# ***Curso basico de MATLAB***

## 14. Operadores lógicos y operadores relacionales

### *14.1 Operadores lógicos*

Los operadores lógicos de ***Matlab*** son los siguientes:

| | |
|---|---|
| **&** | ***and***. Se evalúan siempre ambos operandos, y el resultado es *true* sólo si ambos son *true*. Puede escribirse A & B o and(A,B) |
| **&&** | ***and breve***. Si el primer operando es *false* ya no se evalúa el segundo, pues el resultado final ya no puede ser más que *false*. |
| **\|** | ***or*.** Se evalúan siempre ambos operandos, y el resultado es *false* sólo si ambos son *false*. Puede escribirse A \| B o or(A,B) |
| **\|\|** | ***or breve***. Si el primer operando es *true* ya no se evalúa el segundo, pues el resultado final no puede ser más que *true*. |
| ~ | ***negación lógica***. Puede escribirse ~ A o not(A) |
| **xor(A,B)** | Realiza un ***"or exclusivo"***, es decir, devuelve 0 en el caso en que ambos sean 1 ó ambos sean 0. |

Los operadores lógicos se combinan con los relacionales para poder comprobar el cumplimiento de condiciones múltiples.
Los *operadores lógicos breves* (&&) y (||) se utilizan para simplificar las operaciones de comparación evitando operaciones innecesarias, pero también para evitar ciertos errores que se producirían en caso de evaluar incondicionalmente el segundo argumento. Por ejemplo la siguiente sentencia, evita una división por cero:

```
r = (b~=0) && (a/b>0);
```

### *14.2 Operadores relacionales*

Los operadores relacionales de ***Matlab*** son los siguientes:

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

Si una comparación se cumple, el resultado es 1 (*true*), mientras que si no se cumple es 0 (*false*). Recíprocamente, cualquier valor distinto de cero es considerado como *true* y el cero equivale a *false*.

Cuando se comparan matrices, se compara elemento a elemento. Por ejemplo:

```
>> A=[1 2;0 3];
>>B=[4 2;1 5];

>> A==B
ans =
        0 1
        0 0

>> A~=B
ans =
        1 0
        1 1
```

## 15. Cadena de caracteres

***Matlab*** trabaja también con ***cadenas de caracteres***, con ciertas semejanzas y también diferencias respecto a C.
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'. Si la cadena debe contener comillas, estas se representan por un doble carácter comilla, de modo que se pueden distinguir fácilmente del principio y final de la cadena. Por ejemplo, para escribir la cadena **ni 'idea'** se escribiría **'ni''idea'''**.
Una ***matriz de caracteres*** es una matriz cuyos elementos son caracteres o 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. Por ejemplo:

```
>> c='cadena'
c =
        cadena

>> size(c)                      devuelve las dimensiones del array
ans =
        1 6

>> double(c)                    convierte en números ASCII cada carácter
ans =
        99 97 100 101 110 97

>> char(abs(c))                 convierte números ASCII en caracteres
ans =
        cadena

>> cc=char('más','madera')      convierte dos cadenas en una matriz
cc =
        más
        madera
```

```
>> size(cc)
ans =
        2 6
```

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

| | |
|---|---|
| double(c) | Convierte en números ASCII cada carácter |
| char(v) | Convierte un vector de números **v** en una cadena de caracteres |
| char(c1,c2) | Crea una matriz de caracteres, completando con blancos las cadenas más cortas |
| deblank(c) | Elimina los blancos al final de una cadena de caracteres |
| disp(c) | Imprime el texto contenido en la variable **c** |
| ischar(c) | Detecta si una variable es una cadena de caracteres |
| isletter() | Detecta si un carácter es una letra del alfabeto. Si se le pasa un vector o matriz de caracteres devuelve un vector o matriz de unos y ceros |
| isspace() | Detecta si un carácter es un espacio en blanco. Si se le pasa un vector o matriz de caracteres devuelve un vector o matriz de unos y ceros |
| strcmp(c1,c2) | Comparación de cadenas. Si las cadenas son iguales devuelve un uno, y si no lo son, devuelve un cero. |
| strcmpi(c1,c2) | Igual que *strcmp(c1,c2)*, pero ignorando la diferencia entre mayúsculas y minúsculas |
| strncmp(c1,c2,n) | Compara los **n** primeros caracteres de dos cadenas |
| c1==c2 | Compara dos cadenas carácter a carácter. Devuelve un vector o matriz de unos y ceros |
| s=[s,' y más'] | Concatena cadenas, añadiendo la segunda a continuación de la primera |
| findstr(c1,c2) | Devuelve un vector con las posiciones iniciales de todas las veces en que la cadena más corta aparece en la más larga |
| strmatch(cc,c) | Devuelve los índices de todos los elementos de la matriz de caracteres **cc**, que empiezan por la cadena **c** |
| strrep(c1,c2,c3) | Sustituye la cadena **c2** por **c3**, cada vez que **c2** es encontrada en **c1** |
| [p,r]=strtok(t) | Separa las palabras de una cadena de caracteres **t**. Devuelve la primera palabra **p** y el resto de la cadena **r** |
| int2str(v) | Convierte un número entero en cadena de caracteres |
| num2str(x,n) | Convierte un número real **x** en su expresión por medio de una cadena de caracteres, con cuatro cifras decimales por defecto (pueden especificarse más cifras, con un argumento opcional **n**) |
| str2double(str) | Convierte una cadena de caracteres representando un número real en el número real correspondiente |
| vc=cellstr(cc) | Convierte una matriz de caracteres **cc** en un vector de celdas **vc**, eliminando los blancos adicionales al final de cada cadena. La función ***char()*** realiza las conversiones opuestas |
| sprintf | Convierte valores numéricos en cadenas de caracteres, de acuerdo con las reglas y formatos de conversión del lenguaje C. |

Algunos ejemplos:

```
>> num2str(pi)
ans =
        3.142

>> num2str(pi,8)
ans =
        3.1415927
```

Si se desea convertir los valores numéricos en cadenas de caracteres para poder imprimirlos como títulos en los dibujos o gráficos

```
>> fahr=70; grd=(fahr-32)/1.8;
>> title(['Temperatura ambiente: ',num2str(grd),' grados centígrados'])
```

## *15.1 La función INPUT*

Permite imprimir un mensaje en la línea de comandos de ***Matlab*** y recuperar como valor de retorno un valor numérico o el resultado de una expresión tecleada por el usuario. Después de imprimir el mensaje, el programa espera que el usuario teclee el valor numérico o la expresión. Cualquier expresión válida de ***Matlab*** es aceptada por este comando. El usuario puede teclear simplemente un vector o una matriz. En cualquier caso, la expresión introducida es evaluada con los valores actuales de las variables de ***Matlab*** y el resultado se devuelve como valor de retorno.
Por ejemplo:

```
>> n = input('Teclee el número de ecuaciones')
```

Otra posible forma de esta función es la siguiente:

```
>> nombre = input('¿Cómo te llamas?','s')
```

En este caso el texto tecleado como respuesta se lee y se devuelve sin evaluar, con lo que se almacena en la cadena ***nombre***. Así pues, en este caso, si se teclea una fórmula, se almacena como texto sin evaluarse.

## *15.2 La función DISP*

Permite imprimir en pantalla un mensaje de texto o el valor de una matriz, pero sin imprimir su nombre. En realidad, ***disp*** siempre imprime vectores y/o matrices: las cadenas de caracteres son un caso particular de vectores.
Por ejemplo:

```
>> disp('El programa ha terminado');
El programa ha terminado
```

```
>> A = rand(4,4);
```

**>> disp(A);**
**A =**

| | | | |
|---|---|---|---|
| **0.8147** | **0.6324** | **0.9575** | **0.9572** |
| **0.9058** | **0.0975** | **0.9649** | **0.4854** |
| **0.1270** | **0.2785** | **0.1576** | **0.8003** |
| **0.9134** | **0.5469** | **0.9706** | **0.1419** |

| | | | |
|---|---|---|---|
| **0.8147** | **0.6324** | **0.9575** | **0.9572** |
| **0.9058** | **0.0975** | **0.9649** | **0.4854** |
| **0.1270** | **0.2785** | **0.1576** | **0.8003** |
| **0.9134** | **0.5469** | **0.9706** | **0.1419** |

## 16. 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 *instancias*). Se accede a los miembros o campos de una estructura por medio del ***operador punto*** (.), que une el nombre de la estructura y el nombre del campo.
Por ejemplo, la estructura ***alumno*** puede contener los campos ***nombre*** (una cadena de caracteres) y ***carnet*** (un número).

```
>> alu.nombre='Miguel'
alu =
nombre: 'Miguel'

>> alu.carnet=75482

alu =
nombre: 'Miguel'
carnet: 75482

>> alu
alu =
nombre: 'Miguel'
carnet: 75482
```

También puede crearse la estructura por medio de la función ***struct()***.
Por ejemplo:

```
>> al = struct('nombre', 'Ignacio', 'carnet', 76589)
al =
nombre: 'Ignacio'
carnet: 76589
```

Los ***nombres de los campos*** se pasan a la función ***struct()*** entre apóstrofos ('), seguidos del valor que se les quiere dar. Este valor puede ser la cadena vacía (") o la matriz vacía ([]).

Tambien pueden crearse vectores y matrices de estructuras. Por ejemplo:
**>> alum(10) = struct('nombre', 'Ignacio', 'carnet', 76589)**

Crea un vector de 10 elementos cada uno de los cuales es una estructura tipo ***alumno***. Sólo el elemento 10 del vector es inicializado con los argumentos de la ***función struct()***; el resto de los campos se inicializan con una cadena vacía o una matriz vacía.
***Matlab*** permite añadir un nuevo campo a una estructura en cualquier momento.
Por ejemplo, si se quiere añadir el campo ***edad*** a todos los elementos del vector ***alum***:
**>> alum(5).edad=18;**

## *16.1 Estructuras anidadas*

***Matlab*** permite definir ***estructuras anidadas***, es decir una estructura con campos que sean otras estructuras. Para acceder a los campos de la estructura más interna se utiliza dos veces el operador punto (.).
Por ejemplo:

```
>> clase = struct('curso','primero','grupo','A', 'alum', struct('nombre','Juan', 'edad', 19))
clase =
        curso: 'primero'
        grupo: 'A'
        alum: [1x1 struct]

>> clase.alum(1).nombre
ans =
        Juan
```

## *16.2 Funciones para operar con estructuras*

***Matlab*** dispone de funciones que facilitan el uso de estructuras.

| | |
|---|---|
| fieldnames() | Devuelve un vector de celdas con cadenas de caracteres que recogen los nombres de los campos de una estructura |
| isfield(ST,s) | Permite saber si la cadena **s** es un campo de una estructura **ST** |
| isstruct(ST) | Permite saber si **ST** es o no una estructura |
| rmfield(ST,s) | Elimina el campo **s** de la estructura **ST** |
| getfield(ST,s) | Devuelve el valor del campo especificado. |
| setfield(ST,s,v) | Asigna el valor **v** al campo **s** de la estructura **ST**. |

# 17. Matrices de celdas

Una matriz de celdas es una matriz cuyos elementos son cada uno de ellos una variable de tipo cualquiera. En una matriz ordinaria, todos sus elementos son números o cadenas de caracteres. Sin embargo, en una ***matriz de celdas***, el primer elemento puede ser un número; el segundo una matriz; el tercero una cadena de caracteres; el cuarto una estructura, etc. La matriz se crea utilizando ***llaves*** {}, es decir los valores asignados a cada elemento se definen entre ***llaves*** {...}.
Por ejemplo:

```
>> vc(1)={[1 2 3]}
vc =
        [1x3 double]

>> vc(2)={'mi nombre'}
vc =
        [1x3 double] 'mi nombre'

>> vc(3)={rand(3,3)}
vc =
        [1x3 double] 'mi nombre' [3x3 double]
```

También es posible crear el vector de celdas en una sola operación:

```
vcc = {[1 2 3], 'mi nombre', rand(3,3)}
vcc =
        [1x3 double] 'mi nombre' [3x3 double]
```

## *17.1 Funciones para operar con matrices de celdas*

***Matlab*** dispone de funciones que facilitan el uso de ***cell arrays***:

| | |
|---|---|
| cell(m,n) | Crea un *cell array* vacío de **m** filas y **n** columnas |
| celldisp(ca) | Muestra el contenido de todas las celdas de **ca** |
| cellplot(ca) | Muestra una representación gráfica de las distintas celdas |
| iscell(ca) | Indica si **ca** es una matriz de celdas |
| num2cell() | Convierte un vector numérico en un *cell array* |
| cell2struct() | Convierte un *cell array* en una estructura |
| struct2cell() | Convierte una estructura en un *cell array* |

# 18. Bifurcaciones y bucles

Las ***bifurcaciones*** permiten realizar una u otra operación según se cumpla o no una determinada condición.
Los ***bucles*** permiten repetir las mismas o análogas operaciones sobre datos distintos. Mientras que en C el "cuerpo" de estas sentencias se determina mediante llaves {...}, en ***Matlab***se utiliza la palabra ***end*** con análoga finalidad.

## *18.1 Sentencia IF*

- ***Bifurcación simple***

```
if condicion
        sentencias
end
```

- ***Bifurcación múltiple***

```
if condicion1
                bloque1
elseif condicion2
                bloque2
elseif condicion3
                bloque3
else bloque4
end
```

Donde la opción por defecto ***else*** puede ser omitida: si no está presente no se hace nada en caso de que no se cumpla ninguna de las condiciones que se han chequeado.

## *18.2 Sentencia SWITCH*

```
switch switch_expresion
        case case_expr1,
                        bloque1
        case case_expr2,
                        bloque2
        otherwise,
                        bloque3
end
```

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

## *18.3 Sentencia FOR*

- ***Bucle simple***

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

o bien,

```
for i=n:-0.2:1
        sentencias
end
```

- ***Bucle anidado***

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

La variable **j** es la que varía más rápidamente (por cada valor de **i**, **j** toma todos sus posibles valores)

### *18.4 Sentencia WHILE*

```
while condicion
        sentencias
end
```

### *18.5 Sentencia BREAK*

La sentencia ***break*** hace que se termine la ejecución del bucle ***for*** y/o ***while*** más interno de los que comprenden a dicha sentencia.

### *18.6 Sentencia RETURN*

La sentencia ***return***, incluida dentro del código de una función, hace que se devuelva inmediatamente el control al programa que realizó la llamada.

### *18.7 Sentencia CONTINUE*

La sentencia ***continue*** hace que se pase inmediatamente a la siguiente iteración del bucle ***for*** o ***while***, saltando todas las sentencias que hay entre el ***continue*** y el fin del bucle en la iteración actual.

### *18.8 Sentencia TRY…CATCH …END*

La construcción ***try...catch...end*** permite gestionar los errores que se pueden producir en tiempo de ejecución.

```
try
        sentencias1
catch
        sentencias2
end
```

En el caso de que durante la ejecución del bloque ***sentencias1*** se produzca un error, el control de la ejecución se transfiere al bloque ***sentencias2***.
Si la ejecución transcurriera normalmente, ***sentencias2*** no se ejecutaría nunca.

# 19. Archivos de comandos y funciones

Los archivos con extensión (***.m***) son ficheros de texto sin formato (ASCII) que constituyen el centro de la programación en ***Matlab***. Estos archivos se crean y modifican con un editor de textos cualquiera.
Existen dos tipos de ficheros ****.m***, los ***archivos de comandos*** (llamados *scripts* en inglés) y las ***funciones***.

## *19.1 Archivos de comandos (SCRIPTS)*

Los *scripts* 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 archivo que los contiene (sin la extensión).
Cuando se ejecuta el *script* desde la línea de comandos, las variables creadas pertenecen al espacio de trabajo base de ***Matlab.*** Las variables definidas por los archivos de comandos son variables del espacio de trabajo a partir del momento en el que se ejecuta el archivo. Al terminar la ejecución del *script*, dichas variables permanecen en memoria.
En los archivos de comandos conviene poner los puntos y coma (;) al final de cada sentencia, para evitar una salida de resultados demasiado cuantiosa. Un fichero ***.m*** puede llamar a otros ficheros ***.m***, e incluso se puede llamar a sí mismo de modo recursivo.
El comando ***echo*** hace que se impriman los comandos que están en un *script* a medida que van siendo ejecutados.

| | |
|---|---|
| echo on | Activa el ***echo*** en todos los ***scripts*** |
| echo off | Desactiva el ***echo*** |

## *19.2 Funciones*

***Matlab*** tiene un gran número de funciones incorporadas. Algunas son ***funciones intrínsecas***, es decir funciones incorporadas en el propio código ejecutable del programa. Estas funciones son particularmente rápidas y eficientes.
Existen además funciones definidas en archivos ***.m*** y ***.mex*** que vienen con el propio programa o que han sido aportadas por usuarios. Un archivo ***.m*** puede llamar a otros archivos ***.m***, e incluso puede llamarse a sí mismo de forma recursiva.
Las funciones definidas en archivos ***.m*** 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).

Por ejemplo, la ***primera línea*** de un fichero llamado ***name.m*** 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.

Por lo tanto, el concepto de función en ***Matlab*** 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***
- ***Argumento***.

Los ***argumentos*** son los ***datos*** de la función y los ***valores de retorno*** sus ***resultados.***

***Respecto al nombre de la función:*** Una función se llama utilizando su nombre en una expresión o utilizándolo como un comando más. Los nombres de las funciones no son palabras reservadas del lenguaje.

***Respecto al valor de retorno:*** Una función pueden tener valores de retorno matriciales múltiples. Por ejemplo, en el comando: `[V, D] = eig(A),` la función ***eig()*** calcula los autovalores y los autovectores de la matriz cuadrada **A**. Los autovectores se devuelven como columnas de la matriz **V**, mientras que los autovalores son los elementos de la matriz diagonal **D.** Los valores de retorno se recogen entre corchetes, separados por comas (siempre que haya más de uno). Puede haber funciones sin valor de retorno.
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.
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.

***Respecto a los argumentos:*** Se podría decir que los argumentos de las funciones de siempre se pasan por valor, nunca por referencia. Las funciones nunca devuelven modificadas las variables que se pasan como argumentos, a no ser que se incluyan también como valores de retorno. Si el usuario las modifica dentro de la función, previamente se sacan copias de esas variables (se modifican las copias, no las variables originales). Una característica de 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. Los argumentos pueden ser expresiones y también llamadas a otra función y se definen entre paréntesis separados por comas. No hace falta poner paréntesis si no hay argumentos.

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***).

## *19.3 Variables globales, locales y persistentes*

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. Las variables globales son visibles en todas las funciones (y en el espacio de trabajo base o general) que las declaran como tales.
Las variables se declaran como globales utilizando la palabra ***global*** seguida de los nombres separados por blancos, como por ejemplo:
>> **global VARIABLE1 VARIABLE2**

Las ***variables persistentes*** son variables locales de las funciones (pertenecen al espacio de trabajo de la función y sólo son visibles en dicho espacio de trabajo), que ***conservan su valor*** entre distintas llamadas a la función. Por defecto, las variables locales de una función se crean y destruyen cada vez que se ejecuta la función. Las variables persistentes se pueden definir en funciones, pero no en *scripts*.
Las variables se declaran como persistentes utilizando la palabra ***persistent*** seguida de los nombres separados por blancos, como por ejemplo:
>> ***persistent*** **VARIABLE1 VARIABLE2**

Las variables ***persistent*** se inicializan a la matriz vacía [] y permanecen en memoria hasta que se hace ***clear*** de la función o cuando se modifica el archivo ***.m***

## *19.4 Funciones con un numero variable de argumentos y valores de retorno*

***Matlab*** dispone de una nueva forma de pasar a una función un número variable de argumentos por medio de la variable ***varargin***, que es un *vector de celdas* que contiene tantos elementos como sean necesarios para poder recoger en dichos elementos todos los argumentos que se hayan pasado en la llamada. No es necesario que ***varargin*** sea el único argumento, pero sí debe ser el último de los que haya, pues recoge todos los argumentos a partir de una determinada posición.

De forma análoga, una función puede tener un número indeterminado de valores de retorno utilizando ***varargout***, que es también un *vector de celdas* que agrupa los últimos valores de retorno de la función. Puede haber otros valores de retorno, pero ***varargout*** debe ser el último. El *vector de celdas* ***varargout*** se debe crear dentro de la función y hay que dar valor a sus elementos antes de salir de la función.

Por ejemplo:

```
function varargout=atan3(varargin)

        if nargin==1
                rad = atan(varargin{1});
        elseif nargin==2
                rad = atan2(varargin{1},varargin{2});
        else
                disp('Error: más de dos argumentos')
                return
        end

varargout{1}=rad;
if nargout>1
        varargout{2}=rad*180/pi;
end
```

### *19.5 SUB-Funciones*

Son funciones adicionales definidas en un mismo archivo ***.m***, con nombres diferentes del nombre del archivo (y del nombre de la función principal) y que ***las sub-funciones sólo pueden ser llamadas por las funciones contenidas en ese archivo***, resultando "invisibles" para otras funciones externas.
Por ejemplo:

```
function y=mi_fun(a,b)
        y=subfun1(a,b);

function x=subfun1(y,z)
        x=subfun2(y,z);

function x=subfun2(y,z)
        x=y+z+2;
```

### *19.6 Funciones privadas*

Las funciones privadas (***private***) son funciones que no se pueden llamar desde cualquier otra función, aunque se encuentren en el ***path*** o en el directorio actual. ***Sólo ciertas funciones están autorizadas a utilizarlas***. Las funciones privadas se definen en sub-directorios que se llaman ***private*** y sólo pueden ser llamadas por funciones definidas en el directorio padre del sub-directorio ***private***.
En la búsqueda de nombres que hace ***Matlab*** cuando encuentra un nombre en una expresión, las funciones privadas se buscan inmediatamente después de las sub-funciones, y antes que las funciones de tipo general.

## *19.7 Función de función*

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, resolver una ecuación no lineal, o integrar numéricamente una ecuación diferencial ordinaria (problema de valor inicial).
Por ejemplo, se define la función prueba y después se guarda el archivo con el nombre de *prueba.m*

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

El archivo *prueba.m es una* nueva función que puede ser utilizada como cualquier otra de las funciones de ***Matlab.***
Por ejemplo:

```
>> x=-1:0.1:2;
>> plot(x,prueba(x))
```