

INTRODUCCIÓN A LA PROGRAMACIÓN EN MATLAB

Expresiones lógicas.

MATLAB es capaz de respondernos si ciertas expresiones son verdaderas o falsas. Por ejemplo, podemos formar estas expresiones comparando dos números. Si la comparación es cierta, MATLAB nos responderá con un uno y si es falsa con un cero.

```
>> 3<4
ans=
```

1

```
>> 4<3
ans=
```

0

La siguiente tabla nos muestra el formato de las comparaciones que podemos realizar con MATLAB.

<	menor que
<=	menor o igual que
>	mayor que
>=	mayor o igual que
==	igual que
~=	no igual que

Si comparamos dos matrices del mismo orden, MATLAB nos responde con la matriz que resulta al comparar elementos con mismos índices.

```
>> [1 2; 2 4] >= [1 2; 3 4]
ans=
```

1 1

0 1

Si comparamos una matriz con un número, el resultado es una matriz que muestra el valor lógico de la comparación de cada elemento de la matriz con dicho número.

```

>> [1 2; 3 4] >= 2
ans =

     0     1
     1     1

```

Ejercicio: Pregunta a MATLAB si π es mayor que el número e .

Ejercicio: Pregunta a MATLAB qué elementos de la matriz mágica de orden 6 son múltiplos de 3. Utiliza para este ejercicio la orden **rem**.

Ejercicio: Compara dos cadenas de caracteres de la misma longitud, por ejemplo 'hola' y 'majo'. Explica el resultado.

Ejercicio: Considera el vector $x=[1\ 2\ 4\ 5\ 7\ 9\ 11]$. >Qué se obtiene al escribir $x(x > 5)$? Explica el resultado.

Para los valores lógicos hay definidas tres operaciones fundamentales que en informática reciben los nombres AND, OR y NOT:

$p \& q$ Operación AND. Si las expresiones **p** y **q** son verdaderas devuelve el valor uno y en caso contrario cero.

$p | q$ Operación OR. Si alguna de las afirmaciones **p** o **q** son verdad devuelve uno, en caso contrario devuelve cero.

$\sim p$ Operación NOT. Cambia el valor lógico de **p**.

Seguidamente mostramos algunos ejemplos con dichas operaciones:

```

>> p=[1 1 0 0]; q=[1 0 1 0];
>> [p&q; p|q; ~p]
ans =

     1     0     0     0
     1     1     1     0
     0     0     1     1

```

```

>> 3>2 | 1<0
ans =

```

Veamos seguidamente funciones definidas para los valores lógicos. La instrucción **any** aplicada a un vector devuelve el valor uno si algún elemento del vector vale uno y cero

en caso contrario. La instrucción **all** aplicada a un vector devuelve el valor uno cuando todos los elementos del vector valen uno y devuelve cero en caso contrario.

```
>> any([1 0 0 1])
ans=
```

1

Si se aplica el comando **any** o **all** a una matriz, MATLAB devuelve un vector fila que resulta de efectuar a cada uno de los vectores columnas la operación **any** o **all**, respectivamente.

Ejercicio: Pregunta a MATLAB si la matriz mágica de orden 20 tiene algún elemento que valga cero.

El comando **exist** responde si una variable está definida en el *Workspace*. Notemos que el argumento de **exist** debe ser una cadena con el nombre de la variable.

```
>> a=2; exist('a')
ans=
```

1

Otros comandos similares a **exist** son:

- isempty** Responde si una matriz es vacía.
- isstr** Responde si una variable es una cadena.
- isnan** Responde si una variable surge al calcular una indeterminación.
- finite** Responde si una variable es un número finito.

Ejercicio: Define las matrices **h** y **a** mediante las instrucciones **h=['hola' 33]; a=[3 'adios' 3];**. Pregunta si se trata de una cadena o de una variable numérica. Observa el resultado que se muestra por pantalla al teclear **h**, **a**.

Ejercicio: >Es la matriz **eye(0,7)** una matriz vacía?

Bucles.

Es habitual que al resolver cierto problema tengamos que repetir varias veces un cierto número de instrucciones, como ocurre, por ejemplo, al programar el método de Newton para aproximar raíces. Para realizar esta operación de forma cómoda los lenguajes de programación disponen de ciertas estructuras que reciben el nombre de bucles. En MATLAB, una de las principales instrucciones para generar bucles es **for ... end**. El formato de la misma es:

for *var* = *matr*

Bloque de instrucciones

end

y su funcionamiento es desarrollado en las siguientes líneas. En primer lugar, MATLAB asigna a la variable *var* la primera columna de la matriz *matr* y efectúa el bloque de instrucciones. En segundo lugar, MATLAB asigna a la variable *var* la segunda columna de la matriz *matr* y vuelve a ejecutar el bloque de instrucciones, hasta que, siguiendo este procedimiento, se llega a la última columna de dicha matriz. Un ejemplo de esta instrucción es:

```
>>
for i=[1 2; 3 4]

    [i(1) i(2) i(1)+i(2)]
end
ans=

    1 3 4

ans=

    2 4 6
```

Señalemos que, tras el bucle, la variable utilizada como *var*, en nuestro caso *i*, toma el último valor que se le ha asignado durante el bucle. En nuestro ejemplo la variable *i* ya no sería la unidad imaginaria.

```
>>
i
i=

    2

    4
```

Aunque hemos visto que *var* puede ser una matriz, lo habitual es que no sea más que un contador del número de iteraciones. En esta situación escribimos:

```
>>
for i=1:n
```

donde *n* es el número de iteraciones.

Veamos un ejemplo de utilización de los bucles. En el nos planteamos el cálculo de los cien primeros términos de la sucesión de Fibonacci definida por

$$z_{n+1} = z_n + z_{n-1}, \text{ para } n \geq 2, \quad z_1 = 1 \text{ y } z_2 = 1.$$

Para ello realizamos el siguiente programa.

```

>> z(1)=1; z(2)=1;
>> for n=2:99

z(n+1)=z(n)+z(n-1);

end
>> z

```

Ejercicio: Calcule los 30 primeros términos de la sucesión $z_{n+1} = z_n^2 - z_{n-1}^2$ para $n \geq 2$ $z_1 = 1$ $z_2 = 1$, con $z_1 = 1$ y $z_2 = 1$.

Ejercicio: Aplica el método de Newton para obtener una aproximación de $\sqrt{2}$.
 $x^2 - 2 = 0$

Recordemos que el método de Newton para la ecuación no es más que la sucesión definida por recurrencia

$$x_{n+1} = \frac{x_n^2 + 2}{2x_n}, \quad \text{para } x_0 \text{ dado.}$$

Para el ejercicio toma $x_0 = 2$ y realiza diez iteraciones.

Ejercicio: El método de Euler proporciona una aproximación a la solución de un problema de valor inicial para una ecuación diferencial. Para el problema

$$\begin{cases} y'(t) = y(t), \\ y(0) = 1, \end{cases}$$

obtenemos la siguiente sucesión definida por recurrencia a partir de $t_0 = 0$ e $y_0 = 1$,

$$\begin{cases} t_{n+1} = t_n + h, \\ y_{n+1} = (1 + h)y_n, \end{cases}$$

Donde y_n es una aproximación de los valores $y(t_n)$ y h es el paso del método. Programa este método para el paso $h = 0.1$ en el intervalo $[0, 1]$ y compara la gráfica de los puntos (t_n, y_n) con la de la función $y(x) = e^x$, que es la solución verdadera.

Solución:

```
>> h=0.1; t(1)=0; y(1)=1;
>> for i=2:1/h

t(i)=t(i-1)+h;

y(i)=(1+h)*y(i-1);
end
>> plot(t,y,t,exp(t))
```

MATLAB permite anidar varias instrucciones **for**. Al escribir **end** se cierra el último bucle definido. Para mayor claridad a la hora de escribir el programa conviene variar el margen para observar claramente qué **end** corresponde a cada **for**. Como ejemplo de anidamiento de bucles representamos, en una gráfica, 9nueve circunferencias de radio unidad con centros en los puntos (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1), mediante el siguiente programa:

```
>> t=0:0.1:2*pi;
>> k=1;
>> for i=-1:1

for j=-1:1

X(:,k)=(cos(t)+i)';

Y(:,k)=(sin(t)+j)';

k=k+1;

end
```

```
end
>>
plot(X,Y)
```

Ejercicio: Crea, usando dos bucles **for** anidados, la matriz $\mathbf{H}=(h_{ij})$ de orden 10×10 definida por la expresión $h_{ij} = i^2 j$.

Otra forma de crear bucles es mediante la instrucción **while... end**, cuya estructura es:

```
while valor-lógico
Bloque de instrucciones
end
```

Si el *valor-lógico* es uno se ejecuta el bloque de instrucciones y en caso contrario no, el proceso se repite sucesivamente hasta que *valor-lógico* sea nulo.

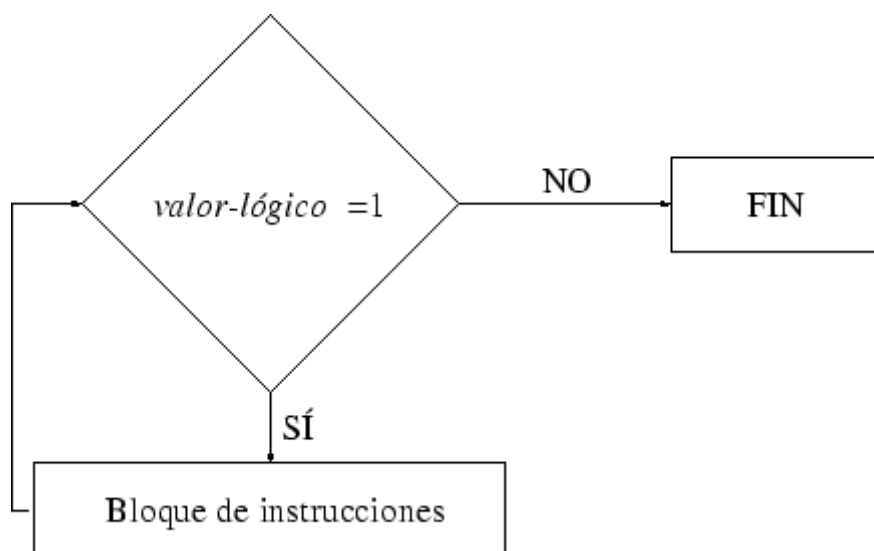

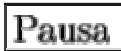




Figura 3.1: Diagrama de flujo de la instrucción **while**

Ejecutemos las instrucciones:

```
>>
while 1, disp('hola'), end
```

Observamos que en la pantalla aparecen ``holas" sin parar y nunca termina por aparecer el símbolo ``>>". Ello es debido a que hemos creado un bucle infinito en el que continuamente se ejecuta el bloque de instrucciones, ya que *valor-lógico* siempre es

uno. Para detener la ejecución de esta instrucción pulsamos  +  o  + .

Como ejemplo de la orden **while** calculamos el primer factorial mayor que 1000.

```
>> n=1;
>> while prod(1:n) < 1000
n=n+1;
end
>> n
```

Ejercicio: Calcula el primer término de la sucesión de Fibonacci que supera 10^6 .

Solución:

```
>> z=1; z1=1;
>> while z<=1e6
z2=z;
z=z+z1;
z1=z2;
end
>> z
```

Ejercicio: El método de Newton para resolver la ecuación $x^2 - 3 = 0$ nos proporciona la sucesión por recurrencia

$$x_{n+1} = \frac{x_n^2 + 3}{2x_n}, \quad \text{para } x_0 \text{ dado.}$$

Un test de parada típico es exigir que dos términos consecutivos disten entre sí menos que cierta cantidad ε . Aproxima $\sqrt{3}$ con $\varepsilon = 10^{-6}$ y $x_0 = 2$.

Ejercicio: Calcula el primer término de la sucesión definida por $x_{n+1} = x_n + n$, con $x_0 = 0$ que sea mayor que 1000.

Decisiones.

En MATLAB la instrucción que sirve para tomar decisiones es **if**. Su formato es

```

if valor-lógico 1
Bloque de instrucciones 1
elseif valor-lógico 2
Bloque de instrucciones 2
⋮
elseif valor-lógico n
Bloque de instrucciones n
else
Bloque de instrucciones final
end

```

En esta estructura se ejecuta solamente el primer bloque de instrucciones cuyo valor lógico sea uno. Si ninguno de los valores lógicos que aparecen son uno entonces MATLAB ejecuta el bloque de instrucciones final. La figura [3.2](#) muestra el funcionamiento de esta instrucción.

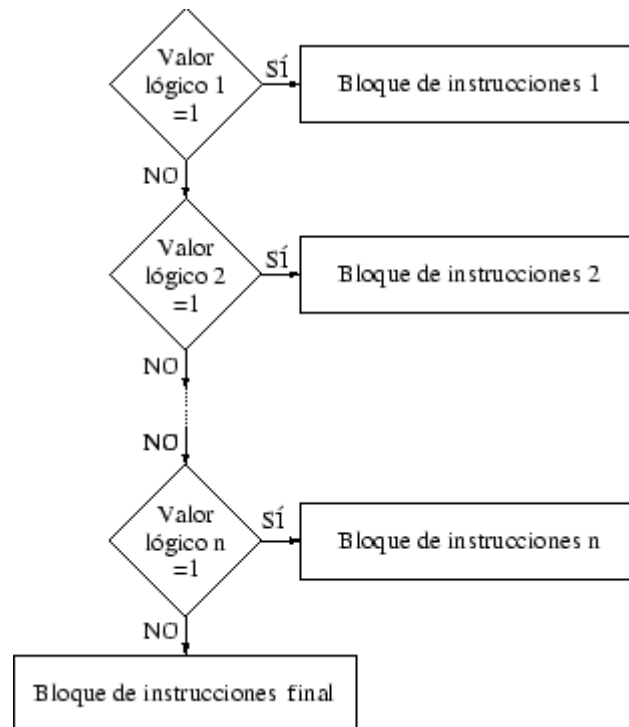


Figura 3.2: Diagrama de flujo del comando **if**

Un ejemplo trivial de funcionamiento de la instrucción **if** es

```

>> if 0 i=1, elseif 0 i=2, else i=3, end
i=

    3
  
```

Seguidamente mostramos un ejemplo no tan trivial en el que construimos un vector formado por los divisores de 748.

```

>> x(1)=0; j=0;
>> for i=1:748

        if rem(748,i)==0

            j=j+1;

            x(j)=i;

        end

    end
  
```

Ejercicio: Calcula todos los divisores de 687.

Ejercicio: Genera un vector con los números primos entre 1 y 200.

Solución:

```
i=1;
for j=2:200

    if all(rem(j*ones(1,j-2),2:(j-1)))~=0)

        x(i)=j;

        i=i+1;

    end

end
```

Archivos .m

Llegados a este punto somos capaces de hacer pequeños programas, pero cada vez que deseemos repetir alguno de ellos tendremos que escribirlo de nuevo. Si el programa tiene cierto tamaño, la tarea puede ser ardua. Por estos motivos, MATLAB permite que se puedan almacenar los programas en ficheros de textos y, posteriormente, ser llamados para su ejecución.

Veamos un ejemplo, para ello abrimos un editor de texto (por ejemplo, el *Bloc de Notas* de Windows) y escribimos el siguiente programa que calcula el n-ésimo término de la sucesión de Fibonacci:

```
f1=1; f2=1;
for i=3:n

    f3=f2+f1;

    f1=f2;

    f2=f3;
end
```

Guardaremos este archivo en la unidad de disco flexible (A:) con el nombre **fibonacci.m**. Es necesario escribir el nombre del programa terminado en **.m** para que MATLAB reconozca que se trata de un programa escrito en su "lenguaje". Además, para ejecutar

los ficheros **.m** es necesario que éstos se encuentren en nuestro directorio de trabajo o en otros directorios que han de especificarse previamente a MATLAB. Para cambiar el directorio de trabajo a la unidad A: escribimos

```
>>
!A:
```

El signo ```!` al inicio de la línea sirve para indicar que lo que siguen son órdenes del sistema operativo.

Una vez efectuadas estas operaciones, al escribir la orden **fibo** se ejecutan todas las instrucciones escritas en dicho archivo.

```
>>
n=10;
>>
fibo
>>
f2
f2=
```

55

Al ejecutarse el programa, la variable **f2** almacena el décimo término, y **f1** el noveno, mientras que la variable **i** vale 10. Este hecho puede resultar incómodo, pues exige tener en cuenta qué variables utiliza el programa para evitar que alguna de las almacenadas en el *Workspace* cambie de valor ``inesperadamente". Para solucionar ese problema, MATLAB posee la instrucción **function** que convierte a las variables del programa en ``locales", es decir, cuando ejecutemos el programa las variables que utilicemos en él no se almacenarán en el *Workspace* y, por lo tanto, no modifican ninguna de las ya existentes. El formato de esta instrucción es

function *arg-sal*= *nombre-programa* (*arg-ent*)

Mediante *arg-ent* se proporcionan valores a la función y una vez ejecutadas las instrucciones, ésta devuelve ciertos valores mediante *arg-sal*. No es necesario asignar valores de entrada o salida en todos los programas. En el programa anterior el valor de entrada era la variable **n** que indicaba el número de términos a calcular, mientras que el valor de salida que nos interesaba era **f2**. Por tanto, añadimos al programa anterior como primera línea:

```
function f2=fibo(n)
```

y guardamos de nuevo el fichero. Al trabajar bajo MS-DOS, el nombre del programa puede tener, a lo sumo, ocho caracteres y debe coincidir con el nombre del fichero.

Ejecutamos, tras borrar todas las variables con **clear**,



```
fibo(10)  
ans=
```

55

Ejercicio: Observa que la variable **i** no ha cambiado de valor y las variables **f1**, **f2** y **f3** no aparecen en el *Workspace*.

El programa que hemos creado queda almacenado en el disco y puede que cuando lo volvamos a utilizar no recordemos exactamente para qué servía. Para evitar este problema es útil añadir ciertos comentarios en el archivo. Dichos comentarios se sitúan en líneas precedidas por el signo ``%".

Tras incluir comentarios oportunos el programa resulta:

```
function f2=fibo(n)  
% Este programa calcula el término n-ésimo de la % sucesión de  
Fibonacci.  
% fibo(n) devuelve el término n de la sucesión de  
% Fibonacci.  
  
% f1 y f2 son los últimos términos de la sucesión.  
f1=1; f2=1;  
for i=3:n  
  
    f3=f2+f1;  
  
    f1=f2;  
  
    f2=f3;  
end
```

Las primeras líneas de comentario, hasta la primera línea en blanco o hasta que comienzan las instrucciones, se pueden examinar desde MATLAB sin tener que abrir el archivo con el editor. Para ello basta utilizar la orden **help**.



```
help fibo
```

```
Este programa calcula el término n-ésimo de la  
sucesión de Fibonacci.  
fibo(n) devuelve el término n de la sucesión de  
Fibonacci.
```

NOTA: Puede ser que en vez de vocales acentuadas aparezcan otros caracteres.

Además, desde MATLAB se puede examinar el programa mediante la instrucción **type**.



```
type fibo
```

Ejercicio: Examina con **type** los programas de las instrucciones **linspace**, **logspace**, **mean** y **sum**.

La última instrucción del ejercicio anterior, **sum**, es un ejemplo de instrucción interna, es decir, definida en el mismo programa MATLAB, mientras que el resto de instrucciones se encuentran en archivos **.m** en el directorio H:\ MATLAB\MATLAB, cuando trabajamos en el centro de cálculo.

Ejercicio: Haz un fichero **.m** que contenga un programa que calcule los números primos menores que una cantidad dada.

Ejercicio: Programa la instrucción **issim** que responda si una matriz es simétrica.

Ejercicio: Repite el programa anterior para matrices normales. Recuerda que una matriz se dice normal si $A^*A = AA^*$, donde A^* es la transpuesta conjugada.

Otras órdenes de programación

Órdenes de entrada/salida.

Cuando se desea mostrar el valor de alguna variable o expresión en la ejecución de un programa es conveniente utilizar la orden **disp** para evitar que aparezca además el nombre de la variable, por ejemplo, si deseamos que en el programa **fibo** vayan apareciendo los términos de la sucesión según los calcula, bastaría insertar como penúltima línea

```
disp(f2)
```

MATLAB ofrece la posibilidad de asignar un valor a una variable desde el teclado en el transcurso de la ejecución de un programa. Con **var= input(' mensaje')** se muestra en pantalla la cadena de caracteres *mensaje* y aparece un cursor parpadeante en la misma hasta que introduzcamos una expresión, que se convertirá en el valor de la variable **var**.

Para mostrar el funcionamiento de la variable **input**, programamos un juego bastante simple que consiste en averiguar un número entre el uno y el diez.

```
function juego
% Programa-juego para averiguar un número aleatorio entre
% uno y diez.

% al=número aleatorio.
% ne=número escogido.
al=0;
while al==0
al=ceil(10*rand(1,1));
end
ne=0;
while(ne ~= al)
```

```

ne=input('escriba un número entre 1 y 10>>');

if ne>a1

b='mayor';

elseif ne<a1

b='menor';

else

b='igual';

end

disp(['el número ' num2str(ne) ' es ' b]);
end
disp('Bien hecho.')

```

Se puede observar que hemos convertido la variable numérica **ne** en una cadena. Si no la hubiésemos convertido, MATLAB no mostraría el valor de **ne**, sino el caracter correspondiente en el código ASCII.

Variables nargin y narginout.

Al ejecutar un programa se asigna a la variable **nargin** el número de argumentos de entrada y a la variable **nargout** el número de argumentos de salida. Esto permite crear programas con diferentes número de argumentos de entrada y de salida. Por ejemplo, podemos crear una función que calcule, mediante el método de Newton, la raíz cuadrada de cierto número con una precisión deseada. De ese modo, necesitamos una función que tenga dos argumentos de entrada: uno para el número, **x**, y otro para la precisión, **pre**. De todos modos, es posible mediante el uso de **nargin**, construir la función para que se pueda llamar con sólo un argumento, correspondiente al número **x**, y tomar como precisión un valor fijo, por ejemplo, 10^{-2} . Las siguientes instrucciones corresponden a la función así definida

```

function [r,nit]=raiz(x,pre)
if nargin==1

pre=1e-2;
end
r=x;
r1=0;
nit=0;
while abs(r-r1)>pre

```

```

r1=r;

r=(r*r+x)/(2*r);

nit=nit+1;
end

```

Como argumentos de salida tenemos el valor de la raíz, **r** y el número de iteraciones del método, **nit**. Escribiendo

```

>> [r,nit]=raiz(3,1e-10)

```

Obtenemos la raíz de 3 con una precisión de 10^{-10} y el número de iteraciones necesarias para alcanzarla.

Ejercicio: Modifica el programa **juego** para que en ausencia de parámetro calcule un número aleatorio entre uno y diez, pero con un parámetro **n** lo haga entre uno y **n**.

Ejercicio: Crea un archivo con el método de Euler para:


$$\begin{cases} \frac{dy}{dt}(t) = y(t), \\ y(0) = 1, \end{cases}$$

ya visto en la sección 1.2, para que cuando no haya argumentos de salida represente una gráfica, si hay uno, dé por salida una matriz en la cual una columna esté formada por los

valores de t_n y la otra por los valores y_n y si hay dos argumentos de salida, en uno muestre los valores de t_n y en el otro los de y_n .

Órdenes de ruptura.

Puede ser que al ejecutar un programa, si se da una determinada circunstancia, queramos parar la ejecución del mismo definitivamente o hasta que pase cierto intervalo de tiempo. Las órdenes **break**, **error**, **return** y **pause** permiten realizar esto, bajo diferentes circunstancias y con los resultados que se detallan.

La orden **break** para la ejecución de todos los ficheros **.m** que se estén ejecutando en ese momento y regresa a MATLAB con el símbolo  y un mensaje de ruptura.

La orden **return** detiene la ejecución del fichero **.m** donde se halle esta instrucción, es decir, si el fichero donde se encuentra la orden **return** había sido llamado por uno anterior, continua la ejecución de éste.

La instrucción **pause(x)** realiza una pausa de **x** segundos antes de ejecutar la siguiente orden de programa.

La orden **error('x')** detiene el desarrollo de la ejecución y muestra el mensaje **x** en pantalla acompañado de un mensaje de error.

Ejercicio: Modifica el fichero **juego.m** para que se produzca un mensaje de error cuando el argumento de entrada del programa sea un número negativo.

Instrucciones eval y feval.

Al programar ciertos algoritmos, como por ejemplo métodos numéricos, nos damos cuenta que si queremos aplicarlos a distintas funciones hemos de editarlos y cambiar el nombre a la función cada vez que esta aparezca en el programa. Esto se puede simplificar en MATLAB si utilizamos la orden **feval**, que se escribe con el siguiente formato:

feval('función', arg1, ... , argn)

y equivale a

función (arg1, ... ,argn).

La utilidad principal de esta instrucción, que puede ser observada en el ejemplo posterior, radica en que podemos introducir el nombre de una función en un programa como una cadena de caracteres, mediante los argumentos de entrada de dicho programa.

Como muestra implementamos el método de Euler con paso fijo **h** para el problema de valores iniciales

$$\begin{cases} y'(t) = f(t, y(t)), \\ y(t_0) = y_0, \end{cases}$$

Donde f es una función cualquiera. Recordemos que el método de Euler nos suministra la sucesión definida por recurrencia

$$\begin{cases} t_{n+1} = t_n + h, \\ y_{n+1} = y_n + hf(t_n, y_n), \end{cases}$$

a partir de los valores t_0 e y_0 .

```

function [t,y]=euledo(fun,t0,t1,y0,h)
% La orden euledo permite resolver numéricamente el
% problema de valores iniciales
%  $y'(t)=fun(t,y(t))$ ,
%  $y(t_0)=y_0$ ,
% la orden euledo tiene el siguiente formato
% [t y]=euledo(fun,t0,t1,y0,h)
% donde
%     t0           es el punto donde damos la condición inicial.
%     t1           es el extremo del intervalo donde calculamos la
% solución.
%     t            es un vector que almacena los puntos de la
% partición del intervalo [t0,t1].
%     y            almacena las aproximaciones a la solución en
% cada punto de la partición.
%     y0           valor de y en la condición inicial.
%     h            paso del método de Euler.
%
if ((t1-t0)*h<=0)|( ~isstr(fun))

    error ('Error en los datos.')
end
t(1)=t0;
it=1;
y(1)=y0;
while (t1-t(it))*h>0

    y(it+1)=y(it)+h*feval(fun,t(it),y(it));

    t(it+1)=t(it)+h;

    it=it+1;
end

```

La instrucción **eval** es similar a **feval**. Su formato es

eval(' str ')

y el resultado que se tiene al ejecutar dicha orden, es el que se obtiene al ejecutar la instrucción *str*. Por ejemplo, la instrucción

```

>> eval('a=2');

```

es equivalente a

```

>> a=2;

```

y si a la variable **fun** se le asigna una cadena con el nombre de una función, por ejemplo **'atan2'**, entonces

```

>> eval([fun ' (3,a) ']);

```

es equivalente a

```
>> feval(fun,3,a);
```

Archivos de datos.

Los resultados que se obtienen al ejecutar una determinada instrucción o programa son perdidos al acabar la sesión de MATLAB. Si queremos almacenarlos en un archivo, para después utilizarlos en otras sesiones necesitaremos usar la instrucción **save**. Por ejemplo, al teclear

```
>> A=rand(3,3)
```

Obtenemos una matriz aleatoria y podemos almacenarla en el fichero *alea.mat* mediante la instrucción

```
>> save alea.mat A
```

Para recuperar dicha matriz en otra sesión basta teclear

```
>> load alea.mat
```

y tendremos de nuevo la matriz **A** definida en el *Workspace*.

Ejercicio: Genera una matriz mágica y guárdala en un archivo **.mat**. Una vez realizadas estas operaciones borra el *Workspace* y vuelve a recuperar la matriz anterior.

Si se teclaea

```
>> save espacio.mat
```

Sin determinar las variables a almacenar, en el archivo *espacio.mat* se guardan todas las variables del *Workspace*.

Los ficheros **.mat** son un tipo particular de archivos que sólo pueden ser leídos por MATLAB. Para almacenar la matriz **A** en un fichero de texto, tecleamos la instrucción

```
>> save alea.dat A /ascii
```

o

```
>> save alea.dat A -ascii
```

Ejercicio: Observa el fichero de texto creado. Borra el *Workspace* y carga el fichero **alea.dat**.

Ejercicio: Busca ayuda sobre la instrucción **diary**. Utiliza la instrucción para guardar una sesión de trabajo.