

1.1 Instalación del software necesario, entorno de desarrollo y primeros pasos

- **MATLAB y Octave**

MATLAB (abreviatura de *MATrix LABoratory*, 'laboratorio de matrices') es un software matemático de cálculo técnico y científico. Fue creado por Cleve Moler en 1984.

MATLAB es un programa especialmente diseñado para efectuar cálculos numéricos con vectores y matrices, que resultan fundamentales a fin de resolver problemas de ingeniería y ciencia. Además, una de sus características más atractivas es la de facilitar la realización de una amplia variedad de gráficos en dos y tres dimensiones. MATLAB posee también un lenguaje de programación propio, de muy alto nivel (lenguaje M). En definitiva, es una magnífica herramienta para desarrollar aplicaciones técnicas, muy utilizada en el campo de la ingeniería y fácil de manejar.

Pueden extender las capacidades del módulo básico de MATLAB con las *cajas de herramientas* (*toolboxes*) que consisten en paquetes de ampliación del software básico y que son aplicables a determinados campos de la ciencia: matemática en general (optimización, bases de datos, estadística, ecuaciones diferenciales, redes neuronales), procesado de señales, procesado de imágenes, adquisición de datos, finanzas, sistemas de control, etc.

MATLAB es un software abierto que puede relacionarse con otras aplicaciones, como Excel, C, Fortran, etc. A modo de ejemplo, los programas escritos en MATLAB pueden ser traducidos a lenguaje C de forma inmediata. En este curso nos basaremos en el uso del software básico de MATLAB.

Octave o GNU Octave es un software libre para realizar cálculos numéricos que también posee un lenguaje de programación propio. Su versión 1.0 apareció en 1994. Es considerado el equivalente libre de MATLAB, ya que aunque existen ciertas diferencias sintácticas, la mayoría son debidas a que Octave es más amplio y admite alguna sintaxis que MATLAB no permite. Luego si se utiliza sintaxis MATLAB los programas van a funcionar en ambos software.

Para seguir este curso, se puede utilizar indistintamente el software MATLAB u Octave. Hablaremos en este curso de lenguaje M para referirnos, indistintamente, al lenguaje que se utiliza en ambos software.

- **Instalación del software necesario para el seguimiento del curso**

MATLAB es un software comercial, por tanto, con coste. El alumno que lo tenga instalado en su computador puede seguir el curso con este programa.

Sin embargo, el curso también puede seguirse usando el software libre, gratuito, Octave. Este software se puede descargar desde (<https://www.gnu.org/software/octave/download.html>). En el vídeo explicativo correspondiente a este tema se indica la forma en la que se debe realizar la instalación de éste.

Para continuar con el curso el alumno debe tener instalado en su computador, bien el software Octave, o bien el software MATLAB (en cualquiera de las versiones).

▪ El entorno de desarrollo integrado en MATLAB y Octave

Para poder escribir y ejecutar un programa en un lenguaje de programación se necesita tener instalado en el computador un compilador o intérprete para tal lenguaje. MATLAB y Octave utilizan un intérprete.

El intérprete no se usa de manera aislada. Suele incluirse en Entornos de Desarrollo Integrados (IDE, por sus siglas en inglés). El entorno que vamos a usar cuenta con:

- Editor de código fuente (*Editor*)
- Explorador de variables (*Workspace*).
- Línea de comandos del intérprete (*Command Window*).
- Historial de comandos (*Command History*)
- Explorador de ficheros (*Current Folder*)
- Depurador (*Debug*)
- Otras herramientas

Sea cual sea la versión de MATLAB que utilicemos o si estamos usando Octave lo verdaderamente importante es saber que existen estas herramientas y utilizarlas para potenciar y acelerar nuestro trabajo.

A continuación indicamos la función principal de las ventanas y/o herramientas principales del IDE:

- *Editor*, es el editor de texto en el que se escribirá el código fuente de los programas, además de poder visionar cualquier archivo de texto plano (texto sin formato).

- *Workspace*, guarda la información de las variables utilizadas en la sesión de trabajo actual.

- *Command Window*, es la ventana más importante, ya que en ella aparece la línea de comandos del intérprete, en la que se teclean las instrucciones a ejecutar para obtener un resultado de inmediato.

. - *Command History*, almacena todas las sentencias que se han ejecutado en la ventana de comandos en las últimas sesiones de trabajo. Se puede navegar a través de ellas mediante la flecha de desplazamiento vertical pulsada desde *Command Window*.

- *Current Folder* indica cuál es el directorio de trabajo y los archivos y/o directorios incluidos en él.

- *Debug*, herramienta que permite actuar en la ejecución del programa mediante múltiples acciones: insertar puntos de parada en el código, ejecutar el código paso a paso, etc.

A continuación se muestra el interfaz que encontramos en las versiones MATLAB R2015a y Octave 4.0.0.

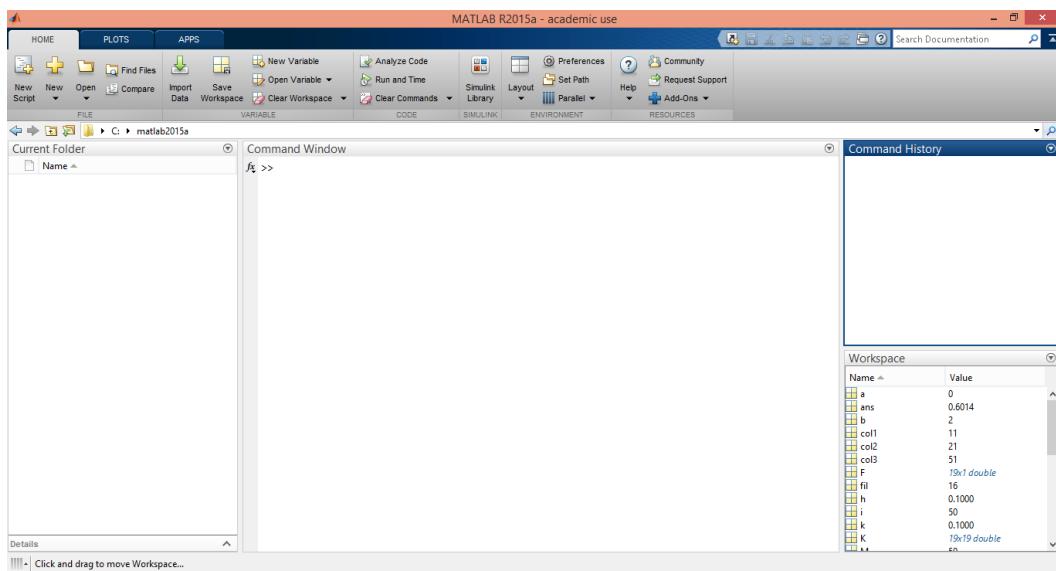


Figura 1.1 Interfaz de MATLAB R2015a

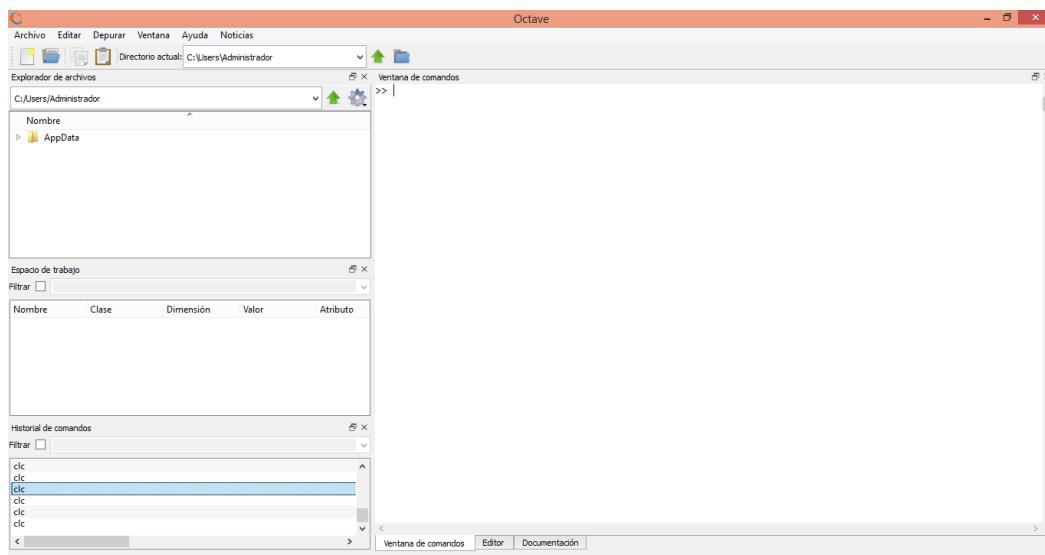


Figura 1.2 Interfaz de Octave 4.0.0

1.2 Uso del software como una calculadora. Iniciación a los operadores y almacenamiento de datos.

▪ Iniciación al manejo de datos

Empezaremos a usar el software en modo calculadora, utilizando la ventana de comandos para escribir la instrucción y observando el resultado.

```
>> 2+3  
ans =  
5
```

Todo resultado de una operación se debe almacenar en la memoria del computador. Si no se indica explícitamente el nombre de la variable donde se guardará éste, el lenguaje M usa la variable reservada *ans*.

Podemos por tanto interpretar que el resultado de la operación $2+3$ se ha almacenado en la variable *ans*.

Cuando el programador use sus propias variables debe utilizar nombres válidos. Como identificador (nombre) se puede utilizar cualquier palabra que empiece por una letra y tenga a continuación cualquier combinación de letras, números o el carácter guión bajo. Es importante conocer que se distinguen mayúsculas de minúsculas, luego, por ejemplo, las variables *A* y *a* son distintas.

Otra cosa importante es saber cómo introducir los valores numéricos en las sentencias de este lenguaje. Los valores numéricos enteros se escriben literalmente, y los valores reales, usando un punto para separar la parte entera de los decimales o también utilizando notación científica. La notación científica es una forma de representar un número utilizando potencias de 10. Por ejemplo, el valor 34.2865 se podría escribir en notación científica como 0.342865×10^2 (se escribirá como 0.342865e2). Veamos un ejemplo en el que asignamos un valor entero a la variable *a*, mientras que a las variables *c* y *d* se les asignan valores reales.

```
>> a=23, c=45.876, d= 1.453e-2
```

M tiene variables predefinidas que contienen un valor por defecto. Por ejemplo, entre otras, existe la variable *pi* con el valor correspondiente a la constante matemática π .

Existen algunas variables que representan cantidades que no son consideradas números como tal. Estas cantidades son:

- Infinito, representado por la variable *Inf* (o *inf*)
- Cantidades indefinidas, representadas por la variable *NaN* (o *nan*)

La cantidad infinito se genera por un desborde en las operaciones o por una división entre cero. El valor *NaN*, que es una abreviación de la frase en inglés '*Not a Number*', es obtenida como resultado de operaciones aritméticas indefinidas tales como $0/0$ ó $\infty - \infty$.

En los ejemplos anteriores aparecen ciertos operadores que conviene analizar con detalle.

▪ Operador de asignación

El operador de asignación ($=$) se utiliza para dar un valor nuevo a una variable. Su utilización es la siguiente:

variable = expresión

donde expresión es cualquier expresión válida en lenguaje M.

Si la variable no existe, ésta se creará con el valor y el tipo de dato de la expresión situada a la derecha del operador de asignación.

Si la variable existe, cada vez que se le asigne un nuevo valor, ésta pierde su estado anterior (tipo de dato y valor) y toma un nuevo estado correspondiente con el valor y el tipo de la expresión de la derecha.

▪ Operadores aritméticos

Los operadores aritméticos aplicables a escalares (ampliaremos éstos cuando se estudien matrices) son los siguientes:

Suma: +

Resta: -

Producto: *

División derecha: /

División izquierda: \

Potenciación: ^

Resto de una división: no existe un operador. A tal efecto se debe utilizar la función rem(A, B) que calcula el resto de dividir A entre B.

Para desarrollar una sentencia en la que aparezcan varios operadores, las operaciones se deben realizar siguiendo la prioridad de operadores. La siguiente tabla muestra la prioridad de operadores aritméticos y asignación de mayor a menor prioridad.

Operador	Símbolos	A igual prioridad se sigue el orden de
Paréntesis	()	
Potencia	^	Izquierda a derecha
Multiplicación, división	* , / , \	Izquierda a derecha
Suma y resta, y cambio de signo (-)	+ , -	Izquierda a derecha
Asignación	=	Derecha a izquierda

Ejemplos:

```
>> 2.4*6  
ans=14.4000  
  
>> 2^3  
ans=8  
  
>> c=-1^4  
c=-1  
  
>> c=(-1)^4  
c=1  
  
>> 3/4  
ans= 0.7500  
  
>> 3\4  
ans=1.3333  
  
>> 2/3^2  
ans=0.2222  
  
>> x=2/3^2  
x=1.3333  
  
>> b=rem(6.7,5)  
b=1.7000
```

▪ Formatos para el muestreo de resultados en pantalla

Independientemente de la precisión con la que un dato se ha memorizado en la memoria RAM, podemos elegir entre diferentes formatos de escritura de los datos en pantalla. Existen varios modos de trabajo en MATLAB y Octave. Se indican a continuación, los más significativos:

`[format short]` punto fijo con cinco cifras significativas¹. Si el dato no puede mostrarse adecuadamente con este formato opta por un formato en coma flotante.

`[format long]` punto fijo con quince cifras significativas. Si el dato no puede mostrarse adecuadamente con este formato opta por un formato en coma flotante.

`[format short e]` formato en coma flotante. Para representar la mantisa se usan cinco cifras significativas.

¹ Cantidad de cifras con las que se muestra un número, sin contabilizar los ceros a la izquierda en números reales y los ceros a la derecha en números enteros.

`format long e` formato en coma flotante. Para representar la mantisa se usan quince cifras significativas.

`format rat` aproximación por formato racional (cociente de enteros)

El modo de trabajo por defecto en lenguaje M es `format short` luego los datos se mostrarán con cinco cifras significativas mientras que el usuario no quiera modificar el formato.

A continuación se utiliza MATLAB, como si de una calculadora se tratara, para calcular los valores 3^{100} y $5+\pi$, cambiando el formato de muestreo de resultados. (Adelantamos aquí el uso del operador potenciación (^)).

```
>>3^100
ans=5.1538e+047

>>5+pi
ans=8.1416

>>format short e
>>3^100
ans=5.1538e+047

>>5+pi
ans=8.1416 e+000

>>format long
>>3^100
ans=5.153775207320113e+047

>>5+pi
ans=8.14159265358979

>>format long e
>>3^100
ans=5.153775207320113e+047

>>5+pi
ans=8.141592653589793e+000

>>format rat
>> 5+pi
ans=920/113
```

Tecleando `format` se vuelve al formato por defecto, es decir, `format short`.

Usemos ahora la ventana de comandos para ver un ejemplo de los errores en el almacenamiento de datos. Escribamos en la ventana de comandos el número 2, y a continuación el resultado de la operación $(\sqrt{2})^2$ (Adelantamos aquí el uso de la función raíz cuadrada (`sqrt`)). Mostremos 15 cifras significativas.

```
>> 2
ans =
2
>> format long
>> (sqrt(2))^2
ans =
2.000000000000000
```

Matemáticamente ambos resultados son iguales y computacionalmente parecen iguales. Vamos a comprobarlo:

```
>> 2 - (sqrt(2))^2
ans =
-4.440892098500626e-016
```

El resultado debería ser cero, pero como el ordenador ha tenido que redondear $\sqrt{2}$ ya que tiene infinitos decimales, arrastra este error y la diferencia resulta distinta de cero. Se observa que hay diferencia a partir del decimal número 16, puede parecer un error pequeño, pero si no se tiene en cuenta, puede ocurrir que:

- si estamos comparando dos cantidades, la diferencia nunca puede ser cero y el programa fallará.
- si estamos realizando un cálculo de ingeniería, se puede producir una propagación y acumulación de errores que dé lugar a un resultado final con un error inaceptable.

Podemos averiguar cuál es la precisión relativa en punto flotante usando el comando `eps`.

```
>> eps
ans = 2.220446049250313e-16
```

Esto significa que si tenemos el valor 1 guardado, el siguiente valor que podemos almacenar es $1 + \text{eps}$. Es debido a que en coma flotante y doble precisión se tienen 52 bits para la mantisa, siendo $\text{eps} = 2^{-52}$.

1.3 Tipos de datos elementales, operadores y comandos utilitarios

▪ Tipos de datos elementales

M es fundamentalmente un lenguaje para cálculo matricial. Todos los datos que maneja son matrices, pudiendo también trabajar con vectores y escalares, pero considerando a éstos casos particulares de matrices: un vector fila es una matriz de una fila; un vector columna, es una matriz de una columna; un escalar es una matriz de una fila y una columna.

El lenguaje M, en general, trabaja con datos numéricos, utilizando, por defecto, para su almacenamiento doble precisión (tipo de dato `double`), es decir, manejando 8 bytes de memoria para cada dato, con 15 cifras significativas en punto flotante. También puede trabajar con otros tipos de datos como enteros, reales de simple precisión, datos lógicos, cadenas de caracteres y con tipos de datos más avanzados: hipermatrices, estructuras, matrices de celdas y clases y objetos.

Combinando el tipo de dato por defecto (`double`) junto con datos lógicos y de carácter podría realizarse casi cualquier programa, por ello, son los tipos de datos que manejaremos en este curso.

▪ Datos numéricos `double`

Las variables reales en doble precisión (tipo `double`), son las variables por defecto que maneja el lenguaje M. Utilizan 8 bytes en memoria para almacenar el dato. No hay que hacer ninguna declaración, simplemente se usan. Los valores máximos y mínimos que se pueden almacenar son `1.7977e+308` y `2.2251e-308`. Sin embargo, la precisión de los cálculos no es superior a 15 cifras significativas.

▪ Datos de tipo lógico

Es habitual al trabajar con cualquier lenguaje de programación que aparezcan datos de contenido lógico (`true` o `false`). Estos datos se manejan en lenguaje M con los valores 1 o `true` (cierto) y 0 o `false` (falso).

Lo normal es que estos datos se generen automáticamente como resultado de ciertas operaciones.

Si un valor no lógico se tiene que evaluar como lógico se hará la transformación de la siguiente manera:

- Cualquier dato distinto de 0 equivale a cierto (1).
- Cualquier dato igual a cero equivale a falso.

▪ Cadenas de caracteres

Además de datos numéricos es necesario manejar texto. M puede definir variables que contengan cadenas de caracteres, para ello las cadenas de texto se deben delimitar entre apóstrofos o comillas simples. En el siguiente ejemplo se define la variable `s` que contiene un texto.

```
s = 'Esto es una cadena'
```

M maneja la cadena como una matriz de una fila (vector fila) distinguiendo cada carácter como si fueran distintas componentes del vector. Así, en este caso tenemos un vector de 18 componentes.

Veamos un fragmento de programa en el que se puede observar la necesidad de entrecollar un valor de carácter para diferenciarlo del nombre de una variable.

```
a=7;  
v=a;  
w='a'
```

Fíjense que la variable `v` contiene el valor 7 y la variable `w` el carácter 'a'.

- **Otros Operadores**
 - **Operadores relacionales y de igualdad**

Relacionan dos valores numéricos y dan como resultado un valor lógico (cierto (1) o falso (0)). Se muestran en la siguiente tabla:

Menor que	<
Mayor que	>
Menor o igual que	<=
Mayor o igual que	>=
Equivale a	==
Distinto de	~=

Ejemplos de utilización de estos operadores:

```
>> u=3.5;  
>> v=7.3;  
>> w=7.3;  
>> u<v  
ans =  
    1  
>> v>w  
ans =  
    0
```

```

>> v>=w
ans =
1

>> u~=v
ans =
1

>> v==w
ans =
1

```

— Operadores lógicos

Se aplican a valores lógicos resultando otro valor lógico. Son los siguientes:

- Operador lógico OR: se representa con el carácter `|`. Se utiliza entre dos valores lógicos dando como resultado cierto, si ambos o uno de ellos son ciertos. Sólo si los dos son falsos resulta falso. La función equivalente es `or (A, B)` .

C=A B		
A	B	C
1	1	1
1	0	1
0	1	1
0	0	0

Ejemplo: si la calificación de un examen se guarda en la variable `nota`, la expresión que resulta cierta si la nota es errónea es: `nota<0 | nota>10`.

- Operador lógico AND: se representa con el carácter `&`. Se utiliza entre dos valores lógicos dando como resultado *cierto* sólo si ambos son *ciertos*. Si uno o los dos son *falsos* el resultado es *falso*. La función equivalente es `and (A, B)` .

C= A&B		
A	B	C
1	1	1
1	0	0

0	1	0
0	0	0

Ejemplo: si la calificación de un examen se guarda en la variable nota, la expresión que resulta cierta si la nota es igual a notable es: nota<9¬a>=7

- Operador lógico OR EXCLUSIVO: se utiliza con la sintaxis xor (A, B) , resultando cierto cuando A o B son ciertos, pero no ambos.

C=xor (A, B)		
A	B	C
1	1	0
1	0	1
0	1	1
0	0	0

Ejemplo: supongamos que un alumno realiza dos pruebas cuyas calificaciones se guardan en las variables nota1 y nota2, y se ofrece una recuperación a los alumnos que hayan suspendido sólo una de las dos, la expresión que resulta cierta en ese caso sería: xor (nota1<5, nota2<5)

- Operador lógico NOT: se representa con el carácter ~. Actúa sobre el valor lógico situado a su derecha, resultando el valor lógico contrario a éste. La función equivalente es not(A) . Suele utilizarse para hacer referencia a una condición contraria a la que se esté estudiando.

C=~A	
A	C
1	0
0	1

Ejemplo: si la calificación de un examen se guarda en la variable nota, la expresión que resulta cierta si la nota es suspensa es: ~(nota>=5)

- Operadores lógicos BREVES (&&) y (||) : son realmente el AND y el OR pero permiten simplificar la operación. Veamos las diferencias:

$A \& B$: estudia siempre las condiciones A y B.

$A \& \& B$: estudia la condición A, si es cierta estudia la B, pero si es falsa ya no estudia la B, porque el resultado final se conoce que es falso.

$A | B$: estudia siempre las condiciones A y B.

$A | | B$: estudia la condición A, si es falsa estudia la B, pero si es cierta ya no estudia la B, porque el resultado final se conoce que es cierto.

Esto además de simplificar los cálculos se puede utilizar para evitar posibles errores. Por ejemplo, en la expresión

$c = a \sim= 0 \& \& b / a > 5,$

si el valor de la variable a es cero, ya no se desarrolla la segunda expresión y así se evita el error que supondría ésta (división entre cero).

— Prioridad de operadores

En la tabla siguiente se muestra el orden de prioridad todos los operadores conocidos hasta el momento. Se han ordenado de mayor a menor prioridad (de arriba hacia abajo).

Operador	Símbolos	A igual prioridad se sigue el orden de
Paréntesis, or exclusivo	(), xor()	
No lógico	\sim	Derecha a izquierda
Potencia	$^$	Izquierda a derecha
Multiplicación, división	$*$, $/$, \backslash	Izquierda a derecha
Suma y resta, y cambio de signo (-)	$+$, $-$	Izquierda a derecha
Operador :	:	Izquierda a derecha
Relacionales	$<$, $<=$, $>$, $>=$	Izquierda a derecha
Igualdad	$= =$, $\sim =$	Izquierda a derecha
Y lógico	$\&$	Izquierda a derecha
O lógico	$ $	Izquierda a derecha
Asignación	$=$	Derecha a izquierda

- **Comandos utilitarios en el manejo del lenguaje M**
 - **Almacenamiento del texto de la ventana workspace**

Si se desea almacenar en un fichero de texto las entradas que se van a teclear en la ventana de comandos y las salidas ofrecidas por el programa, lo más sencillo es utilizar el comando *diary*.

Tecleando

```
diary nombrefichero
```

se crea el fichero con el nombre indicado en la ubicación actual almacenándose en él el contenido que se inserte en la ventana *workspace* a partir de ese momento. Si no se indicara nombre de fichero, se crearía el archivo de nombre *diary* para tal fin. Si en un momento de la sesión se quiere desactivar la grabación en el fichero, se escribirá

```
diary off
```

cuando quiera reanudarse la grabación, se tecleará

```
diary on
```

También se puede pasar del estado *on* al *off*, o viceversa, escribiendo simplemente *diary*.

- **Almacenamiento de variables en ficheros binarios**

Cuando se finalice una sesión de trabajo, para evitar perder los datos obtenidos (variables) se pueden guardar éstos (con sus valores) dentro de un fichero binario, que puede ser cargado en otro momento para continuar con el trabajo anterior. Esta operación se puede realizar utilizando el comando *save*. A continuación se ejemplifican diferentes opciones de utilización.

Si se quieren guardar todas las variables del espacio de trabajo en el fichero de nombre *guardavariables*, se utilizaría:

```
save guardavariables
```

Si sólo interesa guardar algunas variables, por ejemplo, las variables de identificadores *w* y *z*, se procede así:

```
save guardavariables w z
```

Para cargar de nuevo las variables guardadas en un fichero binario se utiliza el comando *load* de la forma siguiente:

```
load guardavariables
```

- **Otros comandos útiles**
 - Comentarios en el código

Los comentarios en el código fuente de un programa se añaden para hacer el código más entendible al programador de cara a futuras utilizaciones o a compartir el código con terceros. Se trata de texto que se incrusta en el código pero que es ignorado por el compilador o intérprete.

En lenguaje M, para insertar un comentario debemos preceder éste por el símbolo %. En el siguiente ejemplo, la primera línea es un comentario, también aparece un comentario después de la instrucción de la tercera línea.

```
% radio y altura del cilindro  
r=7.3;h=25.4;  
a=pi*r^2*h; %volumen del cilindro
```

- Continuación de una sentencia en otra línea

En ocasiones la escritura de una sentencia queda demasiado larga y conviene continuarla en la línea posterior. Para ello se debe terminar la línea que continuará en la siguiente con tres puntos. Veamos un ejemplo:

```
>> b=7;c=6;  
>> x=b+c^3-c*b...  
+4/b  
x =  
181.5714
```

Sin embargo no se permite realizar este proceso en una cadena de caracteres. El siguiente ejemplo produciría un error:

```
>>x='No es posible continuar un texto...  
en la siguiente linea'
```

- Eliminación de variables del espacio de trabajo

Si se desea eliminar todas las variables del espacio de trabajo, se utiliza el comando **clear**. Para eliminar únicamente algunas variable se emplea el mismo comando seguido de los nombres de las variables a eliminar. Por ejemplo la siguiente instrucción borra las variables **a**, **b** y **c**.

```
-----  
clear a b c  
-----
```

- Limpieza de texto de la ventana de comandos

Para eliminar todo el texto escrito en la ventana de comandos se utiliza el comando **clc**. La ejecución de esta orden no afecta a las variables de la sesión de trabajo (la ventana **workspace** sigue manteniendo las variables).

- Directorio de trabajo

Para conocer la ruta al directorio de trabajo actual se utiliza el comando **pwd**.

Para obtener un listado de los ficheros que existen en la carpeta actual, se utiliza el comando `dir`.

Para cambiar de carpeta de trabajo se utiliza el comando:

```
cd nombrenuevacarpeta.
```

– Utilización de la ayuda

Para obtener ayuda sobre la utilización de un comando o función determinada se utiliza la sentencia:

```
help nombredelcomando
```

Para buscar un texto que esté contenido en la primera línea de comentario de los programas se utiliza el comando `lookfor` de la forma:

```
lookfor 'texto a buscar'
```

– Tiempo que tarda un cálculo en ser efectuado

Cuando se implementa un algoritmo en un lenguaje de programación es habitual que se pueda obtener la solución esperada mediante diferentes códigos fuente. Una forma de comparar los códigos y elegir cuál resulta más conveniente es calcular el tiempo de ejecución. El lenguaje M cuenta con las funciones `tic` y `toc` que trabajan juntas y calculan el tiempo transcurrido entre la ejecución de la primera y la segunda.

`tic`: activa un contador temporal en segundos que finaliza al utilizar el comando `toc`.

`toc`: devuelve el tiempo transcurrido en segundos desde que se activó el contador con `tic`.

Veamos unos ejemplos que muestran dos formas de utilización. En ambas se almacena el tiempo de ejecución en la variable `tiempo`:

```
tic
suma=0;
for i=1:1000
    suma=suma+i;
end
tiempo=toc
```

```
inicio=tic
suma=0;
for i=1:1000
    suma=suma+i;
end
tiempo=toc(inicio)
```

1.4 Ficheros m. Entrada y salida de datos por consola.

▪ Ficheros m

Hasta el momento todas las órdenes del lenguaje M las hemos ejecutado desde la ventana de comandos. De esta manera obtenemos una respuesta inmediata a la instrucción.

Sin embargo si queremos realizar un proceso formado por una secuencia de instrucciones M que vamos tecleando una a una en la ventana de comandos:

- ¿qué ocurre si cometemos una imprecisión en una instrucción a lo largo del proceso? Habría que volver a realizarlo de nuevo, orden a orden, escribiendo esta vez sin cometer errores.
- ¿qué ocurre si se quiere volver a repetir este proceso en el futuro? Habría que volver a realizarlo de nuevo, ya que las instrucciones de la ventana de comandos no se pueden guardar como tal.

Estamos de acuerdo que esta forma de actuar no es eficiente. Para resolver estos problemas existen los ficheros M. Estos ficheros no son más que archivos de texto en los que se escriben las instrucciones, en lugar de en la ventana de comandos, que quedan guardados de forma permanente, y después pueden ser ejecutados por MATLAB u Octave.

Los programas en M se escriben en ficheros de extensión .m (ficheros M) con un editor especial situado dentro del entorno de MATLAB u Octave, aunque es válido cualquier editor de texto no formateado, almacenándose en disco con un nombre cualquiera, por ejemplo, *programa.m*.

Para ejecutar el programa desde la ventana de comandos simplemente se debe teclear el nombre del fichero M que contiene el programa (sin la extensión). En el caso anterior sería:

```
>> programa
```

Entonces, se van ejecutando de manera secuencial todas las órdenes escritas en el fichero M, obteniendo los resultados en la ventana de comandos.

A modo de ejemplo, el alumno puede probar a teclear una a una las siguientes instrucciones en la ventana de comandos:

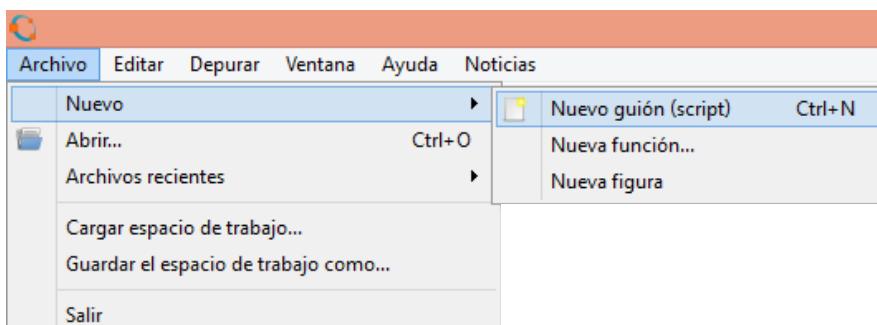
```
>>A=3  
>>c=A^3  
>>B=A+c
```

Imaginemos ahora que la variable A se ha tecleado incorrectamente, ya que queríamos escribir

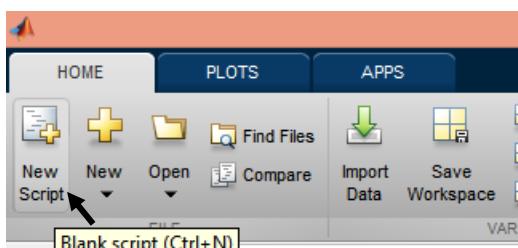
```
>> A=13
```

¿Cómo se resolvería esto desde la ventana de comandos? Evidentemente habría que volver a teclear las tres órdenes.

Probemos ahora a generar un archivo .m. Por ejemplo, desde Octave, siguiendo esta secuencia:



O desde MATLAB 2016:



Escribamos las tres instrucciones en él:

```
A=3  
c=A^3  
B=A+c
```

y guardémoslo como *ejemplo1.m*

Se ejecuta desde la ventana de comandos:

```
>> ejemplo1
```

obteniéndose el resultado.

En este caso la corrección del error en la variable A se resolvería corrigiendo el valor correspondiente en el fichero y volviendo a ejecutar éste. Además queda grabado en disco, luego puede ejecutarse en cualquier otro momento.

Podemos decir que hemos realizado nuestro primer programa M.

A partir de este momento se trabajará con programas, es decir escribiendo las instrucciones en ficheros M. Poco a poco se irá enriqueciendo el contenido de estos programas.

El objetivo del resto de este tema es conseguir que nuestros programas puedan tomar, en tiempo de ejecución, valores que introduce el usuario por teclado (entrada de datos) y ofrecer respuestas a través de la ventana de comandos.

■ Entrada de datos por teclado

La función `input` permite introducir datos en un programa cuando está en modo de ejecución. La utilización es como sigue:

```
v=input('Texto de petición del dato')
```

`input` realiza dos tareas:

- 1) hace que aparezca en pantalla la cadena de caracteres que lleva como argumento.
- 2) espera a que se tecleen los datos como respuesta al texto y los memoriza en la variable `v`.

`input` no permite disponer varias variables a la izquierda de la asignación, luego todos los datos que se introduzcan deben constituir una única variable con sintaxis matricial. Ejemplo:

```
>> P=input('Introduce el radio del círculo: ')  
Introduce el radio del círculo: 1.47  
P =  
1.4700
```

Si como respuesta a `input` se quiere introducir una cadena de caracteres, el usuario del programa debe introducirla entre apóstrofos. El olvido de éstos provoca el fallo del programa.

```
>> Nombre=input('Introduce tu nombre y apellidos: ')  
Introduce tu nombre y apellidos: Pedro Perez  
error: 'Pedro' undefined near line 1 column 1
```

Para prevenirlo, se usa la función `input` con un segundo argumento '`s`' que hace que el dato introducido se tome como una cadena de caracteres sin necesidad de delimitarlo por los apóstrofos.

```
>>Nombre=input('Introduce tu nombre y apellidos: ','s')  
Introduce tu nombre y apellidos: Juan Pérez González  
Nombre =  
Juan Pérez González  
%La variable Nombre contiene una cadena de caracteres
```

A veces un programa puede fallar porque el usuario no responda a la petición del dato y pulse simplemente la tecla `enter`. Para detectarlo puede ser de utilidad la función `isempty` que devuelve cierto si la variable que tiene como argumento está vacía. Véase el ejemplo en el módulo 3 -comandos repetitivos-.

- **Salida de datos por pantalla**

Para que un programa en modo de ejecución pueda escribir textos por pantalla, se puede utilizar la función:

```
disp('Mensaje')
```

que escribe en pantalla la cadena de caracteres que tiene como argumento.

Para escribir el valor de una variable, se utiliza de esta forma:

```
disp(v)
```

que muestra en pantalla el valor de la variable v.

Ejemplos:

```
>> z=2;  
>> disp(z)  
2  
>> disp('Escritura en pantalla')  
Escritura en pantalla
```

Con la función `disp` sólo se puede escribir una cadena de caracteres o una matriz (vector o escalar). Además el lenguaje M lo escribirá con formato libre, es decir, el usuario no puede seleccionar otro formato, ni realizar la escritura de una combinación de texto y datos. Para poder realizar estas acciones, se dispone de la función `fprintf`.

La función `fprintf` escribe en pantalla una combinación de datos y/o texto con el formato elegido por el programador.

Veamos la utilización general de la función `fprintf`

Para la escritura solo de texto se utiliza de igual manera que `disp`, aunque en este caso se debe terminar el texto con un salto de línea `\n`:

```
fprintf('texto\n')
```

Para escritura de texto y datos en pantalla, o sólo datos, se utiliza la función con la siguiente sintaxis

```
fprintf('formato\n', variables)
```

donde:

`variables` será la lista de variables a escribir

formato será la especificación del formato de escritura de las variables así como el texto que se quiera intercalar entre ellas. Además, también se podrán insertar los siguientes caracteres de control, entre otros:

\n: salto de línea

\r: retorno de carro al comienzo de la línea

\t tabulación horizontal

\b: espacio hacia atrás

Los formatos más utilizados para escribir variables son:

%d : adecuado para datos enteros, lógicos y para reales con decimales igual a cero. Escribe el dato como un entero. Si el dato a escribir es un real con su parte decimal no nula, no se trunca el número, se escribe con sus decimales y en formato de punto flotante.

%f : escribe cualquier dato numérico como un real con 6 decimales

%s : escribe cadenas de caracteres como tal. Si se emplean los formatos anteriores para escribir una cadena se imprimen todos los códigos ASCII de los caracteres que la forman.

Para clarificar lo explicado se ofrecen a continuación variados ejemplos:

```
>> a=2;  
  
>> fprintf('El dato es %d\n',a)  
  
El dato es 2  
  
>> fprintf('%f\n',a);  
  
2.000000  
  
>> b=3.2;  
  
>> fprintf('%d\n',b);  
  
3.200000e+000
```

En el último ejemplo, el dato no es escrito como un entero ya que para ello, se debería truncar su valor. Es escrito en formato de punto flotante. Veamos otros ejemplos:

```
>> fprintf('los resultados son %d y %f\n',a,b);  
  
los resultados son 2 y 3.200000  
  
>> fprintf('Dato 1: %d\nDato 2: %f\n',a,b);
```

```
Dato 1: 2  
Dato 2: 3.200000  
  
>> fprintf('%s','Error en el programa');  
Error en el programa
```

Cuando no se especifiquen formatos suficientes para la escritura de todos los datos, el formato se reutiliza desde el principio las veces necesarias. Por ejemplo, a continuación se pretende escribir en pantalla el valor de tres variables, sin embargo solo aparecen dos formatos; obsérvese cómo se reutiliza el formato desde el principio hasta justo antes del que no se necesita, para poder escribir el tercer dato.

```
>>a=3; b=2; c=7.3;  
>>fprintf('los resultados son %d y %d\n',a,b,c);  
los resultados son 3 y 2  
los resultados son 7.300000e+00 y >>
```

■ **Modificadores de formato en escritura de datos**

Cuando se escriben datos en pantalla con la función `fprintf`, se puede modificar el aspecto por defecto con el que se muestran mediante el uso de modificadores de formato. Su utilización se indica a continuación:

- Para datos escritos con formato `%f`, se puede incluir la siguiente información adicional en el formato:

%-n.mf

siendo:

`n`, el número mínimo de espacios utilizados en la escritura (anchura mínima de campo).

`m`, el número de decimales con los que se escribirá el dato.

signo: si se incluye el signo negativo se obliga a la justificación izquierda en el campo; si no se escribe, la justificación es a la derecha.

Veamos unos ejemplos de utilización en los que, para clarificar el formato impreso, se ha usado el símbolo para identificar el espacio en blanco.

```

>>x=56.45;
>>fprintf('%7.3f es el valor calculado\n',x)
 56.450 es el valor calculado

>>fprintf('%10.1f es el valor calculado\n',x)
 56.5 es el valor calculado

>>fprintf('-%-10.1f es el valor calculado\n',x)
 56.5 es el valor calculado

```

- Para datos escritos con formato %d, también se puede añadir la misma información adicional en el formato:

% -n.m f

pero en este caso, m no puede indicar el número de decimales sino que fija el número mínimo de dígitos a escribir. Ejemplos:

```

>>y=5;
>> fprintf('-%10.5d es el valor calculado\n',y);
 00005 es el valor calculado

>>y=236;
>> fprintf('%6.2d es el valor calculado\n',y);
 236 es el valor calculado

>>y=236;
>> solucion=sprintf('%6.2d es el valor calculado',y)
solucion=
 236 es el valor calculado

```

- Para datos escritos con %s

Utilizaremos la anchura de campo y el signo de manera similar a los casos anteriores.

```
>>fprintf('%7s\n', 'Hola', 'Mundo')  
      Hola  
      Mundo
```

1.5 Practicando con programas(I). Ejercicios de autoevaluación

En esta lección se pretende que el alumno simplemente practique realizando los ejercicios que se proponen a continuación. Después de los enunciados de los ejercicios se encuentran las soluciones para que en caso de dificultad, en estos primeros acercamientos a la programación, se puedan analizar los programas bien planteados.

- **Ejercicio 1.5.1** Escribir un programa M que pida al usuario el radio de un círculo y escriba en pantalla el área del mismo, que se mostrará con tres decimales, de la forma:

El área es xx.xxx

- **Ejercicio 1.5.2** Escribir un programa M que calcule el área de un trapecio. Los datos de entrada serán: lado superior, base y altura. El programa debe leer los datos por teclado y producir como salida a pantalla (con dos cifras decimales):

El área del trapecio es xxxxxxxxx.xx

- **Ejercicio 1.5.3** Escribir un programa M que pida al usuario su edad, almacenándola en la variable edad, el año actual, almacenándolo en la variable fecha_act y su nombre, almacenándolo en la variable de cadena nombre. Y que para un ejemplo de datos introducidos

27

2016

Pedro

produzca la siguiente salida de resultados:

Pedro, naciste en 1989

Solución del ejercicio 1.5.1 El programa se ha guardado en el fichero M de nombre areacirculo con el siguiente contenido:

```
r=input('Introduce el valor del radio: \n');
area=pi*r*r;
fprintf('El área es %.3f\n',area)
```

Ejecución del programa desde la ventana de comandos:

```
>> areacirculo
Introduce el valor del radio:
4
```

El área es 50.265

▪ **Solución del ejercicio 1.5.2.** Una solución válida es la siguiente:

```
base=input('Introduce base del trapecio: ');
altura=input('Introduce la altura del trapecio: ');
ladosup=input('Introduce el lado superior del trapecio: ');
area=(base+ladosup)*altura/2;
fprintf('el área del trapecio es %.2f\n',area)
```

▪ **Solución del ejercicio 1.5.3.** Una solución válida es la siguiente:

```
edad=input('Introduce tu edad: ');
fecha_act=input('Introduce el año actual: ');
nombre=input('Introduce tu nombre: ','s');
fprintf('%s, naciste en %d\n',nombre,fecha_act-edad)
```

2.1 Generación y manejo de vectores y matrices.

▪ Generación de vectores de forma directa

En el lenguaje M se va a trabajar habitualmente con los corchetes ([]) para la generación de vectores y matrices. Su uso implica un ensamblado de los elementos que aparecen en su interior para formar una construcción más compleja. Hacemos notar que no tiene el significado matemático similar a los paréntesis.

Para definir un vector no hace falta establecer de antemano su tamaño (de hecho, éste cambia de forma dinámica cuando es preciso). Simplemente, se disponen los valores de los elementos que lo van a componer entre corchetes, separados por espacios o una coma, en el caso de un vector fila, o por el carácter punto y coma (;) o pulsaciones intro, en el caso de un vector columna.

Al teclear

```
>>b=[1 2 3 4 5]
```

o bien

```
>>b=[1,2,3,4,5]
```

se genera el vector fila b: 1 2 3 4 5,

que aparecerá como tal en la ventana *Workspace*.

Mientras que:

```
>>c=[1;2;3]
```

o bien

```
>>c=[1  
2  
3]
```

genera el vector columna c:

```
1  
2  
3
```

▪ Generación rápida de vectores. Operador (:)

Se van a analizar a continuación otras formas de generación de vectores que no necesitan de la escritura explícita de todos sus elementos. Implícitamente se trabajará con el operador (:) que sirve para crear sucesiones numéricas, por ejemplo:

1:4, es equivalente a 1,2,3,4

```
variable=[vin:vfin]
```

Define el vector cuyos primer y último elemento son los especificados por vin y vfin, estando los componentes intermedios separados por una unidad. Está permitido no utilizar los corchetes o sustituirlos por paréntesis.

```
variable=[vin:incr:vfin]
```

Define el vector cuyos primer y último elemento son los especificados por vin y vfin, estando los componentes intermedios separados por incr. Está permitido no utilizar los corchetes o sustituirlos por paréntesis.

```
variable=linspace (x1,x2,n)
```

Genera un vector con n valores igualmente espaciados entre x1 y x2.

Ejemplos:

```
>>v=[1:10]
v=
1 2 3 4 5 6 7 8 9 10

>>v=1:10
v=
1 2 3 4 5 6 7 8 9 10

>>v=(1:10)
v=
1 2 3 4 5 6 7 8 9 10

>>v=[1:2:10]
v=
1 3 5 7 9

>> v=linspace(1,10,7)
v =
1.0000 2.5000 4.0000 5.5000 7.0000 8.5000 10.0000
```

■ Extracción de elementos de vectores

Para extraer uno o varios elementos de un vector se debe indicar después del nombre de éste y entre paréntesis, los índices correspondientes a las posiciones a extraer. En lenguaje M la numeración de los índices comienza en 1. Se puede utilizar la partícula `end` para indicar el último elemento.

Sea el vector `x`:

`x(n)` devuelve el componente enésimo del vector `x`.

`x(end)` devuelve el último elemento del vector `x`.

`x(a:b)` devuelve los componentes del vector `x` situados entre el `a`-ésimo y el `b`-ésimo respectivamente (`a < b`).

`x(a:p:b)` devuelve los elementos del vector `x` situados entre el `a`-ésimo y el `b`-ésimo respectivamente (`a < b`), pero separados en `p` unidades.

`x(b:-p:a)` devuelve los componentes del vector `x` situados entre el `b`-ésimo y el `a`-ésimo respectivamente (`a < b`), pero separados en `p` unidades,

Ejemplos:

```
>>v=[2:2:20]

v=
2 4 6 8 10 12 14 16 18 20

>>v(7)
14

>>a=v(1:5)

a=
2 4 6 8 10

>>b=v(2:3:10)

b=
4 10 16

>>c=v(10:-3:2)

c=
20 14 8
```

■ Generación de matrices de forma directa

Se seguirá el mismo proceso estudiado en vectores. Los elementos se escribirán por filas, separando una fila de la siguiente por el carácter punto y coma (;) o pulsaciones intro. Los elementos de cada fila se separan por espacios o una coma.

Por ejemplo, el siguiente comando define una matriz A de dimensión (3x3):

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

La respuesta del programa es:

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

Sin embargo, aunque las matrices haya que introducirlas por teclado ordenadas por filas, M las memoriza de forma lineal (a modo de vector), ordenando los elementos por columnas. Por ejemplo, la matriz A del caso anterior tendría la siguiente disposición en memoria:

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

■ Generación rápida de matrices. Operador (:)

Al igual que en el caso de vectores, se puede generar los elementos de las filas sin tener que escribirlos uno a uno. Ejemplo:

```
>> A=[1:5;5:-1:1;linspace(0,11,5)]  
  
A =  
  
1.0000 2.0000 3.0000 4.0000 5.0000  
5.0000 4.0000 3.0000 2.0000 1.0000  
0 2.7500 5.5000 8.2500 11.0000
```

■ Composición de matrices

Un caso especialmente interesante es el de crear una nueva matriz componiendo como submatrices otras matrices definidas previamente. Se utiliza la misma técnica de ensamblado vista en los apartados anteriores. A modo de ejemplo, ejecútense las siguientes líneas de comandos y obsérvense los resultados obtenidos:

```

>> A=[0, 1; 1, 2];

>> B=[7, 3; 5, -8];

>> C=[A ; B]

C =

    0      1
    1      2
    7      3
    5     -8

>> D=[11; 12; 13; 14]

D =

    11
    12
    13
    14

>> X=[C, D]

X =

    0      1      11
    1      2      12
    7      3      13
    5     -8      14

```

La matriz C de tamaño 4x4 se forma por composición en vertical de las matrices A y B. La matriz X se forma por composición en horizontal de las matrices C y D. Al igual que con simples escalares, las submatrices que quieran unirse en horizontal se separan con *blancos* o *comas*, mientras que las que se ensamblen en vertical se separan entre sí con el carácter *intro* o *punto y coma*. Los tamaños de las submatrices deben de ser coherentes (igual número de filas para ensamblado horizontal e igual número de columnas para ensamblado vertical).

Se pueden realizar varias uniones en la misma sentencia colocando las parejas de corchetes adecuadamente:

```

>> X=[ [A ; B], D]

X =

    0      1      11
    1      2      12
    7      3      13
    5     -8      14

```

■ Creación de submatrices

Para extraer uno o varios elementos de una matriz se procede igual que con los vectores pero indicando las posiciones de filas y columnas de los elementos a extraer. Veamos a continuación todas las posibilidades:

`A(m, n)` define el elemento (m, n) de la matriz A.

`A(a:b, c:d)` define la submatriz de A formada por la intersección de las filas que hay entre la a-ésima y la b-ésima y las columnas que hay entre la c-ésima y la d-ésima.

`A(a:p:b, c:q:d)` define la submatriz de A formada por la intersección de las filas que hay entre la a-ésima y la b-ésima tomándolas de p en p, y las columnas que hay entre la c-ésima y la d-ésima tomándolas de q en q.

`A([a b], [c d])` define la submatriz de A formada por la intersección de las filas a-ésima y b-ésima y las columnas c-ésima y d-ésima.

`A([a b c ...], [e f g ...])` define la submatriz de A formada por la intersección de las filas a,b,c,... y las columnas e,f,g,.....

`A(:, c:d)` define la submatriz de A formada por la intersección de todas las filas de A y las columnas que hay entre la c y la d.

`A(end, c:d)` define la submatriz de A formada por la intersección de la última fila de A y las columnas que hay entre la c y la d.

`A(:, [c d e ...])` define la submatriz de A formada por la intersección de todas las filas de A y las columnas c,d,e,.....

`A(a:b, :)` define la submatriz de A formada por la intersección de todas las columnas de A y las filas que hay entre la a y la b.

`A([a b c], :)` define la submatriz formada por la intersección de todas las columnas de A y las filas a,b,c.

`A(a, :)` define la fila a-ésima de la matriz A.

`A(:, b)` define la columna b-ésima de la matriz A.

`A(:)` define un vector columna cuyos elementos son las columnas de A colocadas por orden una debajo de otra.

`A(b)` define el elemento b-ésimo al colocar una columna de la matriz a continuación de la otra.

Ejemplos:

```
>>A=[1 2 3 4;5 6 7 8;9 10 11 12;13 14 15 16];  
  
>>A(3,4)  
12  
  
>>B1=A(1:3,2:4)  
2 3 4  
6 7 8  
10 11 12  
  
>> B2=A([1 3],[2 4])  
2 4  
10 12  
  
>>B3=A(:,2:4)  
2 3 4  
6 7 8  
10 11 12  
14 15 16  
  
>>B4=A(:,[1 3 4])  
1 3 4  
5 7 8  
9 11 12  
13 15 16
```

```
>>B5=A(2:4,:)
      5   6   7   8
      9 10 11 12
     13 14 15 16

>>B6=A(3,:)
      9 10 11 12

>>B7=A(:,3)
      3
      7
     11
     15

>>B8=A(:) El resultado es el siguiente vector (en columna):
      1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16

>>B9=A(7)
      10
```

2.2 Operaciones con matrices. Funciones específicas.

▪ Operaciones con matrices mediante operadores

M puede operar con matrices⁽¹⁾ por medio de *operadores* y por medio de *funciones*.

Sean A y B dos matrices y c un escalar. Los operadores matriciales del lenguaje M son los siguientes:

- Operadores suma (+) y resta (-)

Utilizado entre matrices (siempre con el mismo tamaño) obtiene la suma/resta matricial (elemento a elemento). Utilizado entre una matriz y un escalar, suma/resta el escalar a cada elemento de la matriz. No existen los operadores (+.) y (-.).

- Operador producto (*)

Utilizado entre matrices resuelve el producto matricial. Las dimensiones de las matrices deben ser congruentes. Utilizado entre una matriz y un escalar, multiplica el escalar por cada elemento de la matriz.

- Operador producto elemento a elemento (.*)

A.*B da como resultado una matriz cuyo elemento ij es $A_{ij} \cdot B_{ij}$.

- Operador potenciación (^)

Para utilizarlo, al menos uno de los operandos debe ser un escalar y la matriz debe ser cuadrada.

A^c resuelve el producto matricial $A \cdot A \cdot A \cdot \dots \cdot A$, c veces.

c^A se computa mediante autovalores y autofunciones.

- Operador potenciación elemento a elemento (.^)

A continuación aparecen todas las posibilidades de utilización:

$A.^B$ da como resultado una matriz cuyo elemento ij es $A_{ij} \cdot B_{ij}$.

$A.^c$ da como resultado una matriz cuyo elemento ij es $A_{ij} \cdot c$.

$c.^A$ da como resultado una matriz cuyo elemento ij es $c \cdot A_{ij}$.

- Operador división (/) (\)

La utilización entre una matriz y un escalar obtiene el cociente elemento a elemento. La utilización entre matrices se verá más adelante.

- Operador división elemento a elemento (./) (.\)

A continuación aparecen todas las posibilidades de utilización:

$A./B$ da como resultado una matriz cuyo elemento ij es A_{ij} / B_{ij} .

$A.\backslash B$ da como resultado una matriz cuyo elemento ij es B_{ij} / A_{ij} .

¹ En esta sección, hablaremos de matrices de forma global, considerando también los vectores.

- Operador traspuesta (')

A' da como resultado la matriz traspuesta de A.

Algunos ejemplos:

```
>>a=[1 2 3];b=[4 5 6];c=3;

>>a+b

ans=      5    7    9

>>a.*b

ans=      4   10   18

>>a-b

ans=     -3   -3   -3

>>a.^b

ans=      1   32   729

>>a+c

ans=      4    5    6

>>a*c

ans=      3    6    9

>>a.^c

ans=      1    8   27

>>c.^a

ans=      3    9   27
```

- Operadores relacionales, lógicos y de igualdad

Son válidos los analizados en la sección 1.3. Aplicados entre matrices se emplean elemento a elemento, luego el tamaño de las matrices debe coincidir. El resultado es una matriz de tipo lógico.

```
>> A=1:9;

>>P= (A>2) & (A<6)

P=

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

■ Operaciones con matrices mediante funciones

Además de los operadores analizados en la sección anterior, existen funciones M que permiten realizar otro tipo de operaciones con vectores:

`dot(v,w)` producto escalar de los vectores v y w.

`cross(v,w)` producto vectorial de los vectores v y w (máximo tres componentes de cada uno): $[v_2w_3 - v_3w_2 \quad v_3w_1 - v_1w_3 \quad v_1w_2 - v_2w_1]$.

`length(v)` calcula el número de componentes del vector v, o el máximo tamaño de las dimensiones si v es una matriz.

`[m,n]=size(A)` devuelve el número de filas y de columnas de la matriz A. Si la matriz es cuadrada basta recoger el primer valor de retorno.

`size(A,1)` devuelve el número de filas.

`size(A,2)` devuelve el número de columnas.

`max(v)` devuelve el valor de la mayor componente del vector v, o si v es una matriz genera un vector fila con el máximo de cada columna de la matriz.

`min(v)` devuelve el valor de la menor componente del vector v, o si v es una matriz genera un vector fila con el mínimo de cada columna de la matriz.

`norm(v)` obtiene el módulo del vector v

`sum(v)` devuelve la suma de las componentes del vector v. Si v es matriz genera un vector fila, siendo cada elemento igual a la suma de la columna correspondiente de la matriz.

`prod(v)` devuelve el producto de las componentes del vector v. Si v es matriz genera un vector fila, siendo cada elemento igual al producto de la columna correspondiente de la matriz.

`sort(v)` ordena las componentes de v de menor a mayor. Si v es una matriz, ordena sus columnas de menor a mayor. Por ejemplo, para ordenar un vector v, bastaría con ejecutar la siguiente instrucción:

```
v=sort(v)
```

A continuación se detallan algunas funciones que operan con matrices:

`inv(A)` da como resultado la matriz inversa de A.

`det(A)` da como resultado el determinante de A.

`trace(A)` da como resultado la traza de A.

■ Funciones para definir matrices particulares

Existen en el lenguaje M varias funciones orientadas a definir con gran facilidad matrices de tipos particulares. Algunas de estas funciones son las siguientes:

`[eye (n)]` forma la matriz *identidad* de tamaño (nxn)

`[eye (m, n)]` forma la matriz *identidad* de tamaño (mxn)

`[zeros (m, n)]` forma una matriz de *ceros* de tamaño (mxn)

`[zeros (n)]` forma una matriz de *ceros* de tamaño (nxn)

`[ones (n)]` forma una matriz de *unos* de tamaño (nxn)

`[ones (m, n)]` forma una matriz de *unos* de tamaño (mxn)

Ejemplos:

```
>>eye (2)

      1   0
      0   1

>>zeros (2, 3)

      0   0   0
      0   0   0

>>ones (4)

      1  1  1  1
      1  1  1  1
      1  1  1  1
      1  1  1  1
```

■ Números y matrices aleatorios

Para la generación de números y matrices pseudoaleatorios, M dispone de las siguientes funciones:

`[rand]` este comando genera números pseudoaleatorios distribuidos uniformemente entre 0 y 1. Cada llamada proporciona un nuevo número.

`[rand(n)]` genera una matriz de números pseudoaleatorios entre 0 y 1, con distribución uniforme, de tamaño (nxn).

`[rand(m, n)]` igual que en el caso anterior pero de tamaño (mxn).

Ejemplos:

```
>>rand  
0.9501  
>>rand(3)  
0.2311 0.8913 0.0185  
0.6068 0.7621 0.8214  
0.4860 0.4565 0.4447
```

▪ **Borrado de elementos de matrices y vectores**

Se pueden crear nuevas matrices eliminando elementos de otras ya existentes. Para borrar elementos de una matriz o vector, se debe asignar a éstos el valor vacío entre corchetes []. Ejemplo:

```
A(3,:)=[ ];
```

con esta orden se elimina la fila 3 de la matriz A.

▪ **Extracción de elementos de tablas mediante índices lógicos**

Conocemos que para extraer elementos de una tabla, se deben indicar los índices correspondientes a los elementos; sin embargo, si como índices de una tabla disponemos un vector de tipo lógico, estamos indicando que se extraigan los elementos situados en las posiciones de valor lógico 1. Ejemplo:

```
>>v=1:10  
v =  
1 2 3 4 5 6 7 8 9 10  
>> in=v>=5  
in =  
0 0 0 0 1 1 1 1 1 1  
>> v(in)  
ans =  
5 6 7 8 9 10  
% se extraen los elementos de v que cumplen la condición, es decir los  
elementos de v mayores o iguales a 5
```

- **Reutilización del formato en escritura de matrices en pantalla**

Cuando en lugar de un escalar se quiere escribir una matriz, se imprimirán los elementos en orden columnas, necesitando por cada elemento un formato. Sin embargo, en lenguaje M, ya conocemos que no es necesaria la disposición explícita de un formato por cada dato ya que cuando se termina de usar el formato especificado se reutiliza éste al completo hasta conseguir escribir el resto de datos. A continuación se escriben ejemplos relativos a esta propiedad:

```
>> A=[1 2;3 4]; B=[3.56 7.89];  
  
>> fprintf('%d %d\n',A,B);  
  
1 3  
2 4  
3.560000e+000 7.890000e+000  
  
  
>> fprintf('El dato es: %f\n',B);  
  
El dato es: 3.560000  
El dato es: 7.890000
```

2.3 Resolución de sistemas de ecuaciones.

En la mayor parte de problemas de ingeniería o ciencias aparecen sistemas de ecuaciones. Llegado este momento tenemos todas las herramientas necesarias para poder proceder a su resolución, siendo éste el objetivo de este tema.

Sea el sistema de ecuaciones que se expresa matricialmente de la forma $\vec{A}\vec{x} = \vec{b}$, siendo A la matriz de coeficientes, y \vec{b} el vector de términos independientes. Este sistema puede resolverse en MATLAB y Octave utilizando simplemente el operador *backslash* o *división izquierda* (\) de la forma:

$$x = A \setminus b$$

siendo b un vector columna. El operador \ examina los coeficientes de A antes de intentar resolver el sistema y actúa de la siguiente forma:

- Si A es triangular (superior o inferior), entonces se utiliza sustitución hacia atrás o hacia delante.
- Si A es simétrica y los elementos diagonales de A son positivos, entonces se intenta la factorización de Cholesky. No siempre es posible realizarla porque aunque la matriz A cumpla las condiciones dichas podría no ser definida positiva.
- Si las condiciones anteriores no se cumplen, se realiza una factorización LU.
- Si A no es cuadrada, tiene tamaño $M \times N$ y se cumple que $M > N$, el sistema se dice que es sobre determinado. El sistema lineal resultante puede no tener solución. La solución que se adopta consiste en encontrar valores que minimicen el error dado por la siguiente expresión, lo que se denomina Método de los Mínimos Cuadrados:

$$e = \sum_{i=1}^M (b_i - \sum_{j=1}^N a_{ij}x_j)^2$$

Por ejemplo dado el sistema

$$\begin{aligned} 2x + 3y &= 1, \\ 4x - y &= 2, \\ x - y &= 3, \\ x + y &= 0, \end{aligned}$$

el conjunto de instrucciones (programa) que finalizan con la resolución de éste mediante el método de mínimos cuadrados es

```
A=[2 3;4 -1;1 -1;1 1];  
b=[1 2 3 0];  
x=A\b'
```

x =

0.6154

-0.2692

- Si A no es cuadrada, tiene tamaño $M \times N$ y se cumple que $N > M$, el sistema se denomina infradeterminado. Estos sistemas tienen infinitas soluciones y el operador \ se limita a seleccionar una sin enviar ningún mensaje de advertencia. Por ejemplo, si consideramos el sistema infradeterminado,

$$x + y + z = -2,$$

$$2x - 3y = -4,$$

el siguiente programa obtiene la solución indicada.

```
A=[1 1 1; 2 -3 0];
b=[-2,-4];
x=A\b'
```

```
x=
-2.0000
0.0000
0.0000
```

Sin embargo, existen otras muchas soluciones, por ejemplo: $x=1, y=2, z=-5$.

2.4 Estructuras de control de tipo condicional.

Las sentencias de un programa son ejecutadas por el ordenador según su flujo natural, es decir, de arriba hacia abajo y de forma consecutiva a no ser que, de alguna manera, se altere este orden. Los comandos que realizan esta función se denominan comandos de control de flujo y las sentencias a las que pertenecen sentencias de control de flujo.

Las sentencias condicionales permiten realizar ciertas instrucciones del programa sólo si se cumple la condición asociada a ellas.

- Sentencia `if` simple

La forma básica de una sentencia condicional `if` es la siguiente:

```
if     expresión_lógica  
      sentencias  
end
```

o bien

```
if expresión_lógica , sentencias , end
```

Las sentencias se ejecutan sólo si la expresión lógica se evalúa como cierta, en el caso de que la expresión sea evaluada como falsa, el programa continúa por la instrucción siguiente a `end` sin realizar las sentencias asociadas al condicional.

- Bloque `if` unicondicional

Se trata de otro tipo de sentencia más compleja pero en la que sigue apareciendo una única expresión lógica. La sintaxis se indica a continuación.

```
if   expresión_lógica  
      bloque 1 sentencias  
    else  
      bloque 2 sentencias  
    end
```

Si la expresión lógica se evalúa como cierta se ejecuta el bloque 1 de sentencias, si se evalúa como falsa se ejecuta el bloque 2. Nunca se ejecutan los dos bloques de sentencias a la vez, ni ninguno de los dos. Se utiliza para elegir uno de dos '*'caminos'* en el programa.

- Bloque `if` multicondicional

A continuación mostramos la sintaxis de otra expresión condicional en la que se pueden evaluar dos o más condiciones.

```
if expresión_lógica_1
    bloque 1 sentencias
elseif expresión_lógica_2
    bloque 2 sentencias
.
.
.
elseif expresión_lógica_n
    bloque n sentencias
else
    bloque n+1 sentencias
end
```

Su funcionamiento es simple, la primera expresión lógica que se evalúe como cierta provoca la ejecución de sus sentencias asociadas y el posterior abandono de la estructura completa. Se utiliza para elegir uno de varios '*'caminos'* en el programa.

La última rama (`else`) es opcional. Si se escribe, sólo se ejecuta el bloque perteneciente a ella si todas las expresiones lógicas anteriores han resultado falsas.

Veamos unos ejemplos de estas estructuras.

En el siguiente programa se estudia si un número `n` es múltiplo de 2 y/o de 3.

```
n=input ('Introduce un entero');

if rem(n,2)== 0
    disp('Es múltiplo de 2')
end

if rem(n,3)== 0
    disp('Es múltiplo de 3')
end
```

Obsérvese que las dos condiciones a estudiar son independientes: el cumplimiento o no de la primera no induce a que la segunda se cumpla o no. Cuando ocurra esto se deben estudiar los casos de forma independiente usando `if` simples.

En el siguiente programa, se elige entre una de dos alternativas posibles (par o impar). Un número es par o es impar, no puede ser las dos opciones a la vez ni ninguna de ellas. Cuando ocurran estos casos se debe utilizar la estructura `if/else`.

```
n=input ('Introduce un entero positivo');

if    rem(n,2)== 0

    disp('n es par')

else

    disp('n es impar')

end
```

A continuación se incorpora al programa anterior una tercera opción, número negativo, que aumenta a tres las posibilidades de elección. Cuando tengamos más de dos posibilidades excluyentes se debe usar la estructura `if/elseif/else`.

```
n=input ('Introduce un entero');

if  n<0

    disp('n es negativo')

elseif    rem(n,2)== 0

    disp('n es par')

else

    disp('n es impar')

end
```

2.5 Practicando con programas (II). Ejercicios de autoevaluación del módulo 2.

Ejercicio 2.5.1 Diseñar un programa que compruebe si un valor x_0 es raíz de una ecuación de segundo grado. Se pedirán por teclado los valores de los coeficientes a , b , c que se almacenarán en el vector `coef`. A continuación se comprobará si x_0 verifica la ecuación:

$$a x_0^2 + b x_0 + c = 0$$

La salida del programa a pantalla será una de estas dos frases:

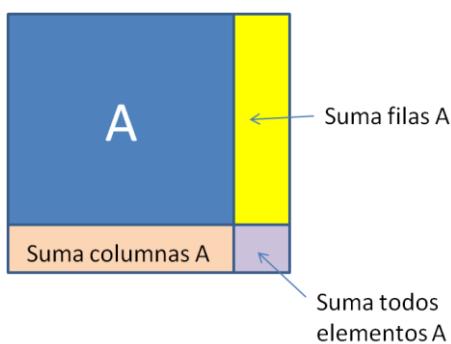
RESULTADO DE LA COMPROBACIÓN: es raíz

RESULTADO DE LA COMPROBACIÓN: no es raíz

Ejercicio 2.5.2 Sea un triángulo de lados a , b , y c . Escribir un programa que lea el tamaño de los lados y escriba en pantalla si el triángulo es o no rectángulo.

Ejercicio 2.5.3 Sea un cubo centrado en el origen de lado L , y un punto en el espacio cuyas tres coordenadas están en el vector P . Escribir un programa que pida estos datos por teclado (lado y punto) y escriba en pantalla la posición relativa entre el punto y el cubo: interior, frontera o exterior.

Ejercicio 2.5.4 Escribir un programa `M` que pida al usuario que introduzca una matriz cuadrada llamada `A`. A continuación haciendo uso de la función `sum`, calcular la suma de cada fila de `A`, de cada columna de `A` y de todos los elementos de la matriz. Se formará la matriz `B` ensamblando a la matriz `A` los resultados anteriores, tal como se indica en la figura posterior. Se mostrará la matriz `B` en pantalla.



A modo de ejemplo, si

$$A = \begin{pmatrix} 1 & 7 & -2 \\ 3 & 0 & 11 \\ -7 & 1 & 0 \end{pmatrix}, \text{ se obtiene } B = \begin{pmatrix} 1 & 7 & -2 & 6 \\ 3 & 0 & 11 & 14 \\ -7 & 1 & 0 & -6 \\ -3 & 8 & 9 & 28 \end{pmatrix}$$

Ejercicio 2.5.5 Haciendo uso del operador '*división izquierda*' (backslash):

- a) Resolver el siguiente sistema de ecuaciones:

$$\begin{aligned} 3x + 2y - z &= 0 \\ x - 4y + 2z &= 7 \\ 8x + 5y - 8z &= 5 \end{aligned}$$

- b) Hallar la solución por el método de mínimos cuadrados del sistema:

$$\begin{aligned} 7x + 2y &= -7 \\ 2x - 4y &= 20 \\ -3x + 6y &= 3 \end{aligned}$$

Solución Ejercicio 2.5.1

A continuación se muestra un programa válido:

```
coef=input('Introduce los coeficientes de la ecuación: ');
x0=input('Introduce posible raíz ');
comp=coef(1)*x0^2+coef(2)*x0+coef(3)==0;
if comp
    respuesta='es raíz';
else
    respuesta='no es raíz';
end
fprintf('RESULTADO DE LA COMPROBACIÓN: %s\n',respuesta)
```

Solución Ejercicio 2.5.2

```
lados=input('Introduce el tamaño de los tres lados del triángulo');
lados=sort(lados);
a=lados(1);b=lados(2); %catetos
c=lados(3); %hipotenusa
if a^2+b^2==c^2
    disp('El triángulo es rectángulo');
else
    disp('El triángulo no es rectángulo');
end
```

Solución Ejercicio 2.5.3

```
L=input('Introduce el lado del cubo');
P=input('Introduce las coordenadas del punto');
if max(abs(P))<L/2
    disp('El punto es interior al cubo');
elseif max(abs(P))==L/2
    disp('El punto está en la frontera del cubo');
else
    disp('El punto es exterior al cubo');
end
```

Solución Ejercicio 2.5.4

```
A=input('Introduce una matriz');
B=[ [A,sum(A')]'; [sum(A),sum(sum(A))]];
disp('La matriz resultado es: ');
disp(B);
```

Solución Ejercicio 2.5.5

Apartado a) Se escriben las siguientes sentencias obteniendo el resultado indicado.

```
A=[ 3 2 -1; 1 -4 2; 8 5 -8]
b=[0;7;5]
x=A\b
```

```
x =
1.0000
-1.9091
-0.8182
```

Apartado b) El sistema es sobredeterminado (tres ecuaciones y dos incógnitas), el operador *backslash* da una solución aproximada al problema mediante el método de mínimos cuadrados. Se escriben las siguientes sentencias obteniendo el resultado indicado.

```
A=[ 7,2;2,-4;-3,6];
b=[-7;20;3];
x=A\b
```

```
x =
-0.5769
-1.4808
```

3.1 Funciones de biblioteca

- **Concepto de función. Parámetros o argumentos**

Una función es un módulo independiente programado para realizar una tarea específica. Internamente está constituida por un conjunto de instrucciones y puede utilizarse desde cualquier sentencia escrita en la ventana de comandos, desde cualquier programa realizado por el usuario, así como desde otra función.

MATLAB y Octave disponen de gran cantidad de funciones propias (intrínsecas o que vienen con el propio software) que facilitan la realización de cálculos y programas.

Una función puede considerarse como una entidad que en general, recibe unos datos, (parámetros o argumentos de entrada), realiza operaciones con éstos, generando unos resultados (parámetros o argumentos de salida).

Para llamar a una función de biblioteca, en primer lugar se debe conocer los argumentos de entrada que necesita y los resultados que va a devolver. Para ello es útil utilizar el comando help referido a la función.

Supongamos una función de nombre `nombrefuncion`, la llamada a esta función se realizaría de la forma:

```
[s_1,s_2,...]=nombrefuncion( arg_1, arg_2, ...)
```

donde:

`arg_i` son los datos que queremos enviar a la función. Se denominan argumentos reales o actuales. Se transmiten pasando una copia de su valor a los argumentos de entrada de la función (argumentos formales).

`s_i` son las variables que reciben los datos de salida de la función.

Cuando una función es aplicable a escalares, si se aplica a una matriz realizará la operación elemento a elemento: $f([A]) = [f(A_{ij})]$

En los módulos anteriores se han estudiado algunas funciones de biblioteca, se analizan a continuación otras, divididas en diferentes categorías, que pueden ser de utilidad en el uso del lenguaje M. Estas funciones sólo son una pequeña parte de la totalidad de las funciones existentes. El alumno puede investigar otras fácilmente a través de la ayuda del programa.

- **Funciones matemáticas**

- **Funciones para cálculos básicos**

`abs(x)` obtiene el valor absoluto de x

`rem(x,y)` obtiene el resto de la división de x entre y

`sqrt(x)` obtiene la raíz cuadrada de x

`log(x)` obtiene el logaritmo neperiano de x

`log2(x)` obtiene el logaritmo base 2 de x

`log10(x)` obtiene el logaritmo base 10 de x

`exp(x)` obtiene la exponencial de x (e^x)

`sign(x)` retorna 1 si $x>0$, 0 si $x=0$, y -1 si $x<0$

- **Funciones para cálculos básicos estadísticos**

`mean(x)` calcula la media de los valores del vector x

`std(x)` calcula la desviación típica de los valores del vector x

- **Funciones trigonométricas**

Obtienen las siguientes razones trigonométricas de x :

`sin(x)` `sinh(x)` seno y seno hiperbólico

`cos(x)` `cosh(x)` coseno y coseno hiperbólico

`tan(x)` `tanh(x)` tangente y tangente hiperbólica

`cot(x)` `coth(x)` cotangente y cotangente hiperbólica

`csc(x)` `csch(x)` cosecante y cosecante hiperbólica

`sec(x)` `sech(x)` secante y secante hiperbólica

`asin(x)` `asinh(x)` arcoseno y arcoseno hiperbólico

`acos(x)` `acosh(x)` arcocoseno y arcocoseno hiperbólico

`atan(x)` `atanh(x)` arcotangente y arcotangente hiperbólica

`acot(x)` `acoth(x)` arcocotangente y arcocotangente hiperbólica

`acsc(x)` `acsch(x)` arcocosecante y arcocosecante hiperbólica

`asec(x)` `asech(x)` arcosecante y arcosecante hiperbólica

- **Funciones de redondeo**

- `[fix(x)]` elimina la parte decimal del dato x
- `[floor(x)]` obtiene el mayor entero menor o igual a x
- `[ceil(x)]` obtiene el menor entero por encima de x
- `[round(x)]` redondea x al entero más cercano

- **Funciones para trabajar con polinomios**

El lenguaje M dispone de funciones para realizar operaciones estándar con polinomios: búsqueda de raíces, evaluación, interpolación,

`[polyval(p,x)]` evalúa el polinomio de coeficientes p en x . Si x es una matriz evalúa el polinomio en cada elemento de x . Para una evaluación matricial se debe usar la función `polyvalm(p,x)`.

`[roots(p)]` obtiene un vector columna con las raíces del polinomio de coeficientes p .

`[poly(r)]` obtiene un vector con los coeficientes del polinomio de raíces r .

`[polyder(p)]` obtiene un vector con los coeficientes del polinomio resultado de derivar p .

`[conv(u,v)]` obtiene los coeficientes del polinomio resultado de multiplicar los polinomios de coeficientes u y v .

`[[p,q]=deconv(u,v)]` devuelve los polinomios cociente y resto de la división de los polinomios u entre v .

`[polyfit(x,y,n)]` devuelve el polinomio de grado n que ajusta los puntos (x, y) en el sentido de mínimos cuadrados.

- **Funciones para búsqueda de condiciones lógicas**

`[any(condición)]` si en la condición interviene un vector, la función devuelve 1 si esta condición se cumple al menos en un elemento del vector, cero en caso contrario. Si en la condición interviene una matriz, se aplica la función a cada columna de ésta dando como resultado un vector de 0 y 1.

`all(condición)` similar a la anterior, pero en este caso se chequea si todos los elementos cumplen la condición, en ese caso devuelve cierto.

`find(condición)` realiza la búsqueda de los elementos que cumplen la condición. El resultado es un vector columna con los índices de los elementos que cumplan la condición. Si en la condición interviene una matriz, el resultado también es un vector columna, al considerar la matriz como una columna detrás de otra.

Veamos a continuación unos ejemplos de utilización:

```
>> x=[1 4 7]

x =
    1      4      7

>> any(x<0)

ans =
    0

>> A=[1 2 3;-1 7 2;6 1 3]

A =
    1      2      3
   -1      7      2
    6      1      3

>> any(A<0)

ans =
    1      0      0

>> all(x>0)
```

```
ans =  
1  
  
>> all(A>0)  
  
ans =  
0      1      1  
  
>> find(x>0)  
  
ans =  
1      2      3  
  
>> find(A<0)  
  
ans =  
2
```

3.2 Estructura repetitiva *for*

Una estructura repetitiva o bucle se utiliza cuando se quiere repetir un conjunto de sentencias un número determinado de veces o mientras se mantenga el cumplimiento de una condición.

El bucle for utiliza la primera opción usando una variable numérica capaz de controlar el número de iteraciones. Esta variable es conocida como variable de control.

Su sintaxis es la siguiente:

```
for variable = inicio:fin  
    Sentencias  
end
```

La utilización del operador (:) en la cabecera del bucle es la conocida hasta el momento. La variable de la cabecera va tomando sucesivamente cada uno de los valores asignados (no es un vector), y para cada uno de ellos se repiten las sentencias asociadas al bucle. En el caso anterior el incremento de la variable que controla el bucle es 1. Para incrementos distintos se debe utilizar:

```
for variable = inicio:incremento:fin  
    Sentencias  
end
```

También puede darse a la variable una sucesión de valores cualesquiera, sin necesidad de utilizar el operador (:). La variable que controla el bucle irá tomando cada uno de los valores indicados efectuando para cada uno una iteración.

```
for variable=[valor1,valor2,valor3,.....]  
    Sentencias  
end
```

Véanse a continuación unos ejemplos en los que se utilizan bucles ordinarios.

En el siguiente programa se inicializan al valor 1 las n primeras componentes de un vector.

```
n=input('Introduce un número natural');

for i=1:n
    v(i)=1;
end
```

En el siguiente programa se crea la matriz de Hilbert de tamaño m×n.

```
m=input('Introduce el número de filas');

m=input('Introduce el número de columnas');

for i=1:m
    for j=1:n
        A(i,j)=1/(i+j-1);
    end
end
```

El siguiente programa pide por teclado cada elemento de una matriz 4x3 con un mensaje adecuado a cada uno.

```
for i=1:4
    for j=1:3
        fprintf('Elemento [%d,%d]\n',i,j);
        A(i,j)=input('Introduce dato');
    end
end
```

En los dos últimos programas aparecen bucles anidados. Expliquemos con detalle el último de ellos. Para cada valor de la variable *i*, se ejecuta el bucle *j* para los valores 1, 2, 3. En cada iteración se genera un texto con *fprintf* que sirve como mensaje previo a la petición del dato en la sentencia posterior, en la que se lee un escalar que se guarda en *A(i,j)*. Por el orden en el que están dispuestos los bucles los datos leídos se guardan en el orden de las filas de *A*. El orden de lectura sería: *A(1,1)*, *A(1,2)*, *A(1,3)*, *A(2,1)*, *A(2,2)*, *A(2,3)*, *A(3,1)*, *A(3,2)*, *A(3,3)*, *A(4,1)*, *A(4,2)*, *A(4,3)*.

3.3 Estructura repetitiva *while*

Cuando el control de las iteraciones de un bucle se realiza mediante el valor de una expresión lógica se debe utilizar el bucle *while*.

En la cabecera del bucle aparece una expresión lógica que, en cada iteración, debe evaluarse como cierta o falsa. Si se evalúa como cierta se realiza la ejecución de las instrucciones asociadas al bucle, sin embargo, la primera vez que se evalúe como falsa se interrumpe el bucle y el programa prosigue en la sentencia posterior a *end*. Su sintaxis es:

```
while    expresión lógica  
        sentencias  
end
```

Ejemplo:

```
v=1:9;  
  
i=1;  
  
while v(i)<7  
    disp(v(i));  
    i=i+1;  
end
```

La ejecución de este programa produce la siguiente salida a pantalla, en forma de vector columna: 1 2 3 4 5 6

En el siguiente ejemplo, aparece un fragmento de programa en el que se utiliza un bucle *while* y la función *isempty* para conseguir que el programa no prosiga si el usuario no introduce un dato. La condición del bucle *while* puede completarse con otras para que el dato introducido además cumpla ciertos criterios.

```
A=input('Introduce una matriz');  
  
while isempty(A)  
    A=input('Debes introducir un dato. Introduce una matriz');  
end
```

Como ejemplo de lo visto en esta sección, pedimos que el alumno piense en una solución para el siguiente problema:

Se quiere escribir un programa que pida una matriz cualquiera A por teclado y no permita que el usuario no introduzca una respuesta. A continuación debe solicitar un número entero n que se identificará con un número de fila de la matriz. Si el número introducido no cumple las condiciones (entero y ser un número de fila de la matriz) se debe volver a pedir las veces necesarias. Con estos datos, se escribirá en pantalla la matriz A con la fila n eliminada de su posición y situada como última fila.

Una solución válida al problema es:

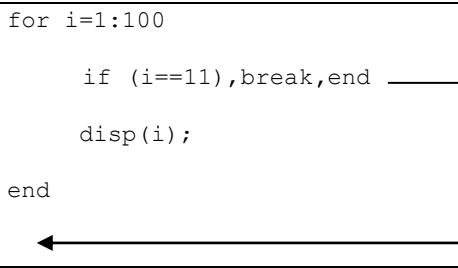
```
A=input('Introduce una matriz: ') ;  
  
while isempty(A)  
  
    A=input('Debes introducir un dato. Introduce una matriz');  
  
end  
  
filas=size(A,1);  
  
n=input('Introduce número de fila');  
  
while n~=fix(n)|n>filas|n<1  
  
    n=input('Datos Erróneo. Introduce número de fila');  
  
end  
  
v=A(n,:);  
  
A(n,:)=[];  
  
A=[A;v];  
  
disp(A)
```

3.4 Interrupciones de bucles y programas

■ La sentencia break

La sentencia `break` se puede utilizar únicamente en el interior de cualquiera de los bucles del lenguaje M. Permite interrumpir el bucle desde cualquier punto de su cuerpo. Cuando se ejecute el comando `break`, el bucle termina inmediatamente y el programa continúa en la sentencia que sigue a éste. Por ejemplo, este programa imprime los números del 1 al 100.

```
for i=1:100
    if (i==11),break,end
    disp(i);
end
```



Cuando `i` toma el valor 11, la condición asociada a `if` es cierta y se ejecuta `break`; en ese momento se interrumpe el bucle (no se imprime el valor 11 ni los siguientes). Si no se hubiera escrito la línea de comando `break`, se imprimirían los números del 1 al 100.

Si `break` pertenece a un bucle que es interior a otro, sólo produce la interrupción del bucle al que pertenece, siguiendo el programa en el bucle exterior.

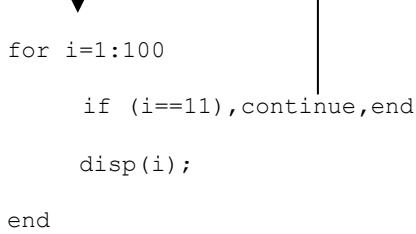
■ La sentencia continue

Esta sentencia también se utiliza únicamente dentro de cualquiera de los dos bucles del lenguaje M.

La sentencia `continue` envía el flujo del programa a la cabecera del bucle en el que se encuentre, dejándose de ejecutar, en esa iteración, las líneas de comando que se encuentren entre `continue` y el final del bucle. No se interrumpe un bucle sino una iteración.

En el siguiente ejemplo, se ha realizado un programa semejante al del apartado anterior empleando el comando `continue` en lugar de `break`.

```
for i=1:100
    if (i==11),continue,end
    disp(i);
end
```



El bucle comienza con el valor 1 de la variable de control, que se va incrementando en una unidad en cada iteración imprimiéndose en pantalla. Cuando *i* vale 11 se cumple la condición asociada a *if* y, por tanto, se ejecuta *continue*, que envía el flujo del programa a la cabecera del bucle sin ejecutarse la iteración actual (el *disp* correspondiente) pero sí las siguientes. Debido a esto, el valor 11 no aparece en pantalla. El siguiente valor que toma *i* es 12, que se imprimirá, y así sigue el proceso hasta el valor 100.

En el siguiente programa se introducen las notas de 10 alumnos, si alguna no es correcta (menor que 0 o mayor que 10) se volverá a pedir esa nota (*continue* evita ejecutar el incremento de *i*, con lo cual se mantiene en el mismo alumno). Además se calcula la nota media.

```
i=1;media=0;

while(i<=10) ←

    fprintf('Alumno %d\n', i);

    notas(i)=input('Introduce nota');

    if (notas(i)>10 | notas(i)<0) continue;
    end;

    media=media+notas(i);

    i=i+1;

end

fprintf('La nota media es %f\n', media/10)
```

- La sentencia *return*

Esta sentencia se puede utilizar en cualquier lugar de un programa o función M. Provoca la finalización anticipada de la función o programa.

Por ejemplo, el siguiente programa termina anticipadamente cuando el dato introducido se considera no válido.

```
notas=input('Introduce las tres notas del alumno\n');

if (max(notas)>10 || min(notas)<0 )

    disp('Datos erróneos');

    return;

end

media=sum(notas)/length(notas);

fprintf ('La nota media es: %f\n', media)
```

3.5 Practicando con programas (III). Ejercicios de autoevaluación del módulo.

Ejercicio 3.5.1 Diseñar un programa que pida un vector cualquiera por teclado y calcule el máximo y mínimo de los valores de sus componentes. Además debe escribir en pantalla la información de cada elemento y los valores máximos y mínimos (usando las funciones *max* y *min*) tal como se indica a continuación:

El elemento 1 del vector es

El elemento 2 del vector es

El elemento 3 del vector es

.....

El elemento ... del vector es

Los valores máximo y mínimo son y

Ejercicio 3.5.2 Diseñar un programa que pida un vector cualquiera por teclado, y calcule el valor máximo de sus elementos mediante el uso del bucle for. No se puede usar la función *max*.

Ejercicio 3.5.3 Realizar el mismo proceso que en el ejercicio anterior pero en este caso se introducirá por teclado una matriz. Realizar una primera solución en la que intervengan dos bucles for anidados. En otra solución alternativa se debe vectorizar la matriz y continuar como en el ejercicio 3.5.2.

Ejercicio 3.5.4

a) Escribir una programa que lea un vector por teclado y que realice con él un proceso idéntico al que ejecutan las funciones de librería *fliplr* o *flipud*.

Es decir, si se introduce el vector $\mathbf{v}=[1, 2, 3, 4]$, al finalizar el programa se tiene $\mathbf{v}=[4, 3, 2, 1]$. De igual manera, si se introduce el vector columna $\mathbf{w}=[1, 2, 3]'$, al finalizar el programa se tiene el vector columna $\mathbf{w}=[3, 2, 1]'$.

b) Modificar el programa anterior para conseguir voltear cada columna de una matriz cualquiera leída por teclado. Es decir, si tenemos la matriz

$$A = \begin{pmatrix} 1 & 3 & 8 \\ 5 & 7 & 9 \\ -4 & 2 & 10 \end{pmatrix}$$

se convertirá en la matriz

$$A = \begin{pmatrix} -4 & 2 & 10 \\ 5 & 7 & 9 \\ 1 & 3 & 8 \end{pmatrix}$$

Ejercicio 3.5.5 Escribir un programa que conteste a la siguiente pregunta: ¿cuántas veces hemos de lanzar una moneda al aire para que el número de caras menos el número de cruces sea igual a 5?

El suceso de lanzar una moneda se puede simular mediante la utilización de la función `rand`. Si el resultado obtenido es menor que 0.5 podemos suponer, por ejemplo, que se ha obtenido cara y en otro caso cruz.

Ejercicio 3.5.6

Diseñar un programa que comience obteniendo un entero aleatorio entre 1 y 10. Llámese `x` a ese dato. A continuación pregunte al usuario del programa, un número `n` entre 1 y 10, y si éste no coincide con el seleccionado previamente por el ordenador, vuelva a hacer la pregunta las veces necesarias hasta que se produzca la coincidencia. Se contabilizará la cantidad de intentos necesaria para adivinar el número.

Al final, el programa escribirá en pantalla la frase:

El número es Lo has adivinado en intentos

Solución Ejercicio 3.5.1

A continuación se muestra un programa válido:

```
v=input('Introduce el vector: ');

n=length(v);

for i=1:n

    fprintf('El elemento %d del vector es %f\n',i,v(i));

end

fprintf('Los valores máximo y mínimo son %f y %f\n',max(v),min(v));
```

Solución Ejercicio 3.5.2

```
v=input('Introduce el vector: ');

n=length(v);

maximo=v(1);

for i=2:n

    if v(i)>maximo

        maximo=v(i);

    end

end

fprintf('El valor máximo del vector es: %f\n',maximo);
```

Solución Ejercicio 3.5.3

Solución 1

```
A=input('Introduce la matriz: ');

[f,c]=size(A);

maximo=A(1,1);

for i=1:f

    for j=1:c

        if A(i,j)>maximo

            maximo=A(i,j);

        end

    end

end
```

```

end

fprintf('El valor máximo de la matriz es: %f\n',maximo);

```

Solución 2

```

A=input('Introduce la matriz: ');

v=A(:,);

n=length(v);

maximo=v(1);

for i=2:n

    if v(i)>maximo

        maximo=v(i);

    end

end

fprintf('El valor máximo de la matriz es: %f\n',maximo);

```

Solución Ejercicio 3.5.4

Apartado a)

```

v=input('introduce vector');

n=length(v);

for i=1:fix(n/2)

    aux=v(i);

    v(i)=v(n-i+1);

    v(n-i+1)=aux;

end

disp(v);

```

Apartado b)

```

A=input('introduce matriz');

[n,c]=size(A);

for j=1:c

    v=A(:,j);

    for i=1:fix(n/2)

```

```

aux=v(i);

v(i)=v(n-i+1);

v(n-i+1)=aux;

end

A(:,j)=v;

end

disp(A);

```

Solución Ejercicio 3.5.5

Una solución correcta para el problema planteado es:

```

cara=0; %número de caras

cruz=0; %número de cruces

nvec=0; %contador del número de tiradas

while cara-cruz~=5

    x=rand();

    nvec=nvec+1;

    if x<0.5

        cara=cara+1;

    else

        cruz=cruz+1;

    end

end

fprintf('Se ha conseguido en %d veces',nvec);

```

Solución Ejercicio 3.5.6

Una solución al problema es:

```

x=round(0.5+10*rand());

n=input('Introduce un numero entre 1 y 10: ');

intentos=1;

while(x~=n)

```

```
n=input('No has acertado. Introduce otro numero entre 1 y 10: ');\n\nintentos=intentos+1;\n\nend\n\nfprintf('El numero es %d. Lo has adivinado en %d intentos\\n',x,intentos);
```

4.1 Gráficos bidimensionales

▪ Gráficos de puntos en el plano

El comando

```
plot(x1,y1,str1)
```

crea un gráfico a partir de la pareja de vectores $(x1, y1)$ con el estilo indicado en la cadena de texto str1. El resultado es el gráfico que representa los puntos cuyas abscisas están en el vector x1 y cuyas ordenadas están en el vector y1. Los puntos se representarán, por defecto, marcados con un punto y unidos con línea continua de color azul. Si se desea cambiar el color, estilo de línea o marcador de los puntos se debe incluir el tercer argumento (cadena str1) pudiendo contener ésta los siguientes atributos (el orden es indiferente):

- Tipos de marcadores de los puntos: . * x o +
- Colores: y: amarillo, g: verde, m: magenta, b: azul, c: cian, w: blanco, r: rojo, k: negro.
- Tipos de línea: los puntos se unen con una línea con las siguientes posibilidades de apariencia:
 - (línea continua)
 - (línea formada por trazos discontinuos)
 - (línea formada por puntos y trazos)
 - : (línea formada por puntos)

Además, se puede modificar el grosor de línea incluyendo:

```
'Linewidth', número_indicativo_del_grosor
```

y el tamaño de los marcadores de puntos incluyendo:

```
'Markersize', número_indicativo_del_tamaño
```

En el siguiente ejemplo se incluyen las instrucciones para representar gráficamente los puntos de coordenadas (1,1), (6,0), (5,4), (2,3) y finalmente de nuevo (1,1).

```
>> x=[1 6 5 2 1];
>> y=[1 0 4 3 1];
>> plot(x,y) % se dibujan los puntos unidos con línea continua azul

>> plot(x,y,'-*g')% se dibujan los puntos unidos con línea continua, %marcando
los puntos con *, y en color verde

>> plot(x,y,'-*', 'Linewidth',2, 'Markersize',4)
% se cambia el grosor de línea a 2
% y el tamaño del marcador de puntos a 4
```

El resultado del último comando se puede ver en la figura 4.1.

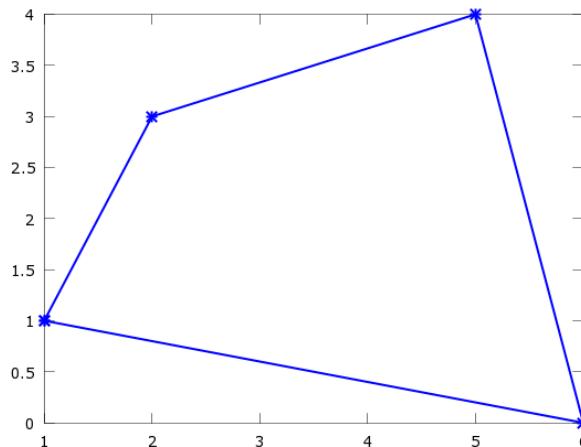


Figura 4.1 Representación de puntos unido con línea continua

■ Gráficos de funciones de una variable

Se puede utilizar el comando `plot` para representar gráficamente funciones de una variable. Para ello, se deben obtener las coordenadas de varios puntos consecutivos pertenecientes a la función y representar estos puntos como se ha visto anteriormente.

Supongamos que se desea obtener la gráfica de la función $f(x) = x^2$ en el intervalo $x \in [-10,10]$. Pasos a seguir:

- Vector con las abscisas de los puntos. Serán equidistantes en el intervalo en el que se quiera obtener el gráfico.

```
x= -10:0.1:10;
```

- Vector con las ordenadas de los puntos. Se aplica a cada abscisa la función a representar. Se realiza la operación con cada elemento del vector de abscisas.

```
y=x.^2;
```

- Se realiza el gráfico. El resultado obtenido se muestra en la figura 4.2.

```
plot(x,y)  
% para mantener la misma escala en ambos ejes  
% se incluye el siguiente comando  
axis equal % este comando se explica en la lección 4.3
```

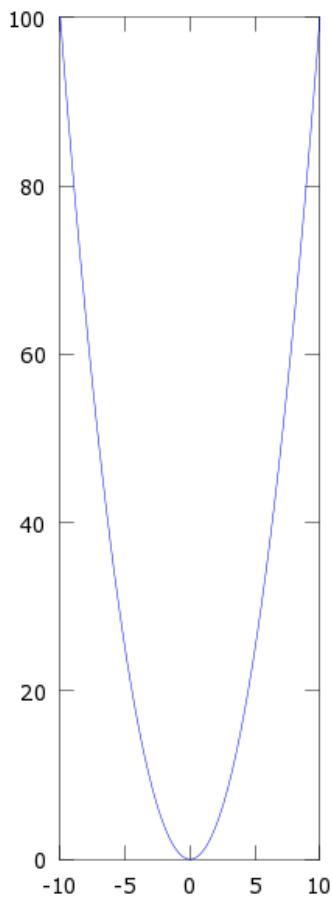


Figura 4.2 Gráfico de una función

- **Gráficos de curvas en el plano dadas por sus coordenadas paramétricas u otros sistemas de coordenadas**

Dada una ecuación cartesiana de una curva, por ejemplo $y=f(x)$, si tanto x como y pueden expresarse en función de un parámetro entonces la curva quedaría representada por las llamadas ecuaciones paramétricas.

Por ejemplo, sea la curva de ecuación $y=x^2$, una parametrización posible de ésta sería:

$$\begin{cases} x = t, \\ y = t^2. \end{cases}$$

Para representar con MATLAB / Octave curvas en el plano expresadas por sus ecuaciones paramétricas, simplemente hay que dar diferentes valores al parámetro y y obtener las coordenadas x, y resultantes, todo ello operando vectorialmente. Finalmente se representará la curva con el comando `plot`.

Veamos un ejemplo:

Sea una circunferencia de radio 2 y centro el origen de coordenadas representada por las siguientes ecuaciones paramétricas:

$$\begin{cases} x = 2 \cos(t) & , t \in [0, 2\pi] \\ y = 2 \sin(t) \end{cases}$$

El programa que genera el gráfico de la curva es el siguiente:

```
t=0:0.1:2*pi; % se dan diferentes valores al parámetro
x=2*cos(t); %se calcula x para los valores anteriores del parámetro
y=2*sin(t); %se calcula y para los valores anteriores del parámetro
plot(x,y)
```

El gráfico resultante se muestra en la figura 4.3.

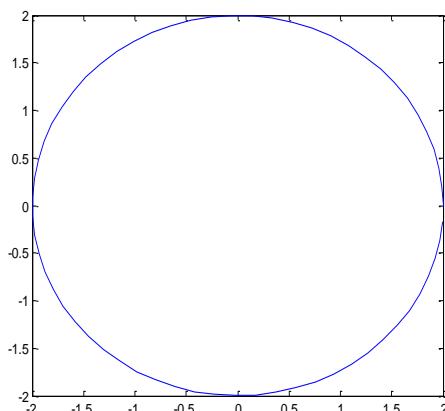


Figura 4.3 Gráfico de una circunferencia de radio 2 con el comando `plot`

Si la ecuación de la curva viene dada en coordenadas polares tenemos un caso particular.

Para transformar las coordenadas polares a cartesianas se tienen que utilizar las ecuaciones:

$$x = r \cos(\theta),$$

$$y = r \sin(\theta).$$

a las que al aplicar la ecuación de la curva se obtienen las ecuaciones cartesianas en función de un parámetro.

Por ejemplo, dada la curva de ecuación en coordenadas polares,

$$r = 1 - \cos(\theta), \quad \theta \in [0, 2\pi]$$

al sustituir en las ecuaciones de conversión de polares a cartesianas se obtiene,

$$x = (1 - \cos(\theta)) \cos(\theta),$$

$$y = (1 - \cos(\theta)) \sin(\theta).$$

donde θ es el parámetro.

El programa M que obtiene un gráfico de la curva dada es

```
tt=0:pi/60:2*pi;
x=(1-cos(tt)).*cos(tt);
y=(1-cos(tt)).*sin(tt);
plot(x,y)
```

El gráfico obtenido aparece en la figura 4.4.

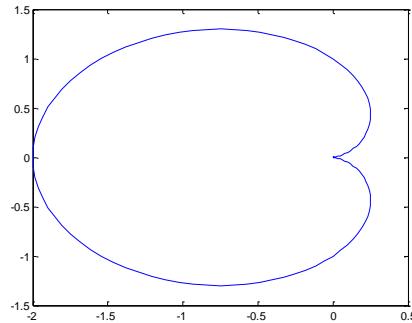


Figura 4.4 Representación de una curva en el plano dada por sus ecuaciones polares

4.2 Gráficos de superficies

▪ Superficie como función de dos variables

Una función de dos variables $z = f(x, y)$ queda representada gráficamente por una superficie. Supongamos que se quiere representar $f(x, y)$ en el intervalo $x \in [a, b]$, $y \in [c, d]$. Se procede en tres etapas:

- 1) Se generan los vectores \mathbf{X} e \mathbf{Y} que contienen la discretización del segmento $[a, b]$ del eje x y $[c, d]$ del eje y respectivamente.

A continuación, se genera una retícula rectangular a partir de las discretizaciones \mathbf{X} e \mathbf{Y} , mediante la función `meshgrid`, de la forma

```
[Mx, My] = meshgrid(X, Y)
```

Esta función devuelve dos matrices \mathbf{Mx} , \mathbf{My} , con las coordenadas x e y respectivamente de los puntos de la retícula obtenida cuando se 'cruzan' las discretizaciones \mathbf{X} e \mathbf{Y} .

Para clarificar el contenido de las matrices retornadas por `meshgrid`, supongamos que \mathbf{X} e \mathbf{Y} son los vectores,

```
X = [0, 1, 2];
```

```
Y = [2, 3];
```

Las matrices que devuelve el uso de la sentencia `[Mx, My] = meshgrid(X, Y)`, son

$\mathbf{Mx} =$

0	1	2
0	1	2

$\mathbf{My} =$

2	2	2
3	3	3

que se corresponden con las coordenadas x, y de los puntos de la retícula obtenida, que se representa en la figura 4.5.

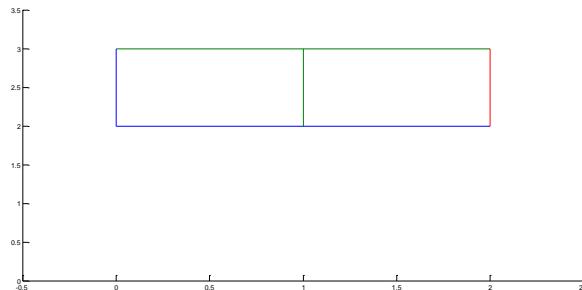


Figura 4.5 Ejemplo de malla obtenida con `meshgrid`

- 2) Se construye la matriz con los valores de $f(x, y)$ sobre los puntos de la retícula anterior.

$$Mz=f(Mx, My)$$

- 3) Se dibuja la superficie. Existen varias posibilidades, las más utilizadas son:

`mesh (Mx, My, Mz)` dibuja la superficie mediante líneas entrecruzadas.

`surf (Mx, My, Mz)` dibuja la superficie con el área de la retícula coloreada.

`plot3 (Mx, My, Mz)` dibuja la superficie con varias curvas (una por cada columna de Mx , My , Mz). Se puede incluir una cadena que indique marcadores de puntos, tipo de línea, etc, Por ejemplo, si se indica `plot3 (Mx, My, Mz, '*')` se dibuja la superficie sin líneas de unión entre puntos, marcando los puntos con asteriscos.

Ejemplo: sea la función de dos variables $f(x, y) = \sin(x)\cos(y)$, a continuación aparece el programa que, siguiendo los pasos anteriores, logra representarla gráficamente en $[-2\pi, 2\pi]_x [-2\pi, 2\pi]$.

```
X=[-2*pi:pi/10:2*pi];
Y=X;
[Mx,My]=meshgrid(X,Y);
Mz=sin(Mx).*cos(My);
mesh(Mz);
```

En este caso se ha utilizado el comando `mesh`. La superficie obtenida se muestra en la figura 4.6.

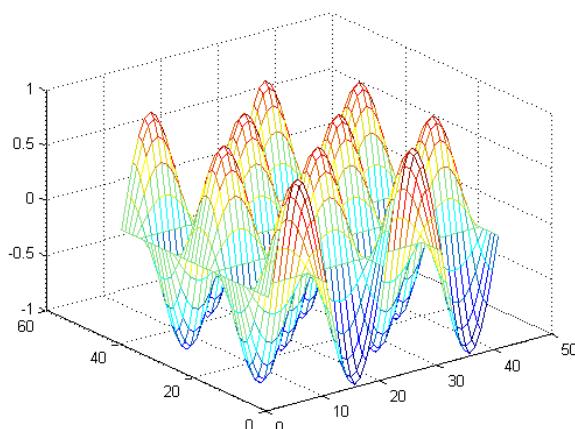


Figura 4.6 Gráfico de una superficie usando `mesh`

- **Superficie dada por sus ecuaciones paramétricas u otro sistemas de coordenadas**

Dada una ecuación cartesiana de una superficie, por ejemplo $z=f(x, y)$, si x , y , z pueden expresarse en función de dos parámetros entonces la superficie quedaría representada por las llamadas ecuaciones paramétricas, de la forma:

$$x=x(u, v)$$

$$y=y(u, v)$$

$$z=z(u, v) \quad \text{siendo } u, v \text{ los parámetros.}$$

Para representar superficies expresadas en ecuaciones paramétricas, se forman los vectores con la discretización del dominio en los parámetros u, v ; a continuación se generan las matrices de la malla usando `meshgrid`, aplicándolas en las ecuaciones paramétricas. Las matrices resultantes contienen las coordenadas cartesianas de los puntos a representar y permiten representar la superficie con el comando `surf` u otro de los conocidos.

Como ejemplo, se va a representar gráficamente una esfera de radio $R=2$ dada por las ecuaciones paramétricas

$$\begin{aligned} x &= R * \cos(u) * \cos(v), \quad u \in [0, 2\pi], v \in [-\pi/2, \pi/2] \\ y &= R * \sin(u) * \cos(v) \\ z &= R * \sin(v) \end{aligned}$$

El programa M y la figura obtenida son los siguientes:

```
u=linspace(0,2*pi,60);
v=linspace(-pi/2,pi/2,30);
[U,V]=meshgrid(u,v);
x=2*cos(U).*cos(V);
y=2*sin(U).*cos(V);
z=2*sin(V);
surf(x,y,z)
```

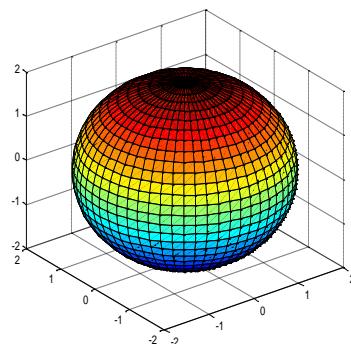


Figura 4.7 Representación de una esfera con `surf`

Si la superficie viene determinada por sus ecuaciones en el sistema de coordenadas cilíndricas o esféricas se tratará como un caso particular de superficie dada por ecuaciones paramétricas, ya que después de un pequeño proceso analítico se obtienen ecuaciones en función de dos parámetros.

El sistema de coordenadas cilíndricas define la posición de un punto del espacio mediante un ángulo, una distancia a un eje y una altura en la dirección del mismo eje.

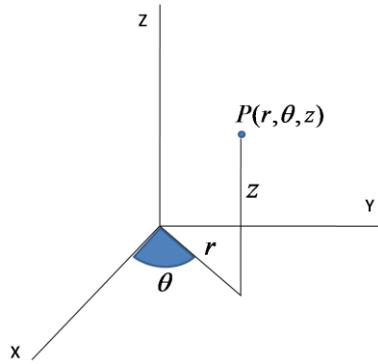


Figura 4.8 Sistema de coordenadas cilíndricas

Según la figura 4.8, un punto P quedará representado por (r, θ, z) , donde:

- r es la distancia del punto al eje z .
- θ es el ángulo que forma la proyección del punto sobre el plano XY con el eje X en sentido antihorario.
- z es la distancia, con el signo correspondiente, al plano XY .

Para transformar las coordenadas cilíndricas en cartesianas se tienen que utilizar las ecuaciones:

$$x = r \cos(\theta),$$

$$y = r \sin(\theta),$$

$$z = z.$$

Si a estas ecuaciones se aplica la ecuación de la superficie, nos quedan en función de dos parámetros pudiéndose obtener el gráfico de la forma indicada antes.

Ejemplo. Representar gráficamente la superficie dada por sus ecuaciones en coordenadas cilíndricas

$$\rho^2 = z, \quad z \in [0, 1], \quad \theta \in [0, 2\pi]$$

Paso previo analítico. Se sustituye la ecuación de la superficie en las ecuaciones de transformación de cilíndricas a cartesianas.

$$\begin{aligned}
 x &= r \cos(\theta), \\
 y &= r \sin(\theta), \\
 z &= z \rightarrow z = r^2
 \end{aligned}$$

El programa es el siguiente:

```

tt=0:pi/30:2*pi;
r=0:0.1:1;
[R, T]=meshgrid(r,tt);
X=R.*cos (T);
Y=R.*sin (T);
Z=R.^2;
surf (X,Y,Z)

```

El gráfico obtenido se muestra en la figura 4.9

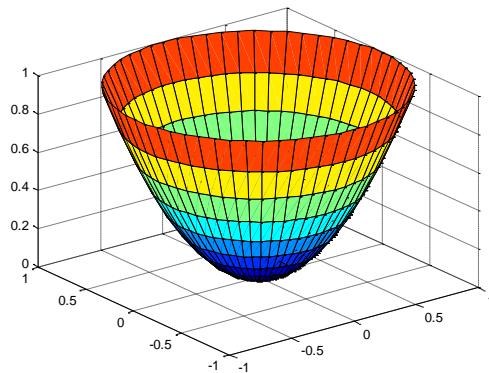


Figura 4.9 Gráfico de una superficie dada por sus ecuaciones cilíndricas

El sistema de coordenadas esféricas define la posición de un punto del espacio mediante dos ángulos y una distancia a un punto.

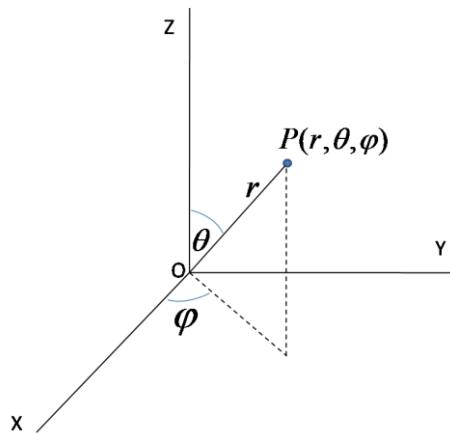


Figura 4.10 Sistema de coordenadas esféricas

Según la figura anterior, un punto P quedará representado por (r,θ,φ) , donde:

- r es la distancia del punto al origen de coordenadas.
- θ es el ángulo que forma el vector de posición \overrightarrow{OP} con el eje z.
- φ es el ángulo que forma la proyección del punto sobre el plano XY con el eje X en sentido antihorario.

Para transformar las coordenadas esféricas en cartesianas se tienen que utilizar las ecuaciones:

$$\begin{aligned}x &= r \sin(\theta) \cos(\varphi), \\y &= r \sin(\theta) \sin(\varphi), \\z &= r \cos(\theta).\end{aligned}$$

Si a estas ecuaciones se aplica la ecuación de la superficie, nos quedan en función de dos parámetros pudiéndose obtener el gráfico de la forma indicada antes.

Ejemplo. Representar gráficamente la superficie dada por sus ecuaciones en coordenadas esféricas

$$r=1, \quad \theta \in [0, \pi/2], \varphi \in [0, 2\pi]$$

Paso previo analítico. Se sustituye la ecuación de la superficie en las ecuaciones de transformación de esféricas a cartesianas.

$$\begin{aligned}x &= \sin(\theta) \cos(\varphi), \\y &= \sin(\theta) \sin(\varphi), \\z &= \cos(\theta), \quad \theta \in [0, \pi/2], \varphi \in [0, 2\pi]\end{aligned}$$

El programa es el siguiente:

```
tt=0:pi/30:pi/2;
fi=0:pi/30:2*pi;
[T,F]=meshgrid(tt,fi);
X=sin(T).*cos(F);
Y=sin(T).*sin(F);
Z=cos(T);
surf(X,Y,Z)
```

Se puede ver el gráfico obtenido en la figura 4.11

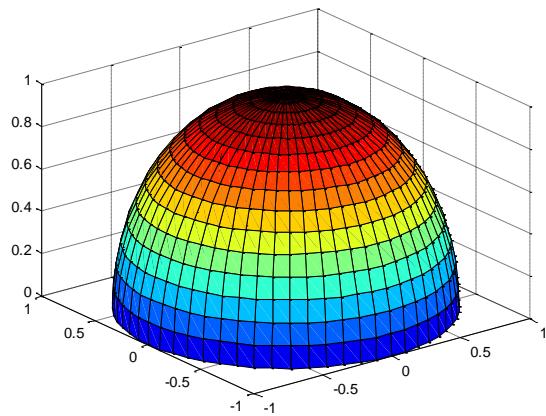


Figura 4.11 Superficie representada con `surf` dada por sus ecuaciones esféricas

4.3 Manejo de múltiples gráficos y comandos auxiliares

▪ Comandos auxiliares en el trazado de gráficos

Es habitual tener que modificar la apariencia de un gráfico que ofrece por defecto MATLAB u Octave. Para adaptar el gráfico a los requerimientos particulares, se pueden incluir leyendas, títulos en la figura, etiquetas en los ejes, modificar colores, líneas, etc. Se incluyen ahora los modificadores que consideramos más útiles.

- Añadiendo rótulos

Para añadir rótulos a los ejes y título al dibujo, se utilizan los siguientes comandos:

```
xlabel('cadena de caracteres')  
ylabel('cadena de caracteres')  
zlabel('cadena de caracteres')  
title('cadena de caracteres')
```

Se pueden añadir propiedades a los rótulos, por ejemplo, modificar el tamaño de letra:

```
zlabel('Eje z', 'FontSize', 16)
```

Cuando se tienen varios gráficos en la misma figura, es necesario etiquetar cada una de las figuras que aparecen incluyendo una leyenda de la siguiente forma:

```
legend('Etiqueta figura 1', 'Etiqueta figura 2', ...)
```

Las leyendas se asignan a las figuras según su orden de aparición.

- Punto de vista de una figura 3D

El punto de observación tridimensional standard de una figura se corresponde con azimut -37.5º, (ángulo de rotación en sentido horario en el plano xy medido desde el semieje -y) y elevación 30º (con respecto al plano xy). Ver figura 4.12

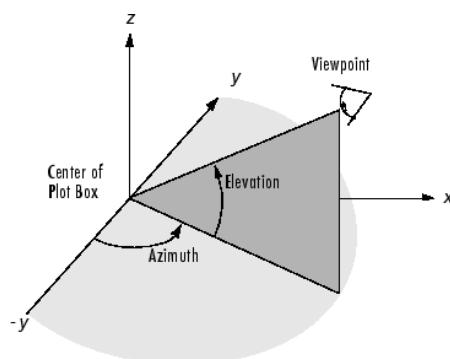


Figura 4.12 Punto de vista de una figura tridimensional

(Figura obtenida de la página web de mathworks:

<http://es.mathworks.com/help/matlab/visualize/setting-the-viewpoint-with-azimuth-and-elevation.html>

Se puede modificar este punto de observación mediante el comando:

```
view([az,el])
```

que sitúa el punto de vista de la figura en el punto de azimut `az` y elevación `el`. Las dos magnitudes se miden en grados.

El comando `view([x y z])` sitúa el punto de observación en `[x y z]`.

El comando `view(3)` devuelve el punto de vista al punto de observación standard.

– Modificando el gráfico de una superficie

Puede ser interesante representar una superficie sin que aparezcan dibujadas en ella las líneas de borde de la retícula o mallado, o bien difuminar los colores de fondo de la retícula.

El comando `shading` actúa sobre las propiedades de color de borde '`EdgeColor`' y color de fondo '`FaceColor`' del gráfico de una superficie.

Para eliminar los bordes de la retícula, se utiliza

```
shading flat
```

después de la sentencia que dibuja la superficie.

Si se utiliza

```
shading interp
```

además de eliminar las líneas de borde, se produce un degradado, mediante interpolación, en los colores de relleno de la retícula.

En la figura 4.13, obsérvese el gráfico de la misma superficie, con su representación convencional (`surf`) y con las modificaciones aportadas por `shading`.

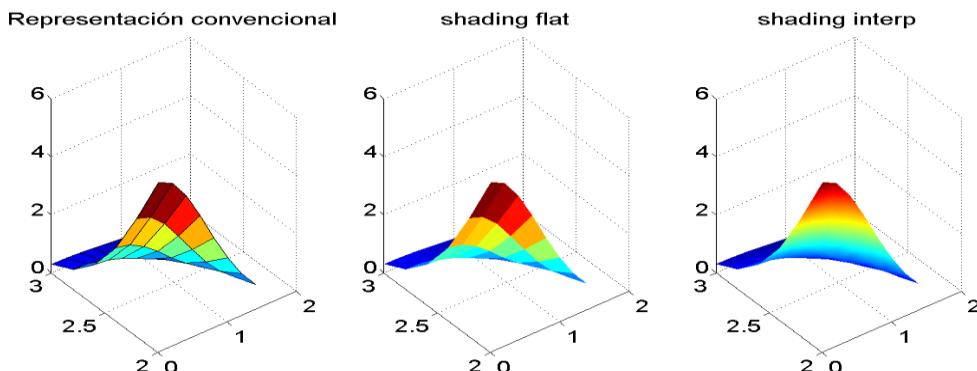


Figura 4.13 Utilización del comando `shading`

– Manejo y modificación de los ejes de coordenadas

Se utilizará el comando `axis`, que controla la escala y la apariencia de los ejes. A continuación se indican algunos de sus usos:

`axis off` elimina de la figura los ejes y planos coordinados.

`axis on` vuelve a dibujar los ejes eliminados con el comando anterior.

`axis([xmin xmax ymin ymax])` establece la escala para los ejes x e y.

`axis([xmin xmax ymin ymax zmin zmax])` similar al anterior, pero en este caso se controla también el eje z.

`axis tight` obliga a que los límites de los ejes coincidan con los de los datos representados.

`axis equal` obliga a que las marcas de unidades en los ejes x, y, z sean iguales en tamaño. Se suele utilizar cuando se representan figuras con secciones circulares. Por ejemplo, si se representa una esfera sin usar este comando la apariencia sería de un elipsoide.

`axis image` igual a `axis tight + axis equal`

- Otros comandos que pueden ser de utilidad:

`clf` borra la figura activa

`colorbar` muestra en la figura la escala de color para el gráfico representado

`grid off` se eliminan las líneas discontinuas en los planos coordinados que marcan las divisiones de los ejes

`grid on` se vuelven a incluir en la figura las líneas eliminadas con el comando anterior

- **Gráficos en diferentes ventanas. Comando figure**

El comando

```
figure
```

crea una nueva ventana de visualización. El gráfico que se genere a continuación se situará en esa ventana. Se puede utilizar todas las veces que se desee creando tantas ventanas de figura como sea necesario.

En el siguiente programa se generan dos ventanas de visualización con dos gráficos distintos, el de la función seno y el de la función coseno en el intervalo $[-3\pi, 3\pi]$.

```
x=-3*pi:pi/10:3*pi;  
y=sin(x);  
z=cos(x);  
  
figure  
plot(x,y);  
title('seno(x)');  
  
figure  
plot(x,z);  
title('coseno(x)');
```

El resultado obtenido se muestra en la figura 4.14.

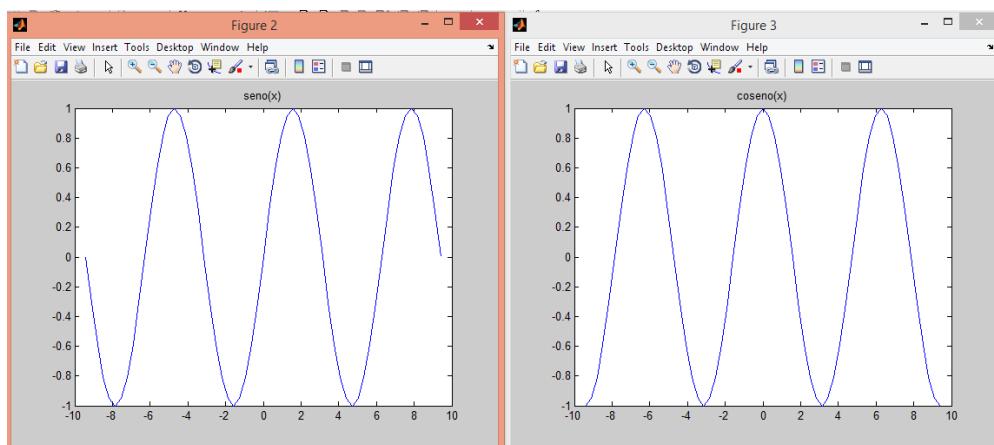


Figura 4.14 Generación de gráficos en diferentes ventanas

■ Superposición de gráficos

Todos los gráficos que se ordene dibujar entre los comandos `hold on` y `hold off` se representan en la misma figura y ejes. Si hubiera una figura abierta (última ventana gráfica generada) se dibujan en ésta.

En el siguiente ejemplo se dibujan en la misma figura las gráficas de las funciones seno y coseno en el intervalo $[-3\pi, 3\pi]$. El resultado obtenido se muestra en la figura 4.15.

```
hold on  
x=[-3*pi:pi/10:3*pi];  
plot(x,sin(x))  
plot(x,cos(x), '-.')  
legend('seno(x)', 'coseno(x)');  
hold off
```

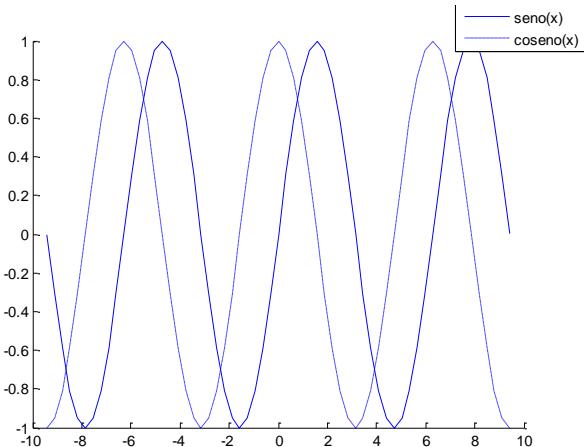


Figura 4.15 Superposición de gráficos en la misma figura

■ Creación de subventanas en la misma figura

Para dividir una ventana de visualización en diferentes subventanas en las que se puedan representar gráficos distintos se utiliza el comando:

```
subplot(m, n, p)
```

que crea una ventana de visualización dividida en $m \times n$ subventanas y crea ejes para dibujar en cada una de ellas, activándose la subventana p -ésima (contando en orden filas). Obsérvese el siguiente ejemplo y el gráfico resultante en la figura 4.16.

```

x=-3*pi:pi/10:3*pi;
y=sin(x);
z=cos(x);
t=tan(x);
s=sec(x);
subplot(2,2,1); % se activa la primera subventana
plot(x,y);
title('seno(x)');
subplot(2,2,2); % se activa la segunda subventana
plot(x,z);
title('coseno(x)');
subplot(2,2,3); % se activa la tercera subventana
plot(x,t);
title('tangente(x)');
subplot(2,2,4); % se activa la cuarta subventana
plot(x,s);
title('secante(x)');

```

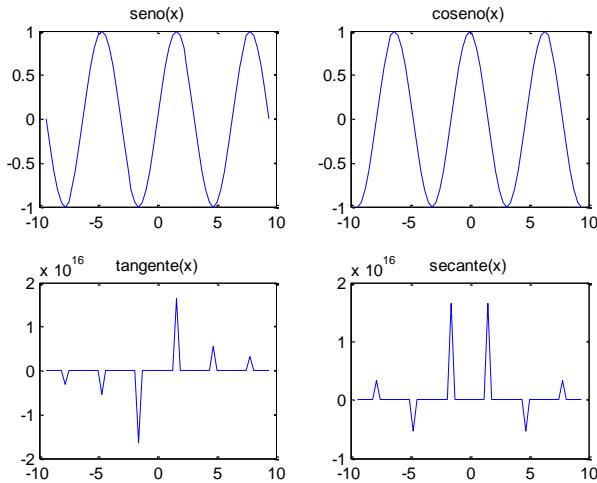


Figura 4.16 Gráficos en diferentes subventanas de la misma figura

4.4 Visualización de campos vectoriales y curvas de nivel

▪ Campos vectoriales en dos dimensiones

Dado un campo vectorial $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ definido en un dominio D, se puede visualizar dicho campo utilizando la siguiente función de MATLAB y Octave:

```
quiver(Mx, My, U, V)
```

que dibuja vectores en los puntos de la malla dada por las matrices M_x, M_y. Los vectores se dibujan como flechas con componentes (U, V). Las matrices M_x, M_y, U, V deben tener todos el mismo tamaño y contener las correspondientes componentes de posición y valores del campo. quiver automáticamente escala las flechas para que se ajusten al mallado.

Ejemplo: Procedemos ahora a representar el campo vectorial $F = [x^2 y, 3\sin(x+y)]$ en $D = [-1,1] \times [-1,1]$.

Usamos el programa M siguiente:

```
x=linspace(-1,1,10);  
  
y=x;  
  
[Mx,My]=meshgrid(x,y);  
  
U=Mx.^2.*My;  
  
V=3*sin(Mx+My);  
  
quiver(Mx,My,U,V)  
  
axis equal  
  
xlabel('Eje x')  
ylabel('Eje y')
```

El resultado se muestra en la figura 4.17.

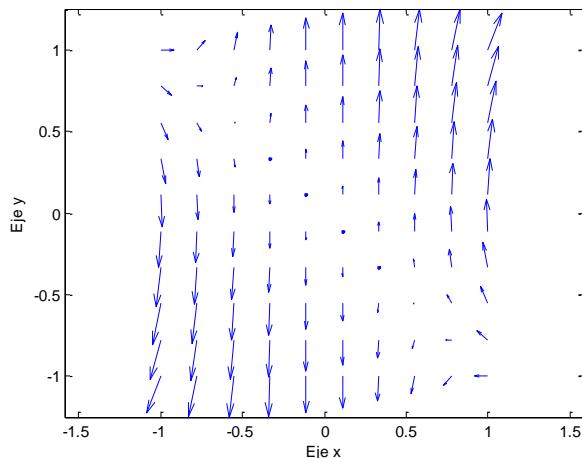


Figura 4.17 Campo vectorial 2D obtenido con quiver

■ Campos vectoriales en tres dimensiones

MATLAB y Octave también permiten visualizar campos vectoriales del tipo:
 $f : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. Para ello se dispone de la siguiente función:

```
quiver3 (Mx, My, Mz, U, V, W)
```

que dibuja los vectores del campo en los puntos de la malla tridimensional dada por Mx, My, Mz. Los vectores se representan como flechas con componentes (U,V,W). Las matrices Mx, My, Mz, U, V, W tienen que ser todas del mismo tamaño y contener las correspondientes componentes de posición y del campo. quiver3 automáticamente escala las flechas para que se puedan visualizar correctamente.

Ejemplo. Consideremos el siguiente campo vectorial definido en $[-3,3] \times [-3,3] \times [-1,1]$:

$$F(x, y, z) = [1, x^2 + y^2, 3z]$$

Para visualizar este flujo tridimensional, se va a utilizar el siguiente código:

```
x=linspace (-3,3,6);
y=x;
z=linspace (-1,1,6);
[Mx,My,Mz]=meshgrid(x,y,z)
U=1+0*Mx;
V=Mx.^2+My.^2;
W=3*Mz;
quiver3 (Mx, My, Mz, U, V, W)
 xlabel('Eje x','Fontsize',16)
 ylabel('Eje y','Fontsize',16)
 zlabel('Eje z','Fontsize',16)
 grid off
```

El resultado se muestra en la figura 4.18.

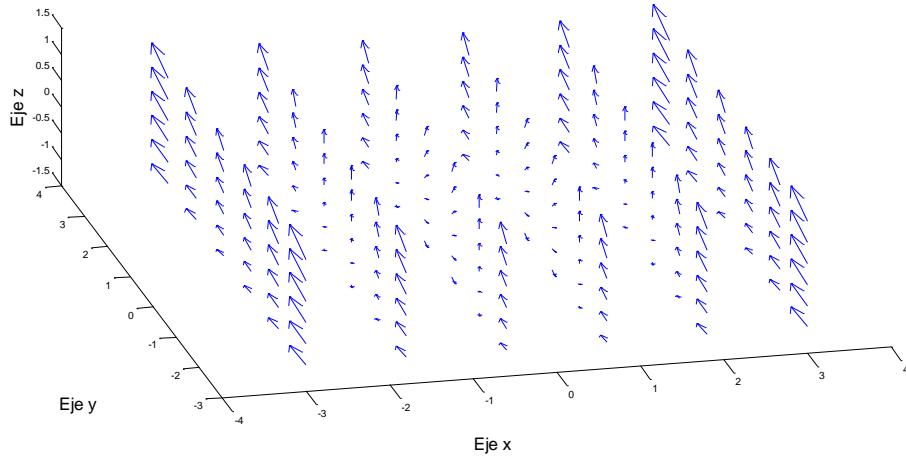


Figura 4.18 Campo vectorial 3D obtenido con quiver3

■ Curvas de nivel

El conjunto de puntos (x, y) donde una función toma un determinado valor es llamado conjunto de nivel de la función. Si la función es $f(x, y)$, y el valor que toma la función es c podemos expresarlo:

$$S_c = \{(x, y) : f(x, y) = c\}$$

A menudo, estos conjuntos son curvas en el plano x-y y son llamados **curvas de nivel**. Podemos visualizar estas curvas cortando el gráfico tridimensional de la función $f(x, y)$, con un plano $Z = c$ y proyectando el conjunto resultante sobre el plano x-y.

Para representar las curvas de nivel, se suele dibujar en el mismo gráfico, además de éstas, la superficie proyectada sobre el plano x-y. Para realizar estas tareas MATLAB y Octave disponen de las siguientes funciones:

```
pcolor (Mx, My, Mz)
```

Crea el gráfico igual que se haría con el comando `surf` pero con un punto de vista $z \rightarrow +\infty$.

```
contour (Mx, My, Mz, N, str)
```

Dibuja líneas de nivel de la superficie definida por las matrices M_x , M_y , M_z , con el color y estilo de línea definido en `str`. El número y/o posición de las líneas de nivel queda definido por el cuarto parámetro `N`. Existen estas posibilidades:

- Si `N` es un escalar, dibuja `N` líneas de nivel de forma automática.
- Si `N` es un vector, dibuja `length(N)` líneas de nivel en los valores de `z` especificados en los elementos de `N`.
- Si `N` es un vector de dos elementos de valores iguales dibuja un sólo contorno al nivel especificado en los elementos de `N`.

Ejemplo. Se quieren dibujar curvas de nivel de una superficie dada por la siguiente función en el dominio $[-2, 2] \times [-2, 3]$:

$$f(x, y) = 6e^{-3x^2-y^2} + x/2 + y$$

Se comienza obteniendo las matrices de la retícula en el dominio $[-2, 2] \times [-2, 3]$.

```

x=-2:0.05:2; y=-2:0.05:3;
[Mx,My]=meshgrid(x,y);
Mz=6*exp(-3*Mx.^2-My.^2)+0.5*Mx+My;

```

Continuamos dibujando en la primera subventana la superficie de la forma habitual (`surf`) y en la segunda subventana la superficie desde $z \rightarrow +\infty$ (`pcolor`). Mantenemos activa esta segunda subventana (`hold on`).

```

subplot(1,2,1);
surf(Mx,My,Mz);
shading flat
subplot(1,2,2)
pcolor(Mx,My,Mz);
shading flat
hold on

```

Se añade ahora el dibujo de 7 líneas de nivel:

```

contour(Mx,My,Mz,7,'k')
hold off

```

El resultado aparece en la figura 4.19.

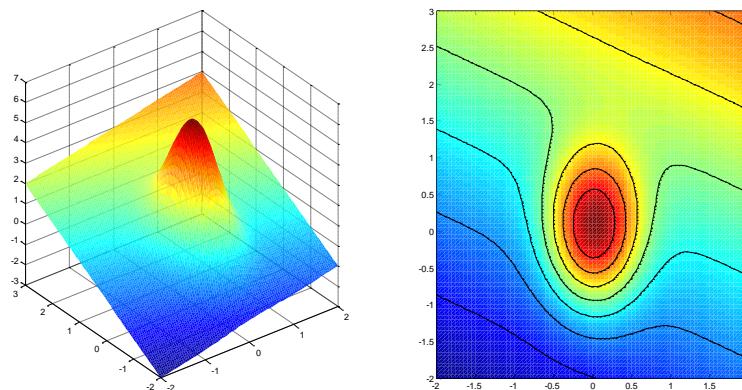


Figura 4.19 Surf frente a pcolor y contour. Se representan 7 líneas de nivel

4.5 Practicando con programas (IV). Ejercicios de autoevaluación del módulo.

Ejercicio 4.5.1 Se considera la curva definida en coordenadas polares por la ecuación:

$$\rho(\theta) = \frac{\sin(2\theta)}{\cos^3(\theta) + \sin^3(\theta)}.$$

- Escribir un programa M que dibuje en línea continua de color rojo, la parte de la curva comprendida entre $0 \leq \theta \leq \frac{\pi}{2}$. Se tomarán, en dicho intervalo, 10^3 valores y se mostrará la rejilla correspondiente a las marcas de los ejes.
- Añadir las sentencias necesarias al programa anterior para que calcule la longitud (aproximada) de la curva dibujada, como suma de las longitudes de los tramos rectos parciales, correspondientes a la discretización utilizada, mostrando en pantalla el resultado obtenido.

Ejercicio 4.5.2 Cuando se intentan dibujar algunas funciones en dos y tres dimensiones, a veces sucede que la función toma valores muy grandes (o muy pequeños) en regiones de tamaño pequeño comparado con el dominio. Cuando sucede esto, la escala vertical del gráfico es tal que la mayoría de los detalles se pierden. En este ejercicio veremos una técnica de recortado para poder mostrar el gráfico con más detalle.

Sea la función:

$$f(x, y) = \operatorname{sen}\sqrt{x^2 + y^2} + 1 / (x^2 + y^2 + 0.001)$$

- Dibujar la gráfica '*normal*' de esta función y observar la pérdida de detalle que se produce en un entorno del punto (0,0).
- Representar ahora la misma superficie pero cambiando la z de los puntos cuya cota sea superior a 4 unidades a ese valor.

Ejercicio 4.5.3 Sea la siguiente ecuación de una superficie dada en coordenadas cilíndricas, $\rho = |z^3 - 2|$, $z \in [-1, 1]$, $\theta \in [0, 2\pi]$.

Escribir un programa M que logre dibujar la superficie correspondiente.

Ejercicio 4.5.4

Sea la superficie cuyas ecuaciones paramétricas son las siguientes:

$$\begin{aligned}x &= \cos(u)(2 + \sin(v)), \\y &= \sin(u)(2 + \sin(v)), \\z &= u + \cos(v), \quad u \in [0, 4\pi], \quad v \in [0, 2\pi].\end{aligned}$$

Representar en dos subventanas en horizontal de la misma ventana gráfica, en primer lugar la superficie indicada (en una escala de los ejes [-3,3] para x , y , y [0,10] para z) y en segundo lugar las curvas de nivel correspondientes a los valores de $z=3,5$ y 7 .

¿Cómo se modificaría el programa para dibujar sólo la curva de nivel en $z=3$? ¿y para dibujar 10 curvas de nivel de forma automática? .

Ejercicio 4.5.5

Consideremos los siguientes campos vectoriales $f : \mathbb{R}^3 \rightarrow \mathbb{R}^3$:

a) $f(x, y, z) = [xy, yz, x^2 + y^2]$

b) $f(x, y, z) = [x^2y, \cos z, x - y + z^2]$

Dibujar las gráficas de ambos campos, en dos ventanas de visualización distintas, en el recinto $[-2, 2]^3$.

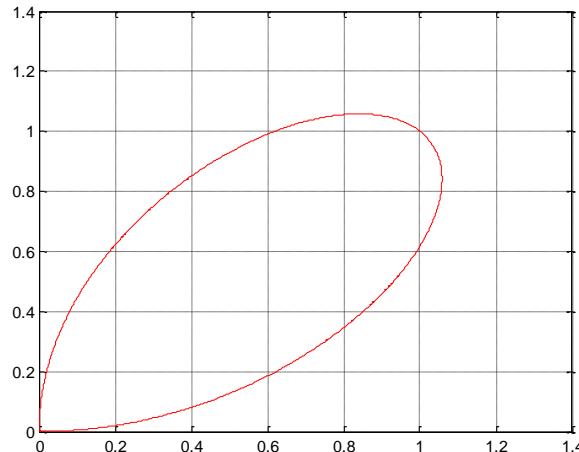
Solución Ejercicio 4.5.1

El programa que obtiene los resultados pedidos en los dos apartados es el siguiente:

```
t = linspace( 0 , pi/2 , 1000 );
r = sin(2*t) ./ ( cos(t).^3 + sin(t).^3 );
x = r .* cos(t);
y = r .* sin(t);
plot( x , y , 'r')
grid on
long = sum(sqrt((x(2:end) - x(1:end-1)).^2 + (y(2:end) - ...
y(1:end-1)).^2));
fprintf( 'Longitud = %20.8f\n' , long )
```

Obsérvese que en este programa, para calcular la longitud de la curva, no se han utilizado bucles. El lector puede llegar a una solución distinta usándolos.

La gráfica obtenida es

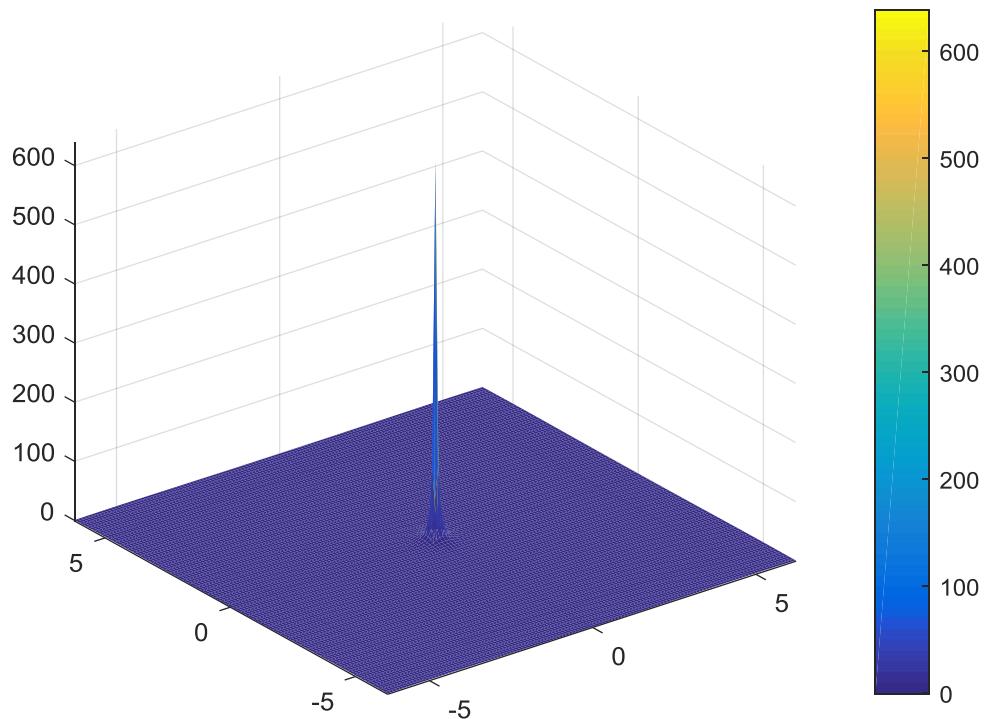


Solución Ejercicio 4.5.2

Para representar la superficie sin ningún tipo de modificación, se utilizaría el programa siguiente:

```
[X,Y]=meshgrid(-2*pi:.1:2*pi, -2*pi:.1:2*pi);  
Z=sin(sqrt(X.^2+Y.^2))+(X.^2+Y.^2+0.001).^(-1);  
surf(X,Y,Z);  
shading flat  
axis tight;  
colorbar
```

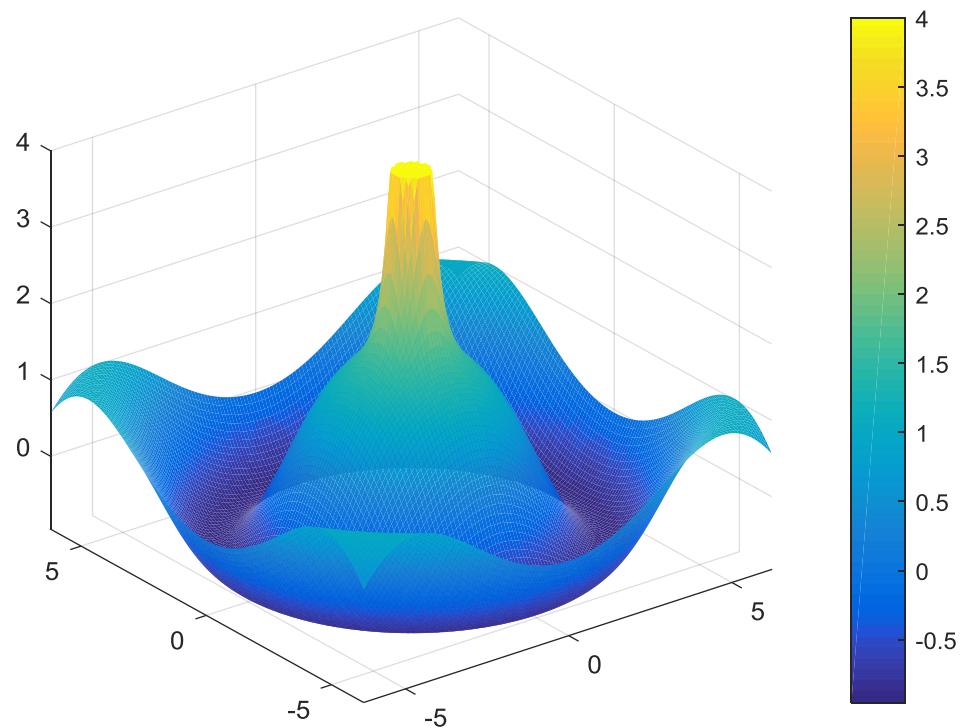
El resultado es:



Para realizar el recortado del gráfico, se debe utilizar el programa:

```
[X,Y]=meshgrid(-2*pi:.1:2*pi);
Z= sin(sqrt(X.^2+Y.^2))+(X.^2+Y.^2+0.001).^(−1);
Z(Z>4)=4;
surf(X,Y,Z);
shading flat
axis tight;
colorbar
```

El resultado es:



Solución Ejercicio 4.5.3

Al sustituir las ecuaciones de la superficie en las ecuaciones de transformación de cilíndricas a cartesianas, quedan las siguientes:

$$X = \text{abs}(z^3 - 2) \cos(\theta)$$

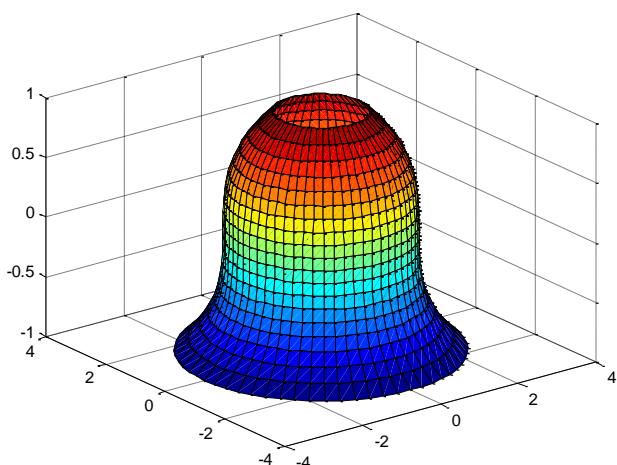
$$Y = \text{abs}(z^3 - 2) \sin(\theta)$$

$$Z = z \quad z \in [-1, 1], \quad \theta \in [0, 2\pi]$$

El código MATLAB / Octave sería:

```
tt=0:pi/30:2*pi;
z=linspace(-1,1,20);
[Z,T]=meshgrid(z,tt);
X=abs(Z.^3-2).*cos(T);
Y= abs(Z.^3-2).*sin(T); % Z=z
surf(X,Y,Z)
```

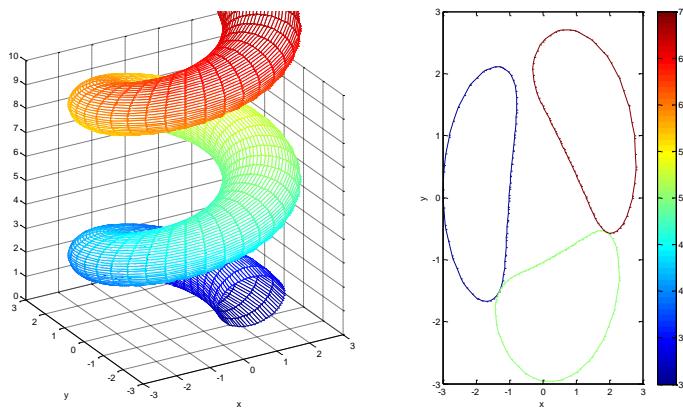
Y la superficie obtenida se ofrece a continuación:



Solución Ejercicio 4.5.4

A continuación aparece el programa M y la figura obtenida.

```
u=0:pi/18:4*pi;
v=0:pi/32:2*pi;
[Mu,Mv]=meshgrid(u,v);
Mx=cos(Mu).* (2+sin(Mv));
My=sin(Mu).* (2+sin(Mv));
Mz=Mu+cos(Mv);
subplot(1,2,1)
mesh(Mx,My,Mz)
xlabel('x')
ylabel('y')
axis([-3 3 -3 3 0 10])
subplot(1,2,2)
contour(Mx,My,Mz,[3,5,7])
xlabel('x')
ylabel('y')
colorbar
```



Para dibujar una línea de nivel en $z=3$,

```
contour(Mx,My,Mz,[3,3])
```

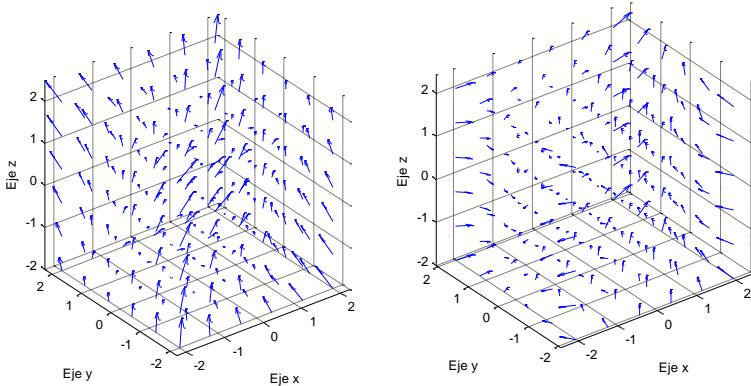
Para dibujar 10 líneas de nivel de forma automática,

```
contour(Mx,My,Mz,10)
```

Solución Ejercicio 4.5.5

Si se ejecuta el siguiente programa M se obtienen las figuras que aparecen después.

```
x=linspace (-2,2,6) ;  
  
y=x;  
  
z=x;  
  
[Mx,My,Mz]=meshgrid(x,y,z);  
  
U1=Mx.*My;  
  
V1=My.*Mz;  
  
W1=Mx.^2+My.^2;  
  
figure  
  
quiver3(Mx,My,Mz,U1,V1,W1);  
  
axis image  
  
xlabel('Eje x')  
  
ylabel('Eje y')  
  
zlabel('Eje z')  
  
U2=Mx.^2.*My;  
  
V2=cos (Mz) ;  
  
W2=Mx-My+Mz.^2;  
  
figure  
  
quiver3(Mx,My,Mz,U2,V2,W2);  
  
axis image  
  
xlabel('Eje x')  
  
ylabel('Eje y')  
  
zlabel('Eje z')
```



5.1 Programación modular.

Una forma eficaz de resolver un problema es dividirlo en tareas sencillas que puedan tratarse con facilidad. Esta técnica también se usa en programación y se denomina programación modular, igualmente conocida como de diseño descendente o top-down.

Observemos la figura 5.1 en la que se simula un programa que resuelve un problema determinado tal como lo conocemos hasta ahora, es decir, realizando todas las tareas del problema.

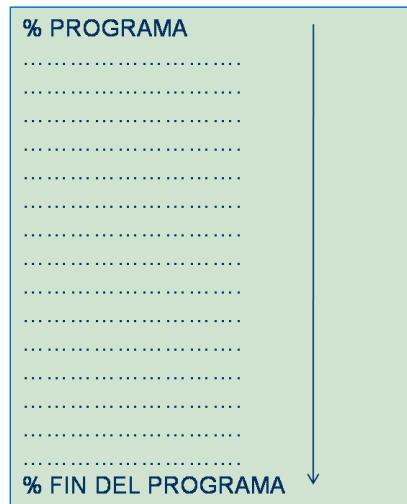


Figura 5.1 Sin programación modular

Cuando se ejecute el programa se realizarán todas las tareas desde el mismo archivo .m

Veamos en la figura 5.2 la diferencia que se tendría al usar programación modular.

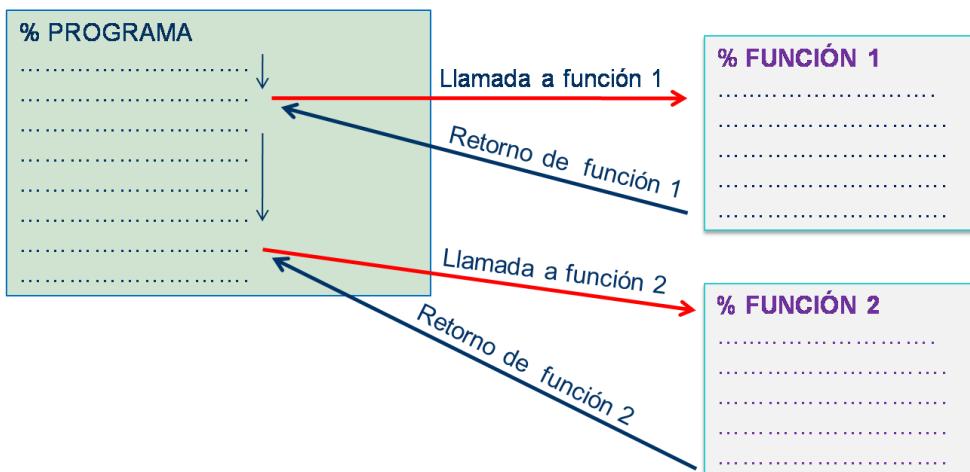


Figura 5.2. Con programación modular

En este caso nos encontramos con tres archivos .m. Uno del programa, y dos de funciones que apoyan al programa realizando tareas específicas.

El funcionamiento es como sigue: cuando se ejecute el programa, se comienza desarrollando la primera sentencia, en la segunda instrucción se produce la llamada a la función 1 que traslada el flujo del programa a un archivo .m distinto, cuando finaliza éste se regresa al programa principal en el mismo lugar en el que se había producido la llamada, se continúa ejecutando sus sentencias, hasta que se produce la llamada a la función 2, que pasa el flujo del programa a otro archivo .m, cuando finaliza éste se retorna al programa principal, realizando las sentencias que faltan para la terminación de éste.

La programación modular presenta las siguientes ventajas:

- Claridad: aporta claridad al código al estar dividido en tareas sencillas.
- Desarrollo independiente o en paralelo: permite desarrollar cada módulo de manera independiente, depurando los errores de cada parte sin que afecte al resto.
- Reutilización del código: los módulos pueden reutilizarse en otros programas.

En lenguaje M para poder trabajar con módulos se deben definir funciones que realicen las tareas específicas necesarias para cada problema. En la próxima lección se aprenderá a definir funciones y a utilizarlas desde un programa u otras funciones.

5.2 Funciones definidas por el programador

Las funciones definidas por el usuario en lenguaje M, al igual que los programas, se escriben en ficheros .m. En este caso es necesario que el nombre de la función y del fichero que la contiene coincidan.

En un fichero .m de nombre *nombre_funcion.m* se escribe la función de acuerdo a la siguiente sintaxis:

```
function [s_1,s_2,...] = nombre_funcion(e_1,e_2,...)
sentencias
s_1=.....
s_2=.....
end % opcional
```

Siendo *s_i* el argumento de salida i-ésimo , y *e_i* el argumento de entrada i-ésimo.

La llamada a la función se realiza de la misma manera que se indicó con las funciones de librería, es decir:

```
[B_1,B_2,...]= nombre_funcion(A_1,A_2,...)
```

Veamos unos ejemplos sencillos que sirvan de iniciación a la utilización de funciones.

Ejemplo 1. Se desea diseñar una función de nombre cuad que reciba un dato numérico y devuelva el cuadrado de éste. La función es la siguiente:

```
function y=cuad(x)
y=x*x;
end
```

Una llamada a la función y el resultado obtenido se muestra a continuación:

```
>>z=cuad(7)
z= 49.0000
```

Ejemplo 2. Se quiere escribir la función tiempoinv que reciba una matriz cuadrada y devuelva el tiempo que tarda en calcular la inversa. Si la matriz de entrada no es cuadrada debe finalizar la función dando valor 0 al dato de retorno. Un código válido aparece a continuación:

```

function t=tiempoinv(A)
[m,n]=size(A);

if m~=n
    t=0;
    return;
end

tic;
inv(A);
t=toc;
end

```

El siguiente programa realiza una llamada a la función anterior, escribiendo en pantalla el resultado.

```

B=[3,4;6,-8];
tiempo=tiempoinv(B);
fprintf('El tiempo de ejecución de la inversa es %f\n', tiempo);

```

Se imprime en pantalla:

El tiempo de ejecución de la inversa es 0.000029

Ejemplo 3. Sea la siguiente función matemática definida a trozos:

$$f(x) = \begin{cases} 0, & \text{si } x \leq -3 \\ x^3, & \text{si } -3 < x < -2 \\ x^2, & \text{si } -2 \leq x \leq 2 \\ x, & \text{si } 2 < x < 3 \\ 0, & \text{si } x \geq 3 \end{cases}$$

Se va a diseñar una función M de nombre trozos que reciba un dato numérico que se corresponda con una abscisa y devuelva el valor correspondiente a esa abscisa según la definición de la función matemática anterior. El código de la función trozos es el siguiente:

```

function y=trozos(x)

if x<=-3
    y=0;
elseif -3<x & x<-2
    y=x^3;
elseif -2<=x & x<=2
    y=x^2;
elseif 2<x & x<3
    y=x;
elseif x>=3
    y=0;
end

```

Ejemplo 4. A continuación se escribe la función `stat` que recibe un conjunto de datos en forma de matriz y realiza dos cálculos estadísticos: la media de los datos y la desviación típica. Dado el conjunto de datos x_1, x_2, \dots, x_N , se definen la media y la desviación típica como sigue:

$$\bar{x} = \frac{\sum_{i=1}^N x_i}{N}, \quad \sigma = \sqrt{\frac{\sum_{i=1}^N (xi - \bar{x})^2}{N}}$$

El código de la función `stat` se escribe a continuación:

```

function [media,destip]=stat(v)

v=v(:);
lon=length(v);
media=sum(v)/lon;
destip=sqrt(sum((v-media).^2)/lon);

```

La llamada a esta función, se puede realizar del siguiente modo:

```

>> [a,s]=stat([1 5 6 7 9])
a =
5.6000
s =
2.6533

```

- **Ámbito de las variables de una función**

Puede observarse que cuando se realiza una llamada a una función, las variables que se utilizan en el código de la función no aparecen en el espacio de trabajo. Esto es debido a que las variables que intervienen en un fichero de función son locales: su ámbito es la propia función y no son visibles desde otra función o programa.

Supongamos que se ejecuta el siguiente programa que hace una llamada a la función cuad.

Programa

```
m=input('Introduce un numero');  
z=cuad(m);
```

Función

```
function y=cuad(x)  
y=x^2;
```

Si a la petición del dato en la primera sentencia del programa se responde tecleando el número 6, las variables que aparecen en el espacio de trabajo y por tanto son accesibles desde otros programas son las siguientes:

variable	valor
m	6
z	36

Las variables propias de la función, que no son utilizables desde fuera de ella, son:

variable	valor
x	6
y	36

Pongamos ahora el mismo ejemplo cambiando los nombres de las variables lo que puede aumentar el nivel de dificultad en la interpretación.

```
x=input('Introduce un numero');  
y=cuad(x);
```

```
function x=cuad(y)  
x=y^2;
```

Si a la petición del dato en la primera sentencia se responde tecleando el número 6, las variables que aparecen en el espacio de trabajo y por tanto son accesibles desde otros programas son las siguientes:

variable	valor
x	6
y	36

Las variables propias de la función, que no son utilizables desde fuera de ella, son:

variable	valor
y	6
x	36

No hay confusión posible entre ellas porque su ámbito es distinto.

5.3 Ficheros de datos (1)

Hasta el momento, la comunicación de nuestros programas con el entorno se ha hecho a través de la ventana de comandos, para imprimir los datos que salen del programa o para teclear los datos que se quieren suministrar al programa. Sin embargo, en el caso de lectura de datos, si se tiene que suministrar un número elevado de datos que no cambia en cada ejecución, no es viable tener que teclearlos en cada ejecución. Asimismo, en el caso de escritura de datos, éstos se perderán al salir de MATLAB u Octave. Es necesario que nuestros programas puedan comunicarse con ficheros almacenados en disco para leer los datos presentes en él o volcar resultados de ejecución. Veremos a continuación las principales operaciones para el manejo de ficheros de datos.

- **Apertura del fichero**

Para poder acceder a un fichero desde un programa M se debe realizar una primera operación de apertura mediante la función `fopen`, que pondrá en comunicación el programa con el fichero. Se usará la siguiente sintaxis:

```
variable=fopen ('nomrefichero', 'permiso')
```

donde:

`variable` es el identificador del fichero. A partir de este momento M trabaja con el fichero por medio de este identificador, es decir, no volverá a usarse el nombre externo. El contenido de esta variable es un entero positivo si el proceso de apertura se ha realizado correctamente, por el contrario si toma el valor -1 indica un error, por ejemplo, que el fichero no ha sido encontrado.

'`nomrefichero`' especifica el nombre externo del fichero, si su ubicación es la carpeta actual, o la ruta completa al fichero, si se encuentra en una ubicación distinta de la actual.

'`permiso`' indica el modo de apertura del fichero. Los más utilizados son:

r abre un fichero existente para leer datos de él.

w crea un fichero nuevo para escritura. Si ya existe se borra su contenido

a abre un fichero para escritura. Si existe escribe al final de lo ya escrito.

■ Cierre del fichero

Cuando se haya finalizado la utilización del fichero se debe desvincular éste del programa empleando la función `fclose` con la siguiente sintaxis:

```
fclose(fid)
```

que cierra el fichero de identificador `fid`, devolviendo 1 si el cierre es correcto y 0 si es incorrecto.

■ Escritura de datos en ficheros

Para escribir en un fichero cualquier tipo de dato con el formato deseado por el usuario se utiliza la función `fprintf` de manera similar a como se usó `fprintf` para escribir en pantalla. La sintaxis es la siguiente:

```
fprintf(fid,'format',A,...)
```

que escribe los elementos especificados en `A` (que en general es una matriz) en el fichero de identificador `fid` (previamente abierto) con el formato especificado en '`format`'.

Como ejemplo, si se ejecuta el siguiente programa:

```
x=0:.1:1  
y=[x;exp(x)];  
  
fid=fopen('resultado.txt','w');  
  
fprintf(fid,'%4s %12s \n', 'x', 'exp(x)');  
fprintf(fid,'%6.2f %12.8f\n', y);  
  
fclose(fid)
```

se generará el fichero `resultado.txt` con el siguiente contenido:

x	exp (x)
0.00	1.00000000
0.10	1.10517092
0.20	1.22140276
0.30	1.34985881
.....
1.00	2.71828183

- **Rebobinado de ficheros**

Cuando se abre un fichero el cursor de lectoescritura se encuentra en posición inicial (primera fila y primera columna). Según se va escribiendo o leyendo, este cursor se desplaza por el fichero. Si en un momento determinado se quiere volver a la posición inicial se puede conseguir mediante la siguiente función

```
frewind(fid)
```

que rebobina el fichero con identificador fid.

- **Encontrar el final del fichero en la lectura**

La función de retorno lógico feof contiene información sobre la situación del cursor de lectoescritura respecto al final del fichero. Su sintaxis se indica a continuación:

```
st=feof(fid)
```

Devuelve el valor 1 si el indicador del final del fichero de identificador fid ha sido alcanzado y 0 en otro caso, tal como se indica en la siguiente tabla:

Situación en el fichero:	feof(fid)	~feof(fid)
antes del final	0	1
en el final	1	0

Se puede utilizar para repetir una lectura de dato mientras no se esté en el final, es decir mientras ~feof(fid) sea cierto. Una estructura típica de utilización se ofrece a continuación:

```
fid=fopen('res.txt','r'); %apertura de fichero  
n=0;  
  
while (~feof(fid))  
  
    ---%leer un dato (se estudia en siguiente lección)  
  
    n=n+1;%n contabiliza el número de datos leidos  
  
end  
  
fclose(fid)
```

5.4 Ficheros de datos (2)

▪ Lectura de datos de ficheros

La lectura de los datos contenidos en un fichero se puede realizar a partir de varias funciones, entre ellas utilizaremos las siguientes:

fgetl, que lee una línea completa del fichero en una cadena de caracteres.

fgets, que lee un número determinado de caracteres o una línea completa almacenándolo en una cadena de caracteres.

fscanf, que lee datos de diferentes tipos según un formato especificado por el programador.

Veremos con más detalle estas funciones a continuación.

- Función fgetl

Para leer una línea completa del fichero se puede utilizar la función fgetl con la siguiente sintaxis:

Cadena=fgetl(fid)

leyendo, en la variable Cadena, todos los caracteres desde la posición actual del cursor de lectoescritura hasta el final de la línea, sin incluir el carácter salto de línea o fin de fichero. La función devuelve el valor -1 si se encuentra con el final del fichero, con ello, una forma de contar el número de líneas de un fichero sería:

```
f=fopen('fichero.csv','r');

n=0;

while 1

    linea=fgetl(f);

    if linea== -1, break, end

    n=n+1;

end

fprintf('%d líneas\n',n);

fclose(f);
```

- Función fgets

La sintaxis a emplear para leer líneas completas con fgets es:

Cadena=fgets(fid)

que lee desde la posición actual del cursor hasta el final de la línea y lo guarda en la cadena de caracteres Cadena. En este caso formarán parte de la cadena el salto de línea con el que termina cada línea, luego la lectura es más *limpia* con fgetl.

Como la lectura de los datos de un fichero se realiza de manera secuencial, es decir, para llegar a un dato hay que pasar por todos los anteriores, es necesario usar funciones de lectura que permitan de una manera ágil llegar a la línea deseada aunque lo leído no se vaya a usar en el programa. Las funciones anteriores, además de poder leer el texto de líneas completas, se pueden utilizar de manera auxiliar para llegar a una línea determinada.

Para leer un número determinado de caracteres desde la posición actual, se utiliza

```
Cadena=fgets(fid,n)
```

que lee un máximo de *n* caracteres (si antes de conseguir leer *n* caracteres encuentra un salto de línea o el final del fichero detiene la lectura) almacenándolo en la cadena de caracteres Cadena.

- Función fscanf

Para leer cualquier tipo de dato se utiliza esta función

```
[A,cont]=fscanf(fid,'formato',size)
```

que lee datos con el formato especificado en 'formato' del fichero abierto con el identificador *fid* y los almacena en las columnas de la matriz *A* de tamaño *size*. El segundo argumento de retorno *cont* guarda el número de datos leídos, no es necesario incluirlo si no se necesita esa información.

En lugar del tamaño *size*, se puede incluir un número entero *n*, entonces se hará referencia a el número máximo de datos que se quieren leer, almacenándose en el vector columna *A*:

```
[A,cont]=fscanf(fid,'formato',n)
```

Como formato se incluirán únicamente especificadores de formato de datos.

Los formatos más utilizados para lectura de datos son:

%d : adecuado para leer un dato entero.

%f : adecuado para leer un dato real.

%s : adecuado para leer texto hasta el primer espacio en blanco.

%c : adecuado para leer un carácter. Si se quieren leer varios caracteres se indicará el número de la siguiente forma %nc donde *n* es el número de caracteres a leer. En este caso no se detiene al encontrar el espacio en blanco.

Igual que ocurre en la escritura, el formato se reutiliza para poder leer todos los datos que se indiquen.

Si no se indica tamaño `size`, ni número de datos a leer `n`, se leerán todos los datos del fichero (si es posible) desde la posición actual, guardándose en el vector columna `A`:

```
[A, cont]=fscanf(fid, 'formato')
```

Otra forma de poder leer todos los datos del fichero es indicar un tamaño de lectura `size` en el que el número de columnas se deja variable, utilizando `inf`.

```
[A, cont]=fscanf(fid, 'formato', [numfil, inf])
```

En este caso se leerán todos los datos desde la posición actual del fichero, almacenándose en orden columnas en la matriz `A`. `A` tendrá `numfil` filas, y el número de columnas necesario para leer todos los datos. Si una columna se queda inacabada se completa con ceros.

■ Ejemplos resueltos de lectura de datos de ficheros

Ejemplo 1. Se supone que en la carpeta de trabajo actual se encuentra un archivo de nombre `datos.txt`, cuyo contenido es:

```
1 2 3 4 5  
6 7 8 9 10
```

Si se ejecuta el siguiente programa:

```
fid=fopen('datos.txt', 'r')  
[A, cont]=fscanf(fid, '%d')
```

La salida a pantalla es el vector columna `A` que contiene todos los datos del fichero: 1 2
3 4 5 6 7 8 9 10 y `cont=10`.

Si se añade al programa la siguiente sentencia:

```
[A, cont]=fscanf(fid, '%d', [2, 5])
```

se obtiene

```
A= ''  
cont=0
```

Este resultado se debe a que el fichero ha sido anteriormente leído por completo y ya no hay datos desde la posición actual del cursor de lectoescritura. Es necesario rebobinarlo si se quiere leer de nuevo. Entonces, si cambiamos la sentencia anterior por las siguientes:

```
frewind(fid)  
[A, cont]=fscanf(fid, '%d', [2, 5])
```

se observa que se leen los datos desde el comienzo; se leerán 2x5 (10) datos almacenándose en la matriz A de ese tamaño en orden columnas.

```
A=
```

```
1 3 5 7 9
```

```
2 4 6 8 10
```

```
cont=10
```

Añadiendo las siguientes sentencias se obtiene el resultado indicado:

```
frewind(fid)
```

```
[A,cont]=fscanf(fid,'%d',[3,3])
```

```
A=
```

```
1 4 7
```

```
2 5 8
```

```
3 6 9
```

```
cont=9
```

```
frewind(fid)
```

```
[A,cont]=fscanf(fid,'%d',[4,4])
```

```
A=
```

```
1 5 9
```

```
2 6 10
```

```
3 7 0
```

```
4 8 0
```

```
cont=10
```

Puede observarse que el tamaño indicado es 4x4 pero como sólo hay 10 datos en el fichero, sólo utiliza las columnas que necesita para almacenar todos los datos, en este caso tres. Si faltan datos para completar la última columna se añaden ceros.

```
frewind(fid)
```

```
[A,cont]=fscanf(fid,'%d',[4,inf])
```

```
fclose(fid)
```

Se está indicando con inf que se utilice el número de columnas mínimo necesario para almacenar todos los datos.

El resultado obtenido es el mismo que en el caso anterior a éste.

Ejemplo 2. Sea ahora el fichero `resultados.txt` que se encuentra en la carpeta de trabajo con el siguiente contenido:

Calificaciones del examen final			
Nombre y apellidos	Ej1	Ej2	Ej3
Juan Segundo Mole	9	4.5	6
Raquel Chueca Rivera	7	8.1	3.9
.....			

El fichero tiene dos líneas de encabezamiento (que explican la información que contiene) y después un número indeterminado de líneas que contienen el nombre del alumno (en un campo máximo de 30 caracteres) y las tres calificaciones de los ejercicios. Se necesita hacer un programa que calcule la media de las calificaciones medias del examen. Una solución posible aparece a continuación:

```
fid=fopen('notas.txt','r');

aux=fgetl(fid);

aux=fgetl(fid);

n=0;

med=[];

while(~feof(fid))

    aux=fgets(fid,30);

    not=fscanf(fid,'%f',[1,3]);

    aux=fgets(fid);%para comerse el salto de línea después del tercer número

    med=[med,sum(not)/3];

    n=n+1;

end

fprintf('La media de notas es: %f\n',sum(med)/n);
```

5.5 Practicando con programas (V). Ejercicios de autoevaluación del módulo.

Ejercicio 5.5.1 Construir una función llamada `area` que calcule el área de un triángulo de vértices `p`, `q` y `r`.

Una vez construida, realizar la llamada a la función desde la ventana de comandos con los siguientes datos para los vértices: (1,0,0); (0,1,0); (0,0,1) y almacenar el resultado en la variable `areatri`. Si la función es correcta, el resultado que se debe obtener es 0.8660.

Nota: El área de un triángulo puede calcularse obteniendo el módulo del producto vectorial de dos de sus lados con origen en el mismo vértice y dividiendo entre dos esta cantidad.

Ejercicio 5.5.2 Construir una función llamada `sfac` tal que dado un número natural `n`, calcule la suma de 1 a `n` y el factorial de `n`. Hacer la llamada a la función para realizar el cálculo con el valor 8. Guardar el resultado en dos variables `sum` y `fac`.

Ejercicio 5.5.3 Diseñar una función de nombre `burbuja` que reciba un vector `v` y devuelva el mismo vector ordenado de menor a mayor utilizando el método de la burbuja.

El *Método de la Burbuja* utiliza de forma masiva el algoritmo de intercambio. Se procede a ir comparando elementos consecutivos dos a dos, de tal manera que si están desordenados, se pasa a intercambiarlos. Este proceso aplicado una vez de izquierda a derecha consigue situar el elemento mayor a la derecha. Si se repite el proceso, el elemento más grande de los restantes queda situado al lado del mayor y así sucesivamente.

Veamos esto con un ejemplo. Sea el vector a ordenar: 5 4 3 2 1

Primera etapa de comparaciones:

5 4 3 2 1	→	4 5 3 2 1
4 5 3 2 1		4 3 5 2 1
4 3 5 2 1		4 3 2 5 1
4 3 2 5 1		4 3 2 1 5

Segunda etapa de comparaciones:

4 3 2 1 5	3 4 2 1 5
3 4 2 1 5	3 2 4 1 5
3 2 4 1 5	3 2 1 4 5

Tercera etapa de comparaciones:

3 2 1 4 5	2 3 1 4 5
2 3 1 4 5	2 1 3 4 5

Cuarta y última etapa de comparaciones:

2 1 3 4 5	1 2 3 4 5
-----------	-----------

Los elementos ordenados van apareciendo de uno en uno como las burbujas de un líquido en ebullición. Esta analogía da nombre al método.

Ejercicio 5.5.4 Sea el fichero matriz.txt que contiene n líneas de datos, teniendo en cada línea n datos numéricos enteros. A priori no se conoce el valor de n . Se desean almacenar todos los datos del fichero, en disposición idéntica a como se encuentran en éste, en una matriz A ($n \times n$). Además se quiere obtener la suma de todos los elementos de la matriz que no estén en la diagonal principal.

Se pide escribir un programa M que realice las tareas anteriores, imprimiendo en pantalla tanto la matriz A , como la suma que se pide, de la forma:

La suma es

Ejercicio 5.5.5 Sea un sistema de n partículas. Cada partícula viene determinada por su vector de posición \vec{r}_i y su masa m_i . Se define el centro de masas como el punto cuyo vector de posición \vec{r} se calcula mediante la siguiente expresión

$$\vec{r} = \frac{1}{m} \sum_{i=1}^n m_i \vec{r}_i$$

Sea el fichero sistema.txt que contiene la información del sistema de partículas. Tiene n líneas (n se tendrá que calcular) con 4 números en coma flotante en cada una. Los tres primeros números forman el vector de posición y el cuarto la masa de cada partícula.

Se pide diseñar un programa en lenguaje M que lea los datos del fichero y los guarde en una matriz A de $n \times 3$ (para los vectores de posición) y en un vector masas de n componentes (para las masas). Se debe obtener como salida a pantalla el vector de posición del centro de masas del sistema de partículas.

Solución Ejercicio 5.5.1

En el fichero `area.m` se escribirá:

```
function a=area(p,q,r)

%esta función calcula el área de un triángulo de vértices a,b,c

u=q-p;
v=r-p;

a=1/2*norm(cross(u,v));
end
```

La llamada a la función desde la ventana de comandos con los datos suministrados en el enunciado aparece a continuación:

```
>> a=[1 0 0];b=[0 1 0];c=[0 0 1];
>> areatri=area(a,b,c)
areatri =
0.8660
```

Solución Ejercicio 5.5.2

En el fichero `sfac.m` se escribirá:

```
function [suma, fact]=sfac(n)

suma=0;fact=1;

for i=1:n

    suma=suma+i;

    fact=fact*i;

end
```

Por otro lado, se llama a la función desde la ventana de comandos con el valor de entrada 8 y se almacena el resultado en las variables `sum` y `fac`.

```
>> [sum,fac]=sfac(8)
sum =
36
fac =
40320
```

Solución Ejercicio 5.5.3

A continuación, detallamos una función M que implementa el algoritmo de la burbuja:

```
function vo=burbuja(v)
n=length(v);
for k=1:n-1
    ord=1;
    for j=1:n-k
        if v(j)>v(j+1)
            aux=v(j);
            v(j)=v(j+1);
            v(j+1)=aux;
            ord=0;
        end
    end
    if (ord==1)break; end
end
vo=v
```

Veamos una solución alternativa:

```
function vo=burbuja(v)
n=length(v);
ord=0; % variable para detectar cuando está ordenado
k=0; % contador del numero de pasos
% algoritmo de la burbuja
while ~ord
    ord=1
    k=k+1
    for j=1:n-k
        if v(j)>v(j+1)
            aux=v(j);
            v(j)=v(j+1);
            v(j+1)=aux;
            ord=0;
        end
    end
end
vo=v
```

Solución Ejercicio 5.5.4

Una solución al ejercicio propuesto se indica a continuación:

```
fid=fopen('matriz.txt','r');
[A,cont]=fscanf(fid,'%d');
n=sqrt(cont);
frewind(fid);
[A,cont]=fscanf(fid,'%d',[n,n]);
A=A';
disp(A);
suma=0;
for i=1:n
    for j=1:n
        if (i==j) continue;end
        suma=suma+A(i,j);
    end
end
fprintf('La suma es %d\n',suma)
fclose(fid);
```

Solución Ejercicio 5.5.5

Una solución al problema es:

```
p=fopen('sistema.txt','r');
[A,cont]=fscanf(p,'%f');
n=cont/4;
frewind(p);
[A,cont]=fscanf(p,'%f',[4,n]);
fclose(p);
A=A';
masas=A(:,4)';
A(:,4)=[];
v=[0 0 0];
for i=1:n
    v=masas(i)*A(i,:)+v;
end
m=sum(masas);
v=v/m;
fprintf('El centro de masas es: %f %f %f\n',v)
```

6.1 Integración numérica.

Para obtener la solución de muchos problemas científicos es necesario resolver una integral. En numerosos casos no existe una primitiva conocida de la función a integrar, o bien la expresión de la integral es difícil de manejar o la función sólo se conoce en una serie de puntos. En todas estas situaciones es necesaria la utilización de métodos numéricos.

El problema que planteamos consiste en dada una función continua $f(x)$ en $[a,b]$ obtener de forma aproximada la siguiente integral sobre un intervalo acotado

$$I = \int_a^b f(x)dx .$$

Para ello consideramos una discretización del intervalo $[a,b]$ $a = x_0 < x_1 < \dots < x_m = b$. Tenemos $m+1$ puntos y m subintervalos I_k , de tal manera que la integral I es igual a la suma de las m integrales en los subintervalos I_k :

$$I = \int_a^b f(x)dx = \sum_{k=1}^m \int_{I_k} f(x)dx$$

Cada una de las m integrales pueden aproximarse sustituyendo la función $f(x)$ por un polinomio adecuado al intervalo I_k , con lo que obtenemos una aproximación a la integral (Q) y se genera un error (E).

$$I = \int_a^b f(x)dx = Q[f] + E[f].$$

Las diferentes fórmulas que se obtienen para $Q[f]$, según el polinomio por el que se sustituya la función $f(x)$, se denominan fórmulas de cuadratura. El término $E[f]$ es llamado error de truncación.

Veamos ahora las fórmulas de Newton-Cotes más sencillas.

- **Fórmula del punto medio o del rectángulo**

Con la fórmula del punto medio se aproxima la función en cada intervalo I_k por un polinomio de grado 0 que pase por el punto central del intervalo, es decir una recta horizontal $y=cte$.

Observemos en la figura 6.1, parte izquierda, el intervalo $[a,b]$ dividido en tres subintervalos, estando marcado el valor de la función en el punto central de cada uno de ellos con un punto, y en la parte derecha la representación gráfica de la integral obtenida al sustituir la función en cada subintervalo por el polinomio de grado 0. Se observa que es el área de un rectángulo, de ahí el nombre de este método.

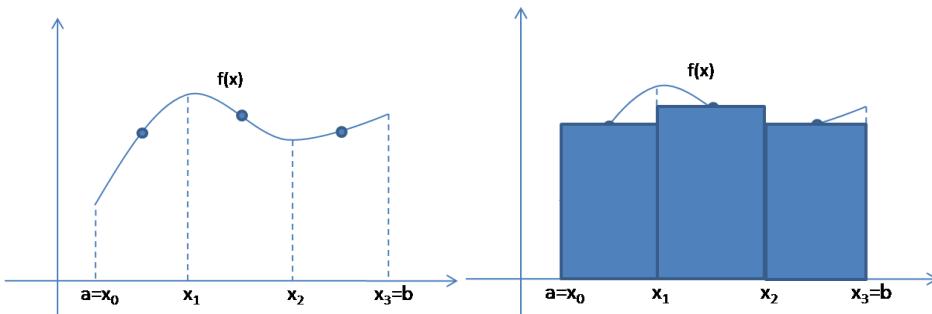


Figura 6.1 Fórmula del punto medio

Suponemos que $x_k = x_0 + hk$ son nodos equiespaciados con paso $h = \frac{b-a}{m}$ y el punto medio del subintervalo $[x_{k-1}, x_k]$ es $\bar{x}_k = \frac{x_{k-1} + x_k}{2}$. La integral queda:

$$I = \int_a^b f(x)dx \approx h \sum_{k=1}^m f(\bar{x}_k)$$

La siguiente función M implementa el método del punto medio teniendo como datos de entrada el texto de la función a integrar `fun`, los extremos de integración `a`, `b` y el número de subintervalos `m`.

```
function [int]=rectangulo(fun,a,b,m)

f=inline(fun);

h=(b-a)/m;

x=a:h:b; %m+1 puntos, m intervalos

int=0;

for i=1:m %para cada intervalo

    %calculamos punto medio

    xm=(x(i)+x(i+1))/2; %ver aclaración después de este código

    int=int+f(xm);

end

int=int*h;
```

Como aclaración al código debemos resaltar que la notación matemática y computacional siempre difieren en una unidad, por ejemplo el primer subintervalo $m=1$, tiene como extremos los puntos denotados matemáticamente como x_0 y x_1 , sin embargo, computacionalmente estos puntos son los dos primeros elementos del vector `x`, es decir, $x(1)$ y $x(2)$.

Notación matemática	Notación computacional
x_0	$x(1)$
x_1	$x(2)$
x_{k-1}	$x(k)$

La llamada a la función `rectangulo` que obtiene el resultado de

$$I = \int_0^{10} x^2 dx$$

tomando 15 subintervalos es la siguiente:

```
>> resultado=rectangulo('x^2',0,10,15)
```

```
resultado =
```

```
332.9630
```

Ya que la solución exacta de la integral es $1000/3$, el error obtenido es

-3.703703703703809e-001.

■ Fórmula del trapecio

Con la fórmula del trapecio se aproxima la función en cada intervalo I_k por el polinomio de grado 1 que pasa por los extremos del intervalo, es decir una recta que pasa por los extremos.

Observemos en la figura 6.2, parte izquierda, el intervalo $[a,b]$ dividido en tres subintervalos con el valor de la función en los extremos de cada uno de ellos marcado con un punto, y en la parte derecha la representación gráfica de la integral obtenida al sustituir la función por la recta que pasa por los extremos. Observemos que se obtienen trapecios, de ahí el nombre de este método.

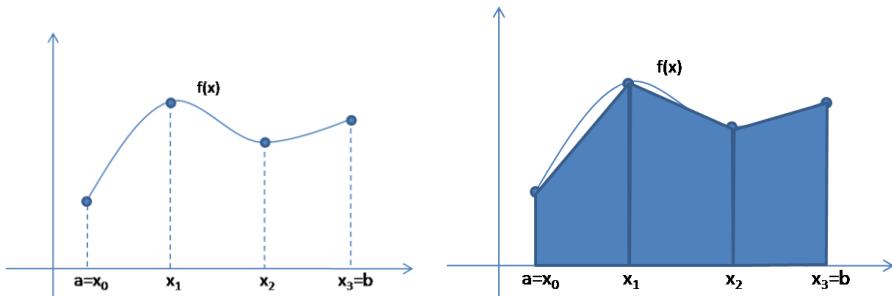


Figura 6.2 Fórmula del trapecio

Suponemos que $x_k = x_0 + hk$ son nodos equiespaciados con paso $h = \frac{b-a}{m}$, la expresión matemática de la integral queda:

$$I = \int_a^b f(x)dx \approx \frac{h}{2} \sum_{k=1}^m (f(x_{k-1}) + f(x_k)) = \frac{h}{2}(f(a) + f(b)) + h \sum_{k=1}^{m-1} f(x_k)$$

La siguiente función M implementa el método del trapecio teniendo como datos de entrada el texto de la función a integrar `fun`, los extremos de integración `a`, `b` y el número de subintervalos `m`.

```
function [int]=trapezio(fun,a,b,m)
f=inline(fun);
h=(b-a)/m;
x=a:h:b; %m+1 puntos, m intervalos
int=0;
for i=1:m %para cada intervalo
    %extremos x(i), x(i+1)
    int=int+f(x(i))+f(x(i+1));
end
int=h/2*int;
```

La llamada a la función `trapezio` que obtiene el resultado de

$$I = \int_0^{10} x^2 dx$$

es la siguiente:

```
>> resultado=trapezio('x^2',0,10,15)
resultado =
334.0741
```

Ya que la solución exacta de la integral es $1000/3$, el error obtenido usando 15 subintervalos es $-7.407407407407050e-001$.

6.2 Sistemas de ecuaciones lineales. Métodos iterativos

Existen dos formas principales de acometer la resolución numérica de sistemas de ecuaciones lineales: métodos directos e iterativos.

Los métodos directos obtienen una solución exacta del problema en un número finito de pasos. La exactitud de la solución no ocurre realmente debido a los inevitables errores de redondeo. Algun ejemplo de método directo es el método de Gauss, el de factorización LU y Cholesky.

Sin embargo, en el caso de tener sistemas de ecuaciones muy grandes no es viable la utilización de métodos directos por la gran cantidad de operaciones a realizar (del orden de N^3) y con ello el tiempo de ejecución que se necesitaría y la pérdida de precisión por propagación de errores de redondeo. En ingeniería es habitual encontrarse con este tipo de sistemas que además suelen tener la característica de poseer una gran cantidad de coeficientes igual a cero (matrices poco densas que se tratan utilizando técnicas *sparse* de almacenamiento). Por ejemplo, el método de Gauss presentaría en estos casos varias dificultades:

- Puede ser muy lento, su complejidad temporal es $O(N^3)$.
- Modifica la matriz del sistema, lo cual puede hacer que desaparezca la ventaja inicial de tener muchos coeficientes iguales a cero.
- Necesita almacenar matrices completas.
- Para obtener la solución es necesario realizar todo el tratamiento. En las etapas intermedias no se dispone de ninguna aproximación de la solución.

Los métodos iterativos resuelven el sistema $\vec{Ax} = \vec{b}$ de forma aproximada. Para ello comienzan descomponiendo la matriz $A = P - Q$, con lo que el sistema queda,

$$(P - Q)\vec{x} = \vec{b} \rightarrow P\vec{x} = Q\vec{x} + \vec{b}.$$

A continuación se elige una primera aproximación $x^{(0)}$ a la solución (generalmente de forma arbitraria, en ausencia de otra información) y se calcula una nueva aproximación según la fórmula,

$$Px^{(1)} = Qx^{(0)} + b,$$

para seguir con este proceso hasta que el método converja, es decir se debe aplicar el esquema iterativo

$$Px^{(n+1)} = Qx^{(n)} + b, \quad n = 0, 1, \dots$$

hasta que se cumpla un criterio de parada que puede ser, por ejemplo, que la distancia entre dos aproximaciones sucesivas $\|x^{(n+1)} - x^{(n)}\|$ sea menor que la tolerancia admisible en el método.

Estos métodos no siempre convergen en un número razonable de iteraciones y en ocasiones ni siquiera convergen pero presentan las siguientes ventajas:

- No modifican la matriz del sistema y en general sólo precisan multiplicar por la matriz del sistema o por partes de ella.
- Son bastante estables frente a los errores de redondeo.
- Permiten disponer en cada etapa de una aproximación de la solución.

Trataremos a continuación uno de estos métodos, el método de Jacobi.

Dada la matriz de coeficientes

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2N} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \dots & a_{NN} \end{pmatrix},$$

el método de Jacobi se basa en la siguiente descomposición de A:

$$A = D + L + U$$

siendo:

$$D = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{NN} \end{pmatrix}, \quad L = \begin{pmatrix} 0 & 0 & \dots & 0 \\ a_{21} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & 0 \end{pmatrix}, \quad U = \begin{pmatrix} 0 & a_{12} & \dots & a_{1N} \\ 0 & 0 & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}.$$

Como habíamos indicado que estos métodos iterativos descomponen la matriz $A = P - Q$, en el método de Jacobi se toma $P = D$ y $Q = -(L + U)$, es decir el esquema iterativo queda

$$Dx^{(n+1)} = -(L + U)x^{(n)} + b, \quad n = 0, 1, \dots$$

Si la matriz A es de tamaño $N \times N$, las sucesivas soluciones $x^{(n+1)}$ son vectores de tamaño $N \times 1$. En cada iteración, por tanto, se debe calcular las N componentes del vector $x^{(n+1)}$. Observemos que dadas las características de las matrices, las operaciones a realizar en cada iteración son las siguientes,

$$\begin{aligned}
a_{11}x_1^{(n+1)} &= -\text{fila1}_{(L+U)} \cdot x^{(n)} + b_1, \\
a_{22}x_2^{(n+1)} &= -\text{fila2}_{(L+U)} \cdot x^{(n)} + b_2, \\
a_{33}x_3^{(n+1)} &= -\text{fila3}_{(L+U)} \cdot x^{(n)} + b_3, \\
&\dots
\end{aligned}$$

donde la notación $x_j^{(n+1)}$ indica componente j-ésima del vector solución de la iteración $(n+1)$. En lo anterior hemos supuesto $a_{jj} \neq 0$, $j = 1, \dots, N$.

Observemos que las únicas operaciones a realizar son productos escalares entre vectores además de operaciones escalares y que no se necesita almacenar ninguna matriz además de A.

En la lección 6.4 aparece un ejercicio donde se utiliza el método de Jacobi.

6.3 Ecuaciones no lineales

- **Método iterativo para resolver ecuaciones no lineales: método de Newton-Raphson**

El método de Newton-Raphson es un método iterativo que obtiene una solución para la ecuación

$$f(x) = 0, \quad (1)$$

es decir, obtiene uno de los llamados '*ceros*' de la función.

Ya que cualquier ecuación no lineal $g(x) = h(x)$, se puede expresar de la forma dada en (1), $g(x) - h(x) = 0$, este método puede obtener una solución de una ecuación no lineal.

Una manera sencilla de exponer el funcionamiento del método de Newton-Raphson es mediante su interpretación gráfica reflejada en la figura 6.3.

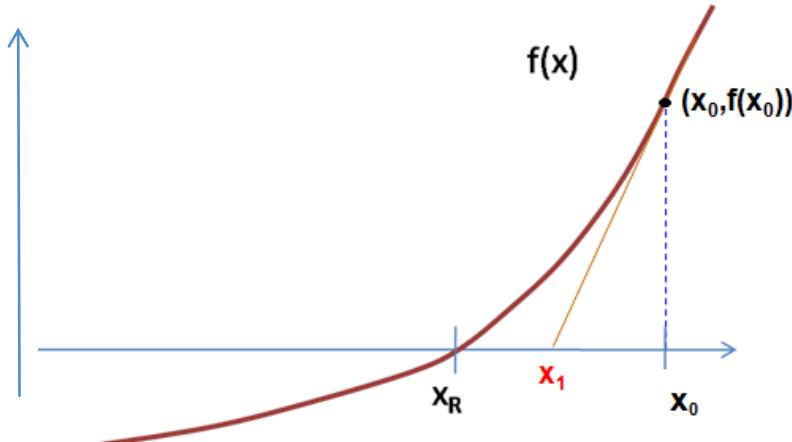


Figura 6.3 Primera iteración del método de Newton-Raphson

Observemos como la tangente a la función $f(x)$ en el punto $(x_0, f(x_0))$ corta al eje de abscisas en x_1 . Se cumple que

$$f'(x_0) = \frac{f(x_0)}{x_0 - x_1} \Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Repitiendo este proceso, a partir del punto $(x_1, f(x_1))$ llegaríamos a una nueva aproximación a la solución x_2 , y así sucesivamente, con lo que el esquema iterativo del método de Newton-Raphson queda:

$$x_{k+1} = x_k - (f(x_k) / f'(x_k)), \quad k = 0, 1, 2, \dots$$

Una estimación del error viene determinada por la distancia entre dos aproximaciones sucesivas, por lo que un método para detener el proceso es el cumplimiento de la siguiente desigualdad

$$|x_{k+1} - x_k| < \text{tolerancia} .$$

Además se puede limitar el número de iteraciones a un valor máximo.

El siguiente programa M implementa el método de Newton-Raphson leyendo por teclado la forma analítica de la función y derivada (como cadenas de texto), el número máximo de iteraciones, la tolerancia y el punto de partida x_0 . Como método de parada del proceso se utiliza la distancia entre dos aproximaciones sucesivas. Obtiene como resultado, en caso de convergencia, la solución y el número de iteraciones utilizadas, en otro caso informa de la divergencia o de la no convergencia para la tolerancia fijada.

```
fun=input('Introduce la forma analítica de la función: ','s');
f=inline(fun);
der=input('Introduce la forma analítica de la derivada: ','s');
df=inline(der);
tol=input('Introduce la tolerancia: ');
maxit=input('Número máximo de iteraciones');
x0=input('valor inicial');
niter=0;
while niter<maxit
    x1=x0-f(x0)/df(x0)
    niter=niter+1;
    if abs(x1-x0)<tol
        fprintf('Solución: %f\n',x1);
        fprintf('Iteraciones: %d\n',niter);
        return;
    end
    x0=x1;
end
disp('El método no converge');
```

Aplicaremos este programa a la resolución de la ecuación

$$x = \cos(\sin(x)) .$$

Para ello debemos expresarla de la forma $f(x)=0$, es decir,

$$x - \cos(\sin(x)) = 0 .$$

Utilizando como punto de partida $x_0=0$, tolerancia $\text{tol}=10^{-6}$ y número máximo de iteraciones $\text{maxit}=1000$, el resultado obtenido es :

Solución: 0.768169

- **Métodos de aproximación por intervalos para resolver $f(x)=0$: Método de la bisección de Bolzano**

El Teorema de Bolzano garantiza que dado un intervalo $[a,b]$ en el que la función $f(x)$ presenta signo distinto en cada uno de los extremos ($f(a)f(b) < 0$) existe al menos un punto $x^* \in (a,b)$ tal que $f(x^*) = 0$.

El desarrollo del método es el siguiente:

Paso 1. Búsqueda de dos valores a y b tales que $f(a)f(b) < 0$. Supongamos que existe un único cero en el intervalo (a,b) .

Paso 2. Se selecciona el punto medio $c=(a+b)/2$ y se analizan las tres posibilidades que podrían ocurrir:

1. Si $f(a)$ y $f(c)$ tienen distinto signo, hay un cero en $[a,c]$. Se debe llamar de nuevo a y b a los extremos de este intervalo.
2. Si $f(c)$ y $f(b)$ tienen distinto signo, hay un cero en $[c,b]$. Se debe llamar de nuevo a y b a los extremos de este intervalo.
3. Si $f(c)=0$, entonces c es la solución buscada. Fin del algoritmo.

Paso 3. En los casos 1 y 2 del paso 2 se ha encontrado un intervalo, de longitud mitad del inicial, que contiene una solución. Se debe repetir el proceso una y otra vez hasta que el intervalo sea lo suficientemente pequeño (menor que la tolerancia prefijada). De hecho, al cabo de n bisecciones la anchura del intervalo que contiene la solución es $(b-a)/2^n$. Se puede tomar como mecanismo de parada del método el cumplimiento de la siguiente desigualdad:

$$|b-a| < \text{tolerancia}.$$

Se pedirá la implementación del método con un programa M en el ejercicio de evaluación de este módulo.

6.4 Practicando con programas (VI). Ejercicios de autoevaluación del módulo.

Ejercicio 6.4.1 Otra fórmula de cuadratura para aproximar la integral de una función es la fórmula de Simpson que approxima la función en cada intervalo I_k por un polinomio de grado 2 que pase por los extremos y el centro del intervalo. Como en casos anteriores tendremos m subintervalos de tamaño h y $m+1$ puntos que son los extremos de los subintervalos. Se utilizará además el punto central de cada subintervalo.

Suponemos que $x_k = x_0 + hk$ son nodos equiespaciados con paso $h = \frac{b-a}{m}$ y el punto medio del subintervalo $[x_{k-1}, x_k]$ es $\bar{x}_k = \frac{x_{k-1} + x_k}{2}$ y $f_k = f(x_k)$, la expresión matemática de la integral queda:

$$I = \int_a^b f(x)dx \approx \frac{h}{6} \sum_{k=1}^m (f(x_{k-1}) + 4f(\bar{x}_k) + f(x_k))$$

Escribir una función llamada simpson que implemente el método de Simpson teniendo como argumentos de entrada la función a integrar, los extremos del intervalo de integración y el número de subintervalos en los que se divide éste. La función retornará la integral aproximada.

Probar la función calculando la siguiente integral con 5 subintervalos:

$$\int_0^1 \sin(\ln(x))dx$$

Sabiendo que tiene como resultado exacto

$$\frac{1}{2} x(\sin(\ln(x)) - \cos(\ln(x)))|_1^2$$

Calcular el error obtenido.

Ejercicio 6.4.2

Se considera el sistema lineal $\vec{Ax} = \vec{b}$ donde:

$$A = \begin{pmatrix} 4 & -1 & 1 \\ 4 & -8 & 1 \\ -2 & 1 & 5 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 7 \\ -21 \\ 15 \end{pmatrix}$$

Se trata de calcular usando un programa M las soluciones sucesivas que se obtienen utilizando el método de Jacobi explicado en este módulo. ¿Cuál es la solución del sistema utilizando tolerancia 10^{-6} ?

Ejercicio 6.4.3 Sea el sistema de ecuaciones lineales $\vec{Ax} = \vec{b}$. El método de Gauss-Seidel es un método iterativo en el que la matriz de coeficientes A se descompone de la misma manera que en el método de Jacobi:

$$A = D + L + U$$

En estos métodos iterativos se cumple que la matriz $A = P - Q$, en el método de Gauss-Seidel se toma $P = D + L$ y $Q = -U$, es decir el esquema iterativo queda

$$(D + L)x^{(n+1)} = -Ux^{(n)} + b, \quad n = 0, 1, \dots$$

Si la matriz A es de tamaño $N \times N$, las sucesivas soluciones $x^{(n+1)}$ son vectores de tamaño $N \times 1$. En cada iteración, por tanto, se debe calcular las N componentes del vector $x^{(n+1)}$. Observemos que dadas las características de las matrices, las operaciones a realizar en cada iteración son las siguientes,

$$\begin{aligned} a_{11}x_1^{(n+1)} &= -\text{fila}_1_U \cdot x^{(n)} + b_1, \\ a_{21}x_1^{(n+1)} + a_{22}x_2^{(n+1)} &= -\text{fila}_2_U \cdot x^{(n)} + b_2, \\ a_{31}x_1^{(n+1)} + a_{32}x_2^{(n+1)} + a_{33}x_3^{(n+1)} &= -\text{fila}_3_U \cdot x^{(n)} + b_3, \\ &\dots \end{aligned}$$

donde la notación $x_j^{(n+1)}$ indica componente j-ésima del vector solución de la iteración $(n+1)$.

Observemos que las únicas operaciones a realizar son productos escalares entre vectores además de operaciones escalares y que no se necesita almacenar ninguna matriz además de A.

Si lo escribimos en componentes, y $a_{jj} \neq 0$, $j = 1, \dots, N$ resultaría:

$$x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(n+1)} - \sum_{j=i+1}^N a_{ij}x_j^{(n)} \right) \quad i = 1, \dots, N$$

SE PIDE Escribir una función M que implemente el método iterativo de Gauss-Seidel. Los datos de entrada serán la matriz de coeficientes del sistema, el vector columna de términos independientes, el número máximo de iteraciones permitido y la tolerancia del método. Los datos de retorno serán la solución del sistema y el número de iteraciones en las que se ha llegado a ésta. Aplicarla a resolver el sistema lineal planteado en el ejercicio anterior con la misma tolerancia.

Ejercicio 6.4.4 El método de Newton hace uso de la función f y de su primera derivada f' lo cual lleva a tener que calcular la derivada de la función. Esto puede evitarse usando fórmulas de derivación numérica para aproximar la derivada. Existen varias posibilidades, una de las fórmulas es conocida como diferencia finita regresiva y viene dada con la expresión

$$f'(x_n) \approx \frac{f(x_n) - f(x_n - h)}{h}$$

que aproxima la derivada usando la pendiente de una recta que pasa por $(x_n, f(x_n))$ y un punto de la función situado a su izquierda h unidades.

Empleando otra notación tendríamos

$$f'(x_k) \approx (f(x_k) - f(x_{k-1})) / (x_k - x_{k-1})$$

que sustituyendo en la fórmula iterativa del método de Newton resulta el siguiente método iterativo:

$$x_{k+1} = x_k - f(x_k) \frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}, k = 1, 2, \dots$$

Este método se conoce como **Método de la Secante**. Observemos que para el cálculo de la aproximación x_{k+1} se utilizan las dos aproximaciones anteriores x_k y x_{k-1} y por tanto para comenzar el proceso iterativo son necesarios los dos primeros puntos iniciales x_0, x_1 . Se supone que $f(x_k) - f(x_{k-1}) \neq 0$ para $k \geq 1$. El método de la secante es un ejemplo de método iterativo *multietapa*.

SE PIDE escribir una función M de nombre secante que implemente el método de la Secante. Utilizar la distancia entre dos iteraciones sucesivas como método de finalización del proceso. Aplicarla a solucionar las ecuaciones

- a) $(x - 2)^2 = 0$ con $x_0 = 2.3, x_1 = 2.1$.
- b) $(x - 2)^5 = 0$ con $x_0 = 2.3, x_1 = 2.1$.
- c) $\cos(x) = 0$ con $x_0 = 2.9, x_1 = 3, x_0 = 3.9, x_1 = 4$ y $x_0 = 1.2, x_1 = 1.3$.

Solución Ejercicio 6.4.1

La función simpson es la siguiente:

```
function [int]=simpson(fun,a,b,m)
f=inline(fun);
h=(b-a)/m;
x=a:h:b; %m+1 puntos, m intervalos
int=0;
for i=1:m %para cada intervalo
    %extremos x(i), x(i+1)
    xm=(x(i)+x(i+1))/2;
    int=int+f(x(i))+4*f(xm)+f(x(i+1));
end
int=h/6*int;
```

La llamada a la función y el resultado obtenido se escribe a continuación:

```
x2=2;x1=1;
exacta=1/2*x2*(sin(log(x2))-cos(log(x2)))-(1/2*x1*(sin(log(x1))-...
cos(log(x1))));
I1=simpson('sin(log(x))',1,2,5);
fprintf('Aproximación con Simpson: %f.\nError: %e\n',I1, ...
abs(exacta-I1));
```

Aproximación con Simpson: 0.369722.

Error: 3.755307e-007

Solución Ejercicio 6.4.2

Un programa válido es el siguiente:

```
A=input('Introduce matriz de coeficientes');

b=input('Introduce vector columna de términos independientes');

niter=input('Introduce número máximo de iteraciones');

tol=input('Introduce la tolerancia del método');

N=length(b);

xx=zeros(N,1); % primera aproximación

x=zeros(N,1); % almacena las aproximaciones sucesivas

for j=1:niter

    for i=1:N

        x(i)=(-A(i,1:i-1)*xx(1:i-1)-A(i,i+1:N)*xx(i+1:N)+b(i))/A(i,i);

    end

    if norm(x-xx)<1.e-6

        fprintf('Número de iteraciones: %d\n',j);

        disp('Solución');

        disp(x);

        return;

    end

    xx=x;

end

disp('Solución no encontrada');
```

Las soluciones obtenidas son las siguientes:

a)

$$\mathbf{x} = \begin{pmatrix} 1.5286 \\ 2.2500 \\ 1.2000 \\ -0.6000 \end{pmatrix}$$

b)

$$\mathbf{x} = \begin{pmatrix} -3.6417 \\ 6.5833 \\ 3.8000 \\ 0.6000 \\ 1.6667 \end{pmatrix}$$

Solución Ejercicio 6.4.3

La función Gauss-Seidel en MATLAB / Octave es:

```
function [x,niter]=gauss_seidel(A,b,maxiter,tol)
N=length(b);
xx=zeros(N,1);
x=zeros(N,1);
for j=1:maxiter
    for i=1:N
        x(i)=(-A(i,i+1:N)*xx(i+1:N)+b(i)-...
            A(i,1:i-1)*x(1:i-1))/A(i,i);
    end
    if norm(x-xx)<tol
        niter=j;
        return;
    end
    xx=x;
end
disp('Solución no encontrada');
```

Aplicándola al problema concreto resulta:

```
>> [sol,iteraciones]=gauss_seidel(A,b,100,1.e-6)
```

```
sol =
2.0000
4.0000
3.0000
iteraciones =
```

Solución Ejercicio 6.4.4

Indicamos a continuación el código de la función secante:

```
function [x1,k]=secante(fun,x0,x1,epsilon,maxit)

% Entrada:
% fun- es la función objetivo .
% x0 y x1- son las aproximaciones iniciales al cero de f
% epsilon- es la tolerancia para el valor de la función
% maxit- es el numero máximo de iteraciones

% Salida:
% x1- es la aproximación por el método de la Secante a la solución.
% k- es el numero de iteraciones

f=inline(fun);
for k=1:maxit

    x2=x1-f(x1)*(x1-x0)/(f(x1)-f(x0));
    err=abs(x2-x1);

    x0=x1;
    x1=x2;

    if(err<epsilon), return, end
end
x1='No converge';
k=maxit;
```

Apartado a) Llamada a la función:

```
>> [sol,iteraciones]=secante('(x-2)^2',2.3,2.1,1e-7,100)
```

```
sol =
2.0002
```

```
iteraciones =
```

Apartado b) Llamada a la función:

```
>> [sol,iteraciones]=secante('(x-2)^5',2.3,2.1,1e-7,100)
```

```
sol =  
2.0374
```

```
iteraciones =  
7
```

Apartado c) Llamadas a la función:

```
>> [sol,iteraciones]=secante('cos(x)',2.9,3,1e-7,100)
```

```
sol =  
4.7124
```

```
iteraciones =  
9
```

```
>> [sol,iteraciones]=secante('cos(x)',3.9,4,1e-7,100)
```

```
sol =  
4.7124  
iteraciones =  
4
```

```
>> [sol,iteraciones]=secante('cos(x)',1.2,1.3,1e-7,100)
```

```
sol =  
1.5708  
iteraciones =  
3
```