Capítulo 1

El programa Matlab

1.1 El entorno de trabajo

Al inicio de MATLAB se muestra un *escritorio* con un mosaico de ventanas de funcionalidad diversa. Múltiples ventanas pueden mantenerse abiertas y tenerse a la vista en el mosaico o superpuestas en las sub-ventanas o independientes del escritorio.

Las seis ventanas siguientes pueden abrirse y cerrarse desde el menú:

`Command Window`	Ventana de comandos: para la ejecución de órdenes y la muestra de resultados
`Command History`	Historial de comandos: muestra la historia de las órdenes introducidas en la ventana de comandos y permite copiarlas y ejecutarlas.
`Current Directory`	Directorio actual: muestra los ficheros del directorio actual (guarda historial de directorios) y permite abrirlos, ejecutarlos, etc.
`Workspace`	Espacio de trabajo: muestra las variables definidas y sus datos. Permite borrarlas, cambiar los valores, etc.
`Launch Pad`	Permite obtener ayuda, demos y bucear por las extensiones de MATLAB.

`Help`	Ventana de ayuda.

Además de estas tenemos las ventanas siguientes:

`Editor/Debugger`	Editor y depurador de ficheros `*.m`. Para editar ficheros de ordenes de MATLAB, ejecutarlos y seguir paso a paso su ejecución facilitando la depuración de errores.
`Array Editor`	Para construir y manipular matrices como una tabla.
`Plot Windows`	Ventanas gráficas (pueden abrirse varias) para mostrar las gráficas y dibujos.

Salvo las ventanas gráficas, todas las demás pueden incorporarse al escritorio de MATLAB, basta elegir en el menú de la ventana `View/Dock Nombre_Ventana`. Las ventanas del escritorio pueden ser cambiadas de tamaño o *arrastradas* para cambiar su ubicación.

1.1.1 Ventana de comandos

Es la más importante, donde se ejecutan los comandos de MATLAB. La marca "`>>`" indica que el programa está preparado para recibir instrucciones (es el *prompt* –indicador de entradas–).

En la ventana de comandos se trabaja principalmente en modo interactivo, es decir, se van introduciendo las órdenes y se producen resultados. La edición del texto en esta ventana es un poco pobre, pues es un editor por líneas: sólo puede añadirse, quitarse o cambiar el texto en la línea activa (aunque pueden utilizarse varias líneas para escribir una instrucción sólo la última es la activa).

Las líneas de comandos pueden ser muy largas (1024 caracteres), pero el texto inicial se pierde de vista por la izquierda y hay que usar la barra de desplazamiento para verlo. Teniendo la opción `Wrap Lines` activada (menú `File/Preferences/Command Window`) las líneas se cortan automáticamente al llegar al final de lo visible en la ventana y siguen en la siguiente (con la opción activada, desaparece la barra de desplazamiento).

1.1.1.1 Edición en línea

↵, *Enter*	Finaliza la entrada de la expresión y la evalúa.
...	Continúa la entrada de la expresión en la línea siguiente.
,	Permite separar instruciones en la misma ejecución (misma línea o lineas continuadas).
;	Ejecuta la instrucción pero no muestra el resultado.
← →	Permite desplazarse por la línea activa.
Esc	Borra la línea
%	Todo lo que aparece en la línea después del símbolo `%` se considera un comentario.
↑ ↓	Permite desplazarse por el historial de comandos
`clc`	Borra el contenido de la ventana de comandos y coloca el prompt (`>>`) en la primera línea de la ventana.
`home`	Desplaza el prompt (`>>`) a la primera línea de la ventana sin borrar lo anterior.
fun. Tab	Con las iniciales del nombre de una función y luego *Tab* se completa el nombre. Si hay varias posibles, pulsando de nuevo *Tab* muestra todas las opciones.

1.1.1.2 Petición de ayuda

`help` *nombre*	Muestra ayuda en línea sobre el item *nombre* solicitado (comando, función, librería, etc.). Ejecutado solo muestra un listado de los apartados generales.
`helpwin` *nombre*	Como `help` pero lo muestra en la ventana de ayuda, lo que permite enlaces a temas relacionados o ampliar la ayuda del tema con `Go to online doc`.
`doc` *nombre*	Abre la ventana de ayuda. Seguido de *nombre* muestra información detallada referente a ese item (como la opción `Go to online doc` de `helpwin`).

Seleccionando el nombre de un comando, con el boton derecho del ratón podemos acceder a la página del Help sobre dicha función.

1.1.1.3 Ejecución de ficheros de comandos

Los ficheros `*.m` (o M-ficheros) son ficheros de texto ASCII, que contienen conjuntos de comandos o definición de funciones. La importancia de estos M-ficheros es que, introduciéndolos en la linea de comandos, se ejecutan uno tras otro todos los comandos contenidos en dicho fichero. El poder guardar instrucciones y grandes matrices en un fichero permite ahorrar mucho trabajo de tecleado.

Cuando al ejecutar un M-fichero se produce un error y se obtiene el correspondiente mensaje en la `Command Window`, MATLAB muestra mediante un subrayado un enlace a la línea del fichero fuente en la que se ha producido el error. Pinchando en ese enlace se va a la línea correspondiente del fichero por medio del `Editor/Debugger`.

1.1.2 Editor/Debugger de ficheros `*.m`

Aunque los M-ficheros se pueden crear con cualquier editor de ficheros ASCII, MATLAB dispone de un editor que permite tanto crear y modificar estos ficheros, como ejecutarlos paso a paso para ver si contienen errores (proceso de Debug o depuración). Este editor muestra con diferentes colores los diferentes tipos o elementos constitutivos de los comandos (en verde los comentarios, en rojo las cadenas de caracteres, etc.). Se preocupa también de que las comillas o paréntesis que se abren, no se queden sin el correspondiente elemento de cierre. Colocando el cursor justo después de una apertura o cierre de corchete o paréntesis, el editor muestra subrayado con qué cierre o apertura se empareja el elemento considerado; si no se empareja con ninguno, aparece con una rayita de tachado.

Seleccionando varias líneas y clicando con el botón derecho aparece un menú contextual que permite entre otras cosas comentar con el carácter "`%`" todas las líneas seleccionadas.

La ejecución de un fichero de comandos controlada con el `Debugger` se comienza eligiendo el comando `Run` en el menú, pulsando *F5*, pinchando en el botón `Continue|Run|Save and Run` de la barra de herramientas del Editor o tecleando el nombre del fichero en la línea de comandos de la Command Window.

Si se ha introducido un `breakpoints` (puntos de parada), que aparecen como pontos rojos en el margen izquierdo, se detiene la ejecución del programa; una flecha verde indica la sentencia en que está detenida la ejecución (antes de ejecutar dicha sentencia) y se activan los botones que corresponden al `Debugger`. El significado de estos botones, que aparece al colocar el cursor sobre cada uno de ellos, es el siguiente:

`Set/Clear Breakpoint`	Coloca o borra un breakpoint en la línea en que está el cursor.
`Clear All Breakpoints`	Elimina todos los breakpoints que haya en el fichero.
`Step`	Avanzar un paso sin entrar en las funciones de usuario que se llamen en esa línea.

`Step In`	Avanzar un paso, y si en ese paso hay una llamada a una función cuyo fichero `.m` está accesible, entra en dicha función.
`Step Out`	Salir de la función que se está ejecutando en ese momento.
`Continue`	Continuar la ejecución hasta el siguiente breakpoint.
`Quit Debugging`	Terminar la ejecución del Debugger.

Cuando el cursor se coloca sobre una variable aparece una pequeña ventana con los valores numéricos de la variable en ese momento de la ejecución.

En cualquier momento de la ejecución del `Debugger`, se puede ir a la línea de comandos y teclear una expresión para ver su resultado. También se puede seleccionar con el ratón una sub-expresión en cualquier línea y evaluarla con el el botón derecho. El resultado de evaluar esa sub-expresión aparece en la línea de comandos de MATLAB.

El `Debugger` es un programa que interesa conocer bien, pues es muy útil para detectar y corregir errores y enormemente útil para aprender métodos numéricos y técnicas de programación. Para aprender a manejarlo lo mejor es practicar.

1.2 Aritmética elemental

Las operaciones aritméticas básicas se utilizan en MATLAB con los símbolos: suma ($+$), resta ($-$), producto ($*$), división ($/$) y potencia ($\hat{}$).

Existe además otra operación denominada *división inversa* o *por la izquierda* ($\backslash$), y su resultado se obtiene de $\mathtt{a}\backslash\mathtt{b} = \frac{1}{\mathtt{a}} * \mathtt{b}$. Su significado y operatividad real es algo más complejo, sobre todo con matrices, pero no entraremos en ello; puede consultarse la bibliografía o la ayuda (`help mldivide`).

1.2.1 Constantes y elementos especiales

`pi` π.

`exp(1)` El número e.

`Inf`, `inf` Infinito. Como en `1/0`, se devuelve `Inf`.

`NaN`, `nan` Indeterminación (no es un número *Not a number*). Como en `0/0 -> NaN`.

1.2.2 Algunas funciones básicas

`abs(a)` Valor absoluto de a.

`sign(a)` Signo de a (1, 0 ó -1).

`floor(a)` El mayor entero menor o igual a a (función suelo).

`ciel(a)` El menor entero mayor o igual a a (función techo).

`fix(a)` Elimina la parte decimal de a (función parte entera).

`rem(n,m)` Resto de la división entera de n entre m (`rem(n,m)` $=$ `n` $-$ `m` $*$ `floor(n/m)`).

`mod(n,m)` Resto de la división entera de n entre m (`mod(n,m)` $=$ `n` $-$ `m` $*$ `fix(n/m)`).

Si `sign(n)` $=$ `sign(m)` entonces `rem` $=$ `mod`.

`gcd(n,m)`	Máximo común divisor
`lcm(n,m)`	Mínimo común multiplo
`mod(n,m)`	Mínimo común multiplo
`max(a,b)`	Máximo de los valores a y b
`min(a,b)`	Mínimo de los valores a y b

1.3 Vectores y matrices

MATLAB es fundamentalmente un programa para realizar cálculos numéricos con vectores y matrices. En general, maneja los vectores como matrices fila y así, las operaciones y la mayoría de las funciones predefinidas en el programa para las matrices les son aplicables; por ello, *sólo en aquellos casos en que el funcionamiento sea distinto, distinguiremos los vectores de las matrices.*

Para introducir una matriz desde el teclado, entre corchetes se introducen los elementos de cada fila separados por comas o espacios en blanco y las filas de la matriz separadas por punto y coma (todas las filas deben tener el mismo número de elementos). Así

- ⋆ `[a11 a12 a13; a21 a22 a23]` ó `[a11, a12, a13; a21, a22, a23]`
- ⋆ `A = [a11 a12 a13; a21 a22 a23]` ó `A = [a11, a12, a13; a21, a22, a23]`

En MATLAB se definirán las matrices y las expresiones que las combinan para obtener resultados. Si estos resultados se asignan a variables podrán ser usados posteriormente en otras expresiones (arriba, hemos asignado a la variable A el valor de la matriz).

Dimensiones de una matriz

`size(A)`	Devuelve el vector $[\mathtt{n_F}, \mathtt{n_C}]$ con el tamaño (filas y columnas) de la matriz A.
`size(A,1)`	Devuelve el número de filas de la matriz A.
`size(A,2)`	Devuelve el número de columnas de la matriz A.
`length(A)`	Si A es un vector (fila o columna), `length` devuelve el número de elementos del vector.

1.3.1 Variables matriciales

Una *variable* es el nombre que se da a una entidad numérica, que puede ser un escalar, un vector o una matriz. El valor de la variable, incluso el tipo de entidad que representa, puede cambiar. Los *nombres de variable* comienzan siempre con una letra y pueden constar hasta de 31 letras o números. El caracter "_" se considera una letra y MATLAB *distingue entre mayúsculas y minúsculas.*

Para crear una variable (o cambiar su valor) basta colocarla a la izquierda del **operador de asignación** (=).

Podemos evaluar una expresión en MATLAB almacenando el resultado en una variable

```
variable=expresión
```

o sencillamente para obtener el resultado

```
expresión
```

en cuyo caso el valor se asigna en una variable interna llamada `ans` (de *answer*=respuesta) que almacena el último resultado obtenido (y puede usarse a continuación). Se considera que una expresión termina cuando se introduce *Intro* (*Return* o "↵"). Si se desea que una expresión continúe en la línea siguiente, hay que introducir tres puntos

(...) antes de pulsar Intro. También se pueden incluir varias expresiones en una misma línea separándolas por comas (,) o puntos y comas (;). Si una expresión termina en punto y coma (;) su resultado se calcula, pero no se escribe en pantalla.

1.3.1.1 Gestión de variables

La ventana del espacio de trabajo (`Workspace`) muestran las variables usadas en la sesión, pudiendo ser eliminadas total o parcialmente. Esto puede hacerse también desde la línea de comandos mediante:

`who` `whos`	Lista las variables del espacio de trabajo, con sus características si usamos `whos`.
`clear` `clear` $v,\ w$	Elimina todas las variables (excepto las globales) o las variables indicadas.

El comando `clear` admite más posibilidades en función del carácter de las variables: globales, función, etc. No entraremos en eso aunque si aparecerá algún comentario en el capítulo de Programación, 1.4.

1.3.2 Operaciones con matrices

Hay dos tipos. Las operaciones aritméticas matriciales que se rigen por las reglas conocidas del Álgebra lineal: suma $(+)$, resta $(-)$, producto $(*)$, división $(/)$ y potencia $(\hat{})$; y las operaciones con las matrices que se realizan *elemento a elemento*: producto e.e. $(.*)$, división e.e. $(./)$ y potencia e.e. $(.\hat{})$.

También son válidas, y en las mismas condiciones, la división por la izquierda $(\backslash)$ y la división por la izquierda e.e. $(.\backslash)$.

Para aplicar las operaciones e.e. entre matrices se requiere que los operandos tengan el mismo tamaño (evidentemente aplicadas entre escalares producen el mismo resultado que las operaciones habituales).

En la siguiente relación mostramos cual es el funcionamiento de todas las operaciones entre escalares y matrices, y él de las nuevas entre matrices:

Suma o resta de escalar y matriz	$\mathtt{a} \pm \mathtt{V} = [\mathtt{a} \pm \mathtt{v}_1, \mathtt{a} \pm \mathtt{v}_2, \ldots, \mathtt{a} \pm \mathtt{v}_n]$	$= \mathtt{V} \pm \mathtt{a}$
Escalar multiplicado por matriz	$\mathtt{a} * \mathtt{V} = [\mathtt{a} * \mathtt{v}_1, \mathtt{a} * \mathtt{v}_2, \ldots, \mathtt{a} * \mathtt{v}_n]$	$= \mathtt{V} * \mathtt{a} = \mathtt{a}. * \mathtt{V}$
Matriz dividido por escalar	$\mathtt{V}/\mathtt{a} = \left[\frac{\mathtt{v}_1}{\mathtt{a}}, \frac{\mathtt{v}_2}{\mathtt{a}}, \ldots, \frac{\mathtt{v}_n}{\mathtt{a}}\right]$	
Escalar dividido por matriz e.e.	$\mathtt{a}./\mathtt{V} = \left[\frac{\mathtt{a}}{\mathtt{v}_1}, \frac{\mathtt{a}}{\mathtt{v}_2}, \ldots, \frac{\mathtt{a}}{\mathtt{v}_n}\right]$	
Matriz elevada e.e. a escalar	$\mathtt{V}.\hat{}\,\mathtt{a} = \left[\mathtt{v}_1^{\mathtt{a}}, \mathtt{v}_2^{\mathtt{a}}, \ldots, \mathtt{v}_n^{\mathtt{a}}\right]$	
Escalar elevado a matriz e.e.	$\mathtt{a}.\hat{}\,\mathtt{V} = \left[\mathtt{a}^{\mathtt{v}_1}, \mathtt{a}^{\mathtt{v}_2}, \ldots, \mathtt{a}^{\mathtt{v}_n}\right]$	
Matriz por matriz e.e.	$\mathtt{V}. * \mathtt{W} = \left[\mathtt{v}_1 * \mathtt{w}_1, \mathtt{v}_2 * \mathtt{w}_2, \ldots, \mathtt{v}_n * \mathtt{w}_n\right]$	
Matriz dividida por matriz e.e.	$\mathtt{V}./\mathtt{W} = \left[\frac{\mathtt{v}_1}{\mathtt{w}_1}, \frac{\mathtt{v}_2}{\mathtt{w}_2}, \ldots, \frac{\mathtt{v}_n}{\mathtt{w}_n}\right]$	
Matriz elevada a matriz e.e.	$\mathtt{V}.\hat{}\,\mathtt{W} = \left[\mathtt{v}_1{}^{\mathtt{w}_1}, \mathtt{v}_2{}^{\mathtt{w}_2}, \ldots, \mathtt{v}_n{}^{\mathtt{w}_n}\right]$	

Con caracter general, como ocurre arriba, en las operaciones (y otras relaciones) donde aparecen involucrados un escalar y una matriz, el funcionamiento es como si el escalar fuera una matriz de elementos idénticos, del tamaño adecuado, y se operara elemento a elemento.

1.3.3 Otras formas de construir matrices

1.3.3.1 Juntando matrices

`[A1 A2 A3 ...]` `horzcat(A1, A2, ...)`	Crea una matriz *pegando* las matrices A_1, A_2, ..., (todas deben tener igual número de filas).
`[A1 ; A2 ; ...]` `vertcat(A1, A2, ...)`	Crea una matriz *apilando* las matrices A_1, A_2, ..., una debajo de otra (todas deben tener igual número de columnas).

Pegar y apilar puede hacerse también en una sola expresión.

1.3.3.2 Usando funciones predefinidas

`A'`	Devuelve la matriz traspuesta de A.
`inv(A)`	Devuelve la matriz inversa de A.
`triu(A)`	Devuelve la parte triangular superior de la matriz A.
`tril(A)`	Devuelve la parte triangular inferior de la matriz A.
`diag(V)`	Crea una matriz diagonal con el vector V en la diagonal.
`diag(A)`	Extrae la diagonal principal de la matriz A como vector columna.
`eyes(n)`	Crea la matriz identidad de orden n.

`eyes(m,n)`	Crea una matriz de tamaño $m \times n$ con unos en la diagonal principal y ceros en el resto.
`eyes(size(A))`	Crea una matriz del mismo tamaño que A con unos en la diagonal principal y ceros en el resto.

`ones(n)`
`ones(m,n)`
`ones(size(A))`
	Crea una matriz de unos del tamaño indicado.

`zeros(n)`
`zeros(m,n)`
`zeros(size(A))`
	Crea una matriz de ceros del tamaño indicado.

`linspace(a,b,n)`	Crea un vector con n valores equiespaciados entre a y b.

1.3.3.3 Usando el operador " : "

El operador " : " genera un vector de números en progresión aritmética. Se usa en la forma:

`a:p:b` $= [\mathtt{a}, \mathtt{a}+\mathtt{p}, \mathtt{a}+2\mathtt{p}, \ldots,]$ mientras `a+kp` sea menor o igual a `b`

Si a es menor que b, p debe ser positivo y si $a > b$ entonces $p < 0$; en caso contrario se crea un vector vacío.

En el caso particular de que el incremento sea de una unidad, $p = 1$, puede escribirse `a:b` en lugar de `a:1:b`.

Como hemos dicho el operador " : " genera vectores, y puede usarse para crearlos, `V=a:p:b`; pero es más interesante su uso como vector de índices para entresacar elementos de los vectores y matrices:

`V(n)`	Devuelve el n-ésimo elemento del vector V.

`V(end)` `V(length(V))`	Devuelve el último elemento del vector V.
`V(n1:n2)`	Devuelve los elementos de V entre el n_1-ésimo y el n_2-ésimo, ambos inclusive.
`V(n1:p:n2)`	Devuelve el vector de los elementos de V entre el n_1-ésimo y el n_2-ésimo, tomados de p en p unidades.
`V(n2:-p:n1)`	Devuelve el vector de los elementos de V entre el n_2-ésimo y el n_1-ésimo, tomados de p en p unidades.
`V(1:end)` `V(1:length(V))`	Devuelve el propio vector V.
`V(:)`	Devuelve todos los elementos de V como vector columna[1].

Para las matrices es similar a los vectores, pero hay que selecionar filas y columnas:

`A(m,n)`	Devuelve el elemento a_{mn} de la matriz A.
`A(m,:)`	Devuelve la fila m-ésima de la matriz A.
`A(end,:)` `A(size(A,1),:)`	Devuelve la última fila de la matriz A.

[1]En un epígrafe posterior, el 1.3.3.5, se comenta este resultado.

`A(:,n)`	Devuelve la columna n-ésima de la matriz A.
`A(:,end − n : end)`	Devuelve las últimas $n+1$ columnas de la matriz A.
`A(:,:)`	Devuelve la matriz A.
`A(m1 : m2, :)`	Define la submatriz de A formada por las filas entre la m_1-ésima y la m_2-ésima
`A(:, n1 : n2)`	Define la submatriz de A formada por las columnas entre la n_1-ésima y la n_2-ésima
`A(m1 : m2, n1 : n2)`	Define la submatriz de A formada por los elementos de las filas y columnas indicadas.

El operador "`:`" también puede usarse en la forma `a:p:b`. Tanto en este caso como en los anteriores los valores que se obtengan deben representar ordinales de filas y columnas existentes (enteros positivos que no excedan las dimensiones de la matriz).

1.3.3.4 Indexando matrices a partir de vectores

El uso del operador "`:`", comentado antes, es un caso particular de este: elegir los elementos de una matriz indiciados por un vector.

`V(J)` `V([j1 j2 j3 ...])`	Devuelve el vector formado por los elementos de V indiciados por J. Es decir, el vector $[V(j_1)\ V(j_2)\ V(j_3)\ \ldots]$.
`A(I, :)` `A([i1 i2 i3 ...], :)`	Define una matriz formada por las filas de A indiciadas por I. Es decir, por las filas i_1-ésima, i_2-ésima, etc. de A, en este orden.

`A(:,J)` `A(:,[j1 j2 j3 ...])`	Como el anterior para las columnas.
`A(I,J)`	Como el anterior para los elementos de esas filas y esas columnas.

Los vectores de índices deben contener valores enteros positivos y que no excedan la dimensión correspondiente de la matriz o vector. Pueden aparecer valores repetidos.

1.3.3.5 Matrices como vectores

Aunque las matrices se introducen por filas, MATLAB las almacena como una columna colocando las columnas de la matriz una debajo de otra. Esto permite también acceder a los elementos de una matriz con un sólo índice.

`A(:)`	Devuelve un vector columna formado por las columnas de la matriz A una detras de otra.
`A(n)`	Devuelve el n-ésimo elemento del vector columna $A(:)$.
`A(K)`	Devuelve una matriz del mismo tamaño que la matriz K, formada por los elementos de A indiciados por el elemento correspondiente de K. Es decir, el elemento ij será el k_{ij}-ésimo elemento de $A(:)$, $A(k_{ij})$.
`reshape(A,m,n)`	Devuelve una matriz de tamaño $m \times n$ cuyas columnas se forman tomando elementos sucesivos de $A(:)$. Si A tiene menos de $m \times n$ elementos se produce un error.

Nota: La matriz de índices deben contener valores válidos. Si A es $m \times n$, deben ser $1 \leq k_{ij} \leq mn$. Los valores pueden estar repetidos.

1.3.3.6 Matrices dispersas

Cuando una matriz tiene muchos elementos nulos y pocos elementos no nulos –que suele denominarse matriz dispersa (`sparse`)–, puede resultar ventajoso trabajar aprovechando el reducido número de elementos no nulos.

`sparse(I,J,V)` `sparse(I,J,V,m,n)`	Crea una matriz dispersa de tamaño $m \times n$, cuyos elementos no nulos son los dados por el vector V y los vectores I y J contienen respectivamente los índices de las filas y de las columnas que deben ocupar esos elementos no nulos. Si m y n no aparecen se toma como tamaño el mayor índice de la filas y el mayor índice de las columnas.
`sparse(A)`	Convierte la matriz (completa) A en una matriz dispersa.
`full(S)`	Convierte la matriz dispersa S en una matriz completa.
`S(:)`	Convierte la matriz dispersa S en un vector columna disperso colocando una columna tras otra.
`nnz(A)` `nnz(S)`	Devuelve el número de elementos distintos de cero de la matriz.
`nonzeros(A)` `nonzeros(S)`	Devuelve un vector columna completo con los elementos no nulos de la matriz.
`find(A)` `find(S)`	Devuelve un vector columna completo con los índices de los elementos no nulos del vector columna $A(:)$ (del vector columna disperso $S(:)$).

`[I, J] = find(A)` `[I, J] = find(S)`	Devuelve dos vectores columna con los índices de las filas y de las columnas de los elementos no nulos de la matriz.
`[I, J, K] = find(A)` `[I, J, K] = find(S)`	Devuelve tres vectores columna con los índices de las filas y de las columnas y los elementos no nulos de la matriz.

1.3.4 Operaciones y funciones lógicas

En las comparaciones y operaciones lógicas, se devuelve `1` (*verdadero*) si el resultado es cierto y `0` (*falso*) si no lo es.

Las operaciones lógicas se aplican sobre resultados y matrices lógicas o *booleanas* (de ceros y unos). En este sentido, hay que tener en cuenta que si se usa una matriz numérica cualquiera, se considera cualquier valor distinto de cero como *verdadero* (como 1) y sólo el cero es considerado *falso*.

1.3.4.1 Operadores relacionales

Si se comparan matrices, devuelve otra matriz del mismo tamaño con el resultado de la comparación elemento a elemento. (Si operamos matriz y escalar, devuelve la comparación del escalar con cada elemento de la matriz.)

`<`	Menor que	`>`	Mayor que	`==`	Igual que
`<=`	Menor o igual que	`>=`	Mayor o igual que	`~=`	Distinto que

Nota: el símbolo (∼) se obtiene con `Alt+Ctrl+4`.

1.3.4.2 Operadores lógicos

Las expresiones relacionales pueden operarse o combinarse mediante las siguientes operaciones:

`~P`	`Negación (NOT)`. Devuelve 1 si P es *falso* y 0 si es *verdadero*.
`P&Q`	`Conjunción lógica (AND)`. Devuelve 1 si P y Q son *verdaderos* y 0 en otro caso.
`P|Q`	`Disyunción lógica (OR)`. Devuelve 1 si P o Q son *verdaderos* y 0 si P y Q son *falsos*.
`xor(P,Q)`	`Disyunción exclusiva (XOR)`. Devuelve 1 si P o Q son *verdaderos*, pero no los dos, y 0 en otro caso.

Si P y Q son matrices, se devuelve otra matriz del mismo tamaño con el resultado de las operaciones elemento a elemento.

1.3.4.3 Funciones lógicas

`any(V)`	Devuelve 1 si alguno de los elementos del vector V es distinto de cero y 0 si todos son nulos.
`any(A)` `any(A,2)`	Devuelve un vector de ceros y unos, resultado de aplicar la función `any` a cada vector columna de la matriz A. Se hace por filas y devuelve vector columna.
`all(V)`	Devuelve 1 si todos los elementos del vector V son distintos de cero y 0 si alguno es nulo.
`all(A)` `all(A,2)`	Devuelve un vector de ceros y unos, resultado de aplicar la función `all` a cada vector columna de la matriz A. Se hace por filas y devuelve vector columna.

`find(V)`	Devuelve un vector con los índices (lugares) de los elementos no nulos del vector V.
`I = find(A)`	Devuelve un vector columna con los índices de los elementos no nulos del vector columna $A(:)$.
`[I, J] = find(A)`	Devuelve dos vectores columna con los índices de las filas y de las columnas de los elementos no nulos de A. (Si A es un vector devuelve vectores fila.)
`[I, J, K] = find(A)`	Devuelve tres vectores columna con los índices de las filas y de las columnas y los elementos no nulos de A. (Si A es un vector devuelve vectores fila.)
`isequal(A, B)`	Devuelve 1 si las matrices A y B son iguales y 0 en otro caso.
`ismember(A, B)`	Devuelve una matriz del mismo tamaño que A, con 1 para cada elemento de A que está en B y 0 para cada elemento de A que no está en B.
`islogical(A)`	Devuelve 1 si A es una matriz booleana (de ceros y unos) y 0 en otro caso.
`issparse(A)`	Devuelve 1 si A es una matriz dispersa y 0 en otro caso.

También son funciones lógicas las siguientes:

`isempty(A)`	Devuelve 1 si A es una matriz vacía (alguna de sus dimensiones es cero) y 0 en otro caso.
`isnan(A)`	Devuelve 1 para cada elemento indeterminado (`NaN`) de A y 0 para los que no lo son.
`isinf(A)`	Devuelve 1 para cada elemento infinito (`Inf`) de A y 0 para los que no lo son.

`isfinite(A)`	Devuelve 1 para cada elemento finito de A y 0 para los que no lo son.
`exista` `exist('a')`	Devuelve 0 si la variable o función a no existe y un número mayor que cero, según el tipo, si existe (ver `help exist`).

1.3.5 Mas funciones definidas

`abs(A)`	Matriz con el valor absoluto de cada elemento.
`sign(A)`	Matriz con el signo de cada elemento (1, 0 ó -1).
`floor(a)`	El mayor entero menor o igual a a (función suelo). En una matriz lo hace elemento a elemento
`ciel(A)`	El menor entero mayor o igual a a (función techo). En una matriz lo hace elemento a elemento
`fix(A)`	Elimina la parte decimal de a (función parte entera). En una matriz lo hace elemento a elemento
`rem(n,m)`	Resto de la división entera de n entre m. En una matriz lo hace elemento a elemento
`gcd(n,m)`	Máximo común divisor
`lcm(n,m)`	Mínimo común multiplo
`mod(n,m)`	Mínimo común multiplo
`max(a,b)`	Máximo de los valores a e b

`max(A,B)`	Devuelve una matriz del mismo tamaño haciendo los máximos elemento a elemento.
`max(V)` `m = max(V)`	Devuelve el máximo de los valores del vector.
`[m,i] = max(V)`	Devuelve el máximo y el lugar que ocupa (si hay varios devuelve el menor índice).
`max(A)` `M = max(A)`	Devuelve un vector con el máximo de cada columna de la matriz.
`[M,I] = max(A)`	Devuelve además un vector con los índices de las filas donde se encuentra el máximo de cada columna (si hay varios devuelve el menor índice).

`max(A,[],2)`
`M = max(A,[],2)`
`[M,I] = max(A,[],2)`
	Como el anterior pero hace los máximos por filas, y devuelve vectores columna.

`min`	Calcula el mínimo. Funciona igual que `max`.
`sort(V)` `S = sort(V)`	Devuelve un vector con los elementos ordenados de menor a mayor.
`[S,I] = sort(V)`	Devuelve además un vector con los índices de las posiciones anteriores a la ordenación.
`sort(A)` `S = sort(A)`	Devuelve un matriz con los elementos de cada columna ordenados de menor a mayor.

`[S, I] = sort(A)`	Devuelve además una matriz con los índices de las filas donde estaban antes de la ordenación.

`sort(A, 2)`
`S = sort(A, 2)`
`[S, I] = sort(A, 2)`
	Como el anterior pero ordena los elementos de las columnas.

`sum(V)`	Suma los elementos del vector V.
`sum(A)`	Vector con las sumas de los elementos de las columnas de la matriz A.
`sum(A, 2)`	Vector columna con las sumas de los elementos de las filas de la matriz A.

1.3.6 Asignaciones: cambiar y borrar matrices

Hemos usado el *operador asignación* "=" para dar nombre a matrices, aunque en realidad "hemos asignado a una variable el valor del contenido de una matriz". Este es el significado del operador:

$\texttt{P}_{\texttt{izq}} = \texttt{P}_{\texttt{dch}}$	Asigna al elemento de la izquierda el valor de la derecha.

Con caracter general debemos tener en cuenta que:

- ⋆ Si $\texttt{P}_{\texttt{izq}}$ es un nombre de variable válido, crea la variable y le asigna el resultado de la derecha. Si ya existe cambia su asignación por lo nuevo (es como si la creara de nuevo).
- ⋆ La misma variable puede usarse en los dos términos de la asignación (por ejemplo `A=A+A`).

- ⋆ Si `P_izq` es parte del contenido de una variable sólo puede cambiarse por algo coherente (si es uno de los elementos escalares de una matriz debe cambiarse por otro escalar, etc.).
- ⋆ En una expresión sólo puede hacerse una asignación, aunque puede ser múltiple si `P_dch` es una función con multiples salidas (como `size` o `max`). Veremos esto al hablar de las funciones.

Cambiar valores de matrices.

`V(n) = d`	Sustituye el n-ésimo elemento del vector V por el valor d.
`A(m,n) = d`	Sustituye el elemento a_{mn} de la matriz A por el valor d.
`A(n) = d`	Sustituye el n-ésimo elemento del vector $A(:)$ por el valor d.
`A(I,J) = D`	Sustituye los elementos de A indiciados con los vectores I y J, por los elementos correspondientes de la matriz D. Las matrices $A(I,J)$ y D deben ser del mismo tamaño[2].
`A(K) = D`	Sustituye los elementos de A indiciados con la matriz K, por los elementos correspondientes de la matriz D. Nota: En realidad hace `A(K(:)) = D(:)`,[3] por lo que las matrices K y D deben tener el mismo número de elementos (aunque no sean del mismo tamaño).

[2]Ver apartados 1.3.3.3 y 1.3.3.4 sobre indexado de matrices con vectores
[3]Ver apartado 1.3.3.5 sobre matrices como vectores

A tener en cuenta: Si los índices usados exceden las dimensiones de la matriz que modificamos, se amplía esta y se pone 0 en los huecos creados y no asignados. Si la matriz a modificar no existe previamente, se crea con las dimensiones pertinentes.

Eliminar elementos de matrices. Asignando la matriz vacía "`[]`", podemos eliminar contenidos de las variables:

`V(I) = []`	Elimina los elementos del vector V indiciados por I. Devuelve el V resultante.
`A(I,:) = []` `A(:,J) = []`	Elimina las filas de la matriz A indiciadas por I, resp. las columnas indiciadas por J. Devuelve la A resultante. No pueden eliminarse filas y columnas a la vez.
`A(K) = []`	Elimina los elementos de $A(:)$ indiciados por K. Devuelve un vector fila.

1.3.7 Guardar el trabajo realizado

1.3.7.1 Guardar variables y estados de una sesión

Al salir del programa MATLAB todo el contenido de la memoria se borra automáticamente. Para guardar el trabajo realizado y poderlo recuperar más tarde en el mismo punto en el que se dejó (con las mismas variables definidas, con los mismos resultados intermedios, etc.), se usan los comandos:

`save fichero A a ...`	Guarda en el fichero `fichero.mat` las variables especificadas `A a ...` de la sesión actual. (El nombre del fichero debe preceder a las variables.)
`load fichero A a ...`	Recupera del fichero de sesión `fichero.mat` las variables especificadas.

Si en los comandos no se incluye el nombre de fichero se utiliza por defecto un fichero llamado `matlab.mat` o `matlab`. Si no se indica ningún nombre de variable se suponen todas.

1.3.7.2 Guardar sesión y copiar salidas

Se puede almacenar en un fichero de texto lo que el programa va haciendo en la sesión (la entrada y salida de los comandos utilizados). Esto se hace con el comando `diary`:

```
diary on
diary off
diary
```

La opción `on` activa la función `diary` y `off` la desactiva. `diary` sin opciones cambia el estado.

```
diary fichero.txt
```

Activa la función `diary` y lo guarda en `fichero.txt`. Si no se incluye nombre de fichero usa por defecto un fichero llamado `diary` sin extensión.

Los ficheros generados por `diary` son ficheros de texto ASCII y pueden abrirse con cualquier editor.

1.4 Programación

Hacer un *programa* no es más que escribir una tras otra todas las operaciones necesarias para obtener un resultado. En el fondo, los lenguajes de programación no hacen más que simplificar esta tarea, fundamentalmente mediante el uso de estructuras complejas que permiten simplificar tareas tediosas: bifurcaciones y bucles, funciones, variables, etc.

1.4.1 Bifurcaciones y bucles

MATLAB posee un lenguaje de programación que también dispone de sentencias para realizar bifurcaciones y bucles. Las bifurcaciones permiten realizar una u otra operación según se cumpla o no una determinada condición, y los bucles permiten repetir las mismas o análogas operaciones sobre datos distintos.

1.4.1.1 Bifurcaciones

Sentencia `if`. La sentencia `if` es la bifurcación más sencilla: si una condición es cierta hace una tarea, si no es cierta no la hace o hace otra.

```
si Condición
   hacer: Tarea
fin
```

$\Longrightarrow$

```
if Cond.
   Tarea
end
```

```
si Cond.
   hacer: Tarea_1
sino
   hacer: Tarea_2
fin
```

$\Longrightarrow$

```
if Cond.
   Tarea_1
else
   Tarea_2
end
```

Las sentencias `if` pueden anidarse, es decir, que dentro de la *Tarea* haya otra sentencia `if`. Para ello existe también la bifurcación múltiple, en la que pueden concatenarse tantas condiciones como se desee, y que se simplifica mediante el uso de `elseif`:

```
si Cond_1
   hacer: Tarea_1
sino
   hacer: si Cond_2
             hacer: Tarea_2
          sino
             hacer: Tarea_3
          fin
fin
```

$\Longrightarrow$

```
if Cond_1
     Tarea_1
elseif Cond_2
     Tarea_2
else      Tarea_3
end
```

Sentencia `switch`. Cuando las opciones de bifurcación se presentan según el resultado de una expresión con varios casos distintos, es mucho más económico usar la sentencia `switch`:

```
según Expresión
     en el Caso_1
          hacer: Tarea_1
     en el Caso_2
          hacer: Tarea_2
     ..................
     en otro caso
          hacer: Tarea_Def
fin
```

$\Longrightarrow$

```
switch  Expresión
     case Caso_1
          Tarea_1
     case Caso_2
          Tarea_2
     ..............
     otherwise
          Tarea_Def
end
```

donde la opción por defecto `otherwise` (en otro caso) puede ser omitida: si no está presente y no se cumple ninguno de los casos anteriores, no se hace nada.

Sólo se ejecuta una de las tareas, por lo que si hay casos coincidentes siempre se ejecutará la tarea correspondiente al primero de ellos.

1.4.1.2 Bucles

Como ya hemos dicho, los bucles son estructuras iterativas para repetir varias veces las mismas operaciones. Las operaciones se pueden repetir un número determinado de veces (`for`) o variable, mientras sea necesario (`while`).

Sentencia `for`. En esta estructura se repiten las operaciones un número fijo de veces, que viene determinado por el número de elementos de un *vector de recorrido*. Se usa con una *variable contador* que va tomando uno tras otro todos los valores del vector.

```
para contador recorriendo Vector
     hacer: Tarea
fin
```

$\Longrightarrow$

```
for Contador=Vector
     Tarea
end
```

Es habitual dar la expresión del vector de recorrido con el operador ":", como `1:t:n`, que se corresponde más con la sintaxis habitual de la estructura `for` en otros lenguajes (`for i=1 to n by t`, desde que `i` vale 1 hasta que vale `n` en incrementos de `t`). Pero puede usarse cualquier expresión que de como salida un vector o sencillamente su nombre. Si ponemos como vector de recorrido una matriz, la variable contador es vectorial y toma en cada paso una columna de la matriz.

Aunque es posible modificar el vector de recorrido dentro del bucle, no debe hacerse (los resultados son casi siempre impredecibles).

Sentencia `while`. Si lo que queremos es un bucle donde las operaciones se repitan hasta conseguir el resultado, pero no sabemos cuantas iteraciones vamos a necesitar, debemos usar la sentencia `while`:

```
mientras Condición
     hacer: Tarea
fin
```

$\Longrightarrow$

```
while Condición
     Tarea
end
```

donde se ejecuta la Tarea mientras la Condición sea cierta.

Si la condición es siempre verdadera nos metemos en un bucle infinito. Por ello debemos tener cuidado en asegurar que el valor de la condición cambiará en algún momento al ejecutar las operaciones. (Si ocurriera que entramos en un bucle infinito, pulsando `Ctrl+C` se sale de él.)

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

Sentencia `break`. La sentencia `break` hace que se termine la ejecución del bucle más interno de los que comprenden a dicha sentencia.

1.4.2 M-ficheros: ficheros de comandos y funciones

Existen dos tipos de ficheros `*.m`, los ficheros de comandos (llamados *scripts* en inglés) y las funciones. Los primeros contienen simplemente un conjunto de comandos que se ejecutan sucesivamente cuando se teclea el nombre del fichero (sin la extensión) en la línea de comandos de MATLAB o se incluye dicho nombre en otro M-fichero.

Las *funciones* son M-ficheros declarados como funciones, asociándoles unos argumentos de entrada y de salida. La diferencia fundamental entre ambos es su comportamiento en la ejecución: mientras que los ficheros de comandos

se ejecutan como si formaran parte del programa que los llama, las funciones se ejecutan "a parte" devolviendo los resultados pedidos sin interactuar con el resto del programa.

Es decir, las funciones crean su propio "espacio de trabajo" donde sus variables son *locales* que desaparecen al salir de la función, sin modificar las variables existentes en el espacio de trabajo base (aunque se llamen igual), salvo las usadas para obtener la salida de la función.

Las variables creadas en los ficheros de comandos se incorporan a las variables del espacio de trabajo desde donde se llaman (el espacio de trabajo base de MATLAB o el de una función) y pueden modificar variables existentes previamente.

El comando `echo` hace que se impriman los comandos que están en un M-fichero a medida que van siendo ejecutados. Este comando tiene varias formas:

`echo on`	activa el `echo` en todos los ficheros de comandos
`echo off`	desactiva el `echo`
`echo file on`	donde "file" es el nombre de un fichero de función, activa el `echo` en esa función
`echo file off`	desactiva el `echo` en la función
`echo file`	pasa de `on` a `off` y viceversa
`echo on all`	activa el `echo` en todas las funciones
`echo off all`	desactiva el `echo` de todas las funciones

1.4.2.1 Definición de funciones

El uso de las funciones ayuda a resolver situaciones muy frecuentes en programación:

- ⋆ Un mismo trabajo que debe ser realizado varias veces en un mismo programa
- ⋆ Utilización de un programa por parte de otro programa.
- ⋆ Un programa que contiene módulos o subprogramas encargados de realizar ciertas tareas, más o menos importantes, complejas o rutinarias, y cuyo desarrollo completo produce un programa demasiado confuso y extenso.

Para declarar un M-fichero como una función hay que incluir como primera orden el comando `function`. Tiene la siguiente sintaxis:

```
function [ret_1,ret_2,...,ret_n]=nombre_de_funcion(arg_1,arg_2,...,arg_m)
```

donde `nombre_de_funcion` es el nombre de la función. Entre corchetes y separados por comas van los *valores de retorno* o salidas de la función, y entre paréntesis también separados por comas van los *argumentos*. Puede haber funciones sin valor de retorno y también sin argumentos. Si no hay valores de retorno se omiten los corchetes y el signo igual (`=`); si sólo hay un valor de retorno no hace falta poner corchetes. Tampoco hace falta poner paréntesis si no hay argumentos.

En MATLAB una función no modifica nunca los argumentos que recibe; los resultados de la función se obtienen siempre a través de los valores de retorno, que pueden ser múltiples y matriciales. Tanto el número de argumentos como el de valores de retorno no tienen que ser fijos, y pueden depender de cómo el usuario llama a la función (ver por ejemplo la función `size`)[4].

[4]Para más información sobre este punto consultar la bibliografía recomendada.

- ⋆ El nombre del fichero `*.m` que guarda una función debe coincidir con el nombre de la función.
- ⋆ Como cualquier M-fichero, las funciones pueden llamar o a otra función o incluso llamarse a si mismas de forma recursiva.
- ⋆ Es aconsejable que cada expresión acabe en punto y coma "`;`", para evitar que aparezcan en pantalla los resultados intermedios.

Sentencia `return`. De ordinario las funciones devuelven el control después de que se ejecute la última de sus sentencias. La sentencia `return`, incluida dentro del código de una función, hace que se devuelva inmediatamente el control al programa que realizó la llamada.

Help para las funciones de usuario Las funciones permiten definir funciones nuevas enteramente análogas a las predefinidas de MATLAB, con su nombre, sus argumentos y sus valores de retorno. Las funciones creadas por los usuarios permiten extender las posibilidades de MATLAB; las bibliotecas de M-ficheros (`toolkits`) son un ejemplo.

También las funciones creadas por el usuario pueden tener su help, análogo al que tienen las propias funciones de MATLAB. Al pedir ayuda sobre cualquier fichero `*.m`, el programa responde con el primer bloque de líneas de comentario que encuentra en el fichero, es decir, las primeras líneas consecutivas que comiencen por "`%`".

Podemos pues, en ese bloque, incluir toda la información que nos parezca relevante, qué hace, qué variables usa, si es función qué argumentos y valores de retorno tiene definidos, etc. Es particularmente importante la primera línea, aparece como descripción del fichero en la ventana del directorio actual y es la chequeada en las búsquedas.

1.4.3 Lectura y escritura interactiva de variables

Una forma sencilla de leer variables desde teclado y escribir mensajes en la pantalla es mediante las dos funciones siguientes:

`disp(A)` `disp('Mensaje')`	Muestra un mensaje de texto o el valor de una matriz, pero sin imprimir su nombre.
`n = input('Mensaje')` `n = input('Mensaje','s')`	Muestra un mensaje en la línea de comandos y recupera como valor de retorno un valor numérico o el resultado de una expresión tecleada por el usuario. Con el parámetro '`s`' el texto tecleado como respuesta se lee y se almacena sin evaluar (como cadena de caracteres).

Con la función `input`, después de imprimir el mensaje, el programa espera que el usuario teclee el valor numérico o la expresión. Cualquier expresión válida de MATLAB es aceptada por este comando y, si no es un valor numérico, la expresión introducida se evalua con los valores actuales de las variables de MATLAB y el resultado de la evaluación es lo que se almacena como valor de retorno. Con el parámetro '`s`' se inhibe la evaluación de la expresión, que se almacena como texto sin evaluarse.

1.4.4 Recomendaciones generales de programación

Las funciones vectoriales de MATLAB son mucho más rápidas que sus contrapartidas escalares. En la medida de lo posible es muy interesante vectorizar los algoritmos de cálculo, es decir, realizarlos con vectores y matrices, y no con variables escalares dentro de bucles.

Conviene desarrollar los programas incrementalmente, comprobando cada función o componente que se añade. De esta forma siempre se construye sobre algo que ya ha sido comprobado y que funciona: si aparece algún error, lo

más probable es que se deba a lo último que se ha añadido, y de esta manera la búsqueda de errores está acotada y es mucho más sencilla. De ordinario el tiempo de corrección de errores en un programa puede ser 4 ó 5 veces superior al tiempo de programación. El debugger es una herramienta muy útil a la hora de acortar ese tiempo de puesta a punto.

En este mismo sentido, puede decirse que pensar bien las cosas al programar (sobre una hoja de papel en blanco, mejor que sobre la pantalla) siempre es rentable, porque se disminuye más que proporcionalmente el tiempo de depuración y eliminación de errores.

Otro objetivo de la programación debe ser mantener el código lo más sencillo y ordenado posible. Al pensar en cómo hacer un programa o en cómo realizar determinada tarea es conveniente pensar siempre primero en la solución más sencilla.

El código debe ser escrito de una manera clara y ordenada, introduciendo comentarios, utilizando líneas en blanco para separar las distintas partes del programa, sangrando las líneas para ver claramente el rango de las bifurcaciones y bucles, utilizando nombres de variables que recuerden su significado, etc.

En cualquier caso, la mejor forma (y la única) de aprender a programar es programando.

1.5 Bibliografía

[1] Javier García de Jalón, y otros, *Aprenda Matlab 6.1 (como si estuviera en primero)*, E.T.S. de Ingenieros Industriales, UPM. (en formato pdf: `B_Matlab61Pro.pdf`).

[2] Óscar Angulo Torga, y otros, *Curso de iniciación a Matlab*, Dpto. Matemática Aplicada, EUP, UVA. (en formato pdf: `B_Curso-MatLab.pdf`).

[3] Alfred V. Aho, y otros, *Estructura de datos y algoritmos*, Addison-Weslwy Iberoamericana, 1988, USA.