3 Programación en Matlab

Una de las características que dan más potencia y versatilidad a Matlab es su lenguaje de programación. Matlab, aparte de ser una potente herramienta de cálculo, incorpora un completo interprete de programación utilizando un lenguaje propio.

La mayor parte de las funciones proporcionadas por Matlab están organizadas en paquetes adicionales (***Toolboxes***), que no son más que programas escritos en código Matlab que extienden la funcionalidad básica de la aplicación. Adicionalmente se puede enlazar código escrito en otros lenguajes como *C*, *C++*, *Java*,… pero en este caso requiere del uso de un compilador especial, quedando este tipo de programación fuera del alcance de este libro.

El lenguaje Matlab es un lenguaje interpretado de alto nivel, el cual permite organizar y estructurar el código permitiendo la implementación de algoritmos complejos o tareas repetitivas. Al tratarse de un lenguaje interpretado, Matlab realiza una comprobación del código, chequeando cualquier error en tiempo de ejecución. En cambio, como en todos los lenguajes interpretados, la ejecución es más lenta frente a una implementación compilada como en los lenguajes C o C++.

En cualquier caso, disponemos de todas las estructuras de control, acceso a ficheros e interfaz de usuario habituales en los lenguajes de programación de alto nivel, unido al potente cálculo matemático tanto escalar como matricial, representación gráfica y gestión transparente de variables proporcionado por Matlab. Adicionalmente, en las versiones actuales de Matlab, junto a la programación funcional clásica, podemos realizar programación orientada a objetos, aunque este aspecto queda fuera del alcance de este tutorial.

Se pueden distinguir dos tipos de programas en Matlab:

- **Scripts**: se trata de una secuencia de comandos Matlab tal como los introducimos en la consola, pero pudiendo incorporar estructuras de control y bucles. Se caracteriza por que accede al espacio de variables global *Workspace*. Se almacenan en un fichero con extensión '`.m`', utilizando el nombre del fichero para ejecutar el script.

- **Funciones**: se trata de un código que tiene un espacio de variables local aislado, siendo preciso declarar un conjunto de variables de entrada y de salida. Un ejemplo de uso de una función en la consola es: '`>> a=sin(x)`'. La función tiene un nombre '`sin`' y entre paréntesis indicamos las variables de entrada '`(x)`', el valor devuelto es asignado utilizando el signo '`=`' a la izquierda del nombre de la función. Las funciones son útiles cuando queremos encapsular una tarea sobre un número limitado de datos devolviendo un resultado. Igual que en el caso anterior, se almacenan en un fichero con extensión '`.m`', y el nombre del fichero debe coincidir con el nombre de la función, aunque en este caso las primeras líneas del fichero tienen una sintaxis especial para indicar las variables de entrada y salida.

En ambos casos, Matlab debe poder localizar los ficheros con el código para poder ejecutarlos, por lo que o bien están ubicados en el directorio actual indicado en la consola, o bien debemos incluir el directorio donde estén ubicados en la configuración de Matlab **`setpath`** (ver capítulo 1).

La sintaxis del código que podemos implementar en ambos casos es el mismo y solo se diferencian por el alcance de las variables que pueden utilizar. Por ello vamos a detallar los diferentes tipos de variables en Matlab según su alcance.

3.1 Espacios de variables

El lenguaje Matlab no es un lenguaje tipado, lo que significa que para usar una variable no es necesario declararla previamente indicando el tipo de datos que va almacenar. La gestión de las variables en memoria es manejada por Matlab y el usuario o programador no tiene que preocuparse por su asignación o su liberación.

Asimismo, las variables pueden cambiar su tipo de forma dinámica durante la ejecución según el resultado de una expresión. Por ejemplo una misma variable puede almacenar una matriz en una parte de la ejecución, y esa misma variable puede almacenar posteriormente una cadena de caracteres. En aquellas expresiones que utilicen variables con datos de diferente tipo, Matlab puede realizar conversiones automáticas para ejecutar la expresión. En todo el proceso, la asignación de memoria o su liberación es transparente al usuario.

En cualquier caso Matlab sí que se diferencia entre dos ámbitos o espacios de variables:

- **Espacio global** (*Workspace*): las variables almacenadas en el Workspace tienen alcance global y cualquier sentencia o script puede acceder a ellas (lectura y escritura). Estas variables son visibles en la ventana *Workspace* de la aplicación Matlab tal como se ha visto en capítulos previos. Las funciones pueden acceder a estas variables pero requiere de un código específico para indicar al sistema que queremos leer o escribir en una variable del W*orkspace*. Estas variables son persistentes (se conservan mientras la aplicación esté ejecutándose) y para borrarlas debemos utilizar la orden '**`clear'`**. Adicionalmente, con se vio en el capítulo 1, estas variables pueden guardarse en disco y recuperarlas posteriormente.

- **Espacio local**: este espacio de variables está aislado y corresponde a cada función implementada en código Matlab. Está compuesto por las variables de entrada y salida de la función y todas las variables que se usen en el código de la función. Cada función tiene un espacio local de variables totalmente independiente por lo que el mismo nombre la variable tendrá valores separados e independientes en diferentes funciones. Este espacio es temporal y solo existe mientras la función está en ejecución (efímeras), aunque es posible que alguna de ellas se mantenga en memoria entre diferentes ejecuciones de una función usando el operador '**`persistent'`**.

Tal como hemos comentado previamente, los **sripts** tienen acceso al espacio global de variables (*Workspace*), mientras que las **funciones** por defecto acceden a su espacio local de variables. El acceso de una función a una variable global (Workspace) exige la declaración específica de esta variable como global usando el operador '**`global`**' tanto en el Workspace como en la función antes de ser usada, por lo se considera un uso excepcional y no recomendado.

3.2 Scripts

Como hemos comentado previamente un script es una colección de comandos Matlab por lo que una forma sencilla de crear uno es a partir de la ventana '***Command History***'. Seleccionaremos el código que queramos ejecutar en el script y pulsaremos el botón derecho del ratón, con lo que nos aparecerá un menú contextual (figura 3.1) . Pulsando en la opción '*Create Srcript*' nos mostrará el editor de código Matlab con el código que habíamos seleccionado (figura 3.2). Este editor es el mismo tanto para scripts como para las funciones. Grabaremos el fichero desde el menú '*File/Save*' o pulsando el botón de acceso directo (*Ctrl*+*S*) (figura 3.3). El nombre del fichero será el nombre del comando para ejecutarlo en Matlab

```
>> ej1
```

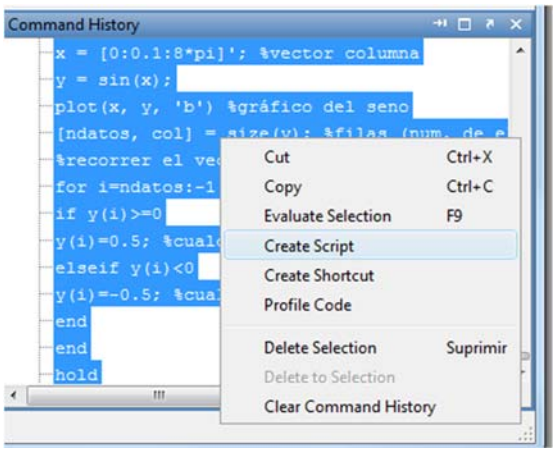

Figura 3.1 Crear un script a partir del histórico de comandos

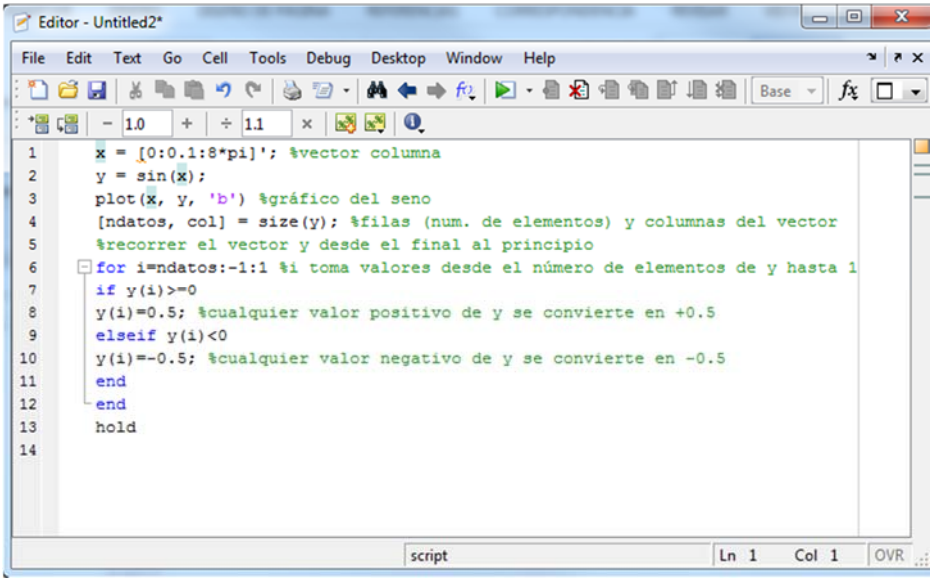

Figura 3.2 Editor de Código Matlab

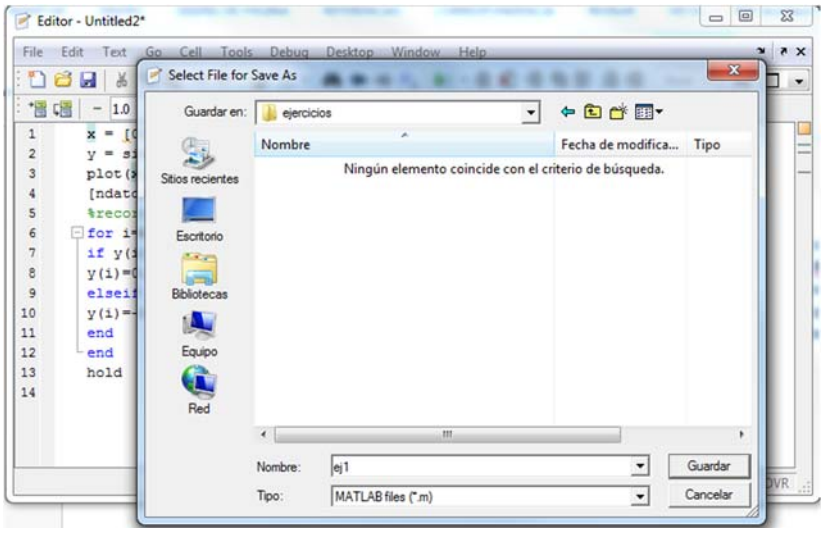

Figura 3.3 Almacenar el script en disco

También podemos crear un *Script* desde cero, para ello pulsaremos en el menú de la aplicación Matlab '*File/New/Script*' (figura 3.4). Se abrirá el editor de código como en el caso anterior pero mostrando en este caso una página en blanco (figura 3.5).

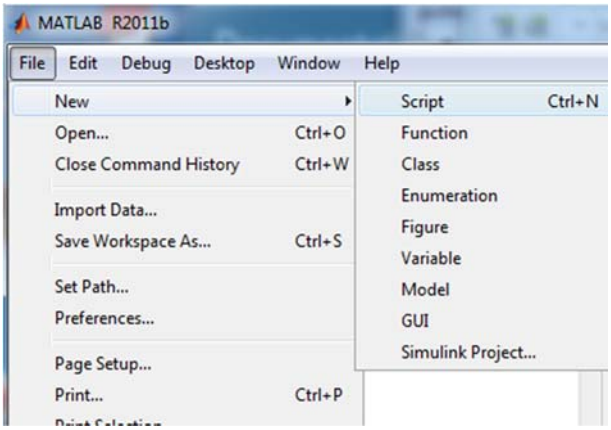

Figura 3.4 Crear un nuevo Script

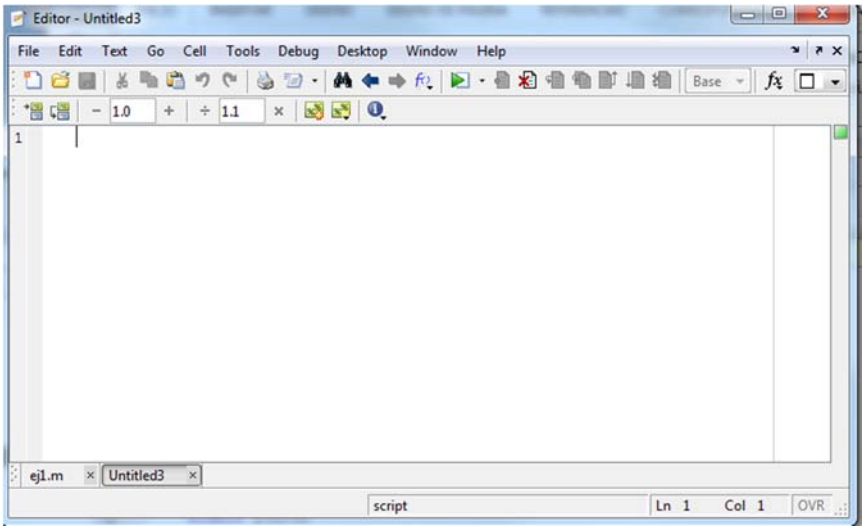

Figura 3.5 Editor de Código Matlab Nuevo Script

Una vez creado un script es importante poner en las primeras líneas un comentario usando el símbolo '**%**' que indique las tareas que realiza nuestro código. De este modo cuando usemos la ayuda con el comando '**help**' indicando el nombre de nuestra aplicación, mostrará el contenido de las primeras líneas en la cabecera que sean comentarios. En el ejemplo mostrado en la figura 3.6 al solicitar ayuda del programa en la consola de Matlab obtendríamos el siguiente resultado:

```
>> help ej2
  Descripción del código
  Detalles
```

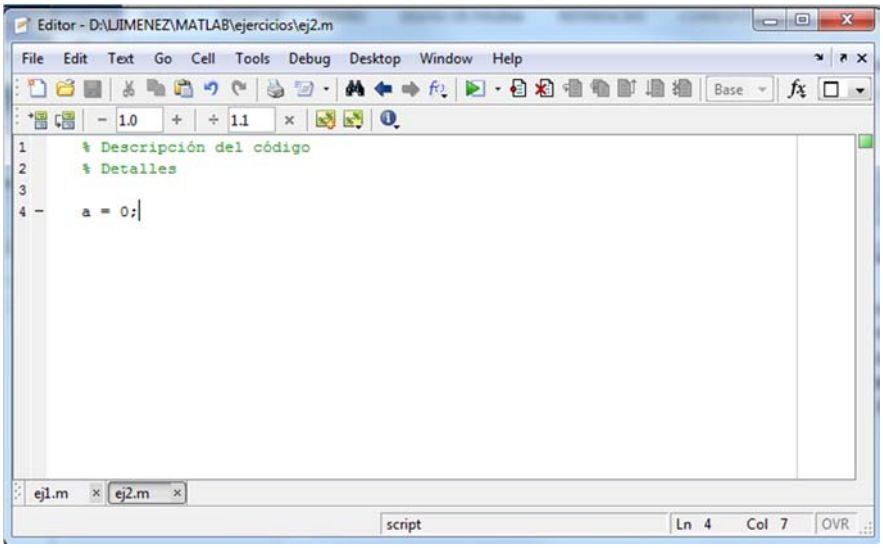

Figura 3.6 Comentarios de la cabecera de un Script

En el apartado 3.4 se comentará más en detalle la sintaxis del lenguaje Matlab para estructurar nuestro código ya que ésta es común entre Scripts y Funciones.

3.3 Funciones

Pulsando en la opción '*File/New/Function*' (figura 3.7). Se abrirá el editor de código como en el caso anterior pero mostrando en este caso la página con el código específico de una función (figura 3.8).

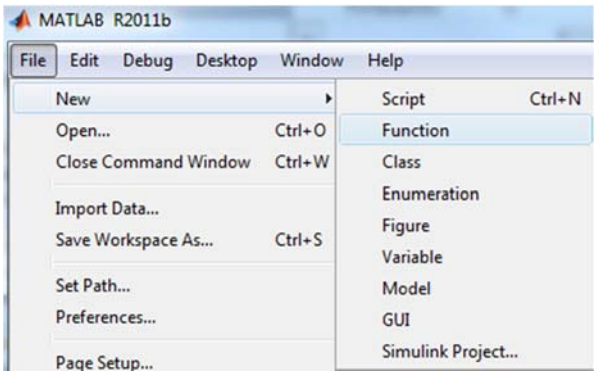

Figura 3.7 Crear una Función

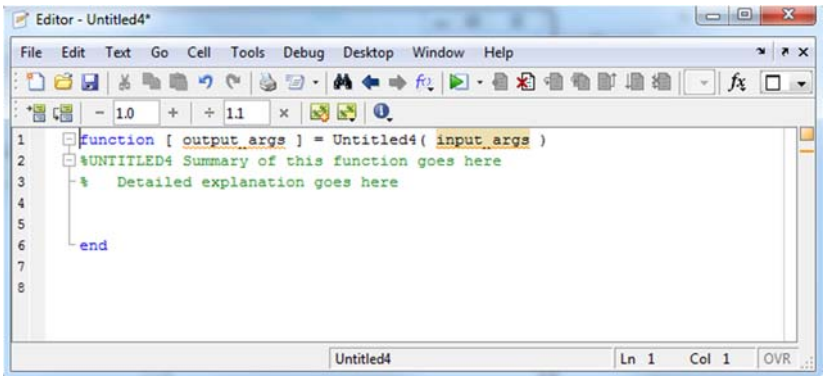

Figura 3.8 Ventana de Edición de una Función

Por defecto, al crear una función Matlab le asigna un nombre temporal (en el ejemplo '**Untitled4**'). Lo primero que haremos es sustituir esta etiqueta por el nombre de nuestra función (en este ejemplo '**fun1**'), que debe coincidir también con el nombre del fichero que usaremos para guardar la función. Grabaremos el fichero desde el menú '*File/Save*' o pulsando el botón de acceso directo (*Ctrl+S*).

Como podemos ver, una función tiene una sintaxis especial para la primera línea y debe terminar con la etiqueta **end** al final del cuerpo de la función. Vamos a ver en detalle las diferentes partes de una función.

```
function [ res ] = fun1( a, b )
%fun1 El resumen de la función va aquí
%   La explicación detalla va aquí

      % cuerpo de la función
      res = a*b;
end
```

- Una función debe empezar por la etiqueta reservada '**function**'. El código de la función termina cuando se alcanza la etiqueta '**end**'.
- A continuación se indican los parámetros de salida (devueltos por la función) seguidos por el operador '**=**'. Si se van a devolver varias variables deben disponerse en un vector usando los delimitadores de vector '**[ ]**' . En el caso de devolver una sola variable los corchetes son opcionales. También es posible dejar vacío este campo, pero es poco usual ya que el objetivo de una función es procesar datos ofreciendo un resultado.
- A continuación va el nombre de la función (en nuestro ejemplo '**fun1**'). Como se ha comentado previamente, este nombre debe ser el mismo que el nombre del fichero.
- La primera línea termina indicando entre paréntesis el nombre de los parámetros de entrada. Estas variables pertenecen al espacio local de variables de la función.
- Las dos líneas que van debajo de la definición de la función corresponden a la ayuda. Las líneas que empiezan por el símbolo '**%**' son comentarios de código. De este modo cuando usemos la ayuda con el comando '**help'** indicando el nombre de nuestra función, se mostrará el contenido de las primeras líneas que sean comentarios. Normalmente en la primera línea se indica el nombre de la función y una breve descripción, dejando las líneas siguientes para la descripción detallada. El comando '`help`' mostrará todas las líneas de comentario hasta que aparezca una línea en blanco o que no sea un comentario
- A continuación va el cuerpo de la función donde ubicaremos el código de nuestra función, teniendo cuidado de asignar un valor adecuado a los parámetros de salida de la función.
- La función termina con la etiqueta '**end**'. Se puede también terminar prematuramente una función llamando al comando del sistema **`return`**.

Para ejecutar la función basta con poner el nombre de la función y a continuación entre paréntesis los parámetros de entrada. En el monteo que es llamada la función las variables locales de entrada **a** y **b** tomarán los valores **2** y **3** indicados en el ejemplo.

```
>> c = fun1(2,3)
c =
     6
```

Dentro de un fichero puede haber más de una función, pero solo aquella cuyo nombre coincide con el nombre del fichero (función principal) puede ser llamada externamente. El resto de funciones son internas y solo pueden ser llamadas desde la función principal, permitiendo estructurar y encapsular mejor el código.

3.4 Sintaxis del código Matlab

El código Matlab se estructura en líneas. Una instrucción termina cuando se alcanza el fin de línea (retorno de línea), o bien ponemos el operador '**;**', Esta último operador produce el efecto adicional de inhibir la salida del resultado de cualquier expresión por la consola. Añadir a final el oeprador '**;**' será la opción habitual cuando creamos código para evitar la salida continua de resultados que haría muy confusa la visualización. En el apartado 3.6 veremos las opciones para mostrar resultados de forma controlada y formateada dentro de nuestro código.

Por su puesto hay ciertos comandos o estructuras de control que no generan una salida o resultado, y por tanto no es preciso terminar la línea con el símbolo '**;**'. Otro uso del operador '**;**' es la posibilidad de compactar el código poniendo varias expresiones en la misma línea.

```
>> a=5; b=6;
```

Como hemos comentado, una expresión termina con el final de una línea. Si necesitamos que una expresión de nuestro código se extienda por más de una línea, debemos hacer uso del operador '**…**'. Añadir tres puntos seguidos al final de una línea indica al intérprete de Matlab que la expresión se extiende a la siguiente línea. A continuación se muestra un ejemplo con una expresión que asigna valores a una matriz organizando cada fila de la matriz en una línea de código:

```
b = [ 1 2 3 ;...
      4 5 6 ;...
      7 8 9  ];
```

Las líneas que empiezan por el símbolo '**%**' son comentarios de código. No ejecutan ninguna función y son meramente informativas. Es importante que incluyamos comentarios distribuidos por nuestro código indicando lo que realiza cada parte del mismo para facilitar las tareas de depuración.

Otro aspecto importante en la programación en Matlab, es que es un lenguaje sensible a las mayúsculas, lo que implica que en el uso de operadores, y el nombrado de variables y funciones se distingue entre mayúsculas y minúsculas por lo que deben ser escritas exactamente. Para los operadores, etiquetas del sistema y funciones intrínsecas se usará siempre minúsculas. En el caso de variables o funciones definidas por el usuario esto queda a su elección.

En los siguientes apartados se desarrollarán las estructuras de control y funciones básicas de entra/salida de datos para crear aplicaciones en Matlab.

3.5 Estructuras de control

En este apartado se describirá la sintaxis y ejemplos de uso de las estructuras de control necesarias para implementar los algoritmos de nuestra aplicación. Estas estructuras básicas nos permitirán ejecutar código de forma condicional, realizar bucles para ejecutar código de forma repetitiva, y controlar las excepciones (errores, resultados imprevistos) para que nuestro código sea más robusto.

La mayoría de las estructuras de control permiten incluir más de una línea de código por lo que al final de mismo debemos incluir la etiqueta **end**, que indica al interprete Matlab cuando termina el código afectado por la instrucción.

3.5.1 Bifurcaciones

Una bifurcación (decisión) permite que la aplicación ejecute diferente código según el estado de una variable o expresión. Disponemos de varias alternativas:

a) Estructura: *if/else*

Permite ejecutar una decisión binaria. Ejecuta la expresión lógica y si el resultado es cierto (*true*) ejecuta las instrucciones hasta la etiqueta '**else**'. Si la expresión lógica es falsa ejecuta las instrucciones entre las etiquetas '**else**' y '**end**'

```
if expresión_lógica
   instrucciones_si_exp_es_verdadera
else
   instrucciones_si_exp_es_falsa
end
```

Ejemplo:

```
if a>b
        res = b/a;
else
        res = a/b;
end
```

En cada apartado de la estructura puede haber múltiples líneas/expresiones. La etiqueta '**else'** es opcional en el caso de que solo queramos ejecutar una acción cuando se cumple una condición.

b) Estructura: *if/elseif/else*

Permite ejecutar una decisión múltiple. Ejecuta múltiples expresiones lógicas y si el resultado es cierto (*true*) ejecuta las instrucciones hasta la siguiente etiqueta '**elseif**' o '**else**' . Si ninguna es verdadera se ejecutarán las sentencias entre as etiquetas '**else**' y '**end**'.

```
if expresión_lógica1
   instrucciones_si_exp1_es_verdadera
elseif expresión_lógica2
   instrucciones_si_exp2_es_verdadera
…
else
   instrucciones_si_ninguna_es_verdadera
end
```

Es importante tener en cuenta que solo una de las opciones será ejecutada. Si una expresión lógica es verdadera se ejecuta su código y salta a la posición **end**, por lo que no se evalúan las expresiones siguientes.

Ejemplo:

```
if a>100
        res = 'alto';

elseif a>10 && a<=100
        res = 'medio';
else
        res = 'bajo';
end
```

c) Estructura: *switch/case/otherwise*

Como en el caso anterior permite ejecutar una decisión múltiple pero operando sobre una expresión que puede ser numérica o una cadena de caracteres. Según el valor resultante de la expresión, se ejecuta el código asociado a la etiqueta **`case`** específica. En el caso de que la expresión sea un cadena de caracteres se hace una comparación exacta, distinguiendo entre mayúsculas y minúsculas o la presencia de espacios.

Las expresiones asociadas a cada etiqueta **`case`** pueden ser valores numéricos, cadenas de caracteres o listas. En este último caso, se comprueba si cualquiera de los elementos de la lista es igual a la expresión del **`switch`**.

```
switch expresión_switch
   case expresión_case1
      instrucciones_si_ expresión_switch== expresión_case1
   case expresión_case2
      instrucciones_si_ expresión_switch== expresión_case2
  case { expresión_case3_1, expresión_case3_2 }
      instrucciones_si_algun_elemento_lista==expresión_switch

    :
   otherwise
      instrucciones_si_no_es_igual_a_ninguna
end
```

Como en el caso de la estructura anterior, solo una de las opciones será ejecutada. Si una expresión lógica es verdadera se ejecuta su código y salta a la posición **end**, por lo que no se evalúan las expresiones siguientes.

Ejemplo:

```
switch valor
     case 1
        disp('opción 1');
     case 2
        disp('opción 2');
     case {3, 4}
        disp('opción 3 o 4');
     otherwise
        disp('defecto');
 end
```

3.5.2 Bucles

Los bucles nos permiten ejecutar código de forma iterativa. Disponemos de dos estructuras básicas para implementar bucles:

a) **Estructura: *while***

La estructura **`while`** ejecuta un conjunto de sentencias hasta la etiqueta **`end`** mientras se cumpla una condición lógica (`expresión`). Esta expresión normalmente tiene en cuenta alguna variable modificada por el código ejecutado para que en algún momento termine el bucle. En todo caso, como veremos más adelante, siempre podemos terminar el bucle de forma anticipada desde el código interno del bucle.

```
while expresión
   instrucciones_si_expresión_es_cierta

end
```

Ejemplo: muestra el cuadrado de los números 1:9

```
a=1;
while a<10
   disp(a*a);
   a=a+1;
end
```

b) Estructura: *for*

La estructura **`for`** ejecuta el conjunto de sentencias hasta la etiqueta **`end`** un número de veces predefinido por el tamaño del **`vector`** o **`lista`** indicado. Los valores de este vector o lista son asignados a una **`variable`** en cada iteración del bucle, que puede ser utilizada dentro de las sentencias del mismo. Habitualmente el vector suele ser un contador creado usando el operador de rango '**`:`**' visto en el capítulo 1, pero cualquier expresión que genere un vector es válida. Como en el caso anterior, podemos terminar el bucle de forma anticipada desde el código interno del bucle.

```
for variable = vector_lista
   instrucciones

end
```

Ejemplo: muestra el cuadrado de los números 1:9

```
for a = 1:9
    disp(a*a);
end
```

Ejemplo: muestra una lista de cadenas.

```
lista = { 'el(1)', 'el(3)', 'el(2)'};
for a = lista
    disp(a);
end
```

Interrupción de un bucle: *continue, break*

Las dos estructuras iterativas anteriores pueden ser interrumpidas desde dentro del bucle utilizando los siguientes operadores:

`continue`: salta la ejecución al final del bucle, continuando con la siguiente iteración si existe. Por lo tanto, todas las sentencias entre los operadores `continue` y `end` no se ejecutarán es esa iteración. Nos permite introducir condiciones adicionales para la ejecución de las sentencias dentro del bucle.

break: termina la ejecución del bucle de forma inmediata. Permite terminar de forma anticipada un bucle por una condición independientemente de la condición lógica del bucle *while* o el vector de iteración del bucle *for*.

Ejemplo: muestra la inversa de los componente de un vector (salta valores nulos)

```
vec = [ 1 2 3 0 4 5 6];
for a = vec
    if a==0
        disp('NaN');
        continue
    end

    disp(1/a);
 end

>> ej3
     1
    0.5000
    0.3333
NaN
    0.2500
    0.2000
    0.1667
```

3.5.3 Control de excepciones (try catch)

Cuando escribimos código deberemos tener en cuenta que ciertas expresiones o llamadas a funciones pueden fallar si los parámetros no tienen un valor adecuado. Para recuperar una aplicación de estos errores disponemos de dos alternativas:

a) Comprobar el valor devuelto por cada expresión o función llamada, ejecutando un código alternativo en caso de fallo (instrucción **`if`**). Esto provoca que el código sea muy complejo y difícil de seguir y depurar.

b) Utilizar el gestor de excepciones de Matlab, que nos permite seleccionar aquella parte del código que sea crítica y asignarle un manejador, que en caso de fallo saltará a un segmento de código especial que recibirá como dato el origen del error producido para actuar en consecuencia. Esto se realiza mediante las instrucciones **`try catch`**.

La sintaxis de esta estructura de control es la siguiente:

```
try
   instrucciones_a_monitorizar

catch error
   instrucciones_si_error
end
```

Normalmente se ejecutará solo el código entre las etiquetas **`try`** y **`catch`**. En caso de se produzca algún error de ejecución (excepción), se ejecutará el código entre las etiquetas **`catch`** y **`end`**. Así mismo, nos proporciona la variable **`error`** que es una estructura que nos incluye información detallada de la causa del error (ver el apartado 3.8 sobre el uso de estructuras).

Ejemplo: suma de dos vectores de diferente dimensión

```
A = [ 1 2]; B=[1 2 3];
try
   C=A+B;

catch error
    disp('Error en la operación:');
    disp(error);
end
```

```
>> ej3
Error en la operación:
  MException
  Properties:
        identifier: 'MATLAB:dimagree'
        message: 'Matrix dimensions must agree.'
        cause: {0x1 cell}
        stack: [1x1 struct]
```

3.6 Entrada y salida de datos del usuario

Cuando programamos una aplicación debemos tener presente al necesidad de interactuar con el usuario de la misma, recabando datos y generando resultado adecuadamente formateados. En esta sección veremos las funciones básicas para introducir en el código Matlab la entrada y salida de datos del usuario.

3.6.1 Entrada de Datos del usuario:

La función **`input`** para la ejecución del programa y espera la introducción de datos por teclado. El parámetro **`prompt`** es una cadena de caracteres que se mostrará al usuario antes del cursor. Por defecto esta función lee datos numéricos o expresiones evaluables como un número. Si deseamos leer una cadena de caracteres y que no sea evaluada, debemos añadir el parámetro **`'s'`**.

```
x = input(prompt)
str = input(prompt,'s')
```

Ejemplos:

```
>> a = input('Valor de a: ')
Valor de a: 3
a =
     3

>> a = input('Valor de a: ')
Valor de a: 4/3
a =
    1.3333

>> a = input('Valor de a: ', 's')
Valor de a: cadena
a =
    cadena
```

En caso de leer cadenas de caracteres, si deseamos posteriormente evaluarlo como un valor numérico disponemos de la función **`str2num`**. Esta función permite evaluar vectores y matrices siguiendo la sintaxis de Matlab. En la tabla 3.1 se muestran varios ejemplos de uso.

```
x = str2num(str)
```

Ejemplos:

str	Salida Numérica	Tipo de objeto
`'500'`	500	double
`'500 250 125 67'`	500 250 125 67	1x4 vector fila de double
`'500; 250; 125; 62.5'`	500.0000 250.0000 125.0000 62.5000	4x1 vector columna de of double
`'1 23 6 21; 53:56'`	1 23 6 21 53 54 55 56	2x4 matriz de double
`'12e-3 5.9e-3'`	0.0120 0.0059	1x2 vector fila de double
`'false true'`	0 1	1x2 vector fila de logical

Tabla 3.1. Ejemplos de uso de la función `str2num`

3.6.2 Visualización de resultados:

Matlab provee varias funciones para mostrar datos formateados en la consola.

Función `disp` : `disp(X)`

Muestra una variable Matlab por pantalla. Esta puede ser una cadena de caracteres o una expresión numérica escalar, vectorial o matricial, o una estructura. A continuación se muestran algunos ejemplos de uso:

```
>> disp(a*2);
24.2469

>> disp(A)
     1     2
     3     4

>> disp('cadena');
Cadena

>> disp(['cadena1 '  'cadena2']);
cadena1 cadena2
```

Destacar que si queremos concatenar varias variables podemos usar el operador corchete para crear un vector, pero es conveniente que todos los elementos sean del mismo tipo. En caso de querer visualizar números junto a una cadena de caracteres disponemos de la función **`num2str`**. Esta función permite elegir el formato de la representación de los datos y permite convertir números escalares o matrices, generando en este último caso una matriz de cadenas.

```
s = num2str(A)
s = num2str(A,precision)
s = num2str(A,formatSpec)
```

Disponemos de tres opciones según los parámetros. Si solo pasamos el número o matriz de datos numéricos realiza un formateo automático. Si adicionalmente le pasamos la precisión, ajustará el número de decimales. En el último caso le pasamos junto a los datos a convertir, otra cadena adicional donde le indicamos el formateo deseado (similar a la utilizada en la función `fprintf` que se verá a continuación).

```
>> disp(['Valor de a: '  num2str(a)]);
Valor de a: 12.1235
```

Función `error` : `error('ERRMSG')`

Si queremos mostrar un error de ejecución podemos usar esta función alternativa que permite adicionalmente abortar la ejecución de la función actual.

```
if a==0
    error('El divisor es cero');
end
```

Función `fprintf`:

Si queremos realizar un formateo de los datos más complejo disponemos de la función **`fprintf`**. Se trata de una función similar a la existente en el lenguaje C, en la que el usuario especifica en una cadena para los datos a visualizar, su tipo y su formateo. A continuación de esta cadena se pasan los datos como una lista variable de parámetros.

```
fprintf(FORMAT, A, ...)
```

Cada valor a visualizar se especifica según la siguiente sintaxis (figura 3.9):

- Empieza con el símbolo '%'
- Identificador: n$ indica la posición en la lista de parámteros del valor a mostrar. Si no se indica, se utilizan en el mismo orden que son pasados los parámetros.
- A continuación se pueden incluir flags y modificadores que controlan el tipo de salida (tabla 3.3)
- Número de dígitos que debe ocupar (ancho mínimo de campo incluyendo parte entera, parte decimal y exponente),
- Separada por el punto decimal se indica la precisión (número de dígitos decimales)
- Tipo de conversión (tabla 3.2)

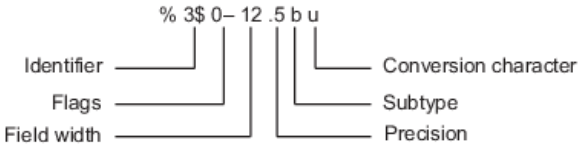

Figura 3.9 Campos de formato fprintf

Ejemplo:

```
>>a =12.12345678;

>> fprintf('Valor: %10.4f\n', a );

Valor:    12.1234
```

Tipo	Conversión	Detalles
Entero con signo	%d ó %i	Base 10
Entero sin signo	%u	Base 10
	%o	Base 8 (octal)
	%x	Base 16 (hexadecimal), letras minúsculas a–f
	%X	Igual que %x, letras mayúsculas A–F
Número en Coma Flotante	%f	Notación coma fija (el operador de precisión especifica el número de dígitos decimales)
	%e	Notación exponencial (científica), 3.141593e+00 (el operador de precisión especifica el número de dígitos decimales)
	%E	Notación exponencial (científica) en mayúsculas, 3.141593E+00 (el operador de precisión especifica el número de dígitos decimales)
	%g	Forma más compacta de %e o %f sin ceros a la derecha (el operador de precisión especifica el número de dígitos significativos)
	%G	Forma más compacta de %E o %f sin ceros a la derecha (el operador de precisión especifica el número de dígitos significativos)
Caracteres o cadena	%c	Carácter simple
	%s	Cadena (vector) de caracteres.

Tabla 3.2. Tipos de conversión en el formateo de la función fprintf

Flags	Acción
'-'	Justificar a la izquierda
'+'	Poner el signo del número (+ o -)
' '	Insertar un espacio antes del valor
'0'	Rellenar con ceros a la izuierda hasta completar el ancho de campo
'#'	Modifica algunas conversiones numéricas: - Para %o, %x, o %X, imprime el prefijo 0, 0x, o 0X - Para %f, %e, o %E, imprime el punto decimal incluso con precisión 0 - Para %g o %G, no elimina los ceros ceros al final.

Tabla 3.3. Flags en el formateo de la función fprintf

3.7 Entrada y salida de datos (ficheros)

El código de nuestra aplicación actúa procesando datos (variables) para obtener unos resultados (variables). Cuando nuestra aplicación es compleja también lo serán los datos utilizados, por lo que uno de los problemas que debemos resolver es la carga de esos datos y el almacenamiento de los resultados.

En el capítulo 1 ya se vieron algunas funciones básicas para almacenar las variables del Workspace en ficheros en un formato específico de Matlab `'.mat'`. Cuando escribimos código, en ocasiones necesitaremos acceder a ficheros que no usan ese formato, o son demasiado extensos para ser cargados en memoria de forma permanente, por lo que necesitaremos utilizar las funciones de entrada/ salida que nos ofrece Matlab.

3.7.1 Almacenamiento en ficheros .mat

Las funciones disponibles son **`load`** y **`save`** que permiten almacenar y recuperar variables tanto globales (W*orkspace*) como locales.

La función `save` guarda todo el espacio de variables en el fichero indicado en el primer parámetro. Opcionalmente se pueden indicar aquellas variables a almacenar añadiendo como parámetros el nombre de cada variable en formato cadena de caracteres. Si se llama desde un script se guarda el *Workspace* global. Si se llama desde una función se guarda el espacio de variables local de la función.

```
save(filename)
save(filename, variables)
```

Ejemplo: almacena en el fichero '`datos.mat`' las variables `a,b,c`

```
save('datos.mat', 'a', 'b', 'c');
```

La función `load` lee las variables almacenadas en el fichero indicado en el primer parámetro. Opcionalmente se pueden indicar aquellas variables a leer se forma similar a la función anterior. Si se llama desde un script las variables se crean en el *Workspace* global. Si se llama desde una función las variables se crean el espacio de variables local de la función.

```
load(filename)
load(filename, variables)
```

Ejemplo: Lee del fichero '`datos.mat`' las variables `a,b,c`

```
load('datos.mat', 'a', 'b', 'c');
```

3.7.2 Almacenamiento en ficheros de texto o binarios

En la tabla 3.4 se detallan las funciones para abrir leer o escribir en ficheros en formato texto o binario. En este apartado se describirán las más habituales con algunos ejemplos de uso.

Función	Descripción
`fclose`	Cierra un fichero
`feof`	Comprueba si se ha alcanzado el final del fichero
`ferror`	Información de errores de E/S
`fgetl`	Lee una línea del fichero. Elimina retornos de línea
`fgets`	Lee una línea del fichero. Mantiene retornos de línea
`fileread`	Lee los contenidos de un fichero como un texto
`fopen`	Abre un fichero
`fprintf`	Escribe un cadena formateada en un fichero
`fread`	Lee datos de un fichero binario
`frewind`	Mueve el indicador de posición al principio del fichero
`fscanf`	Lee datos formateados de un fichero de texto
`fseek`	Mueve el indicador de posición a un punto cualquiera
`ftell`	Posición del indicador de posición en un fichero
`fwrite`	Escribe datos de un fichero binario
`type`	Visualiza el contenido de un fichero

Tabla 3.4. Funciones de lectura y escritura de ficheros

Lectura de un fichero en una cadena

La función más habitual para la lectura de ficheros de texto, es `fileread` ya que integra en una sola llamada todas las etapas de acceso a un fichero (lo abre, lee el contenido asignándolo a una variable de tipo *string* y cierra el fichero)

```
text = fileread(filename);
```

El resultado es almacenado en una cadena de caracteres que contendrá los retornos de línea. Vamos a ver un ejemplo en el que leemos un fichero de texto que contiene en cada fila un vector de datos y una etiqueta:

Fichero: datos.txt				
0.12	14.3	2.3	1	tuerca
0.92	8.45	6.7	2	tornillo
0.42	6.23	4.6	3	arandela

```
>> txt =fileread('datos.txt');
>> whos txt
  Name      Size            Bytes  Class    Attributes
  txt       1x78              156  char
>> txt
txt =
0.12  14.3  2.3   1     tuerca

0.92  8.45  6.7   2     tornillo

0.42  6.23  4.6   3     arandela
```

Nota: podemos visualizar el contenido de un fichero de texto mediante el operador **`type`**. Ejemplo:

```
>> type datos.txt

0.120 14.300 2.300 tuerca

0.920 8.450 6.700 tornillo

0.420 6.230 4.600 arandela
```

Escritura en un fichero de datos formateados

Para la escritura de datos formateados en un fichero podemos usar la función **`fprintf`** vista en el apartado anterior añadiendo un parámetro más que es el identificador (FID) del fichero (para las opciones de formato ver el apartado 3.6.2). El identificador de fichero se obtiene mediante la llamada a la función **`fopen`**. En la apertura de un fichero debemos especificar los permisos de acuerdo a la tabla 3.5. Por último, una vez hemos terminado debemos cerrar el fichero llamando a la función **`fclose`** que garantiza que los datos almacenadas en caché sean escritos en el fichero físico.

```
fprintf(FID, FORMAT, A, ...)

FID = fopen(FILENAME,PERMISSION)

estado = fclose(FID)
```

Permisos fopen	Descripción
'r'	Abre un fichero para lectura
'w'	Abre o crea un fichero para escritura. Descarta el contenido existente
'a'	Abre o crea un fichero para escritura. Añade datos al final del contenido existente
'r+'	Abre (no lo crea) un fichero para lectura y escritura
'w+'	Abre o crea un fichero para lectura y escritura. Descarta el contenido existente
'a+'	Abre o crea un fichero para lectura y escritura. Añade datos al final del contenido existente

Tabla 3.5. Parámetro de permiso en la apertura de un fichero con fopen

A continuación se muestra un ejemplo de escritura de datos formateados del fichero a partir de una matriz similar al fichero *'datos.txt'* anterior.

```
datos = [0.12 14.3 2.3; 0.92 8.45 6.7; 0.42 6.23 4.6];
y = {'tuerca'; 'tornillo'; 'arandela'};

fid = fopen('datos.txt', 'w');
for i=1:size(datos,1)
    fprintf(fid, '%.3f\t %.3f\t  %.3f\t  %s\n', ...
                    datos(i,1), datos(i,2), datos(i,3), y{i});
end

fclose(fid);
```

Lectura de un fichero de datos formateados

Para la lectura de datos formateados (numéricos), disponemos de la función `fscanf`. Esta función lee los datos de un fichero identificado pro `FID`, los convierte de acuerdo a la cadena `FORMAT` y devuelve los resultado s en la matriz A. Opcionalmente devuelve en `COUNT` el número de datos leídos.

```
[A,COUNT] = fscanf(FID, FORMAT)
```

```
[A,COUNT] = fscanf(FID, FORMAT, SIZE)

estado = feof(FID)
```

Opcionalmente se puede indicar como parámetro el tamaño que pone un límite al número de elementos leídos (tabla 3.6).

Size	Descripción
N	Lee a lo sumo N elementos en un vector columna
inf	Lee hasta el final de línea o del fichero
[M,N]	Lee a lo sumo MxN elementos en una matriz de dimensión MxN. N puede ser inf

Tabla 3.6. Parámetro de tamaño de la función fscanf

A continuación se muestra un ejemplo para leer los datos formateados del fichero datos.txt anterior.

```
clear;
datos =[]; y = {};
fid = fopen('datos.txt', 'r');
while ~feof(fid)
    % para cada línea del fichero
    vect = fscanf(fid, '%f', inf);  % lee los 3 campos numéricos
    etq =  fscanf(fid, '%s\n', 1);  % lee la etiqueta (string)
    datos = [ datos; vect' ];       % almacena los datos
    y{size(y,1)+1, 1} = etq;       % almacena la etiqueta
end
fclose(fid);
```

3.7.3 Manejo de directorios

En la tabla 3.7 se describen las funciones para el manejo de directorios. Estas nos permiten conocer el contenido de un directorio, crearlo, borrarlo, copiarlo consultar y modificar el directorio de trabajo por defecto.

Función	Descripción
dir	Da el listado del contenido de un directorio
exist	Comprueba la existencia de una variable, script, función, carpeta o clase
isdir	Determina si es in directorio
pwd	Identifica el directorio actual
cd	Cambia el directorio actual
copyfile	Copia un fichero o carpeta
delete	Borra ficheros u objetos
mkdir	Crea un nuevo directorio
movefile	Mueve un fichero o directorio
rmdir	Elimina un directorio

Tabla 3.7. Funciones de manejo de directorio

```
D = dir('directory_name')
```

La función **`dir`** nos devuelve un vector con los ficheros ubicados en el directorio indicado. En el argumento directorio se pueden añadir filtros para el tipo de ficheros

buscados. Los datos devueltos consisten en una vector de estructuras (Mx1) (ver apartado 3.8) que contienen los detalles de cada fichero. Los campos de esta estructura son los siguientes:

```
name    -- Nombre fichero
date    -- Fecha modificación
bytes   -- Tamaño en bytes del fichero
isdir   -- 1 si es un directorio 0 si no lo es
datenum -- Fecha modificación (serial)
```

A continuación se muestra un ejemplo:

```
ficheros = dir('*.*');

for f = ficheros'       % vector de estructuras traspuesto
     disp(['- Nombre: ', f.name, ' - Fecha: ', f.date, …
                       ' - Tamaño: ', f.bytes, ' bytes'] );
     if f.isdir
         disp('  Tipo: directorio');
     else
         disp('  Tipo: fichero');
     end
 end
```

3.7.4 Almacenamiento en un servidor de red:

En ciertas aplicaciones el origen de los datos está ubicado en servidores remotos. Matlab provee funciones básicas para establecer una conexión TCP/IP con un servidor para la lectura y escritura de datos. En este caso es preciso conocer el protocolo de comunicación (formato de datos y mensajes) usado por el servidor. En la tabla 3.8 se describen las funciones disponibles para implementar este tipo de comunicación.

Función	Descripción
`tcpclient`	Crea un objeto cliente TCP/IP
`read`	Lee datos de un servidor remoto sobre TCP/IP
`write`	Escribe datos en un servidor remoto sobre TCP/IP

Tabla 3.8. Funciones de acceso a datos en un servidor de red

La implementación de una aplicación de red depende en gran medida del protocolo utilizado por el servidor, que puede ir de un simple protocolo en texto plano de tipo petición respuesta, a un protocolo más complejo basado en ficheros normalizados XML.

3.8 Estructuras de datos

Matlab proporciona un conjunto de tipos de datos básicos que a su vez, se pueden agrupar formando vectores, matrices o listas. Cuando necesitamos manejar estructuras de datos más complejas Matlab proporciona un método para definirlas mediante el comando **struct**. El uso de estructuras es similar al de otros lenguajes como C, definiendo un conjunto de campos con su nombre y valor (en Matlab no es necesario definir el tipo). El acceso a estos campos se realiza con el operador '**.**'. A continuación se enumeran las diferentes opciones para crear una estructura:

```
s = struct
s = struct(field,value)
s = struct(field1,value1,...,fieldN,valueN)
```

```
s = struct([])
```

La primera opción (sin parámetros) crea una estructura vacía, ésta puede ser posteriormente modificada creando campos usando el operador '.'. Los métodos 2 y 3 permiten crear una estructura con un conjunto de campos y valores ya definidos, indicando como parámetros el nombre del campo como un string y el valor asociado. Para dejar el valor de un campo vacío se debe indicar como un vector o lista vacía: '[]' ó '{}'. Veamos un ejemplo utilizando ambos métodos:

Método 1:

```
s = struct;    %declara una estructura vacía
s.etiqueta = 'pieza1';
s.id = 1;
s.descriptor = [ 0.5 0 0.8 0.9];

>> s
s =
      etiqueta: 'pieza1'
            id: 1
    descriptor: [0.5000 0 0.8000 0.9000]
```

Método 2:

```
>> s = struct('etiqueta', 'pieza1', 'id', 1, 'descriptor', [ 0.5 0 0.8 0.9]);
>> s
s =
    etiqueta: 'pieza1'
        id: 1
  descriptor: [0.5000 0 0.8000 0.9000]
```

La estructura creada por defecto es de tipo escalar, pero posteriormente pueden insertarse nuevos elementos convirtiéndolo en un array (vector) de estructuras:

```
>> s(2) = struct('etiqueta', 'pieza2', 'id', 2, 'descriptor', …
                          [0.1 0 0.6 0.8]);
>> s(2)
ans =
      etiqueta: 'pieza2'
            id: 2
    descriptor: [0.1000 0 0.6000 0.8000]
```

Por último, la creación con `struct([])` devuelve un estructura vacía, útil para crear arrays de estructuras. En la tabla 3.9 se da un listado de las funciones aplicable a una estructura.

Función	Descripción
`fieldnames`	Obtiene nombres de campos
`getfield`	Obtiene contenido de campos
`isfield`	Verdad si un campo esta en estructura
`isstruct`	Verdad si es estructura
`rmfield`	Elimina un campo de la estructura
`setfield`	Establece el valor de un campo
`struct2cell`	Convierte estructura en celda

Tabla 3.9. Funciones aplicables a estructuras

3.9 Funciones con parámetros con valor por defecto (nargin)

Cuando se declara una función se especifican los parámetros de entrada y salida. Si la función es llamada con un número diferente de parámetros nos devolverá un error. Matlab integra la posibilidad de asignar valores por defecto a los últimos parámetros en la llamada a una función, de forma que se pueda ejecutar una función con un número menor de parámetros. Esto se realiza mediante la función del sistema **`nargin`**, y es especialmente útil para crear una función que proporcione funcionalidades extra sin necesidad de reescribir una nueva función.

La función **`nargin`** devuelve el número de parámetros pasados a una función, lo que nos permite integrar el código para asignar valores por defecto a los parámetros no pasados. Veamos un ejemplo (figura 3.10):

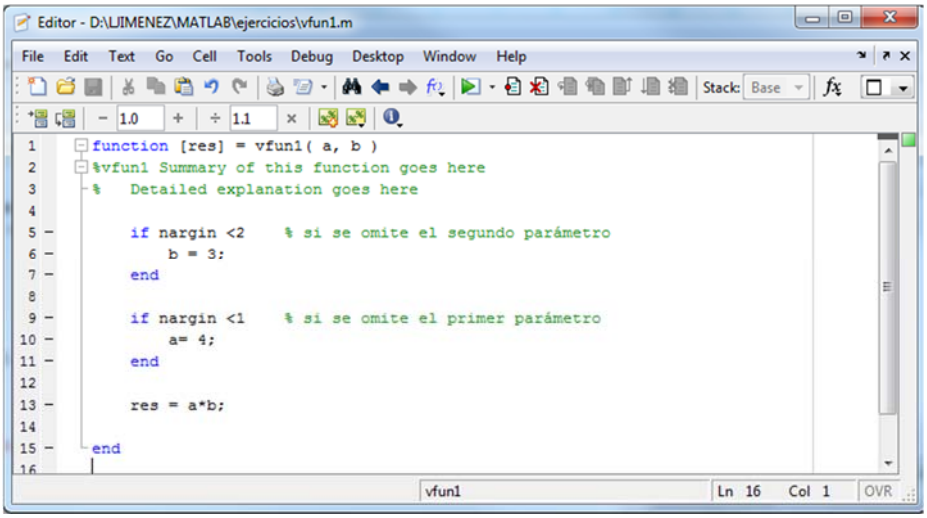

Figura 3.10 Parámetros con valores por defecto

Podemos usar la función omitiendo uno o los dos parámetros:

```
>> vfun1(6, 3)
ans =
    18
>> vfun1(6)
ans =
    18
>> vfun1
ans =
    12
```

La única limitación es que solo se pueden omitir los parámetros finales, es decir si en una función de 5 parámetros queremos omitir el parámetro 3º deberemos omitir también todos los posteriores.

3.10 Funciones con número de parámetros variable (varargin)

El método anterior nos permite omitir algunos parámetros en la llamada de una función, pero el número total de parámetros es fijo y predefinido, ya que se debe indicar en la declaración de la función. Matlab incluye la opción de declarar un número indefinido de parámetros de entrada y salida de una función, de forma similar a lo que podemos usar en la función **`fprintf()`**.

Como se ha comentado, la función **`fprintf`** permite formatear una salida de datos, indicando una cadena con la expresión formateada de las variables a mostrar, y un número variable no predefinido de parámetros. En Matlab esto se realiza mediante el uso de las variables del sistema **`varargin`** y **`varargout`**. Estas variables son de tipo lista por lo que para acceder a sus contenidos debemos utilizar el operador llave '**`{ }`**'.

Veamos un ejemplo de una función con lista de parámetros variables

```
function [] = fun4(varargin)
%fun4 Summary of this function goes here
%   Detailed explanation goes here

    optargin = size(varargin,2);
    fprintf('  Entradas en varargin(%d):\n', optargin);
    for k = 1 : optargin
         disp(varargin{k});
    end
end
```

De forma similar podemos asignar una lista de parámetros de salida variable usando la variable de tipo lista **`varargout`**, aunque suele ser menos usual.