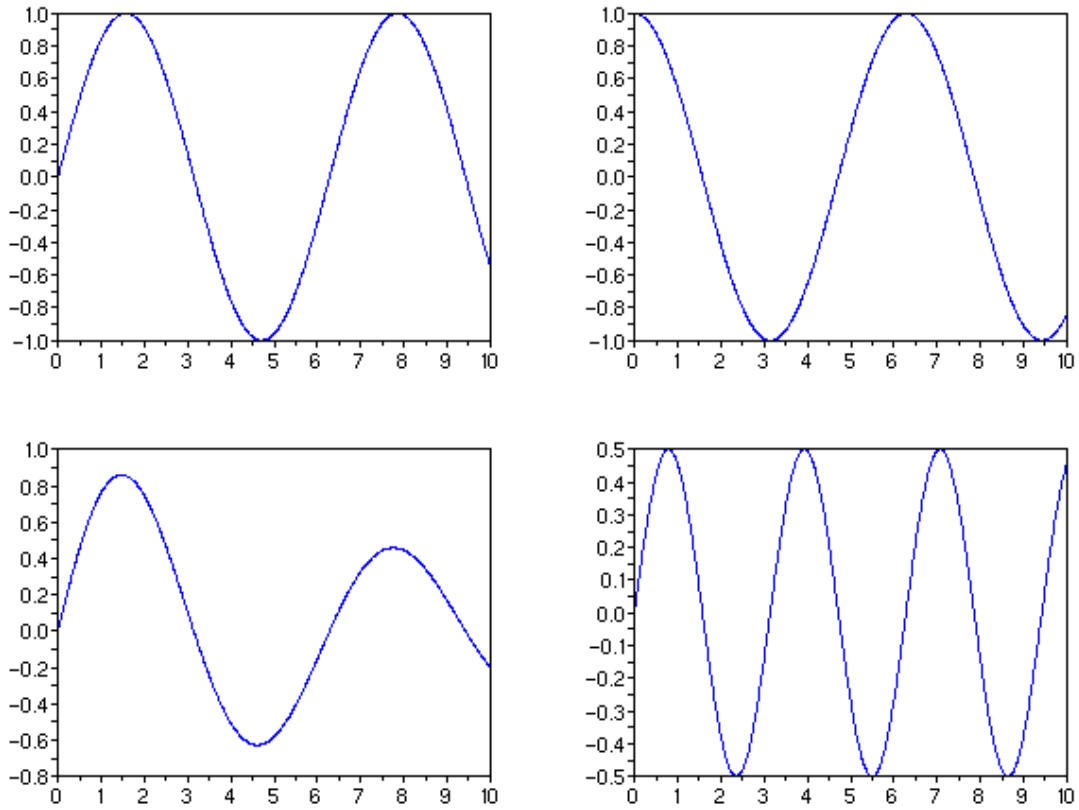
-->x=0:0.01:10;y=sin(x); z=cos(x); w=exp(-x*.1).*y; v=y.*z;

-->subplot(2,2,1), plot(x,y)

-->subplot(2,2,2), plot(x,z)

-->subplot(2,2,3), plot(x,w)

-->subplot(2,2,4), plot(x,v)



Se puede practicar con este ejemplo añadiendo títulos a cada subplot, así como rótulos para los ejes. Se puede intentar también cambiar los tipos de línea.

4.1.5. Control de los Ejes

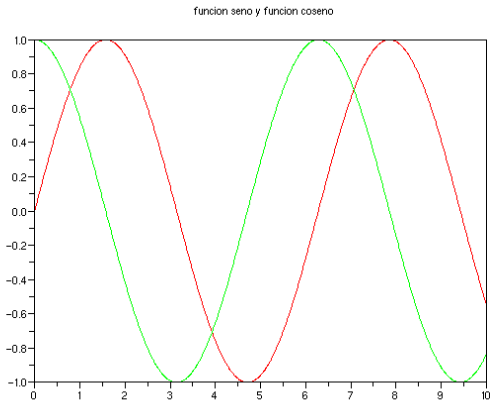
También en este punto Scilab tiene sus opciones por defecto, que en algunas ocasiones puede interesar cambiar. El comando básico es el comando *a=get("current_axes")* . Por defecto, Scilab ajusta la escala de cada uno de los ejes de modo que varíe entre el mínimo y el máximo valor de los vectores a representar. Este es el llamado modo "auto", o modo automático. Ahora tenemos en la variable *a* las propiedades de los ejes, modificando **a** que es un **struct,** podemos modificar los ejes del gráfico.

4.2. Control de ventanas gráficas:

La función **clf** cierra la figura activa, mientras que **clf(n)** cierra la ventana o figura número n.

Para practicar un poco con todo lo que se acaba de explicar, ejecútense las siguientes instrucciones de Scilab, observando con cuidado los efectos de cada una de ellas en la ventana activa.

```
-->plot(x,sin(x),'r',x,cos(x),'g')
-->a=get('current_axes')
-->a.title.text=’funcion seno y funcion coseno’
```



4.3. Otras funciones gráficas 2-D

Existen otras funciones gráficas bidimensionales orientadas a generar otro tipo de gráficos distintos de los que produce la función plot() y sus análogas. Algunas de estas funciones son las siguientes (para más información sobre cada una de ellas en particular, utilizar help nombre_función):

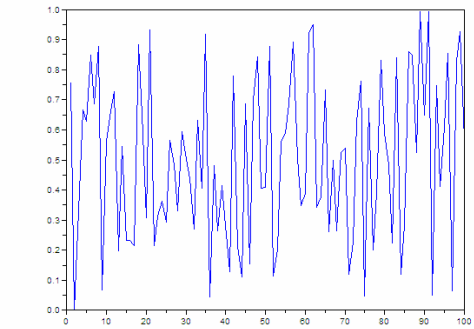
2d plotting

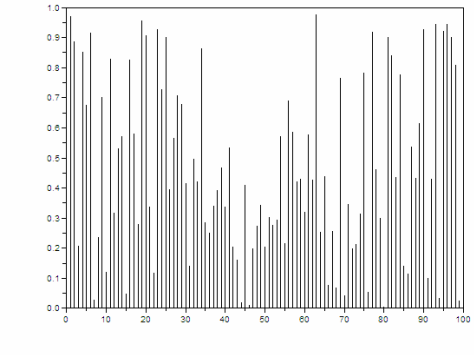
- plot2d2 : dibuja un diagrama de escalera
- plot2d3 : dibuja un diagrama de barras
- plot2d4 : diagrama de flechas
- fplot2d : dibuja una función
- contour2d : dibuja curvas de nivel en 2 dimensiones
- fcontour2d : dibuja curvas de nivel en 2 dimensionesa partir de una función.
- grayplot : dibuja una superficie con colores
- fgrayplot : idem a partir de una función
- errbar : dibuja un diagrama de errores
- histplot : dibuja un histograma

-->x=rand(100,1);

-->plot2d(x)

-->plot2d3(x)

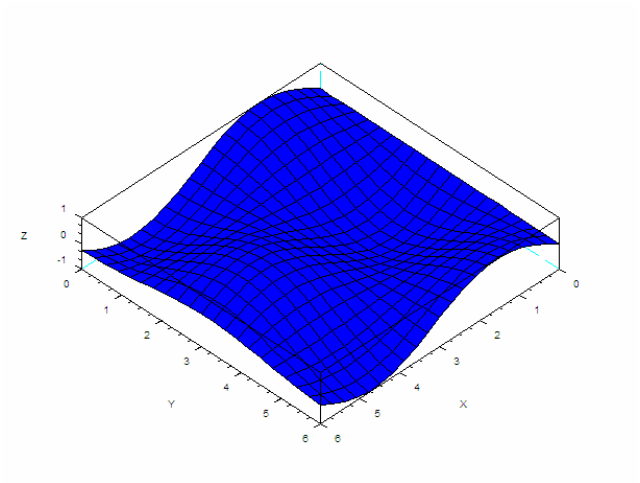




5. Gráficos tridimensionales

SCILAB tiene posibilidades de realizar varios tipos de gráficos 3D. La primera forma de gráfico 3D es la función plot3d. Esta función dibuja una superficie en 3 dimensiones.

```
-->t=[0:0.3:2*%pi]';

-->z=sin(t)*cos(t');

-->plot3d(t,t,z)
```

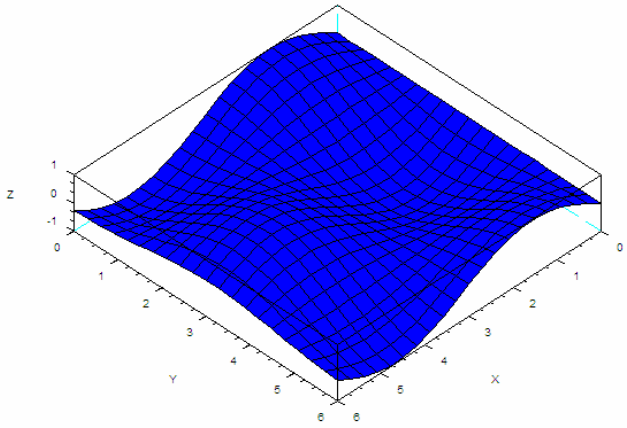


Ahora se verá cómo se representa una función de dos variables. Para ello se va a definir una función de este tipo en un fichero llamado test3d.sce. La fórmula será la siguiente:

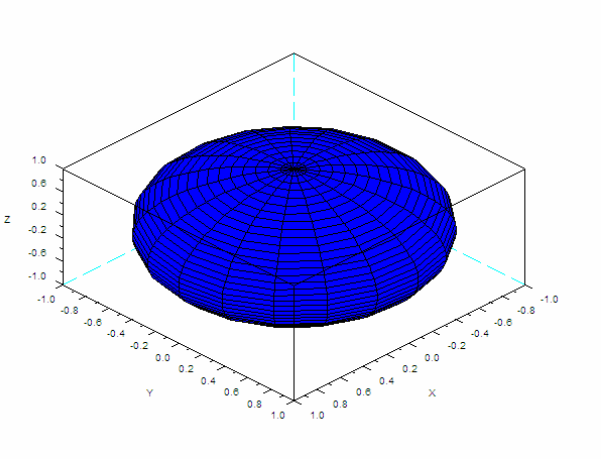
```
function z=test3d(x,y)
z = 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
  - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
      – 1/3*exp(-(x+1).^2 – y.^2);
```

Ahora, vamos a ejecutar los siguientes comandos:

```
t=[0:0.3:2*%pi]';
z=sin(t)*cos(t');
[xx,yy,zz]=genfac3d(t,t,z);
clf()
plot3d(xx,yy,zz)
```




```
-->u = linspace(-%pi/2,%pi/2,40);

-->v = linspace(0,2*%pi,20);

-->X = cos(u)'*cos(v);

-->Y = cos(u)'*sin(v);

-->Z = sin(u)'*ones(v);

-->plot3d2(X,Y,Z);
```



Estos son unos ejemplos de las posibilidades que Scilab ofrece para el uso de gráficos 3-D.

5.1 Dibujo de líneas: Función PARAM3D

La función *param3d* es análoga a su homóloga bidimensional *plot2d*. Su forma más sencilla es la siguiente:

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

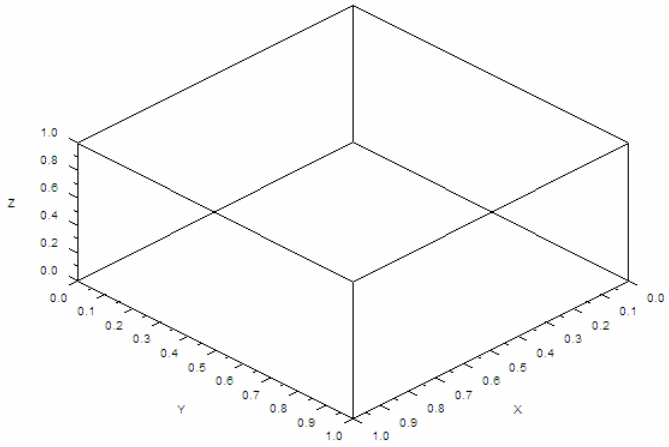
que dibuja una línea que une los puntos (x(1), y(1), z(1)), (x(2), y(2), z(2)), etc. y la proyecta sobre un plano para poderla representar en la pantalla. Al igual que en el caso plano, se puede incluir una cadena de 1, 2 ó 3 caracteres para determinar el color, los markers, y el tipo de línea, también se pueden utilizar tres matrices X, Y y Z del mismo tamaño:

```
-->param3d(X,Y,Z)
```

en cuyo caso se dibujan tantas líneas como columnas tienen estas 3 matrices, cada una de las cuales está definida por las 3 columnas homólogas de dichas matrices.

A continuación se va a realizar un ejemplo sencillo consistente en dibujar un cubo. Para ello se creará una matriz que contenga las aristas correspondientes, definidas mediante los vértices del cubo como una línea poligonal continua (obsérvese que algunas aristas se dibujan dos veces). La matriz A cuyas columnas son las coordenadas de los vértices, y cuyas filas son las coordenadas ***x, y*** y ***z*** de los mismos:

```
-->A=[0 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 0
-->      0 0 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1 1
-->      0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0];

-->param3d(A(1,:)',A(2,:)',A(3,:)')
```



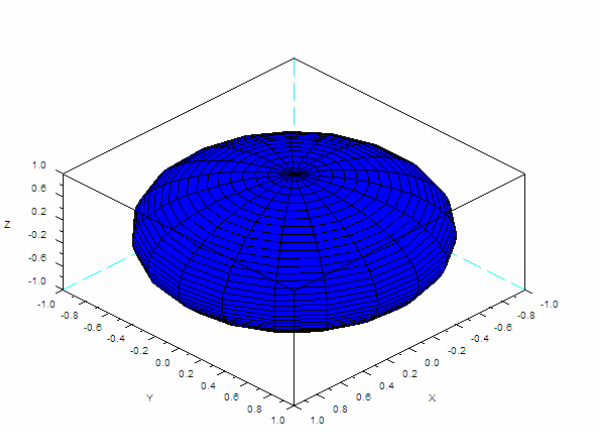
5.2. Dibujo de mallados: Funciones PLOT3D, PLOT3D2, PLOT3D3

Ahora se verá con detalle cómo se puede dibujar una función de dos variables (z=f(x,y)) sobre un dominio rectangular. Se verá que también se pueden dibujar los elementos de una matriz como función de los dos índices.

Sean x e y dos vectores que contienen las coordenadas en una y otra dirección de la retícula (grid) sobre la que se va a dibujar la función. Después hay que crear dos matrices X (cuyas filas son copias de x) e Y (cuyas columnas son copias de y). Estas matrices se crean con la función genfac3d. Estas matrices representan respectivamente las coordenadas x e y de todos los puntos de la retícula. La matriz de valores Z se calcula a partir de las matrices de coordenadas X e Y. Finalmente hay que dibujar esta matriz Z con la función *plot3d3*, cuyos elementos son función elemento a elemento de los elementos de X e Y. Véase como ejemplo el dibujo de la función sen(r)/r (siendo r=sqrt(x2+y2); para evitar dividir por 0 se suma al denominador el número pequeño eps):

```
-->clf
```

```
-->u = linspace(-%pi/2,%pi/2,40);

-->v = linspace(0,2*%pi,20);

-->X = cos(u)'*cos(v);

-->Y = cos(u)'*sin(v);

-->Z = sin(u)'*ones(v);

-->plot3d2(X,Y,Z);
```



Se habrá podido comprobar que la función *plot3d2* dibuja en perspectiva una función en base a una retícula de líneas de colores, rodeando cuadriláteros del color de fondo, con eliminación de líneas ocultas. Ejecútese ahora el comando:

```
-->plot3d2(X,Y,Z)
```

En vez de líneas aparece ahora una superficie faceteada (aunque no es fácilmente visible, pero de manera teórica es así). El color de las facetas depende también del valor de la función.

5.3. Dibujo de línea de contorno: Función CONTOUR

Una forma distinta de representar funciones tridimensionales es por medio de isolíneas o curvas de nivel.

```
-->t=%pi*[-10:10]/10;

-->deff("[z]=surf(x,y)","z=sin(x)*cos(y)"); z=feval(t,t,surf);


-->rect=[-%pi,%pi,-%pi,%pi,-1,1];

-->contour(t,t,z,10,35,45," ",[0,1,0],rect)
```

6. Otros aspectos de Scilab

6.1. Guardar variables y estados de una sesión:

Comandos *save* y *load*

En muchas ocasiones puede resultar interesante interrumpir el trabajo con Scilab y poderlo recuperar más tarde en el mismo punto en el que se dejó (con las mismas variables definidas, con los mismos resultados intermedios, etc.). Hay que tener en cuenta que al salir del programa todo el contenido de la memoria se borra automáticamente.

Para guardar el estado de una sesión de trabajo en el ***directorio actual*** existe el comando ***save***. Si se teclea:

-->save('filename')

antes de abandonar el programa, se crea un fichero binario con el estado de la sesión (excepto los gráficos, que por ocupar mucha memoria hay que guardar aparte).

Dicho estado puede recuperarse la siguiente vez que se arranque el programa con el comando:

-->load('filename')

Esta es la forma más básica de los comandos ***save*** y ***load***. Se pueden guardar también matrices y vectores de forma selectiva y en ficheros con nombre especificado por el usuario. Por ejemplo, el comando:

-->save(filename [,x1,x2,...,xn])

guarda las variables **x1,x2,…,xn** en un fichero binario llamado ***filename***. Para recuperarlas en otra sesión basta teclear:

-->load(filename [,x1,x2,...,xn])

Si no se indica ningún nombre de variable, se guardan todas las variables creadas en esa sesión.

6.2. Guardar sesión: Comando *diary*

Los comandos ***save*** y ***load*** crean ficheros binarios o ASCII con el estado de la sesión. Existe otra forma más sencilla de almacenar en un fichero un texto que describa lo que el programa va haciendo. El comando *diary* nos permite guardar el proceso de trabajo en un archivo de tipo texto:

```
octave:32>diary “filename”
```

6.3. Medida de tiempos y de esfuerzo de cálculo

GNU Octave dispone también de funciones que permiten calcular el tiempo empleado en las operaciones matemáticas realizadas. Algunas de estas funciones son las siguientes:

- **cputime** devuelve el tiempo de CPU (con precisión de centésimas de segundo) desde que el programa arrancó. Llamando antes y después de realizar una operación y restando los valores devueltos, se puede saber el tiempo de CPU empleado en esa operación. Este tiempo sigue corriendo aunque GNU Octave esté inactivo.
- **etime(t2, t1)** tiempo transcurrido entre los vectores **t1** y **t2** (¡atención al orden!), obtenidos como respuesta al comando ***clock***.
- **tic** *operaciones* **toc** imprime el tiempo en segundos requerido por ***ops***. El comando ***tic*** pone el reloj a cero y ***toc*** obtiene el tiempo transcurrido

6.4. Funciones de función

En Scilab existen funciones a las que hay que pasar como argumento el nombre de otras funciones, para que puedan ser llamadas desde dicha función. Así sucede por ejemplo si se desea calcular la integral definida de una función. Utilizamos una función llamada ***prueba*** que se va a definir en un fichero llamado ***prueba.sci***

```
function y=prueba(x)
y = 1./((x-.3).^2+.01)+1./((x-.9).^2+.04)-6;
```

- **Integración Numérica de Funciones**

Lo que se va a hacer es calcular la integral definida de esta función entre dos valores de la abscisa **x**. Calculamos el área comprendida bajo la función entre los puntos 0 y 1 (obsérvese que el nombre de la función a integrar se pasa entre apóstrofos, como cadena de caracteres):

```
-->integrate('prueba(x)','x',0,%pi)
 ans  =

    23.16678
```

Si se teclea ***help integrate*** se puede obtener más de información sobre esta función.