UNIVERSIDAD DE JAÉN
Departamento de Informática

PROGRAMACIÓN EN MATLAB

Francisco Martínez del Río

El objetivo de estos apuntes de la asignatura "Complementos de Informática", del Máster en Ingeniería Industrial de la Universidad de Jaén, es avanzar en el estudio de las herramientas que proporciona un lenguaje de programación para resolver problemas, siguiendo la tarea empezada en la asignatura "Informática" del primer curso del grado.

Como lenguaje de programación se ha optado por MATLAB por varios motivos. En primer lugar, todos aquellos estudiantes que han cursado la asignatura "Informática" ya lo conocen, por lo que no es necesario explicar el lenguaje desde cero. No obstante, en estos apuntes se presupone poco conocimiento inicial del lenguaje; aunque se obvian algunos detalles, especialmente en el primer tema, donde no se describen los menús, las ventanas del entorno de ejecución de MATLAB, los tipos de datos básicos o el concepto de guión. En segundo lugar, MATLAB es uno de los lenguajes más utilizados en entornos científicos y de ingeniería, destacando especialmente por sus facilidades para el trabajo con matrices y para la visualización gráfica de datos.

En cuanto a los contenidos de los apuntes, los temas primero y tercero cubren las estructuras de control secuenciales, condicionales e iterativas comunes a todos los lenguajes que permiten expresar el flujo de control del programa. El tema segundo trata exhaustivamente la estructura de datos central de MATLAB: el *array*. Los temas cuarto y noveno describen cómo generar gráficos en MATLAB. Aquí se destacan las características usadas más frecuentemente pues, dada su amplitud, no es posible un estudio sistemático de todas las posibilidades gráficas de MATLAB. El tema quinto describe las funciones en detalle; las funciones permiten organizar el código, facilitando el desarrollo de aplicaciones más estructuradas que son más fáciles de desarrollar, comprender y mantener. El tema décimo describe la recursividad, que consisten en desarrollar funciones que se invocan a sí mismas. Por último, los temas sexto, séptimo y octavo describen tres estructuras de datos: las cadenas de caracteres, los *arrays* de celdas y las estructuras. A diferencia de los *arrays* numéricos, estas estructuras de datos no están especializadas en la realización de cálculos aritméticos, sirviendo para representar información genérica como el historial de un paciente o los detalles de la realización de un experimento.

A lo largo de todos los apuntes se detalla cómo se puede almacenar el contenido de las distintas estructuras de datos en archivos para su almacenamiento permanente. Se trabaja con archivos de texto y archivos binarios nativos de MATLAB, no describiéndose el trabajo general con archivos binarios.

Índice general

Tema 1
Variables y estructuras de programación secuenciales

En este tema se describe brevemente las estructuras de programación secuenciales de MATLAB. Estas estructuras tienen la propiedad de ejecutarse en secuencia, es decir, una detrás de otra en el orden en que aparecen en el programa. Las estructuras secuenciales son la asignación y las instrucciones de entrada y salida. También se comenta ciertos aspectos de las variables: cómo construir un identificador válido, cómo consultar las variables activas o cómo guardar variables en archivos.

1.1. Asignación de valores a variables

En MATLAB el operador de asignación de un valor a una variable es el símbolo =. MATLAB es un lenguaje interpretado y permite que a una variable se le asignen valores de distintos tipos. Por ejemplo, en la Figura 1.1 se utiliza la ventana de órdenes para asignarle a la variable x tres valores de tipos distintos. En primer lugar el vector [1 5], después la cadena de caracteres El extranjero y por último el escalar 8. La última asignación ilustra que a la derecha del operador de asignación se puede utilizar una expresión que incluya valores literales, variables, operadores y llamadas a funciones.

Observa que siempre que se realiza una asignación el resultado se reproduce en la pantalla. Se puede terminar la asignación con un punto y coma para evitar la reproducción:

```
>> valor = 3
valor =
     3
>> saldo = 520.5;
>> saldo
saldo =
  520.5000
```

```
>> x = [1  5]
x =
     1     5
>> x = 'El extranjero'
x =
El extranjero
>> x = 8
x =
     8
>> y = x^2 + floor(2.5)
y =
    66
```

Figura 1.1: Asignación de valores.

El valor asignado a saldo no se ha reproducido tras la asignación, aunque ésta tenga lugar. Por defecto MATLAB usa bastantes espacios para visualizar las asignaciones, las opciones compact y loose de la orden format permiten ajustar este espaciado:

```
>> mensaje = 'hola'

mensaje =

hola

>> format compact
>> mensaje = 'hola'
mensaje =
hola
>> format loose
>> mensaje = 'hola'

mensaje =

hola
```

En estos apuntes utilizaremos el formato compacto para ahorrar espacio. El formato por defecto es loose. Fíjate también que MATLAB presenta las cadenas de caracteres a partir de la primera columna y el resto de datos indentados. El motivo es diferenciar una cadena numérica de un número:

```
>> cadena = '66'
cadena =
```

```
66
>> numero = 66
numero =
     66
```

1.1.1. Consultas sobre variables

Es posible consultar las variables activas en el espacio de trabajo—*workspace*—usando la orden who:

```
>> who
Your variables are:
x  y
```

La orden whos muestra información adicional sobre las variables:

```
>> whos
  Name        Size                Bytes  Class     Attributes
  x           1x1                     8  double
  y           1x1                     8  double
```

como la cantidad de elementos que contiene la variable, el tamaño en *bytes* que ocupa o su tipo. El *tipo* o *clase* de una variable determina los valores que se pueden almacenar en la variable y el tipo de operaciones que se pueden realizar con la variable. Se puede consultar el tipo de una variable con la función class:

```
>> class(x)
ans =
double
```

La función class devuelve una cadena de caracteres que indica el tipo de la expresión que recibe como parámetro. Existen varias funciones que permiten consultar si una expresión es de un determinado tipo, de entre ellas destacamos ischar, islogical e isnumeric, que devuelven un valor lógico indicando si la expresión que reciben como parámetro es de tipo cadena de caracteres, lógica o numérica respectivamente. Veamos un ejemplo de uso:

```
>> nombre='Juan';
>> casado=true;
>> edad=30;
>> ischar(nombre)
ans =
     1
>> islogical(casado)
```

```
ans =
     1
>> isnumeric(edad)
ans =
     1
>> isnumeric(nombre)
ans =
     0
```

En las salidas previas ten en cuenta que MATLAB muestra el valor true como 1 y false como 0.

1.1.2. Borrado de variables

La orden clear permite borrar variables del espacio de trabajo:

```
>> whos
  Name        Size                Bytes  Class     Attributes
  ans         1x6                    12  char
  x           1x1                     8  double
  y           1x1                     8  double
>> clear x      %borra x
>> whos
  Name        Size                Bytes  Class     Attributes
  ans         1x6                    12  char
  y           1x1                     8  double
>> clear        %borra todas las variables
>> who
```

1.1.3. Nombre de una variable

El nombre de una variable debe verificar las siguientes reglas, que indican las restricciones para construir un identificador correcto:

- Debe comenzar por una letra del alfabeto inglés.
- Puede estar formada por números, letras y el símbolo de subrayado.
- Puede tener cientos de caracteres, pero los n primeros no deben coincidir con el nombre de otra variable—la función namelengthmax indica este número.
- No puede coincidir con el nombre de ninguna palabra reservada como if o function.

MATLAB es un lenguaje sensible a las mayúsculas por lo que los nombres radio y Radio son distintos. Veamos ejemplos de nombres válidos e inválidos:

```
>> if = 4       % error, palabra reservada
 if = 4
    |
Error: The expression to the left of the equals sign is not a valid ...
    target for an assignment.
>> radio_4 = 2.3    % valido
radio_4 =
    2.3000
>> 4radio = 2.3    %no es valido, no empieza por letra
 4radio = 2.3
 |
Error: Unexpected MATLAB expression.
>> namelengthmax    %maximo de caracteres tenidos en cuenta en un nombre
ans =
    63
```

Es posible utilizar un nombre de variable que coincida con el nombre de una variable o función interna de MATLAB, pero la nueva variable oculta el nombre anterior:

```
>> pi
ans =
    3.1416
>> pi = 4
pi =
     4
>> pi
pi =
     4
>> sind(90) %seno de un angulo en grados
ans =
     1
>> sind = 7.8
sind =
    7.8000
>> sind(90)
??? Index exceeds matrix dimensions.
```

En la última expresión—sind(90)—se considera que sind es una variable numérica y se intenta acceder al elemento de índice 90 de sind. Sin embargo, sind sólo tiene un elemento, por lo que se indica que el índice está fuera de dimensión.

1.2. Almacenamiento de variables en archivos

A veces se tiene que terminar una sesión de trabajo sin haber terminado todos los cálculos deseados. En ese caso resulta muy útil poder guardar variables que contienen cálculos intermedios en un archivo o fichero para poder recuperarlos en sesiones posteriores.

MATLAB nos permite guardar variables en archivos de texto y en archivos binarios. Los archivos binarios se denominan archivos MAT porque tienen una estructura nativa. En este capítulo vamos a trabajar con estos archivos, cuya extensión es *.mat*, porque son más flexibles y más fáciles de usar que los archivos de texto. La desventaja de usar archivos MAT es que al ser su estructura nativa no nos permite intercambiar información con otros programas. Sin embargo, si sólo vamos a trabajar con MATLAB son la mejor elección.

Su uso es muy sencillo, veámoslo con ejemplos. Se pueden guardar todas las variables del espacio de trabajo con la función save—alternativamente se puede utilizar la orden save:

```
>> clear
>> x = 6; y = 7;
>> who
Your variables are:
x  y
>> save('sesion.mat')  %guardar espacio de trabajo en sesion.mat
>> who -file sesion    %consultar variables guardadas en sesion.mat
Your variables are:
x  y
>> clear
>> x
Undefined function or variable 'x'.
>> load('sesion.mat')  %se cargan las variables de sesion.mat
>> who
Your variables are:
x  y
>> x
x =
     6
```

En el listado anterior se utiliza la función save para guardar las variables del espacio de trabajo en el archivo MAT *sesion.mat* en el directorio de trabajo. Después se utiliza la opción file de la orden who para consultar qué variables almacena el archivo. Tras borrar las variables con clear se utiliza la función load para cargar en el espacio de trabajo las variables almacenadas en el archivo MAT *sesion.mat*.

También es posible guardar sólo algunas variables en un archivo MAT, añadir variables a un archivo MAT y cargar sólo algunas variables de un archivo MAT:

```
>> a = 1; b = 2; c = 3;
>> save('sesion2.mat', 'a', 'c')        %solo se guardan a y c
>> who -file sesion2
Your variables are:
a  c
>> save('sesion2.mat', 'b', '-append') %se añade b
>> who -file sesion2
Your variables are:
a  b  c
>> clear
>> load('sesion2.mat','c')              %se carga solo c
>> who
Your variables are:
c
```

Para guardar sólo algunas variables hay que utilizar la siguiente sintaxis:

```
save(nombrearchivo, 'var1', 'var2', ...)
```

donde nombrearchivo es un archivo con extensión *.mat* y después viene un listado de las variables separadas por comas. Cada variable se especifica mediante una cadena de caracteres que almacena el nombre de la variable. Para añadir variables la sintaxis es:

```
save(nombrearchivo, 'var1', 'var2', ..., '-append')
```

Si la lista de variables está vacía se guarda todo el espacio de trabajo. Por último, para cargar variables selectivamente la sintaxis es:

```
load(nombrearchivo, 'var1', 'var2', ...)
```

1.3. Entrada o lectura de datos

La función input permite leer información del teclado. Su sintaxis es la siguiente:

```
resultado = input(texto)
cadena = input(texto,'s')
```

Vamos a estudiar su funcionamiento mediante los ejemplos de uso de la Figura 1.2:

- La función input toma como primer parámetro una cadena de caracteres. En el primer ejemplo—línea 1—la cadena es `Introduzca un valor`. Esta cadena se muestra en la pantalla y se espera a que el usuario teclee el valor apropiado terminado por la pulsación de la tecla Intro— *Enter*. input analiza el valor leído de teclado y, si es correcto, lo

devuelve transformando la secuencia de caracteres leída a la representación adecuada; por ejemplo, a punto flotante si el valor leído es un número con decimales.

- El segundo ejemplo—línea 5—sólo ilustra que input es una función, lo que permite utilizar su valor de retorno—el valor leído de teclado—en una expresión. En el ejemplo, el valor devuelto por input se utiliza como parámetro de la función abs.
- En la línea 10 se introduce un valor que input no es capaz de procesar, pues no sigue la sintaxis esperada.
- La línea 15 muestra que input puede leer vectores si el usuario utiliza la sintaxis adecuada para introducir el vector—también es capaz de leer matrices.
- Las líneas 18–22 demuestran que se puede usar como entrada a input el nombre de una variable existente, en cuyo caso el valor almacenado en la variable es el que se toma como entrada.
- Los dos últimos ejemplos son sobre lectura de cadenas de caracteres. Éstas deben escribirse precedidas de comillas simples—línea 24—, salvo que se utilice como segundo parámetro de input el literal de cadena 's', en cuyo caso no hay que introducir las comillas al escribir la cadena de entrada.

1.4. Salida o escritura de datos

MATLAB dispone de varias funciones para mostrar datos en la pantalla. La más sencilla es disp, que toma como parámetro un valor y tras convertirlo en una cadena de caracteres lo muestra en la pantalla. Como ilustra la Figura 1.3 disp permite mostrar en la pantalla valores escalares, vectores, matrices y cadenas de caracteres. Si se quiere mostrar en la pantalla una cadena formada por la concatenación de valores numéricos y cadenas de caracteres entonces hay que utilizar funciones para formar una cadena con el resultado deseado e invocar a disp con la cadena resultado. Por ejemplo, el Guión 1.4 solicita al usuario dos números y muestra el resultado de su multiplicación como una cadena en la que se concatena información numérica y textual. La segunda línea del guión utiliza la función num2str—NUMber to STRing—que transforma un número de su representación numérica interna a una cadena de caracteres. También se utiliza el hecho de que una cadena de caracteres es un vector de caracteres y se puede concatenar; por ejemplo ['ab' 'cd'] produce 'abcd'.

Una función especialmente útil para mostrar una concatenación de números y texto es fprintf. Esta función tiene su origen en el lenguaje de programación C, su sintaxis es algo compleja pero resulta muy versátil. El lector puede obtener una descripción completa de su funcionalidad tecleando help fprintf en la ventana de órdenes. Aquí describiremos sus características más utilizadas. Su sintaxis es:

```
>> x = input('Introduzca un valor: ')
Introduzca un valor: -2
x =
    -2
>> x = abs(input('Introduzca un valor: '))
Introduzca un valor: -2
x =
     2
>> x = input('Introduzca un valor: ')
Introduzca un valor: 14a
 14a
  |
Error: Unexpected MATLAB expression.
>> v = input('Introduce un vector: ')
Introduce un vector: [2 4 6]
v =
     2     4     6
>> mi_altura = 1.78;
>> a = input('Introduce una altura: ')
Introduce una altura: mi_altura
a =
    1.7800
>> libro = input('Introduzca un libro: ')
Introduzca un libro: 'El proceso'
libro =
El proceso
>> libro = input('Introduzca un libro: ', 's')
Introduzca un libro: El proceso
libro =
El proceso
```

Figura 1.2: Uso de la función input.

```
fprintf('cadena con formato', expr1, expr2, ...)
```

El primer parámetro es una cadena de caracteres que se mostrará en la pantalla. La cadena puede contener especificadores de conversión que se sustituirán por el resultado de ir evaluando las expresiones que siguen al primer parámetro. Los especificadores de conversión más utilizados son:

- %f para números con decimales.
- %d para enteros.
- %s para cadenas de caracteres.
- %c para un carácter.

Se puede obtener un guión similar al Guión 1.4 con el siguiente código:

```
v = input('Introduce un vector con dos factores: ');
fprintf('%fx%f = %f\n', v(1), v(2), v(1)*v(2));
```

En este último guión se ha incluido al final de la cadena de formato el carácter de escape \n, que representa un salto de línea. Los caracteres de escape más importantes son:

- \n salto de línea.
- \t tabulador horizontal.
- '' comilla simple.
- %% signo de porcentaje.
- \\ barra invertida.

Por ejemplo:

```
>> fprintf('He''s gained 20%% more.\n')
He's gained 20% more.
```

También se puede especificar un ancho de campo en el especificador de conversión, que indica cuántos caracteres como mínimo se emplean al visualizar los datos. Veamos un ejemplo. Si ejecutamos el siguiente guión—en la siguiente sección se describe cómo ejecutar un guión:

```
fprintf('Edad  Sexo\n')
fprintf('----  ----\n')
fprintf('%4d  %-4c\n', 18, 'm')
fprintf('%4d  %-4c\n', 22, 'f')
```

se obtiene la salida:

```
>> disp(2^5)
    32
>> disp([pi 3; 4 5])
    3.1416    3.0000
    4.0000    5.0000
>> disp('El castillo')
El castillo
```

Figura 1.3: Uso de la función disp.

```
v = input('Introduce un vector con dos factores: ')
cadena = [num2str(v(1)) 'x' num2str(v(2)) ' = ' num2str(v(1)*v(2))];
disp(cadena);
```

Figura 1.4: Guión que concatena cadenas de caracteres y números.

```
Edad  Sexo
----  ----
  18  m
  22  f
```

Para especificar un ancho de campo hay que preceder el indicador de conversión con un número que indica el ancho del campo. Si el número es positivo se justifica a la derecha y si es negativo a la izquierda.

Para números en punto flotante también se puede especificar la cantidad de decimales; por ejemplo, %8.2f significa un campo de 8 incluyendo el punto y dos decimales. También se puede especificar la cantidad de decimales únicamente; por ejemplo, %.2f indica que se muestre dos decimales.

```
>> x = 1/3;
>> fprintf('%f\n', x)
0.333333
>> fprintf('%.2f\n', x)
0.33
>> fprintf('%9.4f\n', x)
   0.3333
```

```
%Guion superficie de un triangulo
%Entradas: la base y altura de un triangulo
%Salidas: la superficie del triangulo

%Instrucciones
clc    %borra la pantalla
clear %borra todas las variables del espacio de trabajo
base = input('Introduce la longitud de la base: ');
altura = input('Introduce la altura: ');
fprintf('La superficie es %f\n', base*altura/2);
```

Figura 1.5: Guión que utiliza las tres estructuras secuenciales.

1.5. Ejemplo de guión

Para terminar el tema vamos a mostrar un guión que combina las tres estructuras secuenciales para calcular el área de un triángulo—véase la Figura 1.5.

Si ejecutas el guión e introduces una base de 15 y una altura de 7 debes obtener una superficie de 52.5. Se puede sustituir la última línea, que contiene la instrucción fprintf, por:

```
cadena = ['La superficie es ' num2str(base*altura/2)];
disp(cadena)
```

Para poder ejecutar el guión éste debe estar almacenado en un *archivo M*—es decir, un archivo de texto con extensión .m—en la carpeta de trabajo. El nombre del archivo es un identificador y, por tanto, debe verificar las reglas de escritura de un identificador—Sección 1.1.3. Para ejecutar el guión se puede escribir el nombre del archivo que lo contiene—sin el .m—en la ventana de órdenes. También es posible obtener una ayuda de lo que hace el guión usando la orden help y el nombre del archivo que contiene el guión—sin el .m:

```
>> help guion
  Guion superficie de un triangulo
  Entradas: la base y altura de un triangulo
  Salidas: la superficie del triangulo
```

Observa que la ayuda incluye todas las líneas iniciales consecutivas de comentarios.

1.6. Ejercicios

1. La calificación final de un estudiante es la media ponderada de tres notas: la nota de prácticas que cuenta un 30 % del total, la nota teórica que cuenta un 60 % y la nota de

participación que cuenta el 10 % restante. Escribe un programa que lea las tres notas de un alumno y escriba en la pantalla su nota final.

Puedes probar este programa con los siguientes datos: nota de teoría (7), nota de prácticas (5) y nota de participación (10). La calificación final para estos datos es 6.7.

2. Escribe un programa que lea los dos catetos de un triángulo rectángulo y muestre en la pantalla la hipotenusa.

3. Escribe un programa que calcule la desviación estándar de cinco números:

$$\sigma = \sqrt{\frac{1}{4}\sum_{i=1}^{5}(x_i - \overline{x})^2}$$

Puedes comprobar el resultado usando la función std de MATLAB. Por ejemplo:

```
>> std([1 2 3 4 5]) %Desviacion estandar de 1,2,3,4,5
ans =
    1.5811
```

4. Utiliza la siguiente fórmula alternativa para calcular la desviación estándar de cinco números:

$$\sigma = \sqrt{\frac{1}{4}(\sum_{i=1}^{5} x_i^2 - 5\overline{x}^2)}$$

5. Un vector bidimensional puede representarse mediante sus coordenadas cartesianas: (x, y) o mediante sus coordenadas polares: r y θ. Escribe un guión que dadas las coordenadas polares de un punto calcule sus coordenadas cartesianas. Ten en cuenta la relación:

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned} \tag{1.1}$$

6. Cuando se compra un producto en una máquina expendedora y no se introduce el importe exacto, la máquina utiliza un programa para devolver el mínimo número de monedas. Escriba un programa—considerando únicamente monedas de 5, 10, 20 y 50 céntimos de euro—que lea de teclado el importe de un producto y la cantidad de dinero introducida por el comprador en la máquina y escriba en la pantalla las monedas devueltas por la máquina.

Nota: En MATLAB el cociente de la división entera x/y puede calcularse como floor(x/y) y el resto como rem(x,y).

7. Escribe un guión que solicite el nombre, la edad y el sexo—M o F—de una persona y almacene esta información en un archivo MAT.

8. Escribe un guión que lea del archivo MAT creado en el ejercicio anterior los tres datos que almacena y lo muestre en la pantalla. Una salida puede ser:

```
Nombre: Cristina
Edad: 22
Sexo: F
```

Tema 2
Arrays

Un *array* es una colección homogénea de datos, es decir, un *array* almacena una serie de datos del mismo tipo. Para acceder a los datos hay que utilizar el nombre del *array* y los índices o posiciones a los que se quiere acceder. Los *arrays* de una y dos dimensiones son comúnmente denominados vectores y matrices respectivamente. MATLAB destaca frente a otros lenguajes de programación por sus facilidades para el trabajo con *arrays*. Incluso en su nombre—MATrix LABoratory—se destaca la importancia del uso de matrices ; no en vano los vectores y matrices juegan un papel fundamental en la resolución de muchos problemas de ingeniería. En este tema se hace un repaso al trabajo con *arrays* en MATLAB.

2.1. Vectores

Un vector es un *array* unidimensional. En la Figura 2.1 se ilustra distintas formas de crear vectores, vamos a comentarlas:

- En la primera línea se crea un vector introduciendo sus elementos entre corchetes. Los elementos se separan con comas, pero también es posible utilizar espacios en blanco—pruébalo.
- En la línea 4 se utiliza el operador dos puntos para crear un vector formado por una secuencia de elementos, en este caso los números pares del 2 al 20. La sintaxis es inicio :incremento:fin. Si se omite el incremento se toma un incremento de uno. El incremento puede ser negativo, así 10:-3:2 genera el vector [10 7 4].
- Existen funciones que devuelven un vector. Una de ellas es linspace que genera una serie de números equidistantes entre dos límites. linspace es particularmente útil para visualizar una función en un intervalo. Por ejemplo, para crear un gráfico de la función x^2 en el rango $[-5, 5]$ se puede teclear lo siguiente:

```
datos = linspace(-5,5,20) ;
plot(datos,datos.^2) ;
```

```
>> v = [8, 10, 23]
v =
     8    10    23
>> v = 2:2:20
v =
     2     4     6     8    10    12    14    16    18    20
>> v = linspace(1,10,5)
v =
    1.0000    3.2500    5.5000    7.7500   10.0000
>> v = zeros (1,5)
v =
     0     0     0     0     0
>> v = ones(1,5)
v =
     1     1     1     1     1
>> v = rand(1,5)
v =
    0.8147    0.9058    0.1270    0.9134    0.6324
```

Figura 2.1: Creación de vectores.

- Las funciones zeros, ones, rand y randn crean una matriz; estas funciones tienen dos parámetros, el primero indica el número de filas y el segundo el número de columnas de la matriz creada. Especificando que la matriz tiene una única fila se obtiene un vector. rand genera números pseudoaleatorios uniformemente distribuidos en el intervalo abierto $(0, 1)$. randn genera números pseudoaleatorios que siguen una distribución normal de media 0 y desviación típica 1.

Las siguientes subsecciones describen cómo se puede trabajar con los elementos de un vector.

2.1.1. Longitud de un vector

Un vector es una matriz con una sóla fila—o una matriz con una única columna. La forma más sencilla de obtener la longitud de un vector es utilizar la función length que devuelve la dimensión mayor de un *array*. La Figura 2.2 muestra el uso de la función length. Alternativamente se puede utilizar la función size que devuelve las distintas dimensiones de un *array*. En el ejemplo, como el argumento de size es un vector fila la variable nc almacenará la longitud del vector. La penúltima instrucción de la Figura 2.2 ilustra cómo se puede crear un vector sin elementos—su longitud es cero.

```
>> length([5 9 3])
ans =
     3
>> [nf,nc] = size([5 9 3])
nf =
     1
nc =
     3
>> v = []
v =
     []
>> length(v)
ans =
     0
```

Figura 2.2: Longitud de un vector.

2.1.2. Acceso a los elementos de un vector

Se puede acceder a los elementos de un vector utilizando un vector numérico o lógico.

Indexado numérico

En el indexado numérico se utiliza un vector para especificar los índices o posiciones del vector a los que se quiere acceder. Hay que tener en cuenta que en MATLAB los índices de un vector comienzan en 1; por lo tanto, si se accede—por ejemplo—al elemento de índice 2 se está accediendo al segundo elemento del vector. Si se accede para consultar los valores del vector los índices deben existir. En la Figura 2.3 se muestran cuatro ejemplos de consulta de los valores de un vector. En el primer ejemplo se utiliza un índice individual. En los dos siguientes se utiliza un vector para especificar los índices. Es posible utilizar el operador : para generar el vector de índices. Al indexar se puede utilizar la palabra reservada end para indicar la posición del último elemento de un vector.

Se puede modificar los elementos de un vector utilizando el operador de asignación y especificando en la parte izquierda de la asignación los índices implicados. En la Figura 2.4 se muestran varias posibilidades. En la primera asignación se usa un único índice. En la segunda asignación se modifica el primer y último elemento del vector al valor 1. En la tercera asignación se modifica el primer y último elemento del vector a los valores 8 y 9 respectivamente. La última asignación asigna un elemento que no existía en el vector original, observa que el vector ha crecido y que los valores entre las posiciones 7 y 9 se han rellenado con ceros. Cuando se asignan elementos a un vector, en la parte derecha de la asignación—tras el operador =—se puede utilizar:

```
>> v = [8 2 1 5 4 7]
v =
     8     2     1     5     4     7
>> v(2) %elemento de indice 2
ans =
     2
>> v([2 4]) %elementos de indices 2 y 4
ans =
     2     5
>> v(1:2:end) %elementos de indices impares
ans =
     8     1     4
>> v(end) %ultimo elemento
ans =
     7
>> v(length(v)) %ultimo elemento
ans =
     7
```

Figura 2.3: Indexado numérico de un vector.

- un único valor. En cuyo caso ese valor se asigna a todos los valores especificados en la parte izquierda de la asignación. Por ejemplo: `v([1 end]) = 1`. Se asigna 1 al primer y último elementos del vector.
- un vector. Entonces el vector de la parte izquierda debe tener el mismo tamaño y los elementos se asignan uno a uno. Por ejemplo: `v([1 end]) = [8 9]`. Se asigna 8 al primer elemento del vector y 9 al último.

Indexado lógico

Es posible indexar un vector utilizando un vector de valores lógicos o *booleanos*. Por ejemplo, en la Figura 2.5 se ha creado un vector de valores lógicos llamado `b` y en la última orden se utiliza para indexar el vector `v`. Observa dos cosas: 1) las posiciones en las que el vector lógico tiene el valor verdadero son las que se seleccionan y 2) la longitud de los dos vectores no tienen por qué coincidir, pero el vector lógico no puede tener un valor de verdadero en una posición en la que el vector indexado no contenga elementos.

La Figura 2.6 ilustra la gran utilidad del indexado lógico. Con la expresión $v \geq 0$ se obtiene un vector lógico con valor verdadero en aquellas posiciones del vector `v` con elementos no negativos. A continuación se utiliza el vector lógico para indexar el vector `v` y obtener los valores no negativos del vector. Como ejercicio intenta seleccionar los valores pares del

```
>> v = [8 2 1 5 4 7]
v =
     8     2     1     5     4     7
>> v(2) = 1
v =
     8     1     1     5     4     7
>> v([1 end]) = 1
v =
     1     1     1     5     4     1
>> v([1 end]) = [8 9]
v =
     8     1     1     5     4     9
>> v(10) = 1
v =
     8     1     1     5     4     9     0     0     0     1
```

Figura 2.4: Modificación de un vector.

```
>> v = [-2 3 -1 -4 5]
v =
    -2     3    -1    -4     5
>> b = [true false true false]
b =
     1     0     1     0
>> v(b)
ans =
    -2    -1
```

Figura 2.5: Indexación con un vector lógico.

```
>> v = [-2 3 -1 -4 5]
v =
    -2     3    -1    -4     5
>> v ≥ 0
ans =
     0     1     0     0     1
>> v(v≥0) %valores no negativos de v
ans =
     3     5
>> length(v(v≥0)) %cantidad de valores no negativos en v
ans =
     2
>> sum(v≥0) %cantidad de valores no negativos en v
ans =
     2
```

Figura 2.6: Extracción de información lógica de un vector.

vector v—sugerencia: prueba rem(v,2)==0. En la Figura 2.6 observa que la expresión sum(b), donde b es un vector lógico, produce la cantidad de valores verdaderos en el vector b.

Una función relacionada con los vectores lógicos es find. Esta función es relativamente compleja, puedes consultar su funcionalidad con help find, aplicada a un vector lógico devuelve las posiciones del vector con valores de verdadero. Por lo tanto, el código de la Figura 2.7 produce resultados análogos al código de la Figura 2.6.

2.1.3. Borrado de elementos de un vector

Los vectores de MATLAB son dinámicos, es decir, pueden cambiar de tamaño cuando se ejecuta MATLAB. Para borrar uno o varios elementos hay que asignarles el vector vacío. Por ejemplo, el siguiente código:

```
>> v = 5:5:55
v =
     5    10    15    20    25    30    35    40    45    50    55
>> v(2) = []
v =
     5    15    20    25    30    35    40    45    50    55
>> v([1:2:end]) = []
v =
    15    25    35    45    55
```

```
>> v = [-2 3 -1 -4 5]
v =
    -2     3    -1    -4     5
>> v ≥ 0
ans =
     0     1     0     0     1
>> find(v≥0)
ans =
     2     5
>> v(find(v≥0)) %valores no negativos de v
ans =
     3     5
>> length(find(v≥0)) %cantidad de valores no negativos en v
ans =
     2
```

Figura 2.7: Ejemplo de uso de find.

crea un vector, borra su segundo elemento y, después, de los elementos que quedan en el vector, borra los que ocupan posiciones impares.

2.1.4. Concatenación de vectores

Para concatenar vectores simplemente hay que utilizar los corchetes como en la creación de vectores. Por ejemplo:

```
>> v = [1 9]; %concatenacion simple
>> w = [2, 8] %concatenacion simple
w =
     2     8
>> x = [v w 6 3:-1:1] %concatenacion
x =
     1     9     2     8     6     3     2     1
```

Como ejercicio intenta formar un vector formado por los elementos pares seguidos de los impares de un vector de enteros. Dado el vector [1 2 4 5] debe generar el vector [2 4 1 5].

2.1.5. Operaciones con vectores

En esta sección vamos a estudiar los principales operadores y funciones que trabajan con vectores. Haremos una distinción entre operaciones aritméticas, lógicas y funciones que se aplican a vectores.

Operador	Tipo de operación
.^	potencia elemento a elemento
',^	traspuesta y potencia
+,-,~	operadores unitarios
.*,./,.\,*,/,\	Multiplicación, división y división inversa
+,-	suma y resta
:	Operador dos puntos
<, <=, >, >=, == y ~=	relacionales
&	AND elemento a elemento
|	OR elemento a elemento
&&	AND lógico
||	OR lógico

Tabla 2.1: Principales operadores de MATLAB en orden de precedencia

Operaciones aritméticas

Las operaciones aritméticas se pueden realizar colectivamente sobre los elementos de un vector siempre que los dos vectores tengan la misma longitud o uno de los vectores sea un escalar. La Figura 2.8 ilustra las principales operaciones aritméticas con vectores. Observa que los operadores *, / y ^ se reservan para la multiplicación, división y potencia de matrices respectivamente. Si se quiere trabajar elemento por elemento hay que utilizar los operadores .*, ./ y .^.

Operaciones lógicas

Se puede utilizar los operadores relaciones (<, <=, >, >=, == y ~=) para comparar dos vectores del mismo tamaño o un vector y un escalar produciendo un vector de valores lógicos—véase la Figura 2.9. También se pueden usar los operadores lógicos (&, |, ~) para realizar una operación lógica elemento a elemento sobre dos vectores lógicos de la misma longitud—Figura 2.10.

La Tabla 2.1 incluye los operadores de MATLAB ordenados por precedencia. Se puede utilizar los paréntesis para alterar el orden de ejecución de las subexpresiones.

Funciones de biblioteca

La mayor parte de las funciones de la biblioteca de MATLAB que son aplicables a escalares—como sqrt, sin o round—también son aplicables a vectores. Por ejemplo:

```
>> sqrt(25)
ans =
```

```
>> v = 1:3
v =
     1     2     3
>> w = 3:-1:1
w =
     3     2     1
>> v + 2 %vector mas escalar
ans =
     3     4     5
>> v + w %suma de vectores
ans =
     4     4     4
>> v .* w %producto elemento a elemento
ans =
     3     4     3
>> w' %traspuesta
ans =
     3
     2
     1
>> v * w' %multiplicacion de matriz (1x3) por matriz (3x1)
ans =
    10
>> v .^ 2 %potencia elemento a elemento
ans =
     1     4     9
>> v ./ w %division elemento a elemento
ans =
    0.3333    1.0000    3.0000
>> -v %cambio de signo elemento a elemento
ans =
    -1    -2    -3
```

Figura 2.8: Operaciones aritméticas con vectores.

```
>> v = [6 0 3 4 5]
v =
     6     0     3     4     5
>> w = [5 2 7 4 4]
w =
     5     2     7     4     4
>> v > 4
ans =
     1     0     0     0     1
>> v ≥ w
ans =
     1     0     0     1     1
```

Figura 2.9: Operaciones relacionales con vectores.

```
>> v = [true false false true]
v =
     1     0     0     1
>> w = [true false true false]
w =
     1     0     1     0
>> v & w % operador AND
ans =
     1     0     0     0
>> v | w % operador OR
ans =
     1     0     1     1
>> ~v % operador NOT
ans =
     0     1     1     0
```

Figura 2.10: Operadores lógicos con vectores.

```
     5
>> sqrt(4:4:16)
ans =
    2.0000    2.8284    3.4641    4.0000
>> radianes = linspace(0,2*pi,10);
>> plot(cos(radianes),sin(radianes)) %circunferencia de radio 1
>> radianes = linspace(0,2*pi,100);
>> plot(cos(radianes),sin(radianes)) %con mayor resolucion
```

También existen funciones que están pensadas para su aplicación sobre vectores y matrices. Entre ellas destacan sum, prod, mean y mode que devuelven la suma, el producto, la media y la moda respectivamente de los elementos de un vector. Por ejemplo, el siguiente código calcula la distancia euclídea entre dos puntos y el producto escalar de dos vectores bidimensionales:

```
>> v = [0 1]; w = [1 0];
>> sqrt(sum((v-w).^2)) %distancia euclidea
ans =
    1.4142
>> norm(v-w)          %distancia euclidea utilizando la norma (longitud)
ans =
    1.4142
>> sum(v.*w)          %producto escalar (otra forma es v*w')
ans =
     0
>> dot(v,w)  %la funcion interna dot realiza el producto escalar
ans =
     0
```

Otras funciones interesantes son min y max, que devuelven el mínimo y máximo repectivamente de los elementos de un vector así como el índice en que se encuentra dicho mínimo o máximo; sort, que ordena en orden creciente un vector o la ya comentada find.

```
>> v = [2 7 6 8 1];
>> [valor pos] = max(v)
valor =
     8
pos =
     4
>> sort(v)
ans =
     1     2     6     7     8
```

cumsum y cumprod devuelven la suma y el producto acumulado respectivamente de los elementos de un vector:

```
>> cumsum(2:6)
ans =
     2     5     9    14    20
>> cumprod(1:5) % factoriales del 1 al 5
ans =
     1     2     6    24   120
```

Por último, la función cross realiza el producto vectorial de dos vectores de longitud 3.

2.2. Matrices

Una matriz es un *array* bidimensional. En la Figura 2.11 se muestran distintas formas de crear una matriz, vamos a comentarlas:

- Se puede crear una matriz tecleando sus elementos y separando las filas con punto y coma o mediante saltos de línea.
- Las funciones zeros, ones y rand crean una matriz. rand genera números pseudoaleatorios uniformemente distribuidos en el intervalo abierto $(0, 1)$.
- Cuando se utiliza la función diag pasándole como parámetro un vector, ésta devuelve una matriz cuadrada cuya diagonal es el vector que recibe como parámetro.

Las siguientes subsecciones describen cómo se puede trabajar con los elementos de una matriz.

2.2.1. Tamaño de una matriz

La función size permite obtener las dimensiones de una matriz. El valor que devuelve size depende de la forma en que se la invoque. Los tipos de invocación más comunes son:

```
>> m = [1 2; 3 4; 5 6]
m =
     1     2
     3     4
     5     6
>> v = size(m)
v =
     3     2
>> [nf nc] = size(m)
```

```
>> m = [1  2; 3  4]
m =
      1      2
      3      4
>> n = [ 1  2.5  6
         7  8  10]
n =
    1.0000    2.5000    6.0000
    7.0000    8.0000   10.0000
>> M = ones(2,4)
M =
      1      1      1      1
      1      1      1      1
>> diag([2  4  6]) %o diag(2:2:6)
ans =
      2      0      0
      0      4      0
      0      0      6
```

Figura 2.11: Creación de matrices.

```
nf =
      3
nc =
      2
```

En la primera llamada a size se obtiene un vector con dos elementos que almacenan las dimensiones de la matriz, en la primera posición del vector se almacena el número de filas y en la segunda posición el número de columnas. En la segunda llamada se obtiene lo mismo, pero cada dimensión se almacena en una variable distinta.

La función numel devuelve el número de elementos de una matriz:

```
>> m = [1:3;4:6]
m =
      1      2      3
      4      5      6
>> numel(m)
ans =
      6
>> numel(5:10)
ans =
      6
```

2.2.2. Acceso a los elementos de una matriz

A partir de una matriz se puede obtener cualquier tipo de submatriz. Por ejemplo:

```
>> m = [1 2; 3 4; 5 6]
m =
     1     2
     3     4
     5     6
>> m(2,2)          % acceso a un elemento
ans =
     4
>> m(3,:)          % tercera fila, equivale a m(3,1:end)
ans =
     5     6
>> m(:,2)          % segunda columna, equivale a m(1:end,2)
ans =
     2
     4
     6
>> m([1 3],:)      % primera y tercera filas
ans =
     1     2
     5     6
>> m(:,end)        % ultima columna
ans =
     2
     4
     6
>> m(end-1:end,:) % dos filas ultimas
ans =
     3     4
     5     6
```

En un contexto de indexación : es un abreviatura de `1:end`, es decir, especifica todas las posiciones de una dimensión de la matriz. También se puede indexar una matriz para modificarla en una instrucción de asignación:

```
>> m = [1 2; 3 4; 5 6]
m =
     1     2
     3     4
     5     6
>> m(2,2) = 10 % modifica un elemento
m =
```

```
      1      2
      3     10
      5      6
>> m(3,:) = 20 %toda la fila 3 a 20
m =
      1      2
      3     10
     20     20
>> m(:,2) = [2 3 4] %cambia la columna 2
m =
      1      2
      3      3
     20      4
>> m([1 3],:) = [4 8 ; 12 16] %cambia las filas 1 y 3
m =
      4      8
      3      3
     12     16
```

Hay que tener en cuenta que el valor asignado debe ser un escalar, en cuyo caso se asigna ese valor a toda la submatriz, o una submatriz de las mismas dimensiones de la submatriz a modificar. Una matriz puede cambiar de tamaño si asignamos un valor a un elemento que no existe, por ejemplo:

```
>> m = [1 1; 2 2]
m =
      1      1
      2      2
>> m(3,3) = 3
m =
      1      1      0
      2      2      0
      0      0      3
```

Observa que se rellenan con ceros los nuevos elementos necesarios para que la matriz sea rectangular.

2.2.3. Borrado de elementos de una matriz

Se puede eliminar elementos de una matriz siempre que la matriz resultado siga siendo rectangular. Por ejemplo:

```
>> m = [1 1; 2 2]
m =
```

```
     1     1
     2     2
>> m(1,1) = []
Subscripted assignment dimension mismatch.
 >> m(1,:) = []
m =
     2     2
```

2.2.4. Concatenación de matrices

MATLAB permite la concatenación de matrices:

- Horizontalmente, siempre que todas las matrices tengan el mismo número de filas: R = [A B ... F]
- Verticalmente, siempre que todas las matrices tengan el mismo número de columnas: R = [A; B; ... F]

Un ejemplo:

```
>> A = [2 3; 4 5];
>> B = [2.1 2.2];
>> [A B']
ans =
    2.0000    3.0000    2.1000
    4.0000    5.0000    2.2000
>> [A; B]
ans =
    2.0000    3.0000
    4.0000    5.0000
    2.1000    2.2000
```

2.2.5. Operaciones con matrices

Al igual que con los vectores, con las matrices se puede realizar operaciones aritméticas, lógicas y también se les puede aplicar funciones de biblioteca.

Operaciones aritméticas

Es aplicable todo lo comentado para los vectores, por ejemplo:

```
>> m = zeros(2,2)+3 %suma matriz-escalar
m =
```

```
     3     3
     3     3
>> m .* ones(2,2)*2 % multiplicacion elemento a elemento
ans =
     6     6
     6     6
>> m * ones(2,2)*2 % multiplicacion de matrices
ans =
    12    12
    12    12
```

Operaciones lógicas

Al igual que con los vectores, se puede aplicar operadores relacionales a las matrices numéricas y operadores lógicos a las matrices lógicas. Ejemplos:

```
>> m = [1 2 1; 3 1 2]
m =
     1     2     1
     3     1     2
>> m == 1           % operacion logica
ans =
     1     0     1
     0     1     0
>> m == [1 2 1; 3 3 2] % otra operacion logica
ans =
     1     1     1
     1     0     1
>> m = [-1 2; 3 -4]
m =
    -1     2
     3    -4
>> m < 0       % valores negativos
ans =
     1     0
     0     1
>> m(m<0) = 0 % pone a cero los valores negativos
m =
     0     2
     3     0
```

Las funciones true y false permiten crear matrices de valores verdadero y falso respectivamente. Por ejemplo:

```
>> false(2)
ans =
     0     0
     0     0
>> true(1,4)
ans =
     1     1     1     1
```

Funciones de biblioteca

Existe un gran número de funciones de biblioteca que se pueden aplicar a matrices:

```
>> cos([pi 0; pi/2 2*pi])
ans =
   -1.0000    1.0000
    0.0000    1.0000
```

Cuando se aplican a matrices, las funciones sum, prod, cumsum, cumprod, mean, min, max, all o any realizan su función por columnas, aunque existen opciones para que trabajen por filas. Por ejemplo:

```
>> m = [1 2 1; 3 1 2]
m =
     1     2     1
     3     1     2
>> sum(m)        %suma por columnas. Produce un vector fila
ans =
     4     3     3
>> sum(sum(m)) %suma todos los elementos
ans =
    10
>> sum(m,2)     %suma por filas. Produce un vector columna
ans =
     4
     6
>> sum(m')      %Alternativa a la suma por filas. Produce un vector fila
ans =
     4     6
>> max(m)       %maximo por columnas
ans =
     3     2     2
>> max(m,[],2) %maximo por filas
ans =
     2
```

```
     3
```

Por supuesto, MATLAB tiene una función para calcular el determinante de una matriz cuadrada. Otra función interesante es reshape, que permite cambiar las dimensiones de una matriz:

```
>> det([1 1 3; 2 2 2; 4 1 3]) %determinante
ans =
   -12
>> x = [1 2 3 4; 5 6 7 8]
x =
     1     2     3     4
     5     6     7     8
>> reshape(x, 4, 2) %x pasa a tener 4 filas y dos columnas
ans =
     1     3
     5     7
     2     4
     6     8
```

Otras funciones son:

- fliplr (m): obtiene la matriz simétrica de m respecto a un eje vertical—FLIP Left Right.
- flipud(m): obtiene la matriz simétrica de m respecto a un eje horizontal—FLIP Up Down.
- rot90: permite rotar en sentido horario o antihorario los elementos de una matriz.
- repmat: permite replicar el contenido de una matriz.

Las tres funciones primeras pueden ser muy útiles si la matriz representa una imagen o fotografía. La función repmat se utiliza con mucha frecuencia, veamos un ejemplo de uso:

```
>> repmat(4,1,4)
ans =
     4     4     4     4
>> repmat([1 1;2 2],2,3)
ans =
     1     1     1     1     1     1
     2     2     2     2     2     2
     1     1     1     1     1     1
     2     2     2     2     2     2
```

2.2.6. Acceso lineal a matrices

En la Sección 2.2.2 se explicó cómo acceder a los elementos de una matriz. En esta sección vamos a comentar otra posibilidad más, se trata de acceder a una matriz especificando una sóla dimensión, es decir, utilizando un único índice. Veamos cómo se transforma el espacio bidimensional de índices de una matriz a un espacio unidimensional. Dada una matriz, la primera columna forma los primeros índices lineales, seguidos de la segunda columna y así sucesivamente. Por ejemplo, dada la matriz:

$$\begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix}$$

la primera columna (ADG) forma los primeros índices (A-1,D-2,G-3), seguidos de la segunda (B-4,E-5,H-6) y tercera columna (C-7,F-8,I-9). Existen funciones como find que devuelven índices lineales. La Figura 2.12 muestra algunos ejemplos de acceso lineal a una matriz.

2.3. Escritura y lectura de matrices en archivos de texto

En la Sección 1.2 estudiamos cómo almacenar una variable—nombre y contenido—en un archivo MAT. Por lo tanto, es fácil almacenar una variable que contenga una matriz o un vector en un archivo MAT. En esta sección vamos a ver cómo guardar el contenido de una matriz en un archivo de texto y cómo leer un archivo de texto que contiene una matriz. La sintaxis es muy parecida a la empleada con los archivos MAT.

Veámoslo con un ejemplo:

```
>> m = rand(2,2)
m =
    0.9575    0.1576
    0.9649    0.9706
>> save('datos.txt','m','-ascii')
>> type datos.txt
  9.5750684e-001  1.5761308e-001
  9.6488854e-001  9.7059278e-001
>> save('datos2.txt','m','-ascii','-double')
>> type datos2.txt
  9.5750683543429760e-001  1.5761308167754828e-001
  9.6488853519927653e-001  9.7059278176061570e-001
```

Para guardar una matriz en un archivo utilizamos la función save, usando como parámetros tres cadenas de caracteres: el nombre del archivo de texto, el nombre de la variable que

```
>> m = [-1  2; 3  -4; -5  -6]
m =
     -1      2
      3     -4
     -5     -6
>> m(4)  %acceso lineal a la posicion 4
ans =
      2
>> m(2:3)  %acceso lineal a las posiciones 2 y 3
ans =
      3     -5
>>  find(m<0)  %find devuelve un vector de indices lineales
ans =
      1
      3
      5
      6
>> m(find(m<0)) = 0  %asignar 0 a todos los valores negativos
m =
      0      2
      3      0
      0      0
>> sum(m(find(m>0)))  %suma los elementos positivos
ans =
      5
```

Figura 2.12: Acceso lineal a una matriz.

contiene la matriz y la constante -ascii. Podemos ver el contenido del archivo creado con cualquier editor de texto; en MATLAB también se puede utilizar la orden type para consultar el contenido de un archivo de texto. Observa que la matriz se guarda almacenando cada fila en una línea distinta y separando las columnas con espacios en blanco, los números siguen una notación exponencial. Fijémonos ahora en el segundo ejemplo de uso de save; utilizando la opción -double los números se almacenan con mayor exactitud, lo que producirá cálculos más precisos.

Se puede añadir datos a un archivo de texto utilizando la opción -append:

```
>> m2 = [1  1; 2  2]
m2 =
     1     1
     2     2
>> save('datos.txt','m2','-ascii','-append')
>> type datos.txt
  9.5750684e-001  1.5761308e-001
  9.6488854e-001  9.7059278e-001
  1.0000000e+000  1.0000000e+000
  2.0000000e+000  2.0000000e+000
```

Aunque la matriz añadida puede tener cualquier dimensión, si queremos que luego pueda ser leída en su totalidad deberíamos utilizar una matriz que tenga el mismo número de columnas que la que ya está en el archivo.

Para leer una matriz almacenada en un archivo de texto se utiliza la función load:

```
>> load('datos.txt') %o load('datos.txt','-ascii')
>> datos
datos =
    0.9575    0.1576
    0.9649    0.9706
    1.0000    1.0000
    2.0000    2.0000
>> mat = load('datos.txt')
mat =
    0.9575    0.1576
    0.9649    0.9706
    1.0000    1.0000
    2.0000    2.0000
```

Para que la lectura sea correcta el archivo debe tener el formato esperado. Observa que load guarda la matriz en una variable con el mismo nombre del archivo, sin la extensión. Si queremos elegir el nombre de la variable podemos utilizar la segunda sintaxis de ejemplo.

Alternativamente, se puede utilizar las órdenes save y load que tienen una sintaxis parecida. Otras funciones útiles son dlmwrite y dlmread, que permiten separar los valores con cualquier carácter:

```
>> m = [1 2; 3 4];
>> dlmwrite('M.txt', m, ';') %separa con ;
>> type M.txt
1;2
3;4
>> dlmread('M.txt',';')
ans =
     1     2
     3     4
>> dlmwrite('M2.txt', [4 5; 6 7], ',')
>> type M2.txt
4,5
6,7
```

Una ventaja importante de dlmwrite (DeLiMited WRITE) con respecto a save es que no hay que especificar la matriz mediante una cadena de caracteres, sino como una expresión.

Para intercambiar datos con una hoja de cálculo son útiles la funciones csvread, xlsread y xlswrite. Puedes utilizar la orden help o un manual para consultar su funcionamiento, así como otras opciones de save, load, dlmread y dlmwrite.

2.4. Visualización de matrices en la pantalla

Como se vio en la Sección 1.4 se puede utilizar la función disp para visualizar en la pantalla el contenido de una matriz:

```
>> m = [1:4; 5:8];
>> disp(m)
     1     2     3     4
     5     6     7     8
```

También se puede utilizar la función fprintf, aunque su uso resulte un tanto engorroso, pues visualiza los datos por columnas. La ventaja de fprintf frente a disp es que permite escribir datos en archivos de texto como veremos en el Tema 6, Sección 6.8. Veamos algunos ejemplos de uso de fprintf:

```
>> m = [1:2; 5:6];
>> fprintf('%d\n', m) %muestra los datos por columnas
1
5
```

```
2
6
>> fprintf('%d\n', m') %con la traspuesta muestra los datos por filas
1
2
5
6
```

Para mostrar todos los elementos por filas en una única línea se puede utilizar el siguiente código:

```
m = [1:2; 5:6];
fprintf('%d ', m')
fprintf('\n')
```

Al ejecutar este guión se obtiene la siguiente salida:

```
1 2 5 6
```

Para obtener una salida similar a la de disp utilizando fprintf hay que utilizar ciclos, que se describen en el siguiente tema. El guión:

```
m = [1:2; 5:6];
for fila = m'
    fprintf('%d ', fila)
    fprintf('\n')
end
```

utiliza un ciclo for produciendo la siguiente salida:

```
1 2
5 6
```

2.5. Matrices dispersas

Una matriz numérica mxn precisa mxnxt *bytes* para almacenar sus elementos, donde t es el número de *bytes* necesarios para almacenar un número—normalmente t valdrá ocho. Así, una matriz 1000x1000 de números de tipo *double* precisa 8Mb. Por lo tanto, una matriz puede consumir muchos recursos, tanto de memoria como de tiempo de procesamiento. Algunas matrices sólo tienen unos pocos valores distintos de cero. A este tipo de matrices se les llama *matrices dispersas*. MATLAB ofrece soporte para el trabajo con matrices dispersas, con el objeto de ahorrar tanto memoria como tiempo de procesamiento. Para ello sólo almacena los valores distintos de cero, junto con sus índices. A continuación, se describe brevemente el

trabajo con matrices dispersas, para un tratamiento en profundidad deberás consultar otras fuentes.

Se puede crear una matriz dispersa con la función sparse y la sintaxis:

```
sparse(filas,col,valores,nf,nc)
```

Crea una matriz nfxnc. Los vectores filas, col y valores especifican los valores de la matriz, de tal forma que M(filas(k),col(k)) vale valores(k). Por ejemplo:

```
>> M = sparse([1  3], [4  2], [2  6], 5, 5);
>> M
M =
   (3,2)         6
   (1,4)         2
>> full(M)
ans =
     0     0     0     2     0
     0     0     0     0     0
     0     6     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

Con el código previo se ha creado una matriz dispersa 5x5 con sólo dos valores distintos de cero. Concretamente M(1,4)=2 y M(3,2)=6. Observa que, por defecto, una matriz dispersa se visualiza mostrando los índices y valores distintos de cero. La función full permite la visualización tradicional de una matriz. Teclea spy(M) para obtener un gráfico que representa los valores no nulos de M. Si se quiere añadir nuevos valores no nulos a una matriz dispersa, se puede utilizar el siguiente código:

```
>> M = M + sparse(5,5,1)  %se incluye el valor M(5,5)=1
M =
   (3,2)         6
   (1,4)         2
   (5,5)         1
```

En el guión de la Figura 2.13 se compara el tiempo de ejecución necesario para elevar al cuadrado una matriz identidad de dimensión 1000, utilizando una representación normal y dispersa. En la Sección 3.8 del Tema 3 se explica el funcionamiento de las funciones tic y toc que permiten medir tiempos de ejecución. Observa que para crear una matriz identidad normal—es decir, no dispersa—se utiliza la función eye:

```
>> eye(3)
ans =
     1     0     0
```

```
dim = 1000;
inicio1 = tic;
M = eye(dim);
X = M^2;
tiempo1 = toc(inicio1);
fprintf('Con matriz normal: %.4f segundos\n', tiempo1)

inicio2 = tic;
M = sparse(1:dim,1:dim,1,dim,dim);
Z = M^2;
tiempo2 = toc(inicio2);
fprintf('Con matriz dispersa: %.4f segundos\n', tiempo2)
fprintf('Con matriz dispersa: %.2f veces más rápido\n',tiempo1/tiempo2)
```

Figura 2.13: Comparativa de elevar al cuadrado una matriz identidad normal y dispersa

```
     0     1     0
     0     0     1
>> sparse(1:3,1:3,1,3,3)
ans =
   (1,1)         1
   (2,2)         1
   (3,3)         1
```

En mi ordenador el cálculo de la potencia usando la matriz dispersa es más de 100 veces más rápido que el cálculo usando la matriz normal.

2.6. Generación de números aleatorios

Los números aleatorios se utilizan con frecuencia en programación. A veces se utilizan para probar los programas, como una forma de generar una entrada al programa. También se usan en simulaciones.

Los números aleatorios que genera un ordenador no son en realidad aleatorios, se generan utilizando un algoritmo que a partir de una *semilla*—un valor entero—va generando sucesivamente distintos números determinísticamente. Es decir, si se utiliza la misma semilla se generan los mismos números, aunque la secuencia generada no muestra patrones deterministas desde un punto estadístico. Por lo tanto, si se utilizan distintas semillas se pueden generar distintas secuencias sin patrones deterministas con lo que se obtiene un comportamiento relativamente aleatorio. Es por ello que frecuentemente se habla de *números pseudoaleatorios*.

MATLAB siempre utiliza la misma semilla al iniciar el generador de números aleatorios, pero se puede especificar una semilla distinta utilizando la función rng:

- rng('shuffle'): la semilla se elige "aleatoriamente" (teniendo en cuenta la hora del reloj).
- rng(entero): se especifica una semilla.
- rng('default'): se utiliza la semilla con la que MATLAB inicia el generador.

Cuando se depura un programa, por ejemplo una simulación, que trabaja con números aleatorios es interesante ejecutar siempre el programa con la misma secuencia de números aleatorios, de forma que el programa siempre produzca la misma salida. Para ello basta con utilizar la misma semilla, por ejemplo:

```
>> rng(150)    %se establece la semilla a 150
>> rand(1,6)
ans =
    0.9086    0.2580    0.8777    0.7390    0.6981    0.5172
>> rng(150)
>> rand(1,6)    %se restablece la semilla a 150
ans =
    0.9086    0.2580    0.8777    0.7390    0.6981    0.5172
```

2.6.1. Números aleatorios enteros

Se puede generar números aleatorios enteros uniformemente distribuidos utilizando rand:

```
>> round(rand(1,3)*5)       %3 numeros aleatorios del 1 al 5
ans =
     3     2     3
>>  round(rand(1,3)*5)+2    %3 numeros aleatorios del 2 al 7
ans =
     3     6     4
```

Sin embargo, la función randi está pensada exclusivamente para generar números aleatorios enteros uniformemente distribuidos, su sintaxis es ligeramente distinta a rand:

```
>> randi([5,10],2)        %matriz 2x2 de enteros en el rango [5,10]
ans =
     7     8
     6     7
>> randi([5,10], 2, 8)    %matriz 2x8 de enteros en el rango [5,10]
ans =
     7     7     5     9     7     9    10     5
```

```
     6     9     7     8     9    10     6     5
>> randi(20,2)            % matriz 2x2 de enteros en el rango [1,20]
ans =
     8    14
    20    14
```

2.7. Ejercicios

1. Escribe un programa que calcule la desviación estándar de los elementos de un vector:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

   No utilices la función std en el programa, pero hazlo para comprobar el resultado.

2. Utiliza la siguiente fórmula alternativa para calcular la desviación estándar:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}x_i^2 - n\overline{x}^2}$$

3. Genera una matriz 2x8 de números aleatorios uniformemente distribuidos:
   - En el intervalo (0,1).
   - En el intervalo (0,5).
   - En el intervalo (5,15).
   - En el intervalo [5,15], los números deben ser enteros.

4. Se está experimentando con varias técnicas de predicción de temperaturas. La siguiente matriz indica las predicciones de la temperatura máxima en los 7 días próximos. En cada fila se almacena las predicciones de una técnica. La primera columna de una fila indica el código de la técnica y las demás columnas las predicciones para los 7 días próximos.

$$\begin{pmatrix} 10 & 22{,}5 & 22{,}2 & 23{,}6 & 24{,}8 & 25{,}3 & 24{,}2 & 21{,}9 \\ 22 & 22{,}7 & 22{,}9 & 23{,}5 & 25{,}0 & 26{,}0 & 25{,}1 & 22{,}4 \\ 30 & 22{,}6 & 22{,}5 & 23{,}2 & 25{,}1 & 25{,}5 & 25{,}0 & 22{,}6 \\ 41 & 22{,}8 & 23{,}1 & 23{,}8 & 24{,}6 & 25{,}9 & 24{,}3 & 22{,}6 \\ 50 & 22{,}0 & 23{,}5 & 24{,}2 & 25{,}3 & 24{,}5 & 25{,}0 & 21{,}3 \end{pmatrix}$$

   transcurridos siete días se registran los valores verdaderos que son los siguientes:

22.3 22.4 23.2 24.9 25.0 24.7 22.0

Escribe fragmentos de código que permitan:

- Mostrar la temperatura máxima media prevista por cada método.
- Mostrar la temperatura máxima media prevista para cada día por los distintos métodos.
- Mostrar las temperaturas previstas por el método cuyo código es leído de teclado.
- Calcular el método que predice mejor. Se considera que el método que predice mejor es aquel cuya suma de los errores absolutos de las predicciones es menor.

5. En la siguiente matriz se guardan algunos resultados de un campeonato de fútbol:

$$\begin{pmatrix} 10 & 20 & 1 & 1 \\ 10 & 30 & 2 & 1 \\ 40 & 10 & 0 & 1 \\ 20 & 40 & 1 & 0 \\ 30 & 40 & 2 & 2 \end{pmatrix}$$

La primera columna almacena el código del equipo que juega en casa y la segunda columna el código del equipo visitante. La tercera columna indica el número de goles marcados por el equipo que juega en casa y la cuarta columna los goles del equipo visitante. Escribe un guión que permita:

- Dado el código de un equipo saber cuántos partidos ha jugado.
- Dado el código de un equipo saber cuántos puntos lleva.
- Equipos que jugaron el partido con más goles.

6. Escribe un guión que calcule el factorial de un entero no negativo. Sugerencia: la función prod calcula el producto de los elementos de un vector.

7. Realiza un guión que intercale los elementos de dos vectores de la misma longitud. Por ejemplo, dados los vectores [1 2 3] y [4 6 8] se debe generar el vector [1 4 2 6 3 8].

8. Escribe un guión que dado un vector de enteros calcule un vector con sus factoriales, para los valores negativos debe calcular el valor 0. Por ejemplo, dado el vector [-1 2 4 -3 5] debe calcular el vector [0 2 24 0 120]. Sugerencia: puedes utilizar la función factorial .

9. Escribe un guión que dado un vector de números genere un vector con los mismos números, pero sin valores negativos. Por ejemplo, dado el vector [-1 2 4 -3 5] debe calcular el vector [2 4 5].

10. Escribe un guión que dado un vector de números elimine el valor mínimo—puede haber más de uno. Por ejemplo, dado el vector [2 1 6 1 7] debe actualizarlo a [2 6 7].

11. Escribe un programa que dado un vector numérico genere un vector con los mismos valores pero normalizados al rango [0,1]. Por ejemplo, si el vector de entrada es [1 6 3 11], el vector normalizado es [0 0.5 0.2 1].

12. Escribe un guión que tenga un comportamiento análogo a la función linspace.

13. Existen muchos métodos numéricos capaces de proporcionar aproximaciones a Π. Uno de ellos es el siguiente:

$$\Pi = \sqrt{\sum_{i=1}^{\infty} \frac{6}{i^2}}$$

    Crea un programa que lea el número de términos de la sumatoria y calcule un valor aproximado de Π.

14. La función diff calcula las diferencias entre elementos consecutivos de un vector. Experimenta con su uso y escribe un guión que implemente su funcionalidad.

15. La función sign toma como parámetro una matriz numérica y sustituye cada elemento e por:

    - -1, si $e < 0$
    - 0, si $e = 0$
    - 1, si $e > 0$

    Escribe un guión que actúe de forma parecida a sign.

Tema 3
Estructuras condicionales y repetitivas

Las estructuras condicionales y repetitivas—o iterativas o cíclicas—permiten la ejecución selectiva y repetida de código respectivamente. En este tema se estudian las instrucciones de MATLAB que dan soporte a este tipo de estructuras de programación.

3.1. Expresiones lógicas

Las estructuras condicionales e iterativas se basan en la evaluación de una condición lógica para determinar qué curso de ejecución seguir. Una expresión lógica está formada por una combinación de operadores, variables, valores literales y llamadas a funciones que, sintácticamente, producen un valor lógico. Entre estos elementos destacamos los siguientes:

- Operadores relacionales: <, <=, >, >=, == y ~=. Son operadores binarios que producen un resultado de tipo lógico.
- Operadores lógicos: &&, || y ~. Los dos primeros son operadores binarios y el último es un operador unitario. Aplicados a subexpresiones lógicas producen un resultado de tipo lógico.
- all y any. Aplicados a un vector de valores lógicos all devuelve un valor lógico indicando si todos los elementos del vector son verdaderos, mientras que any devuelve si al menos un elemento del vector es verdadero.

La Figura 3.1 ilustra el uso de expresiones lógicas. Una característica interesante de las operaciones AND y OR es que se ejecutan en cortocircuito. Esto significa que evalúan sus operandos de izquierda a derecha, no evaluando el operando derecho si no es preciso. Por ejemplo, en la Figura 3.1 al evaluar la condición $x \geq 10$ && $x \leq 15$ se evalúa primero la subexpresión $x \geq 10$ y, como ésta es falsa, ya no se evalúa la expresión $x \leq 15$, pues independientemente de su valor la expresión será falsa. Del mismo modo, en una operación OR, si la

```
>> x = 6; y = 8;
>> x ≥ 10 && x ≤ 15 %esta x en el rango [10,15]?
ans =
     0
>> mod(y,2) == 0 || mod(y,3) == 0 %es y un multiplo de 2 o de 3?
ans =
     1
>> v = [-2 -3 5];
>> all(v < 0) %son todos los elementos de v menores que cero?
ans =
     0
>> any(v < 0) %existe algun valor en v menor que cero?
ans =
     1
```

Figura 3.1: Expresiones lógicas.

subexpresión a la izquierda es verdadera entonces no se evalúa la subexpresión a la derecha. Esto no sólo es eficiente, sino que permite escribir código del estilo: if i ≤ length(v) && ... v(i) == 7. Observa que en esta expresión el orden de las subexpresiones es importante, pues la segunda accede a un elemento de un vector sólo si el índice no excede la longitud del vector.

3.2. Estructuras condicionales

Las estructuras condicionales permiten seleccionar qué código se ejecuta entre varios posibles. En MATLAB la instrucción condicional más general es if, que tiene la siguiente sintaxis:

```
if <expresion logica 1>
   bloque de codigo 1
elseif <expresion logica 2>
   bloque de codigo 2
   .
   .
   .
elseif <expresion logica n>
   bloque de codigo n
else
   bloque de codigo por defecto
end
```

```
v = input('Introduce un vector de notas: ');
if all(v ≥ 5)
    disp('Todos han aprobado')
elseif any(v == 10)
    disp('Algun suspenso, pero al menos un 10')
else
    disp('Alguien suspendio y nadie obtuvo un 10')
end
if v(1) == max(v)
    disp('La primera nota es la mayor')
end
```

Figura 3.2: Ejemplo de uso de if.

Las partes elseif y else son opcionales. Esta instrucción va evaluando las expresiones lógicas según su orden de aparición, si alguna es verdadera entonces ejecuta su código de bloque asociado y termina. En caso de que ninguna expresión sea verdadera y exista parte else, entonces se ejecuta su código asociado. El guión de la Figura 3.2 muestra un ejemplo de uso de if. Observa que el primer if siempre muestra un mensaje. Si todos han aprobado muestra un mensaje, si no es así, entonces muestra otro mensaje si al menos hay un 10 en las notas. Si no se verifica ninguna de esta condiciones se muestra un mensaje por defecto. Sin embargo, el segundo if mostrará un mensaje sólo si la primera nota del vector es la mayor, en otro caso no muestra nada pues no existe parte else.

La otra estructura condicional de MATLAB es switch. Esta instrucción es menos general que if y tiene la siguiente sintaxis:

```
switch <expresion>
   case <lista de casos 1>
      bloque de codigo 1
   case <lista de casos 2>
      bloque de codigo 2
   .
   .
   .
   case <lista de casos n>
      bloque de codigo n
otherwise
   codigo por defecto
end
```

La expresión debe de ser de tipo escalar o cadena de caracteres. Las listas de casos pueden limitarse a un único valor o una serie de valores encerrados entre llaves y separados

```
%Programa calculadora minima
op1 = input('Introduce el primer operando: ');
op2 = input('Introduce el segundo operando: ');
op = input('Introduce la operacion (+,-,*,x,/): ', 's');
switch op
    case '+'
        r = op1+op2;
    case '-'
        r = op1-op2;
    case {'*','x'}
        r = op1*op2;
    case '/'
        r = op1/op2;
    otherwise
        op = 'E';
end
if op == 'E'
    fprintf('Operacion incorrecta\n');
else
    fprintf('%.2f %c %.2f=%.2f\n',op1, op, op2, r);
end
```

Figura 3.3: Ejemplo de uso de switch.

por comas—hablando técnicamente los valores encerrados entre llaves son un *array* de celdas, concepto explicado en el Tema 7. La parte otherwise es opcional. La ejecución de la instrucción switch se realiza de la siguiente forma. En primer lugar se evalúa la expresión. A continuación se va comparando el valor asociado a la expresión con las distintas listas de casos según el orden de aparición de los casos. Si el valor de la expresión coincide con alguno de los valores de la lista de casos entonces se ejecuta su bloque de código asociado y se termina la ejecución. Si no se encuentra ninguna coincidencia con las listas de casos y se ha especificado la parte otherwise, entonces se ejecuta el bloque de código por defecto.

El guión de la Figura 3.3 contiene un ejemplo de uso de la sentencia switch. El guión solicita dos operandos y un operador, que debe ser el carácter +,-,*,x o /. La instrucción switch se utiliza para determinar qué operador se ha introducido y calcular la operación correcta. Observa el uso de las llaves en el tercer caso, pues se admiten dos caracteres (* y x) para el operador de multiplicación.

```
n1 = input('Introduce un número: ');
n2 = input('Introduce otro número: ');
if n1-n2
    disp('Los números son distintos')
else
    disp('Los números son iguales')
end
```

Figura 3.4: Guión que usa una expresión aritmética como una expresión lógica.

3.3. Expresiones aritméticas como expresiones lógicas

En MATLAB es posible utilizar una expresión aritmética donde se precisa una expresión lógica. En dicho caso se evalúa la expresión aritmética y, si su resultado es un número distinto de cero, se considera que la expresión lógica es verdadera, en otro caso la expresión lógica es falsa. El guión de la Figura 3.4 ilustra el uso de una expresión aritmética como expresión lógica. El programa lee dos números y determina si son distintos, para ello utiliza la expresión: n1-n2, que produce un valor distinto de cero si los números son diferentes. Por supuesto, el guión podría haber utilizado una expresión lógica como n1-n2 == 0 o como n1 $\neq$ n2.

También es posible utilizar un vector numérico como una expresión lógica. En dicho caso el resultado es verdadero si todos los valores del vector son distintos de cero, en otro caso el resultado es falso. Por ejemplo, el siguiente guión comprueba si todos los elementos de dos vectores de la misma longitud son distintos:

```
v1 = [2 3 1]; v2 = [2 3 4];
if v1-v2
    disp('Todos distintos')
else
    disp('Algunos iguales')
end
```

Se puede obtener el mismo resultado utilizando una expresión más fácil de entender como, all (v1$\neq$v2).

3.4. Un error muy común

En esta sección se describe un error que cometen algunos principiantes al programar en MATLAB. Se trata de escribir un código como el siguiente:

```
x = input('Introduce un valor en el rango [0,5]: ');
if 1 ≤ x ≤ 5
   disp('x esta en el rango [1,5]')
end
```

En este código la expresión lógica $1 \leq x \leq 5$ trata de comprobar si el valor almacenado en la variable x está en el rango $[1, 5]$; sin embargo, esto debe realizarse mediante una expresión como la siguiente: $1 \leq x$ && $x \leq 5$. No obstante, la expresión lógica $1 \leq x \leq 5$ es sintácticamente válida, por lo que MATLAB no mostrará ningún error al analizarla. La expresión $1 \leq x \leq 5$ se evalúa de la siguiente manera. En primer lugar se evalúa la subexpresión $1 \leq x$ que produce un valor de verdadero o falso en función del valor almacenado en x. A continuación se compara este valor lógico para ver si es menor o igual que 5; esto provoca que se realice una conversión implícita del valor lógico a entero. La conversión implícita de un valor de tipo lógico a uno de tipo entero produce un 0 para un valor falso y un 1 para un valor verdadero. Puesto que tanto 0 como 1 son menores que 5, la expresión $1 \leq x \leq 5$ es siempre verdadera, con independencia del valor almacenado en la variable x. Luego el código anterior siempre mostrará el mensaje `x esta en el rango [1,5]` independientemente del valor leído en x.

3.5. Estructuras repetitivas

Las estructuras repetitivas o iterativas, también llamadas ciclos o bucles, permiten ejecutar un bloque de código varias veces. MATLAB dispone de dos estructuras de este tipo, los ciclos for y while. Vamos a describir en primer lugar el ciclo for. Este tipo de ciclo es menos general que el ciclo while, su sintaxis es la siguiente:

```
for vcb = v
    bloque de codigo
end
```

vcb es una variable que llamaremos *variable de control del bucle* y v es un vector. El bloque de código se va a ejecutar tantas veces como elementos tenga el vector y en las distintas ejecuciones del bloque de código la variable de control del bucle va a tomar como valor los distintos elementos del vector. Por ejemplo:

```
suma = 0;
for x = [1 4 6]
    suma = suma + x;
    fprintf('%.2f ---> total: %.2f\n', x, suma);
end
```

en este guión se van a mostrar en la pantalla los distintos valores del vector [1 4 ... 6], así como la suma parcial de los elementos del vector. Podemos obtener una alternativa menos elegante al guión anterior generando un vector con los índices del vector:

```
suma = 0;
v = [1 4 6];
for ind = 1:length(v)
    suma = suma + v(ind);
    fprintf('%.2f ---> total: %.2f\n', v(ind), suma);
end
```

Nota: Hemos visto que el ciclo for permite iterar por los elementos de un vector. Éste es su uso más habitual, aunque realmente permite iterar por las columnas de una matriz. Por ejemplo, el siguiente fragmento de código muestra la suma de las columnas de una matriz:

```
i = 1;
for col = [-1 4; 6 8]
    fprintf('Suma de la columna %d: %.2f\n', i, sum(col));
    i = i + 1;
end
```

Pasemos ahora al ciclo while, cuya sintaxis es:

```
while <expresion logica>
    bloque de codigo
end
```

La expresión lógica se evalúa al llegar la ejecución del programa al ciclo. Si es verdadera se ejecuta el bloque de código y se repite la evaluación-ejecución mientras que la condición de entrada al ciclo sea verdadera. El guión anterior se puede expresar mediante un ciclo while así:

```
suma = 0;
v = [1 4 6];
ind = 1;
while ind ≤ length(v)
    suma = suma + v(ind);
    fprintf('%.2f ---> total: %.2f\n', v(ind), suma);
    ind = ind + 1;
end
```

Este guión es menos elegante que los basados en el ciclo for, lo que resulta lógico si tenemos en cuenta que el ciclo for está pensado para recorrer los elementos de un vector. Sin embargo, la instrucción while permite expresar cualquier tipo de iteración. Por ejemplo, el

```
%Programa calculadora minima
continuar = 's';
while continuar == 's'
    op1 = input('Introduce el primer operando: ');
    op2 = input('Introduce el segundo operando: ');
    op = input('Introduce la operacion (+,-,*,x,/): ', 's');
    switch op
        case '+'
            r = op1+op2;
        case '-'
            r = op1-op2;
        case {'*','x'}
            r = op1*op2;
        case '/'
            r = op1/op2;
        otherwise
            op = 'E';
    end
    if op == 'E'
        fprintf('Operacion incorrecta\n');
    else
        fprintf('%.2f%c%.2f=%.2f\n',op1, op, op2, r);
    end
    continuar = input('Quieres continuar (s/n): ', 's');
end
```

Figura 3.5: Ejemplo de uso de while.

guión de la Figura 3.5 permite que el usuario ejecute de forma reiterada cálculos sencillos. Esta iteración no se puede expresar mediante un ciclo for.

3.5.1. Las instrucciones break y continue

Las instrucciones break y continue permiten modificar el flujo de ejecución de un ciclo. Algunos autores critican el uso de estas instrucciones, porque según ellos generan programas menos legibles. En mi opinión, el uso adecuado de estas instrucciones produce programas más fáciles de leer, entender y, en consecuencia, mantener; pero su uso incorrecto tiene el efecto contrario.

Vamos a empezar con break. La ejecución de la instrucción break provoca el término de la ejecución del ciclo más interno que la contiene. Supongamos que queremos calcular si un vector está ordenado de forma creciente. Esto equivale a comprobar que cada elemento del vector es mayor o igual que el elemento que le precede en el vector o, dicho de otro modo,

```
v = [1 3 5 7 6 8]; %vector de prueba
ordenado = true;
for x = 2:length(v)
    if v(x) < v(x-1)
        ordenado = false;
    end
end
if ordenado
    disp('Vector ordenado')
else
    disp('Vector desordenado')
end
```

Figura 3.6: Comprueba si un vector está ordenado. Versión 1

que el vector no contiene un elemento menor que el elemento que le precede en el vector. El guión de la Figura 3.6 implementa esta idea. Sin embargo, no es eficiente. El programa recorre todo el vector buscando si alguno de sus elementos es menor que el elemento que le precede. Sin embargo, en cuanto se descubre que un elemento verifica esta condición ya sabemos que el vector no está ordenado crecientemente y podemos dejar de recorrer el vector. Esta idea se incorpora en los guiones de las Figuras 3.7 y 3.8. El primero utiliza un ciclo while, mientras que el segundo combina el uso de for y break.

También es posible comprobar si un vector está ordenado realizando una operación lógica entre vectores, el guión de la Figura 3.9 ilustra cómo hacerlo. Aunque la forma más sencilla es utilizar la función interna issorted.

La instrucción continue se utiliza con menos frecuencia que break. Al igual que break, continue debe escribirse en el interior de un ciclo. Al ejecutarse produce el término de la ejecución del bloque de instrucciones del ciclo e inicia la ejecución de una nueva iteración, siempre que se siga verificando la condición de ejecutar el ciclo. Por ejemplo, el siguiente ciclo utiliza la sentencia continue para crear un vector con los elementos no negativos de otro vector.

```
v = [2 -1 -4.5 6 -3.2 7.4];
pos = [];
for x = v
    if x < 0
        continue %pasa a la siguiente iteracion
    end
    pos(end+1) = x;
end
```

```
v = [1 3 5 7 6 8]; %vector de prueba
ordenado = true;
x = 2;
while x ≤ length(v) && ordenado
    if v(x) < v(x-1)
        ordenado = false;
    end
    x = x + 1;
end
if ordenado
    disp('Vector ordenado')
else
    disp('Vector desordenado')
end
```

Figura 3.7: Comprueba si un vector está ordenado. Versión 2

```
v = [1 3 5 7 6 8]; %vector de prueba
ordenado = true;
for x = 2:length(v)
    if v(x) < v(x-1)
        ordenado = false;
        break
    end
end
if ordenado
    disp('Vector ordenado')
else
    disp('Vector desordenado')
end
```

Figura 3.8: Comprueba si un vector está ordenado. Versión 3

```
v = [1 3 5 7 6 8]; %vector de prueba
if any(v(2:end)<v(1:end-1))
    disp('Vector desordenado')
else
    disp('Vector ordenado')
end
if all(v(2:end)≥v(1:end-1)) %alternativamente
    disp('Vector ordenado')
else
    disp('Vector desordenado')
end
```

Figura 3.9: Comprueba si un vector está ordenado. Versión 4

Se puede obtener el mismo resultado sin utilizar continue:

```
v = [2 -1 -4.5 6 -3.2 7.4];
pos = [];
for x = v
    if x ≥ 0
        pos(end+1) = x;
    end
end
```

De hecho, el cálculo se puede hacer sin utilizar ciclos, así: pos = v(v≥0).

3.6. Crecimiento dinámico de un vector

En esta sección vamos a describir dos formas de generar un vector que va creciendo dinámicamente. Para ilustrar las dos técnicas escribiremos código para invertir el contenido de un vector. La primera posibilidad ya se utilizó en la sección anterior al explicar el funcionamiento de continue. Consiste en utilizar la palabra reservada end al indexar el vector. Como sabemos, al utiliza end al indexar un vector se obtiene el último índice del vector:

```
v = [2 4 9 10]; %vector de prueba
r = [];
for ind = length(v):-1:1
    r(end+1) = v(ind);
end
disp(r)
```

Utilizando la expresión end+1 se obtiene el índice siguiente al último índice de un vector. La otra posibilidad es utilizar la concatenación de vectores:

```
v = [2 4 9 10]; % vector de prueba
r = [];
for ind = length(v):-1:1
    r = [r v(ind)];
end
disp(r)
```

3.7. Código vectorizado frente a ciclos

Una de las características de MATLAB es su gran soporte para el procesamiento de vectores y matrices. Como consecuencia, es posible realizar cálculos sobre vectores y matrices utilizando expresiones sencillas. En otros lenguajes de programación es necesario el uso de ciclos para realizar dichos cálculos. A este tipo de código que no utiliza ciclos para hacer cálculos con *arrays* se le llama *código vectorizado*. Por ejemplo, los fragmentos de código de la sección anterior que usan un ciclo para invertir el contenido de un vector se pueden escribir en MATLAB con el siguiente código vectorizado: disp(v(end:-1:1) o con disp(fliplr (v)).

En esta sección vamos a ver otro ejemplo de código vectorizado frente a código que utiliza ciclos. Supongamos que tenemos una matriz numérica y queremos calcular la media de sus elementos, así como el elemento de la matriz más cercano a la media. El guión de la Figura 3.10 utiliza ciclos para resolver el problema, un código parecido se utilizaría en la mayoría de lenguajes de programación. El primer ciclo anidado se utiliza para iterar por los elementos de la matriz acumulando sus valores para calcular la media. El segundo ciclo anidado se utiliza para calcular el elemento más próximo de la matriz a la media. Por cierto, en MATLAB es posible iterar por los elementos de una matriz sin utilizar índices. El primer ciclo anidado puede cambiarse por el siguiente:

```
suma = 0;
for col = m
    for el = col'
        suma = suma + el;
    end
end
```

El código vectorizado se lista a continuación:

```
m = [2 6 8; -3 4 2.2];          % datos de prueba
media = sum(sum(m))/numel(m);   % o mean(mean(m))
dif = abs(m - media);           % diferencias con respecto a la media
minimo = min((min(dif)));       % minimo de las diferencias
```

```
m = [2 6 8; -3 4 2.2]; %datos de prueba
[nf,nc] = size(m);
suma = 0;
for f = 1:nf
    for c = 1:nc
        suma = suma + m(f,c);
    end
end
media = suma/numel(m);

masCercano = m(1,1);
distancia = abs(m(1,1)-media);
for f = 1:nf
    for c = 1:nc
        if abs(m(f,c)-media) < distancia
            masCercano = m(f,c);
            distancia = abs(m(f,c)-media);
        end
    end
end
fprintf('La media es %f\n', media)
fprintf('El elemento más cercano a la media es el %f\n', masCercano)
```

Figura 3.10: Media de una matriz y elemento más cercano usando ciclos

```
posMin = dif == minimo;
masCercano = m(posMin);
fprintf('La media es %f\n', media)
fprintf('El elemento mas cercano a la media es el %f\n', masCercano)
```

El número de instrucciones se ha reducido drásticamente. ¿Qué código es mejor? En cuanto a elegancia, el código vectorizado es más corto, especialmente en la parte del cálculo de la media, y siempre es preferible un programa corto a uno largo. En cuanto a eficiencia, depende. El código vectorizado está precompilado, por lo que si calculamos un máximo, mínimo o una suma las funciones internas max, min o sum serán más rápidas que su código equivalente escrito mediante ciclos. En cuestiones de tiempo de ejecución lo mejor es experimentar con un cronómetro. En concreto, para estos dos guiones he tomados tiempos de ejecución empleando las funciones que se explican en la sección siguiente y utilizando una matriz aleatoria de dimensión 5000x5000. En mi ordenador, la versión vectorizada es más de 11 veces más rápida que la versión que usa ciclos. ¡Estudia la siguiente sección y prueba en tu ordenador!

3.8. Preasignación de memoria y toma de tiempos

En la Sección 3.6 se ha descrito cómo un vector puede crecer dinámicamente. A veces este tipo de crecimiento es necesario, pero, si se puede, se debe evitar. El motivo es la eficiencia. Los elementos de un *array* se almacenan en posiciones contiguas de memoria. Cuando un *array* crece, los nuevos elementos deben situarse tras los existentes. Si la zona de memoria ubicada tras el *array* está ocupada por otros datos, entonces hay que mover el *array* a una zona con suficiente espacio para almacenar a los nuevos elementos. Este desplazamiento de elementos en memoria ralentiza el tiempo de ejecución.

Si se sabe de antemano el tamaño final de un *array*, entonces se puede evitar el crecimiento dinámico. La solución es preasignar la memoria creando un *array* del tamaño final. Por ejemplo, en el problema de invertir el contenido de un vector conocemos de antemano el tamaño del vector solución, es el mismo tamaño que el vector original. Por lo tanto, podemos utilizar esta técnica de preasignación de memoria para obtener un código más eficiente, como el siguiente:

```
v = [2 4 9 10];            % vector de prueba
r = zeros(1,length(v));    % preasignacion de memoria
pos = 1;
for ind = length(v):-1:1
    r(pos) = v(ind);
    pos = pos + 1;
end
disp(r)
```

Vamos a experimentar con el efecto de la preasignación de memoria en el tiempo de ejecución de un programa. Comenzamos describiendo las funciones tic y toc, que sirven para cronometrar. El siguiente guión muestra su funcionamiento básico—usa la orden help para obtener una descripción completa:

```
inicio = tic;
n = input('Escribe tu nombre: ', 's');
tiempo = toc(inicio);
fprintf('%s, tardaste %.2f segundos en escribir tu nombre\n', n, tiempo)
```

La función tic inicia una toma de tiempos, devolviendo un identificador. Cuando llamamos a la función toc con dicho identificador como parámetro, termina el cronometraje y toc devuelve el tiempo transcurrido en segundos entre la llamada a tic y la llamada a toc. De este modo, el guión anterior muestra en pantalla el número de segundos que el usuario ha tardado en escribir su nombre.

```
v = 1:100000; % vector de prueba
inicio1 = tic;
r = [];
for ind = length(v):-1:1
    r(end+1) = v(ind);
end
tiempo1 = toc(inicio1);
fprintf('Sin preasignación: %.4f segundos\n', tiempo1)

inicio2 = tic;
r = zeros(1,length(v));  % preasignacion de memoria
pos = 1;
for ind = length(v):-1:1
    r(pos) = v(ind);
    pos = pos + 1;
end
tiempo2 = toc(inicio2);
fprintf('Con preasignación: %.4f segundos\n', tiempo2)
fprintf('Con preasignación: %.2f veces más rápido\n',tiempo1/tiempo2)
```

Figura 3.11: Comparando el tiempo de ejecución de código con y sin preasignación de memoria

La Figura 3.11 compara el tiempo de ejecución del código para invertir un vector sin y con preasignación de memoria. Observa que se ha utilizado un vector con cien mil elementos. La salida del guión en mi ordenador es la siguiente:

```
Sin preasignacion: 0.0383 segundos
Con preasignacion: 0.0012 segundos
Con preasignacion 31.46 veces mas rapido
```

La diferencia es significativa. Experimenta en tu ordenador ejecutando varias veces el guión y observa cómo varían los tiempos de ejecución. Ten en cuenta que el tiempo de ejecución de un programa depende de la carga de trabajo del ordenador, pues los distintos programas en ejecución se reparten el uso de los recursos del ordenador, incluida la CPU. Por lo tanto, cuando quieras tomar tiempos de ejecución es conveniente que tengas el menor número de programas en ejecución.

3.9. Uso de menús

Cuando un guión permite realizar cálculos reiteradamente éstos pueden presentarse al usuario en la forma de un menú. El guión de la Figura 3.12 ilustra el código típico para la creación

de un menú. Consiste en utilizar un ciclo, en cada iteración se realizan las siguientes acciones:

1. Se muestra al usuario un menú con las opciones a ejecutar utilizando las funciones disp o fprintf—líneas 4–9.
2. Se solicita al usuario que elija una opción mediante la función input—línea 10.
3. Se ejecuta la opción elegida por el usuario. Se suele utilizar una instrucción switch para determinar qué opción se eligió—líneas 11–20.

El ciclo se ejecuta reiteradamente hasta que el usuario elige que no desea realizar más cálculos. En el guión de la Figura 3.12 se utiliza la función clc para borrar la pantalla antes de visualizar el menú. Con el objeto de que el usuario pueda ver los cálculos realizados antes de que se borren al mostrar el menú, el guión mantiene los resultados en la pantalla hasta que se pulsa una tecla. Esto se consigue con la función pause, que detiene la ejecución del programa hasta que el usuario pulsa una tecla.

La función interna menu facilita la creación de menús. Esta función visualiza un menú en una ventana en modo gráfico, disponiendo las distintas opciones en botones y devolviendo la opción elegida por el usuario. menu toma como parámetros de entrada las cadenas de caracteres que indican las distintas opciones del menú. La primera cadena es un título con el que se preceden las opciones. El guión de la Figura 3.13 realiza la misma funcionalidad que el guión de la Figura 3.12, pero utilizando la función menu. La Figura 3.14 muestra el menú de ventana creado en el guión de la Figura 3.13. Ten en cuenta que menu devuelve un entero en el rango $[0, n]$, donde n representa el número de opciones del menú. Se devuelve cero si el usuario ha cerrado la ventana, en otro caso el número representando la opción elegida. Por ejemplo:

```
>> op = menu('menu','opcion1','opcion2')
op =
     2
```

En este caso el usuario ha elegido la opción segunda, cuya cadena asociada es `opcion2`.

3.10. Iterar por los elementos de una matriz

En la Sección 3.7 hemos comentado que MATLAB permite en muchas ocasiones usar código vectorizado donde en otros lenguajes de programación sería preciso usar ciclos. Sin embargo, hay ocasiones en que se necesita iterar por los elementos de una matriz. En ese caso existen varias posibilidades y en esta sección vamos a describir varias de ellas. En el guión de la Figura 3.15 se muestras cuatro posibilidades. Las cuatro alternativas realizan el mismo

```
v = [];
opcion = 1;
while opcion ≠ 5
    clc
    disp('1. Introducir vector')
    disp('2. Sumar elementos')
    disp('3. Multiplicar elementos')
    disp('4. Visualizar elementos')
    disp('5. Salir')
    opcion = input('Elige una opción: ');
    switch opcion
        case 1
            v = input('Introduce el vector: ');
        case 2
            fprintf('La suma es %.2f\n', sum(v))
        case 3
            fprintf('El producto es %.2f\n', prod(v))
        case 4
            disp(v)
    end
    if opcion ≠ 5
        disp('Pulsa una tecla para continuar ...')
        pause
    end
end
```

Figura 3.12: Programa gestionado con menú

```
v = [];
opcion = 1;
while opcion ≠ 0 && opcion ≠ 5
    opcion = menu('Elige una opción','Introducir vector','Sumar ...
       elementos','Multiplicar elementos','Visualizar ...
       elementos','Salir');
    switch opcion
        case 1
            v = input('Introduce el vector: ');
        case 2
            fprintf('La suma es %.2f\n', sum(v))
        case 3
            fprintf('El producto es %.2f\n', prod(v))
        case 4
            disp(v)
    end
end
```

Figura 3.13: Ejemplo de uso de la función menu

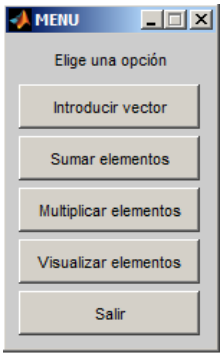

Figura 3.14: Ejemplo de ventana de menú creada con la función menu.

cometido: recorrer los elementos de una matriz y mostrar todos los elementos en una línea de texto en la pantalla.

Empecemos analizando las dos primeras formas de recorrer los elementos. Éstas implementan el tipo de recorrido que se realiza en la mayoría de lenguajes de programación. Se accede a los elementos por filas; las variables `f` y `c` van tomando los distintos índices de los elementos de la matriz. Las dos primeras formas de recorrido son casi iguales. La única diferencia es el modo en que obtienen las dimensiones de la matriz.

Si no es necesario calcular los distintos índices de la matriz, entonces se pueden emplear los dos últimos tipos de recorrido de la Figura 3.15. Observa que la última opción sólo emplea un ciclo y accede a los elementos usando indexación lineal—descrita en la Sección 2.2.6. Las dos últimos tipos de recorrido acceden a los elementos columna a columna, pero se puede transponer la matriz si se quiere acceder fila a fila.

3.11. Ejercicios

1. Realiza un guión que lea un vector de números e indique si todos los números están en el intervalo $[0, 10]$.

2. Realiza un programa que lea números hasta que se introduzca un cero. En ese momento el programa debe terminar y mostrar en la pantalla la cantidad de valores mayores que cero leídos.

3. Un número perfecto es un número natural que es igual a la suma de sus divisores positivos, sin incluirse él mismo. Por ejemplo, 6 es un número perfecto porque sus divisores positivos son: 1, 2 y 3; y 6 = 1 + 2 + 3. El siguiente número perfecto es el 28. Escribe un guión que lea un número natural e indique si es perfecto o no. Realiza un versión que use ciclos y otra que utilice código vectorizado.

4. Realiza un programa que calcule los cuatro primeros números perfectos. Nota: el resultado es 6, 28, 496, 8128.

5. Escribe un programa que dado un entero positivo n calcule su triángulo de Pascal asociado. Éste es una matriz $nx(2n-1)$ cuya primera fila consta de ceros, salvo la posición central que contiene un uno. El resto de elementos contienen la suma de los elementos a la izquierda y derecha de la fila superior. Por ejemplo, para $n = 5$ se tiene el siguiente triángulo:

0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0
0	0	1	0	2	0	1	0	0
0	1	0	3	0	3	0	1	0
1	0	4	0	6	0	4	0	1

```
m = [1 1 1; 2 2 2];

%Forma 1: ciclo anidado, recorre elementos por filas
[nf nc] = size(m);
for f = 1:nf
    for c = 1:nc
        fprintf('%d ', m(f,c))
    end
end
fprintf('\n')

%Forma 2: ciclo anidado, recorre elementos por filas
for f = 1:size(m,1)
    for c = 1:size(m,2)
        fprintf('%d ', m(f,c))
    end
end
fprintf('\n')

%Forma 3: ciclo anidado, recorre elementos por columnas
for col = m
    for dato = col'
        fprintf('%d ', dato)
    end
end
fprintf('\n')

%Forma 4: ciclo simple, recorre elementos por columnas
for ind = 1:numel(m)
    fprintf('%d ', m(ind))
end
fprintf('\n')
```

Figura 3.15: Diversas formas de iterar por los elementos de una matriz

6. Realice un programa que compruebe si el relleno de un Sudoku es válido.

   Análisis del problema: El objetivo del Sudoku es rellenar una matriz de 9x9 celdas (81 celdas), dividida en submatrices de 3x3 (también llamadas "cajas"), con las cifras del 1 al 9, partiendo de algunos números ya dispuestos en algunas de las celdas. No se debe repetir ninguna cifra en una misma fila, columna o caja.

   Un ejemplo de relleno válido de un Sudoku es el siguiente:

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

7. Realiza un guión que calcule la unión de dos vectores. En la implementación el vector solución debe crecer dinámicamente. Haz una comparativa de tiempos de ejecución entre tu algoritmo y la función interna union.

8. Implementa la funcionalidad del guión anterior utilizando menús. Incluye las siguientes opciones:

   - Introducir el primer vector.
   - Introducir el segundo vector.
   - Mostrar los dos vectores y la unión calculada por los dos métodos.
   - Realizar una comparativa de tiempos con vectores de distintos tamaños generados aleatoriamente.

9. Haz un guión que simule el lanzamiento de un dado n veces—usa la función randi. Cuenta las ocurrencias de cada número. Como randi genera números uniformemente distribuidos, el número de ocurrencias debe ser similar, especialmente al aumentar n—la frecuencia relativa debe ser próxima a $\frac{1}{6}$. Implementa una versión con ciclos y otra con código vectorizado.

10. Realiza un guión que dado el capital inicial de un plazo fijo y un interés anual fijo muestre en la pantalla cómo evoluciona el dinero invertido año a año, hasta llegar a un año en el que los intereses superen el 50 % del capital invertido. Un ejemplo de ejecución del guión es:

```
Introduce el capital inicial: 1000
Introduce el interes anual fijo: 5
Ano  capital
---  -------
  1  1050.00
  2  1102.50
  3  1157.63
  4  1215.51
  5  1276.28
  6  1340.10
  7  1407.10
  8  1477.46
  9  1551.33
```

11. Realiza un guión que genere una matriz aleatoria 1000x1000 de valores en el rango $[-0{,}5, 0{,}5]$. Escribe código para sumar los elementos por filas y por columnas. Toma tiempos para ver cuál es más rápido—debe ser la suma por columnas. Compara los resultados con el uso de la función interna sum.

Tema 4
Gráficos sencillos

Las facilidades para la generación de gráficos y el cálculo con matrices son dos de las características más destacables de Matlab. En este tema estudiamos cómo generar gráficos sencillos en dos dimensiones, limitándonos a explicar los usos más habituales de las funciones más importantes, pues las posibilidades gráficas de Matlab son tan amplias que su estudio requeriría unos apuntes exclusivos. Se invita al lector a que utilice la orden help para estudiar todas las opciones de las funciones vistas.

4.1. La función plot

La función más usada para crear gráficos en 2D es plot. Esta función es bastante versátil—prueba help plot—, siendo su uso más común con la sintaxis plot(x, y, cad), donde x e y son vectores de la misma longitud, que contienen las coordenadas x e y de los datos a visualizar, mientras que cad es una cadena de caracteres que especifica el modo de visualizar las líneas que unen los datos y los propios datos. Si se omite el parámetro x entonces las coordenadas x son los índices del vector y, es decir, 1:length(y). Veamos un ejemplo:

```
>> x = -2:2;
>> plot(x,x.^2)
```

Estas dos instrucciones visualizan la función x^2 para los valores [-2 -1 0 1 2]. Al ejecutar estas instrucciones se creará una ventana con la Figura 4.1. Esta ventana contiene un menú que permite editar el gráfico y cambiar su modo de visualización. Animamos al lector a que experimente con las opciones del menú. En estos apuntes se describirá cómo configurar algunas propiedades de un gráfico mediante instrucciones acopladas en un guión, de forma que la configuración sea reproducible.

Vamos a estudiar ahora las opciones de visualización que se especifican con el tercer parámetro de la función plot. Veremos dos ejemplos de uso y después la descripción general. La instrucción plot(x,x.^2, 'r*') visualiza sólo los puntos o datos, sin líneas que los conecten,

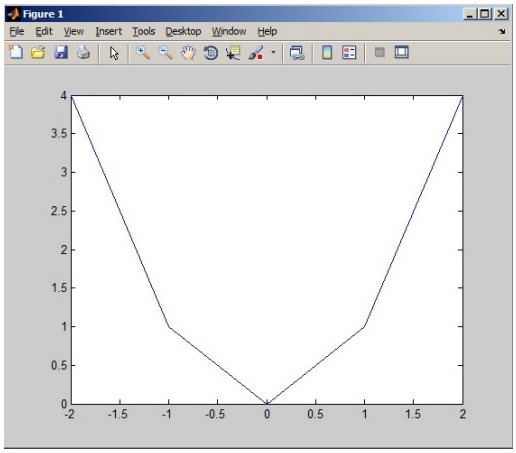

Figura 4.1: Ventana creada al utilizar la función plot.

de color rojo y con forma de asterisco. Pruébalo, obtendrás un gráfico como el izquierdo de la Figura 4.2. La instrucción plot(x,x.^2,'blacko--') visualiza líneas y puntos de color negro, los puntos con forma de círculo y las líneas discontinuas—gráfico derecho de la Figura 4.2.

A continuación listamos los posibles colores, tipos de línea y puntos. El formato por defecto es b-, es decir, azul, con línea continua y sin resaltar los puntos. Comencemos por los colores:

b blue
c cyan
g green
k black
m magenta
r red
w white
y yellow

Se puede especificar el carácter o el nombre completo del color. Listamos ahora los tipos de puntos:

o círculo
d diamante o rombo
h estrella de seis puntas (*hexagram*)
p estrella de cinco puntas (*pentagram*)
+ más
. punto
s cuadrado

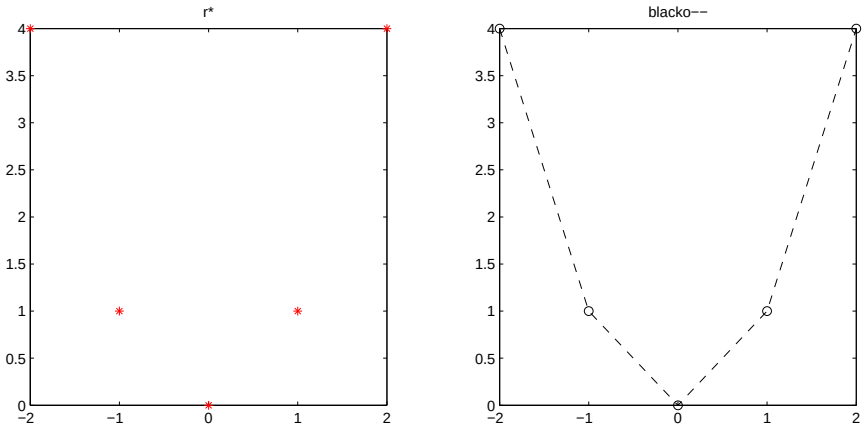

Figura 4.2: Dibujando la función x^2 con distintas opciones de visualización.

*	asterisco
v	triángulo hacia abajo
	triángulo hacia arriba
<	triángulo hacia la izquierda
>	triángulo hacia la derecha
x	x

Los tipos de línea se especifican con los caracteres:

--	discontinua con rayas
-.	discontinua con guiones y puntos
:	discontinua con puntos
-	continua

4.1.1. Gráficos paramétricos

Un gráfico no tiene que restringirse a una variable dependiente, normalmente sobre el eje y, que depende de una variable independiente visualizada en el eje x. En un gráfico paramétrico las variables en los distintos ejes dependen de una variable independiente. Esa variable independiente define un camino sobre la superficie de dibujo. Por ejemplo, el siguiente guión dibuja una circunferencia de radio 2:

```
radianes = linspace(0,2*pi,40);
radio = 2;
plot(cos(radianes)*radio, sin(radianes)*radio)
axis('equal')
```

4.1.2. Entrada gráfica

La función ginput permite capturar las coordenadas de un número ilimitado de puntos de la figura activa utilizando el ratón o las flechas de desplazamiento. Prueba a ejecutar:

```
>> x = -2:2;
>> plot(x,x.^2,'r*')
>> [x y] = ginput
x =
   -1.2949
    0.0323
    0.6129
y =
    2.9532
    1.7836
    3.1637
```

Se puede seleccionar puntos pulsando el ratón. Las coordenadas de los puntos seleccionados se guardan en $x(i), y(i)$. Pulsando la tecla *Intro* se termina la lectura de puntos. También es posible la sintaxis ginput(n) para leer n puntos.

4.2. Funciones para configurar la visualización de un gráfico

En esta sección describimos una serie de funciones y órdenes que permiten configurar el aspecto de un gráfico. Cuando se quiere visualizar un gráfico relativamente sofisticado es interesante escribir un guión con las instrucciones que lo generan. De esta forma el gráfico es reproducible, es decir, se puede visualizar siempre que se desee. Además, se puede editar el guión para realizar mejoras o modificaciones al gráfico. Veamos algunas de estas funciones en el siguiente guión que produce la Figura 4.3:

```
x = linspace(0,2*pi,20);
plot(x,sin(x),'ko-')
hold on
plot(x,cos(x),'bo-')
hold off
axis([-1 7 -1.5 1.5])
grid on
xlabel('x')
ylabel('Seno y coseno')
title('Comparativa seno y coseno')
legend('seno','coseno')
```

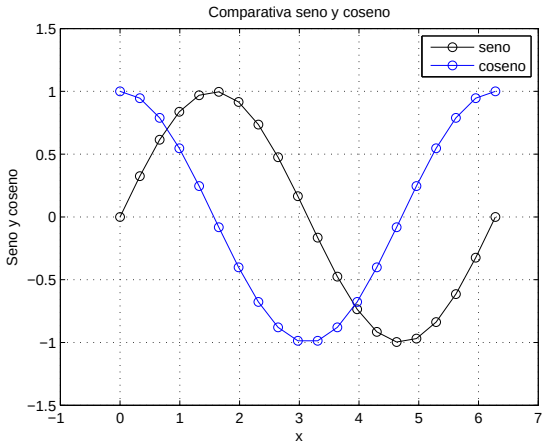

Figura 4.3: Comparativa de las funciones seno y coseno.

El guión produce un gráfico comparando las funciones seno y coseno en el intervalo $[0,2\pi]$. Las funciones se dibujan con líneas continuas, destacando con círculos los valores calculados de las funciones. La función seno se dibuja en negro y la función coseno en azul. Describamos las funciones y órdenes nuevas que incluye el guión:

- hold. Cuando se llama a la función plot, por defecto se borra el gráfico previo y se dibuja uno nuevo. Esto puede evitarse utilizando la opción on de la orden hold. Luego, hold on provoca que los nuevos gráficos se visualicen junto con los gráficos previos. hold off vuelve al comportamiento por defecto.

- axis. Por defecto, los valores máximos y mínimos de los ejes de coordenadas se establecen a los valores máximos y mínimos de los datos a visualizar. Con axis([xmin xmax ... ymin ymax zmin zmax]) se puede establecer estos valores. En caso de que el gráfico sea 2D el eje Z no debe especificarse. La función axis tiene muchas más posibilidades; por ejemplo: axis('off') elimina los ejes y axis('equal') hace que se utilice el mismo factor de escala en la pantalla para los ejes, lo que produce que una circunferencia se vea redonda y no como una elipse. Puedes consultar toda la funcionalidad de axis con la orden help.

- grid. Con grid on se visualiza una malla cuadrangular, con grid off se elimina.

- xlabel, ylabel, zlabel y title sirven para especificar un texto para la coordenada x, la coordenada y, la coordenada z y el título del gráfico respectivamente.

- legend. Sirve para crear una caja con leyendas de los distintos resultados de llamar a plot que aparecen en una figura. Por defecto, la caja aparece en la esquina superior

derecha, pero existen opciones para ubicarla en otras zonas, consulta la ayuda de la función si quieres averiguar cómo se hace.

Otra función interesante es text(x, y, {z,}, cadena) que sitúa un texto en la localización (x, y) en un gráfico 2D y en la (x, y, z) en un gráfico 3D. La función gtext('texto') sólo sirve en 2D y permite utilizar el ratón para situar el texto—¡pruébalo!.

La función plot permite visualizar más de un gráfico. Por ejemplo, se puede obtener un resultado análogo al guión previo sustituyendo las instrucciones en las líneas 2–5 por:

```
plot(x,sin(x),'ko-', x,cos(x),'bo-')
```

También se puede invocar a plot(x,y) siendo x y/o y una matriz. Si sólo x o y son matrices, entonces se dibujan las columnas de la matriz frente a los valores del vector, utilizando un color diferente para cada gráfico. Si tanto x como y son matrices entonces se dibuja cada columna de x frente a cada columna de y. Por ejemplo, podemos obtener una figura parecida a la obtenida previamente en esta sección con el código:

```
x = linspace(0,2*pi,20);
y(:,1) = sin(x);
y(:,2) = cos(x);
plot(x,y)
```

donde x es un vector e y una matriz. Por último, plot(m), donde m es una matriz, dibuja las columnas de m frente a los números de fila.

4.3. Varios gráficos en una figura: subplot

La función subplot permite visualizar varios gráficos en diferentes zonas de una figura. Con la sintaxis subplot(fila,col,n) se divide la figura actual en una matriz con $filaxcol$ áreas de dibujo y se establece la n-ésima como la actual para dibujar. La numeración de las áreas va de la fila superior a la inferior y dentro de cada fila de la izquierda a la derecha. El siguiente guión:

```
subplot(2,2,1)
plot(1:100)
title('x')
subplot(2,2,2)
plot(sqrt(1:100))
title('sqrt(x)')
subplot(2,2,3)
plot(log(1:100))
title('logartimo neperiano')
subplot(2,2,4)
```

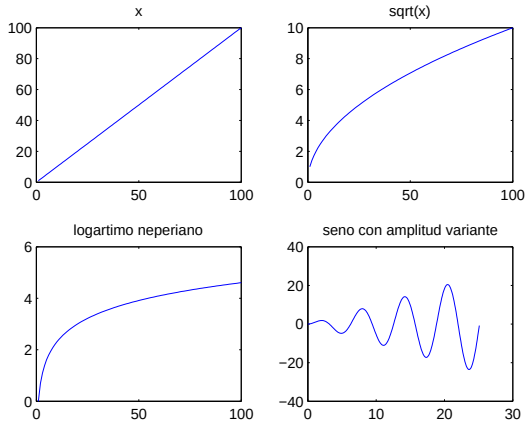

Figura 4.4: Ejemplo de uso de subplot.

```
x = 0:0.1:8*pi;
plot(x,sin(x).*x)
title('seno con amplitud variante')
```

divide la figura actual en una matriz $2x2$, donde se dibujan distintos gráficos, obteniéndose la Figura 4.4.

4.4. Gestión de varias figuras: figure

En Matlab se utiliza el término *figura*—*figure*—para designar a un contenedor en el que se puede dibujar. La función plot crea una figura nueva si no existe ninguna. En caso de que exista una, dibuja sobre ella, o la suplanta, dependiendo de cómo se haya usado hold. Es posible, sin embargo, trabajar con más de una figura. Para ello hay que utilizar la función figure, con figure(n) creamos una nueva figura, la Figura n, o, si ya existe, la seleccionamos como la figura activa, es decir, la figura en que se dibuja. El siguiente guión crea dos figuras y dibuja un gráfico en cada uno de ellas:

```
close all                   % cierra todas las figuras
x = 0:0.5:8*pi;
plot(x,sin(x),'o-');        % dibuja en la Figura 1
figure(2)                   % crea una nueva figura y la pone como activa
plot(x,sin(x)+x,'o-');      % dibuja en la figura activa (la Figura 2)
```

Como se comenta en el guión, close all cierra todas las figuras creadas. Otra función relacionada con la gestión de figuras es clf, que borra el contenido de la figura activa.

4.5. Distribuciones de frecuencias: bar, stem, pie e hist

En esta sección describimos cuatro funciones que se utilizan principalmente para generar un gráfico que permite visualizar la distribución de frecuencias de los datos de una muestra o una población. bar, stem y pie se utilizan con datos categóricos y numéricos discretos, mientras que hist su utiliza con valores numéricos continuos.

Empecemos con los datos discretos. bar produce un diagrama de barras, mientras que pie produce un diagrama de sectores. Supongamos que tenemos una clase con alumnos de 20 a 25 años y queremos obtener un gráfico de barras para observar visualmente la distribución de los alumnos por edad. Esto se puede hacer con las siguientes instrucciones:

```
>> edades = randi([20 25], 1, 20)
edades =
    24    22    20    21    20    21    22    23    22    25    23    ...
       25    23    25    21    24    21    24    24    20
>> ocurrencias = histc(edades,unique(edades))
ocurrencias =
     3     4     3     3     4     3
>> bar(unique(edades),ocurrencias)
```

La primera instrucción genera un vector con edades aleatorias uniformemente distribuidas entre 20 y 25 años, de esta forma generamos las edades de una forma sencilla. La segunda instrucción obtiene un vector con las ocurrencias de cada edad del vector edades, en orden creciente de edades. El resultado obtenido nos dice que en edades hay 3 elementos con el valor 20, 4 elementos con el valor 21 y así sucesivamente. La última instrucción llama a la función bar. El primer parámetro es un vector ordenado con las edades y el segundo es un vector con el número de ocurrencias de cada edad. El resultado de la ejecución de bar puede observarse en la Figura 4.5. Como los datos han sido generados aleatoriamente siguiendo una distribución uniforme el número de alumnos obtenidos de cada edad es parecido. La función unique toma como parámetro una matriz y devuelve sus elementos ordenados y sin repetidos en un vector.

La función barh produce un diagrama con barras horizontales, teclea barh(unique(edades),ocurrencias) y observa la figura generada. También se puede visualizar varios diagramas de barras usando bar—consulta la ayuda. La función stem es parecida a bar produciendo un diagrama de tallos y hojas—*stem plot*. stem(unique(edades),ocurrencias) produce la Figura 4.6.

La función pie genera un diagrama de sectores. Por ejemplo, el siguiente guión genera la Figura 4.7:

```
subplot(1,2,1)
pie(ocurrencias)
title('Con porcentajes')
subplot(1,2,2)
```

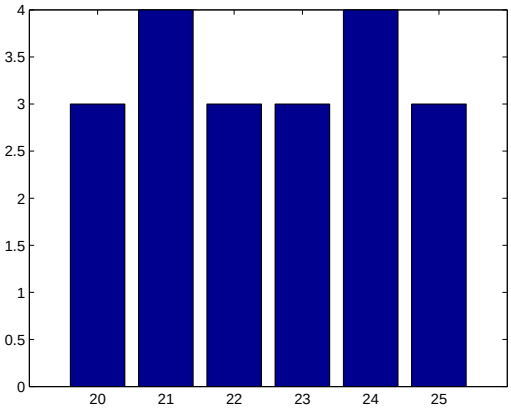

Figura 4.5: Diagrama de barras mostrando la distribución por edades.

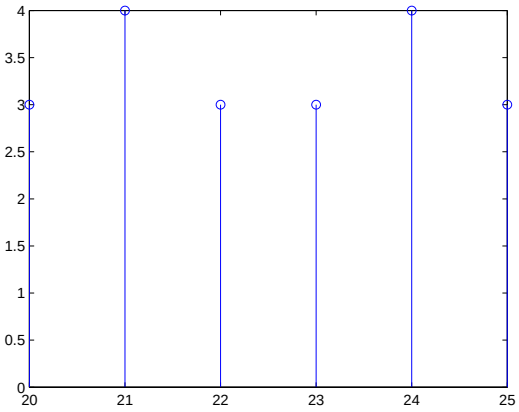

Figura 4.6: Diagrama de tallos y hojas mostrando la distribución por edades.

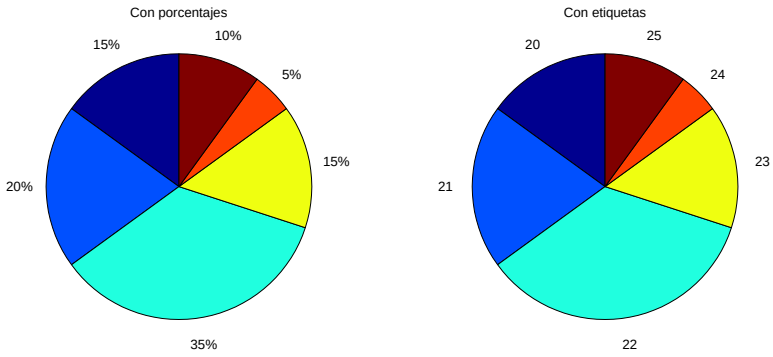

Figura 4.7: Diagrama de sectores con porcentajes y con etiquetas.

```
pie(ocurrencias, {'20','21','22','23','24','25'})
title('Con etiquetas')
```

En la primera llamada a pie se indican las ocurrencias de cada dato y éstas se etiquetan con su porcentaje de ocurrencia. En la segunda llamada se incluye un segundo parámetro con las etiquetas de las ocurrencias. Las etiquetas se especifican como un array de celdas, esta estructura de datos se explica en el Tema 7. En este ejemplo se han indicado las etiquetas manualmente, en la Sección 7.5.3 se muestra cómo hacerlo mediante código. En un diagrama de sectores el tamaño de cada sector es proporcional a la frecuencia de ocurrencia que representa.

Cuando se quiere visualizar la forma de la distribución de unos datos que se representan mediante valores numéricos continuos, como la altura de un grupo de personas, entonces no se puede utilizar un diagramas de barras, puesto que la frecuencia de ocurrencia de cada dato sería muy pequeña. En ese caso se utiliza un *histograma*, que, a partir de los datos, crea una serie de franjas y dibuja un diagrama de barras en el que la altura o el área de cada barra es proporcional a la cantidad de datos que hay en la franja asociada a la barra. En Matlab, hist(vector) crea un histograma con 10 franjas de la misma anchura e hist(vector,n) crea un histograma con n franjas. Por ejemplo:

```
>> alturas = 1.74 + randn(1,100)*0.05;
>> hist(alturas)
>> ocu = hist(alturas)
ocu =
     1     1    13    12    24    21    24     1     2     1
```

La primera instrucción crea un vector con 100 números aleatorios siguiendo una distribución normal de media 1.74 y desviación típica 0.05—simula 100 alturas distribuidas

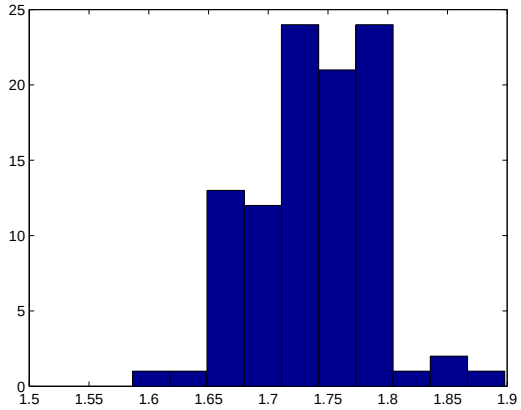

Figura 4.8: Histograma de alturas.

normalmente con media 1.74 y desviación típica de 5 centímetros. La segunda instrucción genera un histograma de los datos con 10 franjas, véase la Figura 4.8. La última instrucción muestra un segundo uso de la función hist en la que se obtiene un vector indicando el número de datos que caen en cada franja. Con la sintaxis:

```
>> [ocu, centros] = hist(alturas);
```

también se obtienen los centros de las franjas.

4.6. Otras funciones que generan gráficos 2D

Describimos aquí varias funciones que también permiten generar gráficos 2D. Empezamos con area, que permite visualizar el área que queda debajo de una función. Su sintaxis es area(x,y,nivel), donde x e y son vectores con las coordenadas x e y repectivamente; nivel indica la altura de la base del área y vale 0 por defecto. Por ejemplo, el siguiente guión muestra el área de la función $x^2 - 2x$ en el intervalo $[-5, 5]$ por encima de la línea $y = -2$.

```
x = linspace(-5,5,100);
area(x, x.^2-2*x, -2)
```

La función fill (x,y,color) produce un polígono relleno definido por las coordenadas x e y, el color se especifica mediante el tercer parámetro—un vector de tres elementos especificando la intensidad de los componentes rojo, verde y azul. El siguiente ejemplo genera un cuadrado rojo con esquinas $(-0{,}5, -0{,}5)$ y $(0{,}5, 0{,}5)$:

```
x = [-0.5  0.5  0.5  -0.5];
y = [-0.5  -0.5  0.5  0.5];
fill(x, y, [1 0 0])
axis([-1 1 -1 1])
```

La función polar(radianes, radios) tiene un comportamiento similar a plot, pero recibe como parámetros coordenadas polares, es decir, pares angulo–radio. El siguiente guión dibuja 20 puntos equidistantes en una circunferencia de radio 3 y centro el origen de coordenadas:

```
angulos = linspace(0,2*pi,20);
polar(angulos, repmat(3, 1, numel(angulos)), 'o')
```

4.7. Ejercicios

1. Un paseo aleatorio—*random walk*—es una formulación matemática que sirve para modelar ciertos fenómenos como los movimientos de un animal en busca de comida o el comportamiento de un índice bursátil. Un paseo aleatorio se especifica mediante la siguiente expresión:

$$x_t = x_{t-1} + a_t$$

donde a_t representa una variable aleatoria. En nuestro caso vamos a suponer que a_t sigue una distribución normal estándar—media 0 y desviación típica 1. Supondremos también que se está modelando el valor de una variable—x—a lo largo del tiempo. El primer valor es $x_1 = 0$. Por lo tanto, x_2 se calcula como $x_2 = x_1 + a_2$ y así sucesivamente. Realiza un guión que haga lo siguiente:

   - Genera 20 paseos aleatorios de longitud 100. Almacena cada paseo en una columna distinta de una matriz.
   - Genera un gráfico en que se observen conjuntamente los 20 paseos aleatorios.
   - Genera un histograma que refleje la distribución de x en los instantes 25, 50, 75 y 100. Usa subplot para generar los cuatro histogramas en la misma figura.

   Las Figuras 4.9 y 4.10 son un ejemplo de los gráficos que debe generar el guión. Escribe el guión de forma que el número de paseos aleatorios y la longitud de los mismos se almacenen en variables.

2. En un archivo de texto se encuentran almacenadas las notas de un examen. Cada nota aparece en una fila distinta. Las notas son numéricas y están en el rango [0,10]. Escribe un guión que realice un diagrama de barras, un diagrama de barras y hojas y un

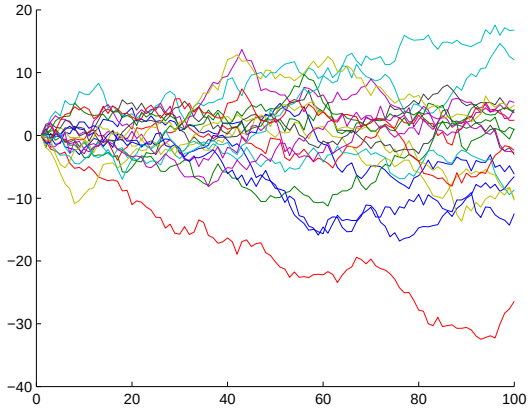

Figura 4.9: Paseos aleatorios.

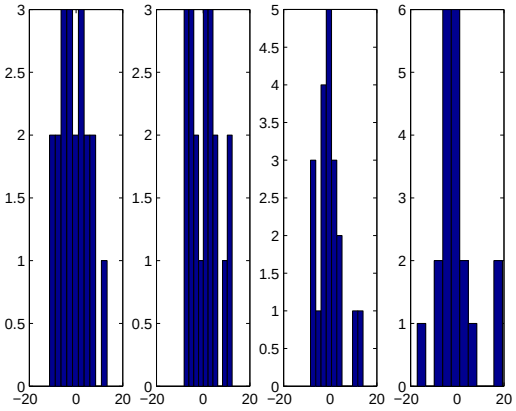

Figura 4.10: Histogramas de los paseos aleatorios en los tiempos 25, 50, 75 y 100.

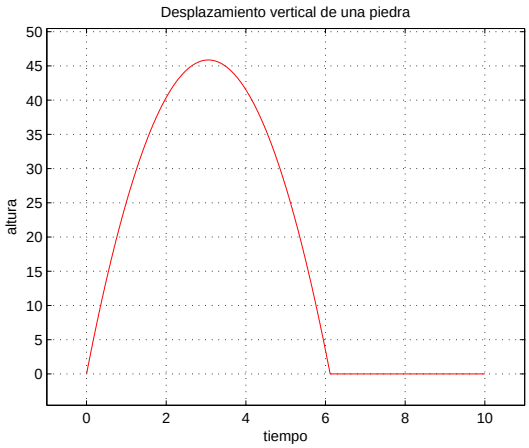

Figura 4.11: Desplazamiento vertical de una piedra en el tiempo.

diagrama de sectores con la distribución de notas del archivo. Los diagramas deben reflejar las notas como suspenso, aprobado, notable y sobresaliente.

3. Si se lanza una piedra en sentido vertical y hacia arriba con una velocidad inicial v_i, su desplazamiento vertical h tras un periodo de tiempo t viene dado por la fórmula $h = v_i t - gt^2/2$, donde g es la aceleración de la gravedad (9.81 m/s^2)—la resistencia del aire se ignora. Realiza un guión que solicite la velocidad inicial del lanzamiento y un tiempo t y calcule la altura que alcanza la piedra en distintos momentos del intervalo $[0, t]$, mostrando una gráfica de la evolución altura-tiempo—observa que la fórmula puede producir alturas negativas, corrígelo para que la altura mínima sea 0. La Figura 4.11 muestra un ejemplo de resultado del guión para una velocidad inicial de 30 m/s y un tiempo de 10 segundos.

4. Suponiendo que un proyectil se lanza con un ángulo horizontal α y una velocidad inicial v_i desde una altura ai, entonces su desplazamiento vertical y horizontal—ignorando la resistencia del aire—viene dado por las fórmulas:

$$dv = ai + v_i seno(\alpha)t - gt^2/2 \tag{4.1}$$

$$dh = v_i cos(\alpha)t \tag{4.2}$$

Escribe un guión que solicite la altura y velocidad inicial y el ángulo de lanzamiento y genere un gráfico en que se visualice al trayectoria del proyectil. La Figura 4.12 muestra un ejemplo de gráfico para una altura inicial de 20 metros, una velocidad inicial de 100 m/s y un ángulo de 30 grados.

5. Modifica el guión anterior de forma que se introduzcan 3 ángulos y se dibujen las 3 trayectorias correspondientes de forma solapada. La Figura 4.13 muestra una salida

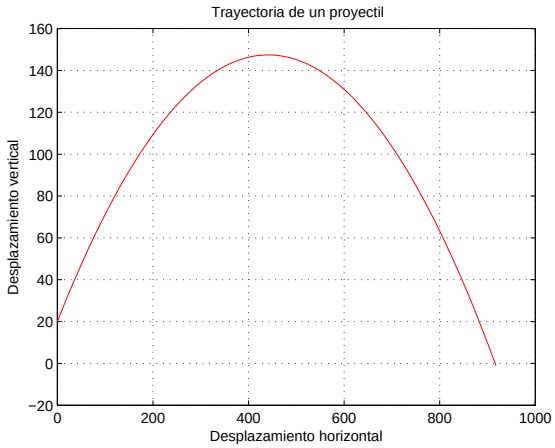

Figura 4.12: Trayectoria de un proyectil ($ai = 20m, vi = 100m/s, \alpha = 30^0$).

del guión para una altura inicial de 20 metros, una velocidad inicial de 100 m/s y ángulos de 30, 45 y 60 grados.

6. Escribe un guión que genere n puntos aleatorios uniformemente distribuidos en el cuadrado formado por los vértices (-1,-1) y (1,1). Dibuja los puntos como asteriscos, rojos si caen dentro del círculo unidad—radio 1 y centro (0,0)—y azules si caen fuera. En la Figura 4.14 se muestra un ejemplo de ejecución del guión en que se han generado 3000 puntos aleatorios.

7. La disposición de las pepitas de un girasol sigue un modelo matemático. La n-ésima semilla tiene coordenadas polares $r = \sqrt{n}$ y $\alpha = 137,51\pi n/180$. Escribe un guión que solicite el número de pepitas y las dibuje como círculos—la Figura 4.15 contiene 1000 pepitas. Modifica el guión para que las pepitas se vayan añadiendo una a una—debes usar la función pause.

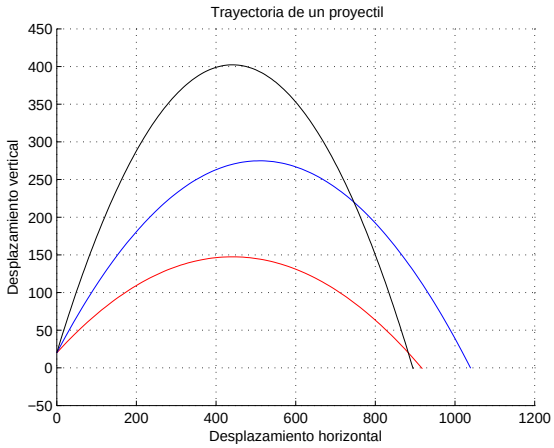

Figura 4.13: Trayectoria de un proyectil ($ai = 20m, vi = 100m/s, \alpha = [30^0, 45^0, 60^0]$).

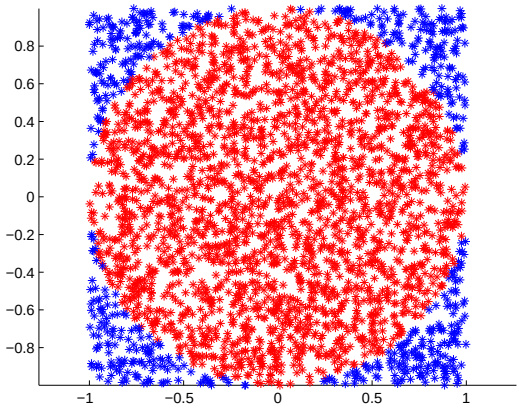

Figura 4.14: Ejemplo de ejecución del ejercicio del círculo con 3000 puntos.

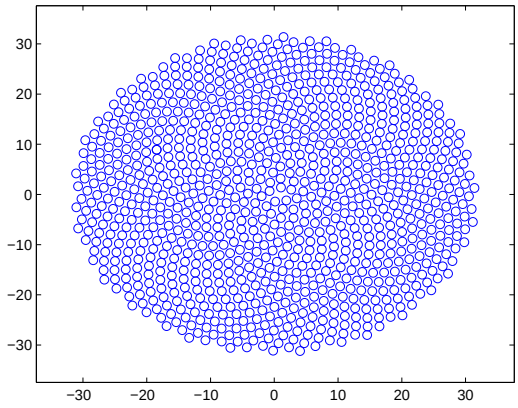

Figura 4.15: Hoja de girasol con 1000 pepitas

Tema 5
Funciones

Una función permite escribir un fragmento de código parametrizado. De esta forma, es posible escribir un bloque de código y ejecutarlo para distintos datos. Una función puede considerarse un subprograma que resuelve una subtarea. La segunda gran ventaja de las funciones es que permiten estructurar u organizar el código de un programa. Cuando hay que resolver un problema complejo, en lugar de intentar solucionarlo mediante un programa muy extenso es mejor descomponerlo en subproblemas. Los subproblemas deben tener una complejidad moderada, de forma que sean resueltos por subprogramas—como las funciones—sencillos. De esta forma, en lugar de utilizar un programa muy grande para resolver un problema complejo se emplean distintos subprogramas que resuelven tareas sencillas y que se combinan para producir una solución final más simple. Hasta ahora hemos utilizado muchas funciones internas de MATLAB, como max o sum. En este tema aprenderemos a escribir nuestras propias funciones.

5.1. Formato de una función

En MATLAB una función se define utilizando el siguiente formato:

```
function [vr1 vr2 ... vrm] = nombre (param1, param2, ..., paramn)
  %documentacion
  bloque de codigo
end
```

Describamos sus elementos:

- function es una palabra reservada con la que hay que iniciar la definición. La primera línea de la definición de una función se llama *cabecera de la función* y contiene la interfaz de uso de la función. Si una función está bien documentada, basta con conocer su cabecera y su documentación para poder utilizarla sin tener que leer su código. En

este sentido una función proporciona un mecanismo de *encapsulación* de código que nos permite abstraernos de los detalles de implementación.

- [vr1 vr2 ... vrm] son las *variables de retorno* de la función—también llamados *parámetros de salida*. Se utiliza una sintaxis similar a la de creación de un vector, pero representan los valores que devuelve la función. A veces una función no devuelve ningún valor; en ese caso se puede utilizar cualquiera de los dos formatos siguientes:

```
function [] = nombre (param1, param2, ..., paramn)
function nombre (param1, param2, ..., paramn)
```

- nombre es el nombre de la función y debe verificar las mismas reglas sintácticas que el nombre de una variable—véase la Sección 1.1.3.

- (param1, param2, ..., paramn) es una lista de nombres de variables encerrada entre paréntesis y separada por comas. Representa los *parámetros formales* de la función, que en MATLAB se corresponde con los *parámetros de entrada* de la función. Aunque no sea habitual, una función puede no tener parámetros, en cuyo caso se puede definir de estas dos formas:

```
function [vr1 vr2 ... vrm] = nombre()
function [vr1 vr2 ... vrm] = nombre
```

- Tras la cabecera de la función se puede escribir—no es obligatorio—varias líneas de comentarios que empiezan con el carácter especial %. Estas líneas aparecen en la ventana de órdenes si se teclea la orden:

```
>> help nombre
```

  donde nombre es el nombre de la función. El objeto de estos comentarios es documentar el comportamiento de la función. Serán útiles si a un usuario le basta con conocer la cabecera y la documentación de una función para ser capaz de utilizarla.

- Por último, se tiene un bloque de código que contiene la implementación de la función. El bloque de código termina con la palabra reservada end, aunque si la función aparece en un archivo que contiene únicamente a esa función el end no es necesario—en cualquier caso recomendamos siempre su uso.

En resumen, una definición de función consta de tres partes: cabecera, documentación e implementación. La cabecera indica la interfaz de uso y, si viene acompañada de una buena documentación de esa interfaz, nos abstrae de los detalles de su implementación. Como se ha indicado anteriormente, las funciones son un mecanismo de encapsulación de código.

```
function [inf sup] = rango (m)
    %rango Devuelve el rango de los elementos de una matriz
    %function [inf sup] = rango (m)
    %Parámetros de entrada:
    %   - m: una matriz numérica
    %Valores de retorno:
    %   - inf: límite inferior de los elementos de m
    %   - sup: límite superior de los elementos de m
    inf = min(min(m));
    sup = max(max(m));
end
```

Figura 5.1: Función que calcula el rango de los elementos de una matriz.

La Figura 5.1 muestra un ejemplo de función. Se trata de una función que calcula el rango de los elementos de una matriz. La función toma como parámetro de entrada una matriz y devuelve dos valores que se corresponden con los límites inferiores y superiores de los elementos de la matriz. Para devolver un valor el código de una función debe asignar el valor deseado a la variable de retorno correspondiente. El último valor asignado a una variable de retorno será el valor que devuelva dicha variable.

5.2. Uso de funciones

Para poder utilizar una función en MATLAB, ésta debe estar almacenada en un archivo M, es decir, un archivo de texto con extensión .m. Lo normal es ponerle al archivo el mismo nombre que la función, pero con la extensión .m. Por ejemplo, para la función de la Figura 5.1 el archivo se llamará *rango.m*. Ten en cuenta que si usas nombres distintos para nombrar a la función y al archivo—lo cual desaconsejamos—es el nombre del archivo y no el de la función el que hay que usar para invocar a la función. Para que se pueda invocar a la función el archivo debe estar almacenado en la carpeta o directorio de trabajo.

Una vez guardado el archivo en el directorio actual o de trabajo, se puede consultar la ayuda sobre la función:

```
>> help rango
  rango Devuelve el rango de los elementos de una matriz
  function [inf sup] = rango (m)
  Parámetros de entrada:
    - m: una matriz numérica
  Valores de retorno:
    - inf: límite inferior de los elementos de m
    - sup: límite superior de los elementos de m
```

5.3. Invocación de funciones

Las funciones definidas por el usuario se invocan como cualquier función de MATLAB, ya sea desde la ventana de órdenes, desde un guión o desde una función. Por ejemplo:

```
>> [i,s] = rango([5 1:3]) %obtenemos los dos limites
i =
     1
s =
     5
>> i = rango([5 1:3]) %obtenemos solo el limite inferior
i =
     1
>> [~,s] = rango([5 1:3]) %obtenemos solo el limite superior
s =
     5
```

Cuando una función devuelve más de un valor, hay que especificar una lista de variables encerradas entre corchetes para recibir los valores de retorno. Si sólo interesan los primeros valores, entonces podemos especificar sólamente los que nos interesen. Si algún valor de los primeros no nos hace falta, se puede utilizar el símbolo ~ para indicar que no estamos interesados en ese valor de retorno.

Al invocar a una función hay que especificar una expresión del tipo adecuado para cada parámetro de entrada de la función. A los parámetros con los que se invoca una función se les llama *parámetros reales*.

5.4. La sentencia return

Normalmente las funciones terminan su ejecución cuando llegan a su última instrucción. La sentencia return permite terminar la ejecución de una función. Cuando en una función existen varias condiciones que, si se verifican, implican el término de los cálculos, el uso de return suele producir funciones más sencillas. Por ejemplo, supongamos que tenemos que escribir una función que calcule si un entero positivo es primo. La función de la Figura 5.2 realiza este cómputo. Inicia la variable esPrimo a true si el número es mayor que uno—el uno no es primo. Después utiliza un ciclo while para comprobar si el número tiene algún divisor. Del ciclo se sale cuando se ha encontrado algún divisor—esPrimo valdrá false—o cuando se han agotado las pruebas de todos los posibles divisores. La función de la Figura 5.3 contiene un código alternativo que utiliza la sentencia return. Si el número es el uno o si se encuentra un divisor, se le asigna el valor false a la variable esPrimo y se termina la ejecución de la función.

```
function esPrimo = primo(n)
    esPrimo = n > 1;
    x = 2;
    while x ≤ floor(sqrt(n)) && esPrimo
        if rem(n,x) == 0
            esPrimo = false;
        end
        x = x + 1;
    end
end
```

Figura 5.2: Función que calcula si un entero positivo es primo

```
function esPrimo = primo2(n)
    if n == 1
        esPrimo = false;
        return
    end
    esPrimo = true;
    for x = 2:floor(sqrt(n))
        if rem(n,x) == 0
            esPrimo = false;
            return
        end
    end
end
```

Figura 5.3: Función que calcula si un entero positivo es primo usando return

5.5. Número variable de parámetros

Al igual que otros lenguajes de programación, MATLAB proporciona facilidades para la definición de funciones con un número variable de parámetros y/o valores de retorno. En MATLAB, dos de estas facilidades consisten en las siguientes funciones:

- nargin. Cuando se invoca sin parámetros dentro del cuerpo de una función, devuelve con cuántos parámetros de entrada se ha invocado a la función.

- nargout. Cuando se invoca sin parámetros dentro del cuerpo de una función, devuelve cuántas variables se han utilizado al llamar a la función para recibir los valores de retorno.

Vamos a ilustrar el uso de nargout modificando la función rango de forma que si se invoca especificando dos variables para las variables de retorno devuelva el rango en dos variables y si se especifica una sola variable devuelva el rango en un vector. Este comportamiento lo tienen algunas funciones como size:

```
>> m = zeros (3,4);
>> d = size(m)          %devuelve resultado en un vector
d =
     3     4
>> [nf,nc]= size(m)     %devuelve resultado en dos variables
nf =
     3
nc =
     4
```

La Figura 5.4 contiene la nueva versión de la función rango. En las variables inf y sup se almacena el mínimo y máximo, respectivamente, de los elementos de la matriz. Después se invoca a nargout para saber si la función fue invocada con un sólo parámetro, en cuyo caso se almacena el rango como un vector de dos elementos en la variable inf. A continuación se muestran ejemplos de invocación de rango2 con una y dos variables de retorno.

```
>> r = rango2([6 1 5])                  %una variable de retorno
r =
     1     6
>> [minimo maximo] = rango2([6 1 5])    %dos variables de retorno
minimo =
     1
maximo =
     6
```

```
function [inf sup] = rango2 (m)
    inf = min(min(m));
    sup = max(max(m));
    if nargout == 1
        inf = [inf sup];
    end
end
```

Figura 5.4: Función que calcula el rango de los elementos de una matriz. Versión 2

```
function v = miRand (n, inf, sup)
    v = rand(1,n);
    if nargin == 3
        v = v.*(sup-inf) + inf;
    end
end
```

Figura 5.5: Función que genera números pseudoaleatorios uniformemente distribuidos

Observa que la función rango2 no aparece documentada. En general, en estos apuntes no documentaremos las funciones con el objeto de obtener un código más corto. Existe otra razón para no documentar, en el texto describimos lo que hace la función.

La Figura 5.5 contiene una función que ejemplifica el uso de nargin. Esta función genera un vector con n números pseudoaleatorios distribuidos uniformemente; si la función se invoca con un sólo parámetro los números estarán en el intervalo $(0, 1)$, si se utilizan tres parámetros en el intervalo (inf, sup). La función miRand puede invocarse, pues, con uno o tres parámetros:

```
>> miRand(6)
ans =
    0.0811    0.9294    0.7757    0.4868    0.4359    0.4468
>> miRand(6, 2, 5)
ans =
    2.9190    3.5255    3.5323    4.4529    4.3845    3.9330
```

En el Tema 7, Sección 7.5.2, se describe otra facilidad de MATLAB para escribir funciones con un número indeterminado de parámetros de entrada o de salida.

```
function res = miSum(m, tipo)
    res = [];
    if ischar(tipo)
        switch tipo
            case 'filas'
                res = sum(m,2);
            case 'columnas'
                res = sum(m,1);
            case 'total'
                res = sum(sum(m));
        end
    elseif isnumeric(tipo)
        res = sum(m, tipo);
    end
end
```

Figura 5.6: Función con un argumento que puede tomar valores de distintos tipos

5.6. Argumentos que toman valores de distintos tipos

MATLAB tiene muchas funciones internas que aceptan parámetros que pueden ser invocados con valores de distintos tipos. En esta sección vamos a estudiar cómo implementar funciones con estas características. Supongamos que queremos implementar una versión de la función sum para sumar matrices. El primer parámetro de nuestra función será la matriz y el segundo parámetro el tipo de suma. Para el tipo de suma se van a aceptar las cadenas de caracteres *filas*, *columnas* y *total* indicando si se quiere la suma por filas, por columnas o total de los elementos de la matriz respectivamente. También aceptamos como segundo parámetro los números 1 y 2, al estilo de sum, que expresan la suma por columnas y por filas respectivamente. La Figura 5.6 contiene el código de la función, observa el tratamiento dado al segundo parámetro. Cuando se aceptan parámetros que pueden tomar valores de distintos tipos, hay que utilizar funciones como ischar, isnumeric o islogical para distinguir el tipo del parámetro real con que se invocó a la función y, una vez detectado el tipo, actuar en consecuencia. Otra alternativa es usar la función class para obtener el tipo del parámetro; como ejercicio puedes implementar una función con un comportamiento idéntico al de la Figura 5.6 pero que utilice class para determinar el tipo de su segundo parámetro.

A continuación mostramos algunas invocaciones a la función para comprobar su funcionamiento:

```
>> m = [2 4 6; 1 3 5];
>> miSum(m, 'filas')
ans =
```

```
     12
      9
>> miSum(m, 'total')
ans =
     21
>> miSum(m,2)
ans =
     12
      9
>> miSum(m,1)
ans =
      3       7      11
```

5.7. Ámbito y tiempo de vida de una variable. Variables locales

Se denomina *ámbito de una variable* a la zona de código donde se puede utilizar la variable. Otro concepto interesante es el *tiempo de vida de una variable*, que significa, a grosso modo, el tiempo que una variable permanece almacenada en la memoria del ordenador.

Las variables que se asignan en la ventana de órdenes o en un guión forman parte del denominado *espacio de trabajo base—base workspace*. El tiempo de vida de estas variables empieza cuando se les asigna un valor por primera vez y termina al acabar la sesión de trabajo con MATLAB, salvo que se utilice clear para borrarlas. En cuanto a su ámbito, estas variables pueden utilizarse en la ventana de órdenes y en cualquier guión.

Las variables utilizadas en una función se llaman *variables locales*. Las variables locales de una función vienen constituidas por sus parámetros de entrada y salida, así como cualquier otra variable que se utilice en la función. El tiempo de vida de una variable local es el tiempo que dura la ejecución de la función. El ámbito de una variable local es la función de la que forma parte.

El nombre de una variable local puede coincidir con el nombre de otras variables del espacio de trabajo base o de otras variables locales. Cuando se ejecuta un guión o una instrucción en la ventana de órdenes se utiliza la variable del espacio de trabajo base y cuando se ejecuta una función se utiliza la variable local.

Veamos un ejemplo. En la Figura 5.7 se muestra un guión y en la Figura 5.8 una función que devuelve un vector con los elementos de la matriz que recibe como parámetro que son mayores que su media. Tanto el guión como la función contienen una variable llamada media. Si ejecutamos el guión, en su primera instrucción se crea la variable del espacio de trabajo base media y se le asigna el valor 3. A continuación se invoca a la función mayoresMedia que utiliza la variable local media para almacenar la media de los elementos de la matriz que recibe como parámetro. En esa ejecución de la función a la variable local se le asigna el valor 5.5. Tras ejecutarse la función mayoresMedia la variable local media deja de existir y se

```
media = mean(5:-1:1);
disp(mayoresMedia(1:10))
fprintf('El valor de media es %.2f\n', media)
```

Figura 5.7: Guión que llama a la función mayoresMedia.

```
function res = mayoresMedia(m)
  media = mean(mean(m));
  res = m(find(m > media));
end
```

Figura 5.8: Función que calcula los elementos de una matriz mayores que su media.

reanuda la ejecución del guión que vuelve a utilizar a la variable del espacio de trabajo base media, que sigue almacenando el valor 3.

5.8. Funciones con memoria: variables persistentes

Como se ha indicado en la sección anterior, el tiempo de vida de una variable local coincide con el tiempo de ejecución de la función en que aparece. Esto hace que las funciones "no tengan memoria", en el sentido de que no pueden recordar datos de una invocación a otra. Sin embargo, es útil que algunas funciones puedan recordar valores de una invocación a otra. Como ejemplo, supongamos que queremos escribir una función que proporciona identificadores enteros únicos, ¿cómo va a hacerlo si no puede recordar qué identificadores ha asignado?

En MATLAB una *variable persistente* es una variable local que recuerda valores entre llamadas a la función. Es decir, como cualquier variable local su ámbito es el código de la función, pero su tiempo de vida, al igual que las variables del espacio de trabajo base, se prolonga hasta que termina la sesión con MATLAB.

La Figura 5.9 contiene una "función con memoria", es decir, que utiliza una variable persistente. La línea

```
persistent identificador
```

declara la variable identificador como persistente. Una declaración es una instrucción en la que se especifican características de una variable u otro objeto. En MATLAB, a diferencia de la mayor parte de lenguajes, no se declaran las variables, se empiezan a usar asignándoles

```
function id = obtenerIdentificador
  persistent identificador
  if isempty(identificador)
      identificador = 0;
  end
  identificador = identificador + 1;
  id = identificador;
end
```

Figura 5.9: Función con memoria que devuelve un identificador entero único.

un valor. Sin embargo, las variables persistentes son especiales y hay que indicarlo con una declaración antes de poder utilizarlas.

La función obtenerIdentificador va devolviendo identificadores sucesivos: 1, 2, 3, ... La variable persistente identificador almacena el último identificador asignado. Queremos iniciarla a cero la primera vez que se invoque a la función, pero no más veces, pues si no siempre devolvería el identificador 1. Para lograr esto se utiliza la función isempty, que devuelve un valor lógico que indica si un array no tiene elementos. La primera vez que se invoque a obtenerIdentificador la condición del if se verifica e identificador se inicia a cero. En las siguientes llamadas a obtenerIdentificador la condición no se verifica e identificador sólo se incrementa en uno—no se inicia a cero.

Vamos a invocar a obtenerIdentificador varias veces desde la ventana de órdenes para comprobar su funcionamiento:

```
>> a = obtenerIdentificador
a =
     1
>> b = obtenerIdentificador
b =
     2
```

clear funcion borra las variables persistentes de la función funcion y clear functions borra todas las variables persistentes.

```
>> clear obtenerIdentificador
>> c = obtenerIdentificador
c =
     1
```

La función mlock utilizada dentro de una función hace que sus variables persistentes no se puedan borrar, mientras que munlock posibilita el borrado.

5.9. Funciones auxiliares

Un archivo M puede contener varias funciones. Sin embargo, sólo la primera de ellas puede ser invocada desde fuera del archivo: desde un guión, desde la ventana de órdenes o desde otra función ubicada en otro archivo. ¿Cual es la utilidad entonces de escribir varias funciones en un archivo M? La respuesta es sencilla, a veces una función es lo suficientemente compleja como para dividir su código en varias funciones. A la primera función ubicada en un archivo M se le llama *función principal* y al resto *funciones auxiliares*. Las funciones ubicadas en un mismo archivo pueden invocarse entre sí, pero desde fuera del archivo sólo se puede llamar a la principal, que constituye la interfaz de uso del archivo. Veamos un ejemplo. En la Figura 5.10 se muestra el contenido de un archivo con dos funciones. La función principal devuelve el elemento de una matriz que ocurre con más frecuencia—si hay más de uno devuelve un vector con los elementos que ocurren más veces. Se ha optado por emplear una función auxiliar para calcular cuántas veces aparece un elemento en una matriz. La función principal puede invocarse desde la ventana de órdenes, pero la auxiliar no:

```
>> m = [1 2 3 4; 5 6 3 4];
>> maxOcurrencias(m)
ans =
     3     4
>> ocurrencias(m,2)
 Undefined function or method 'ocurrencias' for input arguments of type 'double'.
```

5.10. Funciones anónimas

Una *función anónima* es una función con una única línea de código que se define de una forma especial que hace que no tenga nombre. La ventaja de las funciones anónimas es que no tienen que almacenarse en un archivo M. Cuando se desarrolla una aplicación grande el uso de funciones anónimas puede reducir drásticamente el número de archivos M de la aplicación. La sintaxis para crear una función anónima es la siguiente:

```
puntero = @(argumentos) expresion
```

puntero es el nombre de una variable que se empleará para utilizar la función. La variable almacenará un *puntero o dirección a función*—nota: la documentación de MATLAB habla de *function handle*, pero he preferido utilizar el término puntero a función. @ es un operador que se utiliza para obtener la dirección de una función. Los argumentos de la función se especifican entre paréntesis y después hay que indicar una expresión en lo que queda de línea. Una función anónima devuelve el resultado de ejecutar su expresión asociada. Se puede definir una función anónima en la ventana de órdenes o en el código de un guión

```
function sol = maxOcurrencias(m)
  sol = m(1,1);
  maximo = ocurrencias(m, sol);
  for ind = 2:numel(m)                % recorrido lineal de la matriz
     if any(m(1:ind-1) == m(ind))     % se ha procesado el elemento antes?
         continue
     end
     ocu = ocurrencias(m, m(ind));
     if ocu > maximo
         sol = m(ind);
         maximo = ocu;
     elseif ocu == maximo
         sol(end+1) = m(ind);
     end
  end
end

function ocu = ocurrencias(m,el)
    ocu = length(find(m == el));
end
```

Figura 5.10: Archivo M con una función principal y otra auxiliar

o una función. Veamos un ejemplo. Vamos a definir una función anónima que toma como parámetro un vector e indica si el vector es un palíndromo, es decir, si el vector coincide con su vector inverso.

```
>> pal = @(v) all(v == v(end:-1:1));
>> pal([1 2 3])
ans =
     0
>> pal([1 2 3 2 1])
ans =
     1
```

En la primera línea se ha definido la función anónima y se ha asignado su dirección a la variable pal—ten en cuenta que si no se asigna a una variable no se podría invocar. Para invocar a la función se usa la variable que almacena el puntero con una sintaxis análoga a la invocación de funciones con nombre. El código previo ilustra dos invocaciones. Se puede utilizar la función class para conocer el tipo de un puntero a función:

```
>> class(pal)
ans =
function_handle
```

Una variable que almacena un puntero a una función anónima puede almacenarse en un archivo MAT y después cargarse cuando sea preciso para utilizar su función asociada:

```
>> pal = @(v) all(v == v(end:-1:1));
>> save('funcionAnonima.mat', 'pal')
>> clear
>> load('funcionAnonima.mat')
>> pal('radar')
ans =
     1
```

Se podría añadir más funciones anónimas a este archivo MAT. Aunque una de las ventajas de las funciones anónimas es que no tienen que almacenarse en un archivo M, resulta útil almacenar grupos de funciones anónimas en un—único—archivo MAT. De esta forma, cuando sean necesarias se pueden cargar con una sóla instrucción load.

Como ejercicio modifica el ejemplo de la Figura 5.10. Sustituye la función auxiliar ocurrencias por una función anónima definida al principio de la función maxOcurrencias.

Aunque una función anónima no tenga parámetros de entrada hay que utilizar paréntesis tanto en su definición como al invocarla:

```
>> f = @() disp('hola');
```

```
>> f()
hola
```

5.11. Uso de punteros a funciones

No sólo podemos obtener punteros a funciones anónimas, también es posible obtener punteros a funciones internas o a funciones definidas por el usuario almacenadas en un archivo M. Para ello hay que utilizar el operador @:

```
>> pMin = @min
pMin =
    @min
>> pMin([4 8 2 3])
ans =
     2
```

En el código anterior se utiliza el operador @ para obtener un puntero a la función interna min, que se asigna a la variable pMin. A continuación se muestra que es posible invocar a la función a través del puntero. Sin embargo, este es un uso extraño de un puntero a función. Lo interesante es que existen funciones que toman como parámetros punteros a funciones para poder invocar a las funciones recibidas como parámetro. La documentación de MATLAB llama a este tipo de funciones *function functions*. Veamos un par de ejemplos. La función fplot toma como parámetro un puntero a una función que evalúa un vector que recibe como parámetro y un rango para la coordenada x y produce el gráfico correspondiente a llamar a la función en el rango especificado. El gráfico es continuo. Por ejemplo, el guión:

```
subplot(2,1,1)
fplot(@sin, [0, 4*pi])
subplot(2,1,2)
fplot(@cos, [0, 4*pi])
```

utiliza la función fplot para obtener un gráfico de las funciones seno y coseno en el intervalo $[0, 4\pi]$—véase la Figura 5.11. Una ventaja de fplot es que representa más o menos puntos en un subintervalo en función de cómo de rápido cambia la función en el subintervalo.

Una función muy interesante es arrayfun, que aplica la función recibida como parámetro a los elementos de un *array* que recibe como segundo parámetro. Por ejemplo, la función primo de la Figura 5.2 sólo puede aplicarse a un dato, pero se puede utilizar arrayfun para aplicársela a los elementos de una matriz:

```
>> primo(7)
ans =
```

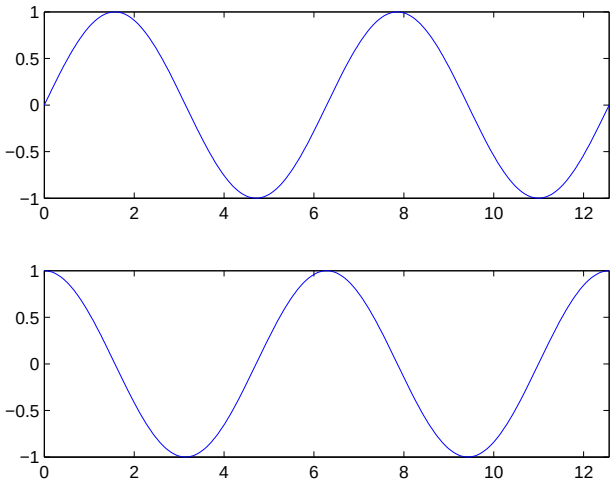

Figura 5.11: Resultado de invocar a fplot.

```
     1
>> arrayfun(@primo, [1:5;6:10])
ans =
     0     1     1     0     1
     0     1     0     0     0
>> find(arrayfun(@primo, 1:20))
ans =
     2     3     5     7    11    13    17    19
```

En el ejemplo, se ha calculado una matriz de valores lógicos. La primera fila indica si los números del 1 al 5 son primos y la segunda si los números del 6 al 10 son primos. En el segundo uso de arrayfun se obtiene un listado de los primos comprendidos entre el 1 y el 20. Otro posible uso sería el siguiente:

```
>> primo3 = @(a) arrayfun(@primo, a);
>> primo3([1:5; 6:10])
ans =
     0     1     1     0     1
     0     1     0     0     0
>> primo3(15)
ans =
     0
```

```
function miPlot(func, inter, nvalores)
    if nargin() == 2
        nvalores = 20;
    end
    x = linspace(inter(1), inter(2), nvalores);
    plot(x, func(x), 'ko')
end
```

Figura 5.12: Función con un parámetro de tipo puntero a función

Ahora hemos escrito una función—primo3—que utiliza arrayfun para generalizar la función primo, de forma que pueda ser invocada con un array. Por cierto, MATLAB tiene su función interna para comprobar si un valor es primo, se llama isprime.

Otras funciones que utilizan punteros a funciones son:

- fzero: busca un cero de una función cercano a una estimación inicial.
- integral: calcula la integral aproximada de una función en un intervalo.
- fminbnd: calcula el mínimo de una función en un dominio.

Otro ejemplo de uso de los punteros a funciones se encuentra en la programación de interfaces gráficas de usuario—*GUIs, Graphics User Interfaces*—, en este caso los punteros a funciones se utilizan para indicar las funciones de retrollamada—*callback functions*.

Se puede escribir funciones definidas por el usuario que tomen parámetros de tipo puntero a función. Por ejemplo, la función de la Figura 5.12 tiene un comportamiento similar a fplot, pero representa puntos y no una línea continua. El número de puntos por defecto es 20. El siguiente guión invoca a miPlot—véase el resultado en la Figura 5.13:

```
subplot(2,1,1)
miPlot(@sin, [0 4*pi])
subplot(2,1,2)
miPlot(@sin, [0, 4*pi], 40)
```

5.12. Funciones que trabajan con matrices

La filosofía de las funciones internas de MATLAB es que, si se puede, admitan como parámetro un *array* en lugar de un escalar. Así, la función log2 permite calcular el logaritmo en base 2 de un escalar, pero también de los elementos de una matriz:

```
>> log2(8)              % array 1x1
ans =
```

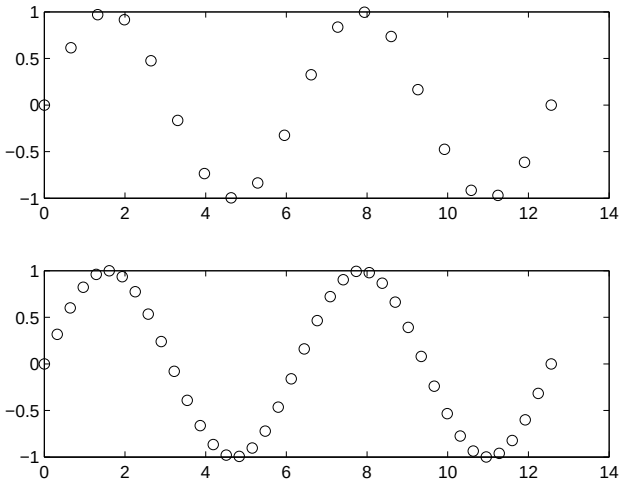

Figura 5.13: Resultado de invocar a miPlot.

```
     3
>> log2([2  4; 8  16]) % array 2x2
ans =
     1     2
     3     4
```

Intenta que tus funciones sigan esta filosofía. A veces, la diferencia radica en utilizar el operador adecuado. Por ejemplo:

```
>> areaTriangulo = @(base,altura) base*altura/2;
>> areaTriangulo(10,5)
ans =
    25
>> areaTriangulo2 = @(base,altura) base.*altura/2;
>> areaTriangulo2([10,7],[5,6])
ans =
    25    21
```

La primera función sólo permite calcular el área de un triángulo, mientras que la segunda permite calcular el área de varios triángulos. La diferencia radica en el uso del operador `.*` en lugar del operador `*`.

Veamos otro ejemplo. Supongamos que queremos escribir una función signo que devuelve el valor 1, 0 ó -1 en función de que su parámetro sea positivo, cero o negativo respectivamente—esta función ya existe en MATLAB, se llama sign. La función de la Fi-

```
function res = signo(valor)
    if valor > 0
        res = 1;
    elseif valor < 0
        res = -1;
    else
        res = 0;
    end
end
```

Figura 5.14: Función que calcula el signo de un escalar

```
function res = signo2(valores)
    res = zeros(size(valores));
    res(valores > 0) = 1;
    res(valores < 0) = -1;
end
```

Figura 5.15: Función que calcula el signo de los elementos de un *array*

gura 5.14 sólo puede trabajar con un escalar, mientras que la función de la Figura 5.15 puede trabajar con un *array* de números.

5.13. Dualidad orden/función

Cuando una función toma argumentos de tipo cadena de caracteres es posible invocarla en modo orden o en modo función. Al invocarla en modo orden no hay que poner comillas y los parámetros se separan mediante espacios en blanco. Por ejemplo:

```
>> disp('hola')
hola
>> disp hola
hola
```

Esto es aplicable a muchas funciones como save, load o title .

5.14. Paso de parámetros de entrada

En MATLAB el paso de parámetros de entrada tiene semántica de *paso por valor o copia*. Esto significa que los parámetros reales y formales son independiente, en el sentido de que una

```
function res = noModifica(m)
    res = m(1,1) + m(1,end) + m(end,1) + m(end,end);
end
```

Figura 5.16: Función que no modifica su parámetro formal

función puede modificar sus parámetros formales sin que las modificaciones afecten a los valores almacenados en los parámetros reales.

Normalmente, el paso de parámetros por copia se implementa haciendo que los parámetros formales se inicien con una copia del valor asociado a los parámetros reales—de ahí el nombre de paso por copia. MATLAB utiliza la técnica *copy on write—copia al escribir*—para implementar el paso de parámetros de entrada. Esta técnica se emplea para usar más eficientemente recursos compartidos. En el caso del paso de los parámetros de entrada en MATLAB esto significa que las funciones trabajan con la zona de memoria asociada a los datos de los parámetros reales. Cuando una función modifica un parámetro formal, entonces se copian los datos, para que la función modifique una copia de los datos y los datos asociados a los parámetros formales y reales permanezcan independientes. Sin embargo, si la función no modifica los parámetros formales no se realiza ninguna copia, con el consecuente ahorro en tiempo de ejecución.

A continuación vamos a comprobar los beneficios de la técnica *copy on write*. Las funciones de las Figuras 5.16 y 5.17 devuelven la suma de los elementos situados en las cuatro esquinas de la matriz que reciben como parámetro. Son prácticamente iguales, pero la segunda función guarda temporalmente el resultado en la posición (1,1) de la matriz. Por lo tanto, la segunda función modifica su parámetro formal, lo que implica que se realiza una copia de la matriz. La primera función no modifica su parámetro formal, con lo que trabaja con la matriz original. El guión de la Figura 5.18 crea una matriz 1000x1000 e invoca 100 veces a cada una de las funciones para ver cómo afecta al tiempo de ejecución la copia de los datos. El resultado en mi ordenador es el siguiente:

```
Sin copia: 0.0029 segundos
Con copia: 0.3051 segundos
Sin copia 103.48 veces mas rapido
```

Como se esperaba, la copia afecta en gran medida al tiempo de ejecución, puesto que las funciones realizan un cálculo despreciable—tres sumas—comparado con la copia de una matriz de un millón de elementos.

```
function res = noModifica(m)
    m(1,1) = m(1,1) + m(1,end) + m(end,1) + m(end,end);
    res = m(1,1);
end
```

Figura 5.17: Función que modifica su parámetro formal

```
clc
m = rand(1000);
inicio1 = tic;
for x = 1:100
    a = noModifica(m);
end
tiempo1 = toc(inicio1);
fprintf('Sin copia: %.4f segundos\n', tiempo1)

inicio2 = tic;
for x = 1:100
    b = modifica(m);
end
tiempo2 = toc(inicio2);
fprintf('Con copia: %.4f segundos\n', tiempo2)
fprintf('Sin copia %.2f veces más rápido\n',tiempo2/tiempo1)
```

Figura 5.18: Guión que comprueba la efectividad de la técnica *copy on write*

5.15. Variables globales

La función de la Figura 5.19 calcula una ocurrencia horizontal de una palabra en una sopa de letras. Para ello se apoya en la función auxiliar buscaHorizontal, ésta a su vez utiliza la función strfind —que se explica en la Sección 6.4.2—para buscar una palabra en una fila de la sopa de letras. La función sopaDeLetras se apoya en el paso de parámetros para comunicarse con buscaHorizontal, le pasa como parámetros de entrada la sopa de letras y la palabra y recibe como parámetros la fila y columna donde empieza la palabra.

Una *variable global* es una variable a la que pueden acceder varias funciones y guiones sin usar paso de parámetros. Para poder acceder como variable global todas las funciones o guiones que la quieran usar tienen que haberla declarado mediante el calificativo global. Es decir, el ámbito de una variable global es todas aquellas funciones que la declaren y su tiempo de vida es el tiempo que dure la sesión de trabajo. En la Figura 5.20 se muestra un programa que hace uso de variables globales para implementar la funcionalidad del programa de la Figura 5.19.

La ventaja del uso de variables globales es la eficiencia, pues se evita el paso de parámetros. Sin embargo, una función que usa variables globales pierde parte de la esencia de las funciones, que es ser un código parametrizado que abstrae la realización de un cómputo. Una función que usa variables globales no es independiente del resto del código, depende de que se ejecute en un entorno en que se han declarado las variables globales que usa. Por lo tanto, se desaconseja el uso de variables globales. Su utilización sólo resulta justificable en programas muy específicos, pequeños y que requieran un alto rendimiento. Un programa cuyas funciones se comunican mediante variables globales resulta difícil de entender y mantener, aunque si el programa es pequeño la dificultad puede ser controlable.

5.16. Ejercicios

1. Un método para encontrar todos los números primos en un rango de 1 a N es la Criba de Eratóstenes. Considera la lista de números entre el 2 y N. Dos es el primer número primo, pero sus múltiplos (4, 6, 8, ...) no lo son, por lo que se tachan de la lista. El siguiente número después del 2 que no está tachado es el 3, el siguiente primo. Entonces tachamos de la lista todos los múltiplos de 3 (6, 9, 12, ...). El siguiente número que no está tachado es el 5, el siguiente primo, y entonces tachamos todos los múltiplos de 5 (10, 15, 20, ...). Repetimos este procedimiento hasta que lleguemos al primer elemento de la lista cuyo cuadrado sea mayor que N. Todos los números que no se han tachado en la lista son los primos entre 2 y N.

   Escribe una función que tome como parámetro un valor N y devuelva un vector de tamaño N de valores lógicos. El valor lógico en la posición p del vector indica si el número p es primo. Un ejemplo de ejecución es:

```
function sopaDeLetras

sopa = ['SPULPEC'
        'MRECOSOA'
        'OOVRNPB'
        'NANEROA'
        'PAJOTOL'
        'SOSAUPL'
        'ERGALLO'];
    palabra = input('Introduce una palabra: ','s');
    [f c] = buscaHorizontal(sopa, palabra);
    if f == 0
        fprintf('%s no aparece horizontalmente\n', palabra);
    else
        fprintf('%s empieza en (%d,%d)\n', palabra, f, c);
    end
end

function [f c] = buscaHorizontal(sl, palabra)
    for f = 1:size(sl,1)
        c = strfind(sl(f,:), palabra);
        if length(c) == 1
            return
        end
    end
    f = 0;
end
```

Figura 5.19: Función para buscar una palabra en las filas de una sopa de letras

```
function sopaDeLetras2
    global sopa
    global palabra
    global f
    global c
    sopa = ['SPULPEC'
            'MRECOSOA'
            'COVRNIPB'
            'NANEROA'
            'PAJOTOL'
            'SOSAUPL'
            'ERGALLO'];

    palabra = input('Introduce una palabra: ','s');
    buscaHorizontal
    if f == 0
        fprintf('%s no aparece horizontalmente\n', palabra);
    else
        fprintf('%s empieza en (%d,%d)\n', palabra, f, c);
    end
end

function buscaHorizontal
    global sopa
    global palabra
    global f
    global c
    for f = 1:size(sopa,1)
        c = strfind(sopa(f,:), palabra);
        if length(c) == 1
            return
        end
    end
    f = 0;
end
```

Figura 5.20: Función con variables globales

```
>> criba(10)
ans =
     0     1     1     0     1     0     1     0     0     0
>> find(criba(10))
ans =
     2     3     5     7
```

2. Escribe una función que devuelva si un número natural es primo, con la restricción de que el número debe pertenecer al rango $[1, 10^6]$. En lugar de calcular si el número es primo cada vez que se invoque a la función, los primos del 1 al millón deben precalcularse utilizando la criba de Eratóstenes. De esta forma, la función devuelve el valor precalculado. Para poder almacenar los valores precalculados la función debe "tener memoria". Cuando se invoque a la función por primera vez, ésta debe realizar la criba y en sucesivas invocaciones utilizar los valores precalculados.

3. Escribe una función que devuelva las coordenadas cartesianas equidistantes de n puntos de una circunferencia. Por defecto, la circunferencia tiene radio 1 y centro (0,0), pero también se puede especificar otro radio y/o centro. El primer punto y el último deben aparecer repetidos. Como ejemplo de invocaciones, el siguiente guión debe producir una figura similar a la Figura 5.21.

```
axis('equal')
hold('on')
grid('on')
[x y] = coordCir(50);
plot(x,y,'red')
[x y] = coordCir(50,2);
plot(x,y,'blue')
[x y] = coordCir(50,1,1,1);
plot(x,y,'green')
```

4. Escribe una función que utilice la fórmula $(f(x+h) - f(x))/h$ para estimar la primera derivada de $f(x) = 3x^2$ en un punto. Para ello utiliza valores decrecientes de h, por ejemplo, 10^{-1}, 10^{-2}, 10^{-3} ...Para de estimar cuando la diferencia absoluta entre las dos últimas estimaciones sea menor que 10^{-4}. Es fácil comprobar si la función es correcta, pues la primera derivada de un polinomio se puede calcular analíticamente, en contreto, para $f(x) = 3x^2$ se verifica que $f'(x) = 6x$.

5. Modifica la función del ejercicio anterior para que la función de la que se calcula la derivada sea un parámetro.

6. Escribe una función que ordene un vector en orden creciente utilizando el algoritmo de selección. El algoritmo es el siguiente:

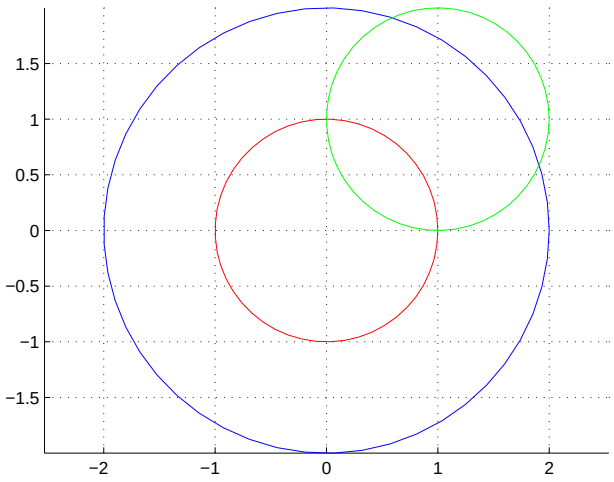

Figura 5.21: Figura que dibuja tres circunferencias con distintos parámetros.

```
Para las posiciones POS de la 1 a la penultima
    Calcula el minimo entre las posiciones [POS,ULTIMA]
    Intercambia el minimo con el elemento en la posicion POS
```

De esta forma en la primera iteración se selecciona el mínimo de los elementos del vector y se intercambia con el elemento en la posición 1. En la segunda iteración se selecciona el mínimo de los elementos del vector entre las posiciones $[2, final]$ y se intercambia con el elemento en la posición 2. En general, en la iteración i se selecciona el mínimo entre las posiciones $[i, final]$ y se intercambia con el valor en la posición i, tras ello se verifica que las posiciones $[1, i]$ del vector contienen los i elementos menores del vector ordenados. Puedes utilizar la función issorted para comprobar si el vector está ordenado.

Tema 6
Cadenas de caracteres

La información de tipo texto permite representar datos como el nombre de una persona o una dirección de correo. Estos datos son imprescindibles si se quiere modelar sistemas como un hospital—piénsese, por ejemplo, en los historiales clínicos de los pacientes—, y también se requieren con cierta frecuencia en entornos científicos y de ingeniería. En este tema estudiamos las facilidades de MATLAB para el trabajo con información de tipo texto. Se empieza estudiando la representación interna de las *cadenas de caracteres*, que es el tipo que emplea MATLAB para almacenar texto. Después se analizan operaciones comunes de trabajo con cadenas, como la búsqueda o el reemplazo de cadenas. También se describe cómo se puede guardar y recuperar información de tipo texto en un archivo.

6.1. Representación interna

Una cadena de caracteres o simplemente una cadena es una secuencia de caracteres y permite expresar información de tipo texto. Las dos formas típicas de crear una variable de tipo cadena son las siguientes:

```
>> apellido = 'Ochoa'
apellido =
Ochoa
>> nombre = input('Introduce el nombre: ', 's')
Introduce el nombre: Sonia
nombre =
Sonia
>> class(apellido)
ans =
char
```

En la primera forma se asigna a una variable una cadena, ésta debe aparecer entre comillas simples. En la segunda forma a la variable se le asigna el valor tecleado por el usuario

como resultado de ejecutar la función input. Observa que el tipo de una variable cadena de caracteres es char.

En MATLAB una cadena de caracteres se representa mediante un vector que almacena la secuencia de caracteres. Por ello, es posible aplicarle a una cadena muchas de las operaciones que se pueden realizar sobre vectores de números. Por ejemplo:

```
>> apellido(1:3) %subcadena formada por los 3 primeros caracteres
ans =
Och
>> nombre'          %traspuesta
ans =
S
o
n
i
a
```

Internamente, cada carácter se codifica como un número, siguiendo una codificación dada. La codificación empleada es la ASCII. El código ASCII permite representar las letras del alfabeto inglés, los dígitos decimales, signos de puntuación y algunos caracteres más, como la letra eñe. Algunos caracteres son visibles, mientras que ciertos caracteres de control no lo son. En MATLAB, una cadena de caracteres se implementa como un vector que almacena en cada una de sus posiciones el código ASCII de cada uno de los caracteres de la cadena.

Las funciones de conversión de tipos son funciones que permiten cambiar la representación de un dato. Dos de estas funciones son char, que convierte su argumento a carácter—es decir, a su código ASCII—y double, que convierte su argumento a representación numérica interna de punto flotante con doble precisión. Estas conversiones nos permiten descubrir el código asociado a los caracteres:

```
>> double('abc')     %permite ver el codigo ASCII de la a, b y c
ans =
    97    98    99
>> char([65 66 67]) %caracteres asociados a los codigos 65, 66 y 67
ans =
ABC
```

La función double permite comprobar que los códigos ASCII asocian números contiguos a caracteres contiguos del alfabeto inglés. La función char convierte una serie de números a la cadena de caracteres cuyos códigos ASCII coinciden con los números.

Es posible mezclar en una expresión cadenas con números, en cuyo caso se realiza una conversión implícita de los caracteres a su representación numérica en punto flotante.

```
>> 'abc' + 1
```

```
ans =
    98    99   100
>> char('a'+1)
ans =
b
```

6.2. Concatenación de cadenas

Se puede concatenar cadenas de caracteres utilizando la concatenación de vectores:

```
>> cad1 = 'El';
>> cad2 = [cad1 ' extranjero']
cad2 =
El extranjero
```

Otra posibilidad es usar la función strcat, ésta concatena sus parámetros, pero elimina los espacios en blanco finales de las cadenas. Por ejemplo:

```
>> cad1 = ' ab '; cad2 = ' cd ';
>> length(cad1)
ans =
     4
>> length(cad2)
ans =
     4
>> cad3 = [cad1 cad2]
cad3 =
 ab  cd
>> length(cad3)
ans =
     8
>> cad4 = strcat(cad1,cad2)
cad4 =
 ab cd
>> length(cad4)
ans =
     6
```

6.3. Comparación de cadenas

Se puede comparar dos cadenas de la misma longitud para ver si son iguales utilizando las facilidades del trabajo con vectores:

```
>> 'abcd' == 'abcd'
ans =
     1     1     1     1
>> all('abcd' == 'abcd')
ans =
     1
>> 'aa' == 'bb'
ans =
     0     0
>> 'abc' == 'xx'
```
Error using ==¿eqMatrix dimensions must agree.

Como vemos en la última comparación, si las cadenas tienen distinta longitud se obtiene un error. Afortunadamente las funciones strcmp, strcmpi, strncmp y strncmpi permiten comparar cadenas de distinta longitud. De estas cuatro funciones, las acabadas en i comparan cadenas sin tener en cuenta las mayúsculas. Las funciones strn comparan sólo los n primeros caracteres. Ejemplos:

```
>> strcmp('abc','abc')
ans =
     1
>> strcmp('abc','abcd')        %compara cadenas de distinta longitud
ans =
     0
>> strncmp('abc','abcd', 3)    %compara los 3 primeros caracteres
ans =
     1
>> strcmp('abc', 'Abc')
ans =
     0
>> strcmpi('abc', 'Abc')       %no tiene en cuenta las mayusculas
ans =
     1
```

Se puede comparar una cadena con un carácter:

```
>> libro = 'El hereje';
>> libro == 'e'
ans =
     0     0     0     0     1     0     1     0     1
>> sum(libro == 'e')  %numero de ocurrencias de e
ans =
     3
>> libro < 'a'
ans =
```

```
     1     0     1     0     0     0     0     0     0
```

Desgraciadamente, MATLAB no tiene una función para comparar dos cadenas en orden lexicográfico.

6.4. Más funciones que trabajan con cadenas

En esta sección vamos a describir varias funciones internas de MATLAB que operan sobre cadenas de caracteres.

6.4.1. Cambiando de mayúsculas a minúsculas

La función upper convierte una cadena a mayúsculas, mientras que lower convierte una cadena a minúsculas:

```
>> upper('Tiene 10 dados')
ans =
TIENE 10 DADOS
>> lower('Tiene 10 dados')
ans =
tiene 10 dados
```

6.4.2. Búsqueda y reemplazo

La función strfind permite encontrar todas las ocurrencias de una subcadena en una cadena. Su sintaxis es strfind(cadena, subcadena) y devuelve las posiciones en el vector cadena en que comienzan una ocurrencia de subcadena:

```
>> c = 'el collar de coco';
>> strfind(c,'co')
ans =
     4    14    16
>> strfind(c,'no')
ans =
     []
>> length(strfind(c,'co')) %numero de ocurrencias
ans =
     3
```

Realmente, la función strfind es más potente. El segundo parámetro puede ser una expresión regular a buscar. La función strrep permite reemplazar las ocurrencias de una sub-

cadena en una cadena por otra subcadena. Su sintaxis es: strrep(cadena, subcadenavieja, ... subcadenanueva). Ejemplos:

```
>> c = '2$ o 3$';
>> strrep(c, '$', ' dolares')
ans =
2 dolares o 3 dolares
```

6.4.3. Otras funciones

En esta sección vamos a describir alguna función más relacionada con las cadenas. Empezamos con blanks que genera una cadena formada por espacios en blanco. Su uso más habitual consiste en añadir espacios en blanco al concatenar cadenas:

```
>> c = blanks(5)   %cadena con 5 blancos (dificil de ver)
c =

>> length(c)
ans =
     5
>> n = ['Juan' blanks(4) 'Osorio']
n =
Juan     Osorio
>> strcmp(c, repmat(' ', 1, 5) )
ans =
     1
```

Como podemos observar se puede obtener el mismo resultado con repmat, pero la expresión es menos elegante. A veces, puede resultar útil crear una cadena vacía:

```
>> c = ''
c =
     ''
>> length(c)
ans =
     0
>> v = []
v =
     []
>> length(v)
ans =
     0
>> ischar(c)
ans =
```

```
     1
>> ischar(v)
ans =
     0
>> class(c)
ans =
char
>> class(v)
ans =
double
```

Observa que una cadena vacía tiene distinto tipo a un vector vacío, aunque ambos tengan longitud cero.

Existen funciones que permiten comprobar si un carácter es de una determinada categoría, como isletter o isspace que comprueban, respectivamente, si un carácter es una letra o un espacio—espacio en blanco, tabulador o salto de línea.

```
>> isletter('ab2.c')
ans =
     1     1     0     0     1
>> isspace(' - -')
ans =
     1     0     1     0
```

En este sentido la función más general es isstrprop, que permite especificar una cadena y una categoría. Por ejemplo:

```
>> isstrprop('Casa', 'upper')
ans =
     1     0     0     0
```

comprueba qué caracteres son letras mayúsculas. Usa la ayuda de MATLAB para ver qué categorías se pueden consultar.

6.5. Almacenamiento de varias cadenas

Es posible almacenar varias cadenas de caracteres en diferentes filas de una matriz. La única restricción es que todas las cadenas deben tener la misma longitud, porque todas las filas y columnas de una matriz deben tener el mismo número de elementos.

```
>> nombres = ['Ana  '; 'Pilar'; 'Luisa']
nombres =
Ana
```

```
Pilar
Luisa
```

Nos hemos visto obligados a añadir dos espacios en blanco a Ana para obtener una matriz 3x5 de caracteres. MATLAB puede hacer esto por nosotros con la función char:

```
>> nom = char('Ana','Pilar','Luisa')
nom =
Ana
Pilar
Luisa
```

La función deblank elimina los espacios en blanco finales de una cadena y resulta útil para extraer las cadenas almacenadas en una matriz de cadenas.

```
>> nombre1 = nom(1,:)
nombre1 =
Ana
>> length(nombre1)
ans =
     5
>> nombre2 = deblank(nom(1,:))
nombre2 =
Ana
>> length(nombre2)
ans =
     3
```

Sin embargo, persiste un problema, no podemos añadir de forma sencilla una cadena a una matriz de cadenas si la nueva cadena es más larga que las demás. En el Tema 7 veremos los *arrays* de celdas, una estructura de datos que permite, entre otras cosas, almacenar varias cadenas de forma elegante.

La función sortrows permite ordenar alfabéticamente una matriz de cadenas.

```
>> sortrows(nom)
ans =
Ana
Luisa
Pilar
```

Ten en cuenta que las mayúsculas van antes que las minúsculas y que la ñ, al no formar parte del alfabeto inglés, no se ordena de forma correcta.

6.6. Mezclando información de tipo numérico y texto

En un ordenador coexiste información de tipo numérico y de tipo texto. En general, la información numérica se almacena en una representación que facilita la realización de operaciones aritméticas a nivel electrónico. Los tipos más comunes de representaciones de este tipo son el complemento a dos para números enteros y el punto flotante para números decimales. La información de tipo texto se almacena siguiendo una codificación, como ASCII, que asigna a cada carácter un determinado valor.

Aunque una cadena de caracteres puede almacenar caracteres numéricos, los representa de distinta forma a como lo hace una variable de tipo numérico:

```
>> c = '123'
c =
123
>> n = 123
n =
    123
>> c+1 %c almacena los codigos ASCII de los caracteres 1, 2 y 3
ans =
     50     51     52
>> n+1
ans =
    124
```

Observa que MATLAB muestra en la ventana de órdenes las cadenas justificadas a la izquierda, mientras que los números los muestra indentados.

La pantalla del ordenador trabaja en modo texto, es decir, con caracteres ASCII. Por lo tanto, siempre que se quiere mostrar en ella un valor almacenado en modo numérico hay que convertirlo a su cadena de caracteres asociada. Por ejemplo, cuando en el ejemplo anterior se ha visualizado el valor asociado a la variable n, o a la expresión $n + 1$, se ha realizado dicha conversión. La función disp también realiza esta conversión cuando tiene que mostrar datos numéricos.

A veces—veremos ejemplos más adelante—resulta útil mezclar información de tipo numérico y de tipo texto en una cadena de caracteres. Esto, de hecho, ya lo hemos hecho al utilizar la función fprintf, que manda una cadena a la pantalla o a un archivo de texto. Pero, ¿y si queremos almacenar la cadena en una variable o pasarla como parámetro a una función? En MATLAB, esta mezcla se puede realizar de varias formas. Una de ellas es convirtiendo los números a su representación mediante cadenas y concatenando las cadenas. La segunda forma, más sencilla, implica el uso de la función sprintf. Veamos las dos formas.

6.6.1. int2str y num2str

La función int2str convierte un entero a cadena de caracteres. El siguiente código utiliza esta función para crear un nombre de archivo que incluye el número de experimento almacenado en la variable numérica experimento:

```
>> experimento = 18;
>> archivo = ['archivo' int2str(experimento) '.txt']
archivo =
archivo18.txt
```

Cuando se quiere convertir un número decimal a cadena hay que utilizar num2str. Por defecto, num2str crea una cadena con cinco decimales, pero se puede cambiar con el segundo parámetro:

```
>> x = 1/3;
>> ['El resultado es ' num2str(x)]
ans =
El resultado es 0.33333
>> ['El resultado es ' num2str(x,2)]
ans =
El resultado es 0.33
```

Las funciones int2str y num2str también permiten la conversión de una matriz de números:

```
>> cad = int2str([1:4;5:8])
cad =
1  2  3  4
5  6  7  8
>> class(cad)
ans =
char
```

6.6.2. sprintf

Quizá la opción más flexible para combinar información de tipo numérico y de tipo texto sea sprintf. Su sintaxis y funcionamiento es idéntico a fprintf, salvo que devuelve una cadena en lugar de enviarla a la pantalla.

```
>> experimento = 18;
>> archivo = sprintf('archivo%d.dat', experimento)
archivo =
archivo18.dat
```

sprintf nos ofrece otra alternativa a la concatenación de cadenas:

```
>> nombre='Paco'; apellido ='Alonso';
>> sprintf('%s %s', nombre, apellido)
ans =
Paco Alonso
```

6.6.3. Utilidad

En las secciones anteriores hemos visto uno de los usos de la combinación de texto con números. Se trata de crear nombres de archivos que incluyan información numérica parametrizada. Otro uso posible es para los títulos, tanto generales como de los ejes, de los gráficos. Por ejemplo, el siguiente guión:

```
disp('Representacion de la funcion seno')
inf = input('Limite inferior: ');
sup = input('Limite superior: ');
fplot(@sin, [inf sup]);
title(sprintf('Funcion seno en el intervalo [%.2f %.2f]', inf, sup))
```

Solicita la introducción de los límites de un intervalo para la representación gráfica de la función seno. Tras su introducción, crea un gráfico de la función y utiliza la función sprintf para crear el título del gráfico, que incluye el intervalo introducido por el usuario. Si el usuario introduce lo siguiente:

```
>> guion
Representacion de la funcion seno
Limite inferior: 0
Limite superior: 2*pi
```

se obtiene la Figura 6.1.

Otro uso de combinar texto y números consiste en la creación del texto usado en la función input. Por ejemplo, el siguiente guión muestra mensajes al usuario en los que se incluye el ordinal del número a introducir:

```
clc
nombre = input('Introduce tu nombre: ', 's');
fprintf('%s, debes introducir tres numeros\n', nombre)
n = zeros(1,3);
for ind = 1:numel(n)
    n(ind) = input(sprintf('Introduce el numero %d: ', ind));
end
```

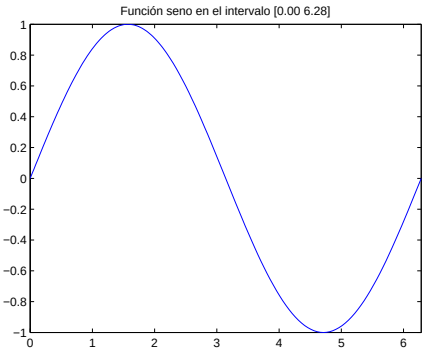

Figura 6.1: Gráfico con título generado con sprintf.

A continuación se muestra una ejecución de este guión:

```
Introduce tu nombre: Mariano
Mariano, debes introducir tres numeros
Introduce el numero 1: 3.5
Introduce el numero 2: 18
Introduce el numero 3: 6.2
```

6.7. Trabajo a bajo nivel con archivos

A lo largo de estos apuntes hemos visto cómo almacenar datos en un archivo y cómo recuperar con posterioridad dichos datos del archivo. En la Sección 1.2 se describe cómo se puede almacenar y recuperar variables en un archivo MAT y en la Sección 2.3 se estudia el almacenamiento y posterior lectura del contenido de matrices en archivos de texto. En este tema vamos a estudiar cómo se puede almacenar información general en un archivo de texto, es decir, cómo se puede guardar el texto que se quiera en el orden que se desee en un archivo de texto y cómo se puede recuperar con posterioridad el texto almacenado en un archivo. Las funciones de trabajo con archivos que hemos estudiado hasta ahora—como load, save o dlmread—se pueden catalogar como de *nivel alto*, en el sentido de que lo único que tenemos que facilitar a la función es el nombre del archivo, la función se encarga de hacer todo el trabajo. Para poder realizar las operaciones con archivos que estudiamos en este tema necesitamos trabajar a un nivel de abstracción más bajo, debemos especificar con exactitud qué datos queremos almacenar o recuperar de un archivo. En esta sección vamos a ver dos operaciones generales que son necesarias para trabajar con un archivo a bajo nivel: la apertura y el cierre. En otras secciones del tema se estudian las funciones que permiten almacenar y extraer información a bajo nivel de un archivo de texto.

En concreto, en esta sección vamos a estudiar las funciones fopen y fclose que permiten abrir y cerrar un archivo respectivamente. Antes de poder utilizar un archivo hay que *abrirlo*. En la operación de apertura se comunica al sistema operativo, que es el encargado de gestionar los archivos, el deseo de trabajar con un archivo. Para abrir un archivo hay que indicar su nombre y el *modo de apertura*. Con el modo de apertura se especifica si se quiere utilizar el archivo para escribir información en él, para leer información de él, para ambas cosas o para añadir información al final del archivo. La sintaxis de fopen es fopen(nombre,modo), donde nombre es una cadena con una trayectoria del archivo y modo es una cadena que indica el modo de apertura. Los modos de apertura son los siguientes:

r apertura para lectura
w apertura para escritura; se pierden los contenidos previos
a se abre o crea un archivo para escritura; se añaden los datos al final del archivo
r+ se abre (no se crea) para lectura y escritura
w+ se abre o crea para lectura y escritura; se pierden los contenidos previos
a+ se abre o crea un archivo para lectura y escritura; se añaden los datos al final del archivo
W apertura para escritura utilizando memoria intermedia (*buffers*)
A se abre para añadir utilizando memoria intermedia

Si se trabaja con archivos de texto, como es nuestro caso, hay que añadir una t al final del modo. fopen devuelve un identificador de archivo y, si no se pudo abrir correctamente el archivo, una cadena indicando el motivo del fracaso en la apertura. El identificador de archivo es un entero positivo que deberá utilizarse en las próximas operaciones de trabajo con el archivo. Si el archivo no se pudo abrir fopen devuelve el identificador -1.

Cuando no se desea trabajar más con un archivo hay que *cerrarlo*. Al cerrarlo el sistema operativo libera todos los recursos que necesita para gestionar su acceso. La función fclose(idA) permite cerrar el archivo de identificador idA; idA debe haberse obtenido tras una llamada a fopen. Es un error tratar de cerrar dos veces un archivo.

El guión de la Figura 6.2 ejemplifica el uso de fopen y fclose. El guión comienza creando un archivo de texto de nombre miMatriz.txt. A continuación intenta abrir los archivos miMatriz.txt y noExisto.txt. Tras el intento de apertura muestra un mensaje en la pantalla indicando si el archivo se abrió con éxito o no. En caso de fracaso indica el motivo y en caso de éxito cierra el archivo. Si el archivo noExisto.txt no existe en su ordenador se debe obtener la siguiente salida:

```
Se abrio con exito el archivo miMatriz.txt
No se pudo abrir noExisto.txt: No such file or directory
```

El guión de la Figura 6.3 realiza el mismo cometido que el guión de la Figura 6.2 pero utilizando un ciclo.

```
clc
m = randi(20, 2, 2);
save('miMatriz.txt','m','-ascii')
[idA, mensaje] = fopen('miMatriz.txt', 'rt');
if idA == -1
    fprintf('No se pudo abrir miMatriz.txt: %s\n', mensaje)
else
    disp('Se abrió con éxito el archivo miMatriz.txt')
    fclose(idA);
end
[idA, mensaje] = fopen('noExisto.txt', 'rt');
if idA == -1
    fprintf('No se pudo abrir noExisto.txt: %s\n', mensaje)
else
    disp('Se abrió con éxito el archivo noExisto.txt')
    fclose(idA);
end
```

Figura 6.2: Apertura y cierre de un archivo.

```
clc
m = randi(20, 2, 2);
save('miMatriz.txt','m','-ascii')
nombres = char('miMatriz.txt','noExisto.txt');
for fila = 1:size(nombres,1)
    nombre = deblank(nombres(fila,:));
    [idA, mensaje] = fopen(nombre, 'rt');
    if idA == -1
        fprintf('No se pudo abrir %s: %s\n', nombre, mensaje)
    else
        fprintf('Se abrió con éxito el archivo %s\n', nombre)
        fclose(idA);
    end
end
```

Figura 6.3: Versión del guión de la Figura 6.2 usando un ciclo.

6.8. Guardar información en un archivo de texto

Para escribir información en un archivo de texto sólo hay que abrir el archivo en modo escritura y utilizar la función fprintf, especificando que su salida debe ir al archivo. Para hacer esto hay que invocar a fprintf utilizando como primer parámetro el identificador de archivo que devolvió fopen.

Hay que tener en cuenta que, por defecto, la escritura de datos en un archivo es secuencial. Esto quiere decir que los datos se almacenan en el archivo uno detrás de otro en el orden en que se escribieron. El guión de la Figura 6.4 ejemplifica la escritura de texto en un archivo. El programa solicita al usuario un entero del 1 al 9. A continuación abre en modo escritura un archivo de nombre TablaN.txt, donde N es el entero introducido por el usuario. Después utiliza fprintf para escribir la tabla de multiplicar del número N en el archivo.

Se puede comprobar que todo ha ido bien desde la misma ventana de órdenes:

```
Introduce un numero entero del 1 al 9: 5
>> type Tabla5.txt

5 x  1 =  5
5 x  2 = 10
5 x  3 = 15
5 x  4 = 20
5 x  5 = 25
5 x  6 = 30
5 x  7 = 35
5 x  8 = 40
5 x  9 = 45
5 x 10 = 50
```

6.9. Lectura de un archivo de texto línea a línea

Un procesamiento habitual de un archivo de texto implica su lectura línea a línea, pues cada línea de texto del archivo puede contener una información independiente a procesar. Por ejemplo, cada línea puede contener información sobre un alumno.

Las funciones fgets y fgetl permiten leer un archivo línea a línea. Estas funciones toman como argumento un identificador de archivo y devuelven una cadena de caracteres con la siguiente línea de texto del archivo asociado al identificador. En caso de que el archivo no contenga más líneas devuelven el valor numérico -1. Al igual que ocurre con la escritura en un archivo, en la lectura los datos se leen secuencialmente, es decir, según el orden en que aparecen en el archivo. La diferencia entre fgets y fgetl es que la primera incluye en la línea devuelta el carácter salto de línea con que acaba cada línea, mientras que la segunda no lo

```
clc
num = input('Introduce un número entero del 1 al 9: ');
nombreArchivo = sprintf('Tabla%d.txt', num);
[idA mensaje] = fopen(nombreArchivo, 'wt');
if idA == -1
    fprintf('No se pudo abrir %s: %s\n', nombreArchivo, mensaje)
else
    for factor = 1:10
        fprintf(idA, '%d x %2d = %2d\n', num, factor, num*factor);
    end
    fclose(idA);
end
```

Figura 6.4: Guión que escribe en un archivo una tabla de multiplicar.

incluye. Normalmente, se utiliza más fgetl, pues el carácter salto de línea no suele interesar, salvo que se vaya a mostrar la línea en pantalla.

El guión de la Figura 6.5 contiene un ejemplo de lectura línea a línea de un archivo. El guión solicita al usuario el nombre de un archivo de texto. Si se puede abrir el archivo en modo lectura, entonces va leyendo las líneas del archivo y las muestra numeradas en la pantalla. Para detectar cuándo no quedan más líneas en el archivo se usa la función ischar aplicada al valor devuelto por fgets, ya que fgets devuelve el número -1 cuando no quedan más líneas en el archivo. Veamos un ejemplo de ejecución del guión:

```
Introduce el nombre del archivo: Tabla5.txt
  1: 5 x  1 =  5
  2: 5 x  2 = 10
  3: 5 x  3 = 15
  4: 5 x  4 = 20
  5: 5 x  5 = 25
  6: 5 x  6 = 30
  7: 5 x  7 = 35
  8: 5 x  8 = 40
  9: 5 x  9 = 45
 10: 5 x 10 = 50
```

Puesto que las líneas leídas se muestran en la pantalla se ha optado por utilizar fgets, también se puede utilizar fgetl, pero en ese caso la función fprintf debe incluir el salto de línea en su cadena de formato—véase la Figura 6.6. Este último guión también ilustra que se puede utilizar la instrucción return dentro de un guión, en cuyo caso se termina su ejecución. En el guión de la Figura 6.6, si no se puede abrir el archivo se termina la ejecución del guión, pues en caso de un fallo en la apertura no estamos interesados en realizar más cálculos.

```
clc
nombreArchivo = input('Introduce el nombre del archivo: ', 's');
[idA mensaje] = fopen(nombreArchivo, 'rt');
if idA == -1
    fprintf('No se pudo abrir %s: %s\n', nombreArchivo, mensaje)
else
    numLinea = 1;
    linea = fgets(idA);
    while ischar(linea)
        fprintf('%3d: %s', numLinea, linea);
        numLinea = numLinea + 1;
        linea = fgets(idA);
    end
    fclose(idA);
end
```

Figura 6.5: Guión que lee un archivo de texto línea a línea.

```
clc
nombreArchivo = input('Introduce el nombre del archivo: ', 's');
[idA mensaje] = fopen(nombreArchivo, 'rt');
if idA == -1
    fprintf('No se pudo abrir %s: %s\n', nombreArchivo, mensaje)
    return
end
numLinea = 1;
linea = fgetl(idA);
while ischar(linea)
    fprintf('%3d: %s\n', numLinea, linea);
    numLinea = numLinea + 1;
    linea = fgetl(idA);
end
fclose(idA);
```

Figura 6.6: Lectura de un archivo de texto línea a línea con fgetl.

En el guión de la Figura 6.5 se solicita al usuario el nombre del archivo mediante la función input. Una alternativa sería utilizar uigetfile. Esta función abre una ventana gráfica en la que el usuario puede utilizar el ratón para navegar por el sistema de archivos para seleccionar un nombre de archivo. Prueba a sustituir la segunda línea del guión por:

```
nombreArchivo = uigetfile();
```

Si se quiere seleccionar un archivo que no se encuentra en la carpeta de trabajo hay que utilizar la siguiente sintaxis:

```
>> [nombre carpeta] = uigetfile()
nombre =
UsoArea.m
carpeta =
C:\Users\Paco\Dropbox\codigoApuntes\graficosSimples\
>> nombre = fullfile(carpeta,nombre)
nombre =
C:\Users\Paco\Dropbox\codigoApuntes\graficosSimples\UsoArea.m
>> type(nombre) %visualizar el contenido del archivo seleccionado

x = linspace(-5,5,100);
area(x, x.^2-2*x, -2)
```

Utilizando dos parámetros de salida uigetfile devuelve el nombre del archivo y la trayectoria absoluta de la carpeta que contiene el archivo. Con la función fullfile se obtiene una trayectoria absoluta del archivo, lista para usarse en funciones como fopen o type.

6.10. Extracción de símbolos de una cadena

Una cadena puede contener información variada separada por *delimitadores*. La función strtok permite extraer información delimitada de una cadena. A cada uno de los elementos de información extraída se le llama *símbolo*—del inglés *token*.

strtok permite extraer un símbolo, por lo que si se quieren extraer n símbolos de una cadena habrá que invocar a strtok n veces. El formato es:

```
[simbolo resto] = strtok(cadena, delimitadores)
```

delimitadores es una cadena que incluye los delimitadores, si no se especifica este parámetro se toma como delimitadores a los espacios blancos—espacio en blanco, tabulador y salto de línea. La función devuelve en simbolo los caracteres iniciales de cadena, ignorando los delimitadores iniciales, hasta que se encuentra un delimitador. En resto se devuelve el resto de la cadena incluido el delimitador que dio término al símbolo. Veamos un ejemplo

```
clc
f = input('Introduce una frase: ', 's');
cont = 1;
[s f] = strtok(f,'.;:,” ');
while ¬isempty(s)
    fprintf('Palabra %d: %s\n', cont, s)
    [s f] = strtok(f,'.;:,” ');
    cont = cont + 1;
end
```

Figura 6.7: Extrae las palabras de una frase usando strtok.

```
>> cad = '   pienso    luego    existo ';
>> [s r] = strtok(cad) %se extrae la primera cadena
s =
pienso
r =
    luego    existo
>> [s r] = strtok(r)   %se extrae la segunda cadena
s =
luego
r =
    existo
```

Cuando no se sabe cuántos símbolos incluye una cadena se debe utilizar strtok en un ciclo. Por ejemplo, el guión de la Figura 6.7 solicita una frase y extrae las palabras que se encuentran separadas por ciertos signos de puntuación:

Veamos un ejemplo de ejecución del guión:

```
Introduce una frase: ”De hoy no pasa”, dijo Sancho.
Palabra 1: De
Palabra 2: hoy
Palabra 3: no
Palabra 4: pasa
Palabra 5: dijo
Palabra 6: Sancho
```

6.10.1. str2num y str2double

Los símbolos extraídos de una cadena pueden representar números. En dicho caso, si se quieren procesar numéricamente habrá que obtener su representación numérica. Las funcio-

nes str2num y str2double permiten convertir cadenas de caracteres que almacenan números a números en punto flotante. La primera permite convertir varios números separados por caracteres blancos, mientras que la segunda sólo permite convertir un número.

```
>> str2double('45.6')
ans =
   45.6000
>> str2num('45.6  23  16.8 ')
ans =
   45.6000   23.0000   16.8000
```

Vamos a ver un ejemplo en que se extraen símbolos con contenido numérico. Se tiene un archivo de texto con líneas con la siguiente estructura:

```
Nombre: edadHijo1 edadHijo2 ...
```

Cada línea del archivo representa un trabajador. Empieza con su nombre y sigue con las edades de sus hijos separadas por espacios. Un ejemplo concreto de archivo es:

```
Luis: 2 4 8
Maite: 11 4
Eduardo:
Julia: 5
```

A partir de un archivo con esta estructura se quiere obtener un listado con los nombres de los empleados, indicándose para cada empleado su número de hijos y la edad del hijo mayor. El guión de la Figura 6.8 realiza este procesamiento; lee el archivo línea a línea y para cada línea usa strtok para extraer el nombre del empleado. El resto de línea, salvo el primer carácter que es el carácter dos puntos, contiene las edades de los hijos separadas por espacios, por lo que se aplica str2num para obtener su representación numérica. La salida generada por el guión para el archivo de ejemplo es:

```
Luis tiene 3 hijos y el mayor 8 anos
Maite tiene 2 hijos y el mayor 11 anos
Eduardo no tiene hijos
Julia tiene 1 hijos y el mayor 5 anos
```

6.10.2. sscanf y fscanf

Las funciones sscanf y fscanf permiten extraer símbolos y, a la vez, convertir texto a números. La primera trabaja con una cadena y la segunda lee de un archivo de texto. Aunque son muy flexibles su uso resulta complejo por lo que no las vamos a estudiar en estos apuntes. Utiliza la ayuda si estás interesado en saber cómo funcionan.

```
clc
[idA mensaje] = fopen('Hijos.txt', 'rt');
if idA == -1
    fprintf('No se pudo abrir Hijos.txt: %s\n', mensaje)
    return
end
linea = fgetl(idA);
while ischar(linea)
    [nombre h] = strtok(linea,':');
    hijos = str2num(h(2:end));
    if isempty(hijos)
        fprintf('%s no tiene hijos\n', nombre)
    else
        fprintf('%s tiene %d hijos y el mayor %d años\n',nombre,...
                length(hijos), max(hijos));
    end
    linea = fgetl(idA);
end
fclose(idA);
```

Figura 6.8: Guión que extrae símbolos numéricos de una cadena.

6.11. Ejercicios

1. La función date devuelve una cadena de caracteres con la fecha actual:

   ```
   >> date
   ans =
   03-Oct-2014
   ```

   Escribe un guión que invoque a date, extraiga de la cadena de salida el año actual y muestre en la salida el próximo año.

2. Implementar el juego del ahorcado. El programa dispondrá de un menú con tres opciones: introducir palabra, adivinar palabra y salir. La primera opción permite introducir la palabra que otro jugador—o nosotros mismos, para probar el programa—ha de adivinar. La segunda opción sólo podrá llevarse a cabo si ha sido introducida previamente una palabra. De ser así aparecerá una cadena formada por guiones—tantos como letras contiene la palabra. El programa irá pidiendo una letra tras otra. Si la letra es válida aparecerá en la cadena en la posición correspondiente; si no es así contaremos un fallo. El programa termina cuando se han acertado todas las letras o se ha fallado seis veces.

3. Cambia la implementación del ejercicio anterior de forma que las palabras se seleccionen aleatoriamente de una matriz de caracteres.

4. Modifica el ejercicio anterior de forma que las palabras procedan de un archivo de texto. En el archivo cada línea debe contener exactamente una palabra. Al iniciarse la ejecución del programa las palabras almacenadas en el archivo deben cargarse en una matriz de caracteres. Observa que para hacer esto debes saber la longitud máxima de las palabras del archivo. Por lo tanto, tienes que leer dos veces el archivo; la primera vez para calcular la palabra más larga y la segunda vez para almacenar las palabras en la matriz. Cuando hayas leído el archivo para calcular la palabra más larga, debes cerrarlo y volver a abrirlo para poder leerlo por segunda vez. Otra posibilidad es usar la función frewind, que sitúa el puntero de lectura de un archivo abierto al principio del contenido del archivo.

5. Realiza un guión o una función que, dados un archivo de texto y dos cadenas, cree un archivo de texto igual al original, pero en el que las ocurrencias de la primera cadena se hayan sustituido por ocurrencias de la segunda cadena.

   Sugerencia: lee el archivo línea a línea y haz las sustituciones en cada línea.

6. Un archivo de texto incluye las coordenadas de varios polígonos. Un ejemplo de archivo es el siguiente:

```
4
0,0
1,0
1,1
0,1
3
2,0
3,0
2.5,1
```

   El formato consiste en una línea con el número de vértices del polígono, seguida por los vértices; cada vértice en una línea con las coordenadas separadas por comas. Escribe un guión que lea un archivo con este formato y dibuje los polígonos con la función fill . La Figura 6.9 muestra el tipo de resultado que debería producir el guión para el archivo de ejemplo.

7. Realiza un programa que permita jugar a las 3 en raya. El tablero se representa mediante una matriz 3x3 de caracteres. Inicialmente la matriz sólo incluye espacios en blanco y después se añaden los caracteres 'x' y 'o' según avanza el juego. Se sugiere realizar una descomposición en funciones del programa. Por ejemplo, puedes escribir funciones que:

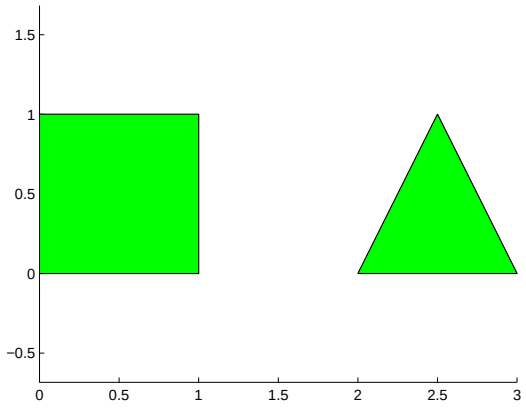

Figura 6.9: Resultado del guión que lee y dibuja polígonos.

- Dado un tablero y un carácter que indica el jugador—es decir, una 'x' o una 'o'—solicite de teclado una posición del tablero e inserte la ficha en la posición. La función debe verificar que es correcto el poner la ficha en la posición.
- Dado un tablero y un carácter indicando el jugador compruebe si el jugador ha ganado la partida.

Tema 7
Arrays de celdas

Una estructura de datos es un tipo de dato que permite almacenar más de un dato o elemento. Los *arrays* que vimos en el Tema 2 son estructuras de datos homogéneas, es decir, almacenan elementos del mismo tipo; además, los datos deben tener una representación interna numérica. Los *arrays de celdas* son *arrays* que almacenan valores de tipos distintos—son estructuras de datos heterogéneas—compartiendo con los *arrays* normales el tipo de indexación, aunque con algunas diferencias fruto de almacenar elementos de distinto tipo.

7.1. *Arrays* de celdas

Como hemos comentado anteriormente, un *array* de celdas es una estructura de datos que permite almacenar un conjunto de datos de distinto tipo. Además, se utilizan índices para acceder a sus elementos. Los elementos de un *array* de celdas pueden, a su vez, ser celdas, con lo que podemos ver a un *array* de celdas como un contenedor de contenedores.

Internamente los *arrays* de celdas almacenan punteros a las celdas. Es decir, las direcciones de memoria donde realmente se guardan los datos asociados a las celdas. El concepto es análogo a los *arrays* de objetos de Java.

Los *arrays* de celdas tienen una sintaxis prolija y algo confusa. En estos apuntes no vamos a estudiar todas las opciones de trabajo con *arrays* de celdas, nos centraremos en las opciones que consideramos más importantes.

7.2. Creación de *arrays* de celdas

La forma más sencilla de crear un *array* de celdas consiste en encerrar sus elementos entre llaves, usando puntos y comas y comas al estilo de los *array* numéricos para separar filas y elementos dentro de las filas.

```
>> ac = {125, 'hola'; 1:5, true}
```

```
ac =
    [        125]    'hola'
    [1x5 double]    [    1]
>> class(ac)
ans =
cell
```

Aquí se ha creado un *array* 2x2 de celdas formado por un escalar, una cadena, un vector numérico y un valor lógico. El tipo de un *array* de celdas es *cell*. También se puede crear un *array* de celdas asignando valores individuales al *array*:

```
>> ac2{1} = 'Luis'
ac2 =
    'Luis'
>> ac2{2} = 28
ac2 =
    'Luis'    [28]
>> class(ac2)
ans =
cell
```

Esto, sin embargo, es ineficiente. Si se conoce de antemano el tamaño del *array*, aunque no sus elementos, es mejor preasignar la memoria utilizando la función cell:

```
>> matrizCeldas = cell(2,2)
matrizCeldas =
    []    []
    []    []
```

Los elementos de un *array* creado con cell son vectores vacíos; posteriormente se puede asignar valores adecuados a sus elementos.

7.3. Indexado de contenido

Al igual que con los *arrays* numéricos, se puede hacer referencia a los elementos individuales de un *array* de celdas. Sin embargo, los elementos de un *array* de celdas son celdas, por lo que existen dos tipos de indexado: uno permite acceder a las celdas y el otro al contenido de las celdas.

En esta sección vamos a estudiar el acceso al contenido de las celdas, éste se logra mediante el uso de llaves.

```
>> celdas = {125, 'hola'; 1:5, true};
>> celdas{1,1}
```

```
ans =
   125
>> celdas{1,1} = 34
celdas =
    [        34]    'hola'
    [1x5 double]    [    1]
```

En este ejemplo se ha accedido al contenido de una celda para visualizar sus datos y para cambiar su valor. Si una celda contiene un vector, se puede acceder a los elementos del vector con la siguiente sintaxis:

```
>> celdas{2,1}      % acceso al vector
ans =
     1     2     3     4     5
>> celdas{2,1}(2)   % acceso al segundo elemento del vector
ans =
     2
>> celdas{2,1}(2) = 4;
>> celdas{2,1}
ans =
     1     4     3     4     5
```

Se puede acceder al contenido de varios elementos de un *array* de celdas. Por ejemplo:

```
>> celdas{1,:} % accede a la primera fila
ans =
   125
ans =
hola
>> [num cadena] = celdas{1,:}
num =
   125
cadena =
hola
```

La función celldisp visualiza el contenido de todos los elementos de un *array* de celdas:

```
>> celldisp(celdas)
celdas{1,1} =
   125
celdas{2,1} =
     1     2     3     4     5
celdas{1,2} =
hola
celdas{2,2} =
```

```
     1
```

Las funciones length, size y numel tienen la funcionalidad ya conocida:

```
>> length(celdas)
ans =
     2
>> size(celdas)
ans =
     2     2
>> numel(celdas)
ans =
     4
```

7.4. Indexado de celdas

Con el indexado de celdas se accede a la celda—contenedor—y no a su contenido. La indexación de celdas se realiza con los paréntesis. Veamos un ejemplo:

```
>> celdas(2,1)
ans =
    [1x5 double]
>> class(celdas(2,1)) %indexado de celdas
ans =
cell
>> class(celdas{2,1}) %indexado de contenido
ans =
double
```

En este ejemplo se accede a la celda de índices (2,1) del *array* y se obtiene un *array* de una celda. Con la siguiente asignación se crea un *array* de celdas formado por la primera fila del *array* celdas:

```
>> subc = celdas(1,:)
subc =
    [125]    'hola'
```

Para eliminar celdas de un *array* de celdas hay que utilizar la indexación de celdas:

```
>> celdas(:,2) = [] %elimina la segunda columna
celdas =
    [     125]
    [1x5 double]
```

Para eliminar el contenido de una celda hay que utilizar indexación de contenido:

```
>> celdas{2,1} = []
celdas =
    [125]
    []
```

La función cellplot crea un gráfico visualizando un *array* de celdas. No es muy útil, pues visualiza las celdas y no sus contenidos.

7.5. Usos de *arrays* de celdas

En esta sección se describen varios usos típicos de los *arrays* de celdas. En realidad ya hemos utilizado alguno. Por ejemplo, cuando en una instrucción switch—Sección 3.2—un caso puede tomar varios valores éstos se especifican mediante un *array* de celdas. También los hemos utilizado para especificar las etiquetas de un diagrama de sectores—Sección 4.5.

7.5.1. Almacenamiento de cadenas de caracteres

En la Sección 6.5 se describió cómo almacenar varias cadenas en distintas filas de una matriz. Un requisito incómodo es que todas las cadenas tienen que tener la misma longitud. Con los *arrays* de celdas no se tiene ese inconveniente.

```
>> nombres = {'Ana','Pilar','Luisa'};
>> length(nombres{1})
ans =
     3
>> length(nombres{2})
ans =
     5
```

Para calcular la longitud de la cadena más larga de un *array* de cadenas podemos emplear un ciclo, pero también se puede aplicar la función cellfun. Esta función es análoga a arrayfun, pero aplicada a *arrays* de celdas

```
>> cellfun(@length, nombres)          %longitud de las cadenas
ans =
     3     5     5
>> max(cellfun(@length, nombres))     %longitud maxima
ans =
     5
```

Existen funciones que permiten realizar conversiones entre *arrays* de caracters y *arrays* de celdas que almacenan cadenas. La función cellstr convierte una matriz de caracteres con espacios en blanco finales a un *array* de celdas en el que se han eliminado las espacios finales de las cadenas.

```
>> nombres = char('Ana','Pilar','Magdalena');
>> nom = cellstr(nombres)
nom =
    'Ana'
    'Pilar'
    'Magdalena'
```

La conversión inversa se puede realizar con char:

```
>> productos = {'peras', 'manzanas', 'higos'};
>> p = char(productos)
p =
peras
manzanas
higos
>> size(p)
ans =
     3     8
```

Las función iscellstr devuelve un valor lógico indicando si su argumento es un *array* de celdas en que todas sus celdas contienen cadenas.

```
>> iscellstr(productos)
ans =
     1
>> iscellstr({1, 'hola'})
ans =
     0
```

La función sort permite ordenar alfabéticamente un *array* de celdas de cadenas.

```
>> sort(productos)
ans =
    'higos'    'manzanas'    'peras'
```

Ten en cuenta que las mayúsculas van antes que las minúsculas y que la ñ, al no formar parte del alfabeto inglés, no se ordena de forma correcta.

7.5.2. Funciones con un número indeterminado de parámetros

En la Sección 5.5 se estudió el uso de las funciones nargin y nargout que permiten que una función conozca con cuántos parámetros de entrada y de salida fue invocada. Esto hace posible escribir una función con varios parámetros y luego utilizar los parámetros que realmente se han usado al invocar a la función. Esto puede ser útil si una función toma un número de parámetros máximo pequeño. Sin embargo, si el número de parámetros es indefinido la escritura de una función basándose en nargin y nargout puede resultar tedioso.

Las variables internas varargin y varargout son útiles cuando una función puede tomar un número indeterminado de parámetros. varargin recibe los parámetros de entrada y en varargout se puede almacenar los parámetros de salida. Ambas son *arrays* de celdas porque una función puede recibir parámetros de distinto tipo.

Vamos a ver un ejemplo de uso de varargin. En la Sección 3.9 se estudió la función interna menu, que permite visualizar un menú en una ventana y devuelve la opción seleccionada por el usuario. Vamos a realizar una versión de esta función en modo texto. La función recibirá los textos asociados a las distintas opciones de un menú, los mostrará numerados en la pantalla y devolverá el número asociado a la elección del usuario. El número de opciones dependerá del menú que quiera construir el usuario de la función, por lo que se presta al uso de la variable varargin. La función de la Figura 7.1 muestra una implementación. Observa el uso de varargin para recibir todas las opciones del menú. A continuación se muestra una llamada a la función desde la ventana de órdenes:

```
>> o = miMenu('Sumar','Multiplicar','Salir')
----------------
1: Sumar
2: Multiplicar
3: Salir
----------------
Selecciona una opcion (1-3): 2
o =
     2
```

Las variables varargin y varargout pueden aparecer en la cabecera de una función con otros parámetros, pero deben ser los últimos parámetros de la lista.

7.5.3. Argumentos de funciones

Algunas funciones tienen parámetros de tipo *array* de celdas; suele ocurrir cuando un parámetro almacena una lista de cadenas de caracteres, pues los *arrays* de celdas son su representación más adecuada. Un ejemplo es pie, que se estudió en la Sección 4.5. Esta función toma como primer parámetro un vector con números de ocurrencias y dibuja un diagrama de sectores que representa el porcentaje de ocurrencias de cada valor. Los sectores se etiquetan

```
function op = miMenu(varargin)
    nopciones = length(varargin);
    marco = repmat('-',1,max(cellfun(@length,varargin))+5);
    mensaje = sprintf('Selecciona una opción (1-%d): ', nopciones);
    op = 0;
    while op < 1 || op > nopciones
        disp(marco)
        for num = 1:nopciones
            fprintf('%d: %s\n', num, varargin{num})
        end
        disp(marco)
        op = input(mensaje);
    end
end
```

Figura 7.1: Función con un número indefinido de parámetros

con los porcentajes, pero es posible etiquetar los sectores con otros valores. Para hacerlo hay que utilizar un segundo parámetro con un *array* de celdas, las celdas contienen cadenas de caracteres que indican los valores a usar como etiquetas.

Como ejemplo vamos a retomar el problema de generar un diagrama de sectores con las edades de una clase— Sección 4.5—y vamos a utilizar un *array* de celdas para especificarle a pie los valores de las edades. El guión de la Figura 7.2 produce dos diagramas de sectores: uno etiquetado con porcentajes y otro con las edades de los alumnos—la Figura 7.3 contiene el resultado de una ejecución del guión. Vamos a comentar el guión:

- La primera instrucción genera 20 edades aleatorias uniformemente distribuidas en el rango $[20, 25]$.
- La segunda instrucción obtiene las edades del vector edades ordenadas y sin repetidos.
- La tercera instrucción calcula cuántas ocurrencias hay de cada valor—edad—del vector edadesSinR en el vector edades.
- Las instrucciones de la 4 a la 6 dibujan un diagrama de sectores etiquetado con los porcentajes de ocurrencia de las edades.
- Las instrucciones de la 7 a la 13 dibujan un diagrama de sectores etiquetado con las edades. Las instrucciones de la 7 a la 10 crean el *array* de celdas que contiene las etiquetas que constituyen el segundo parámetro de pie. Observa el uso de la función int2str para obtener las cadenas de caracteres que precisa la función pie.

```
edades = randi([20 25], 1, 20); %20 edades aleatorias en [20,25]
edadesSinR = unique(edades);     %edades ordenadas sin repetidos
ocurrencias = histc(edades,edadesSinR); %ocurrencias de cada edad
subplot(1,2,1)
pie(ocurrencias)              %diagrama de ocurrencias con porcentajes
title('Etiquetas con porcentajes')
porcentajes = ocurrencias/sum(ocurrencias)*100;
edadesSinRCadena = cell(1,length(edadesSinR)); %array de celdas
for ind = 1:length(edadesSinR)
    edadesSinRCadena{ind} = sprintf('%d (%.1f%%)', edadesSinR(ind), ...
        porcentajes(ind));
end
subplot(1,2,2)
pie(ocurrencias,edadesSinRCadena)
title('Etiquetas con valores y porcentajes')
```

Figura 7.2: Guión que llama a la función pie con un parámetro de tipo *array* de celdas

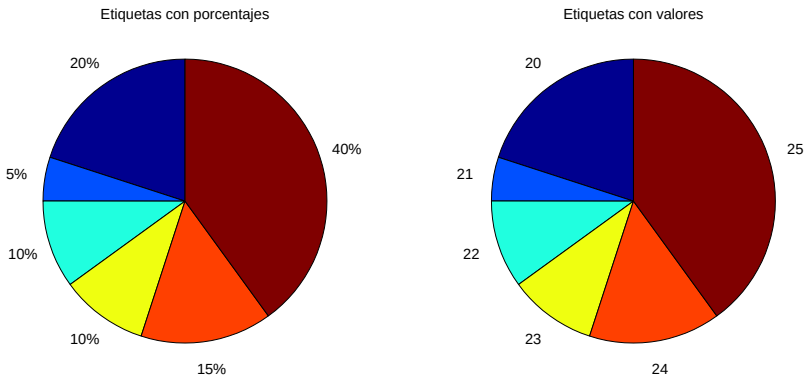

Figura 7.3: Diagrama de sectores resultado de una ejecución del guión de la Figura 7.2.

```
clc
vf = {@length, @sum, @prod};
v = 1:5;
fprintf('Hola, soy el vector [%s]. Ordéname:\n', sprintf('%d ',v))
fprintf('\t1. Número de elementos\n\t2. Suma\n\t3. Producto\n')
op = input('Elige: ');
if op ≥ 1 && op ≤ length(vf)
    fprintf('Solución: %d\n', vf{op}(v))
end
```

Figura 7.4: Ejemplo de uso de *array* de punteros a funciones

7.5.4. *Arrays* de punteros a funciones

En la Sección 5.10 se introdujo el concepto de puntero a función. Los *arrays* de celdas permiten almacenar punteros a funciones. En el guión de la Figura 7.4 se utiliza un *array* de punteros a funciones para almacenar las posibles operaciones con un vector. Es un ejemplo un tanto artificial, pero los *arrays* de punteros a funciones juegan un papel importante en ciertas aplicaciones, como la generación automática de código.

7.6. Ejercicios

1. Modifica el ejercicio sobre trayectorias de proyectiles del Tema 4, de forma que en el gráfico aparezca una leyenda indicando los distintos ángulos, como en la Figura 7.5. Observa que la función legend acepta como parámetro un *array* de celdas con cadenas de caracteres.

2. Modifica el guión 7.2 de forma que las etiquetas indiquen las edades y los porcentajes, por ejemplo, como en la Figura 7.6.

3. Realiza un guión que lea un archivo de texto y lo almacene con las líneas ordenadas alfabéticamente en otro archivo. Si el archivo de texto se llama *archivo.txt*, entonces el archivo ordenado debe llamarse *archivo.ord.txt*.

4. Modifica el ejercicio del juego del ahorcado del tema 6 para que las palabras se lean de un archivo de texto y se almacenen en memoria en un *array* de celdas.

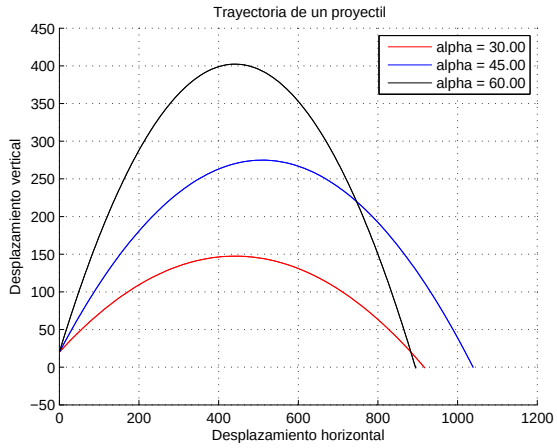

Figura 7.5: Trayectoria de un proyectil ($ai = 20m, vi = 100m/s, \alpha = [30^0, 45^0, 60^0]$).

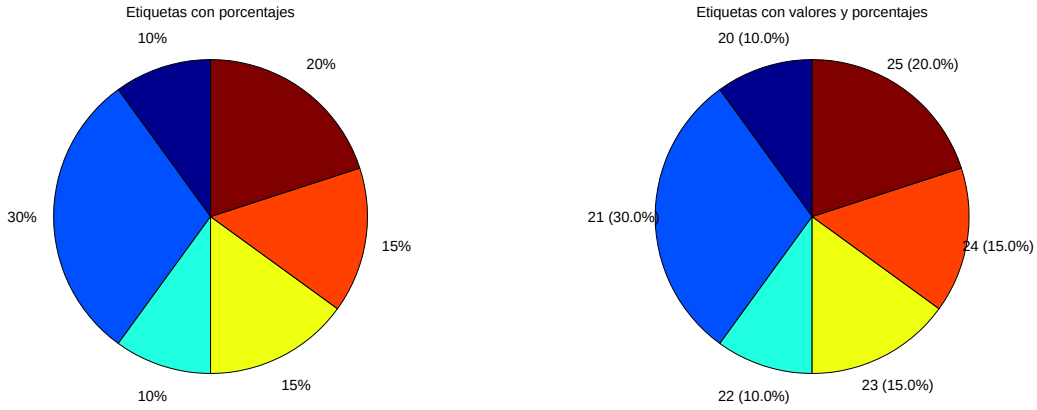

Figura 7.6: Etiquetas con valores y porcentajes

Tema 8
Estructuras

Las *estructuras*, también llamadas *registros* en muchos lenguajes de programación, son un tipo de dato soportado por todos los lenguajes de programación modernos. Una estructura permite crear variables con datos de distintos tipos. El poder almacenar datos de distintos tipos hace que una estructura pueda representar cualquier entidad real o imaginaria, como un alumno, un coche, una coordenada cartesiana ... Cada variable de una estructura se llama *campo* y tiene asociada un nombre. A diferencia de un *array* de celdas, en el que se utiliza un índice para acceder a sus datos, una estructura accede a sus campos a través de sus nombres, lo que hace que las estructuras tengan asociado un código muy fácil de leer y comprender. También es posible crear *arrays* de estructuras para almacenar una colección de estructuras, como los alumnos de una clase.

8.1. Creación de una estructura

Existen varias formas de crear una estructura. Para ilustrarlas vamos a utilizar un ejemplo en que queremos representar información sobre un alumno. Hemos decidido guardar cuatro datos de un alumno: su nombre, dni, edad y nota final. Se utilizará una estructura con cuatro campos—de nombres: nombre, dni, edad y notaFinal—para almacenar los datos de un alumno.

La primera posibilidad de creación de una estructura es utilizar el operador punto (.) para hacer referencia a los campos de la estructura:

```
>> alumno.nombre = 'Juan Viedma Contreras';
>> alumno.dni = '12345678P';
>> alumno.edad = 23;
>> alumno.notaFinal = 8.4;
```

Este código crea una variable alumno, a la que se le añaden campos gradualmente. Esto no es muy eficiente, por lo que se puede utilizar la función struct:

```
function salida = creaAlumno(nombre, dni, edad, notaFinal)
   salida = struct('nombre', nombre, 'dni', dni, 'edad', edad, ...
      'notaFinal', notaFinal);
end
```

Figura 8.1: Función que crea una estructura de tipo alumno a partir de valores de sus campos.

```
>> al = struct('nombre','Miguel Cano Salas','dni','22222222P',...
               'edad', 24, 'notaFinal', 7.5);
```

La función struct toma como parámetros pares nombre del campo y valor del campo. Los nombres de los campos se expresan como cadenas de caracteres. struct devuelve la estructura creada que, en el ejemplo, se asigna a la variable al. Aunque el uso de struct resulta compacto, implica la repetición de los nombres de los campos cada vez que se crea una estructura. Una alternativa consiste en escribir una función que construye las estructuras, como la de la Figura 8.1. Dada esta función es posible crear nuevas estructuras sin especificar los nombres de los campos:

```
>> alu = creaAlumno('Eva Solana Chica', '11111111P', 23, 8);
```

Como la función creaAlumno sólo contiene una instrucción, podría escribirse como una función anónima. Para visualizar el contenido de una estructura podemos escribir el nombre de la variable que la contiene en la ventana de órdenes o usar la función disp:

```
>> alumno
alumno =
        nombre: 'Juan Viedma Contreras'
           dni: '12345678P'
         edad: 23
    notaFinal: 8.4000
>> disp(al)
        nombre: 'Miguel Cano Salas'
           dni: '22222222P'
         edad: 24
    notaFinal: 7.5000
```

No existe un especificador de formato en fprintf para mostrar una estructura. Por lo tanto, para mostrar el contenido de una estructura utilizando fprintf hay que ir mostrando los distintos campos que interesan:

```
>> fprintf('%s: %.2f\n', alu.nombre, alu.notaFinal)
```

```
Eva Solana Chica: 8.00
```

En este caso se ha mostrado los campos nombre y notaFinal de la variable alu. El tipo de una variable estructura es struct:

```
>> class(alu)
ans =
struct
```

La función fieldnames devuelve un *array* de celdas con los nombres de los campos de una estructura:

```
>> fieldnames(alu)
ans =
    'nombre'
    'dni'
    'edad'
    'notaFinal'
```

8.2. Acceso y modificación de una estructura

Para acceder al contenido de un campo de una estructura se puede utilizar el operador punto o la función getfield:

```
>> fprintf('%s (%d)\n', alu.nombre, getfield(alu,'edad'))
Eva Solana Chica (23)
```

En el ejemplo, se ha utilizado el operador punto para acceder al campo nombre de la variable alu y la función getfield para acceder al campo edad. Cuando se utiliza el operador punto, el nombre del campo puede estar almacenado en una variable:

```
>> alu.notaFinal
ans =
     8
>> campo = 'notaFinal';
>> alu.(campo)
ans =
     8
```

La función getfield toma como parámetros una estructura y el nombre de uno de sus campos y devuelve el valor asociado al campo en la estructura. Para modificar un campo de una estructura se puede utilizar el operador punto o la función setfield:

```
>> alumno.edad = alumno.edad+1
alumno =
        nombre: 'Juan Viedma Contreras'
           dni: '12345678P'
         edad: 24
     notaFinal: 8.4000
>> al = setfield(al,'edad', 25)
al =
        nombre: 'Miguel Cano Salas'
           dni: '22222222P'
         edad: 25
     notaFinal: 7.5000
```

Ten en cuenta que la función setfield no modifica la estructura que recibe como primer parámetro, sino que devuelve una nueva estructura con el parámetro modificado. Si se quiere, por tanto, modificar una estructura habrá que asignar el valor devuelto por setfield a la propia estructura.

La función rmfield devuelve una estructura en la que se ha eliminado un campo—una acción que raramente tiene sentido en un programa:

```
>> al = rmfield(al,'dni')
al =
        nombre: 'Miguel Cano Salas'
         edad: 25
     notaFinal: 7.5000
```

Se puede consultar si un nombre es un campo de una estructura con la función isfield:

```
>> isfield(al,'nombre')
ans =
     1
>> isfield(al,'apellido')
ans =
     0
```

8.3. Estructuras anidadas

Un campo de una estructura puede ser a su vez una estructura, lo que da lugar a una *estructura anidada*. Por ejemplo, a continuación se crea una estructura movimiento bancario que consta de un valor y una fecha. La fecha es, a su vez, una estructura que consta de los campos día, mes y año.

```
>> m1 = struct('valor', 1000, 'fecha', ...
    struct('dia',16,'mes',12,'anio',2013))
m1 =
    valor: 1000
    fecha: [1x1 struct]
>> m1.fecha
ans =
     dia: 16
     mes: 12
    anio: 2013
>> m1.fecha.dia
ans =
     16
```

Utilizando el operador punto la estructura se crea así:

```
>> m2.valor = 2000;
>> m2.fecha.dia = 1;
>> m2.fecha.mes = 4;
>> m2.fecha.anio = 2013;
>> m2
m2 =
    valor: 2000
    fecha: [1x1 struct]
>> m2.fecha.mes
ans =
     4
```

8.4. *Arrays* de estructuras

En general, cuando se trabaja con estructuras se desea almacenar colecciones de estructuras para, por ejemplo, almacenar los alumnos de una clase, los clientes de un banco o los parámetros y resultados de varios experimentos. Para almacenar una colección de estructuras se puede usar un *array*. A continuación ilustramos la creación de un vector de estructuras utilizando el ejemplo de los datos de alumnos. Existen varias formas de crear un *array* de estructuras. La primera es indexando el *array* y usando el operador punto. Por ejemplo:

```
>> vector1(1).nombre = 'Juan Viedma Contreras';
>> vector1(1).dni = '12345678P';
>> vector1(1).edad = 23;
>> vector1(1).notaFinal = 8.4;
>> vector1(2).nombre = 'Miguel Cano Salas';
>> vector1(2).dni = '22222222P';
```

```
>> vector1(2).edad = 24;
>> vector1(2).notaFinal = 7.5;
```

Aquí se ha creado un *array* de dos estructuras de nombre vector1. También se puede crear un vector de estructuras utilizando la función struct. Si se utiliza struct para crear un *array* de estructuras, hay que emplear *arrays* de celdas para almacenar los valores de los campos:

```
>> nombres = {'Juan Viedma Contreras','Miguel Cano Salas','Eva Solana ...
    Chica'};
>> dnis = {'12345678P', '22222222P', '11111111P'};
>> edades = {23, 24, 23};
>> notas = {8.4, 7.5, 8};
>> vector2 = ...
    struct('nombre',nombres,'dni',dnis,'edad',edades,'notas',notas);
```

También es posible utilizar un único valor, en cuyo caso todas las estructuras del vector toman ese valor. Por ejemplo, si en el código anterior la variable edades valiera 22, entonces todos los alumnos tendrían la edad 22. Si hemos escrito una función como la de la Figura 8.1 para crear una estructura, también se puede usar para crear los elementos de un vector:

```
>> vector3(1) = creaAlumno('Juan Viedma Contreras','12345678P',23,8.4);
>> vector3(2) = creaAlumno('Miguel Cano Salas','22222222P',24,7.5);
>> vector3(3) = creaAlumno('Eva Solana Chica', '11111111P', 23, 8);
```

Se puede crear un *array* vacío de estructuras con la siguiente sintaxis:

```
>> v = struct([])
v =
0x0 struct array with no fields.
```

8.4.1. Gestión de un *array* de estructuras

Se puede acceder a las estructuras de un *array* utilizando indexación y a los campos utilizando el operador punto. Por ejemplo:

```
>> vector1(1).notaFinal = 6;
>> vector1(1) = setfield(vector1(1),'edad', 25);
```

cambia la nota final y la edad de la primera estructura del vector vector1. Con la función rmfield se puede borrar un campo de las estructuras de un *array*:

```
>> rmfield(vector1,'dni') %borra el campo dni del vector vector1
```

Se puede recuperar todos los valores de un campo de las estructuras de un *array* utilizando el operador punto. Si el campo es de tipo cadena, los valores hay que almacenarlos en un *array* de celdas; si el campo es de tipo numérico, los valores también se pueden guardar en un *array*.

```
>> d = {vector2.dni} %se guardan los dnis en un array de celdas
d =
    '12345678P'    '22222222P'    '11111111P'
>> notas = [vector2.notas] %se guardan las notas en un array numerico
notas =
    8.4000    7.5000    8.0000
```

Una operación muy común es ordenar los valores de un vector de estructuras por el valor de un campo. Para lograr esto primero vamos a describir dos características de la función sort que aún no hemos descrito. En primer lugar, si invocamos a sort con dos parámetros de salida, sort devuelve como segundo parámetro un vector con los índices del vector original ordenados. Por ejemplo:

```
>> x = [200 500 100 400];
>> [o ind] = sort(x)
o =
   100   200   400   500
ind =
     3     1     4     2
```

El vector ind nos indica que x(ind(1)) es el menor elemento de x, x(ind(2)) es el segundo menor elemento de x y así sucesivamente. Luego:

```
>> x(ind) %muestra el vector original ordenado
ans =
   100   200   400   500
```

Visto esto, se puede ordenar un vector de estructuras por un campo, seleccionándolo y obteniendo un vector con los índices ordenados para ese campo:

```
>> nom = {vector3.nombre}
nom =
    [1x21 char]    'Miguel Cano Salas'    'Eva Solana Chica'
>> [o ind] = sort(nom)
o =
    'Eva Solana Chica'    [1x21 char]    'Miguel Cano Salas'
ind =
     3     1     2
>> ordenado = vector3(ind);
```

```
>> ordenado(1)
ans =
        nombre: 'Eva Solana Chica'
           dni: '11111111P'
         edad: 23
     notaFinal: 8
>> ordenado(2)
ans =
        nombre: 'Juan Viedma Contreras'
           dni: '12345678P'
         edad: 23
     notaFinal: 8.4000
```

Con estas órdenes se ha ordenado el vector vector3 por el campo nombre. Supongamos ahora que queremos ordenar el vector de alumnos por nota, pero de mayor nota a menor nota. La función sort ordena en orden creciente por defecto, pero puede ordenar en orden decreciente:

```
>> x = [200 500 100 400];
>> [o ind] = sort(x,'descend') %ordena en orden decreciente
o =
   500   400   200   100
ind =
     2     4     1     3
```

Luego, el siguiente código:

```
>> x = [vector3.notaFinal];
>> [o ind] = sort(x, 'descend');
>> vord = vector3(ind);
```

ordena vector3 por orden decreciente de nota, almacenando el resultado en el vector vord. En realidad, a veces no hace falta almacenar los vectores ordenados, simplemente podemos trabajar con los vectores de índices. Por ejemplo, el siguiente guión:

```
clc
[o indNombre] = sort({vector3.nombre});
[o indNota] = sort([vector3.notaFinal], 'descend');
disp('Ordenados por nombre:')
for z = 1:length(indNombre)
    al = vector3(indNombre(z));
    fprintf('\t%s(%s): %.2f\n',al.nombre,al.dni,al.notaFinal)
end
fprintf('\nOrdenados por nota:\n')
for z = 1:length(indNota)
```

```
    al = vector3(indNota(z));
    fprintf('\t%s(%s): %.2f\n',al.nombre,al.dni,al.notaFinal)
end
```

muestra a los alumnos ordenados por nombre y después por nota—en orden decreciente—, trabajando con los vectores de índices. Si se quiere ordenar por más de un campo se puede utilizar la función sortrows, que ordena filas de una matriz. Por ejemplo:

```
[tmp ind] = sortrows([{vector3.edad}' {vector3.nombre}'], [-1 2]);
ordenado = vector3(ind);
```

genera el vector ordenado, con las estructuras en orden decreciente por edad y creciente por nombre. Consulta la ayuda de sortrows para más detalles.

8.5. *Arrays* de estructuras y archivos

Normalmente los *arrays* de estructuras contienen datos que deseamos preservar. Si no queremos exportar los datos a otros programas lo más sencillo es guardarlos en archivos MAT usando save—Sección 1.2. Por ejemplo:

```
>> save('misDatos.mat', 'vector3')
>> clear vector3
>> who vector3
>> load('misDatos.mat')
>> who vector3
Your variables are:
vector3
```

Si queremos exportar un vector de estructuras para que pueda ser utilizado por otro programa lo normal es guardar los datos en un archivo de texto, separando los campos con un carácter. El carácter separador no debería aparecer en ningún campo de ninguna estructura. Para guardar los datos en el archivo se puede trabajar con las operaciones a bajo nivel con archivos estudiadas en la Sección 6.8. Como ejemplo, el siguiente guión guarda el vector vector3 en un archivo de texto separando los campos con puntos y comas.

```
[idA mensaje] = fopen('salida.txt', 'wt');
if idA == -1
    fprintf('No se pudo abrir salida.txt: %s\n', mensaje)
    return
end
for ind = 1:length(vector3)
    fprintf(idA,'%s;%s;%d;%.2f\n', vector3(ind).nombre, ...
        vector3(ind).dni, vector3(ind).edad, vector3(ind).notaFinal);
```

```
end
fclose(idA);
```

Veamos el contenido del archivo creado:

```
>> type salida.txt
Juan Viedma Contreras;12345678P;23;8.40
Miguel Cano Salas;22222222P;24;7.50
Eva Solana Chica;11111111P;23;8.00
```

A continuación ilustramos la importación de un archivo de texto con campos delimitados por un carácter. Utilizamos como ejemplo al archivo *salida.txt* creado previamente. El siguiente guión usa la función textscan para leer los datos en un vector de estructuras.

```
[idA mensaje] = fopen('salida.txt', 'rt');
if idA == -1
    fprintf('No se pudo abrir salida.txt: %s\n', mensaje)
    return
end
celdas = textscan(idA,'%[^;];%[^;];%d;%f');
datos = struct('nombre',celdas{1}','dni',celdas{2}','edad', ...
        num2cell(celdas{3})','notaFinal',num2cell(celdas{4})');
fclose(idA);
```

El código es un poco complejo y no lo vamos a explicar aquí, aunque el lector interesado puede consultar la ayuda de MATLAB para comprender qué hace cada línea. El resultado es que la variable datos contiene un vector de estructuras con los datos del archivo de texto. Otra alternativa sería utilizar las funciones estudiadas en el Tema 6 para ir leyendo línea a línea el archivo y extraer los campos de las líneas—básicamente esto es lo que hace la función textscan.

8.6. Ejercicios

1. Crea un vector de 20 estructuras de tipo punto. Cada punto tiene una coordenada x y una coordenada y. Los valores de x e y pertenecen al rango $[0, 5]$. Genera los puntos aleatoriamente. Dibuja un gráfico con los puntos, destacando con arcos y con un color distinto los 5 puntos más alejados del origen—como ejemplo, observa la Figura 8.2.

2. Una librería mantiene información de sus libros, clientes y ventas. Por cada libro almacena un código, su título y su precio. Por cada cliente almacena un código y su nombre. Por cada venta almacena el código del libro y el código del cliente implicados en la venta—en una venta un cliente compra un único libro. Se utiliza un vector de estructuras para almacenar el conjunto de libros, otro vector de estructuras para alma-

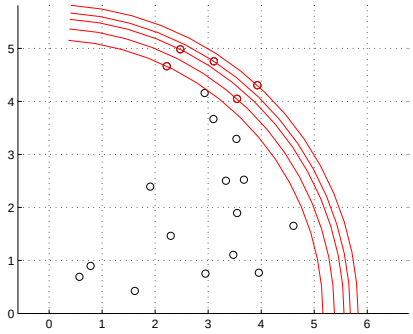

Figura 8.2: Puntos aleatorios, destacando los 5 más alejados del origen.

cenar a los clientes y un vector de estructuras más para almacenar las ventas. Dadas estas estructuras de datos realiza un programa que permita:

a) Recuperar los datos existentes en un archivo mat.

b) Guardar los datos existentes en un archivo mat.

c) Insertar un libro, cliente o venta en el vector correspondiente.

d) Visualizar el contenido de los vectores de estructuras—puedes utilizar la función more para paginar la visualización de los datos.

e) Dado un código de libro obtener cuántos libros con ese código se han vendido.

f) Obtener el título del libro más vendido.

g) Obtener el título del libro que más dinero ha recaudado.

h) Dado un código de cliente obtener los títulos de los libros que ha comprado.

Utiliza funciones para obtener un código estructurado.

Tema 9
Gráficos avanzados

En este tema vamos a seguir estudiando las facilidades que proporciona Matlab para visualizar información gráficamente, haciendo hincapié en el soporte a la visualización en tres dimensiones. Los gráficos tridimensionales son especialmente complejos, por lo que en este tema se analiza muy someramente muchos aspectos como la visualización de objetos translúcidos o el uso de luz y colores.

9.1. La función plot3

La función plot3 es el equivalente a la función plot en 3D. El guión de la Figura 9.1 dibuja un cuadrado en el plano $Z = 0$ y una recta de extremos $(0{,}5, 0{,}5, -0{,}5)$ y $(0{,}5, 0{,}5, 0{,}5)$ que atraviesa el cuadrado perpendicularmente por su centro. Observa que plot3 toma como tercer parámetro la coordenada z de los puntos. En el guión también se utiliza la función zlabel para etiquetar el eje z. El resultado de ejecutar el guión es la Figura 9.2. Cuando se visualiza una figura tridimensional es interesante poder observarla desde distintos puntos de vista. Esto se puede consiguir con la opción **Tools->Rotate 3D** del menú de la ventana que contiene la figura o seleccionando el botón adecuado de la barra de herramientas.

Como ejercicio intenta dibujar las aristas de un cubo. Veamos ahora otro ejemplo, el siguiente guión dibuja las espirales de la Figura 9.3:

```
t = 0:pi/50:10*pi;
st = sin(t);
ct = cos(t);
subplot(1,2,1)
plot3(ct,st,t)
subplot(1,2,2)
plot3(ct.*t,st.*t,t)
```

Sustituye en el guión anterior plot3 por comet3 y verás una versión animada del gráfico.

```
x = [0 1 1 0 0];
y = [0 0 1 1 0];
z = [0 0 0 0 0];
plot3(x,y,z,'r') %dibuja cuadrado
hold('on')
x = [0.5 0.5];
y = [0.5 0.5];
z = [-0.5 0.5];
plot3(x,y,z,'r') %dibuja recta
xlabel('eje x')
ylabel('eje y')
zlabel('eje z')
grid('on')
```

Figura 9.1: Guión que dibuja un cuadrado y una recta que lo atraviesa

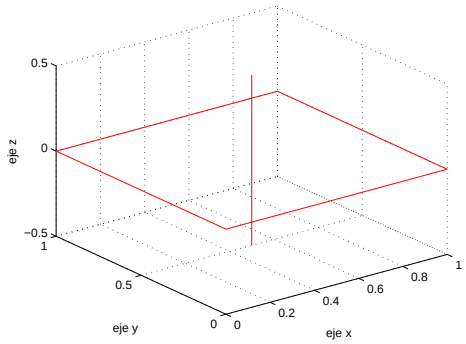

Figura 9.2: Gráfico generado por el guión 9.1 usando la función plot3.

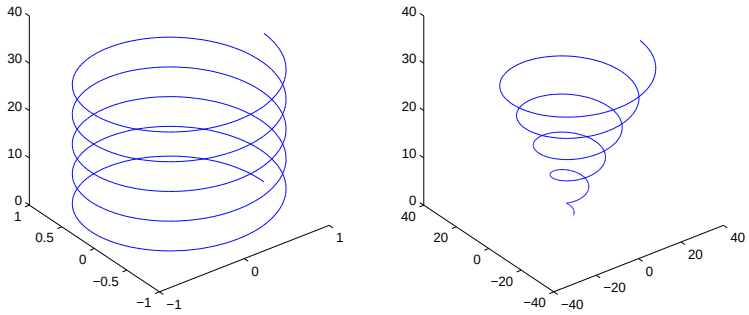

Figura 9.3: Espirales generadas utilizando plot3.

9.2. Distribuciones de frecuencias: bar3, bar3h, pie3, stem3 e hist3

En esta sección vamos a estudiar muy brevemente funciones que permiten visualizar la distribución de frecuencias de una muestra o una población. Las funciones estudiadas son muy parecidas a las vistas en la Sección 4.5. bar3 genera un diagrama de barras tridimensional:

```
>> v = [5 10 6 3];              % vector de frecuencias
>> subplot(1,2,1)
>> bar3(v)
>> m = [5 10 6 3; 1 2 10 8]; % matriz de frecuencias
>> subplot(1,2,2)
>> bar3(m)
```

este código genera los gráficos de la Figura 9.4. Observa que cuando se utiliza una matriz como parámetro, se emplean distintos colores para representar cada columna. La función bar3h representa las barras horizontalmente. pie3 muestra un diagrama de sectores tridimensional:

```
>> v = [5 10 6 3]; % vector de frecuencias
>> subplot(1,2,1)
>> pie3(v)
>> subplot(1,2,2)
>> pie3(v, {'suspenso','aprobado','notable','sobresaliente'})
```

Con este código se obtiene la Figura 9.5. El tamaño de un sector es proporcional al porcentaje de ocurrencias de la categoría que representa. Los sectores se etiquetan por defecto

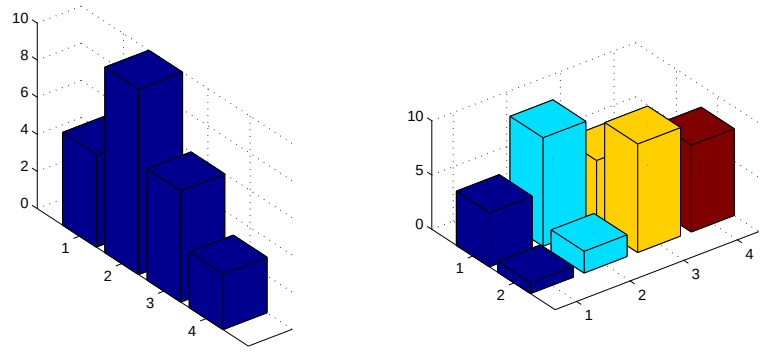

Figura 9.4: Ejemplo de uso de bar3.

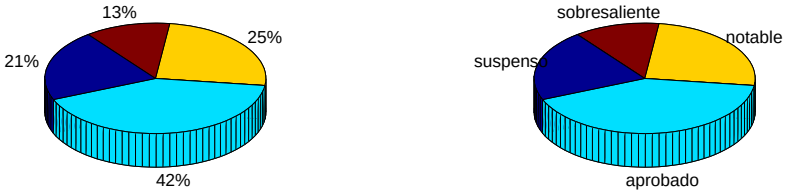

Figura 9.5: Ejemplo de uso de pie3.

con porcentajes de ocurrencia, pero también se pueden especificar etiquetas. Se puede destacar ciertos sectores extrayéndolos del diagrama, para ello se utiliza un vector numérico en el que los sectores a extraer almacenan valores no nulos.

```
>> v = [5 10 6 3];      % vector de frecuencias
>> pie3(v, [1 0 0 0]) % extrae el sector asociado a la frecuencia 5
```

La función stem3 es similar a bar3 pero utiliza ramas y hojas en lugar de barras para representar los datos. Por ejemplo:

```
>> m = [5 10 6 3; 1 2 10 8]; % matriz de frecuencias
>> stem3(m)
```

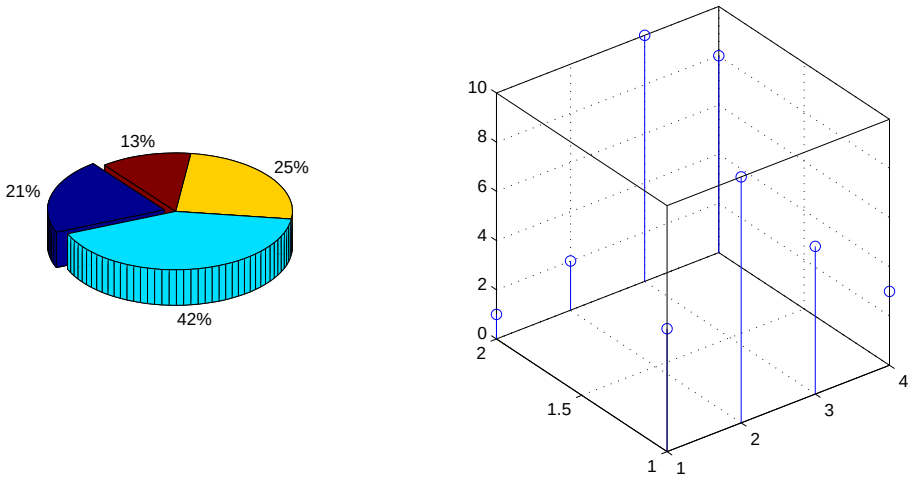

Figura 9.6: Ejemplo de sector destacado y de stem3.

La Figura 9.6 ilustra el resultado de ejecutar los dos últimos fragmentos de código. La función hist3 es algo más compleja, puedes consultar su funcionamiento usando help.

9.3. Superficies de malla de puntos

Las funciones mesh y surf permiten visualizar de una forma sencilla la superficie tridimensional asociada a una malla cuadrangular de puntos 3D, lo que permite, por ejemplo, visualizar la orografía de un terreno. Es común generar la malla de puntos empezando con una malla rectangular ubicada en el plano XY—la malla es bidimensional— utilizando la función meshgrid:

```
>> [x y]=meshgrid(-3:3,-3:3)
x =
     -3     -2     -1      0      1      2      3
     -3     -2     -1      0      1      2      3
     -3     -2     -1      0      1      2      3
     -3     -2     -1      0      1      2      3
     -3     -2     -1      0      1      2      3
     -3     -2     -1      0      1      2      3
     -3     -2     -1      0      1      2      3
y =
     -3     -3     -3     -3     -3     -3     -3
     -2     -2     -2     -2     -2     -2     -2
     -1     -1     -1     -1     -1     -1     -1
```

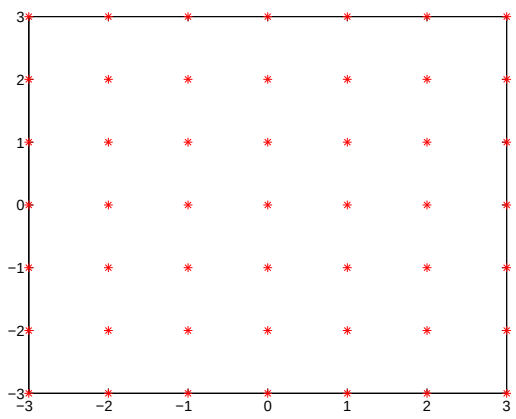

Figura 9.7: Malla cuadrangular 2D generada con meshgrid.

```
     0     0     0     0     0     0     0
     1     1     1     1     1     1     1
     2     2     2     2     2     2     2
     3     3     3     3     3     3     3
>> plot(x,y,'r*')
```

Esto ha generado una malla 7x7—length (-3:3) xlength (-3:3) —de puntos. El primer punto de la malla tiene coordenadas $(x(1,1), y(1,1))$, el segundo $(x(2,1), y(2,1))$ y así sucesivamente. La función plot visualiza los puntos de la malla—Figura 9.7. Si se quiere ver una malla tridimensional hay que especificar un valor constante de coordenada z, por ejemplo 0—gráfico izquierdo de la Figura 9.8:

```
>> plot3(x,y,zeros(size(x)),'r*')
```

Las submatrices x (1:2,1:2) e y (1:2,1:2) determinan un cuadrado de la malla y así sucesivamente. Por ejemplo, vamos a destacar dos cuadrados de la malla—gráfico derecho de la Figura 9.8:

```
>> hold on
>> plot3(x(1:2,1:2),y(1:2,1:2), zeros(2,2),'b*')
>> plot3(x(1:2,4:5),y(1:2,4:5), zeros(2,2),'b*')
```

Para generar una malla tridimensional debemos dotar de altura—coordenada z—a los puntos de la malla. Por ejemplo:

```
>> z = x.^2 + y.^2
z =
```

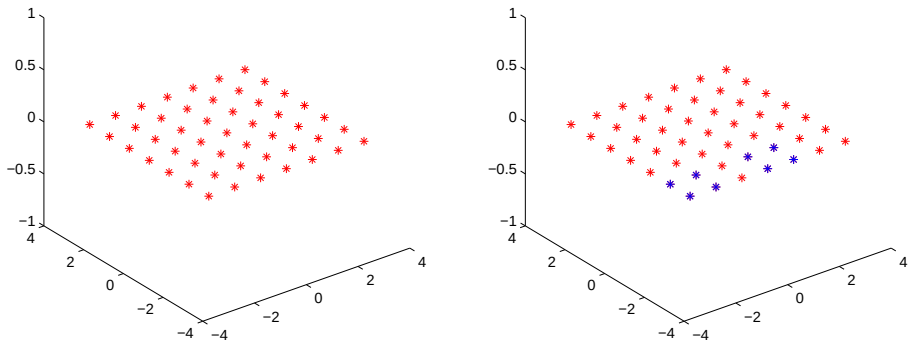

Figura 9.8: Malla cuadrangular 2D visualizada en 3D (plano $Z = 0$)

18	13	10	9	10	13	18
13	8	5	4	5	8	13
10	5	2	1	2	5	10
9	4	1	0	1	4	9
10	5	2	1	2	5	10
13	8	5	4	5	8	13
18	13	10	9	10	13	18

Ahora podemos utilizar la función mesh para visualizar los cuadrados de la malla—gráfico izquierdo de la Figura 9.9:

```
>> hold off
>> mesh(x,y,z)
>> colorbar
```

La función mesh visualiza en blanco los cuadrados de la malla, coloreando las aristas. El color de las aristas depende de su coordenada z. La función colorbar muestra una barra con los colores asociados a las distintas alturas. La función surf es parecida a mesh, pero colorea los cuadrados. Por ejemplo, la siguiente orden produce el gráfico derecho de la Figura 9.9:

```
>> surf(x,y,z)
```

Como curiosidad, el siguiente guión genera los gráficos de la Figura 9.10:

```
[x y] = meshgrid(-5:0.5:5);
z = sqrt(x.^2+y.^2) + eps;
z = sin(z) ./ z;
```

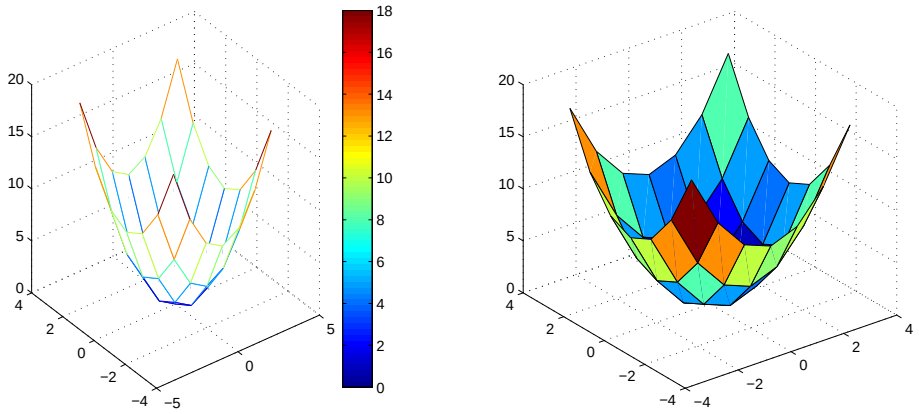

Figura 9.9: Ejemplo de uso de las funciones mesh y surf

```
subplot(1,2,1)
surf(x,y,z)
[x y z] = sphere(20);
subplot(1,2,2)
axis('equal')
surf(x,y,z)
```

Para generar la esfera se utiliza la función sphere(n) que genera una malla cuadrangular de tamaño $(n+1)x(n+1)$.

Las funciones mesh y surf tienen muchas posibilidades. Se sugiere el uso de help para explorar las distintas opciones. A continuación animamos a probar lo siguiente:

- Escribe la orden hidden off tras mesh. Las caras—cuadrados—de la malla serán transparentes.
- Escribe mesh(z), en lugar de mesh(x,y,z). Observa que en los ejes X e Y se muestran los índices de la matriz z. Prueba mesh(rand(7)), para obtener una visualización curiosa de los elementos de una matriz.
- Prueba [x y z] = cylinder(5), mesh(x,y,z).
- Incluye como última instrucción del guión que dibuja la esfera la línea: colormap ... copper. Observa cómo se utiliza otro mapa de colores para representar la altura de los datos.
- Genera una malla de mayor resolución—por ejemplo: [x y] = meshgrid(-3:0.5:3). Regenera los valores de la matriz z y visualiza la malla con mesh o surf. Observa cómo se mejora la calidad de la representación, especialmente en las zonas curvas.

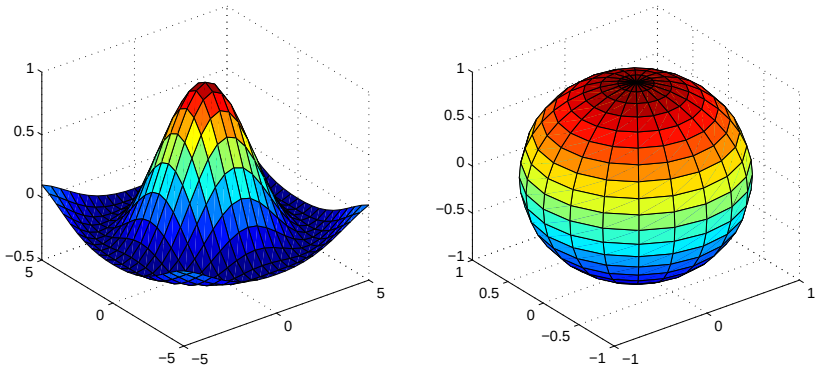

Figura 9.10: Sombrero mejicano y esfera generadas con surf

9.3.1. Curvas de nivel

MATLAB permite visualizar curvas de nivel. Una *curva de nivel* es una línea que une los puntos que contienen el mismo valor. En el caso de MATLAB las curvas de nivel indican puntos con la misma altura. Para ilustrar la visualización de curvas de nivel vamos a utilizar la función peaks, que está pensada para presentar ejemplos del uso de mesh. La función contour muestra las curvas de nivel asociadas a una matriz:

```
z = peaks;
subplot(1,2,1)
mesh(z)
subplot(1,2,2)
contour(z)
```

El resultado de ejecutar el guión se muestra en la Figura 9.11. La función contour permite especificar el número de curvas de nivel o los niveles en los que situar las curvas. La función contour3 permite ver las curvas de nivel en 3D y meshc dibuja las curvas de nivel bajo la superficie. Por ejemplo, el siguiente guión

```
z = peaks;
subplot(1,2,1)
meshc(z)
subplot(1,2,2)
contour3(z)
```

produce los gráficos de la Figura 9.12. La función surfc también permite visualizar las curvas de nivel bajo la superficie—prueba surfc(z).

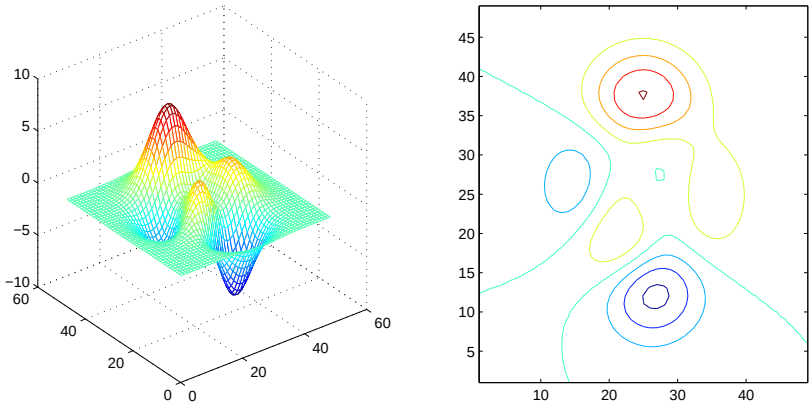

Figura 9.11: Ejemplo de uso de contour

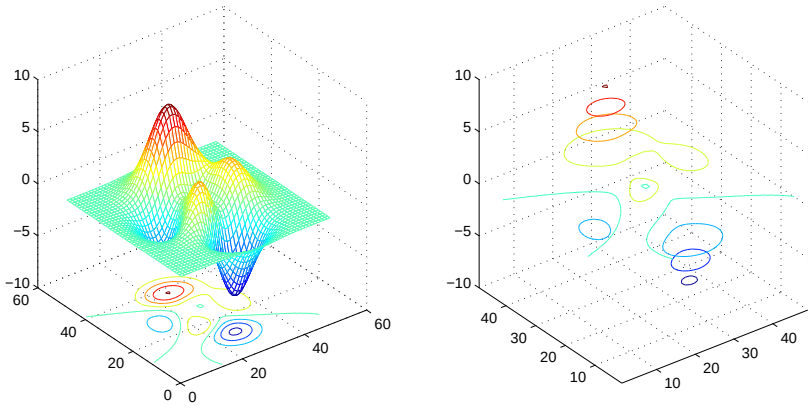

Figura 9.12: Ejemplo de uso de meshc y contour3

9.4. Mallas de polígonos: la función patch

La función patch permite visualizar un objeto bidimensional o tridimensional mediante una serie de polígonos 2D llamados parches. Se puede utilizar patch para dibujar un polígono 2D; así, el guión:

```
x = [1 3 2];   %coordenadas x de los vertices
y = [0 0 2];   %coordenadas y de los vertices
patch(x,y,'green')
```

dibuja un triángulo de vértices $(1,0)$, $(3,0)$ y $(2,2)$—Figura 9.13. La función patch es muy versátil. En el siguiente ejemplo vamos a utilizar patch para dibujar un cubo centrado en el origen. Para visualizar un objeto tridimensional usando patch hay que especificar los polígonos 2D que constituyen su superficie. En la Figura 9.14 se muestra un guión que dibuja el cubo; el resultado de ejecutar el guión se observa en la Figura 9.15. Para especificar la malla de polígonos—o parches—se utiliza una estructura con dos campos. El campo vertices contiene los vértices del objeto; éstos se especifican mediante una matriz de tres columnas, cada fila incluye las coordenadas x, y y z de un vértice. El campo faces indica las caras—polígonos o parches—del objeto tridimensional. faces es una matriz en la que cada fila indica una cara. Las caras se especifican indicando los índices de sus vértices asociados en la matriz vertices. Rota la figura para poder observarla desde distintos puntos de vista. El color de las caras y aristas de la malla es negro por defecto, en el guión se ha especificado que las caras se dibujen de color gris. Con estos parámetros de visualización es clave que las aristas tengan un color distinto a las caras para obtener un efecto tridimensional. Prueba a dibujar las aristas de gris y observa el resultado—cambia la invocación a patch en el guión por la siguiente:

```
patch(parche,'FaceColor',[0.5 0.5 0.5],'EdgeColor',[0.5 0.5 0.5])
```

patch también permite especificar el color de cada cara o, incluso, permite especificar una imagen que se "pega" a la cara; consulta un manual si estás interesado en saber cómo se hace.

Añade como última línea del guión de la Figura 9.14 el código: alpha(0.8). El efecto es que las caras se dibujan con cierto grado de transparencia, lo que permite observar el interior del cubo. En la función alpha el valor 1 significa opaco y 0 transparente, los valores intermedios indican grados de transparencia. alpha también se puede aplicar tras usar mesh o surf.

9.5. Propiedades de los objetos gráficos

Una figura en MATLAB consta de varios objetos gráficos como pueden ser líneas, texto, parches, los ejes de coordenadas o la propia figura. Los objetos gráficos se organizan en

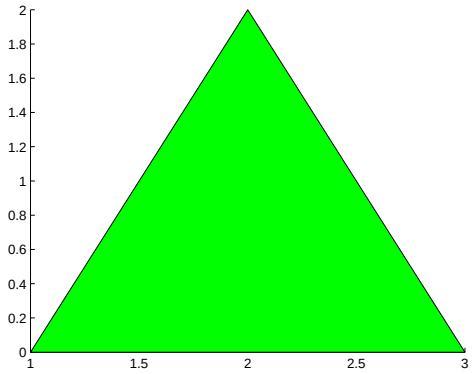

Figura 9.13: Ejemplo de uso de patch para visualizar un polígono 2D

```
parche.vertices = [
    -1 -1 -1
    1 -1 -1
    1 1 -1
    -1 1 -1
    -1 -1 1
    1 -1 1
    1 1 1
    -1 1 1];
parche.faces = [
    1 2 3 4
    4 3 7 8
    1 2 6 5
    3 2 6 7
    1 5 8 4
    5 6 7 8];
patch(parche,'FaceColor', [0.5 0.5 0.5])
axis([-2 2 -2 2 -2 2])
```

Figura 9.14: Guión que usa patch para generar el cubo de la Figura 9.15

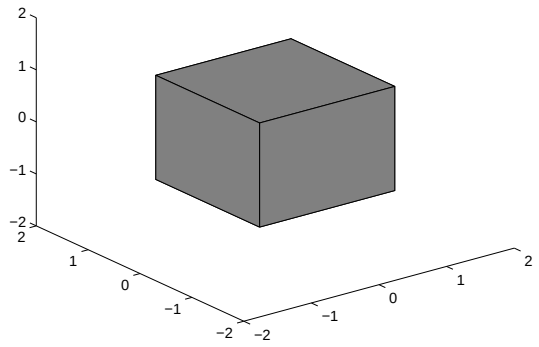

Figura 9.15: Ejemplo de uso de patch para visualizar un cubo

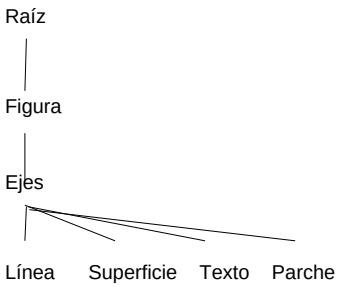

Figura 9.16: Jerarquía de objetos gráficos

una estructura jerárquica en que los hijos heredan propiedades de los padres—Figura 9.16. Cada objeto gráfico tiene una serie de propiedades que indican cómo se visualiza. Estas propiedades pueden ser consultadas y modificadas utilizando funciones o desde la ventana de visualización de la figura, eligiendo las opciones apropiadas del menú.

Para trabajar con las propiedades de un objeto gráfico utilizando código hay que obtener el *gestor del objeto gráfico—handle graphics*. El gestor es un número real que se utiliza para identificar el objeto gráfico con que se quiere trabajar. Las funciones que crean un objeto gráfico devuelven su gestor, por ejemplo:

```
>> x = 0:0.5:2*pi;
>> g = plot(x,sin(x))
g =
  174.0023
```

con las órdenes previas se ha creado un gráfico, cuyo gestor se ha almacenado en la variable g. El gestor puede utilizarse para consultar y modificar las propiedades de su objeto gráfico asociado. La función get permite consultar las propiedades de un objeto gráfico:

```
>> get(g)
            DisplayName: ''
             Annotation: [1x1 hg.Annotation]
                  Color: [0 0 1]
              LineStyle: '-'
              LineWidth: 0.5000
                 Marker: 'none'
             MarkerSize: 6
        MarkerEdgeColor: 'auto'
        MarkerFaceColor: 'none'
                  XData: [1x63 double]
                  YData: [1x63 double]
                  ZData: [1x0 double]
           BeingDeleted: 'off'
         ButtonDownFcn: []
               Children: [0x1 double]
               Clipping: 'on'
              CreateFcn: []
              DeleteFcn: []
             BusyAction: 'queue'
       HandleVisibility: 'on'
                HitTest: 'on'
          Interruptible: 'on'
               Selected: 'off'
    SelectionHighlight: 'on'
                    Tag: ''
                   Type: 'line'
          UIContextMenu: []
               UserData: []
                Visible: 'on'
                 Parent: 173.0018
            XDataMode: 'manual'
          XDataSource: ''
          YDataSource: ''
          ZDataSource: ''
```

Si se asigna el resultado devuelto por get a una variable, se crea una estructura cuyos campos se nombran con los nombres de las propiedades:

```
>> propPlot = get(g);
>> propPlot.Color
```

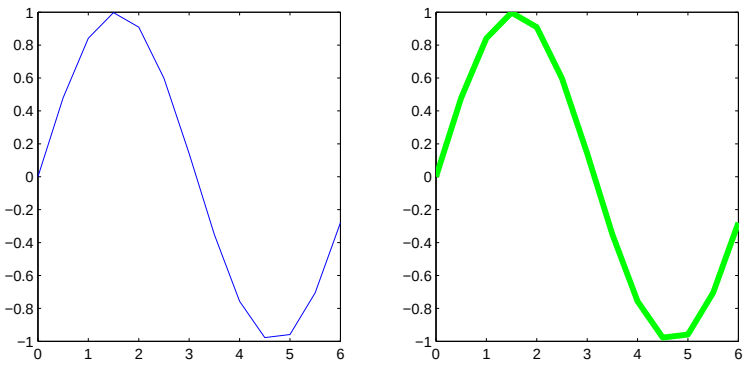

Figura 9.17: Un gráfico antes y después de cambiar sus propiedades de color y grosor

```
ans =
     0     0     1
>> get(g,'Color')
ans =
     0     0     1
```

El código previo ilustra cómo consultar una propiedad individual de un objeto gráfico a partir de la estructura o mediante la función get. A continuación vemos cómo se puede cambiar una propiedad mediante la función set:

```
>> set(g,'LineWidth',4)
>> set(g,'Color', [0 1 0])
```

Observa la función seno, ahora la línea es más gruesa y de color verde—Figura 9.17. También se puede especificar propiedades al crear un objeto gráfico:

```
>> g2 = plot(x,cos(x),'LineWidth',2,'Marker','*','MarkerSize',6);
```

text, xlabel, patch, bar, mesh, figure y demás funciones que generan un objeto gráfico devuelven sus gestores asociados.

Las funciones gcf, gca y gco devuelven el gestor de la figura, eje y objeto actual respectivamente. El objeto gráfico actual es el último dibujado o seleccionado mediante el ratón.

9.6. Color, iluminación y cámara

La función view permite modificar el lugar desde donde se observa una escena—a veces se habla de la posición de la cámara. view puede invocarse con la siguiente sintaxis: view(az,el),

donde az es el acimut—*azimuth*—o ángulo polar—en grados—en el plano XY. az vale por defecto $-37{,}5^0$ y cambiando su valor se puede rotar sobre el eje Z para observar la escena desde distintos puntos de vista. el es la elevación vertical de la cámara u observador. Se mide en grados e indica el ángulo que la línea de visión forma con el plano XY, por defecto vale 30^0, cuando vale 90^0 la escena se observa desde arriba. Un valor negativo de el indica que el punto de observación está situado debajo del objeto.

Ejecuta el siguiente guión. Se trata de una animación que parte de una elevación de 30^0, la elevación se incrementa de 5 en 5 grados hasta que se llega a los 90^0. El efecto animado se obtiene al dibujar el objeto repetidamente cambiando la elevación para ver al objeto desde distintas posiciones. Se hacen pausas de medio segundo entre visualización utilizando la función pause.

```
[x y] = meshgrid(-3:3);
z = x.^2 + y.^2;
az = -37.5;
for el = 15:5:90
    surf(x,y,z)
    view(az,el)
    pause(0.5)
end
```

A continuación vamos a experimentar con el cambio del acimut, lo que nos permite rotar sobre el objeto desde arriba. La siguiente animación parte del acimut por defecto—$-37{,}5^0$—y va incrementando el acimut de 15 en 15 grados, lo que implica una rotación en sentido contrario a las agujas del reloj. Un incremento negativo rota en sentido contrario.

```
[x y] = meshgrid(-3:3);
z = x.^2 + y.^2;
el = 30;
for az = -37.5:15:-37.5+360
    surf(x,y,z)
    view(az,el)
    pause(0.5)
end
```

Cuando se dibujan objetos tridimensionales se puede obtener un mayor grado de realismo si se utilizan técnicas de iluminación. Aunque el estudio de estas técnicas excede las pretensiones de estos apuntes, se ha incluido el siguiente guión para que el lector pueda ver la técnica en funcionamiento:

```
[x y z] = sphere(20);
axis('equal')
surf(x,y,z)
```

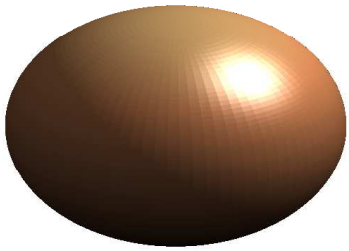

Figura 9.18: Esfera iluminada

```
shading interp
colormap copper
axis off
lightangle(60,45)
```

En el guión se utiliza la función lightangle para establecer una fuente de luz con un acimut de 60^0 y una elevación de 45^0. La Figura 9.18 ilustra el resultado de ejecutar el guión, aunque se ha cambiado la invocación a sphere usando el parámetro 80 para obtener un resultado más realista.

9.7. Animación

En la sección anterior se ha visto un ejemplo de animación, es decir, de un gráfico en movimiento. La técnica empleada es similar a la utilizada para crear una película. Consiste en generar varios gráficos similares a una velocidad adecuada. El cerebro humano reacciona ante esta visión mezclando las imágenes y produciendo la ilusión de movimiento. Cuando los gráficos son sencillos se puede utilizar guiones como los de la sección previa. Si los gráficos son muy complejos puede que no dé tiempo a generarlos tan rápidamente como es preciso para obtener un efecto adecuado. En ese caso se pueden utilizar las funciones getframe y movie. Con getframe se generan gráficos fuera de línea y se almacenan en una matriz. Posteriormente, movie permite visualizar los gráficos almacenados en la matriz.

A continuación vamos a presentar otro ejemplo de animación. Se trata de dibujar en 2D un planeta que es orbitado por un satélite mediante una órbita circular. La Figura 9.19 presenta el código. La función principal dirige la animación. La variable ang almacena el ángulo

```
1  function sistemaSolar
2      for ang = 0:10:360
3          dibujaCircunferencia(5,0,0,0)
4          hold on
5          dibujaCircunferencia(1,9,0,ang)
6          axis([-11 11 -11 11]);
7          hold off
8          pause(0.5)
9      end
10 end
11
12 function dibujaCircunferencia(radio,centrox,centroy,angulo)
13     a = 0:0.1:2*pi;
14     x = cos(a)*radio + centrox;
15     y = sin(a)*radio + centroy;
16     xrotado = x*cosd(angulo) - y*sind(angulo);
17     yrotado = x*sind(angulo) + y*cosd(angulo);
18     plot(xrotado,yrotado,'black-')
19 end
```

Figura 9.19: Función que anima un planeta orbitado por un satélite

de rotación del satélite en grados. ang va de 0^0 a 360^0 en intervalos de 10^0. Los cuerpos celestes se dibujan como circunferencias utilizando la función dibujaCircunferencia. Ésta toma como parámetros el radio de la circunferencia, las coordenadas de su centro cuando empieza la animación y el ángulo de rotación. Las líneas 13–15 calculan las coordenadas de la esfera cuando comienza la animación. Las líneas 16–17 rotan las coordenadas de la esfera. Ten en cuenta que para rotar un punto $p = (x, y)$ θ grados respecto al origen de coordenadas hay que aplicar los cálculos siguientes:

$$x_r = x\cos\theta - y\sin\theta$$
$$y_r = x\sin\theta + y\cos\theta$$

9.8. Ejercicios

1. Dibuja un tetraedro—pirámide de base triangular. La base estará formada por el triángulo de vértices: (-1,-1,0), (1,-1,0) y (0,1,0); y el ápice es el vértice de coordenadas (0,0,5). El gráfico obtenido debe ser similar al gráfico izquierdo de la Figura 9.20.

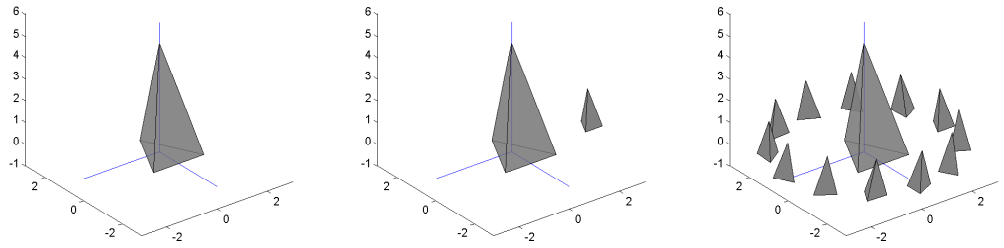

Figura 9.20: Tetraedros

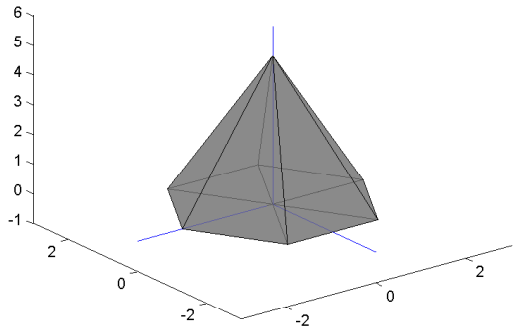

Figura 9.21: Pirámide de base hexagonal

2. Añádele al tetraedro una réplica de escala 1:$\frac{1}{3}$ y desplazado dos unidades en el eje X positivo—ver gráfico central de la Figura 9.20.

3. Añade varias réplicas rodeando el tetraedro grande—ver gráfico derecho de la Figura 9.20.

4. Escribe una función que dado un radio R, una altura A y un valor n genere los vértices y caras asociados a una pirámide de altura A y cuya base es un polígono regular de n lados circunscrito en una circunferencia de centro el origen y radio R. En la Figura 9.21 se puede observar una pirámide de base hexagonal.

5. La distribución de calor sobre una superficie rectangular viene dada por la función $u(x, y) = 80y^2e^{-x^2-0,3y^2}$. Dibuja una malla cuadrangular que refleje la distribución de calor en la superficie $-3 \leq x \leq 3$, $-6 \leq y \leq 6$.

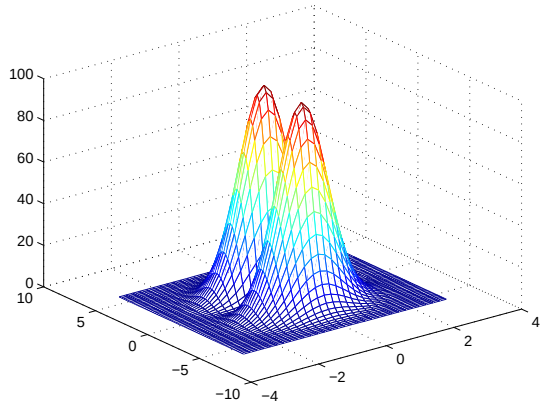

Figura 9.22: Distribución de calor sobre una superficie

6. Vamos a construir un gráfico para representar cómo se siente atraídos los puntos de una malla cuadrangular bidimensional por dos objetos. En primer lugar generaremos la malla con la sentencia: [x y] = meshgrid (0:.2:10,0:.2:10) . Los dos objetos que ejercen atracción están situados en las coordenadas $(8, 2)$ y $(2, 8)$ de la malla respectivamente. Cada punto de la malla se siente atraído por su objeto más cercano. El valor de atracción es la distancia euclídea al objeto. Calcula en una matriz z el valor de atracción de cada punto de la malla y utiliza mesh, meshc, surf o surfc para obtener una representación gráfica de la atracción. La Figura 9.23 representa los valores de atracción de los puntos de la malla utilizando surfc.

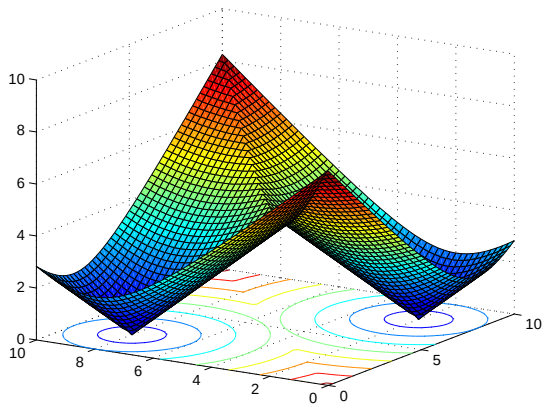

Figura 9.23: Valor de atracción de los puntos de la malla

Tema 10
Recursividad

Los lenguajes de programación modernos permiten las funciones recursivas. Una función recursiva es aquella que se invoca a sí misma. Existen problemas que por su naturaleza recurrente se solucionan de una forma elegante mediante una función recursiva.

10.1. Funciones recursivas

Como se ha indicado con anterioridad una *función recursiva* es aquella que se llama a sí misma. Un ejemplo clásico es el cálculo del factorial. El factorial de un entero no negativo n se define como el producto de todos los enteros positivos menores o iguales que n. Por ejemplo, $4! = 4 * 3 * 2 * 1 = 24$. El 0! vale 1. En MATLAB el factorial del número n se puede calcular trivialmente como prod(1:n) o utilizando la función interna factorial. El factorial de un número también se puede definir de una manera recurrente:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1)! & \text{si } n > 0 \end{cases}$$

La función recursiva fact de la Figura 10.1 calcula el factorial de un número aplicando de una forma natural la definición recurrente del factorial de un número.

```
1 function res = fact(n)
2     if n == 0
3         res = 1;
4     else
5         res = n * fact(n-1); %llamada recursiva
6     end
7 end
```

Figura 10.1: Cálculo del factorial mediante una función recursiva.

Una función recursiva consta de uno o varios casos base y general. Un *caso base* es un valor de los parámetros de la función para los que la función se soluciona sin realizar ninguna llamada recursiva. Por ejemplo, en el función de la Figura 10.1 existe un único caso base: cuando n vale cero; en ese caso el factorial se calcula como 1 sin necesidad de llamadas recursivas. Un *caso general o recursivo* es un valor de los parámetros de la función para los que la función utiliza una o varias llamadas recursivas para calcular su solución. La llamada recursiva de un caso general utiliza un parámetro "más pequeño", en el sentido de que genera un caso más próximo a un caso base. Por ejemplo, en la función de la Figura 10.1 cuando n almacena un valor mayor que cero se genera un caso general, la función fact calcula el factorial utilizando la sentencia `res = n * fact (n-1);`, que implica una llamada recursiva con un caso más pequeño: $n-1$. Toda función recursiva debe constar de al menos un caso base y un caso general. Si no existiera caso base la recursividad nunca terminaría, porque todos los casos serían generales e implicarían una llamada recursiva para su solución. Por otro lado, una función sin caso general no sería una función recursiva porque no utilizaría llamadas recursivas.

10.2. Ejemplos de funciones recursivas sencillas

En las siguientes subsecciones se describen varios ejemplos de problemas sencillos que se pueden resolver mediante funciones recursivas.

10.2.1. Suma de los dígitos de un entero

Suponga que debe calcular la suma de los dígitos de un entero no negativo n. Este problema se puede plantear de una manera recurrente:

$$sumadig(n) = \begin{cases} n & \text{si } n < 10 \\ resto(n, 10) + sumadig(cociente(n, 10)) & \text{si } n \geq 10 \end{cases}$$

Es decir, si n es menor que 10 la suma es el propio número y en otro caso la suma es el resto del número entre 10 más la suma de los dígitos del cociente de dividir n entre 10. Por ejemplo: $sumadig(428) = 8 + sumadig(42)$. Dada esta definición recurrente la función recursiva de la Figura 10.2 implementa esta estrategia para sumar los dígitos de un entero. Este problema también admite una solución iterativa simple, como la función de la Figura 10.3.

10.2.2. La sucesión de Fibonacci

La sucesión de Fibonacci está formada por la serie de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Por definición, los dos primeros números de la serie son 0 y 1 y los números siguientes se

```
function sol = sumadig(n)
    if n < 10
        sol = n;
    else
        sol = rem(n,10) + sumadig(floor(n/10));
    end
end
```

Figura 10.2: Suma los dígitos de un entero no negativo mediante una función recursiva.

```
function sol = sumadig2(n)
    sol = 0;
    while n ≥ 10
        sol = sol + rem(n,10);
        n = floor(n/10);
    end
    sol = sol + n;
end
```

Figura 10.3: Suma los dígitos de un entero no negativo mediante un cálculo iterativo.

calculan como la suma de los dos números previos de la serie. La serie puede expresarse de una forma elegante mediante la siguiente recurrencia:

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n > 1 \end{cases}$$

La función de la Figura 10.4 calcula un término de la sucesión de Fibonacci. Observa que para resolver el caso general se realizan dos llamadas recursivas; hasta que no se resuelva la llamada fibo(n-1) no se realizará la llamada fibo(n -2). Como ejercicio puedes realizar un programa que calcule de una manera iterativa los n primeros términos de la sucesión.

10.2.3. La búsqueda binaria

Otro ejemplo clásico de algoritmo recursivo es el de la búsqueda binaria en un vector ordenado. Se trata de buscar el índice que ocupa un dato en un vector ordenado. Si el dato se encuentra más de una vez se devuelve cualquiera de sus posiciones; si no se encuentra en el vector se devuelve un índice no válido como -1. En caso de que el vector no estuviera ordenado habría que utilizar una función como la de la Figura 10.5. Esta función precisa consultar, en el peor de los casos, todos los elementos del vector para encontrar la posición

```
function sol = fibo(n)
    if n < 2
        sol = n;
    else
        sol = fibo(n-1)+fibo(n-2);
    end
end
```

Figura 10.4: Calcula un término de la suceción de Fibonacci.

```
function indice = buscaSec(v, dato)
    for indice = 1:length(v)
        if v(indice) == dato
            return
        end
    end
    indice = -1;
end
```

Figura 10.5: Búsqueda secuencial de un elemento en un vector

del dato buscado. Si el vector está ordenado se puede disminuir el número de consultas promedio con una función como la de la Figura 10.6. Como el vector está ordenado en orden creciente y lo recorremos en ese orden, en cuanto se encuentra un elemento mayor que el dato buscado se puede detener la búsqueda porque sabemos que el resto de elementos del vector son mayores que el dato buscado.

Sin embargo, si el vector está ordenado se puede realizar una *búsqueda binaria* para encontrar el dato de un modo más eficiente. La estrategia utilizada en la búsqueda binaria es la siguiente. En primer lugar se compara el dato con el elemento situado en la posición central del vector. En caso de que coincidan la búsqueda termina y se devuelve la posición central del vector. Si no coinciden y el elemento central es mayor que el dato, entonces se busca en la mitad inferior del vector, en otro caso se busca en la mitad superior. Las posteriores búsquedas utilizan la misma estrategia. Se sigue buscando hasta que se encuentra el dato o se descubre que el dato no está en el vector. En cada iteración del algoritmo, en caso de que el dato no coincida con el elemento central se descartan la mitad de los elementos sobre los que se busca, esto implica que, en el peor de los casos, se hagan $\log_2 n$ consultas en el vector, donde n es su número de elementos. En contraste, la función de la Figura 10.6 realiza n consultas en el vector en el peor de los casos. La función de la Figura 10.7 muestra una implementación de la búsqueda binaria mediante un algoritmo recursivo. La búsqueda bi-

```
function indice = busca2(v, dato)
    indice = 1;
    while indice ≤ length(v) && v(indice) ≤ dato
        if v(indice) == dato
            return
        end
        indice = indice + 1;
    end
    indice = -1;
end
```

Figura 10.6: Búsqueda secuencial de un elemento en un vector ordenado

naria también se puede implementar mediante un algoritmo iterativo como el de la Figura 10.8.

10.3. Recursividad *versus* iteratividad

Todo algoritmo recursivo se puede resolver mediante un algoritmo iterativo, a veces apoyándose en una estructura de datos de tipo cola o de tipo pila. En la mayoría de los lenguajes de programación la solución iterativa es más eficiente que la recursiva, pues evita la sobrecarga asociada a una llamada a función. Sin embargo, para muchos problemas la solución recursiva es más elegante y fácil de comprender, por lo que se prefiere la solución recursiva pese a su menor eficiencia.

10.4. Algoritmos divide y vencerás: ordenación recursiva

La técnica algorítmica *divide y vencerás* consiste en dividir un problema en subproblemas más pequeños y fáciles de resolver. Si los subproblemas son de la misma naturaleza que el problema original la técnica divide y vencerás se puede implantar mediante una función recursiva.

Un ejemplo de algoritmo que sigue la técnica divide y vencerás es el algoritmo de *ordenación por mezcla—mergesort—*, que ordena los elementos de un vector. La idea del algoritmo es la siguiente. Dado un vector a ordenar el vector se divide en dos subvectores de, aproximadamente, el mismo tamaño que se ordenan recursivamente. Una vez ordenados los dos subvectores se mezclan para obtener el vector ordenado. El caso base consiste en un vector de tamaño uno que, trivialmente, está ordenado. La función de la Figura 10.9 muestra una implementación del algoritmo en MATLAB. La función auxiliar `mezcla` mezcla el contenido de dos vectores ordenados produciendo un vector ordenado. Este algoritmo tiene una com-

```
function indice = busquedaBinRec(v, dato)
    indice = busqueda(v, dato, 1, length(v));
end

function indice = busqueda(v, dato, ini, fin)
   if (ini > fin)
      indice = -1;
      return
   end
   medio = floor((ini + fin) / 2);
   if dato == v(medio)
      indice = medio;
   elseif dato < v(medio)
      indice = busqueda(v, dato, ini, medio-1);
   else
      indice = busqueda(v, dato, medio+1, fin);
   end
end
```

Figura 10.7: Búsqueda binaria recursiva en un vector ordenado

```
function indice = busquedaBinIte(v, dato)
    ini = 1;
    fin = length(v);
    while ini ≤ fin
        medio = floor((ini + fin) / 2);
        if dato == v(medio)
           indice = medio;
           return
        elseif dato < v(medio)
           fin = medio-1;
        else
           ini = medio+1;
        end
    end
    indice = -1;
end
```

Figura 10.8: Búsqueda binaria iterativa en un vector ordenado

plejidad algorítmica $O(n \log_2 n)$ frente a la complejidad $O(n^2)$ de otros algoritmos clásicos de ordenación como la burbuja, selección o inserción. El famoso algoritmo de ordenación *quicksort*, ideado por Tony Hoare, también se basa en la técnica divide y vencerás y presenta una implementación recursiva. Su complejidad también es $O(n \log_2 n)$, pero en la práctica se ejecuta más rápidamente que *mergesort*, siendo el algoritmo de ordenación más eficaz conocido; la función interna sort implementa este algoritmo.

10.5. Algoritmos de recorrido de árboles

Una estructura de datos jerárquica muy frecuente en informática es el *árbol*. La Figura 10.10 muestra un árbol, observa que éste se representa de manera invertida con la raíz arriba y las hojas abajo. Un árbol conecta una serie de *nodos*, las flechas representan gráficamente la conexión. La conexión es de parentesco; al nodo del que sale la flecha se le llama *padre* y el nodo al que llega la flecha se le llama *hijo*. Un nodo sólo puede tener un padre y puede tener varios hijos—o ninguno. Todos los nodos tienen un padre, salvo uno, que se llama *nodo raíz*. Los nodos que tienen hijos se llaman *nodos internos* y los que no tienen hijos *nodos hoja*.

Un árbol tiene una naturaleza recurrente por lo que se adapta a ser procesado mediante algoritmos recursivos. Se puede definir un árbol de forma recurrente definiendo un árbol sin nodos como un *árbol vacío*. Un árbol es un nodo raíz cuyos hijos son árboles—también llamados *subárboles* porque son más pequeños, pudiendo ser árboles vacíos.

Un ejemplo de estructura de árbol es un *sistema de archivos*. Cada unidad de memoria secundaria está dividida en uno o varios sistemas de archivos. Los nodos se corresponden con carpetas y archivos. Una carpeta de la que cuelgan archivos y/o carpetas es un nodo interno. Las carpetas vacías y los archivos son nodos hoja. Cada sistema de archivos tiene una carpeta raíz, por ejemplo en Windows C:/ es la carpeta raíz del sistema de archivos asociado a la partición de arranque del disco duro.

Se puede visitar todos los nodos asociados a un árbol mediante la siguiente función recursiva:

```
function recorrido(nodo)
    if nodo es hoja %caso base
        return
    end
    %caso general
    for cada hijo de nodo
       recorrido(hijo)
    end
end
```

La función recibe como parámetro el nodo raíz de un árbol. Si el nodo es hoja, devuelve el control; en otro caso recorre recursivamente los subárboles asociados a sus hijos. El tipo

```
function ord = mergeSort(v)
    if numel(v) ≤ 1 %es el caso base?
        ord = v;
        return
    end
    %caso general: numel(v) ≥ 2
    medio = ceil(numel(v)/2);
    izq = mergeSort(v(1:medio));
    der = mergeSort(v(medio+1:end));
    ord = mezcla(izq,der);
end

%mezcla los elementos de los vectores ordenados v1 y v2
function mezclado = mezcla(v1, v2)
    indV1 = 1;
    indV2 = 1;
    indM = 1;
    mezclado = zeros(1, numel(v1)+numel(v2));
    while indV1 ≤ numel(v1) && indV2 ≤ numel(v2)
        if v1(indV1) ≤ v2(indV2)
            mezclado(indM) = v1(indV1);
            indV1 = indV1 + 1;
        else
            mezclado(indM) = v2(indV2);
            indV2 = indV2 + 1;
        end
        indM = indM + 1;
    end
    if indV1 ≤ numel(v1)
        mezclado(indM:end) = v1(indV1:end);
    else
        mezclado(indM:end) = v2(indV2:end);
    end
end
```

Figura 10.9: Algoritmo de ordenación *mergesort*

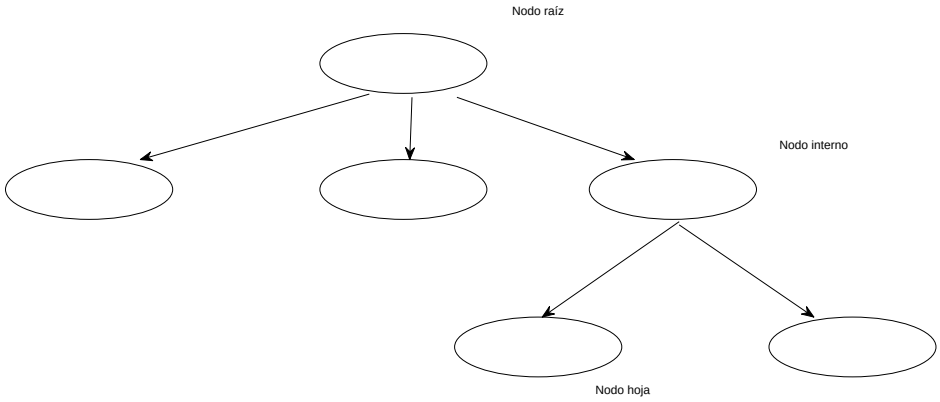

Figura 10.10: Estructura de árbol

de recorrido del árbol realizado por esta función se llama *primero en profundidad*—¿intuyes por qué ese nombre?.

La función de la Figura 10.11 realiza un recorrido primero en profundidad de un subárbol de un sistema de archivos, contando cuántos archivos contiene dicho subárbol. El parámetro de entrada de la función es una trayectoria de la carpeta o directorio que contiene el nodo raíz del subárbol. Si la trayectoria se corresponde con un archivo—líneas 2–5—estamos ante un caso base—nodo hoja—y se devuelve una cuenta de uno. En otro caso es un nodo interno—líneas 7–15—y hay que recorrer el contenido de la carpeta para recorrer recursivamente sus hijos—subárboles. Para obtener el contenido de la carpeta se utiliza la función dir—línea 7—, que devuelve un vector de estructuras, donde cada estructura almacena una entrada de la carpeta—consulta help dir para ver los campos de la estructura. En la línea 10 se obtiene el nombre de la entrada. La sentencia if—línea 11—evita usar las entradas . y .. que representan a la carpeta y a la carpeta padre respectivamente. La función fullfile concatena el nombre de la entrada con la trayectoria de la carpeta.

```
>> cuentaArchivos('.') %archivos en el directorio de trabajo
ans =
     10
>> cuentaArchivos('..') %archivos en el directorio padre
ans =
```

```
1  function cuenta = cuentaArchivos(tray)
2      if ¬isdir(tray)     %comprueba si es un archivo
3          cuenta = 1;
4          return
5      end
6      %es una carpeta
7      hijos = dir(tray); %obtiene información de los hijos
8      cuenta = 0;
9      for ind = 1:length(hijos)
10         nombre = hijos(ind).name;
11         if ¬strcmp(nombre,'.') && ¬strcmp(nombre,'..');
12             nombreCompleto = fullfile(tray,nombre);
13             cuenta = cuenta + cuentaArchivos(nombreCompleto);
14         end
15     end
16 end
```

Figura 10.11: Función que cuenta el número de archivos que cuelgan recursivamente de un directorio

```
     48
>> directorio = uigetdir
directorio =
C:\Users\Paco\Dropbox\ApuntesMatlab
>> cuentaArchivos(directorio)
ans =
    206
```

En el ejemplo la función uigetdir permite seleccionar un directorio mediante una ventana gráfica que permite navegar con el ratón por el sistema de archivos. uigetdir devuelve la trayectoria absoluta del directorio seleccionado.

10.6. Ejercicios

1. Realiza una función recursiva que sume los primeros n enteros positivos. Nota: para plantear la función recursiva ten en cuenta que la suma puede expresarse mediante la siguiente recurrencia:

$$suma(n) = \begin{cases} 1 & \text{si } n = 1 \\ n + suma(n-1) & \text{si } n > 1 \end{cases}$$

2. Escribe una función recursiva que calcule un número elevado a una potencia entera mayor o igual que cero: x^y. Expresa el cálculo mediante una recurrencia y después escribe la función recursiva.

3. Realiza una función recursiva que diga si una cadena de caracteres es un palíndromo. Un palíndromo es una frase o palabra que se lee igual de delante hacia atrás que de atrás hacia delante, por ejemplo: reconocer o anilina. Para simplificar supón que la cadena no contiene ni mayúsculas, ni signos de puntuación, ni espacios en blanco ni tildes.

4. En su libro *Elementos* el matemático griego Euclides describió un método para calcular el máximo común divisor de dos enteros. El método se puede expresar con la siguiente recurrencia:

$$mcd(x,y) = \begin{cases} x & \text{si } y = 0 \\ mcd(y, resto(x,y)) & \text{si } x \geq y \text{ e } y > 0 \end{cases}$$

   Realiza una función recursiva y otra iterativa que calculen el máximo común divisor de dos enteros.

5. Escribe una función recursiva escribeNumeros(ini, fin) que muestre en la pantalla los enteros del ini al fin.

6. Escribe una función que tome como parámetro un directorio y devuelva la suma del tamaño en *bytes* de los archivos que cuelgan del árbol que tiene como raíz dicho directorio.

7. Escribe una función que tome como parámetro un directorio y un nombre de archivo y devuelva el nombre de una carpeta del árbol que tiene como raíz dicho directorio que contiene el archivo o una cadena vacía si el archivo no se encuentra en el árbol.

8. Escribe una función que tome como parámetro un directorio y un nombre de archivo y devuelva un *array* de celdas con los nombres de las carpetas del árbol que tiene como raíz dicho directorio que contienen el archivo—el *array* estará vacío si el archivo no se encuentra en el árbol.