



Apuntes de MATLAB orientados a métodos numéricos elementales

Rosa Echevarría
Dpto. de Ecuaciones Diferenciales y Análisis Numérico
Universidad de Sevilla

6 de febrero de 2020

Índice



1. Introducción a los ordenadores	13
1.1. Componentes básicos de un ordenador	13
1.1.1. Unidad central de proceso	14
1.1.2. Memoria	14
1.1.3. Memoria central	15
1.1.4. Memoria secundaria	15
1.1.5. Dispositivos de entrada/salida	16
1.2. Software	16
1.3. Lenguajes	17
1.4. Tipos de datos	18
1.4.1. Tipos de datos elementales	18
1.4.2. Tipos de datos compuestos	19
1.5. Representación de datos numéricos en el ordenador	20
1.5.1. Bits, bytes y palabras	20
1.5.2. Sistemas de numeración	21
1.5.3. Almacenamiento de números enteros sin signo	28
1.5.4. Almacenamiento de números enteros con signo	29
1.5.5. Almacenamiento de números reales	38
1.5.6. Errores	41
2. Introducción a MATLAB. Operaciones elementales	43
2.1. Introducción	43
2.1.1. Objetos y sintaxis básicos	44
2.1.2. Documentación y ayuda <i>on-line</i>	45
2.1.3. <i>Scripts</i> y funciones. El editor integrado	45
2.1.4. <i>Workspace</i> y ámbito de las variables	48
2.2. Números	49
2.3. Operaciones aritméticas	49
2.3.1. Reglas de prioridad	50

2.4.	Forma en que se muestran los resultados	51
2.5.	Variables	51
2.5.1.	Instrucciones de asignación	52
2.5.2.	Variables pre-definidas	53
2.6.	Funciones matemáticas elementales	53
2.7.	Operaciones de comparación	54
2.8.	Operadores lógicos	56
2.9.	Orden general de evaluación de expresiones	57
2.10.	Matrices	58
2.10.1.	Construcción de matrices	58
2.10.2.	Acceso a los elementos de vectores y matrices	62
2.10.3.	Operaciones con vectores y matrices	64
2.10.4.	Operaciones de comparación y lógicas con matrices	66
2.11.	Ejercicios	70
3.	Gráficos	73
3.1.	Generalidades sobre la representación gráfica de funciones	73
3.1.1.	Representación gráfica de funciones de una variable real	74
3.1.2.	Curvas planas definidas mediante ecuaciones paramétricas	75
3.1.3.	Curvas planas en coordenadas polares	75
3.1.4.	Curvas en tres dimensiones	78
3.1.5.	Gráficas de funciones de dos variables: superficies	79
3.1.6.	Superficies definidas mediante ecuaciones paramétricas	80
3.1.7.	Representación mediante curvas de nivel de una función de dos variables	81
3.2.	Funciones gráficas de MATLAB fáciles de usar	83
3.2.1.	Gráficas de funciones de una variable real	83
3.2.2.	Curvas planas definidas implícitamente	85
3.2.3.	Curvas planas definidas por ecuaciones paramétricas	86
3.2.4.	Curvas planas en coordenadas polares	87
3.2.5.	Sobre los nombres de las variables independientes	88
3.2.6.	Curvas en tres dimensiones	89
3.2.7.	Superficies definidas mediante ecuaciones explícitas	90
3.2.8.	Superficies definidas mediante ecuaciones paramétricas	91
3.2.9.	Representación mediante curvas de nivel de una función de dos variables	92
3.2.10.	Observación importante	93
3.3.	Funciones básicas para el dibujo de curvas	94
3.3.1.	La orden plot	94
3.3.2.	Color y tipo de línea	97
3.3.3.	Personalizar las propiedades de las líneas	98

3.3.4. Personalizar los ejes: orden axis	100
3.3.5. Anotaciones: título, etiquetas y leyendas	101
3.3.6. Dibujar cosas encima de otras	103
3.3.7. Añadir texto a la gráfica	104
3.3.8. Gestión de la ventana gráfica	105
3.3.9. Otros tipos de gráficos de datos planos	105
3.3.10. Varios ejes en la misma gráfica	107
3.4. Ejercicios	109
4. Programación con MATLAB	111
4.1. Operaciones básicas de lectura y escritura	111
4.1.1. Instrucción básica de lectura: input	111
4.1.2. Impresión en pantalla fácil: disp	112
4.1.3. Impresión en pantalla con formato: fprintf	113
4.2. Estructuras condicionales: if	115
4.2.1. Estructura condicional simple (if-end)	115
4.2.2. Estructura condicional doble (if-else-end)	116
4.2.3. Estructuras condicionales anidadas	119
4.2.4. Estructura condicional múltiple (if-elseif-else-end)	120
4.3. Estructuras de repetición o bucles	122
4.3.1. Estructuras de repetición condicionada (while-end)	122
4.3.2. Estructuras de repetición indexada (for-end)	125
4.4. Rupturas incondicionales de secuencia	128
4.5. Gestión de errores: warning y error	129
4.6. Algunos consejos	131
4.7. Ejercicios	133
5. Resolución de sistemas lineales	141
5.1. Los operadores de división matricial	141
5.2. Determinante. ¿Cómo decidir si una matriz es singular?	143
5.3. La factorización LU	146
5.4. La factorización de Cholesky	148
5.5. Resolución de sistemas con características específicas	149
5.6. Matrices huecas (<i>sparse</i>)	150
6. Operaciones de lectura y escritura con ficheros en MATLAB	155
6.1. Ficheros de texto y ficheros binarios	155
6.2. Órdenes save y load	156
6.2.1. Grabar en fichero el espacio de trabajo: orden save	156
6.3. Lectura y escritura en ficheros avanzada	160

6.3.1. Apertura y cierre de ficheros	161
6.3.2. Leer datos de un fichero con <code>fscanf</code>	162
6.3.3. Escribir datos en un fichero con <code>fprintf</code>	165
7. Interpolación y ajuste de datos en MATLAB	169
7.1. Introducción	169
7.2. Interpolación polinómica global	169
7.3. Interpolación lineal a trozos	175
7.4. Interpolación por funciones <i>spline</i>	177
7.5. Ajuste de datos	180
7.5.1. Ajuste por polinomios	180
7.5.2. Otras curvas de ajuste	183
8. Resolución de ecuaciones no lineales	187
8.1. Introducción	187
8.2. Resolución de ecuaciones polinómicas	188
8.3. La función <code>fzero</code>	190
8.4. Gráficas para localizar las raíces y elegir el punto inicial	192
8.5. Ceros de funciones definidas por un conjunto discreto de valores	194
8.6. Algoritmos numéricos para la resolución de ecuaciones	196
8.6.1. El algoritmo de bisección o dicotomía	196
8.6.2. El método de aproximaciones sucesivas	199
8.6.3. El método de Newton	200
9. Integración numérica	203
9.1. La función <code>integral</code>	203
9.2. Aplicaciones de la integral definida	206
9.2.1. Cálculo de áreas	206
9.2.2. Cálculo de longitudes de arcos de curva	211
9.3. Cálculo de volúmenes y superficies de revolución	213
9.4. Cálculo de integrales de funciones definidas por un conjunto discreto de valores	215
9.4.1. Integrar una poligonal: La función <code>trapz</code>	215
9.4.2. Integrar otros interpolantes	216
9.5. Funciones definidas a trozos	221
10. Resolución de ecuaciones y sistemas diferenciales ordinarios	225
10.1. Ecuaciones diferenciales ordinarias de primer orden	225
10.2. Resolución de problemas de Cauchy para edo de primer orden	226
10.3. Resolución de problemas de Cauchy para sdo de primer orden	230
10.4. Resolución de problemas de Cauchy para edo de orden superior	234

11. Resolución de problemas de contorno para sistemas diferenciales ordinarios	237
11.1. La función bvp4c	237
12. Resolución de problemas de minimización	245
12.1. La función fmincon	245

Índice de tablas



1.1.	Unidades de almacenamiento de información	15
1.2.	Rango de valores enteros sin signo representables	28
1.3.	Rango de valores enteros representables en signo-magnitud	29
1.4.	Las dos representaciones del cero en el sistema signo-magnitud con 16 bits.	30
1.5.	Las dos representaciones del cero en el sistema de complemento a 1 con 16 bits.	31
1.6.	Rango de valores enteros representados en el sistema con complemento a 2.	34
1.7.	Decodificación de patrones de 4 bits con los distintos sistemas	37
1.8.	Algunos casos especiales en la norma IEEE 754	39
2.1.	Operadores aritméticos elementales	49
2.2.	Algunas funciones matemáticas elementales	54
2.3.	Operadores de comparación	55
2.4.	Operadores lógicos	56
2.5.	Resultados del operador \sim	56
2.6.	Resultados de los operadores $\&$ y $ $	57
2.7.	Vectores regularmente espaciados	59
2.8.	Algunas funciones para generación de matrices habituales	61
2.9.	Operaciones aritméticas con matrices	64
2.10.	Operaciones con matrices elemento a elemento	65
2.11.	Funciones para cálculos matriciales	66
2.12.	Funciones lógicas para matrices	68
3.1.	Símbolos para definir el color y el tipo de línea a dibujar en la orden <code>plot</code>	97
3.2.	Algunas propiedades de las líneas, controlables con la orden <code>plot</code>	99
3.3.	Algunas propiedades de los títulos y las etiquetas de los ejes	102

Índice de figuras



1.1. Componentes básicos de un ordenador	14
1.2. Memoria central de un ordenador	16
1.3. Paso a binario de un número entero en sistema decimal.	22
2.1. La ventana de MATLAB	44
3.1. Linea poligonal determinada por un conjunto de puntos.	73
3.2. Representación de $y = \operatorname{sen}(x)$ en $[0, \pi]$ con 8 y con 100 puntos.	73
3.3. Curva definida por la relación $x = y \cos(4y)$, $y \in [0, 2\pi]$	75
3.4. Representación de una curva dada por ecuaciones paramétricas	76
3.5. Representación en paramétricas	76
3.6. Representación en paramétricas	76
3.7. Sistema de coordenadas polares.	77
3.8. Coordenadas cartesianas y polares.	77
3.9. Curva en coordenadas polares	77
3.10. Curva en coordenadas polares	77
3.11. Curva 3D en paramétricas	78
3.12. Curva 3D en paramétricas	78
3.13. Mallado en triángulos de un dominio de frontera curva	79
3.14. Mallado en rectángulos de un dominio rectangular	79
3.15. Un triángulo en 3D	80
3.16. Red triangular deformada.	80
3.17. Red rectangular deformada.	80
3.18. Red rectangular deformada coloreada	80
3.19. Cara triangular rellena de color plano.	80
3.20. Mallado deformado con caras de color constante.	80
3.21. Mallado deformado, color dependiente de la altura	81
3.22. Mallado deformado, color interpolado	81
3.23. Superficie cilíndrica	81
3.24. Curvas de nivel de una función de dos variables	82

4.1. Condicional simple	115
4.2. Condicional doble	117
4.3. Condicional doble	120
4.4. Repetición condicional	122
4.5. Ruptura de bucle break	129
4.6. Ruptura de bucle continue	129
6.1. Tabla de los caracteres imprimibles del código ASCII	167
7.1. Interpolante lineal a trozos.	176
8.1. La gráfica «toca» al eje de abscisas en el punto α , lo que significa que α es un cero de la función f , aunque no en ambos casos la función cambia de signo en α	187
9.1. Región delimitada por la gráfica de la función $y = f(x)$, el eje de abscisas y las rectas $x = a$ y $x = b$	206

1

Introducción a los ordenadores



Las palabras españolas **informática** y **ordenador** provienen de las francesas **informatique** y **ordinateur**. La palabra francesa **informatique** se construyó como una contracción de las palabras **INFORmation** y **autoMATIQUE**.

La palabra **computador** (o **computadora**, mayoritariamente utilizada en Sudamérica), proviene del término inglés **computer**. La ciencia que se ocupa de los ordenadores se denomina, en inglés, **Computer Science**, aunque también existe la palabra **Informatics**.

1.1 Componentes básicos de un ordenador

En un ordenador se distinguen principalmente dos aspectos:

- **HARDWARE**, el soporte físico, no modificable (placas, circuitos integrados, chips, módulos, cables, etc.), es decir la “maquinaria”.
- **SOFTWARE**, el conjunto de programas que se ejecutan en el ordenador, *grossso modo* divididos en:
 - Software del sistema o sistema operativo, el conjunto de programas necesarios para que el ordenador tenga capacidad de trabajar (funcionamiento de la pantalla, del teclado, movimientos del ratón, etc.).
 - Software de aplicación que son los programas que maneja el usuario (tratamiento de textos, hojas de cálculo, bases de datos, etc).

El modelo básico de los ordenadores actuales se atribuye a von Neumann¹. Consta principalmente de la unidad central de proceso, la memoria y los dispositivos de entrada/salida, conectados entre sí como se muestra de forma esquemática en el diagrama mostrado en la Figura 1.1.

¹John von Neumann (1903-1957), matemático húngaro-estadounidense, con aportaciones importantes en numerosas áreas, y uno de los pioneros de los modernos ordenadores: a él se atribuye la “arquitectura de von Neumann”, base de los ordenadores modernos. Construyó el EDVAC, uno de los primeros ordenadores programables con programa almacenado en el propio ordenador, junto con los datos, poniendo en práctica la idea teórica debida a otro de los “padres” de la Informática y de las Ciencias de la Computación, Alan Turing (1912-1954).

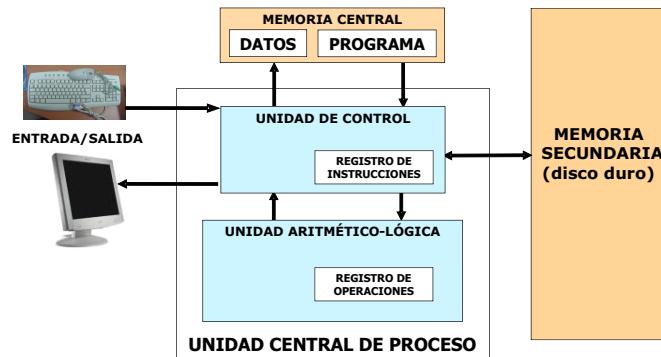


Figura 1.1: Componentes básicos de un ordenador

1.1.1 Unidad central de proceso

Unidad central de proceso o **CPU** (Central Process Unit) es el “cerebro” del ordenador: ejecuta las instrucciones de los programas y controla el funcionamiento de los distintos componentes del ordenador. Suele estar integrada en un chip denominado *microprocesador*. Los dos componentes más importantes de la CPU son:

- La **unidad de control** o **CU** (Control Unit), dirige y coordina la mayoría de las operaciones del ordenador. Interpreta cada instrucción enviada por un programa e inicia la acción apropiada para realizar esa instrucción. Para cada instrucción, la unidad de control repite un grupo de cuatro operaciones básicas, que constituye lo que se llama **ciclo de instrucción**:
 1. Leer (*fetch*) - es el proceso de obtener una instrucción de un programa o datos de la memoria.
 2. Decodificar (*decode*) - es el proceso de traducir la instrucción en comandos que el ordenador pueda ejecutar.
 3. Ejecutar (*execute*) - es el proceso de llevar a cabo los comandos.
 4. Escribir (*writeback*) - es el proceso de almacenar el resultado del paso de ejecución, “escribiéndolo” en la memoria.
- La **unidad aritmético-lógica** o **ALU** (Arithmetic Logic Unit), que realiza las operaciones elementales que constituyen el programa (sumar, multiplicar, comparar, etc).

La velocidad de un procesador se suele medir en **hercios (Hz)**: número de operaciones por segundo que puede realizar.

$$\begin{aligned} \textbf{1 Megahercio (Mz)} &= 10^6 \text{ operaciones/segundo} \\ \textbf{1 Gigahercio (Gz)} &= 10^9 \text{ operaciones/segundo} \end{aligned}$$

1.1.2 Memoria

Son los componentes de hardware en los que se almacena la información procesada por el ordenador. Generalmente, se distinguen entre la **memoria central** y la **memoria secundaria** (véase más adelante). La principal característica de las memorias es su:

- **Capacidad:** indica la cantidad de datos que puede almacenar. Se mide en bits (**bit**: **binary unit**), o múltiplos de bit. Los bits suelen agruparse de ocho en ocho: **1 byte** = 8 bits. Las unidades de almacenamiento están recogidas en la Tabla 1.1.

bit	0 ó 1
byte	B 8 bits
Kilobyte	KB 2^{10} B = 1024 B
Megabyte	MB 2^{10} KB = 1024 KB
Gigabyte	GB 2^{10} MB = 1024 MB
Terabyte	TB 2^{10} GB = 1024 GB

Tabla 1.1: Unidades de almacenamiento de información

1.1.3 Memoria central

Memoria central o principal, almacena tanto la secuencia de instrucciones del programa o programas que se esté ejecutando en cada momento como los datos que éste o éstos necesitan. Se distingue entre:

- Memoria **ROM** (**R**ead **O**nly **M**emory), (memoria de sólo lectura). Es permanente, esto es no se borra al apagar el ordenador y no se puede alterar (o se puede hacer difícilmente). Almacena códigos de programa grabados en fábrica necesarios para el funcionamiento del ordenador, como por ejemplo, la secuencia de instrucciones que hay que ejecutar cuando se enciende el ordenador (BIOS: Basic Input/Output System; POST: Power On Self Test).
- Memoria **RAM** (**RA**cces **M**emory), (memoria de acceso aleatorio). Almacena las instrucciones de los programas mientras que se ejecutan así como los datos que éstos necesitan o generen. Su contenido es volátil, es decir, se borra al apagar el ordenador. La denominación Acceso Aleatorio es antigua y se comenzó a utilizar para diferenciarla de otro tipo de memoria (cintas magnéticas) que era de acceso secuencial: para acceder a un dato había que recorrer todos los anteriores.

Con el objetivo de que el procesador pueda obtener los datos de la memoria central más rápidamente, la mayoría de los procesadores usan un tipo especial de memoria llamada **memoria caché** ó **RAM caché** (pequeña y de acceso muy rápido). En ella, además se almacenan los datos más recientes y las instrucciones inminentes.

La memoria central es un circuito que se puede imaginar como una enorme tabla que almacena información en cada una de sus **celdas** o **posiciones de memoria** (véase Figura 1.2).

Cada celda es una agrupación de bytes que se denomina **palabra**. Dependiendo del ordenador, existen palabras de 4 bytes (32 bits), 8 bytes (64 bits). Las palabras están numeradas correlativamente, comenzando desde 0. El número de una palabra es la **dirección** de esa palabra. La información almacenada en una palabra es su **contenido**.

1.1.4 Memoria secundaria

Como hemos mencionado, toda la información en la memoria central es accesible directamente en un tiempo muy corto; por ello dicha memoria es cara.

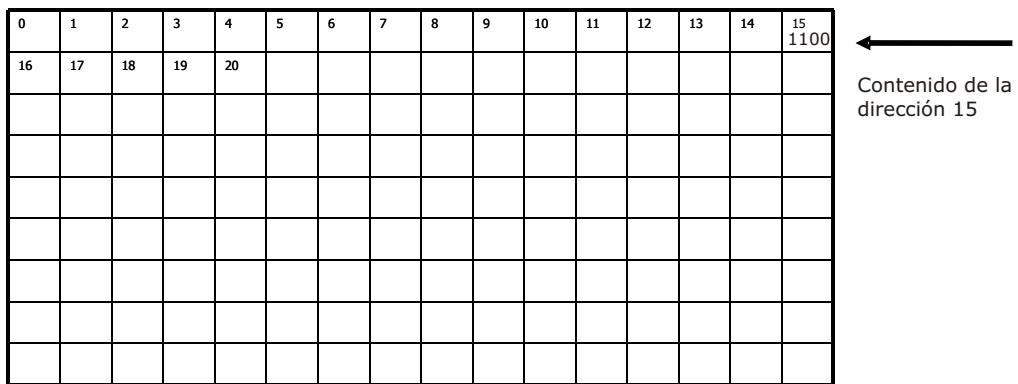


Figura 1.2: Memoria central de un ordenador

Memoria secundaria es una memoria más barata donde, a cambio de mayores tiempos de acceso, se puede conservar gran cantidad de información (disco duro, CD (Compact Disk), DVD (Digital Versatile Disk), memoria flash, etc). En ella se almacena todo lo que la unidad central de procesos (véase la Sección 1.1.1) no necesita urgentemente. La principal característica de este tipo de memorias es que su contenido no se pierde al apagar el ordenador.

1.1.5 Dispositivos de entrada/salida

Dispositivos de **entrada/salida** son todos los componentes de hardware anexos que permiten la comunicación entre el ordenador y el usuario. Los dispositivos de entrada son, por ejemplo, teclados, ratones, lectores de discos, micrófonos, etc. Los dispositivos de salida son, por ejemplo, monitores, impresoras, altavoces, etc.

1.2 Software

El **software** es el conjunto de instrucciones que hacen que el ordenador funcione y lleve a cabo tareas concretas, como procesar textos o conectarse a internet. El software también controla la forma en que se maneja el propio **hardware**.

Un **programa** es un conjunto detallado de instrucciones que indican al procesador la forma de llevar a cabo una tarea concreta.

El **software de sistema** son los programas indispensables para el funcionamiento del ordenador. Son el **sistema operativo**, los **compiladores** e **intérpretes** y los **utilitarios**.

- El sistema operativo es un conjunto complejo de muchos programas que se encargan de coordinar el funcionamiento de los distintos componentes, de iniciar la ejecución de otros programas y atender a sus requerimientos, de gestionar el almacenamiento y lectura de la memoria secundaria, de gestionar la interfaz gráfica de usuario (ventanas, iconos, barras de menús, ...), de interpretar las señales que se reciben desde el teclado o el ratón, de gestionar el envío de documentos a una impresora, etc. Ejemplos de sistemas operativos son Windows, MacOS, Unix, Linux, IOS, Android.
 - Los compiladores y traductores son programas capaces de traducir un programa escrito en

algún lenguaje de programación al lenguaje máquina (véase la Sección 1.3), que es el único que entiende el procesador. Cada lenguaje necesita su propio compilador o traductor.

- Se consideran utilitarios a aquellos programas que realizan tareas complementarias como antivirus, compresores de archivos, editores de texto, recuperación de archivos, visualizadores de imágenes, etc.

1.3 Lenguajes

Un **lenguaje de programación** es un conjunto de símbolos y reglas sintácticas y semánticas que sirven para describir las órdenes que controlan el comportamiento físico y lógico de una máquina.

Los procesadores de las máquinas sólo son capaces de entender y obedecer programas escritos en **lenguaje-máquina**, cuyas instrucciones son cadenas binarias (formadas por 0 y 1) que se pueden “cargar” directamente en la memoria central, sin necesidad de traducción. Sin embargo, el lenguaje-máquina es específico de cada tipo de ordenador, por lo que los programas escritos en dicho lenguaje no son portables en general de una máquina a otra. Además, debido a su representación totalmente numérica, son muy difíciles de escribir, leer y corregir.

El lenguaje **ensamblador** facilita (sólo un poco) esas tareas, ya que permite la escritura de las instrucciones básicas del lenguaje máquina en una forma más legible para el programador².

A modo de ejemplo, una instrucción básica ejecutable por la CPU podría ser:

copiar el contenido de la celda 164 en la celda 183.

En lenguaje-máquina esta instrucción podría escribirse (sólo es un ejemplo)

<u>01101100</u>	<u>10100100</u>	<u>10110101</u>
COPIAR	164	183

mientras que en lenguaje ensamblador se podría escribir de forma parecida a:

CP 164 183

Usualmente hay una correspondencia uno a uno entre las instrucciones simples de un código ensamblador y las de un código máquina. Por ello, el ensamblador sigue siendo un lenguaje de *bajo nivel*: por un lado el programador necesita conocer en profundidad la arquitectura de la máquina, por otro los programas escritos en ensamblador no son portables.

Los **lenguajes de alto nivel**, por el contrario, no obligan al programador a conocer los detalles del ordenador que utiliza. Las instrucciones se escriben en un formato flexible y más “humano”. Además, se pueden escribir, de forma sencilla, instrucciones mucho más complicadas: cada instrucción en un lenguaje de alto nivel corresponde a varias (incluso muchas) de lenguaje-máquina. La instrucción de los ejemplos anteriores, se podría escribir:

²Fue usado ampliamente en el pasado para el desarrollo de software, pero actualmente sólo se utiliza en contadas ocasiones, especialmente cuando se requiere la manipulación directa del hardware o se pretenden rendimientos inusuales de los equipos.

$$A = B$$

El ordenador sólo “comprende” los programas escritos en lenguaje-máquina. Cualquier otro debe ser **traducido**.

La traducción de un programa escrito en ensamblador a lenguaje-máquina la realiza un programa específico denominado también **ensamblador**.

Un programa escrito en un lenguaje de alto nivel se denomina **programa fuente**. Para traducirlo al lenguaje-máquina será necesario usar un **compilador** o un **intérprete**.

- **Compilador:** es un programa informático que traduce un programa fuente a un programa equivalente escrito en lenguaje-máquina, que se denomina **programa objeto**. El programa objeto puede ser almacenado como archivo en la memoria secundaria del ordenador para ser ejecutado posteriormente sin necesidad de volver a realizar la traducción.
- **Intérprete:** traduce el código fuente instrucción a instrucción y la ejecuta en el instante. No se crea un archivo o programa objeto, de modo que hay que volver a traducir cada vez que usemos el programa fuente correspondiente.

1.4 Tipos de datos

Los **datos** son la información que el ordenador almacena en la **memoria** y manipula mediante un programa. Estos datos pueden ser de distintos tipos: números, texto, imágenes, audio, ... La definición de un **tipo de dato** incluye la definición del conjunto de valores permitidos y las operaciones que se pueden llevar a cabo sobre estos valores.

Cuando se utiliza un dato en un programa es preciso que esté determinado su tipo para que el traductor/compilador sepa cómo debe tratarlo y almacenarlo. Dependiendo del lenguaje puede o no ser preciso declarar expresamente el tipo de cada dato. No todos los tipos de datos existen en todos los lenguajes de programación. Hay lenguajes más ricos que otros en este sentido.

1.4.1 Tipos de datos elementales

Los tipos de datos básicos más usuales son:

- **Números enteros:** números pertenecientes a un subconjunto finito de los números enteros.

Ejemplo 1.1

5 22 -1

- **Números reales:** números pertenecientes a un subconjunto finito de los números reales (constan de una parte entera y una parte fraccionaria).

Ejemplo 1.2

0.09 -31.423 3.0

- **Lógicos:** los dos valores lógicos, VERDADERO o FALSO.

Ejemplo 1.3

true false

- **Caracteres:** un conjunto finito de caracteres reconocidos por un ordenador.

Ejemplo 1.4

alfabéticos: A B C ... X Y Z a b c ... x y z

numéricos: 0 1 2 3 4 5 6 7 8 9

especiales: ; . = + - * & \$ < > etc.

1.4.2 Tipos de datos compuestos

Tipos de datos **compuestos** o **estructurados** son los construidos a partir de otros tipos de datos. Como ejemplo se tienen los siguientes:

- **Números complejos:** son datos formados por un par de datos reales y sirven para tratar números complejos.

Ejemplo 1.5

2+3i -3+i (i es la unidad imaginaria)

- **Cadenas de caracteres:** (tambien llamadas **string**) son una sucesión de caracteres.

Ejemplo 1.6

'Esto es una cadena de caracteres' 'string' '123abc'

- **Matrices:** son conjuntos de datos numéricos organizados para formar una matriz o un vector.

Ejemplo 1.7

[1, 0, 3, 4] (vector fila de dimensión 4)

1.5 Representación de datos numéricos en el ordenador

A continuación se explica, de forma simplificada cómo se manejan y almacenan cada uno de los tipos de datos en el ordenador.

Aunque los lenguajes de alto nivel permiten en alguna medida ignorar los detalles de la codificación interna de los datos, es preciso conocer algunos conceptos mínimos.

1.5.1 Bits, bytes y palabras

Bit El **bit** (*binary unit*) es la unidad mínima de información que puede almacenarse en un ordenador o en cualquier dispositivo digital. Un bit es un (diminuto) dispositivo electrónico capaz de tener dos estados: “apagado”, que se asimila al cero y “encendido”, que se asimila al uno.

Debido a que cada bit sólo dispone de dos estados posibles, los ordenadores utilizan para representar datos numéricos el **sistema binario** de numeración, en el que sólo hay dos **dígitos**: **0** y **1**.

Así, con un bit, sólo pueden representarse dos valores.

Byte Para representar más cantidad de valores es necesario usar más bits. Si se usan 2 bits se pueden representar $2^2 = 4$ valores distintos: 00, 01, 10, 11. Si se usan 4 bits se pueden representar $2^4 = 16$ valores distintos: 0000, 0001, 0010, 0011, ... etc. En general, con n bits se pueden representar 2^n valores distintos³.

Los bits se suelen agrupar en grupos de 8, formando un **byte**. En un byte sólo se pueden representar $2^8 = 256$ valores distintos. Por ejemplo, se pueden representar los números enteros no negativos desde 0 hasta 255.

Palabra Para almacenar datos, a su vez, se suelen considerar agrupaciones de 4 u 8 bytes (32 ó 64 bits), a las que se llama **palabra** (*word*). Estas agrupaciones determinan el conjunto de valores que se pueden almacenar en ellas: en 4 bytes (32 bits) se pueden almacenar $2^{32} = 4\,294\,967\,296$ valores distintos, mientras que en 8 bytes (64 bits) se pueden almacenar $2^{64} > 18 \times 10^{18}$ valores distintos.

Es claro que el cardinal (número de elementos) del conjunto de datos de un determinado tipo que se pueden almacenar en un ordenador dependerá del número de bits que se dediquen a ello. Por ejemplo, en una palabra de 32 bits, no se podrán almacenar más de 2^{32} números enteros distintos. Si se dedicaran más bits, el cardinal de dicho conjunto aumentaría, pero siempre será finito. Es decir, **el conjunto de números con los que se puede trabajar en un ordenador es finito**.

Este hecho tiene una importancia fundamental desde el punto de vista del cálculo numérico, ya que en un ordenador, contrariamente a lo que sucede cuando trabajamos con el conjunto de los números reales, el conjunto de números con los que podemos trabajar es **discreto**: entre dos números-máquina consecutivos no hay ninguno otro.

A los números (enteros o reales) que sí se pueden representar en el ordenador se les suele llamar **números-máquina**.

³Variaciones con repetición de 2 elementos (0, 1) tomados de n en $n = 2^n$.

1.5.2 Sistemas de numeración

Sistema de numeración decimal Es un sistema de numeración (posicional) que utiliza 10 dígitos básicos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 para representar cualquier número. La posición en dicha representación de cada dígito indica la potencia de 10 por la que hay que multiplicar el dígito.

Para un número N entero se tiene

$$\begin{aligned} N_{(10)} &= a_k a_{k-1} \dots a_1 a_0 \\ &= a_k \cdot 10^k + a_{k-1} \cdot 10^{k-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0, \quad a_i \in \{0, 1, 2, \dots, 9\} \quad \forall i \end{aligned}$$

Ejemplo 1.8

$$N = 3459 = 3 \cdot 10^3 + 4 \cdot 10^2 + 5 \cdot 10^1 + 9 \cdot 10^0 = 3000 + 400 + 50 + 9$$

Para la parte fraccionaria de un número real R se tiene

$$\begin{aligned} R_{(10)} &= 0.d_1 d_2 d_3 \dots d_n \dots \\ &= d_1 \cdot 10^{-1} + d_2 \cdot 10^{-2} + \dots + d_n \cdot 10^{-n} + \dots \\ &= 0.d_1 + 0.0d_2 + \dots + 0.00\dots d_n + \dots, \quad d_i \in \{0, 1, \dots, 9\} \quad \forall i \end{aligned}$$

Ejemplo 1.9

$$R = 0.325 = 3 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3} = \frac{3}{10} + \frac{2}{100} + \frac{5}{1000} = 0.3 + 0.02 + 0.005$$

Sistema de numeración binario En este sistema de numeración la posición de cada dígito indica la potencia de 2 por la que hay que multiplicar el dígito:

Para un número N entero se tiene

$$\begin{aligned} N_{(2)} &= b_k b_{k-1} \dots b_1 b_0 \\ &= b_k \cdot 2^k + b_{k-1} \cdot 2^{k-1} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0, \quad b_i \in \{0, 1\} \quad \forall i \end{aligned}$$

Ejemplo 1.10

$$N = 10111 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10000 + 100 + 10 + 1$$

Para la parte fraccionaria de un número real R se tiene

$$\begin{aligned} R_{(2)} &= 0.c_1 c_2 c_3 \dots c_n \dots \\ &= c_1 \cdot 2^{-1} + c_2 \cdot 2^{-2} + c_3 \cdot 2^{-3} + \dots + c_n \cdot 2^{-n} + \dots \\ &= 0.c_1 + 0.0c_2 + 0.00c_3 + \dots + 0.00\dots c_n + \dots, \quad c_i \in \{0, 1\} \quad \forall i \end{aligned}$$

Ejemplo 1.11

$$R = 0.1011 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.1 + 0.001 + 0.0001$$

Paso del sistema decimal al binario Para pasar de un número entero N en el sistema decimal a binario se dividen sucesivamente dicho número y los cocientes sucesivos por 2, hasta llegar a un cociente que sea igual a 1. Entonces se tiene $N_{(2)} = C_n R_n R_{n-1} \dots R_2 R_1$, con $C_n = 1$, siendo R_k el resto de la k -ésima división, como se muestra en la Figura 1.3.

:2	N	
	C_1	R_1
	C_2	R_2
	C_3	R_3

	C_n	R_n

Figura 1.3: Paso a binario de un número entero en sistema decimal.

Ejemplo 1.12

Pasar a binario el número $N = 77_{(10)}$

:2	77	
	38	1
	19	0
	9	1
	4	1
	2	0
	1	0

$$N = 77_{(10)} = 1001101_{(2)}$$

Para hacer la operación contraria basta con expresar el significado de la numeración posicional:

$$N = 1001101_{(2)} = 1 + 2^2 + 2^3 + 2^6 = 1 + 4 + 8 + 64 = 77_{(10)}$$

El procedimiento (iterativo) para obtener la representación binaria de un número con parte

fraccionaria en sistema decimal es:

Poner $R_0 = R$; calcular $2R_0$ y tomar $c_1 = [2R_0]$ (parte entera de $2R_0$).	Poner $R_1 = 2R_0 - c_1$; calcular $2R_1$ y tomar $c_2 = [2R_1]$ (parte entera de $2R_1$).
...	...
Poner $R_k = 2R_{k-1} - c_k$; calcular $2R_k$ y tomar $c_{k+1} = [2R_k]$ (parte entera de $2R_k$).	
...	

Este algoritmo se detendrá si:

1. Alguno de los $R_k = 0$, en cuyo caso el resto de dígitos de la parte fraccionaria binaria serán también ceros.
2. Algún R_k se repite, es decir, coincide con alguno anterior. En este caso hay un grupo de dígitos de la parte fraccionaria que se repite de forma periódica.

Si no ocurre ninguno de los casos anteriores, el número en cuestión tiene una parte fraccionaria binaria infinita.

Ejemplo 1.13

Pasar a binario el número $R = 0.23_{(10)}$

$$\begin{array}{lll} R_0 = 0.23; & 2R_0 = 0.46; & c_1 = 0 \\ R_1 = 2R_0 - c_1 = 0.46; & 2R_1 = 0.92; & c_2 = 0 \\ R_2 = 2R_1 - c_2 = 0.92; & 2R_2 = 1.84; & c_3 = 1 \\ R_3 = 2R_2 - c_3 = 0.84; & 2R_3 = 1.68; & c_4 = 1 \\ R_4 = 2R_3 - c_4 = 0.68; & 2R_4 = 1.36; & c_5 = 1 \\ R_5 = 2R_4 - c_5 = 0.36; & 2R_5 = 0.72; & c_6 = 0 \\ R_6 = 2R_5 - c_6 = 0.72; & 2R_6 = 1.44; & c_7 = 1 \end{array}$$

$$R = 0.23_{(10)} = 0.0011101\ldots_{(2)}$$

Ejemplo 1.14

Pasar a binario el número $R = 0.2_{(10)}$

$$\begin{array}{lll} R_0 = 0.2; & 2R_0 = 0.4; & c_1 = 0 \\ R_1 = 0.4; & 2R_1 = 0.8; & c_2 = 0 \\ R_2 = 0.8; & 2R_2 = 1.6; & c_3 = 1 \\ R_3 = 0.6; & 2R_3 = 1.2; & c_4 = 1 \\ R_4 = \textcolor{blue}{0.2} & & \end{array}$$

$$R = 0.2_{(10)} = 0.0011001100110011\ldots_{(2)} = \widehat{0.0011}_{(2)}$$

Ejemplo 1.15

Pasar a binario el número $R = 0.3125_{(10)}$

$$\begin{aligned} R_0 &= 0.3125; & 2R_0 &= 0.6250; & c_1 &= 0 \\ R_1 &= 0.6250; & 2R_1 &= 1.250; & c_2 &= 1 \\ R_2 &= 0.250; & 2R_2 &= 0.5; & c_3 &= 0 \\ R_3 &= 0.5; & 2R_3 &= 1.0; & c_4 &= 1 \\ R_4 &= \mathbf{0} \end{aligned}$$

$$R = 0.3125_{(10)} = 0.0101_{(2)}$$

Obsérvese en los ejemplos 1.13 y 1.14 cómo un número decimal con parte fraccionaria finita en base 10 puede tener una parte fraccionaria infinita en base 2.

Lo contrario no puede suceder: si $0.c_1c_2c_3\dots c_n$ es la expresión en base 2 de un número fraccionario, se tiene:

$$0.c_1c_2c_3\dots c_n_{(2)} = c_1 \cdot 2^{-1} + c_2 \cdot 2^{-2} + \dots + c_n \cdot 2^{-n} = \frac{c_1}{2} + \frac{c_2}{2^2} + \dots + \frac{c_n}{2^n}$$

donde los c_i valen 0 ó 1. Por tanto la expresión anterior no es más que una suma **finita** de términos de la forma $\frac{1}{2^k} = \left(\frac{1}{2}\right)^k = 0.5^k$, que tiene siempre un número finito de cifras decimales.

2^{-1}	$1/2$	0.5	2^{-5}	$1/32$	0.03125
2^{-2}	$1/4$	0.25	2^{-6}	$1/64$	0.015625
2^{-3}	$1/8$	0.125	2^{-7}	$1/128$	0.0078125
2^{-4}	$1/16$	0.0625	2^{-8}	$1/256$	0.00390625

Ejemplo 1.16

Pasar a decimal el número $R = 0.0011101_{(2)}$

$$\begin{aligned} 0.0011101_{(2)} &= \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^7} \\ &= 0.125 + 0.0625 + 0.03125 + 0.0078125 \\ &= 0.2265625_{(10)} \end{aligned}$$

$$R = 0.0011101_{(2)} = 0.2265625_{(10)}$$

Ejemplo 1.17

Pasar a binario el número $N = 178.023_{(10)}$

Pasamos a numeración binaria la parte entera del número $N_e = 178$, por un lado, y por otro la parte fraccionaria $N_f = 0.023$

$N_e = 178$		$N_f = 0.023$		
89	0	0.092	0.184	0
44	1	0.184	0.368	0
22	0	0.368	0.736	0
11	0	0.736	1.472	1
5	1	0.472	0.944	0
2	1	0.944	1.888	1
1	0	0.888	1.776	1
		0.776	1.552	1
		0.552	1.104	1

$$N = 178.023_{(10)} = 10110010.0000101111\dots_{(2)}$$

Operaciones aritméticas en binario Las operaciones en binario se realizan de la misma forma que en el sistema decimal:

Suma

$$\begin{array}{l}
 0 + 0 = 0 \\
 0 + 1 = 1 + 0 = 1 \\
 1 + 1 = 10 = 0 \text{ (acarreo 1)} \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 & 1 & 1 & 1 \\
 & 1 & 1 & 0 & 1 & 0 & 1 & = & 53_{10} \\
 + & 1 & 1 & 1 & 1 & 1 & 0 & = & 30_{10} \\
 \hline
 1 & 0 & 1 & 0 & 0 & 1 & 1 & = & 83_{10}
 \end{array}$$

Resta

$$\begin{array}{l}
 0 - 0 = 0 \\
 1 - 0 = 1 \\
 0 - 1 = 1 \text{ (acarreo 1)} \\
 1 - 1 = 0
 \end{array}
 \quad
 \begin{array}{r}
 & 1 & 1 & 0 & 1 & 0 & 1 & = & 53_{10} \\
 - & 1 & 1 & 1 & 1 & 1 & 0 & = & 30_{10} \\
 \hline
 & 1 & 1 & 1 & 1 & & & = & 23_{10}
 \end{array}$$

Multiplicación

$$\begin{array}{r}
 & 1 & 1 & 0 & 1 & 0 & 1 & = & 53_{10} \\
 \times & & & 1 & 0 & 1 & = & \times & 5_{10} \\
 \hline
 & 1 & 1 & 0 & 1 & 0 & 1 & & \\
 & 1 & 1 & 0 & 1 & 0 & 1 & & \\
 \hline
 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & = & 265_{10}
 \end{array}$$

División

$$\begin{array}{r}
 \begin{array}{r}
 1 & 1 & 0 & 1 & 0 & 1 \\
 - & 1 & 0 & 1 \\
 \hline
 0 & 0 & 1 & 1 & 0 \\
 & - & 1 & 0 & 1 \\
 \hline
 0 & 0 & 1 & 1
 \end{array}
 \quad \left| \begin{array}{r}
 1 & 0 & 1 \\
 \hline
 1 & 0 & 1 & 0
 \end{array} \right. \quad \frac{110101}{101} = 1010 + \frac{11}{101} \\
 \frac{53}{5} = 10 + \frac{3}{5}
 \end{array}$$

Notación hexadecimal La manipulación por humanos de números binarios (cadenas largas de ceros y unos) es tediosa y propensa a la comisión de errores. Por ello, en muchas ocasiones se utilizan otras notaciones. La más utilizada es la notación **hexadecimal**, basada en la numeración en base 16.

En ella se necesitan 16 dígitos para representar un número. Puesto que la numeración decimal sólo dispone de 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), se complementa con las primeras letras del abecedario, hasta conseguir los 16 dígitos: A=10, B=11, C=12, D=13, E=14, F=15. Así, en la numeración hexadecimal todos los números se representan mediante los dígitos

0 1 2 3 4 5 6 7 8 9 A B C D E F

El valor en base 10 de un número hexadecimal viene dado, como en otros casos, por la interpretación posicional de los dígitos.

Ejemplo 1.18

Hallar la representación decimal del número $H = 2AE1_{(16)}$

Escribimos el significado de la numeración posicional:

$$\begin{aligned}
 H = 2AE1_{(16)} &= 2 \times 16^3 + A \times 16^2 + E \times 16^1 + 1 \times 16^0 \\
 &= 2 \times 16^3 + 10 \times 16^2 + 14 \times 16^1 + 1 \times 16^0 \\
 &= 2 \times 4096 + 10 \times 256 + 14 \times 16 + 1 = 10977_{(10)}
 \end{aligned}$$

Para pasar un número decimal a hexadecimal se procede como para otras bases: se divide el número sucesivamente por 16 hasta llegar a un cociente que sea menor que 16 y se toman como dígitos el último cociente seguido de los restos, en orden inverso.

Ejemplo 1.19

Hallar la representación hexadecimal del número $D = 41486_{(10)}$

Dividimos sucesivamente por 16:

$$D = 41486 \quad \left| \begin{array}{r}
 :16 \\
 2592 \quad 14 \\
 162 \quad 0 \\
 10 \quad 2
 \end{array} \right. \quad \text{luego } D = 41486_{(10)} = A20E_{(16)}$$

La representación binaria de los 16 dígitos hexadecimales es:

Bin	Dec	Hex	Bin	Dec	Hex
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

La ventaja que ofrece esta notación es que, al ser 16 una potencia de 2 (2^4), la conversión binario ↔ hexadecimal puede hacerse de forma muy sencilla.

Para obtener la representación hexadecimal de un patrón de bits (un número binario) se puede proceder como sigue:

- Se agrupan los dígitos binarios de 4 en 4 comenzando por la derecha
- Se rellena con ceros a la izquierda si es necesario
- Se sustituye cada grupo de 4 bits por el correspondiente dígito hexadecimal.

En sentido contrario, para obtener la representación binaria de un número hexadecimal basta con sustituir cada dígito hexadecimal por su codificación binaria.

Ejemplo 1.20

Escribir el patrón de bits $P = 1010001000001110$ en notación hexadecimal

$$1010001000001110 = \underbrace{1010}_A \underbrace{0010}_2 \underbrace{0000}_0 \underbrace{1110}_E = A20E$$

En un ejemplo anterior se ha visto que A20E es la representación en base 16 del número decimal 41486. Se deja al lector la verificación de que

$$41486_{(10)} = A20E_{(16)} = 1010001000001110_{(2)} = P$$

La notación hexadecimal es muy utilizada, además de para facilitar la manipulación de patrones de bits, por ejemplo, en la definición de colores en diversos ámbitos (páginas web, editores, ...).

Notación octal La notación octal es similar a la hexadecimal, pero con el sistema de numeración en base 8 ($8 = 2^3$). En este sistema se utilizan 8 dígitos para representar cualquier número, 0, 1, 2, 3, 4, 5, 6, 7, y se necesitan 3 bits para representar cada uno de estos 8 dígitos ($2^3 = 8$).

Bin	Oct	Bin	Oct
000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

La conversión binario ↔ octal se realiza de forma similar al caso hexadecimal, pero agrupando ahora los dígitos binarios de 3 en 3.

Ejemplo 1.21

Escribir el patrón de bits $P = 1010001000001110$ en notación octal

$$1 \ 010 \ 001 \ 000 \ 001 \ 110 = \underbrace{001}_1 \ \underbrace{010}_2 \ \underbrace{000}_0 \ \underbrace{001}_1 \ \underbrace{110}_6 = 12016$$

En un ejemplo anterior se ha visto que P es la representación binaria del número decimal 41486. Luego

$$41486_{(10)} = A20E_{(16)} = 1010001000001110_{(2)} = 12016_{(8)}$$

Observación Distintos lenguajes de programación tienen diversas formas de escribir los números binarios, octales y hexadecimales. Una de las más habituales consiste en añadir un prefijo formado por un cero (0) seguido de una letra que indica el sistema: 0b para indicar un número binario, 0o para indicar un número octal y 0x para indicar un número hexadecimal:

0b100110 indica el número binario 100110

0o7103 indica el número octal 7103

0x21A1 indica el número hexadecimal 21A1

1.5.3 Almacenamiento de números enteros sin signo

Son los enteros no negativos. El rango de valores enteros sin signo que se pueden almacenar dependerá del número de bits que se dediquen a ello. Suelen ser 8, 16 o 32. Si se utilizan N bits, se pueden almacenar desde 0 hasta $2^N - 1$. La tabla 1.2 muestra los enteros sin signo que se pueden almacenar en cada caso.

8 bits	0 ... 255
16 bits	0 ... 65535
32 bits	0 ... 4294967295

Tabla 1.2: Rango de valores enteros sin signo representables, en función del número de bits.

Para almacenar estos números, se pasan a binario y se completan con ceros a la izquierda hasta

rellenar todos los bits disponibles.

Los enteros sin signo se utilizan, por ejemplo, para contar y para designar direcciones de memoria.

Ejemplo 1.22

¿Cómo se almacena el entero sin signo $N = 109_{(10)}$ en un registro de 16 bits?

Pasamos el número a binario:

$$N_e = 109 \quad N = 109_{(10)} = 1101101_{(2)}$$

54	1
27	0
13	1
6	1
3	0
1	1

El resto de dígitos a la izquierda se llenan con ceros:

0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1.5.4 Almacenamiento de números enteros con signo

Existen varios sistemas de almacenamiento de números enteros con signo.

Signo-magnitud En este sistema –el más básico– se reserva un bit para indicar el signo del número. Normalmente es el primero y se asocia el 0 con el signo + y el 1 con el signo -. En los bits restantes se almacena la representación binaria de la magnitud del número.

Al hacer esto se dispone de un bit menos para almacenar la magnitud. Así, mientras que con 8 bits se pueden almacenar, como entero sin signo, desde el 0 hasta el 255, con este sistema sólo se pueden almacenar desde el -127 hasta el +127. En general, con N bits, se podrán representar desde el $-(2^{N-1} - 1)$ hasta el $+(2^{N-1} - 1)$.

8 bits	-127	...	-0	+0	...	+127
16 bits	-32767	...	-0	+0	...	+32767
32 bits	-2147483647	...	-0	+0	...	+2147483647

Tabla 1.3: Rango de valores enteros representables en el sistema signo-magnitud.

Ejemplo 1.23

¿Cómo se almacena el entero $N = -109_{(10)}$ en un registro de 16 bits en el sistema **signo-magnitud**?

Como hemos visto antes, $109_{(10)} = 1101101_{(2)}$.

La representación en el sistema signo-magnitud sería, entonces:

1	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Para decodificar un patrón de bits en signo-magnitud con N bits, sólo habría que pasar a numeración decimal el número binario contenido en los últimos $N - 1$ bits, y luego asignarle signo + o - según que el primer bit sea cero o uno.

Ejemplo 1.24

Decodificar el patrón de bits siguiente, en el sistema signo-magnitud con 16 bits:

1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Se tiene $110001_{(2)} = 49_{(10)}$. Puesto que el primer bit es 1, el número es negativo.

El número almacenado es

-49

Hay que observar que, en este sistema, hay dos representaciones distintas para el cero, lo cual no es deseable (véase la tabla 1.4).

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= +0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= -0

Tabla 1.4: Las dos representaciones del cero en el sistema signo-magnitud con 16 bits.

Este sistema no se utiliza en los ordenadores actuales para almacenar números enteros con signo, ya que las operaciones de sumar y restar no son fáciles con ella y por la doble representación del cero.

Sin embargo, resulta útil para aplicaciones en las que no haya que realizar operaciones aritméticas con los números, como por ejemplo, en la transmisión de señales.

Complemento a 1 Se denomina **complemento a 1** de un número binario de n bits al número que se obtiene cambiando todos los unos por ceros y todos los ceros por unos (incluidos los

ceros no significativos, a la izquierda)⁴.

1	0	1	1	1	0	0	1		Número binario de 8 bits
↓	↓	↓	↓	↓	↓	↓	↓		
0	1	0	0	0	1	1	0		Complemento a 1
0	0	0	1	1	0	0	1		Número binario de 8 bits
↓	↓	↓	↓	↓	↓	↓	↓		
1	1	1	0	0	1	1	0		Complemento a 1

En el sistema de almacenamiento complemento a 1, en primer lugar, los números se almacenan igual que en el sistema signo-magnitud. Si el número es positivo, se queda tal cual. Si el número es negativo, la parte correspondiente a la magnitud se complementa a 1, es decir, se cambian los ceros por unos y viceversa. El primer 1, correspondiente al signo $-$, se queda igual.

El rango de números que se pueden almacenar es el mismo que con signo-magnitud, y el cero sigue teniendo dos representaciones (véase la tabla 1.5).

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= +0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= -0

Tabla 1.5: Las dos representaciones del cero en el sistema de complemento a 1 con 16 bits.

Ejemplo 1.25

¿Cómo se almacena el entero $N = 109_{(10)}$ en un registro de 16 bits en el sistema **complemento a 1**? ¿Y el número $N = -109_{(10)}$?

El número $109_{(10)} = 1101101_{(2)}$, positivo, se almacena

0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

El número $-109_{(10)} = -1101101_{(2)}$, negativo, se almacena con un 1 en el bit del signo y complementado a 1 en el resto, es decir:

1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Para decodificar un patrón de bits en complemento a 1 con N bits, se procede como sigue:

- Si el primer bit es un cero, se procede como en el sistema signo-magnitud: se pasa a numeración decimal el número binario contenido en los últimos $N - 1$ bits. Se le pone signo $+$.
- Si el primer bit es un uno, se aplica el complemento a 1 al número binario contenido en los $N - 1$ últimos bits y el número resultante se pasa a numeración decimal. Se le pone signo $-$.

⁴Las operaciones **bitwise** o **bit a bit** son las que se realizan sobre los números binarios actuando directamente sobre sus bits, uno a uno. Son operaciones *primitivas*, que llevan a cabo dispositivos electrónicos específicos, y son muy rápidas de realizar, incluso para procesadores sencillos de bajo costo. La operación de complemento a 1 es la operación bitwise **NOT**.

Ejemplo 1.26

Decodificar el siguiente patrón de bits, en el sistema de **complemento a 1**, con un registro de 16 bits

1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Puesto que el primer bit es un uno, se aplica el complemento a 1 a la parte de la magnitud (15 bits):

$$C_1^{15}(110001) = 11111111001110$$

Se tiene $11111111001110_{(2)} = 32718_{(10)}$. Puesto que el primer bit es 1, el número es negativo.

El número almacenado es

$$-32718$$

Si se aplica el complemento a 1 a la representación (completa, incluyendo el bit del signo) de un número positivo se obtiene su opuesto negativo y viceversa, si se aplica a uno negativo se obtiene su opuesto positivo. Si se aplica dos veces se obtiene el mismo número.

Este sistema de representación tampoco se usa hoy en día para los ordenadores, por las mismas razones que el anterior. Sin embargo, tiene utilidad en la detección y corrección de errores en la comunicación de datos. Además, es la base para el siguiente sistema de representación, que es el más ampliamente utilizado.

Complemento a 2 El complemento a 2 de un número binario se obtiene sumando 1 al complemento a uno.

$\begin{array}{ccccccccccccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ + & & & & & & & & & & 1 \\ \hline 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$	Número binario Complemento a 1 Sumar 1 Complemento a 2
--	---

Otra forma de obtenerlo es la siguiente: comenzando por la derecha, se dejan los bits como están hasta llegar al primer 1 (incluido). A partir de este, se aplica el complemento a 1 a todo el resto:

$\begin{array}{ccccccccccccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \downarrow & & & & \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$	Número binario Complemento a 2
---	-----------------------------------

En el sistema de representación de complemento a 2 con N bits se procede como sigue:

- se pasa el número a binario, ignorando el signo
- se rellena con ceros a la izquierda hasta completar los N bits
- si el número es positivo, no hay que hacer nada más
- si el número es negativo, se aplica el complemento a 2 a la representación obtenida.

Ejemplo 1.27

¿Cómo se almacena el entero $N = 109_{(10)}$ en un registro de 16 bits en el sistema **complemento a 2?** ¿Y el número $N = -109_{(10)}$?

El número $109_{(10)} = 1101101_{(2)}$, positivo, se almacena:

0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

El número $-109_{(10)}$ es negativo, luego se aplica el complemento a 2 con 16 bits a la representación de su opuesto $109_{(10)}$:

$$C_2^{16}(0000000001101101) = 111111110010011$$

1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Con el procedimiento anterior, se tendrían las siguientes representaciones para el $+0$ y el -0 respectivamente:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Sin embargo, **por convenio**, se asigna la representación del -0 al número -2^{N-1} , es decir, al -128 con registros de 8 bits, al -32768 con registros de 16 bits, etc. Así el rango de números que se pueden representar en el sistema de complemento a 2 queda aumentado en un número negativo más: es el mostrado en la tabla 1.6.

Así, para decodificar un patrón de bits en complemento a 2 con 16 bits se procede como sigue:

- Si el primer bit es un cero, se procede como en el sistema signo-magnitud: se pasa a numeración decimal el número binario contenido en los últimos 15 bits. Se le pone signo **+**.
- Si el primer bit es un uno y todos los demás son ceros, el número representado es $-2^{15} = -32768$.
- Si el primer bit es un uno, se aplica el complemento a 2 al número binario contenido en los últimos 15 bits, y el número binario resultante se pasa a numeración decimal. Se le pone signo **-**.

Ejemplo 1.28

Decodificar el siguiente patrón de bits, en el sistema de **complemento a 2**, con un registro de 16 bits

1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Puesto que el primer bit es un uno, se aplica el complemento a 2 a la parte de la magnitud (15 bits):

$$C_2^{15}(000000000110001) = 11111111001111$$

Se tiene $11111111001111_{(2)} = 32719_{(10)}$. Puesto que el primer bit es 1, el número es negativo.

El número almacenado es

$$-32719$$

8 bits	$-128 \dots + 0 \dots + 127$	
16 bits	$-32768 \dots + 0 \dots + 32767$	
32 bits	$-2147483648 \dots + 0 \dots + 2147483647$	

Tabla 1.6: Rango de valores enteros representados en el sistema con complemento a 2.

Como sucede con el complemento a 1, si se aplica el complemento a 2 a la representación de un número positivo se obtiene su opuesto negativo y viceversa, si se aplica a uno negativo se obtiene su opuesto positivo. Si se aplica dos veces se obtiene el mismo número.

El sistema complemento a dos es, actualmente, el sistema estándar de representación de enteros con signo, sobre todo debido a la simplicidad que presenta para las operaciones.

En efecto, para sumar dos números almacenados en complemento a 2 con n bits basta con sumar como binarios simples los patrones de bits representados (incluido el primer bit, del signo) y, caso de que la última suma (la correspondiente al bit del signo) produzca acarreo de 1, se descarta el 1 acarreado. El patrón de bits resultante es la representación en complemento a 2 con n bits de la suma.

Ejemplos 1.29

Realizar las siguientes sumas en la representación en complemento a 2 con 8 bits:

$$73 + 26 \quad 73 - 26 \quad -73 + 26 \quad -73 - 26$$

Escribimos la representación en complemento a 2 con 8 bits (C_2^8) de todos los números que aparecen:

$$73_{(10)} = 1001001_{(2)} = 01001001_{C_2^8}, \quad -73_{(10)} = -1001001_{(2)} = 10110111_{C_2^8},$$

$$26_{(10)} = 11010_{(2)} = 00011010_{C_2^8} \quad -26_{(10)} = -11010_{(2)} = 11100110_{C_2^8},$$

$$\begin{array}{r}
 + \quad 73_{(10)} = \begin{array}{ccccccc} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array} \\
 + \quad 26_{(10)} = \begin{array}{ccccccc} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array} \\
 \hline
 + \quad 99_{(10)} = \begin{array}{ccccccc} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{array} = 01100011_{C_2^8} = +1100011_{(2)} = +99_{(10)}
 \end{array}$$

$$\begin{array}{r}
 + \quad 73_{(10)} = \begin{array}{ccccccc} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array} \\
 - \quad 26_{(10)} = \begin{array}{ccccccc} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{array} \\
 \hline
 + \quad 47_{(10)} = \boxed{1} \begin{array}{ccccccc} 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{array} = 00101111_{C_2^8} = +101111_{(2)} = +47_{(10)}
 \end{array}$$

$$\begin{array}{r}
 - \quad 73_{(10)} = \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \end{array} \\
 + \quad 26_{(10)} = \begin{array}{ccccccc} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array} \\
 \hline
 - \quad 47_{(10)} = \begin{array}{ccccccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array} = 11010001_{C_2^8} = -101111_{(2)} = -47_{(10)}
 \end{array}$$

$$\begin{array}{r}
 - \quad 73_{(10)} = \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \end{array} \\
 - \quad 26_{(10)} = \begin{array}{ccccccc} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{array} \\
 \hline
 - \quad 99_{(10)} = \boxed{1} \begin{array}{ccccccc} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{array} = 10011101_{C_2^8} = -1100011_{(2)} = -99_{(10)}
 \end{array}$$

Representación sesgada Este sistema consiste en, una vez fijado un número entero positivo K , denominado **sesgo**, sumar este número a todos los demás. Así, en lugar de almacenar el entero con signo N , se almacena $N + K$ como un entero sin signo. Este sistema también se denomina **exceso a K** .

Normalmente, se elige como sesgo $K = 2^{n-1}$ o bien $K = 2^{n-1} - 1$, donde n es el número de bits que se emplean para el almacenamiento.

- Caso $K = 2^{n-1}$.

Puesto que $N + K$ se va a almacenar como un entero sin signo, sólo se pueden almacenar números N tales que $N + K \geq 0$. Así, el número más pequeño que se puede representar por este sistema es N_{min} tal que $N_{min} + 2^{n-1} = 0$, es decir $N_{min} = -2^{n-1}$. El número más grande es N_{max} tal que $N_{max} + 2^{n-1} = 2^n - 1$, es decir $N_{max} = 2^{n-1} - 1$. Ahora el cero tiene una única representación.

Por ejemplo, si se utilizan $n = 8$ bits, será $K = 2^7 = 128$, y se habla de **exceso a 128**. El número más pequeño que se puede representar en exceso a 128 es $N_{min} = -128$ y el más grande es $N_{max} = 127$.

- Caso $K = 2^{n-1} - 1$.

Ahora el número más pequeño que se puede almacenar es N_{min} tal que $N_{min} + 2^{n-1} - 1 = 0$, es decir $N_{min} = -2^{n-1} + 1$, y el más grande N_{max} tal que $N_{max} + 2^{n-1} - 1 = 2^n - 1$, es decir $N_{max} = 2^{n-1}$.

Por ejemplo, si se utilizan $n = 8$ bits, será $K = 2^7 - 1 = 127$, y se habla de **exceso a 127**. El número más pequeño que se puede representar en exceso a 127 es $N_{min} = -127$ y el más grande es $N_{max} = 128$.

Ejemplo 1.30

¿Cómo se almacena el número $-95_{(10)}$ en un registro de 8 bits con exceso a 127? ¿Y el número $95_{(10)}$?

Para almacenar el número $-95_{(10)}$ se comienza por sumarle 127, $-95 + 127 = 32_{(10)}$ y se pasa $32_{(10)}$ a binario, $32_{(10)} = 100000_{(2)}$. El resultado se rellena con ceros a la izquierda hasta completar los 8 bits. El valor binario resultante siempre comenzará por cero.

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

Para el número $95_{(10)}$ se tiene: $95 + 127 = 222_{(10)} = 11011110_{(2)}$. El valor binario resultante comienza por uno, ya que el valor a almacenar es mayor que 128.

1	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---

Para decodificar un patrón de 8 bits en exceso a 127, simplemente hay que decodificarlo como si fuera un entero sin signo y luego restarle el sesgo.

El sistema exceso a $2^{n-1} - 1$ se utiliza como parte de la codificación de números reales, como se verá a continuación.

Decodificación de un patrón de bits La pregunta que quizás cabe hacerse es: ¿cómo sabe la memoria del ordenador en qué sistema está almacenado un determinado patrón de bits en un registro de la memoria? Es decir, ¿cómo sabe cómo hay que decodificarlo? La respuesta es que NO LO SABE.

Una vez almacenado un determinado número (o dato) en la memoria, con cualquier sistema, no queda ningún rastro que indique cuál fue ese sistema. Es el programa que “lea” o el dispositivo que utilice ese dato el que debe gestionar que se decodifique/interprete de la manera adecuada.

A modo de ejemplo, se presenta en la tabla 1.7 cómo sería la representación de los números en los distintos sistemas explicados, en un registro imaginario de 4 bits. Obsérvese que, en todos los casos (salvo el sesgado), el primer bit de la izquierda es un 1 para los números negativos.

Patrón de bits	Sin signo	Signo + magnitud	Complemento a 1	Complemento a 2	Exceso a $2^3 = 8$
0000	0	+0	+0	0	-8
0001	1	+1	+1	+1	-7
0010	2	+2	+2	+2	-6
0011	3	+3	+3	+3	-5
0100	4	+4	+4	+4	-4
0101	5	+5	+5	+5	-3
0110	6	+6	+6	+6	-2
0111	7	+7	+7	+7	-1
1000	8	-0	-7	-8	0
1001	9	-1	-6	-7	1
1010	10	-2	-5	-6	2
1011	11	-3	-4	-5	3
1100	12	-4	-3	-4	4
1101	13	-5	-2	-3	5
1110	14	-6	-1	-2	6
1111	15	-7	-0	-1	7

Tabla 1.7: Decodificación de patrones de 4 bits con los distintos sistemas

Ejemplo 1.31

El patrón de bits

0	0	0	1	0	0	1	1	1	0	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

comienza por un 0, luego representa el mismo número entero en cualquiera de los sistemas: es el entero positivo 5037.

1	2^0	1
0	2^1	
1	2^2	4
1	2^3	8
0	2^4	
1	2^5	32
0	2^6	
1	2^7	128
1	2^8	256
1	2^9	512
0	2^{10}	
0	2^{11}	
1	2^{12}	4096
0	2^{13}	
0	2^{14}	
0	2^{15}	
5037		

Ejemplo 1.32

El patrón de bits siguiente comienza por un 1, luego puede representar distintos números.

1	0	0	1	0	0	1	1	1	0	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Entero sin signo			Signo+magnitud			Complemento a 1			Complemento a 2			Exceso a 32767		
1	2^0	1	1	2^0	1	0	2^0		1	2^0	1	1	2^0	1
0	2^1		0	2^1		1	2^1	2	1	2^1	2	0	2^1	
1	2^2	4	1	2^2	4	0	2^2		0	2^2		1	2^2	4
1	2^3	8	1	2^3	8	0	2^3		0	2^3		1	2^3	8
0	2^4		0	2^4		1	2^4	16	1	2^4	16	0	2^4	
1	2^5	32	1	2^5	32	0	2^5		0	2^5		1	2^5	32
0	2^6		0	2^6		1	2^6	64	1	2^6	64	0	2^6	
1	2^7	128	1	2^7	128	0	2^7		0	2^7		1	2^7	128
1	2^8	256	1	2^8	256	0	2^8		0	2^8		1	2^8	256
1	2^9	512	1	2^9	512	0	2^9		0	2^9		1	2^9	512
0	2^{10}		0	2^{10}		1	2^{10}	1024	1	2^{10}	1024	0	2^{10}	
0	2^{11}		0	2^{11}		1	2^{11}	2048	1	2^{11}	2048	0	2^{11}	
1	2^{12}	4096	1	2^{12}	4096	0	2^{12}		0	2^{12}		1	2^{12}	4096
0	2^{13}		0	2^{13}		1	2^{13}	8192	1	2^{13}	8192	0	2^{13}	
0	2^{14}		0	2^{14}		1	2^{14}	16384	1	2^{14}	16384	0	2^{14}	
1	2^{15}	32768	1		-	1		-	1		-	1	2^{15}	32768
37805			-5037			-27730			-27731			37805-32767=5038		

1.5.5 Almacenamiento de números reales

Los números reales se representan según la norma denominada **IEEE 754**⁵, establecida en 1985 para estandarizar la forma de almacenarlos en todas las marcas de ordenadores.

Esta norma establece dos formatos básicos para representar los números reales: **simple precisión** y **doble precisión**. En ambos casos lo que se representa es una expresión normalizada del número denominada **coma flotante**.

Como flotante La **coma flotante** (en inglés, *floating point*) es un modo “normalizado” de representación de números con parte fraccionaria que se adapta al orden de magnitud de los mismos, permitiendo así ampliar el rango de números representables.

Consiste en trasladar la coma (o punto) decimal hasta situarla detrás de la primera cifra significativa (la primera, comenzando por la izquierda, que no es nula) a la vez que se multiplica el número por la potencia adecuada de la base de numeración.

⁵IEEE son las siglas del *Institute of Electrical and Electronics Engineers*.

Ejemplo 1.33

$$17.02625 = 1.702625 \times 10^1 \text{ (en numeración decimal)}$$

$$0.0000000234 = 2.34 \times 10^{-8} \text{ (en numeración decimal)}$$

$$101.11000101 = 1.0111000101 \times 2^{10} \text{ (en numeración en base 2)}$$

Sea, pues, un número R dado por la siguiente representación binaria en coma flotante:

$$R = (\pm)m_1.m_2m_3\dots m_k \cdot 2^{\pm e},$$

donde $m_1m_2m_3\dots m_k$ se llama **parte significativa** o **mantisa** y el número $\pm e$ (entero, binario) es el **exponente**, es decir, el número de lugares que hay que trasladar la coma a derecha o izquierda para obtener el número inicial. Obsérvese que m_1 tiene que ser siempre igual a 1, ya que es, por definición, la primera cifra no nula empezando por la izquierda.

Ejemplo 1.34

$$R = 5.625_{(10)} = 101.101_{(2)} = 1.01101 \cdot 2^{10}$$

Simple precisión Para almacenar R en simple precisión se utiliza una palabra de 32 bits, que se distribuyen de la siguiente manera:

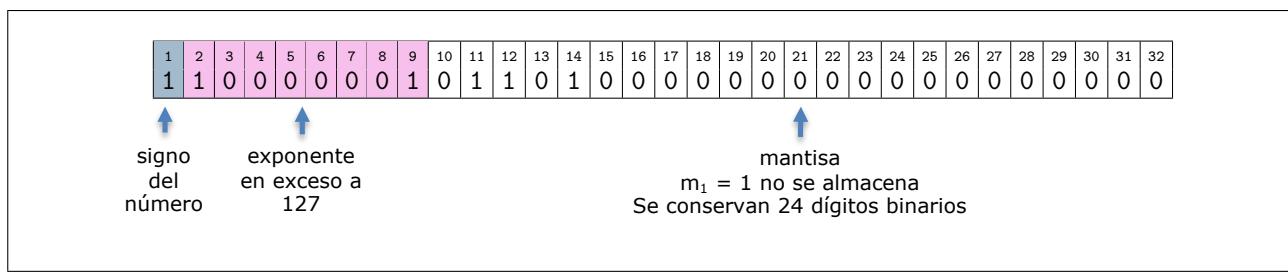
- 1 bit para el signo del número (0 si es positivo, 1 si es negativo).
- 8 bits para el exponente que se guarda codificado en exceso a 127.
- 23 bits para la mantisa. Como m_1 es siempre igual a 1 se utiliza el pequeño “truco” de no almacenarlo. Así, aunque se almacenan 23 cifras binarias se “conocen” en realidad 24. Al bit omitido se le llama **bit implícito**. Esta norma tiene algunas excepciones, que se presentan en la tabla 1.8.

Signo	Exponente	Mantisa	Interpretación
0	111...1	000...00	$+\infty = \text{Inf}$
1	111...1	000...00	$-\infty = -\text{Inf}$
0 o 1	111...1	$\neq 000...00$	NaN
0 o 1	000...0	000...00	Cero

Tabla 1.8: Algunos casos especiales en la norma IEEE 754. Cuando el exponente es todo de ceros, no existe el bit implícito.

Ejemplo 1.35

Representar en simple precisión el número $R = -5.625_{(10)} = -1.01101 \cdot 2^{10}$



Se tiene:

- El mayor exponente positivo que se puede almacenar en simple precisión es, como hemos visto, 128.
- El menor exponente negativo: -127 .
- El número de mayor magnitud: $\approx 2^{128} \approx 3 \times 10^{38}$.
- El número de menor magnitud: $\approx 2^{-127} \approx 5 \times 10^{-39}$.

Doble precisión Para representación en doble precisión se utilizan 64 bits distribuidos de la siguiente forma:

- 1 bit para el signo del número.
- 11 bits para el exponente que se guarda codificado en exceso a $2^{10} - 1 = 1023$.
- 52 bits para la mantisa.

En este caso se tiene:

- El mayor exponente positivo que se puede almacenar en doble precisión es, como hemos visto, $2^{10} = 1024$.
- El menor exponente negativo: -1023 .
- El número de mayor magnitud: $\approx 10^{308}$.
- El número de menor magnitud: $\approx 10^{-307}$.

Overflow y underflow Es claro que usando palabras con un número finito de bits (32 o 64) sólo podemos representar una cantidad finita de números, en consecuencia podemos encontrar los siguientes fenómenos:

Overflow fenómeno que se produce cuando el resultado de un cálculo es un número real con exponente demasiado grande (en el caso de la simple precisión, > 128). Cuando se produce un overflow durante la ejecución de un programa, los ordenadores modernos generan un “evento” de error y como resultado devuelven el símbolo **Inf**.

Underflow fenómeno que se produce cuando el resultado de un cálculo es un número real con exponente demasiado pequeño (en los ejemplos arriba mencionados < -127). Cuando se produce durante la ejecución de un programa, normalmente se sustituye el número por cero.

1.5.6 Errores

Al realizar cálculos en el ordenador, es inevitable cometer errores. Grossos modo, estos errores pueden ser de tres tipos:

1. **Errores en los datos de entrada:** vienen causados por los errores al realizar mediciones de magnitudes físicas.
2. **Errores de truncamiento (o discretización):** En la mayoría de las ocasiones, los algoritmos matemáticos que se utilizan para calcular un resultado no son capaces de calcular su valor exacto, limitándose a calcular una **aproximación** del mismo. Este error es inherente al algoritmo de cálculo utilizado.
3. **Errores de redondeo:** aparecen debido a la cantidad finita de números que podemos representar en un ordenador. Por ejemplo, consideramos las dos siguientes mantisas-máquina consecutivas que podemos almacenar en 23 bits (recordar que el primer dígito es siempre 1 y por tanto no se guarda):

$$R_1 = \underbrace{1.00000000000000000000000}_{24 \text{ dígitos}}$$

$$R_2 = \underbrace{1.00000000000000000000001}_{24 \text{ dígitos}}$$

Ningún número real entre R_1 y R_2 es representable en un ordenador que utilice 23 bits para la mantisa. Tendrá que ser aproximado bien por R_1 , bien por R_2 , cometiendo con ello un error E que verifica:

$$E \leq \frac{|R_1 - R_2|}{2} = \frac{0.00000000000000000000001}{2} = \frac{2^{-23}}{2} = 2^{-24} < 10^{-7}$$

De esta forma, en el almacenamiento de la mantisa se comete un error menor o igual que 10^{-7} o, dicho de otra forma, sólo los 7 primeros dígitos significativos de un número decimal almacenado en el ordenador en simple precisión se pueden considerar exactos.

Ejemplo 1.36

$$\begin{aligned} R = 0.1_{(10)} &= \widehat{0.00011}_{(2)} \\ &= 0.00011001100110011\dots \\ &= \underbrace{1.10011001100110011001100\dots}_{24} \times 2^{-100} \end{aligned}$$

R no es un número-máquina (su mantisa binaria tiene infinitos dígitos).

La aproximación de R por redondeo en simple precisión es:

$$\overline{R} = \underbrace{1.1001100110011001101}_{24 \text{ dígitos}} \times 2^{-100}.$$

Aritmética en coma flotante Para llevar a cabo una operación aritmética en un ordenador, hay que tener en cuenta que los números con los que trabajamos pueden no ser exactamente los de partida, sino sus aproximaciones por redondeo. Incluso aunque los números sean números-máquina (representables en palabras de 32 bits), los resultados que obtengamos puede que no lo sean. A continuación vamos a ver algunos ejemplos de cálculos efectuados en la aritmética de coma flotante. Por facilidad, lo hacemos en el caso (hipótetico) de un ordenador que trabaje con números en expresión decimal con 7 dígitos para la mantisa, y con aproximación por redondeo.

Ejemplo 1.37

Para **sumar** dos números en coma flotante, primero se representan con el mismo exponente, luego se suman y el resultado se redondea:

$$\begin{aligned}x_1 &= 0.1234567 \cdot 10^3 = 0.123456700000 \cdot 10^3 (= 123.4567) \\x_2 &= 0.3232323 \cdot 10^{-2} = 0.000003232323 \cdot 10^3 (= 0.003232323)\end{aligned}$$

$$\begin{array}{r} 0.123456700000 \cdot 10^3 \\ + 0.000003232323 \cdot 10^3 \\ \hline 0.1234599\textcolor{red}{32323} \cdot 10^3 \quad (\text{sólo se almacenan los 7 primeros dígitos}) \\ 0.1234599 \cdot 10^3 \quad (\text{resultado-máquina de } x_1 + x_2) \end{array}$$

Obsérvese que las 5 últimas cifras de x_2 no contribuyen a la suma.

Ejemplo 1.38

Para **multiplicar** (la división se hace por algoritmos más complicados) dos números en coma flotante se multiplican las mantisas y se suman los exponentes. El resultado se normaliza:

$$\begin{aligned}x_1 &= 0.4734612 \cdot 10^3 \\x_2 &= 0.5417242 \cdot 10^5 \\ \\ \times & \begin{array}{r} 0.4734612 \cdot 10^3 \\ \times 0.5417242 \cdot 10^5 \\ \hline 0.2564853\textcolor{red}{8980104} \cdot 10^8 \\ 0.2564854 \cdot 10^8 \quad (\text{resultado-máquina de } x_1 \cdot x_2) \end{array}\end{aligned}$$

2

Introducción a MATLAB. Operaciones elementales



MATLAB es un potente paquete de software para computación científica, orientado al cálculo numérico, a las operaciones matriciales y especialmente a las aplicaciones científicas y de ingeniería. Ofrece lo que se llama un entorno de desarrollo integrado (IDE), es decir, una herramienta que permite, en una sola aplicación, ejecutar órdenes sencillas, escribir programas utilizando un editor integrado, compilarlos (o interpretarlos), depurarlos (buscar errores) y realizar gráficas.

Puede ser utilizado como simple calculadora matricial, pero su interés principal radica en los cientos de funciones tanto de propósito general como especializadas que posee, así como en sus posibilidades para la visualización gráfica.

MATLAB posee un lenguaje de programación propio, muy próximo a los habituales en cálculo numérico (Fortran, C, ...), aunque mucho más *tolerante* en su sintaxis, que permite al usuario escribir sus propios *scripts* (conjunto de comandos escritos en un fichero, que se pueden ejecutar con una única orden) para resolver un problema concreto y también escribir nuevas funciones con, por ejemplo, sus propios algoritmos, o para *modularizar* la resolución de un problema complejo. MATLAB dispone, además, de numerosas *Toolboxes*, que le añaden funcionalidades especializadas.

Numerosas contribuciones de sus miles de usuarios en todo el mundo pueden encontrarse en la web de The MathWorks: <http://www.mathworks.es>

2.1 Introducción

Al iniciar MATLAB nos aparecerá una ventana más o menos como la de la Figura 2.1 (dependiendo del sistema operativo y de la versión)

Si la ubicación de las ventanas integradas es diferente, se puede volver a ésta mediante:

Menú Desktop → Desktop Layout → Default

Se puede experimentar con otras disposiciones. Si hay alguna que nos gusta, se puede salvar guardando con

Menú Desktop → Desktop Layout → Save Layout ...

dándole un nombre, para usarla en otras ocasiones. De momento, sólo vamos a utilizar la ventana principal de MATLAB: **Command Window** (ventana de comandos). A través de esta ventana nos comunicaremos con MATLAB, escribiendo las órdenes en la línea de comandos. El símbolo **>>** al comienzo de una línea de la ventana de comandos se denomina *prompt* e indica que MATLAB está desocupado, disponible para ejecutar nuestras órdenes.

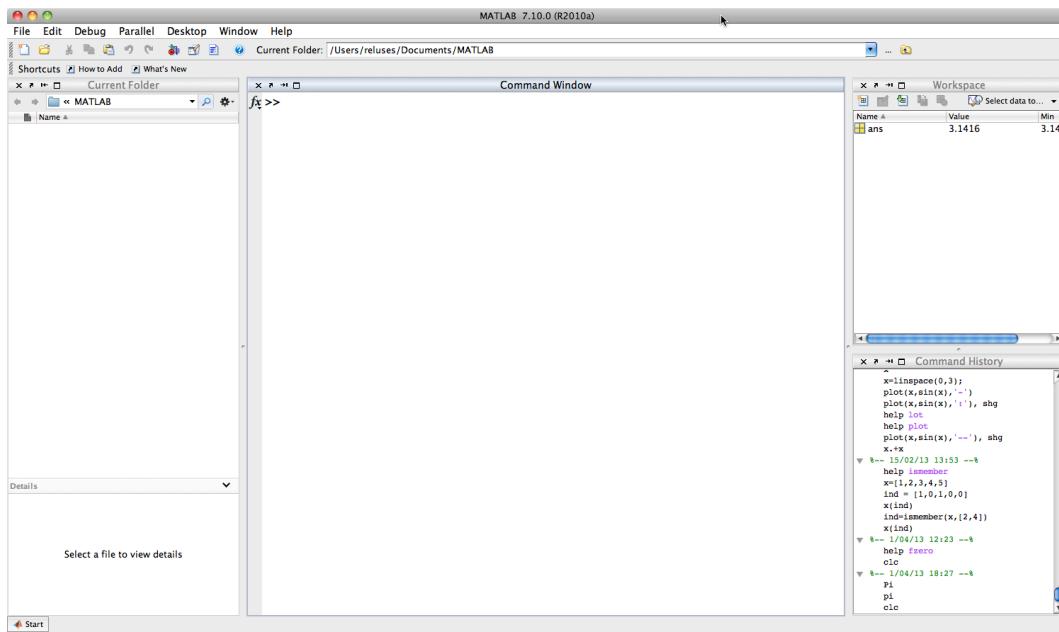


Figura 2.1: La ventana de MATLAB

2.1.1 Objetos y sintaxis básicos

Las explicaciones sobre las funciones/comandos que se presentan en estas notas están muy resumidas y sólo incluyen las funcionalidades que, según el parecer subjetivo de la autora, pueden despertar más interés. La mayoría de las funciones tienen mas y/o distintas funcionalidades que las que se exponen aquí. Para una descripción exacta y exhaustiva es preciso consultar la Ayuda on-line.

Los tipos básicos de datos que maneja MATLAB son números reales, booleanos (valores lógicos) y cadenas de caracteres (string). También puede manipular distintos tipos de números enteros, aunque sólo suele ser necesario en circunstancias específicas.

En MATLAB, por defecto, los números son codificados como números reales en coma flotante en doble precisión. La precisión, esto es, el número de bits dedicados a representar la mantisa y el exponente, depende de cada (tipo de) máquina.

MATLAB manipula también otros objetos, compuestos a partir de los anteriores: números complejos, matrices, *cells*, estructuras definidas por el usuario, clases Java, etc.

El objeto básico de trabajo de MATLAB es una matriz bidimensional cuyos elementos son números reales o complejos. Escalares y vectores son considerados casos particulares de matrices. También se pueden manipular matrices de cadenas de caracteres, booleanas y enteras.

El lenguaje de MATLAB es **interpretado**, esto es, las instrucciones se traducen a lenguaje máquina una a una y se ejecutan antes de pasar a la siguiente. Es posible escribir varias instrucciones en la misma línea, separándolas por una coma o por punto y coma. Las instrucciones que terminan por punto y coma no producen salida de resultados por pantalla.

Algunas constantes numéricas están predefinidas (ver sección 2.5.2)

MATLAB distingue entre mayúsculas y minúsculas: **pi** no es lo mismo que **Pi**.

MATLAB conserva un historial de las instrucciones escritas en la línea de comandos. Se pueden recuperar instrucciones anteriores, usando las teclas de flechas arriba y abajo. Con las flechas izquierda y derecha nos podemos desplazar sobre la línea de comando y modificarlo.

Se pueden salvaguardar todas las instrucciones y la salida de resultados de una sesión de trabajo de MATLAB a un fichero:

```
>> diary nombre_fichero  
>> diary off % suspende la salvaguarda
```

2.1.2 Documentación y ayuda *on-line*

- Ayuda on-line en la ventana de comandos

```
>> help nombre_de_comando
```

La información se obtiene en la misma ventana de comandos. Atención:

- Ayuda on-line con ventana de navegador

```
>> helpwin
```

ó bien Menú Help ó bien Botón Start → Help.

Además, a través del navegador del Help se pueden descargar, desde The MathWorks, guías detalladas, en formato pdf, de cada capítulo.

2.1.3 Scripts y funciones. El editor integrado

Scripts En términos generales, en informática, un *script* (guión o archivo por lotes) es un conjunto de instrucciones (programa), usualmente simple, guardadas en un fichero (usualmente de texto plano) que son ejecutadas normalmente mediante un intérprete. Son útiles para automatizar pequeñas tareas. También puede hacer las veces de un "programa principal" para ejecutar una aplicación.

Así, para llevar a cabo una tarea, en vez de escribir las instrucciones una por una en la línea de comandos de MATLAB, se pueden escribir las órdenes una detrás de otra en un fichero. Para ello se puede utilizar el **Editor integrado de MATLAB**. Para iniciarla, basta pulsar el icono hoja en blanco (**New script**) de la barra de MATLAB, o bien

File → New → Script

Un *script* de MATLAB debe guardarse en un fichero con sufijo **.m** para ser reconocido. El nombre del fichero puede ser cualquiera *razonable*, es decir, sin acentos, sin espacios en blanco y sin caracteres «extraños».

Para editar un *script* ya existente, basta hacer *doble-click* sobre su ícono en la ventana **Current Folder**.

Para ejecutar un *script* que esté en el directorio de trabajo, basta escribir su nombre (sin el sufijo) en la línea de comandos.

M-Funciones Una función (habitualmente denominadas M-funciones en MATLAB), es un programa con una «interfaz» de comunicación con el exterior mediante argumentos de entrada y de salida.

Las funciones MATLAB responden al siguiente formato de escritura:

```
function [argumentos de salida] = nombre(argumentos de entrada)
%
% comentarios
%
.....
instrucciones (normalmente terminadas por ; para evitar eco en pantalla)
.....
```

Las funciones deben guardarse en un fichero con el mismo nombre que la función y sufijo **.m**. Lo que se escribe en cualquier línea detrás de **%** es considerado como comentario.

Ejemplo 2.1

El siguiente código debe guardarse en un fichero de nombre **areaaequi.m**.

```
function [sup] = areaaequi(long)
%
% areaaequi(long) devuelve el area del triangulo
% equilatero de lado = long
%
sup = sqrt(3)*long^2/4;
```

La primera línea de una M-función siempre debe comenzar con la claúsula (palabra reservada) **function**. El fichero que contiene la función debe estar *en un sitio en el que MATLAB lo pueda encontrar*, normalmente, en la carpeta de trabajo.

Se puede ejecutar la M-función en la misma forma que cualquier otra función de MATLAB:

Ejemplos 2.2

```
>> areaaequi(1.5)
ans =
    0.9743
>> rho = 4 * areaaequi(2) +1;
>> sqrt(areaaequi(rho))
ans =
    5.2171
```

Los breves comentarios que se incluyen a continuación de la línea que contiene la cláusula **function** deben explicar, brevemente, el funcionamiento y uso de la función. Además, constituyen la ayuda *on-line* de la función:

Ejemplo 2.3

```
>> help areaequi  
areaequi(long) devuelve el area del triangulo  
equilatero de lado = long
```

Se pueden incluir en el mismo fichero otras funciones, denominadas subfunciones, a continuación de la primera¹, pero sólo serán *visibles* para las funciones del mismo fichero. En versiones anteriores a la R2016, no se podían incluir M-funciones en el fichero de un *script*.

Funciones anónimas Algunas funciones *sencillas*, que devuelvan el resultado de una expresión, se pueden definir mediante una sola instrucción, en mitad de un programa (script o función) o en la línea de comandos. Se llaman funciones anónimas. La sintaxis para definirlas es:

```
nombre_funcion = @(argumentos) expresion_funcion
```

Ejemplo 2.4 (Función anónima para calcular el área de un círculo)

```
>> area_circulo = @(r) pi * r.^2;  
>> area_circulo(1)  
ans =  
    3.1416  
>> semicirc = area_circulo(3)/2  
semicirc =  
    14.1372
```

Las funciones anónimas pueden tener varios argumentos y hacer uso de variables previamente definidas:

Ejemplo 2.5 (Función anónima de dos variables)

```
>> a = 2;  
>> mifun = @(x,t) sin(a*x).*cos(t/a);  
>> mifun(pi/4,1)  
ans =  
    0.8776
```

¹También es posible definir funciones anidadas, esto es, funciones «insertadas» dentro del código de otras funciones. (Se informa aquí para conocer su existencia. Su utilización es delicada.)

Si, con posterioridad a la definición de la función `mifun`, se cambia el valor de la variable `a`, la función no se modifica: en el caso del ejemplo, seguirá siendo `mifun(x,t)=sin(2*x).*cos(t/2)`.

2.1.4 *Workspace* y ámbito de las variables

Workspace (espacio de trabajo) es el conjunto de variables que en un momento dado están definidas en la memoria del MATLAB.

Las variables creadas desde la línea de comandos de MATLAB pertenecen al **Base Workspace** (espacio de trabajo base; es el que se puede «hojear» en la ventana **Workspace**). Los mismo sucede con las variables creadas por un *script* que se ejecuta desde la línea de comandos. Estas variables permanecen en el **Base Workspace** cuando se termina la ejecución del script y se mantienen allí durante toda la sesión de trabajo o hasta que se borren.

Sin embargo, las variables creadas por una M-función pertenecen al espacio de trabajo de dicha función, que es independiente del espacio de trabajo base. Es decir, las variables de las M-funciones son **locales**: MATLAB reserva una zona de memoria cuando comienza a ejecutar una M-función, almacena en esa zona las variables que se crean dentro de ella, y «borra» dicha zona cuando termina la ejecución de la función.

Esta es una de las principales diferencias entre un *script* y una M-función: cuando finaliza la ejecución de un *script* se puede «ver» y utilizar el valor de todas las variables que ha creado el script en el **Workspace**; en cambio, cuando finaliza una función no hay rastro de sus variables en el **Workspace**.

Para hacer que una variable local pertenezca al espacio de trabajo de otra función, hay que declararla **global** en cada uno de ellos: la orden

```
global a suma error
```

en una función hace que las variables `a`, `suma` y `error` sean almacenadas en un sitio especial y puedan ser utilizadas por otras funciones que también las declaren como globales.

2.2 Números

Enteros Los números enteros se escriben sin punto decimal

Ejemplo 2.6

23 321 -34

Reales Los números reales se pueden escribir en notación fija decimal y en notación científica, utilizando el **punto** decimal (no la coma):

Ejemplo 2.7

23. -10.1 22.0765

$2.\text{e-}2 = 2 \times 10^{-2} = 0.02$

$2.07\text{e+}5 = 2.07 \times 10^5 = 207000$

Complejos Se escriben utilizando el símbolo pre-definido **i**:

Ejemplo 2.8

$2 + 3i$ $-4.07 - 2.3i$

2.3 Operaciones aritméticas

Las operaciones aritméticas habituales se representan normalmente mediante los símbolos siguientes:

Descripción	Símbolo
Exponenciación	$^$
Suma	$+$
Resta	$-$
Multiplicación	$*$
División	$/$

Tabla 2.1: Operadores aritméticos elementales

2.3.1 Reglas de prioridad

Las operaciones aritméticas NO se efectúan siempre en el orden en que están escritas. El orden viene determinado por las reglas siguientes:

1. Exponenciaciones.
2. Multiplicaciones y divisiones.
3. Sumas y restas.
4. Dentro de cada grupo, de izquierda a derecha.

Para cambiar este orden se usan los paréntesis.

5. Si hay paréntesis, su contenido se calcula antes que el resto.
6. Si hay paréntesis anidados, se efectúan primero los más internos.

Ejemplos 2.9

$$\begin{aligned}
 2 + 3 * 4 &= 2 + 12 = 14 \\
 (2 + 3) * 4 &= 5 * 4 = 20 \\
 1 / 3 * 2 &= 0.3333\dots * 2 = 0.6666 \\
 1 / (3 * 2) &= 1 / 6 = 0.166666\dots \\
 2 + 3 ^ 4 / 2 &= 2 + 81 / 2 = 2 + 40.5 = 42.5 \\
 4 ^ 3 ^ 2 &= (4 ^ 3) ^ 2 = 64 ^ 2 = 4096 \\
 a+b/2 &= a + \frac{b}{2}, \quad (a+b)/2 = \frac{a+b}{2} \\
 1/2*x &= \frac{1}{2} x, \quad 1/(2*x) = \frac{1}{2 x}
 \end{aligned}$$

Ejercicio 2.10

Calcular $2^{1/2^3}$ mediante una única expresión.

Resultado : 1.0905

Ejercicio 2.11

Calcular

$$\frac{4,1^{\frac{0,2+1}{2}}}{\frac{2}{0,4}}$$

$$\frac{4,1^{\frac{0,2+1}{2}}}{\frac{2}{0,4}} = \frac{4,1^{\frac{3}{2}}}{2^{\frac{1}{3}}}$$

mediante una única expresión.

Resultado : 0.3882

2.4 Forma en que se muestran los resultados

Obsérvese que, por defecto, cuando se hace un cálculo en la ventana de comandos de MATLAB, aparece el resultado asignado a la variable `ans` (*answer*). Los resultados numéricos reales son mostrados en notación fija decimal con 4 cifras decimales si su valor absoluto está comprendido entre 10^{-3} y 10^3 . En caso contrario son mostrados en notación científica.

Ejemplos 2.12

```
>> 0.002 será mostrado como 0.0020  
>> 0.0000123 será mostrado como 1.2300e-05  
>> -709.2121211 será mostrado como -709.2121  
>> 1003.010101 será mostrado como 1.0030e+03
```

Se puede modificar este comportamiento mediante el comando `format`

<code>format</code>	Cambia el formato de salida a su valor por defecto, <code>short</code>
<code>format short</code>	El formato por defecto
<code>format long</code>	Muestra 15 dígitos
<code>format short e</code>	Formato short, en coma flotante
<code>format long e</code>	Formato long, en coma flotante
<code>format rat</code>	Muestra los números como cociente de enteros

2.5 Variables

Cuando hay que hacer cálculos largos interesa guardar resultados intermedios para utilizarlos más adelante. ¿Dónde se guardan estos resultados? En **variables**.

Una **variable** es un nombre simbólico que identifica una parte de la memoria, en la que se pueden guardar números u otro tipo de datos. El contenido de una variable se puede recuperar y modificar cuantas veces se desee, a lo largo de una sesión de trabajo o durante la ejecución de un programa.

En MATLAB, los nombres de las variables deben estar formados por letras y números, hasta un máximo de 19, y comenzar por una letra. Se debe tener en cuenta que se distingue entre letras mayúsculas y minúsculas.

Ejemplos 2.13 (Nombres válidos de variables)

a	peso	HarryPotter
XY	Peso	Darwin
ab12	PESO	PericoPalotes
kr10	pEs0	PagaFantas

En la mayoría de los lenguajes de programación es necesario especificar el tipo de dato que va a contener una variable antes de usarla, declarándolo con las ordenes específicas. En el lenguaje de programación de MATLAB, las variables no necesitan ningún tipo de declaración y pueden almacenar sucesivamente distintos tipos de datos: enteros, reales, escalares, matriciales, caracteres, etc. Se crean, simplemente, **asignándoles un valor**.

2.5.1 Instrucciones de asignación

Una **instrucción de asignación** sirve para almacenar un valor en una variable. Su sintaxis en MATLAB es:

```
variable = expresión
```

que debe ser interpretada como: evaluar el resultado de la **expresión** y almacenarlo en la dirección de memoria correspondiente a **variable**.

El signo **=** significa: **almacenar** el resultado de la **derecha** en la variable que está a la **izquierda**. Por ello, el orden en esta instrucción es fundamental: NO se puede escribir **expresión = variable**.

La acción de almacenar un valor en una variable hace que se pierda el valor que, eventualmente, tuviera dicha variable previamente.

Ejemplos 2.14

```
a = 2                      :: guardar en la variable a el valor 2  
b = -4                     :: guardar en la variable b el valor -4  
raiz = sqrt(2*b+8*a)       :: guardar en la variable raiz el resultado  
                           de la expresión de la derecha  
a = a + 1                  :: sumar 1 al contenido de a (guardar 3 en a)
```

```
variable = expresión;      % cuando la orden se termina por punto y coma  
                           % el resultado no es mostrado en la pantalla
```

Ejemplos 2.15

```
>> a = 1/3                  :: sin punto y coma  
ans =  
     0.3333  
  
>> a = 1/3;                 :: con punto y coma  
>>
```

2.5.2 Variables pre-definidas

`pi` contiene el valor del número π .

`i`, `j` representan la unidad imaginaria.

`Inf` representación simbólica del infinito; aparece cuando se hacen cálculos cuyo resultado es demasiado grande para ser almacenado en el ordenador (*overflow*).

`NaN` (del inglés *Not a Number*) magnitud no numérica que indica que se han hecho cálculos indefinidos.

Ejemplos 2.16

```
>> pi % para ver el valor de pi con 15 cifras,
ans =
3.1416
>> (2+3i)*(1-i) % Producto de números complejos
ans =
5.0000 + 1.0000i
>> 2^2000 % Número no almacenable en el ordenador
ans =
Inf
>> 0/0 % Número imposible de calcular
ans =
NaN
```

2.6 Funciones matemáticas elementales

Los nombres de las funciones elementales son bastante “habituales”. Los argumentos pueden ser, siempre que tenga sentido, reales o complejos y el resultado se devuelve en el mismo tipo del argumento.

La lista de todas las funciones matemáticas elementales se puede consultar en:

Help → MATLAB → Mathematics → Elementary Math

Algunas de las más habituales se muestran en la tabla 2.2:

Algunas funciones matemáticas elementales			
<code>sqrt(x)</code>	raíz cuadrada	<code>sin(x)</code>	seno (radianes)
<code>abs(x)</code>	módulo	<code>cos(x)</code>	coseno (radianes)
<code>conj(z)</code>	complejo conjugado	<code>tan(z)</code>	tangente (radianes)
<code>real(z)</code>	parte real	<code>cotg(x)</code>	cotangente (radianes)
<code>imag(z)</code>	parte imaginaria	<code>asin(x)</code>	arcoseno
<code>exp(x)</code>	exponencial	<code>acos(x)</code>	arcocoseno
<code>log(x)</code>	logaritmo natural	<code>atan(x)</code>	arcotangente

Algunas funciones matemáticas elementales			
<code>log10(x)</code>	logaritmo decimal	<code>cosh(x)</code>	cos. hiperbólico
<code>rat(x)</code>	aprox. racional	<code>sinh(x)</code>	seno hiperbólico
<code>fix(x)</code>	redondeo hacia 0	<code>tanh(x)</code>	tangente hiperbólica
<code>ceil(x)</code>	redondeo hacia $+\infty$	<code>acosh(x)</code>	arcocoseno hiperb.
<code>floor(x)</code>	redondeo hacia $-\infty$	<code>asinh(x)</code>	arcoseno hiperb.
<code>round(x)</code>	redondeo al entero más próximo	<code>atanh(x)</code>	arcotangente hiperb.
<code>mod(x,y)</code>	resto de dividir x por y. Iguales si $x, y > 0$.		
<code>rem(x,y)</code>	Ver help para definición exacta		

Tabla 2.2: Algunas funciones matemáticas elementales

Ejemplos 2.17

```
>> sqrt(34*exp(2))/(cos(23.7)+12)
ans =
    1.3058717

>> 7*exp(5/4)+3.54
ans =
    27.97240

>> exp(1+3i)
ans =
   - 2.6910786 + 0.3836040i
```

2.7 Operaciones de comparación

Imprescindibles para verificar **condiciones** son las **expresiones lógicas**, es decir, expresiones cuya evaluación produce un **valor lógico**. Las más simples son aquéllas en las que se comparan dos datos. Los **operadores de comparación** actúan entre dos datos, que tienen que ser del mismo tipo, y producen un resultado lógico: **true** o **false** (**verdadero** o **falso**). En MATLAB se muestran como **1** y **0** respectivamente.

Los operadores de comparación se representan de distintas formas según el lenguaje. Los que se muestran en la tabla siguiente son los de MATLAB:

Descripción	Símbolo
Igual a	$==$
No igual a	$\sim=$
Menor que	$<$
Mayor que	$>$
Menor o igual que	\leq
Mayor o igual que	\geq

Tabla 2.3: Operadores de comparación

Ejemplos 2.18

```
>> 3<6
ans =
1  (logical) (true)

>> 0>=1
ans =
0  (logical) (false)

>> 'A'=='B'
ans =
0  (logical) (false)
```

Aunque MATLAB muestre los valores lógicos como **0** y **1**, no se deben confundir con los correspondientes valores numéricos. Así por ejemplo, el intento de calcular **sin(3<4)** dará error ya que la función **sin** no admite valores lógicos como argumento de entrada.

Sin embargo, cuando aparecen valores lógicos en expresiones aritméticas junto con datos numéricos, MATLAB transforma el valor lógico **true** en el valor numérico **1** y el valor lógico **false** en el valor numérico **0** antes de realizar la operación.

Ejemplos 2.19

```
>> (3<6) + 2
ans =
3

>> (0>=1) * pi
ans =
0
```

2.8 Operadores lógicos

Son los que actúan entre dos operandos de tipo lógico. Permiten construir expresiones que representen condiciones más complicadas, como que se verifiquen varias condiciones a la vez, que se verifique una entre varias condiciones, etc.

La representación de los operadores lógicos varía bastante de un lenguaje a otro. En estas notas se representarán como en MATLAB:

Descripción	Símbolo
Negación	\sim
Conjunción	$\&$
Disyunción	$ $

Tabla 2.4: Operadores lógicos

El primero de ellos, el operador de negación lógica \sim , es un operador unario, es decir, actúa sobre un solo operando lógico, dando como resultado el opuesto de su valor, como muestra la siguiente tabla:

A	$\sim A$
true	false
false	true

Tabla 2.5: Resultados del operador \sim

Ejemplos 2.20

```
>> ~(6 > 10)
ans =
    1    (logical)
>> ~( 'b' > 'a')
ans =
    0    (logical)
```

Los otros dos operadores lógicos actúan siempre entre dos operandos. Los resultados de su aplicación se muestran en la tabla 2.6.

Ejemplos 2.21

```
>> (1 > 5) & (5 < 10)
ans =
    0    (logical)
>> (0 < 5) | (0 > 5)
ans =
```

A	B	A & B	A B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Tabla 2.6: Resultados de los operadores & y |.

```
1 (logical)
```

Cuando aparecen valores numéricos en una operación lógica, MATLAB transforma el valor numérico 0 en el valor lógico **false** y cualquier otro (distinto de 0) en **true**.

Ejemplos 2.22

```
>> 3 & true
ans =
    1 (logical)
>> 0 & true
ans =
    0 (logical)
```

En una expresión pueden aparecer varios operadores lógicos. En ese caso, el orden en que se evalúan, es el siguiente:

1. La negación lógica **~**
2. La conjunción y disyunción **&** y **|** (de izquierda a derecha en caso de que haya varios).

Ejemplo 2.23

```
>> ~(5 > 0) & (5 < 4)
ans =
    0 (logical)
```

2.9 Orden general de evaluación de expresiones

En una expresión general pueden aparecer operadores de tipo aritmético, de comparación y lógicos, así como funciones. El orden de evaluación es el que sigue:

- Si en una expresión hay paréntesis, lo primero que se evalúa es su contenido. Si hay paréntesis anidados, se comienza por los más internos. Si hay varios grupos de paréntesis disjuntos, se comienza por el que esté más a la izquierda.
- En una expresión sin paréntesis de agrupamiento, el orden de evaluación es el siguiente:
 1. Las funciones. Si el argumento de la función es una expresión, se le aplican estas reglas. Si hay varias funciones, se comienza por la de la izquierda.
 2. Los operadores aritméticos, en el orden ya indicado.
 3. Los operadores de comparación.
 4. Los operadores lógicos, en el orden antes mencionado.

Ejemplos 2.24

```
>> ~ 5 > 0 & 4 < 4 + 1
ans =
0    (logical)
>> -5 < -2 & -2 < 1
ans =
1    (logical)
>> -5 < -2 < 1
ans =
0    (logical)
```

2.10 Matrices

Las matrices bidimensionales de números reales o complejos son los objetos básicos con los que trabaja MATLAB. Los vectores y escalares son casos particulares de matrices.

2.10.1 Construcción de matrices

La forma más elemental de introducir matrices en MATLAB es describir sus elementos de forma exhaustiva (por filas y entre corchetes rectos []) : elementos de una fila se separan unos de otros por comas y una fila de la siguiente por punto y coma.

Ejemplos 2.25 (Construcciones elementales de matrices)

```
>> v = [1, -1, 0, sin(2.88)] % vector fila longitud 4
>> w = [0; 1.003; 2; 3; 4; 5*pi] % vector columna longitud 6
>> a = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12] % matriz 3x4
```

Observación.

El hecho de que, al introducirlas, se escriban las matrices por filas no significa que internamente, en la memoria del ordenador, estén así organizadas: **en la memoria las matrices se almacenan como un vector unidimensional ordenadas por columnas.**

Ejemplo 2.26 (Almacenamiento de una matriz en la memoria)

La matriz definida por

```
>> A = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]
```

se almacena en la memoria del ordenador como:

1	5	9	2	6	10	3	7	11	4	8	12
---	---	---	---	---	----	---	---	----	---	---	----

Los siguientes operadores y funciones sirven para generar vectores con elementos regularmente espaciados (útiles en muchas ocasiones) y para trasponer matrices.

Generación de vectores regularmente espaciados

a:h:b	Genera un vector fila cuyos elementos van desde a hasta un número c≤b en incrementos de h .
a:b	Lo mismo que el anterior, pero con h=1 .
linspace(a,b,n)	Si a y b son números reales y n un número entero, genera un vector fila de longitud n cuyo primer elemento es a , el último es b y cuyos elementos están regularmente espaciados, es decir, una partición regular del intervalo [a,b] con n nodos y n-1 subintervalos.
linspace(a,b)	Como el anterior, con n=100 (vector fila de longitud 100)

Tabla 2.7: Operadores y funciones para generar vectores regularmente espaciados.

Ejemplos 2.27 (Uso del operador : y de linspace)

```
>> v1 = 1:2:10
v1 =
    1   3   5   7   9
% obsérvese que el último elemento es 9 < 10
>> v2 = 2:5
v2 =
    2   3   4   5
>> v3 = 6:-1:0
v3 =
    6   5   4   3   2   1   0
>> v4 = 6:0
v4 =
    []  (vector fila vacío)
>> linspace(1, 10, 5)
ans =
    1     3.25    5.5    7.75    10
>> linspace(10, 1, 5)
ans =
    10     7.75    5.5     3.25     1
```

Ejemplos 2.28 (Matrices traspuestas)

```
>> A = [1, 2; 3, 4];
>> A'
ans =
    1     3
    2     4
>> B = [1+2i, 3+4i; 2-i, 4-i];
>> B'
ans =
    1 - 2i    2 + i
    3 - i    4 + i
>> B.'
ans =
    1 + 2i    2 - i
    3 + 4i    4 - i
>> (1:4)'
ans =
    1
    2
    3
    4
```

Se pueden también utilizar los vectores/matrices como objetos (bloques) para construir otras

matrices.

Ejemplos 2.29 (Matrices construidas con bloques)

```
>> v1 = 1:4;

>> v2 = [v1, 5; 0.1:0.1:0.5]

>> v3 = [v2', [11,12,13,14,15]']
```

$$v_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \quad v_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 0.1 & 0.2 & 0.3 & 0.4 & 0.5 \end{pmatrix} \quad v_3 = \begin{pmatrix} 1 & 0.1 & 11 \\ 2 & 0.2 & 12 \\ 3 & 0.3 & 13 \\ 4 & 0.4 & 14 \\ 4 & 0.5 & 15 \end{pmatrix}$$

Las siguientes funciones generan algunas matrices especiales que serán de utilidad.

Generación de algunas matrices habituales

<code>zeros(n,m)</code>	matriz $n \times m$ con todas sus componentes iguales a cero
<code>ones(n,m)</code>	matriz $n \times m$ con todas sus componentes iguales a uno
<code>eye(n,m)</code>	matriz unidad $n \times m$: diagonal principal = 1 y el resto de las componentes = 0
<code>diag(v)</code>	Si v es un vector, es una matriz cuadrada de ceros con diagonal principal = v
<code>diag(v, k)</code>	Si v es un vector, es una matriz cuadrada de ceros con k -ésima superdiagonal = v (si $k < 0$, la k -ésima subdiagonal)
<code>diag(A)</code>	Si A es una matriz, es su diagonal principal
<code>diag(A, k)</code>	Si A es una matriz, es su k -ésima superdiagonal (resp. subdiagonal si $k < 0$)

Tabla 2.8: Algunas funciones para generación de matrices habituales

MATLAB posee, además, decenas de funciones útiles para generar distintos tipos de matrices. Para ver una lista exhaustiva consultar:

[Help → MATLAB → Functions: By Category → Language Fundamentals → Arrays and Matrices](#)

2.10.2 Acceso a los elementos de vectores y matrices

El acceso a elementos individuales de una matriz se hace indicando entre paréntesis sus índices de fila y columna. Se puede así utilizar o modificar su valor

Ejemplos 2.30 (Acceso a elementos individuales de una matriz)

```
>> A = [1, 2, 3; 6, 3, 1; -1, 0, 0; 2, 0, 4]
A =
    1     2     3
    6     3     1
   -1     0     0
    2     0     4
>> A(2, 3)
ans =
    1
>> A(4, 2) = 11
A =
    1     2     3
    6     3     1
   -1     0     0
    2    11     4
```

Se puede acceder al último elemento de un vector mediante el *índice simbólico end*:

Ejemplos 2.31 (Último índice de un vector)

```
>> w = [5, 3, 1, 7, 3, 0, -2];
>> w(end)
ans =
    -2
>> w(end-1) = 101
w =
    5     3     1     7     3    101     -2
```

Se puede acceder a una fila o columna completa sustituyendo el índice correspondiente por **:** (dos puntos)

Ejemplos 2.32 (Acceso a filas o columnas completas)

```
>> A = [1, 2, 3; 6, 3, 1; -1, 0, 0; 2, 0, 4];
>> A(:, 3)
ans =
    3
    1
    0
    4
```

```
>> A(2, :)
ans =
    6     3     1
>> A(3, :) = [5, 5, 5]
A =
    1     2     3
    6     3     1
    5     5     5
    2     0     4
```

También se puede acceder a una “submatriz” especificando los valores de las filas y columnas que la definen:

Ejemplos 2.33 (Acceso a “submatrices”)

```
>> w = [5, 3, 1, 7, 3, 0, -2];
>> w([1,2,3])           % o lo que es lo mismo w(1:3)
ans =
    5     3     1
>> A = [1, 2, 3; 6, 3, 1; -1, 0, 0; 2, 0, 4];
>> A([1,2], :) = zeros(2, 3)
A =
    0     0     0
    0     0     0
   -1     0     0
    2     0     4
>> B = A(3:4, [1,3])
B =
   -1     0
    2     4
>> C = A(:)            % con esta opción se obtiene un vector
ans =                   % columna con todos los elementos de A
    1                   % en el orden en que están almacenados
    6                   % en la memoria del ordenador
   -1
    2
    2
    3
    0
    0
    3
    1
    0
    4
```

Otra forma de acceder a elementos específicos de una matriz es utilizando vectores de valores

lógicos en lugar de subíndices, como se muestra en el siguiente ejemplo (más adelante se verán ejemplos más interesantes).

Ejemplos 2.34 (Acceso a elementos de una matriz usando valores lógicos)

```
>> w = [5, 3, 1, 7, 3, 0, -2];
>> bool = [true false true true false false];
>> w(bool) % se obtienen los elementos de w correspondientes
ans = % a los elementos de bool que valen true
      5     1     7
>> bool = [false true true false false];
>> w(bool) % si la longitud de bool es menor que la de w,
ans = % se aplica la regla anterior hasta que
      3     1 % se termine bool
```

2.10.3 Operaciones con vectores y matrices

Los operadores aritméticos $+$ $-$ $*$ $/$ y $^{\wedge}$ representan las correspondientes operaciones matriciales siempre que tengan sentido.

Operaciones aritméticas con matrices

A+B, A-B	matrices de elementos respectivos $a_{ij} + b_{ij}, \quad a_{ij} - b_{ij}$ (si las dimensiones son iguales)
A+k, A-k	matrices de elementos respectivos $a_{ij} + k, \quad a_{ij} - k.$
k*A, A/k	matrices de elementos respectivos $\frac{1}{k} a_{ij}, \quad \frac{1}{k} a_{ij}$
A*B	producto matricial de A y B (si las dimensiones son compatibles)
A^n	Si n es un entero positivo, A*A*...*A

Tabla 2.9: Operaciones aritméticas con matrices

Además de estos operadores, MATLAB dispone de ciertos operadores aritméticos que operan **elemento a elemento**. Son los operadores $.*$ $./$ y $.^{\wedge}$, muy útiles para aprovechar las funcionalidades vectoriales de MATLAB. Véase la tabla 2.10.

Operaciones elemento a elemento

Aquí, **A** y **B** son dos matrices de las mismas dimensiones y de elementos respectivos a_{ij} y b_{ij} y **k** es un escalar.

A.*B	matriz de la misma dimensión que A y B de elementos $a_{ij} \times b_{ij}$
A./B	matriz de la misma dimensión que A y B de elementos a_{ij}/b_{ij}
A.^B	matriz de la misma dimensión que A y B de elementos $a_{ij}^{b_{ij}}$
k./A	matriz de la misma dimensión que A , de elementos k/a_{ij}
A.^k	matriz de la misma dimensión que A , de elementos a_{ij}^k
k.^A	matriz de la misma dimensión que A , de elementos $k^{a_{ij}}$

Tabla 2.10: Operaciones con matrices **elemento a elemento**

Por otra parte, la mayoría de las funciones MATLAB están hechas de forma que admiten matrices como argumentos. Esto se aplica en particular a las funciones matemáticas elementales y su utilización debe entenderse en el sentido de **elemento a elemento**. Por ejemplo, si **A** es una matriz de elementos a_{ij} , **exp(A)** es otra matriz con las mismas dimensiones que **A**, cuyos elementos son $e^{a_{ij}}$.

Algunas otras funciones útiles en cálculos matriciales son:

Algunas funciones para cálculos matriciales

size(A)	devuelve un vector fila de dimensión 2, cuyo primer elemento es el número de filas de la matriz A y el segundo es su número de columnas.
[n, m] =size(A)	devuelve el número de filas y columnas de A , pero en variables separadas: n es el número de filas y m es el número de columnas.
size(A, k)	devuelve el número de filas si k=1 , y el número de columnas si k=2 .
length(v)	si v es un vector, es su longitud; si v es una matriz, es la mayor de sus dimensiones.
numel(v)	es el número de elementos de v : si v es un vector, es su longitud; si v es una matriz, es el producto de sus dimensiones.

Algunas funciones para cálculos matriciales

<code>sum(v)</code>	suma de los elementos del vector <code>v</code>
<code>sum(A)</code>	suma de los elementos de la matriz <code>A</code> , por columnas
<code>sum(A,1)</code>	
<code>sum(A,2)</code>	suma de los elementos de la matriz <code>A</code> , por filas
<code>prod(v), prod(A)</code>	como la suma, pero para el producto
<code>prod(A,1)</code>	
<code>prod(A,2)</code>	
<code>max(v), min(v)</code>	máximo/mínimo elemento del vector <code>v</code>
<code>max(A), min(A)</code>	máximo/mínimo elemento de la matriz <code>A</code> , por columnas
<code>mean(v)</code>	promedio de los elementos del vector <code>v</code>
<code>mean(A)</code>	promedio de los elementos de la matriz <code>A</code> , por columnas.
<code>mean(A)</code>	promedio de los elementos de la matriz <code>A</code> , por columnas.
<code>norm(v)</code>	norma euclídea del vector <code>v</code> , esto es
<code>norm(v, 2)</code>	$\ v\ _2 = \left(\sum_i v_i ^2 \right)^{1/2}$
<code>norm(v, p)</code>	norma p del vector <code>v</code> , esto es
	$\ v\ _p = \left(\sum_i v_i ^p \right)^{1/p}$
<code>norm(v, Inf)</code>	norma del máximo del vector <code>v</code> , esto es
	$\ v\ _\infty = \max_i v_i $

Tabla 2.11: Funciones para cálculos matriciales

2.10.4 Operaciones de comparación y lógicas con matrices

Las operaciones de comparación entre una matriz y un escalar dan como resultado una matriz de valores lógicos de la misma dimensión resultado de aplicar la comparación a cada elemento de la matriz. Si la comparación se hace entre dos matrices de la misma dimensión, el resultado se obtiene elemento a elemento.

Ejemplos 2.35 (Operaciones de comparación con matrices)

```
>> w = [5, 3, 1, 7, 3, 0, -2];
>> w <= 3 % elementos de w que son menores que 3
ans =
0 1 1 0 1 1 (logical)
```

```

>> bool = w <= 0 ;           % si se guarda el resultado de la comparación
>> w(bool)                 % en una variable, ésta se puede utilizar para
ans =                         % obtener los elementos de w que la verifican
    0   -2                   % w(w<=0) da el mismo resultado

>> v = [1, 3, -4, 3, 1, 0, 2];
>> w == v
ans =
    0   1   0   0   0   1   0

>> A = [1, 2; 3, 4];
>> bool = A < 4 & A > 1
bool =
    0   1     (logical)
    1   0
>> A(bool)
ans =
    3
    2

>> A( mod(A,2)==0 )        % elementos de A que son múltiplos de 2
ans =
    2
    4

```

Algunas funciones lógicas para matrices

find(v)	si v es un vector, devuelve un vector (de la misma dimensión que v) conteniendo los índices de los elementos de v que sean distintos de cero (o que sean true , si el vector es lógico).
find(A)	si A es una matriz, devuelve un vector columna conteniendo los índices de los elementos de A que sean distintos de cero (o que sean true , si el vector es lógico). Estos índices están referidos al orden unidimensional en que se almacena A en la memoria (ver Ejemplo 2.26).
[fil, col] = find(A)	si A es una matriz, devuelve dos vectores conteniendo los índices de las filas y de las columnas de los elementos de A que sean distintos de cero (o que sean true , si el vector es lógico).
[fil, col, v] = find(A)	como el anterior, pero devuelve también los valores de los elementos no nulos.

Algunas funciones lógicas para matrices

<code>all(v)</code>	si <code>v</code> es un vector, devuelve <code>true</code> si todos los elementos de <code>v</code> son distintos de cero y devuelve <code>false</code> en caso contrario (si alguno de los elementos de <code>v</code> es cero).
<code>all(A)</code> <code>all(A, 1)</code>	si <code>A</code> es una matriz, hace lo mismo que el anterior, pero por columnas; esto es, devuelve un vector fila lógico cada uno de cuyos elementos corresponde a una de las columnas de <code>A</code> .
<code>all(A, 2)</code>	lo mismo que el anterior, pero por filas; devuelve un vector columna lógico.
<code>any(v)</code> <code>any(A)</code> <code>any(A, 1)</code> <code>any(A, 2)</code>	similares a las correspondientes utilizaciones de la función <code>all</code> , pero devuelve <code>true</code> si algún elemento del vector/fila/columna es distinto de cero (o es <code>true</code>) y devuelve <code>false</code> en caso contrario, es decir, si ningún elemento es distinto de cero (o <code>true</code>).

Tabla 2.12: Funciones lógicas para matrices

Ejemplos 2.36 (Uso de la función `find`)

```

>> w = [0.27, 0.20, 0.56, 0.64, 0.41, 0.19, 0.94, 0.08, 0.11, 0.14];
>> find(w < 0.5)
ans =
    1     2     5     6     8     9    10
>> ind = find(w < 0.5); % si se guardan los valores de los índices,
>> w(ind)                % se pueden recuperar los valores de los
ans =                      % elementos de w que cumplen la condición
    0.27   0.20   0.41   0.19   0.08  0.11   0.14

>> bool = w <= 0 ;        % si se guarda el resultado de la comparación
>> w(bool)               % en una variable, ésta se puede utilizar para
ans =                      % obtener los elementos de w que la verifican
    0   -2                  % w(w<=0) da el mismo resultado

>> A = [18 3 1 11; 8 10 11 3; 9 14 6 1; 4 3 15 21]
A =
    18     3     1    11
     8    10    11     3
     9    14     6     1
     4     3    15    21
>> find(A<10)
ans =
    2
    3
    4

```

```
5  
8  
9  
11  
14  
15
```

Ejemplos 2.37 (Uso de las funciones all y any)

```
>> w = [0.27, 0.20, 0.56, 0.64, 0.41, 0.19, 0.94, 0.08, 0.11, 0.14];  
>> all(w > 0)  
ans =  
    1      (logical)  
  
>> any(w > 0.9)  
ans =  
    1      (logical)  
  
>> A = [18 3 1 11; 8 10 11 3; 9 14 6 1; 4 3 15 21]  
A =  
    18      3      1      11  
     8      10     11      3  
     9      14      6      1  
     4      3      15     21  
>> all(A>2)  
ans =  
    1      1      0      0      (logical)  
>> any(A<=3)  
ans =  
    0      1      1      1      (logical)
```

2.11 Ejercicios

1. Comprobar el valor de las siguientes expresiones:

$$A = \frac{42.1 + 455}{2^{10} - 13} = 0.4917$$

$$C = \frac{\ln(123.5 \times 10^{11})}{\sqrt{2 \times 10^5 - 1}} = 0.0674$$

$$E = 2^{\frac{3+\pi}{5}} = 2.3429$$

$$G = \left(\frac{\cos(12) + 1}{\ln(8)} \right)^{1/3} = 0.9607$$

$$I = \frac{e + 45.1^3}{2/3 - (1.56 + 4)^2} = -3.0329e + 003$$

$$K = \ln \left(\frac{\sqrt[3]{3.67 \times \pi^2}}{e + \cos^2(77^\circ 33')} \right) = 0.1796$$

$$M = \sqrt[4]{\frac{\pi}{\frac{2}{e} + \sin^2(17^\circ)}} = 1.3985$$

$$B = \frac{\ln(23.8)}{\sin(6)} = -11.3440$$

$$D = \frac{\sqrt{34 \times e^2}}{\cos(23.7) + 12} = 1.3059$$

$$F = \sqrt[7]{\frac{3 \times \cos(32^\circ 15')}{42.1^3}} = 0.2300$$

$$H = \left(\frac{\tan(\pi/3) + 2}{\exp(3)} \right)^{-1/2} = 2.3199$$

$$J = \frac{4 \ln^2(12) + 6}{\sqrt{e + 3}} = 12.8378$$

$$L = \tan \left(\frac{4 \ln^2(12) + 6}{\frac{1}{e} + \cos^2(57^\circ)} \right) = -1.3296$$

$$N = \frac{\tan^2 \left(\frac{\pi}{\sqrt[3]{5}} \right)}{\exp(-0.66)} = 25.9782$$

2. Definir la matriz $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ y crear, a partir de A , las siguientes matrices:

$$B = \begin{pmatrix} 4 & 5 \\ 7 & 8 \end{pmatrix}, \quad C = (7, 8, 9), \quad D = \begin{pmatrix} 2 & 3 \\ 5 & 6 \\ 8 & 9 \end{pmatrix}, \quad E = \begin{pmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{pmatrix}$$

$$F = (1, 4, 7, 2, 5, 8, 3, 6, 9), \quad G = \begin{pmatrix} 1 & 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 & 1 \\ 7 & 8 & 9 & 0 & 0 \end{pmatrix}, \quad H = \begin{pmatrix} 1 & 2 & 3 & 4 & 0 \\ 4 & 5 & 6 & 4 & 1 \\ 7 & 8 & 9 & 4 & 0 \end{pmatrix}$$

3. Construir las siguientes matrices (sin describirlas expresamente):

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{pmatrix}$$

4. Generar los vectores que se indican utilizando los operadores adecuados de MATLAB (esto es, sin escribir expresamente todas sus componentes):

a) $v1 = (1, 3, 5, \dots, 25)$

c) $v3 = (0, 0.1, 0.2, \dots, 1)$

b) $v2 = (\pi, 2\pi, 3\pi, \dots, 10\pi)$

d) $v4 = (-10, -9, -8, \dots, -1, 0)$

5. Generar los vectores que se indican, de longitud n variable. Esto es, escribir las órdenes adecuadas para construir los vectores que se indican de forma que puedan servir para cualquier valor de n .

- a) $w1 = (1, 3, 5, 7, \dots)$
- b) $w2 = (\pi, 2\pi, 3\pi, 4\pi \dots)$
- c) $w3 = (n, n - 1, n - 2, \dots, 2, 1)$
- d) $w4 = (2n, 2n - 2, \dots, 4, 2)$
- e) $w5 = (0, 2, 2, \dots, 2, 2, 0)$
- f) $w6 = (1, 3, \dots, n - 1, n, n - 2, \dots, 4, 2)$
(suponiendo, aquí, que n es par)

6. Generar un vector x con 30 componentes regularmente espaciadas entre 0 y π . Evaluar en x cada una de las funciones siguientes, esto es, calcular, para cada una de las funciones indicadas, un nuevo vector y tal que $y_i = f(x_i)$:

$$\begin{array}{ll} a) f(x) = \frac{\cos(x/4)}{\ln(2 + x^2)} & c) f(x) = \frac{x}{\ln(2 + x)} \\ b) f(x) = e^{-x^2+2} \operatorname{sen}(x/2) & d) f(x) = \frac{1}{e^x} + x^2 - x \end{array}$$

7. Utilizando las funciones de MATLAB de generación de matrices y vectores, crear las siguientes matrices de dimensión $N \times N$ variable. Esto es, generar las matrices que se indican de forma que las órdenes utilizadas puedan servir para cualquier valor de N :

$$A = \begin{pmatrix} 4 & 0 & \cdots & 0 & 0 \\ 0 & 2 & & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & & 2 & 0 \\ 0 & 0 & \cdots & 0 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & \pi & \cdots & 0 & 0 \\ 0 & 1 & & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & & 1 & \pi \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} -N & -N + 1 & \cdots & -2 & -1 \\ 1 & 0 & & 0 & 0 \\ 0 & 1 & \ddots & 0 & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 1 & 0 \end{pmatrix}$$

8. La función `rand(n, m)` devuelve una matriz $n \times m$ de números aleatorios de distribución uniforme entre 0 y 1 (esto es, que todos los números entre 0 y 1 tienen la misma probabilidad). Construir un vector-fila v de números aleatorios de dimensión par $N \geq 10$. Extraer de él los siguientes conjuntos de elementos:

- a) $w1 = (v_1, v_3, v_5, \dots, v_{N-1})$
- b) $w2 = (v_N, v_{N-1}, \dots, v_2, v_1)$
- c) $w3 = (v_1, v_3, v_5, \dots, v_{N-1}, v_2, v_4, \dots, v_N)$
- d) $w4 = (v_1, v_2, v_3, \dots, v_{\frac{N}{2}}, v_N, v_{N-1}, \dots, v_{\frac{N}{2}+1})$

9. Utilizando la función `rand`, construir vectores-fila de números aleatorios con cada una de las propiedades siguientes:

- a) Sus componentes estén entre 0 y 10.
- b) Ídem entre 10 y 20
- c) Ídem entre 2 y 10
- d) Ídem entre -5 y 5

10. La función `randi` se puede utilizar para obtener matrices de números aleatorios **enteros** (ver en el Help). Construir un vector-fila de números **enteros** aleatorios entre 1 y 6.
11. Generar un vector v de 20 números aleatorios entre -5 y 5.

- a) Obtener los subíndices de las componentes positivas de v .
 - b) Extraer las componentes negativas de v .
 - c) Extraer las componentes de v que tomen valores entre -2 y 2 (inclusive).
12. Generar un vector v de 20 números enteros aleatorios entre 1 y 15. Extraer las componentes de v que sean iguales a 5 ó a 10.
13. Generar un vector v de números aleatorios de longitud 20. Buscando en la ayuda en línea de MATLAB (**help** o **helpwin**) información sobre la función que se indica en cada caso, calcular
- a) La norma euclídea del vector v (función **norm**)
 - b) La suma de las componentes del vector v (función **sum**)
 - c) El producto de los cosenos de las componentes del vector v . (función **prod**)
14. Siendo v un vector cualquiera de números enteros, escribir las órdenes adecuadas para obtener lo que se pide en cada uno de los casos siguientes:
- a) Extraer los elementos pares de v .
 - b) Calcular la suma de las componentes de v que sean mayores o iguales que 3.
 - c) El producto de las componentes de v que sean positivas y múltiplos de 3
15. La función **magic** de MATLAB construye una matriz cuadrada conteniendo un cuadrado mágico (ver **Help**). Dado un número $n > 3$ almacenado en una variable, construir una matriz M que contenga un cuadrado mágico de dimensión n .
- a) Comprobar mediante la función **sum** que todas las filas y columnas de M tienen la misma suma.
 - b) Calcular el rango, el determinante y la traza de M (buscar en el **Help** las funciones adecuadas).
 - c) Resolver el sistema lineal de ecuaciones $Mx = b$, siendo $b = (1, 2 \dots n)^t$. (Observación: la matriz mágica construida es regular para $n > 3$ impar, y es singular para n par).

3

Gráficos



3.1 Generalidades sobre la representación gráfica de funciones

La representación gráfica de una curva en un ordenador es una linea poligonal construida uniendo mediante segmentos rectos un conjunto discreto y ordenado de puntos: $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$.

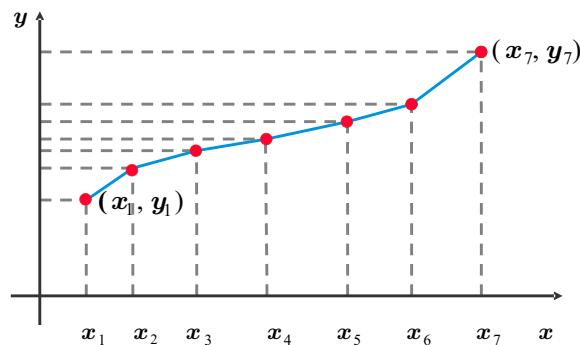


Figura 3.1: Línea poligonal determinada por un conjunto de puntos.

La línea así obtenida tendrá mayor apariencia de “suave” cuanto más puntos se utilicen para construirla, ya que los segmentos serán imperceptibles (véase la Figura 3.2).

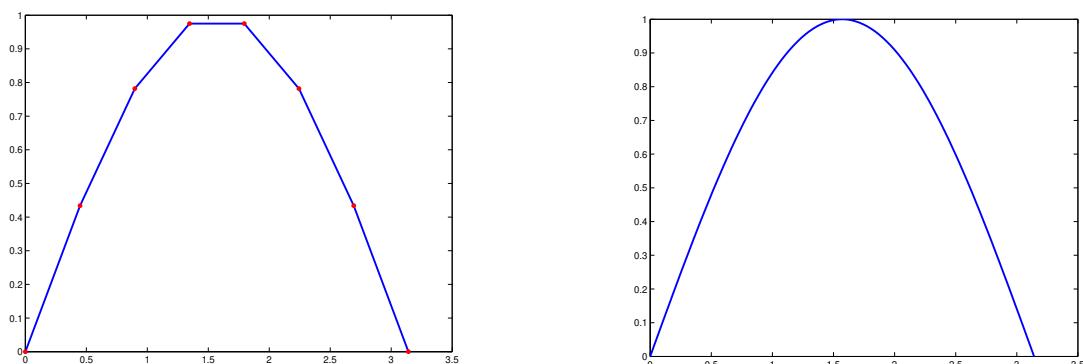


Figura 3.2: Representación de $y = \sin(x)$ en $[0, \pi]$ con 8 y con 100 puntos.

La representación gráfica de objetos (curvas, superficies, ...) tridimensionales presenta un grado

mucho más grande de dificultad. Por un lado, es preciso utilizar técnicas de geometría proyectiva para determinar la perspectiva y conseguir impresión de tridimensionalidad. Por otro, aparece la necesidad de utilizar algoritmos y técnicas complejas para determinar partes ocultas. Y, aún más, iluminación, transparencias, aplicación de texturas, etc. Todo ello queda fuera del ámbito de este curso.

En esta sección se explican, muy brevemente, las formas más habituales de representación gráfica de “objetos” matemáticos bi y tri-dimensionales.

3.1.1 Representación gráfica de funciones de una variable real

La relación $y = f(x)$, donde $f : [a, b] \mapsto \mathbb{R}$ es una función de una variable real, se puede representar gráficamente mediante una curva plana.

La construcción de dicha gráfica en un ordenador básicamente sigue los siguientes pasos (ver la Figura 3.1):

- Construir un conjunto de puntos (tantos como se quiera) en el intervalo $[a, b]$, que serán las abscisas de los puntos que determinan la poligonal a construir. Normalmente, dichos puntos se toman regularmente espaciados y en número suficiente como para que la gráfica tenga aspecto “suave”:

$$a = x_1 < x_2 < \dots < x_n = b$$

- Calcular los valores de la función f en los puntos anteriores:

$$y_1 = f(x_1), y_2 = f(x_2), \dots, y_n = f(x_n)$$

- Unir los puntos (x_i, y_i) consecutivos mediante segmentos rectos.

Cuando una curva viene definida por una relación del tipo $y = f(x)$ se dice que está definida de forma explícita.

En ocasiones, una curva viene descrita por una relación, también explícita, pero del tipo:

$$x = g(y), \quad y \in [a, b].$$

Entonces será necesario construir en primer lugar el conjunto de “ordenadas”

$$a = y_1 < y_2 < \dots < y_n = b$$

y luego calcular las abscisas, como los valores de la función g :

$$x_1 = g(y_1), x_2 = g(y_2), \dots, x_n = g(y_n).$$

Una relación del tipo $f(x, y) = 0$ puede también representar, implícitamente, una curva: la formada por los puntos (x, y) del plano sobre los cuales la función f toma el valor cero. Se puede dibujar esta curva dibujando la curva de nivel $k = 0$ de la función f (ver Sección 3.1.7).

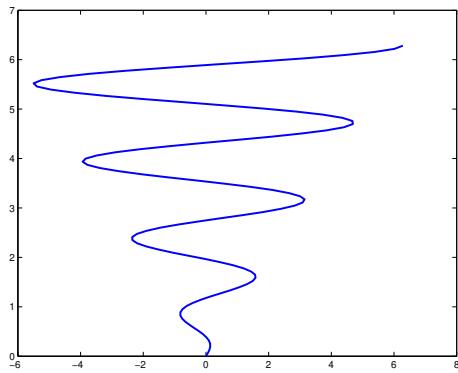


Figura 3.3: Curva definida por la relación $x = y \cos(4y)$, $y \in [0, 2\pi]$.

3.1.2 Curvas planas definidas mediante ecuaciones paramétricas

Otra forma de definir una curva plana es mediante sus **ecuaciones paramétricas**, en la cual los puntos (x, y) que forman la curva vienen dados por dos funciones que dependen de una variable auxiliar:

$$x = f(t), \quad y = g(t), \quad t \in [a, b].$$

La variable t se suele llamar el **parámetro de la curva**.

Para construir la gráfica de una curva definida de esta forma es preciso (ver el ejemplo de la Figura 3.4):

- Construir un conjunto de valores del parámetro $t \in [a, b]$:

$$a = t_1 < t_2 < \dots < t_n = b$$

- Calcular los valores x y de y para dichos valores del parámetro:

$$\begin{aligned} x_1 &= f(t_1), \quad x_2 = f(t_2), \dots, \quad x_n = f(t_n) \\ y_1 &= g(t_1), \quad y_2 = g(t_2), \dots, \quad y_n = g(t_n) \end{aligned}$$

- Unir los puntos (x_i, y_i) consecutivos mediante segmentos rectos.

Mediante ecuaciones paramétricas es posible describir muchas más curvas y más complicadas que mediante una ecuación explícita. Algunas serían prácticamente imposibles de visualizar sin la ayuda de herramientas gráficas informáticas (véanse Figuras 3.5 y Figuras 3.6).

3.1.3 Curvas planas en coordenadas polares

Recordemos que en el **sistema de coordenadas polares** la posición de un punto P queda definida por dos cantidades:

- r , que es la distancia de P a un punto fijo, O , llamado **punto polo** y
- θ , que es el ángulo que forma el segmento OP con una semirrecta fija de origen O denominada **eje polar**.

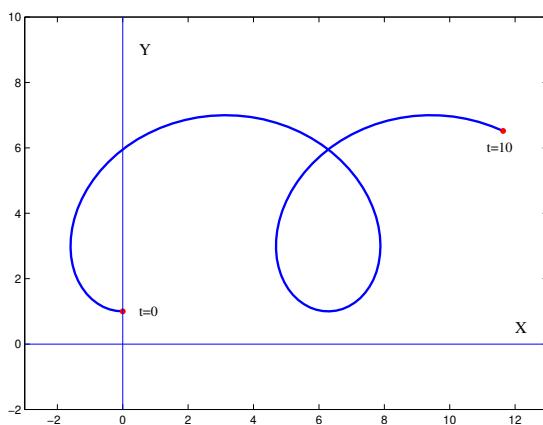


Figura 3.4: Representación de la curva de ecuaciones paramétricas $x = t - 3 \operatorname{sen}(t)$, $y = 4 - 3 \cos(t)$ para $t \in [0, 10]$. Obsérvese que no hay eje t .

t	x	y
0	0	1
1	-1.5	2.4
2	-0.7	5.2
3	2.6	7.0
4	6.3	6
5	7.9	3.1
6	6.8	1.1
7	5.0	1.7
8	5.0	4.4
9	7.8	6.7
10	11.6	6.5

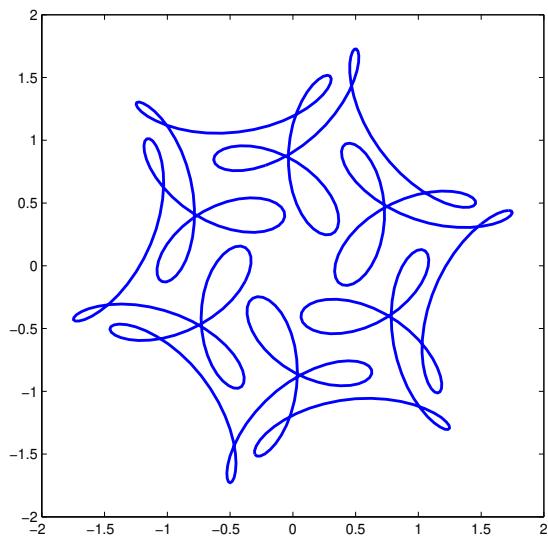


Figura 3.5: Representación de la curva de ecuaciones paramétricas $x = \cos(t) + 1/2 \cos(7t) + 1/3 \operatorname{sen}(17t)$, $y = \operatorname{sen}(t) + 1/2 \operatorname{sen}(7t) + 1/3 \cos(17t)$, para $t \in [0, 2\pi]$.

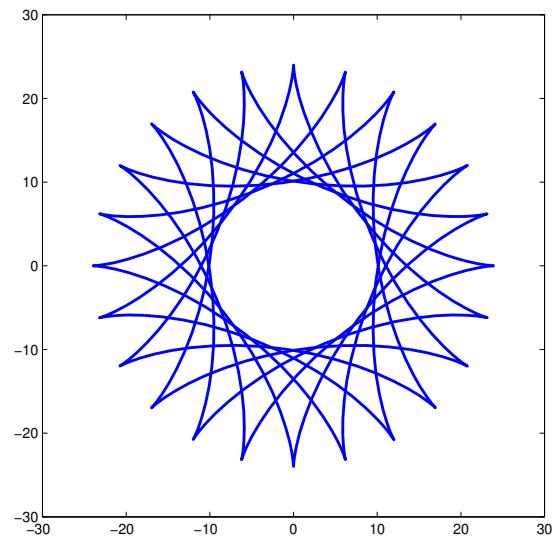


Figura 3.6: Representación de la curva de ecuaciones paramétricas $x = 17 \cos(t) + 7 \cos(\frac{17}{7}t)$, $y = 17 \operatorname{sen}(t) - 7 \operatorname{sen}(\frac{17}{7}t)$, para $t \in [0, 14\pi]$.

En tal sistema de coordenadas, se dice que el par (r, θ) son las **coordenadas polares** del punto P (ver Figura 3.7).

El paso de las coordenadas polares a cartesianas y viceversa se efectúa mediante las siguientes fórmulas, tomando el polo como origen de coordenadas y el eje polar como semi-eje positivo de

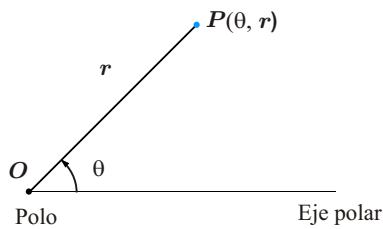


Figura 3.7: Sistema de coordenadas polares.

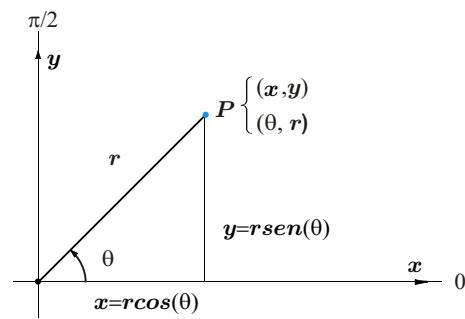


Figura 3.8: Coordenadas cartesianas y polares.

abscisas (ver la Figura 3.8):

$$x = r \cos(\theta), \quad y = r \sin(\theta);$$

$$r = \sqrt{x^2 + y^2}, \quad \theta = \arctan \frac{y}{x}$$

Una relación del tipo $r = f(\theta)$ define de forma explícita una curva en coordenadas polares. Ver ejemplos en las Figuras 3.9 y 3.10.

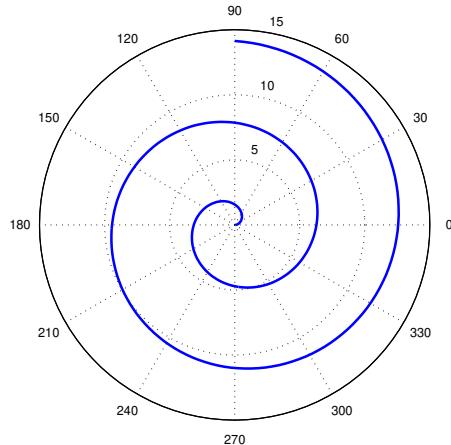


Figura 3.9: Curva de ecuación, en coordenadas polares, $r = \theta$, $\theta \in [0, 9\pi/2]$

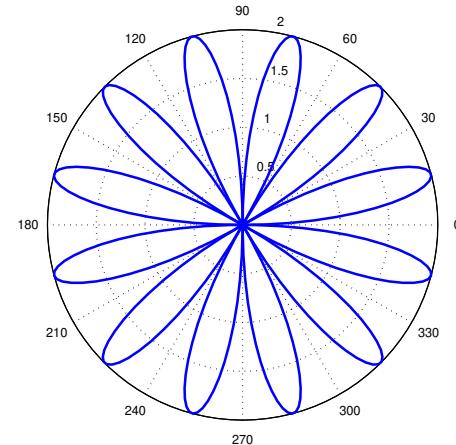


Figura 3.10: Curva de ecuación, en coordenadas polares, $r = 2 \sin(6\theta)$, $\theta \in [0, 2\pi]$.

Los programas de que permiten realizar gráficas suelen disponer de las funciones adecuadas para dibujar curvas utilizando directamente las coordenadas polares. En este caso habrá que proporcionar las coordenadas de los puntos que definen la curva:

$$\theta_1 < \theta_2 < \dots < \theta_n$$

$$r_1 = f(\theta_1), r_2 = f(\theta_2), \dots, r_n = f(\theta_n)$$

En caso de que no se disponga de dichas funciones, habrá que utilizar las fórmulas

$$x_i = r_i \cos(\theta_i), \quad y_i = r_i \sin(\theta_i)$$

para realizar la gráfica en coordenadas cartesianas.

3.1.4 Curvas en tres dimensiones

La gráfica de una curva tridimensional se dibuja, igual que la bidimensional, uniendo mediante segmentos rectos (en 3D) los puntos consecutivos de un conjunto discreto y ordenado. Mediante el software adecuado, estos segmentos se “proyectan” sobre el plano del dibujo para obtener impresión tridimensional.

La forma más sencilla de describir matemáticamente una curva tridimensional es mediante sus ecuaciones paramétricas. Estas ecuaciones describen los valores de las coordenadas (x, y, z) de cada punto de la curva en función de una variable auxiliar, llamada **parámetro**:

$$\begin{cases} x = f(t) \\ y = g(t) & \text{para } t \in [a, b] \\ z = h(t) \end{cases}$$

Para dibujar su gráfica habrá, pués, que construir las coordenadas de un conjunto discreto y ordenado de puntos de la curva. De forma similar a como se hizo en el caso bidimensional, el procedimiento es el siguiente (véanse los ejemplos de las Figuras 3.11 y 3.12):

- Construir un conjunto de valores del parámetro $t \in [a, b]$: $\{a = t_1 < t_2 < \dots < t_n = b\}$
- Calcular los valores de x , de y y de z para dichos valores del parámetro:

$$x = \{x_1, x_2, \dots, x_n\}, \quad x_i = f(t_i), \quad i = 1, 2, \dots, n$$

$$y = \{y_1, y_2, \dots, y_n\}, \quad y_i = g(t_i), \quad i = 1, 2, \dots, n$$

$$z = \{z_1, z_2, \dots, z_n\}, \quad z_i = h(t_i), \quad i = 1, 2, \dots, n$$

- Unir los puntos (x_i, y_i, z_i) consecutivos mediante segmentos rectos.

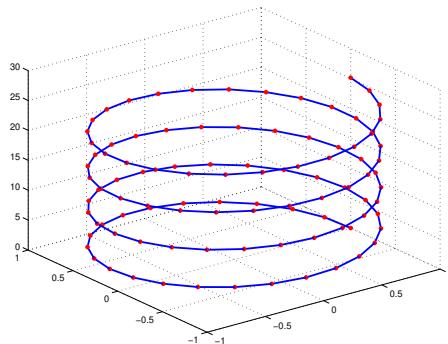


Figura 3.11: Gráfica de la curva 3D de ecuaciones paramétricas $x(t) = \cos(t)$, $y(t) = \operatorname{sen}(t)$, $z(t) = t$, $t \in [0, 8\pi]$.

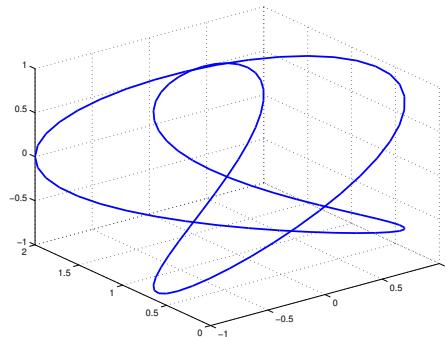


Figura 3.12: Gráfica de la curva $x(t) = \cos(3t)$, $y(t) = 2 \cos^2(t)$, $z(t) = \operatorname{sen}(2t)$, $t \in [-\pi, \pi]$.

3.1.5 Gráficas de funciones de dos variables: superficies

La relación explícita

$$z = f(x, y)$$

con $f : \Omega \subset \mathbb{R}^2 \mapsto \mathbb{R}$, representa una superficie en el espacio \mathbb{R}^3 : a cada punto (x, y) del dominio Ω del plano \mathbb{R}^2 , la función f le hace corresponder un valor z que representa la “altura” de la superficie en ese punto.

Para dibujar la superficie es preciso disponer de una “discretización” del dominio Ω en el que está definida la función, es decir un conjunto de polígonos (normalmente triángulos o rectángulos) cuya unión sea Ω .

Un mallado en rectángulos de un dominio rectangular es fácil de construir a partir de sendas particiones de sus lados. Un mallado en triángulos es más complicado y precisa de algoritmos y programas especializados.

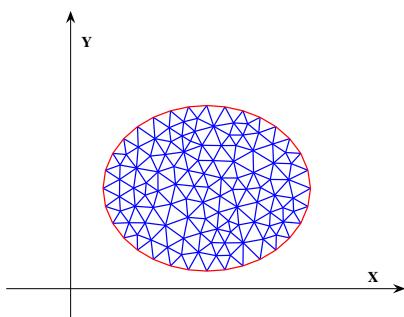


Figura 3.13: Mallado en triángulos de un dominio de frontera curva

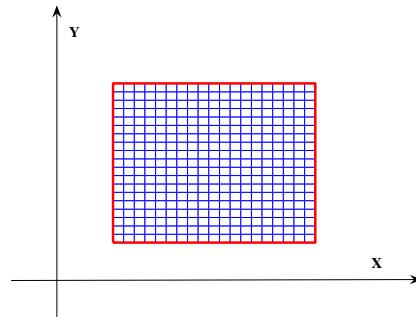


Figura 3.14: Mallado en rectángulos de un dominio rectangular

La forma de proporcionar los datos en uno y otro caso es diferente. Un mallado rectangular de un dominio $\Omega = [a, b] \times [c, d]$ queda definido mediante las particiones de los intervalos $[a, b]$ y $[c, d]$ cuyo producto cartesiano produce los nodos de la malla: $\{x_1, x_2, \dots, x_n\}$ e $\{y_1, y_2, \dots, y_m\}$.

Para definir un mallado mediante triángulos es preciso, por un lado numerar sus vértices y disponer de sus coordenadas, (x_i, y_i) , $1 = 1, \dots, n$ y, por otro, numerar sus triángulos y describirlos enumerando, para cada uno, sus tres vértices.

Elevando cada vértice del mallado según el valor de f en ese punto se consigue una representación de la superficie como una red deformada, como en las Figuras 3.16 y 3.17.

Dar un color a cada arista dependiendo del valor de la función en sus extremos, como en la Figura 3.18, puede resultar útil.

Rellenando de color cada retícula del mallado, la superficie se hace opaca. El color de las caras puede ser constante en toda la superficie, como en la Figura 3.20, constante en cada cara, como en la Figura 3.21, o interpolado, es decir, degradado en cada cara, en función de los valores en los vértices, como se hace en la Figura 3.22.

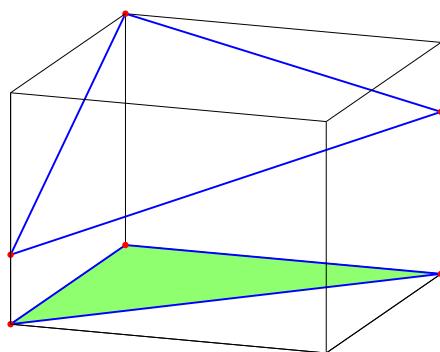


Figura 3.15: Un triángulo en 3D

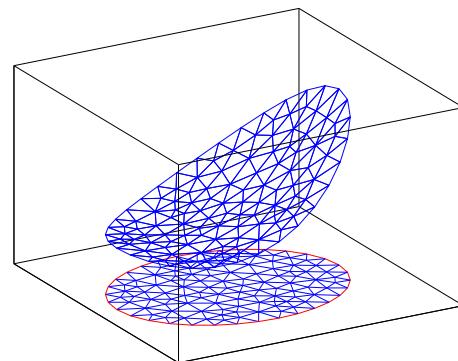


Figura 3.16: Red triangular deformada.

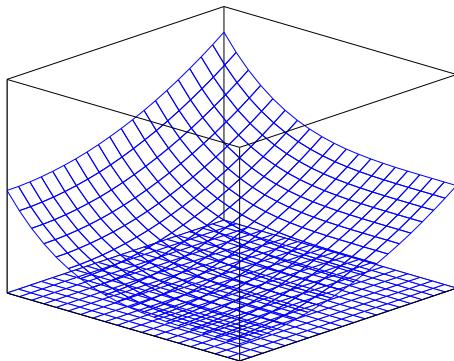


Figura 3.17: Red rectangular deformada.

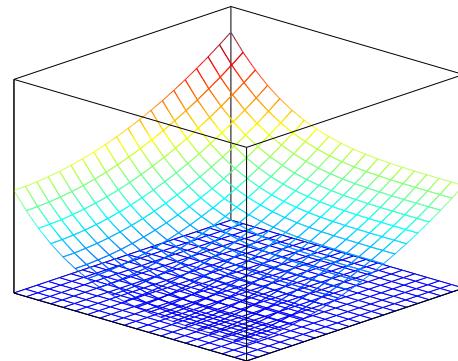


Figura 3.18: Red rectangular deformada. El color de las aristas depende del valor de la función.

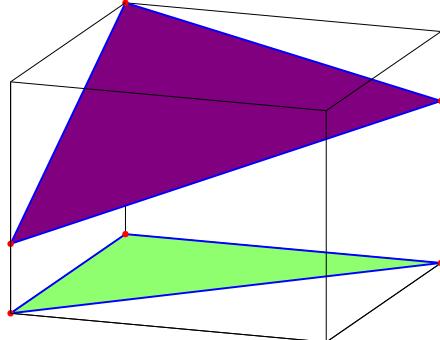


Figura 3.19: Cara triangular rellena de color plano.

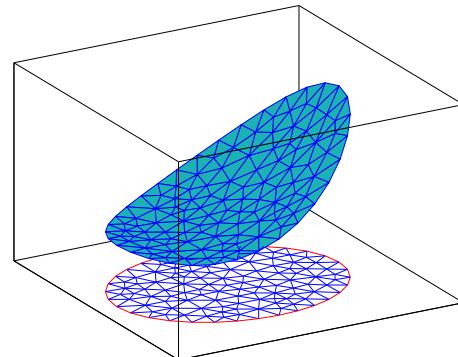


Figura 3.20: Mallado deformado con caras de color constante.

3.1.6 Superficies definidas mediante ecuaciones paramétricas

Una superficie en el espacio de tres dimensiones pueden también venir definida mediante ecuaciones paramétricas.

$$x = f(s, t), \quad y = g(s, t), \quad z = h(s, t), \quad (s, t) \in [a, b] \times [c, d]$$

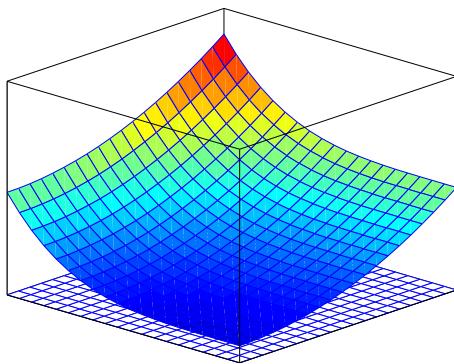


Figura 3.21: Mallado deformado con color plano en cada cara, dependiente de la altura.

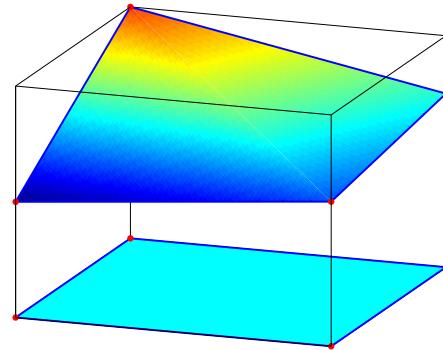


Figura 3.22: Mallado deformado con caras de color interpolado a partir de los valores en los vértices.

En este caso, para construir la gráfica de la superficie es preciso crear una discretización del dominio donde varían los parámetros, $[a, b] \times [c, d]$, y utilizar las ecuaciones paramétricas para calcular los puntos correspondientes sobre la superficie.

Por ejemplo, para dibujar la superficie cilíndrica definida por las ecuaciones

$$\begin{cases} x = f(t, \varphi) = (2 + \cos(t)) \cos(\varphi) \\ y = g(t, \varphi) = (2 + \cos(t)) \sin(\varphi) \\ z = h(t, \varphi) = t \\ t \in [0, 2\pi], \varphi \in [0, 2\pi], \end{cases},$$

hay que construir previamente particiones de los intervalos en que varían los parámetros:

$$\begin{aligned} &\{t_1, t_2, \dots, t_n\}, \\ &\{\varphi_1, \varphi_2, \dots, \varphi_n\} \end{aligned}$$

y luego, calcular los valores de x , y y z para cada par (t_i, φ_j) :

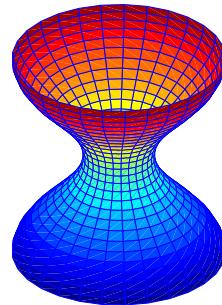


Figura 3.23: Superficie cilíndrica de ecuaciones paramétricas $x = (2 + \cos(t)) \cos(\varphi)$, $y = (2 + \cos(t)) \sin(\varphi)$, $z = t$.

3.1.7 Representación mediante curvas de nivel de una función de dos variables

Una forma habitual de representar gráficamente los valores de una función de dos variables, $f(x, y) = 0$ es dibujando sus líneas o curvas de nivel.

Se llama curva de nivel de valor k de la función $f(x, y)$ a la curva formada por los puntos del plano XY sobre los cuales la función f toma el valor k , es decir la curva implícitamente definida

por la ecuación

$$f(x, y) = k$$

El dibujo de las curvas de nivel correspondientes a un conjunto de valores k proporciona una buena información del comportamiento de la función f .

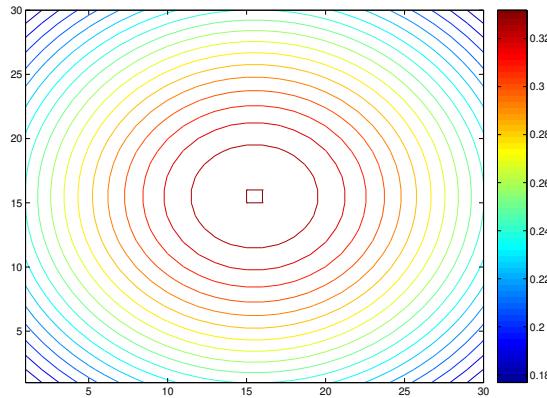


Figura 3.24: 20 curvas de nivel, correspondientes a valores equiespaciados, de la función $f(x, y) = \cos((x^2 + y^2)/4)/(3 + x^2 + y^2)$, $x, y \in [-1, 1]$.

3.2 Funciones gráficas de MATLAB fáciles de usar

Vemos en esta sección una serie de funciones, «fáciles de usar», que permiten realizar gráficos de diversos tipos, como por ejemplo los mostrados aquí debajo, de funciones de manera sencilla, a partir de su expresión analítica.

3.2.1 Gráficas de funciones de una variable real

Para dibujar una curva definida por una relación de la forma $y = f(x)$ donde $f : [a, b] \mapsto \mathbb{R}$ es una función de una variable real dada por una expresión analítica, se puede usar la función **ezplot** de MATLAB, en alguna de estas versiones:

```
ezplot(f)
ezplot(f, [a,b])
```

f es la función a representar y puede ser especificada como sigue:

'expresion' mediante una cadena de caracteres, delimitada por apóstrofes, conteniendo la expresión de la función.

nombre mediante un «handle» (manejador) de una función, que, a su vez, puede ser anónima o una M-función.

[a, b] (opcional) es el intervalo que recorre la variable independiente.

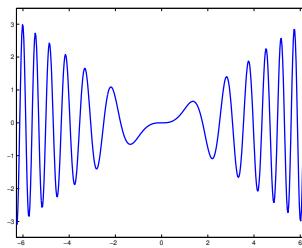
Ejemplo 3.1

Representar, usando **ezplot**, la gráfica de la función

$$f(x) = \frac{x \sin(x^2)}{2} \quad \text{en } [-2\pi, 2\pi]$$

Por defecto, la función **ezplot** hace variar la variable independiente en el intervalo $[-2\pi, 2\pi]$. Por lo tanto, basta escribir en la línea de comandos

```
ezplot('sin(x^2)*x/2')
```



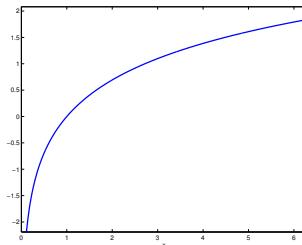
Ejemplo 3.2

Representar, usando `ezplot`, la gráfica de la función

$$f(x) = \ln(x)$$

La función $f(x) = \ln(x)$ no está definida para valores de $x \leq 0$. La función `ezplot` se limitará a dibujar la función en la parte del intervalo $[-2\pi, 2\pi]$ en el que sí está definida: $x \in (0, 2\pi]$. Escribe, en la línea de comandos

```
ezplot('log(x)')
```

**Ejercicio 3.3** Representar, usando `ezplot`, la gráfica de la función

$$f(x) = \sqrt{1 - x^2}$$

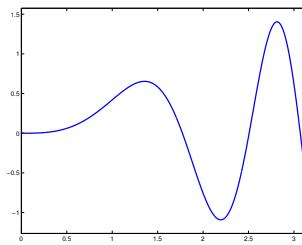
Ejemplo 3.4

Representar, usando `ezplot`, la gráfica de la función

$$f(x) = \frac{x \sin(x^2)}{2} \quad \text{para } x \in [0, \pi]$$

Como ejercicio, utilizamos una función anónima para describir la función:

```
f=@(x) x*sin(x^2)/2;
ezplot(f, [0,pi])
```



3.2.2 Curvas planas definidas implícitamente

Una relación del tipo

$$f(x, y) = 0$$

también puede definir una curva: la formada por los puntos (x, y) del plano sobre los cuales la función f toma el valor cero. Cuando una curva viene definida de esta manera se dice que viene definida de forma implícita. Por ejemplo, la ecuación $x^2 + y^2 = 4$ define una circunferencia de centro el origen y radio 2.

La misma función `ezplot` sirve para dibujar una curva así definida, usada en alguna de estas formas:

```
ezplot(f)
ezplot(f, [a,b])
ezplot(f, [a,b,c,d])
```

f es la función (de dos variables) que describe la ecuación, y puede ser especificada como sigue:

'expresión' mediante una cadena de caracteres, delimitada por apóstrofes, conteniendo la expresión de la función de dos variables.

nombre mediante un «handle» (manejador) de una función de dos variables (de nuevo, anónima o M-función).

[a, b] (opcional) la curva se dibuja en el cuadrado del plano $[a, b] \times [a, b]$. Si no se especifica, `ezplot` usa por defecto $[a, b] = [-2\pi, 2\pi]$.

[a, b, c, d] (opcional) la curva se dibuja en el rectángulo del plano $[a, b] \times [c, d]$.

Ejemplo 3.5

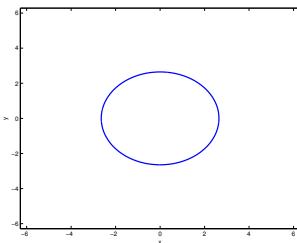
Dibujar, usando `ezplot`, la circunferencia $x^2 + y^2 = 7$

Se trata de una curva definida en forma implícita. Lo primero es escribir la ecuación en forma homogénea, es decir, con segundo miembro nulo:

$$x^2 + y^2 - 7 = 0.$$

De lo que se trata es de dibujar la curva formada por los puntos del plano sobre los cuales la **función de dos variables** $f(x) = x^2 + y^2 - 7$ toma el valor 0. Para ello basta con escribir en la línea de comandos

```
ezplot('x^2+y^2-7')
```



Utilizando una función anónima, hubiéramos tenido que definir la función de dos variables $f(x, y) = x^2 + y^2 - 7$, y luego usar `ezplot`

```
func = @(x,y) x^2+y^2-7
ezplot(f)
```

Aunque también sería válido escribir:

```
ezplot(@(x,y) x^2+y^2-7)
```

Ejercicio 3.6 Dibujar la curva implícitamente definida por la ecuación

$$x^2 \cosh(y) = 7y + 5 \quad \text{para } x \in [-10, 10], y \in [-5, 15]$$

utilizando para ello una función anónima.

3.2.3 Curvas planas definidas por ecuaciones paramétricas

Las curvas definidas mediante ecuaciones paramétricas

$$x = f(t), \quad y = g(t), \quad t \in [a, b].$$

se pueden dibujar también con la función `ezplot`, en la forma siguiente:

```
ezplot(f, g)
ezplot(f, g, [a,b])
```

`f, g` son las dos funciones que definen la curva. Pueden ser especificadas, como antes, mediante sus expresiones o mediante manejadores.

`[a,b]` (opcional) es el intervalo en el que varía el parámetro de la curva, t . Si no se especifica, `ezplot` usa por defecto $[a, b] = [0, 2\pi]$.

Ejemplo 3.7

Dibujar, usando `ezplot`, la curva definida por las ecuaciones (paramétricas)

$$x = f(t) = \sin(3t), \quad y = g(t) = \cos(t) \quad \text{para } t \in [0, 2\pi]$$

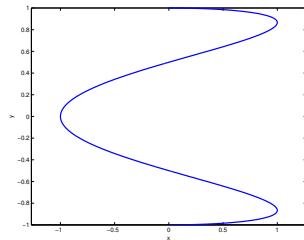
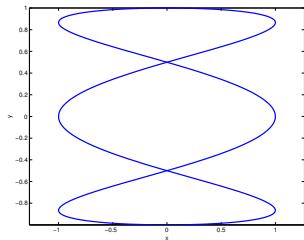
1. Escribiendo directamente en la línea de comandos

```
ezplot('sin(3*t)', 'cos(t)')
```

se obtendrá la figura de la izquierda. Si t recorriera sólo el intervalo $[0, \pi]$, con la orden

```
ezplot('sin(3*t)', 'cos(t)', [0, 2*pi])
```

se obtendría la figura de la derecha.



2. Otra forma de hacerlo, usando funciones anónimas sería:

```
funx = @(t) sin(3*t)
funy = @(t) cos(t)
ezplot(funx,funy)
```

Ejercicio 3.8 Escribir un script que dibuje, usando `ezplot` y funciones anónimas, la curva definida por las ecuaciones:

$$\begin{cases} x = f(t) = \cos(t) + 1/2 \cos(7t) + 1/3 \sin(17t) \\ y = g(t) = \sin(t) + 1/2 \sin(7t) + 1/3 \cos(17t) \end{cases}, \quad \text{para } t \in [0, 2\pi]$$

3.2.4 Curvas planas en coordenadas polares

Una relación del tipo

$$r = f(\theta), \quad \theta \in [a, b],$$

define de forma explícita una curva en coordenadas polares. MATLAB dispone de la función `ezpolar` para dibujar curvas así definidas. Su uso es análogo a la función `ezplot`:

```
ezpolar(f)
ezpolar(f, [a,b])
```

f es la función a representar $f = f(\theta)$. Se especifica en las formas habituales.

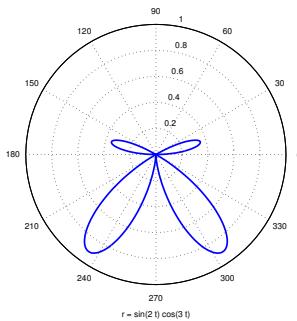
[a, b] (opcional) son los límites del intervalo en el que varía θ , es decir donde varía el ángulo. Si no se especifica, **ezpolar** toma por defecto $[a, b] = [0, 2\pi]$.

Ejemplo 3.9

Representar, usando **ezpolar**, la gráfica de la siguiente función definida en coordenadas polares

$$r = \sin(2\theta) \cos(3\theta), \quad \text{para } \theta \in [0, \pi]$$

```
ezpolar('sin(2*t)*cos(3*t)', [0,pi])
```



Ejercicio 3.10 Representar usando **ezpolar** la gráfica de la siguiente función definida en coordenadas polares

$$r = 2 \sin(6\theta), \quad \text{para } \theta \in [0, 2\pi]$$

3.2.5 Sobre los nombres de las variables independientes

Se observa que las variables de la expresión de las funciones no tienen que llamarse necesariamente t , x , y o θ . Se puede utilizar cualquier nombre válido de variable. Por ejemplo, el comando

```
ezpolar('nombre*sin(nombre)')
```

dibujará la función $r = \theta \sin(\theta)$ en coordenadas polares. El mismo resultado se obtendría con

```
func = @(nombre) nombre*sin(nombre);
ezpolar(func)
```

Por otra parte, cuando se trata de una función de dos variables, y si no se especifica el orden, MATLAB tomará como primer argumento la primera en orden alfabético y como segundo, la segunda. Como ejemplo, compruébese cómo las dos órdenes siguientes producen resultados distintos:

```
ezplot('1+v/(sin(a)+2)')
ezplot('1+v/(sin(w)+2)')
```

Esto no sucede si se utilizan funciones anónimas, ya que se indica expresamente el orden de las variables.

3.2.6 Curvas en tres dimensiones

La curva 3D definida por las ecuaciones

$$\begin{cases} x = f(t) \\ y = g(t) & \text{para } t \in [a, b] \\ z = h(t) \end{cases}$$

se puede dibujar con MATLAB usando la función `ezplot3`:

```
ezplot3(f, g, h)
ezplot3(f, g, h, [a,b])
```

`f, g, h` son las expresiones de las tres funciones dependientes de t . Se proporcionan en cualquiera de las formas habituales

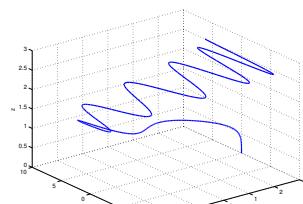
`[a, b]` es el intervalo donde varía el parámetro t . Si no se especifican, `ezplot3` toma por defecto $[a, b] = [0, 2\pi]$.

Ejemplo 3.11

Dibujar, usando `ezplot3`, la siguiente curva tridimensional:

$$x = 3 \cos(t), \quad y = t \operatorname{sen}(t^2), \quad z = \sqrt{t}, \quad t \in [0, 2\pi]$$

```
ezplot3('3*cos(t)', 't*sin(t^2)', 'sqrt(t)')
```



Ejercicio 3.12 Dibujar, usando `ezplot3`, la siguiente curva:

$$x = \cos(t), \quad y = \sin(t), \quad z = t, \quad t \in [0, 8\pi]$$

3.2.7 Superficies definidas mediante ecuaciones explícitas

La ecuación

$$z = f(x, y)$$

con $f : \Omega \subset \mathbb{R}^2 \mapsto \mathbb{R}$, representa una superficie en el espacio \mathbb{R}^3 : a cada punto (x, y) del dominio Ω del plano \mathbb{R}^2 la función f hace corresponder un valor z que representa la «altura» de la superficie en ese punto.

En MATLAB, la función `ezmesh` permite dibujar la superficie $z = f(x, y)$ cuando (x, y) recorre un rectángulo del plano:

```
ezmesh(f)
ezmesh(f, [a,b])
ezmesh(f, [a,b,c,d])
```

f es la expresión de la función (de dos variables) $f(x, y)$. Se puede proporcionar en cualquiera de las formas habituales.

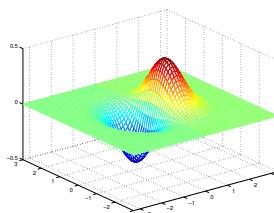
[a, b] (opcional) establece el rectángulo para dibujar la superficie: $[a, b] \times [a, b]$. Si no se especifica, `ezmesh` toma $[a, b] = [-\pi, \pi]$.

[a, b, c, d] (opcional) establece el rectángulo para dibujar la superficie: $[a, b] \times [c, d]$

Ejemplo 3.13

Dibujar la superficie $z = x e^{-(x^2+y^2)}$

```
ezmesh('x*exp(-x^2-y^2)')
```



3.2.8 Superficies definidas mediante ecuaciones paramétricas

La superficie definida por las ecuaciones paramétricas:

$$\begin{cases} x = f(s, t) \\ y = g(s, t), & (s, t) \in [a, b] \times [c, d] \\ z = h(s, t) \end{cases}$$

se puede dibujar usando también **ezmesh**:

```
ezmesh(f, g, h)
ezmesh(f, g, h, [a,b])
ezmesh(f, g, h, [a,b,c,d])
```

f, g, h son las expresiones de las funciones de dos variables $f(s, t)$, $g(s, t)$ y $h(s, t)$

[a,b] (opcional) define el dominio en que varían s y t : $s, t \in [a, b]$. Si no se especifica, **ezmesh** toma $[a, b] = [-2\pi, 2\pi]$.

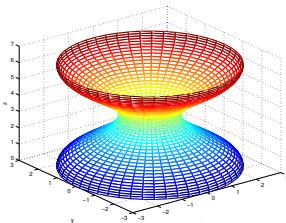
[a,b,c,d] (opcional) define el dominio en que varían s y t : $(s, t) \in [a, b] \times [c, d]$, es decir, $s \in [a, b]$ y $t \in [c, d]$.

Ejemplo 3.14

Dibujar la superficie cilíndrica definida por las ecuaciones paramétricas

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

```
ezmesh('cos(s)*(2+cos(t))','sin(s)*(2+cos(t))','t',[0,2*pi])
```



Ejercicio 3.15 Dibujar la gráfica de la superficie siguiente:

$$\begin{cases} x(s, t) = \cos(s)(2 + \cos(t)) \\ y(s, t) = \sin(s)(2 + 0.3 \cos(t)), & s, t \in [0, 2\pi] \\ z(s, t) = \sin(t) \end{cases}$$

3.2.9 Representación mediante curvas de nivel de una función de dos variables

La representación mediante curvas de nivel de una función $f : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}$ se puede obtener fácilmente mediante la función **ezcontour**:

```
ezcontour(f)
```

La función **ezcontourf** actúa como el anterior, pero rellena con colores los espacios entre curvas

```
ezcontourf(f)
```

La función **ezmeshc** actúa igual que **ezmesh**, pero dibuja también las curvas de nivel

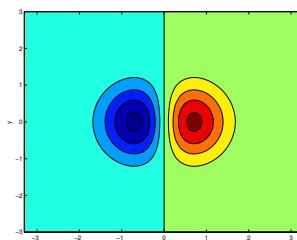
```
ezmeshc(f)
```

Ejemplo 3.16

Dibujar usando **ezcontourf** la curvas de nivel de la función

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

```
ezcontourf('x*exp(-x^2 - y^2)')
```



Ejercicio 3.17 Dibujar usando `ezmeshc` la superficie $z = f(x, y)$ y las curvas de nivel de la función $f(x, y)$, siendo $f(x, y) = \sin(x/2) \sin(y/2)$.

3.2.10 Observación importante

Las funciones `ez****` que se han presentado en esta sección han sido diseñadas para que el usuario pueda obtener la gráfica de una función o dibujar una curva con suma facilidad y sin tener que preocuparse en exceso de las exigencias del lenguaje.

Por ello, “corrigen” un buen número de fallos propios de los usuarios noveles, eventuales o poco cuidadosos.

Por ejemplo, no se produce un error al dar la orden `ezplot('sqrt(x)', [-3, 3])` a pesar de que la función a dibujar no está definida para valores negativos de x . Como tampoco se produce al dar la orden `ezplot('1/x', [-10, 10])`, aunque esta función no está definida en $x = 0$.

Otro ejemplo, de mucha importancia, es que permite escribir las expresiones de forma no vectorizada, es decir, sin hacer uso de los operadores aritméticos elemento a elemento, lo que, en cualquier otro contexto dentro de MATLAB, no produciría el resultado esperado.

Como en casi todas las cosas, lo que se gana en facilidad por un lado se paga en eficacia por otro. En la sección siguiente se ven órdenes, más básicas que éstas, pero que permiten realizar gráficas más personalizadas y menos estándar.

El uso de las funciones `ez****` debería limitarse a ocasiones en que se desea tener una idea rápida de la forma de una curva o superficie, mediante una sola orden directamente en la línea de comandos.

3.3 Funciones básicas para el dibujo de curvas

En esta sección se muestran algunas de las funciones elementales de dibujo de MATLAB, cuya utilización permitirá realizar gráficas más complejas y personalizadas.

3.3.1 La orden `plot`

La orden básica para dibujar una curva plana es

```
plot(x,y)
```

siendo `x` e `y` dos vectores de las mismas dimensiones conteniendo, respectivamente, las abscisas y las ordenadas de los puntos de la gráfica (los puntos (x_i, y_i) de la Figura 3.1).

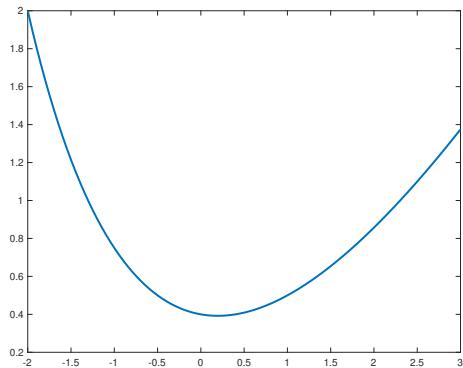
Ejemplo 3.18

Usando la función `plot`, dibujar la curva $y = \frac{x^2 + 2}{x + 5}$ para $x \in [-2, 3]$

Comenzamos por construir un soporte de nodos $\{x_i\}$ regularmente espaciados en el intervalo $[-2, 3]$. Para ello utilizamos la función `linspace` con la opción por defecto de 100 puntos (suficiente para la mayoría de las gráficas). Luego calculamos los valores de la función $f(x) = \frac{x^2 + 2}{x + 5}$ en todos esos puntos (usando operaciones vectoriales cuando haga falta) y dibujamos la curva.

```
x = linspace(-2, 3);
y = (x.^2+2)./(x+5);
plot(x,y)
```

Observamos que la función $f(x)$ es continua en el intervalo $[-2, 3]$. Si la dibujáramos, por ejemplo, en el intervalo $[-7, -2]$, que contiene un punto de discontinuidad de f , obtendríamos un dibujo erróneo que podría resultar equívoco.



Es posible dibujar dos o más curvas a la vez, proporcionando a la función `plot` varios pares de vectores (abscisas, ordenadas).

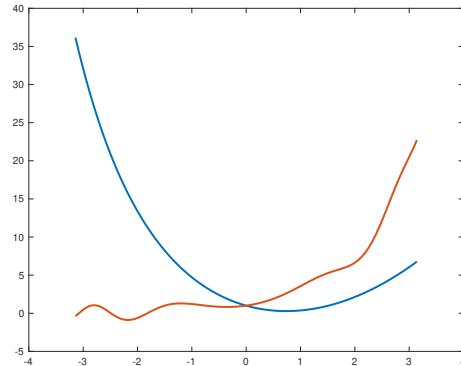
```
plot(x1, y1, x2, y2, x3, y3)
```

Ejemplo 3.19

Usando la función `plot`, dibujar las curvas $y = \frac{1}{e^x} + x^2 - x$ e $y = e^x + \sin(x^2)$ en el intervalo $[-\pi, \pi]$.

Puesto que hay que dibujar ambas curvas en el mismo intervalo $[-\pi, \pi]$, usamos el mismo vector de abscisas para las dos.

```
x = linspace(-pi, pi);
y1 = exp(-x) + x.^2 - x;
y2 = exp(x) + sin(x.^2);
plot(x, y1, x, y2)
```

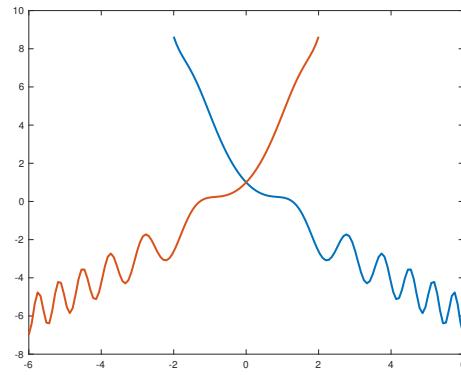
**Ejemplo 3.20**

Usando la función `plot`, dibujar los arcos de curva:

$$\begin{cases} y = \frac{1}{e^x} + \sin(x^2) - x, & x \in [-2, 6] \\ y = e^x + \sin(x^2) + x, & x \in [-6, 2] \end{cases}$$

Las curvas a dibujar tienen distintos intervalos, luego tenemos que crear dos vectores. Vamos, ahora, a definir las funciones que definen las curvas como funciones anónimas.

```
f = @(x) exp(-x) + sin(x.^2) - x;
g = @(x) exp(x) + sin(x.^2) + x;
xf = linspace(-2, 6);
xg = linspace(-6, 2);
yf = f(xf);
yg = g(xg);
plot(xf, yf, xg, yg)
```



Ejercicio 3.21 Representar la gráfica de la función

$$f(x) = \frac{\cos(x/4)}{\ln(1+x^2)},$$

para x entre $-\pi$ y π , evitando la discontinuidad de f en $x = 0$.

Ejemplo 3.22

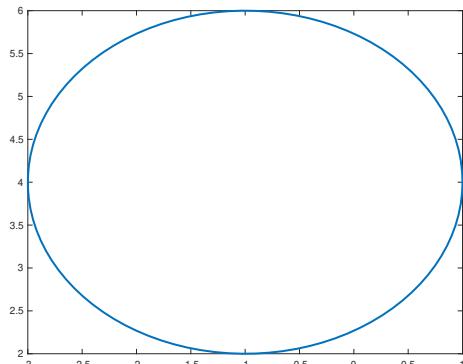
Usando la función `plot`, dibujar la circunferencia de centro el punto $(-1, 4)$ y radio 2.

Lo más cómodo para dibujar una circunferencia es utilizar sus ecuaciones paramétricas: los puntos de la circunferencia de centro (a, b) y radio r vienen dados por

$$\begin{cases} x = a + r \cos(t) \\ y = b + r \sin(t) \end{cases} \quad t \in [0, 2\pi]$$

```
t = linspace(0, 2*pi);
x = -1 + 2*cos(t);
y = 4 + 2*sin(t);
plot(x, y, 'LineWidth', 2)
```

Se observa que la curva dibujada no se ve “redonda”. Ello es debido a que MATLAB aplica distintos factores de escala a los ejes OX y OY . Más adelante se verá cómo evitar esto.



Ejercicio 3.23 Dibujar la semi-circunferencia inferior de centro $(-2, 0)$ y radio 3.

Ejercicio 3.24 Dibujar la curva dada por las ecuaciones:

$$\begin{cases} x = 6 \cos(2/t) - 2 \cos(5t) \\ y = 3(\sin t)^6 \end{cases} \quad t \in [\pi, 3\pi]$$

3.3.2 Color y tipo de línea

La orden `plot` admite un argumento opcional, que se puede añadir a cada par `x,y` para elegir el color y el tipo de línea a dibujar. Este argumento consiste en una cadena de 1,2 o 3 caracteres, de los indicados en la tabla. Los marcadores y las líneas de trazos y/o puntos son de gran utilidad cuando se dibujan varias curvas en una misma gráfica en un documento que va a ser impreso en blanco y negro.

Más opciones de dibujo se pueden tener utilizando los nombres de las **Propiedades** que se muestran en la subsección siguiente.

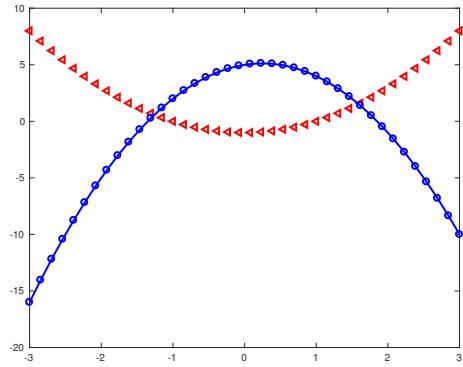
Símbolo	Color	Símbolo	Marcadores (markers)
y	amarillo	.	puntos
m	magenta	o	círculos
c	cian	x	aspas
r	rojo	+	cruces
g	verde	*	asteriscos
b	azul	s	cuadros
w	blanco	d	diamantes
k	negro	^	triángulos hacia arriba
		v	triángulos hacia abajo
Símbolo	Estilo de línea	>	triángulos hacia la derecha
-	continua	<	triángulos hacia la izquierda
:	de puntos	p	estrellas de 5 puntas
-.	de trazos y puntos	h	estrellas de 6 puntas
--	de trazos		

Tabla 3.1: Símbolos para definir el color y el tipo de línea a dibujar en la orden `plot`

Ejemplo 3.25

Dibujar las paráolas $y = x^2 - 1$ e $y = -2x^2 + x + 5$ en el intervalo $[-3, 3]$ usando distintos colores, marcadores y tipo de líneas.

```
f = @(x) x.^2 - 1;
g = @(x) -2*x.^2 + x + 5;
x = linspace(-3,3, 40);
plot(x, f(x), 'r<', x, g(x), 'bo-')
```



3.3.3 Personalizar las propiedades de las líneas

Además del procedimiento “abreviado” explicado en la sección anterior, se puede influir sobre las características de una línea mediante el uso de ciertos parámetros opcionales de la orden `plot`. Estos se utilizan mediante pares compuestos por el **Nombre de la propiedad** y el **Valor de la propiedad**:

```
plot(x, y, 'Propiedad', Valor, ...)
```

Algunas de las propiedades más utilizadas están descritas en la tabla 3.2. Este procedimiento se puede combinar con el método abreviado descrito en la sección 3.3.2.

Para información sobre el resto de propiedades de las líneas, se puede consultar, en el **Help** de MATLAB, **Line Properties**.

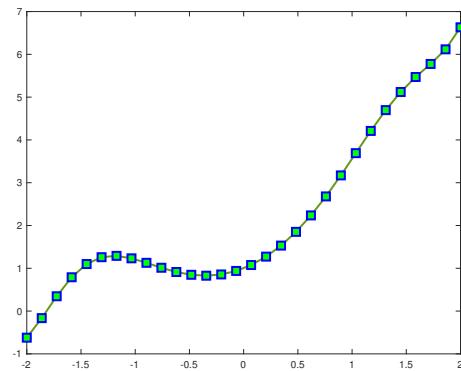
Propiedad	Descripción	Valores admisibles
'LineStyle'	Controla el estilo de la línea: continua, de trazos, de puntos, ...	'—', '—', '-.', ':', 'none'
'LineWidth'	Controla el grosor de la línea	Valores numéricos positivos: 0.5 (por defecto), 1, 2, ...
'Color'	Controla el color de la línea	<ul style="list-style-type: none"> Coordenadas del color en el sistema RGB, por ejemplo [0.6, 0.3, 0.1] Los nombres o abreviaturas de los colores básicos (ver tabla 3.1): 'c', 'b', 'red', 'green' ...
'Marker'	Controla el marcador de la línea	Cualquiera de los indicados en la tabla 3.1. Por defecto es 'none'.
'MarkerSize'	Controla el tamaño de los marcadores	Valor numérico positivo: 6 (por defecto), 10, ...
'MarkerEdgeColor'	Controla el color del borde de los marcadores	Un color
'MarkerFaceColor'	Controla el color de relleno de los marcadores	Un color

Tabla 3.2: Algunas propiedades de las líneas, controlables con la orden `plot`

Ejemplo 3.26

Dibujar la curva de ecuación $y = e^x + \sin(x^2)$ en el intervalo $[-2, 2]$ usando distintos colores, marcadores y tipo de líneas.

```
g = @(x) exp(x) + sin(x.^2);
x = linspace(-2, 2, 30);
plot(x, g(x), 's-', ...
      'Color', [0.4, 0.6, 0.1], ...
      'MarkerSize', 10, ...
      'MarkerEdgeColor', 'blue', ...
      'MarkerFaceColor', 'g', ...
      'LineWidth', 2)
```



3.3.4 Personalizar los ejes: orden axis

En MATLAB, la orden `axis` controla los límites y el aspecto de la caja que ocupa la gráfica. Por defecto, los límites de los ejes vienen determinados por los valores de la gráfica que se dibuja.

<code>axis auto</code>	(valor por defecto)
------------------------	---------------------

Si se desean cambiar, se utiliza la orden:

<code>axis([xmin, xmax, ymin, ymax])</code>	(en 2D)
<code>axis([xmin, xmax, ymin, ymax, zmin, zmax])</code>	(en 3D)

La orden

<code>axis equal</code>

establece el mismo factor de escala para ambos ejes. Esto haría, por ejemplo, que la gráfica de una circunferencia se vea “redonda” en lugar de parecer una elipse.

La orden

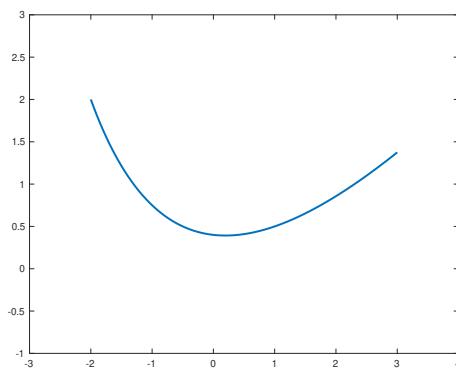
<code>axis off</code>
<code>axis on</code>

elimina/muestra los ejes de la gráfica.

Ejemplo 3.27

Dibujar la curva $y = \frac{x^2 + 2}{x + 5}$ para $x \in [-2, 3]$, modificando los límites de los ejes.

```
x = linspace(-2, 3);
y = (x.^2+2)./(x+5);
plot(x, y, 'LineWidth', 2)
axis([-3 4, -1, 3])
```

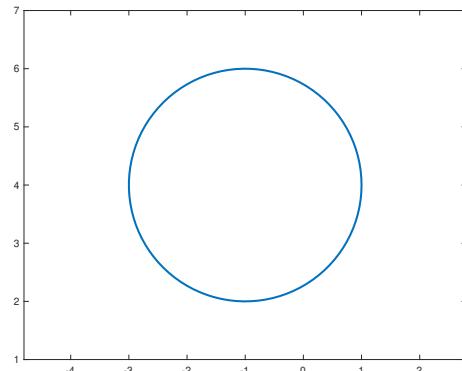


Ejemplo 3.28

Dibujar la circunferencia de centro el punto $(-1, 4)$ y radio 2, imponiendo el mismo factor de escala en los ejes X e Y .

Como en el Ejemplo 3.22, utilizamos las ecuaciones paramétricas de la circunferencia.

```
t = linspace(0, 2*pi);
x = -1 + 2*cos(t);
y = 4 + 2*sin(t);
plot(x, y, 'LineWidth', 2)
axis([-3, 1, 1, 7])
axis equal
```



Las órdenes siguientes muestran/ocultan una cuadrícula:

```
grid on
axis off
```

3.3.5 Anotaciones: título, etiquetas y leyendas

Las funciones

```
title('Texto de título')
xlabel('Etiqueta del eje OX')
ylabel('Etiqueta del eje OY')
```

incluyen, respectivamente, un título, una etiqueta asociada al eje OX (horizontal) y una etiqueta asociada al eje OY (vertical).

La función

```
legend('Leyenda_1', 'Leyenda_2', ...)
```

inserta en el gráfico una leyenda, usada normalmente para explicar el significado de cada curva. Las leyendas de la función **legend** se asignan a las curvas en el orden en que se dibujaron.

El título y las etiquetas de los ejes admiten, entre otras, las propiedades contenidas en la tabla 3.3. Para obtener información sobre las propiedades de la leyenda, buscar **legend properties** en el **Help** de MATLAB.

Propiedad	Descripción	Valores admisibles
'FontSize'	Controla el tamaño de la letra	Valor entero positivo (puntos). Por defecto es 11
'FontName'	Controla el tipo de letra	Tipos soportado por el sistema: 'Arial', 'Helvetica', 'FixedWidth' (por defecto)
'FontWeight'	Controla el grosor de la letra	'normal', 'bold'
'Color'	Controla el color del texto	Un color

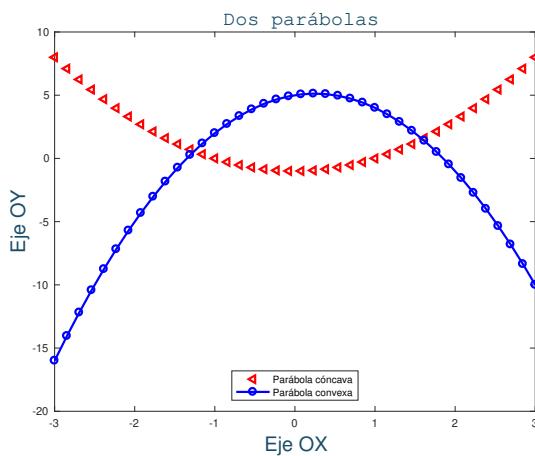
Para más información buscar **title properties** o **xlabel properties** en el **Help** de MATLAB

Tabla 3.3: Algunas propiedades de los títulos y las etiquetas de los ejes

Ejemplo 3.29

Dibujar las parábolas $y = x^2 - 1$ e $y = -2x^2 + x + 5$ en el intervalo $[-3, 3]$ usando distintos colores, marcadores y tipo de líneas y añadiendo título, etiquetas y leyenda.

```
f = @(x) x.^2 - 1;
g = @(x) -2*x.^2 + x + 5;
x = linspace(-3,3, 40);
plot(x, f(x), 'r<', x, g(x), 'bo-', 'LineWidth', 2)
title('Dos par\'abolas', 'FontSize', 18, 'FontName', 'Verdana', ...
      'FontWeight', 'normal', 'Color', [0.1, 0.3, 0.4])
xlabel('Eje OX', 'FontSize',18, 'Color', [0.1, 0.3, 0.4])
ylabel('Eje OY', 'FontSize',18, 'Color', [0.1, 0.3, 0.4])
legend('Par\'abola c\'oncava', 'Par\'abola convexa', 'Location', 'south')
```



3.3.6 Dibujar cosas encima de otras

Cada nueva orden `plot` borra el contenido anterior de la ventana gráfica. Si se desea añadir elementos al dibujo sin borrar lo anterior hay que usar las órdenes

```
hold on  
hold off
```

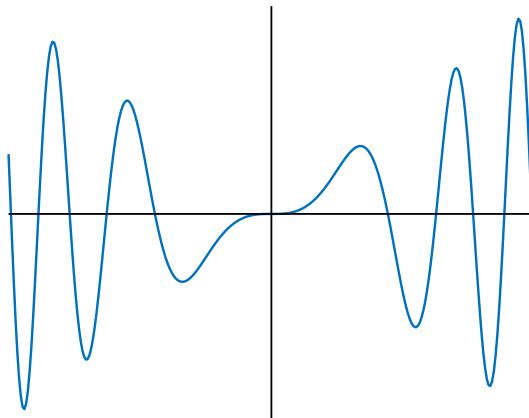
La primera de ellas anula el borrado previo de la gráfica antes de un nuevo dibujo. La segunda devuelve el sistema a su estado normal.

La orden `hold on` debe utilizarse **después** de haber realizado algún dibujo (el primero) o de haber utilizado la orden `axis([xmin, xmax, ymin, ymax])` para fijar los límites de los ejes en los que se va a dibujar.

Ejemplo 3.30

Dibujar la curva de ecuación $y = \frac{x \sin(x^2)}{2}$ para $x \in [-4, 4]$, dibujando también los ejes de coordenadas.

```
f = @(x) 0.5*x.*sin(x.^2);  
x = linspace(-4, 4, 300);  
axis([-4, 4, -2, 2])  
hold on  
plot(x, f(x), 'LineWidth', 2)  
plot([-4, 4], [0, 0], 'k', 'LineWidth', 1.5)  
plot([0,0], [-2, 2], 'k', 'LineWidth', 1.5)  
hold off  
axis off
```



Obsérvese que hemos utilizado un número de puntos superior al habitual para realizar la gráfica, ya que la curva es muy oscilante.

3.3.7 Añadir texto a la gráfica

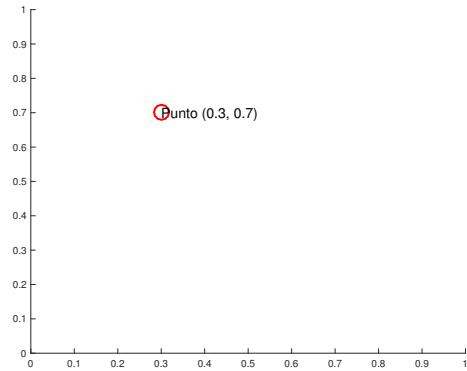
Para añadir texto a una gráfica se utiliza la función `text`, de la que presentamos aquí sólo algunas de sus funcionalidades.

```
text(x, y, texto_a_añadir)
```

inserta el texto `texto_a_añadir` en el punto `(x, y)` del gráfico. Se puede controlar el tamaño de la letra con la propiedad `'FontSize'`. Para obtener más información sobre otras (muchas) propiedades aplicables al texto, consultar `text properties` en el `Help` de MATLAB.

Ejemplo 3.31

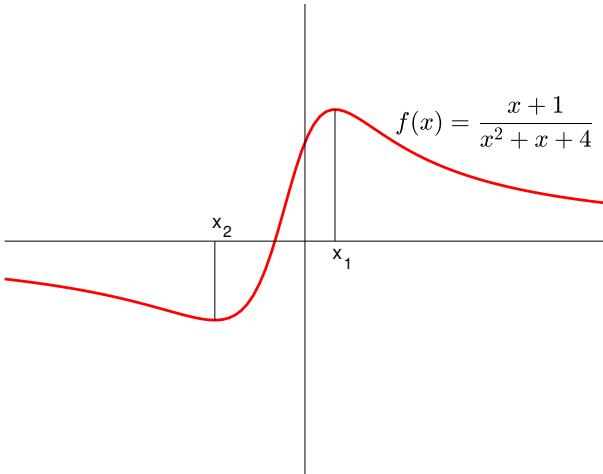
```
axis([0, 1, 0, 1])
hold on
plot(0.3, 0.7, 'ro', 'MarkerSize', 15, ...
      'LineWidth', 2)
text(0.3, 0.7, 'Punto (0.3, 0.7)', ...
      'FontSize', 14)
```



Ejemplo 3.32

Uso de algunas opciones de `text`

```
x = linspace(-10, 10);
y = (x+1)./(x.^2+x+4);
axis([-10, 10, -0.6, 0.6])
hold on
plot(x, y, 'r', 'LineWidth', 2)
plot([-10,10], [0, 0], 'k')
plot([0,0], [-1, 1], 'k')
plot([1, 1], [0, 1/3], 'k')
plot([-3, -3], [0, -1/5], 'k')
text(0.9, -0.04, 'x_1', 'FontSize', 14)
text(-3.1, 0.04, 'x_2', 'FontSize', 14)
ht = text(3, 0.3, '$f(x) = \frac{x+1}{x^2+x+4}$', 'FontSize', 18);
ht.Interpreter = 'latex';
```



3.3.8 Gestión de la ventana gráfica

La función

```
figure
```

abre una nueva ventana gráfica, cuyo nombre será **Figure n**, donde **n** es el primer número entero positivo que esté libre. También se puede abrir una directamente especificando su número: **figure(7)**.

Cuando hay varias ventanas gráficas abiertas, la manera de designar cuál es la activa (en cuál se dibujará) es usar la orden **figure(n)** con el número correspondiente.

La orden

```
clf
```

borra la ventana gráfica activa, sin cerrarla.

La orden

```
shg
```

coloca la ventana gráfica activa delante de todas las demás. Muy útil cuando se está escribiendo programas que dibujan.

3.3.9 Otros tipos de gráficos de datos planos

La función

```
scatter(x, y)
```

dibuja los puntos (x_i, y_i) con marcadores (por defecto, con círculos).

La función

```
stairs(x, y)
```

dibuja los puntos (x_i, y_i) en forma de una función escalonada.

La función

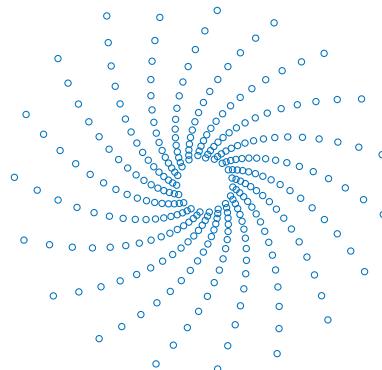
```
stem(x, y)
```

dibuja los datos (x_i, y_i) en forma de tallos.

Ejemplo 3.33

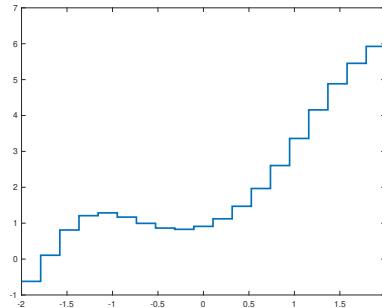
Dibujar los puntos definidos por las ecuaciones paramétricas $\begin{cases} x = e^{2t} \sin(100t) \\ y = e^{2t} \cos(100t) \end{cases}$ para 300 valores de t comprendidos entre 0 y 1.

```
t = linspace(0,1,300);
x = exp(2*t).*sin(100*t);
y = exp(2*t).*cos(100*t);
scatter(x,y)
axis equal
axis off
```

**Ejemplo 3.34**

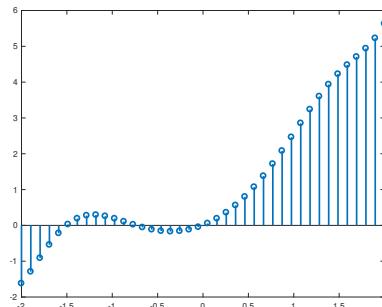
Dibujar datos como una función escalonada

```
g = @(x) exp(x) + sin(x.^2);
x = linspace(-2, 2, 20);
stairs(x, g(x))
```

**Ejemplo 3.35**

Dibujar datos con tallos

```
g = @(x) exp(x) + sin(x.^2)-1;
x = linspace(-2, 2, 40);
stem(x, g(x))
```

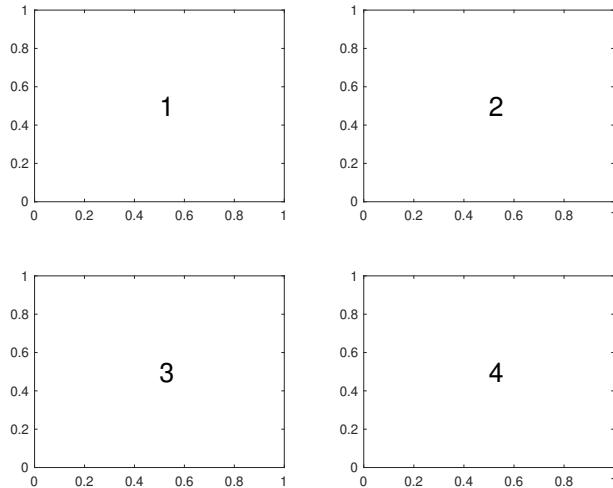


3.3.10 Varios ejes en la misma gráfica

La función siguiente sirve para crear varios ejes en una misma figura.

```
subplot(n, m, k)
```

Esta orden divide el área de la ventana gráfica en una cuadrícula $n \times m$ (n filas y m columnas), y convierte el k -ésimo cuadro en los ejes activos. MATLAB numera los cuadros de izquierda a derecha y de arriba hacia abajo.



Cada uno de los ejes así definidos funciona como si fuera una figura diferente: para dibujar en ellos hay que hacerlos activos con la orden `subplot`; la orden `hold` actúa de forma independiente en cada uno de ellos; cada uno puede tener sus propios títulos, etiquetas, leyendas, ...

Ejemplo 3.36

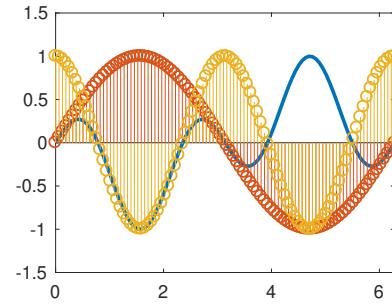
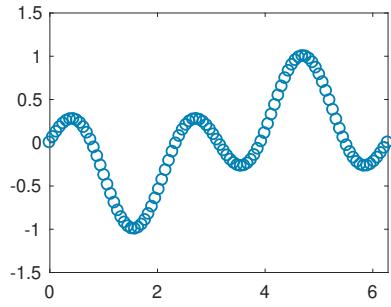
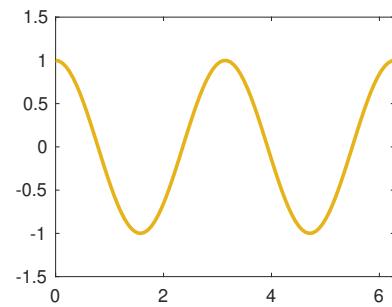
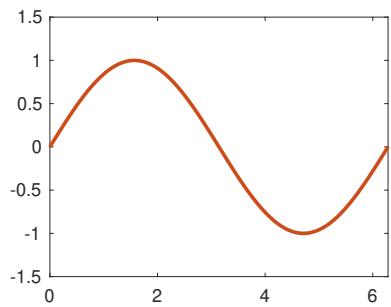
Uso de subplots

```
x = linspace(0, 2*pi);
subplot(2,2,1)
axis([0, 2*pi, -1.5, 1.5])
hold on
y1 = sin(x);
plot(x, y1, 'Linewidth', 2, 'Color', [0.8, 0.3, 0.1])

subplot(2,2,2)
axis([0, 2*pi, -1.5, 1.5])
hold on
y2 = cos(2*x);
plot(x, y2, 'Linewidth', 2, 'Color', [0.9, 0.7, 0.1])
```

```
subplot(2,2,3)
axis([0, 2*pi, -1.5, 1.5])
hold on
y3 = y1.*y2;
scatter(x, y3, 'MarkerEdgeColor', [0, 0.5, 0.7])

subplot(2,2,4)
axis([0, 2*pi, -1.5, 1.5])
hold on
plot(x, y3, 'LineWidth', 2)
stem(x, y1)
stem(x, y2)
```



3.4 Ejercicios

1. Usando las funciones `ez***` adecuadas, representar gráficamente los “objetos” definidos por las siguientes funciones, expresiones y/o ecuaciones:

a) $y = \cos(2x) + \frac{1}{2} \sin(x/2)$, $x \in [-\pi, \pi]$

b) $y = \frac{\sin^2 x}{x^2 - 1}$, $x \in [-2, 2]$.

c) $x^4 + y^3 = 24$, $x, y \in [-6, 6]$

d) $y^2 = x^4(1-x)(1+x)$, $x, y \in [-2, 2]$

e) $\begin{cases} x = t - 3 \sin t, \\ y = 4 - 3 \cos t, \end{cases} t \in [0, 10]$

f) $\begin{cases} x = 4 \cos t + \cos(4t), \\ y = 4 \sin t - \sin(4t), \end{cases} t \in [0, 2\pi]$

g) $\begin{cases} x = e^{t/4} \sin(2t), \\ y = e^{t/4} \cos(2t), \\ z = t/4 \end{cases} t \in [-9, 10]$

2. Para cada una de las funciones siguientes, definir funciones anónimas que las representen (deben admitir vectores como argumentos), y dibujar sus gráficas en un intervalo razonable usando la orden `plot`. Dibujar también, en cada caso los ejes coordenados.

a) $f(x) = \sqrt{\frac{1-x}{1+x}}$

e) $f(x) = \frac{1}{x}$

b) $f(x) = \frac{x}{\ln(x)}$

f) $f(x) = \ln\left(\frac{1}{x-2}\right)$

c) $f(x) = \sqrt{1-x} + \sqrt{x-2}$

g) $f(x) = \sqrt{25-x^2}$

d) $f(x) = 3$

h) $f(t) = \sin^2(\sqrt{3+x})$

3. Representar en los mismos ejes los siguientes pares de funciones, en los límites que se indican en cada caso, de modo que cada curva se distinga perfectamente, tanto en color como en blanco y negro. Añadir leyendas para identificar las curvas. Añadir también un título y etiquetas en los ejes.

a) $f(x) = 2 \sin^3(x) \cos^2(x)$ y $g(x) = e^x - 2x - 3$, $x \in [-1.5, 1.5]$

b) $f(x) = \log(x+1) - x$ y $g(x) = 2 - 5x$, $x \in [0, 5]$

c) $f(x) = 6 \sin(x)$ y $g(x) = 6x - x^3$, $x \in [-\pi/2, \pi/2]$

d) $f(x) = e^{-x^2+2} \sin(x/2)$ y $g(x) = -x^3 + 2x + 2$, $x \in [-1, 2]$

e) $f(x) = \sqrt{r^2 - x^2}$, para $r = 1$ y $r = 4$

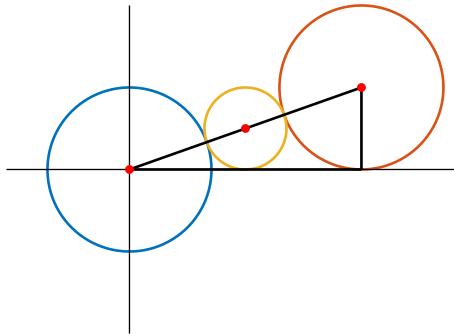
f) $g(x) = 3^x + 2x - 6$ y $h(x) = \operatorname{sen}\left(\frac{\sqrt{x^2 + 4x}}{x^3}\right)$ en $[1, 2] \times [0, 1.5]$

g) $g(x) = \cos(x^2 + 2x + 1)$ y $h(x) = \log\left(\frac{\sqrt{x+2}}{x^3}\right)$ en $[1, 2] \times [-1.5, 2]$

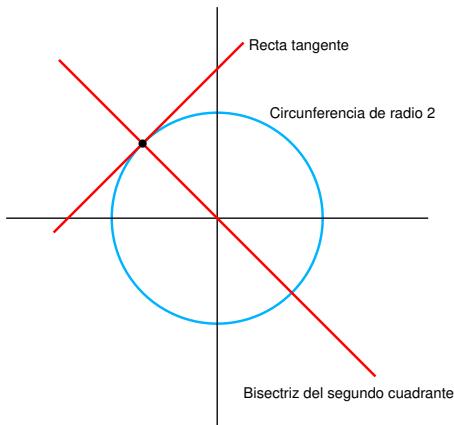
h) $g(x) = \frac{\sin(\pi x)}{1 + \sqrt{2x}}$ y $h(x) = 5e^{-2x} \ln(x^2 + 1)$ en $[0, 2] \times [-0.5, 1]$

i) $f(x) = \ln(x^2 - 27)$ y $g(x) = \frac{(x+10)(x-10)(18-x)}{100}$ en $[-15, 20] \times [-20, 15]$

4. Reproducir el dibujo siguiente. Las circunferencias grandes tienen radio = 1 y la pequeña tiene radio = 0.5.



5. Reproducir el dibujo siguiente.



4

Programación con MATLAB



Este capítulo está dedicado a los conceptos e instrucciones básicas que permiten la escritura de programas.

Comenzamos mostrando las instrucciones básicas que permiten que un programa “se comunique” con el usuario: instrucciones mediante las cuales podemos introducir, de forma interactiva, un dato para uso del programa; e instrucciones que permiten que el programa pueda escribir información y/o resultados en la pantalla del ordenador.

Los condicionales y los bucles o repeticiones son la base de la programación estructurada. Sin ellas, las instrucciones de un programa sólo podrían ejecutarse en el orden en que están escritas (orden secuencial). Las estructuras de control permiten modificar este orden y, en consecuencia, desarrollar estrategias y algoritmos para resolver los problemas.

Los **condicionales** permiten que se ejecuten conjuntos distintos de instrucciones, en función de que se verifique o no determinada condición.

Los **bucles** permiten que se ejecute repetidamente un conjunto de instrucciones, ya sea un número pre-determinado de veces, o bien mientras que se verifique una determinada condición.

4.1 Operaciones básicas de lectura y escritura

4.1.1 Instrucción básica de lectura: `input`

La instrucción `input` permite almacenar en una variable un dato que se introduce a través del teclado. La orden

```
var = input('Mensaje')
```

imprime **Mensaje** en la pantalla y se queda esperando hasta que el usuario teclea algo en el teclado, terminado por la tecla **Retorno**. Lo que se teclea puede ser cualquier expresión que use constantes y/o variables existentes en el **Workspace**. Puede ser tambien un vector o matriz. El resultado de esta expresión se almacenará en la variable **var**. Si se pulsa la tecla **Retorno** sin teclear nada se obtendrá una matriz vacía: `[]`.

El carácter de escape `\n` introducido en el texto del mensaje provoca un salto de línea.

Si lo que se quiere leer son caracteres, hay que encerrarlos entre apóstrofes. O bien se puede añadir el argumento `'s'` a la orden `input`, lo que indica que se van a leer caracteres y, en ese caso, no hay que encerrar lo que se escriba entre apóstrofes.

Ejemplos 4.1 (Uso de `input`)

Experimentar con las órdenes siguientes:

```
>> a = input('Escriba el valor de a: ')  
  
>> a = input('\n Escriba el valor de a: ')  
  
>> nombre = input('Escriba su nombre: ')  
  
>> nombre = input('Escriba su nombre: ', 's')
```

4.1.2 Instrucción básica de impresión en pantalla: `disp`

La instrucción `disp` permite imprimir en la pantalla el valor de una matriz constante o variable, sin imprimir el nombre de la variable ni `ans` y sin dejar líneas en blanco. Su utilidad es muy limitada.

```
disp(algo)
```

Ejemplos 4.2 (Uso de `disp`)

```
>> disp(pi)  
3.1416  
  
>> v = [1, 2, 3, pi];  
>> disp(v)  
1.0000    2.0000    3.0000    3.1416  
  
>> disp('El metodo no converge')  
El metodo no converge
```

Si se quieren mezclar, en una línea a imprimir con `disp`, texto y números, hay que formar con ellos un vector, pero, para ello, hay que transformar el valor numérico en una cadena de caracteres que represente ese valor, como en los siguientes ejemplos.

Ejemplos 4.3 (Uso de `disp`)

Observaciones:

- (a) La orden `date` devuelve una cadena de caracteres con la fecha actual.
- (b) Para concatenar dos cadenas de caracteres, se forma un vector con ellas: `[cadena1, cadena2]`.
- (b) La orden `num2str(num)` devuelve el dato numérico `num` como una cadena de caracteres.

```
>> disp(['Hoy es ', date])
Hoy es 18-Feb-2015

>> x = pi;
>> disp(['El valor de x es: ',num2str(x)])
El valor de x es: 3.1416
```

Ejercicio 4.4 (*HolaMundo.m*) (Uso de `input` y `disp`)

Escribir un *script* que:

- Escriba “Hola” en la pantalla
- Lea el nombre del usuario del teclado y escriba en la pantalla “Hola + nombre”
- Escriba en la pantalla “Hoy es + fecha del día”.

```
disp('')
disp(' HOLA !!! ')
nombre = input('Escribe tu nombre: ', 's');
disp('')
disp([' HOLA ', nombre, ' !!!'])
disp([' Hoy es ', date])
disp(' ')
disp(' >> ;ADIOS! <<')
```

4.1.3 Instrucción de impresión en pantalla con formato: `fprintf`

Esta orden permite controlar la forma en que se imprimen los datos. Su sintaxis para imprimir en la pantalla es

```
fprintf( formato, lista_de_datos )
```

donde:

`lista_de_datos` son los datos a imprimir. Pueden ser constantes y/o variables, separados por comas.

`formato` es una cadena de caracteres que describe la forma en que se deben imprimir los datos. Puede contener combinaciones de los siguientes elementos:

- Códigos de conversión: formados por el símbolo `%`, una letra (como `f`, `e`, `i`, `s`) y eventualmente unos números para indicar el número de espacios que ocupará el dato a imprimir.
- Texto literal a imprimir
- Caracteres de escape, como `\n`.

Normalmente el `formato` es una combinación de texto literal y códigos para escribir datos numéricos, que se van aplicando a los datos de la lista en el orden en que aparecen.

En los ejemplos siguientes se presentan algunos casos simples de utilización de esta instrucción. Para una comprensión más amplia se debe consultar la ayuda y documentación de MATLAB.

Ejemplos 4.5 (Uso de `fprintf`)

```
>> long = 32.067
>> fprintf('La longitud es de %12.6f metros \n',long)
La longitud es de    32.067000 metros
```

En este ejemplo, el formato se compone de:

- el texto literal `'La longitud es de '` (incluye los espacios en blanco),
- el código `%12.6f` que indica que se escriba un número (en este caso el valor de la variable `long`) ocupando un total de 12 espacios, de los cuales 6 son para las cifras decimales,
- el texto literal `' metros '` (también incluyendo los blancos),
- el carácter de escape `\n` que provoca un salto de línea.

```
>> x = sin(pi/5);
>> y = cos(pi/5);
>> fprintf('Las coordenadas del punto son x= %10.6f e y=%7.3f \n', x, y)
Las coordenadas del punto son x=    0.587785 e y=   0.809
```

Observamos que el primer código `%10.6f` se aplica al primer dato a imprimir, `x`, y el segundo código `%7.3f` al segundo, `y`.

```
>> k = 23; err = 0.00000314;
>> fprintf('En la iteracion k=%3i el error es %15.7e \n', k, err)
En la iteracion k= 23 el error es  3.1400000e-06
```

- el código `%3i` indica que se escriba un número entero ocupando un total de 3 espacios,
- el código `%15.7e` indica que se escriba un número en formato exponencial ocupando un total de 15 espacios, de los cuales 7 son para los dígitos a la derecha del punto decimal.

Ejercicio 4.6 (`LeeNumero.m`) (Uso de `input` y `fprintf`)

Escribir un *script* que:

- Lea un número del teclado.
 - Escriba en la pantalla “El numero leido es: + numero”.
-

```
numero = input('Escribe un numero: ');
fprintf('El numero leido es: %8.4f \n',numero);
```

4.2 Estructuras condicionales: if

Son los mecanismos de programación que permiten “romper” el flujo secuencial en un programa: es decir, permiten hacer una tarea si se verifica una determinada **condición** y otra distinta si no se verifica.

Evaluar si se verifica o no una condición se traduce, en programación, en averiguar si una determinada **expresión con valor lógico** da como resultado **verdadero** o **falso**

En casi todos los lenguajes de programación, este tipo de estructuras se implementan mediante una instrucción (o super-instrucción) denominada **if**, cuya sintaxis puede variar ligeramente de unos lenguajes a otros. En MATLAB, concretamente, toma las formas que se enuncian a continuación. La más sencilla es el condicional simple.

4.2.1 Estructura condicional simple (if-end)

En un condicional simple, si se verifica una determinada condición, se ejecuta un cierto grupo de instrucciones; en caso contrario no se ejecuta.

El diagrama de flujo correspondiente a esta estructura es el mostrado en la Figura 4.1: Si **expresión** toma el valor lógico **true**, se ejecutan las instrucciones del bloque **instrucciones** y, después, el programa se continúa ejecutando por la instrucción siguiente. Si, por el contrario, la **expresión** toma el valor lógico **false**, no se ejecutan las **instrucciones**, y el programa continúa directamente por la instrucción siguiente.

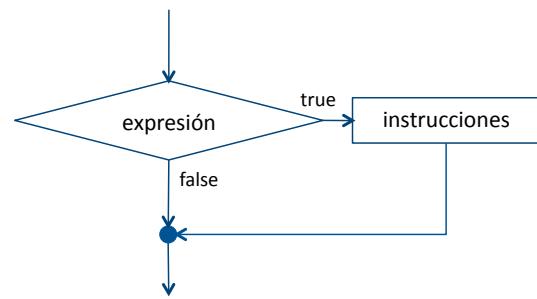


Figura 4.1: Diagrama de flujo de la estructura condicional simple.

En MATLAB esto se implementa con la super-instrucción: **if-end**.

```
instrucion-anterior
if expresion
    bloque-de-instrucciones
end
instrucion-siguiente
```

Ejemplo 4.7 (Uso de un condicional simple if-end)

Escribir una M-función que, dado $x \in \mathbb{R}$, devuelva el valor en x de la función definida a trozos

$$f(x) = \begin{cases} x + 1 & \text{si } x < -1, \\ 1 - x^2 & \text{si } x \geq -1. \end{cases}$$

Obsérvese que se ha incluído, como comentario, una breve descripción de la función.

```
function [fx] = mifun(x)
%
% v = mifun(x) devuelve el valor en x de la función
% f(x) = x+1 si x < -1
% f(x) = 1-x^2 si no
%
fx = x + 1;
if x > -1
    fx = 1 - x^2;
end
```

Ejemplo 4.8 (Maximo.m) (Uso de un condicional simple if-end)

Escribir un *script* que lea dos números x, y , e imprima en la pantalla el mayor de ellos junto con el mensaje: “El máximo es: + número”.

```
n1=input('Escribe un numero : ');
n2=input('Escribe otro numero : ');
mayor=n2;
if (n1 > n2)
    mayor=n1;
end
fprintf('El mayor es: \%6.2f \n ',mayor);
```

4.2.2 Estructura condicional doble (if-else-end)

En un condicional doble, si se verifica una determinada condición, se ejecuta un cierto grupo de instrucciones; en caso contrario, se ejecuta un grupo diferente de instrucciones.

El diagrama de flujo correspondiente a esta estructura es el mostrado en la Figura 4.2: Si **expresión** toma el valor lógico **true**, se ejecutan las instrucciones del bloque **instrucciones 1**. Si, por el contrario, la **expresión** toma el valor lógico **false**, se ejecuta el bloque de **instrucciones 2**.

En ambos casos, el programa continúa ejecutándose por la instrucción siguiente. Su implementación en MATLAB se lleva a cabo mediante la super-instrucción **if-else-end**.

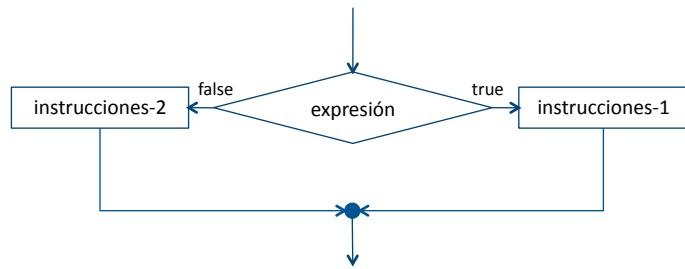


Figura 4.2: Diagrama de flujo de la estructura condicional doble.

```

instrucion-anterior
if expresion
    bloque-de-instrucciones-1
else
    bloque-de-instrucciones-2
end
instrucion-siguiente
  
```

Ejemplo 4.9 (`Ordena.m`) (Uso de un condicional doble **if-else-end**)

Escribir un *script* que lea dos números x, y , y los escriba en pantalla ordenados en orden ascendente.

```

n1 = input(' Escribe un numero : ');
n2 = input(' Escribe otro numero : ');
if (n1 < n2)
    fprintf(' %4i %4i \n', n1, n2)
else
    fprintf(' %4i %4i \n', n2, n1)
end
  
```

El siguiente ejercicio es una variante del anterior, en la que se considera una M-función en lugar de un *script*. Los datos necesarios (**n1** y **n2**) se obtienen ahora a través de los argumentos de entrada (en lugar de leerlos del teclado) y el resultado (los datos ordenados) se obtienen a través de la variable de salida **v** (en lugar de imprimirlo en la pantalla).

Ejemplo 4.10 (`Orden.m`) (Uso de un condicional doble **if-else-end**)

Escribir una M-función que reciba como argumentos de entrada dos números $x, y \in \mathbb{R}$ y devuelva un vector cuyas componentes sean los dos números ordenados en orden creciente.

```
function [v] = Orden(n1, n2)
%
% v = Orden(n1, n2) es un vector que contiene los dos
%         numeros n1 y n2 ordenados en orden creciente
%

if n1 > n2
    v = [n2, n1];
else
    v = [n1, n2];
end
```

4.2.3 Estructuras condicionales anidadas

Estas estructuras se pueden “anidar”, es decir, se puede incluir una estructura condicional dentro de uno de los bloques de instrucciones de uno de los casos de otra. Cada estructura `if` debe tener su correspondiente `end`.

Ejemplo 4.11 (`Cuadrante.m`) (Condicionales anidados)

Escribir una M-función que, dados dos números $x, y \in \mathbb{R}$, devuelva el número del cuadrante del plano OXY en el que se encuentra el punto (x, y) . Si el punto (x, y) está sobre uno de los ejes de coordenadas se le asignará el número 0.

```
function [n] = Cuadrante(x, y)
%
% n = Cuadrante(x, y) es el número del cuadrante del plano OXY
% en el que se encuentra el punto (x,y).
% 2 | 1   n = 0 si el punto está sobre uno de los ejes.
% -- --
% 3 | 4
%
if x*y == 0
    n = 0;
    return
end

if (x > 0)
    n = 4;
    if (y > 0)
        n = 1;
    end
else
    n = 3;
    if (y > 0)
        n = 2;
    end
end
```

En el programa `Cuadrante` se hace uso de la instrucción `return`, cuya descripción se encuentra en el apartado 4.4.

4.2.4 Estructura condicional múltiple (if-elseif-else-end)

Se pueden construir estructuras condicionales más complejas (con más casos). En MATLAB, estas estructuras se implementan mediante la versión más completa de la instrucción **if**.

En un condicional doble, si se verifica una determinada condición (Cond-1), se ejecuta un cierto grupo de instrucciones (Ins-1); en caso contrario, si se verifica una segunda condición (Cond-2), se ejecuta un grupo diferente de instrucciones (Ins-2); si tampoco esta segunda condición es cierta, se podría evaluar una tercera, etc. Finalmente, si ninguna de las condiciones anteriores ha sido cierta, se ejecutaría un grupo último grupo de instrucciones (Ins-n).

El diagrama de flujo correspondiente a esta estructura es el mostrado en la Figura 4.3: Si **expresión-1** toma el valor lógico **true**, se ejecutan las

instrucciones del bloque **instrucciones 1**. En caso negativo, si la **expresión-2** toma el valor lógico **true**, se ejecuta el bloque de **instrucciones 2**. Si ninguna de las expresiones ha tomado el valor **true**, se ejecutarían las instrucciones del bloque **instrucciones 3**. En todos los casos, el programa continúa ejecutándose por la instrucción siguiente a la estructura.

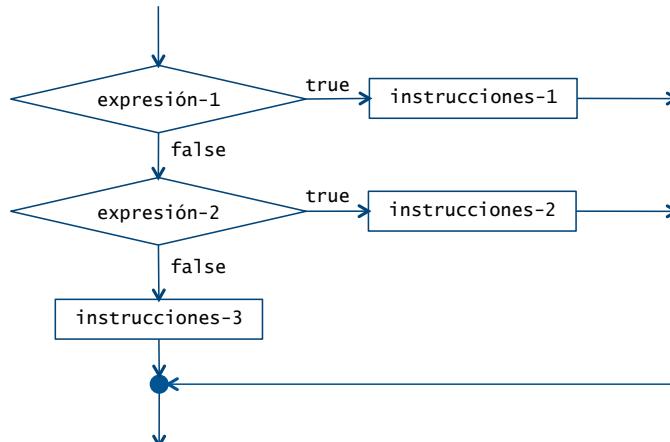


Figura 4.3: Diagrama de flujo de la estructura condicional múltiple.

```

instrucion-anterior
if expresion-1
  bloque-de-instrucciones-1
elseif expresion-2
  bloque-de-instrucciones-2
else
  bloque-de-instrucciones-3
end
instrucion-siguiente
  
```

Se pueden construir estructuras con más casos **elseif**, por ejemplo **if-elseif-elseif-else-end**. Es muy importante darse cuenta de que, en cualquier caso, se ejecutará **sólo uno** de los bloques de instrucciones.

La cláusula **else** (junto con su correspondiente bloque-de-instrucciones) puede no existir. En este caso es posible que no se ejecute ninguno de los bloques de instrucciones.

La implementación en MATLAB de esta estructura se lleva a cabo mediante la forma más general de la super-instrucción **if**.

Ejercicio 4.12 (Donde.m) (Uso de un condicional múltiple if-elseif-else-end)

Escribir una M-función que reciba como argumentos de entrada un número $x \in \mathbb{R}$ y los extremos de un intervalo $[a, b]$, y escriba en la pantalla si $x < a$, $x \in [a, b]$ o $x > b$.

```
function Donde(x, a, b)
%
% Donde(x, a, b) escribe en la pantalla la ubicacion
%     de x respecto del intervalo [a,b]:
%     - si x < a
%     - si a <= x < = b
%     - si x > b
%
if (a <= x) && (x <= b)
    fprintf('El punto %6.2f pertenece al intervalo [%6.2f,%6.2f]\n', x,a,b)
elseif x < a
    fprintf('El punto %6.2f esta a la izq. del intervalo [%6.2f,%6.2f]\n', x,a,b)
else
    fprintf('El punto %6.2f esta a la dcha. del intervalo [%6.2f,%6.2f]\n', x,a,b)
end
end
```

4.3 Estructuras de repetición o bucles

Vemos aquí los mecanismos de programación que permiten repetir un cierto grupo de instrucciones mientras que se verifique una determinada condición o bien un número predeterminado de veces.

4.3.1 Estructuras de repetición condicionada (while-end)

En una repetición condicionada, se repite la ejecución de un cierto bloque de instrucciones mientras que una determinada condición se siga verificando.

El diagrama de flujo de este tipo de repetición es el mostrado en la Figura 4.4. Su funcionamiento es el siguiente:

Al comienzo, se evalúa **expresión**. Si el resultado es **false**, se termina la repetición. En este caso, no se ejecuta el bloque de **instrucciones**. Si, por el contrario, el resultado de **expresión** es **true**, se ejecuta el bloque de **instrucciones**. Cuando se termina, se vuelve a evaluar la **expresión** y se vuelve a decidir.

Naturalmente, este mecanismo precisa que, dentro del bloque de **instrucciones** se modifique, en alguna de las repeticiones, el resultado de evaluar **expresión**. En caso contrario, el programa entraría en un *bucle infinito*. Llegado este caso, se puede detener el proceso pulsando la combinación de teclas **CTRL + C**.

Su sintaxis en MATLAB es la siguiente:

```
instrucion-anterior
while expresion
    bloque-de-instrucciones
end
instrucion-siguiente
```

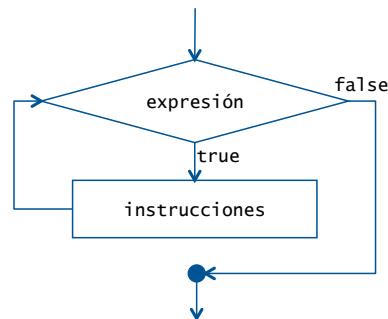
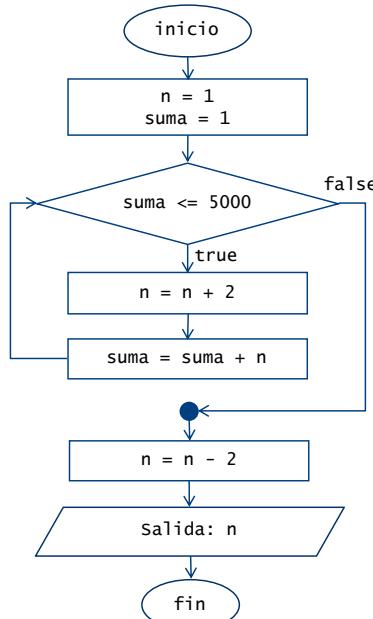


Figura 4.4: Diagrama de flujo de la repetición condicional

Ejemplo 4.13 (SumaImpares) (Uso de while)

Escribir una M-función que calcule y devuelva el mayor número impar n para el cual la suma de todos los números impares entre 1 y n es menor o igual que 5000.

```
function [n] = SumaImpares
%
% n = SumaImpares es el mayor numero impar tal que la suma de todos
% los impares desde 1 hasta n es <= 5000
%
n = 1;
suma = 1;
while suma <= 5000
    n = n + 2;
    suma = suma + n;
end
n = n - 2;
end
```



Ejercicio 4.14 Partiendo de la M-función **SumaImpares** del ejemplo anterior, escribe una M-función que reciba como argumento de entrada un número positivo M y devuelva el mayor entero n tal que la suma de todos los números naturales impares entre 1 y n sea menor o igual que M .

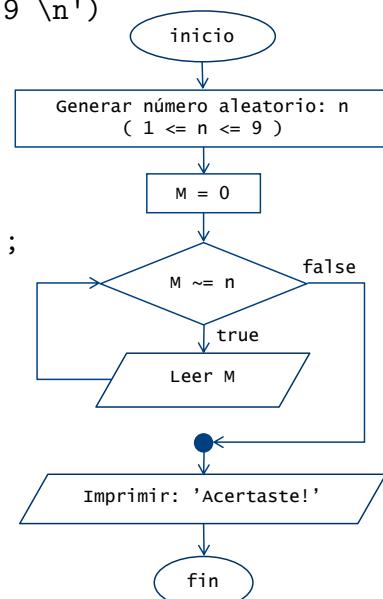
Ejemplo 4.15 (Adivina.m) (Uso de while)

Escribir un *script* que genere de forma aleatoria un número natural del 1 a 9 y pida repetidamente al usuario que escriba un número en el teclado hasta que lo acierte.

```
%-----
% Script Adivina
% Este script genera un numero aleatorio entre 1 y 9
% y pide repetidamente que se teclee un numero hasta acertar
%-----
%
n = randi(9);

fprintf('\n')
fprintf(' Tienes que acertar un numero del 1 al 9 \n')
fprintf(' Pulsa CTRL + C si te rindes ... \n')
fprintf('\n')

M = 0;
while M ~= n
    M = input('Teclea un numero del 1 al 9 : ');
end
beep
fprintf('\n')
fprintf(' ****Acertaste!!!! \n')
```



Observación: la orden **beep** utilizada en el código anterior genera un pitido (si los altavoces están activados).

Ejercicio 4.16 Partiendo del *script* **Adivina** del ejemplo anterior, escribe una M-función que reciba como argumento de entrada un número natural m , genere de forma aleatoria un número natural n entre 1 y m , y pida repetidamente al usuario que escriba un número en el teclado hasta que lo acierte.

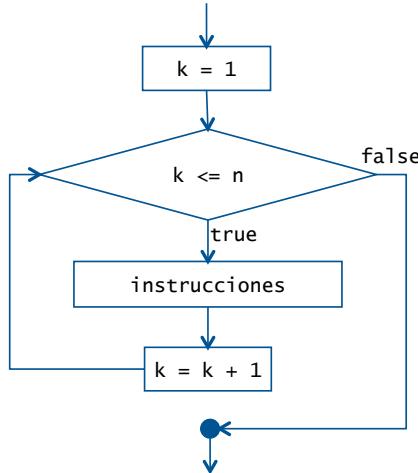
4.3.2 Estructuras de repetición indexada (for-end)

Este tipo de estructuras permite implementar bucles en los que se conoce *a priori* el número de veces que hay que repetir un cierto bloque de instrucciones.

En muchas ocasiones, las repeticiones de un bucle dependen en realidad de una variable entera cuyo valor se va incrementando hasta llegar a uno dado, momento en que se detienen las repeticiones. Esto sucede, especialmente, con los algoritmos que manipulan vectores y matrices, que es lo más habitual cuando se programan métodos numéricos.

En estas ocasiones, para implementar el bucle es preferible utilizar la estructura que se expone en esta sección: **bucle indexado**. En MATLAB (igual que en muchos otros lenguajes) este tipo de mecanismos se implementan mediante una instrucción denominada **for**.

Por ejemplo, el bucle representado mediante el diagrama siguiente



se puede implementar con las órdenes

```

instrucion-anterior
for k = 1 : n
  bloque-de-instrucciones
end
instrucion-siguiente
  
```

Se observa que, con esta instrucción, no hay que ocuparse ni de inicializar ni de incrementar dentro del bucle la variable-índice, **k**. Basta con indicar, junto a la cláusula **for**, el conjunto de valores que debe tomar. Puesto que es la propia instrucción **for** la que gestiona la variable-índice **k**, **está prohibido modificar su valor** dentro del bloque de instrucciones.

El conjunto de valores que debe tomar la variable-índice **k** tiene que ser de números enteros, pero no tiene que ser necesariamente un conjunto de números consecutivos. Son válidos, por ejemplo, los conjuntos siguientes:

```

for k = 0 : 2 : 20          % números pares de 0 a 20
for ind = 30 : -5 : 0        % números 30, 25, 20, 15, 10, 5, 0
for m = [2, 5, 4, 1, 7, 20] % los números especificados
  
```

Observación. Siempre que en un bucle sea posible determinar *a priori* el número de veces que se va a repetir el bloque de instrucciones, es preferible utilizar la instrucción **for**, ya que la instrucción **while** es más lenta.

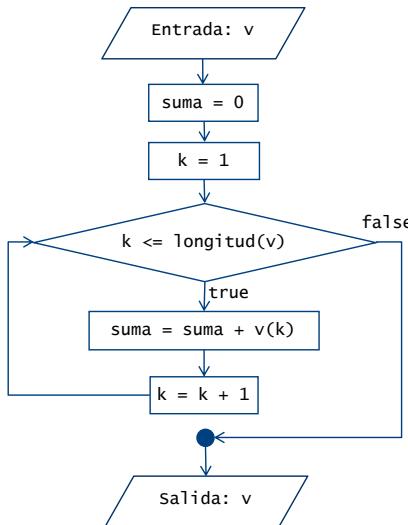
Ejemplo 4.17 (Uso de for)

Escribe una M-función

```
function [suma] = Sumav(v)
```

que reciba como argumento un vector v y devuelva la suma de sus componentes.

```
function [suma] = Sumav(v)
%
% Sumav(v) es la suma de las componentes del vector v
%
suma = 0;
for k = 1:length(v)
    suma = suma + v(k);
end
```



Observación: Lo que es importante observar y comprender aquí es que en este tipo de bucle, no hay que ocuparse explícitamente de la inicialización de la variable-índice **k** ni de su incremento en cada iteración. La instrucción **for** se ocupa de todo cuando se proporciona la lista de valores que debe tomar **k**.

Ejemplo 4.18 (Uso de for)

Escribir una M-función

```
function [y] = Traza(A)
```

que calcule la traza de la matriz cuadrada A .

```
function [tr] = Traza(A)
%
% Traza(A) es la suma de los elementos diagonales de la
%         matriz cuadrada A
%
[n,m] = size(A);
if n ~= m
    error('La matriz no es cuadrada')
end
tr = 0;
for k = 1:n
    tr = tr + A(k,k);
end
```

En el programa **Traza** se hace uso de la instrucción **error**, cuya descripción se encuentra en el apartado [4.4](#).

Los bucles y los condicionales se pueden anidar unos dentro e otros, como en los ejemplos que siguen:

Ejemplo 4.19 (Uso de for + if)

Escribe una M-función

```
function [vmax] = Mayor(v)
```

que reciba como argumento un vector v y devuelva la mayor de sus componentes.

```
function [vmax] = Mayor(v)
%
% Mayor(v) es el maximo de las componentes del vector v
%
vmax = v(1);
for k = 2:length(v)
    if v(k) > vmax
        vmax = v(k);
    end
end
```

Ejemplo 4.20 (Uso de for anidado)

Escribe una M-función

```
function [amax] = MayorM(A)
```

que reciba como argumento una matriz A y devuelva el máximo entre los valores absolutos de todas sus componentes

```
function [amax] = MayorM(A)
%
% MayorM(A) es el maximo de los valores absolutos de
% las componentes de la matriz A
%
amax = 0;
[nf, nc] = size(A);
for i = 1:nf
    for j = 1:nc
        aij = abs(A(i,j));
        if aij > amax
            amax = aij;
        end
    end
end
```

Ejercicio 4.21 Modifica la función **MayorM** del ejemplo anterior, para que devuelva, además, los números de la fila y columna en que se produce el máximo.

4.4 Rupturas incondicionales de secuencia

En ocasiones interesa finalizar la ejecución de un programa en algún punto “intermedio” del mismo, es decir, no necesariamente después de la última instrucción que aparece en el código fuente. Esto se hace mediante la orden

```
return
```

que devuelve el control a la ventana de comandos o bien al programa que llamó al actual.

En otras ocasiones es necesario interrumpir la ejecución de un bucle de repetición en algún punto interno del bloque de instrucciones que se repiten. Lógicamente, ello dependerá de que se verifique o no alguna condición. Esta interrupción puede hacerse de dos formas:

- Abandonando el bucle de repetición definitivamente.
- Abandonando la iteración en curso, pero comenzando la siguiente.

Las órdenes respectivas en MATLAB, tanto en un bucle **while** como en un bucle **for**):

```
break
continue
```

El funcionamiento de estas órdenes queda reflejado en los diagramas de flujo correspondientes (Figuras 4.5 y 4.6).

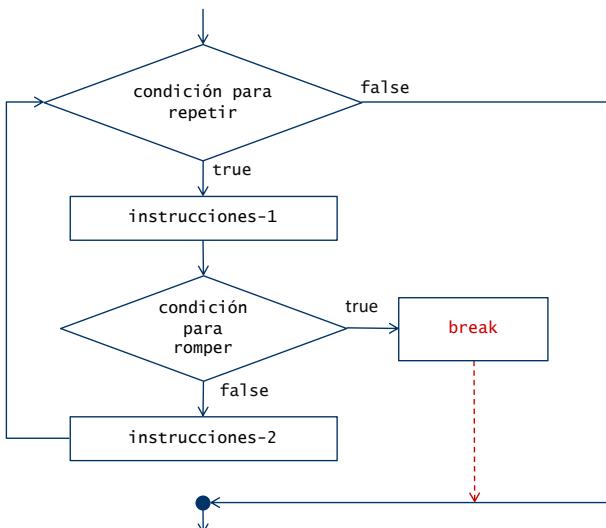


Figura 4.5: Ruptura de bucle **break**: Si en algún momento de un bucle de repetición (de cualquier tipo) se llega a una instrucción **break**, el ciclo de repetición se interrumpe inmediatamente y el programa continúa ejecutándose por la instrucción siguiente a la cláusula **end** del bucle. Si se trata de un bucle indexado **for**, la variable-índice del mismo conserva el último valor que haya tenido.

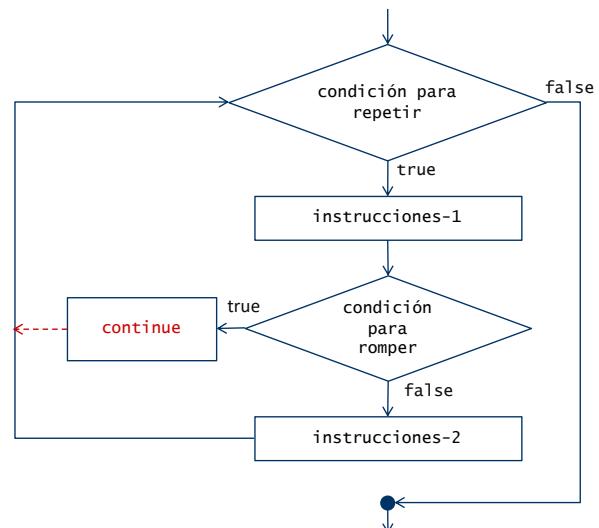


Figura 4.6: Ruptura de bucle **continue**: Si en algún momento de un bucle de repetición (de cualquier tipo) se llega a una instrucción **continue**, la iteración en curso se interrumpe inmediatamente, y se inicia la siguiente iteración (si procede).

4.5 Gestión de errores: warning y error

La instrucción siguiente se utiliza para alertar al usuario de alguna circunstancia no esperada durante la ejecución de un programa, pero no lo detiene. La orden

```
warning('Mensaje')
```

imprime `Warning`: Mensaje en la pantalla y continúa con la ejecución del programa. Por el contrario, la instrucción

```
error('Mensaje')
```

imprime `Mensaje` en la pantalla y detiene en ese punto la ejecución del programa.

Además de estas funciones y sus versiones más completas, MATLAB dispone de otras órdenes para gestionar la detección de errores. Véase [Error Handling](#) en la documentación.

4.6 Algunos consejos

Algunos comentarios generales sobre la escritura de programas:

- A los argumentos de salida de una M-función **siempre hay que darles algún valor**.
- Las instrucciones **for**, **if**, **while**, **end**, **return**, ... no necesitan llevar punto y coma al final, ya que no producen salida por pantalla. Sí necesitan llevarlo, en general, las instrucciones incluidas dentro.
- Las M-funciones no necesitan llevar **end** al final. De hecho, a nivel de este curso, que lo lleven puede inducir a error. Mejor no ponerlo.
- Todos los programas deben incluir unas líneas de comentarios que expliquen de forma sucinta su cometido y forma de utilización. En las M-funciones, estas líneas deben ser las siguientes a la instrucción **function**, ya que de esta manera son utilizadas por MATLAB como el texto de **help** de la función. En los *scripts* deben ser las primeras líneas del archivo.
- Es bueno, en la medida de lo posible y razonable, respetar la notación habitual de la formulación de un problema al elegir los nombres de las variables. Ejemplos
 - Llamar **S** a una suma y **P** a un producto, mejor que **G** o **N**.
 - Llamar **i**, **j**, **k**, **l**, **m**, **n** a un número entero, mejor que **x** ó **y**.
 - Llamar **T** o **temp** a una temperatura, mejor que **P**.
 - Llamar **e**, **err** ó **error** a un error, mejor que **vaca**.
- Cuando hay errores en un programa, **hay que leer con atención** los mensajes de error que envía MATLAB. Contienen mucha información, que en la mayoría de los casos es suficiente para ayudar a detectar el error.
- Hay que comprobar los programas que se hacen, dándoles valores a los argumentos para verificar que funciona bien en **todos los casos que puedan darse**. Esto forma parte del proceso de diseño y escritura del programa.
- Los espacios en blanco, en general, hacen el código más legible y por lo tanto más fácil de comprender y corregir. Se deben dejar uno o dos espacios entre el carácter **%** de cada línea de comentario y el texto del comentario (comparéñse los dos códigos siguientes).

```
function [Ma,Mg]=Medias(N)
%Ma=Medias(N) devuelve la media aritmetica
%de los numeros naturales menores o iguales
%que N
%[Ma,Me]=Medias(N) devuelve tambien la
%media
%geometrica
Ma=0;
Mg=1;
for i=1:N
Ma=Ma+i;
Mg=Mg*i;
end
N1=1/N;
Ma=Ma*N1;
Mg=Mg^(N1);
```

```
function [Ma,Mg]=Medias(N)
%-----
% Ma=Medias(N) devuelve la media
% aritmetica de los numeros
% naturales menores o iguales que N
% [Ma,Me]=Medias(N) devuelve tambien la
% media geometrica
%
Ma = 0;
Mg = 1;

for i=1:N
    Ma = Ma+i;
    Mg = Mg*i;
end

N1 = 1/N;
Ma = Ma*N1;
Mg = Mg^(N1);
```

4.7 Ejercicios

En este listado de ejercicios están incluidos, también, los ejemplos que se han ido mostrando a lo largo de este capítulo.

En todos estos ejercicios se debe, además, dibujar el diagrama de flujo del algoritmo correspondiente.

1. ([HolaMundo.m](#)) Escribir un *script* que pida el nombre al usuario, escriba un mensaje de saludo “Hola + nombre”, y escriba la fecha actual.
2. ([LeeNumero.m](#)) Escribir un *script* que lea un numero y escriba un mensaje “El numero leido es + numero” en una sola linea.
3. ([Maximo.m](#)) Usando la instrucción **if - end** (sin cláusula **else**), escribir un *script* que lea dos números e imprima el máximo entre ellos con un mensaje: “El maximo es + numero”.
4. ([Ordena.m](#)) Usando la instrucción **if - else - end**, escribir un *script* que lea dos números y los imprima en pantalla ordenados en orden creciente, con un mensaje “Los numeros ordenados son + numeros”.
5. ([Maxim.m](#)) Escribir una M-función

```
function [z] = Maxim(x, y)
```

que reciba como argumentos dos numeros y devuelva el mayor de ellos. Se trata del mismo algoritmo que el ejercicio 3, pero en versión M-función.

6. ([Orden.m](#)) Escribe una M-función

```
function [v] = Orden(x, y)
```

que reciba como argumentos dos numeros y devuelva un vector-fila cuyas dos componentes sean los argumentos de entrada ordenados en orden creciente. Mismo algoritmo del ejercicio 4, en versión M-función.

7. ([Cuadrante.m](#)) Escribe una M-función que reciba como argumentos las coordenadas de un punto (x, y) del plano y devuelva un número (1,2,3 o 4) que indique en qué cuadrante del plano se encuentra. El programa debe devolver 0 si el punto se encuentra sobre alguno de los ejes coordinados.

8. ([Donde.m](#)) Escribir una M-función

```
function Donde(x, a, b)
```

que reciba como argumentos de entrada un número $x \in \mathbb{R}$ y los extremos de un intervalo $[a, b]$, y escriba en la pantalla un mensaje indicando si $x < a$, $x \in [a, b]$ o $x > b$.

9. ([AreaTri.m](#)) Escribe una M-función

```
function [A] = AreaTri(x1, x2, x3)
```

que calcule el área de un triángulo a partir de las longitudes de sus lados:

$$A = \sqrt{p(p - x_1)(p - x_2)(p - x_3)}, \text{ con } p = \frac{x_1 + x_2 + x_3}{2}$$

La función debe emitir un error en los casos eventuales en que no se pueda calcular el área:

- a) Si alguna de las longitudes recibidas toma un valor menor o igual que cero.
- b) Si el radicando es negativo, lo cual indica que no existe ningún triángulo con esos lados.

10. (**SumaImparesM.m**) Escribir una M-función

```
function [n] = SumaImparesM(m)
```

que calcule y devuelva el mayor número impar n para el cual la suma de todos los números impares entre 1 y n es menor o igual que m .

11. (**Adivina.m**) Escribe una M-función

```
function Adivina(m)
```

que genere un número aleatorio entre (por ejemplo) 1 y 25, y le pida repetidamente al usuario que escriba un número en el teclado, hasta que lo acierte.

12. (**Mayor.m**) Escribe una M-función

```
function [vmax] = Mayor(v)
```

que reciba como argumento un vector v y devuelva la mayor de sus componentes.

13. (**Traza.m**) Escribe una M-función

```
function [w] = Traza(A)
```

que reciba como argumento una matriz A y devuelva su traza.

14. (**ProdMat.m**) Escribir una M-función

```
function [w] = ProdMat(A, v)
```

que reciba como argumentos de entrada una matriz A y un vector v y devuelva el vector $w = Av$, calculado mediante bucles de repetición. El programa debe limitarse a hacer el cálculo y devolverlo a través de la variable de salida w , sin imprimir nada en la pantalla, a excepción, eventualmente, de mensajes de error y/o de advertencia.

15. (**SimMat.m**) Escribir una M-función

```
function [B] = SimMat(A)
```

que reciba como argumento de entrada una matriz A y devuelva su transpuesta, calculada mediante bucles de repetición.

16. (**Grado1.m**) Escribe un *script* que pida al usuario los coeficientes, a y b , de una ecuación de primer grado $ax + b = 0$ e imprima su solución. El *script* debe asegurarse de que a es distinto de cero y, en caso contrario, volver a pedirlo.
17. (**Grado2.m**) Escribe un *script* que pida al usuario los coeficientes, a , b y c , de una ecuación de segundo grado $ax^2 + bx + c = 0$ e imprima sus soluciones, indicando cuál de los tres casos posibles se da:
 - Dos soluciones reales y distintas
 - Una solución real doble
 - Dos soluciones complejas conjugadas
18. (**Factorial.m**) Escribe una M-función

```
function [fact] = Factorial(n)
```

que reciba como argumento un numero entero positivo n y calcule su factorial por multiplicaciones sucesivas. La función debe escribir una tabla contenido, en cada linea, el valor de la iteración corriente y el valor calculado hasta dicha iteración.

19. (**Medias.m**) Escribe una M-función

```
function [Ma, Mg] = Medias(N)
```

que reciba como argumento un número N y devuelva dos valores: la media aritmética M_a y la media geométrica M_g de los números naturales menores o iguales que N .

20. (**Fibonacci.m**) La sucesión de Fibonacci surgió en el siglo XIII cuando el matemático italiano Leonardo Pisano Fibonacci estudió un problema relativo a la reproducción de los conejos. Esta sucesión $\{f_n\}_{n \geq 1}$ está definida para $n \geq 1$ por $f_{n+2} = f_{n+1} + f_n$, siendo $f_1 = f_2 = 1$. Escribir una M-función

```
function [FN] = Fibonacci(N)
```

que reciba como argumento un número entero $N > 2$ y devuelva el término N -ésimo de la sucesión de números de Fibonacci.

Para cada una de las series infinitas de los ejercicios 21 a 24 que siguen, cuyas sumas se indican en cada caso, se pide escribir una M-función que reciba un valor de x y un valor de N y calcule y devuelva un valor aproximado, S_N , de la suma de la serie, sumando sus $N + 1$ primeros términos (desde $i = 0$ hasta $i = N$):

$$S_N = \sum_{i=0}^N t_i \quad (t_i \text{ es el término general de la serie})$$

El programa debe, también, imprimir en pantalla una tabla, con una línea por iteración, con los valores de: (a) número de la iteración, i ; (b) valor del término sumado en esa iteración, t_i ; (c) valor de la suma parcial, S_i ; (d) error, $|S_i - f(x)|$.

21. (**SumaGeom.m**)

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots + x^n + \cdots = \sum_{i=0}^{\infty} x^i \quad \forall x \text{ tal que } |x| < 1$$

22. (**Sumaxcos.m**)

$$x \cos(x) = x - \frac{x^3}{2!} + \frac{x^5}{4!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n)!} + \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i)!} \quad \forall x \in \mathbb{R}$$

23. (**SumaExp.m**)

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad \forall x \in \mathbb{R}$$

24. (**SumaLog.m**)

$$\ln \sqrt{\left(\frac{1+x}{1-x}\right)} = x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots + \frac{x^{2n+1}}{2n+1} + \cdots = \sum_{i=0}^{\infty} \frac{x^{2i+1}}{2i+1} \quad \forall x \text{ tal que } |x| < 1$$

25. (**Contiene.m**) Escribir una M-función

```
function [bool] = Contiene(v,n)
```

que reciba como argumentos de entrada un vector v de números enteros y un número entero n , y devuelva el valor lógico **true** si alguna de las componentes de v es igual a n , y el valor lógico **false** en caso contrario. No es necesario que el programa compruebe que los datos de entrada son, efectivamente, números enteros.

26. (**DosCirculos.m**) Escribir una M-función

```
function [zona] = DosCirculos(x, y)
```

que devuelva, para cada par (x,y) , un valor **zona** de acuerdo con la regla siguiente:

- **zona=1** si el punto está en el interior o sobre la circunferencia $x^2 + y^2 = 1$.
- **zona=2** si el punto está en el interior o sobre la circunferencia $(x - 7)^2 + y^2 = 1$.
- **zona=0** si el punto es exterior a las dos circunferencias precedentes.

27. (**Separar.m**) Escribir una M-función

```
function [w1, w2] = Separar(v)
```

que reciba como argumento de entrada un vector v de números enteros y devuelva dos vectores $w1$ y $w2$ construidos como sigue:

- **w1** contiene las componentes pares y nulas de v .
- **w2** contiene las componentes impares de v .

28. (**SuperaMedia.m**) Escribir una M-función

```
function [w] = SuperaMedia(v)
```

que reciba como argumento de entrada un vector **v** y proporcione como salida otro vector **w** formado por las componentes de **v** que tengan un valor mayor o igual que la media aritmética de todas las componentes de **v**.

29. (**EsPrimo.m**) Escribe una M-función

```
function [bool] = EsPrimo(n)
```

que reciba como argumento de entrada un número entero **n** positivo y devuelva el valor lógico **true** si **n** es primo y **false** si no lo es. El programa debe comprobar que el valor de **n** recibido es realmente un entero positivo y, en caso contrario, emitir un mensaje de “warning” y devolver el valor lógico **false**.

Para decidir si un número divide a otro se puede utilizar la función intrínseca de MATLAB **mod(x,y)** que devuelve el resto de la división entera **x/y**. Para decidir si un número es entero se puede utilizar la función **round(x)** que devuelve el número entero más próximo a **x**.

30. (**CambioSigno.m**) Escribir una M-función

```
function [index] = CambioSigno(v)
```

que reciba como argumento de entrada un vector **v** y proporcione como salida un entero **index** definido como sigue:

- Si todas las componentes del vector **v** son del mismo signo (todas positivas o todas negativas), entonces **index** =0.
- Si no todas las componentes del vector **v** son del mismo signo, **index** es el menor valor que verifica $v(index) * v(index + 1) \leq 0$.

Por ejemplo, si $v = (2, 5.4, 6.12, 7)$, entonces debe ser **index** = 0; si $v = (3.2, 1.47, -0.9, 0.75)$, entonces debe ser **index** = 2.

31. (**EsBisiesto.m**) Escribir una M-función

```
function [bool] = EsBisiesto(a)
```

que determine si un año dado es bisiesto o no, teniendo en cuenta que son años bisiestos los múltiplos de 4, excepto los que son también múltiplos de 100 pero no lo son de 400.

32. (**FechaValida.m**) Escribir una M-función

```
function [bool] = FechaValida(dia,mes,anno)
```

que permita validar una fecha **dia-mes-anno**. Debe tenerse en cuenta la posibilidad de que sea un año bisiesto. Para comprobar si lo es, se puede hacer uso de la función **EsBisiesto** anterior.

33. (**Divisores.m**) Escribir una M-función

```
function [v]=Divisores(N)
```

que devuelva, como elementos del vector v, los divisores del numero N.

34. (**Perfecto.m**) Escribir una M-función

```
function [bool] = Perfecto(N)
```

Escribir un programa que determine si un número natural N dado es perfecto o no. Se dice que un número es perfecto cuando es igual a la suma de sus divisores, excluido él mismo. Por ejemplo 6 es perfecto, ya que $6 = 1 + 2 + 3$.

35. (**Goldbach.m**) La conjetura de Goldbach es uno de los problemas abiertos más antiguos de la Teoría de Números. Dice que: “Todo número par mayor que 2 se puede obtener como suma de dos números primos”.

Escribir un programa

```
function [p] = Goldbach(N)
```

que descomponga un número par N dado en la suma de dos números primos p=[p1,p2] (puede ser el mismo primo repetido)

36. (**Descomponer.m**) Escribir un programa que reciba como argumento un número entero y lo descomponga en unidades, decenas, centenas, etc. (No se pueden usar cadenas de caracteres para resolver el problema.)

```
function [d]=Descomponer(N)
```

El argumento de salida d será un vector fila, de longitud igual al número de dígitos de N conteniendo su descomposición. Por ejemplo, si N=36201, debe ser d=[1,0,2,6,3].

37. (**Horner.m**) Desde el punto de vista de la economía de operaciones, la manera óptima de evaluar el polinomio

$$p(x) = c_{n+1} + c_n x + c_{n-1} x^2 + \cdots + c_2 x^{n-1} + c_1 x^n$$

es utilizando la siguiente expresión:

$$p(x) = c_{n+1} + x \left(c_n + x \left(c_{n-1} + \dots x(c_2 + c_1 x) \dots \right) \right)$$

Escribir una M-función

```
function [y] = Horner(c, x)
```

que evalúe el polinomio de coeficientes c en el punto x (x puede ser un vector y en ese caso y debe ser un vector de las mismas dimensiones).

38. (**derpol.m**) En MATLAB un polinomio se representa mediante un vector que contiene los valores de sus coeficientes: el polinomio $p(x) = a_1x^n + a_2x^{n-1} + \dots + a_nx + a_{n+1}$ se representa mediante el vector-fila $c = (a_1, a_2, \dots, a_n, a_{n+1})$. Escribir una M-función

```
function [dc] = derpol(c)
```

que recibe como argumento de entrada un vectores, c conteniendo los coeficientes de un polinomio, y devuelva, en el vector dc , los coeficientes de su derivada.

39. (**sumapol.m**) En MATLAB un polinomio se representa mediante un vector que contiene los valores de sus coeficientes: el polinomio $p(x) = a_1x^n + a_2x^{n-1} + \dots + a_nx + a_{n+1}$ se representa mediante el vector-fila $c = (a_1, a_2, \dots, a_n, a_{n+1})$. Escribir una M-función

```
function [c] = sumapol(c1, c2)
```

que recibe como argumentos de entrada dos vectores, c_1 y c_2 (de longitudes desconocidas que pueden ser distintas) representando dos polinomios, y devuelva, en el vector c , los coeficientes del polinomio suma de los dos anteriores.

40. (**Correos.m**) Escribir un *script* que calcule el coste de enviar un paquete de correos, en base a la siguiente tabla de precios:

Tipo de envío	Peso (0-1 kg)	Peso (1-5 kg)	Peso (5-25 kg)
Tierra	1.50 €	1.50 € + 1 € adicional por cada kg o fracción de kg, a partir de 1 kg de peso.	5.50 € + 0.60 € adicionales por cada kg o fracción de kg, a partir de 5 kg de peso.
Aire	3 €	3 € + 1 € adicional por cada kg o fracción de kg, a partir de 1 kg de peso.	10.20 € + 1.20 € adicionales por cada kg o fracción de kg, a partir de 5 kg de peso.
Urgente	18 €	18 € + 6 € adicionales por cada kg o fracción de kg, a partir de 1 kg de peso.	No se realizan envíos urgentes de paquetes con un peso superior a 5 kg.

El programa pedirá por pantalla los datos *peso* y *tipo de envío*. En el caso de un envío por tierra o aire de más de 25 kg, el programa escribirá en pantalla el mensaje “No se realizan repartos de más de 25 kg”. En el caso de envíos urgentes de más de 5 kg, el programa escribirá en pantalla el mensaje “No se realizan repartos urgentes de más de 5 kg”

41. (**Camino.m**) Escribir una M-función

```
function [long] = Camino(x, y)
```

que reciba como argumentos dos vectores x e y conteniendo respectivamente las abscisas y las ordenadas de una sucesión de puntos del plano, y devuelva la longitud del camino que se forma uniendo dichos puntos en el orden dado.

42. (**CaminoOrd.m**) Escribir una M-función

```
function [long] = CaminoOrd(x, y)
```

que reciba como argumentos dos vectores x e y conteniendo respectivamente las abscisas y las ordenadas de una sucesión de puntos del plano, y devuelva la longitud del camino que se forma uniendo dichos puntos en orden creciente de abscisas.

Este ejercicio es como el anterior 41, pero es preciso previamente ordenar los puntos de forma que sus abscisas estén en orden creciente. Para ello hay que usar la orden **sort** cuyo uso se debe consultar en el **Help** de MATLAB.

43. (**PuntoEnTriangulo.m**) Dado un triángulo de vértices X , Y y Z , y un punto P del plano, se verifica que: *El punto P pertenece al triángulo XYZ si y sólo si $A = A_1 + A_2 + A_3$,* donde

A = área del triángulo XYZ

A_1 = área del triángulo XYP

A_2 = área del triángulo YZP

A_3 = área del triángulo ZXP .

Escribir una M-función,

```
function [bool] = PuntoEnTriangulo(X, Y, Z, P)
```

que devuelva $\text{bool} = \text{true}$ si el punto P pertenece al triángulo XYZ y $\text{bool} = \text{false}$ si no. **Indicación:** La función de MATLAB **polyarea** sirve para calcular el área de un polígono plano (ver Help).

5

Resolución de sistemas lineales



5.1 Los operadores de división matricial

Sea A una matriz cualquiera y sea B otra matriz con el mismo número de filas que A ¹. Entonces, la “solución” del “sistema” (en realidad un sistema lineal por cada columna de B)

$$AX = B$$

se calcula en MATLAB mediante el operador no estándar “\” denominado *backward slash*, (barra inversa en español)

```
X = A \ B  
X = mldivide(A, B)      (equivalente a lo anterior)
```

Hay que pensar en él como el operador de “división matricial por la izquierda” ($A \ B \equiv A^{-1} B$). De forma similar, si C es una matriz con el mismo número de columnas que A , entonces la solución del sistema

$$XA = C$$

se obtiene en MATLAB mediante el operador “/” (“división matricial por la derecha”)

```
X = C / A  
X = mrdivide(C, A)      (equivalente a lo anterior)
```

Estos operadores se aplican incluso si A no es una matriz cuadrada. Lo que se obtiene de estas operaciones es, lógicamente, distinto según sea el caso.

De forma resumida, si A es una matriz cuadrada e y es un vector columna, $c = A \ y$ es la solución del sistema lineal $Ac = y$, obtenida por diferentes algoritmos, en función de las características de la matriz A (diagonal, triangular, simétrica, etc.). Si A es singular o mal condicionada, se obtendrá un mensaje de alerta (*warning*). Si A es una matriz rectangular, $A \ y$ devuelve una solución de mínimos cuadrados del sistema $Ac = y$.

¹A menos que A sea un escalar, en cuyo caso $A \ B$ es la operación de división elemento a elemento, es decir $A ./ B$.

Para una descripción completa del funcionamiento de estos operadores, así como de la elección del algoritmo aplicado, véase la documentación de MATLAB correspondiente, tecleando en la ventana de comandos

```
doc mldivide
```

Ejemplo 5.1 (Uso del operador \)

Calcular la solución del sistema (compatible determinado)

$$\begin{cases} 2x_1 + x_2 - x_3 = -1 \\ 2x_1 - x_2 + 3x_3 = -2 \\ 3x_1 - 2x_2 = 1 \end{cases}$$

```
>> A = [2, 1, -1; 2, -1, 3; 3, -2, 0];
>> b = [-1; -2; 1];
>> x = A\b
x =
-0.3636
-1.0455
-0.7727
```

Como comprobación calculamos el residuo que, como es de esperar, no es exactamente nulo, debido a los errores de redondeo:

```
>> A*x - b
ans =
1.0e-15 *
-0.4441
0
0
```

Ejemplo 5.2 (Uso del operador \)

Calcular la solución del sistema (compatible indeterminado)

$$\begin{cases} x_1 + x_2 + x_3 = 1 \\ 2x_1 - x_2 + x_3 = 2 \\ x_1 - 2x_2 = 1 \end{cases}$$

```
>> A = [1, 1, 1; 2, -1, 1; 1, -2, 0];
>> b = [ 1; 2; 1];
>> x = A\b
Warning: Matrix is singular to working precision.
x =
    NaN
    NaN
    NaN
```

Ejemplo 5.3 (Uso del operador \)

Calcular la solución del sistema (incompatible)

$$\begin{cases} 2x + 2y + t &= 1 \\ 2x - 2y + z &= -2 \\ x - z + t &= 0 \\ -4x + 4y - 2z &= 1 \end{cases}$$

```
>> A = [2, 2, 0, 1; 2, -2, 1, 0; 1, 0, -1, 1; -4, 4, -2, 0];
>> b = [ 1; -2; 0; 1];
>> x = A\b
Warning: Matrix is singular to working precision.
x =
    NaN
    NaN
    -Inf
    -Inf
```

5.2 Determinante. ¿Cómo decidir si una matriz es singular?

El determinante de una matriz cuadrada A se calcula con la orden

`det(A)`

Este cálculo se hace a partir de la factorización LU de la matriz A , ya que se tiene $\det(L) = 1$, y $\det(A) = \det(U)$, que es el producto de sus elementos diagonales.

El uso de `det(A) == 0` para testar si la matriz A es singular sólo es aconsejable para matrices de pequeño tamaño y elementos enteros también de pequeña magnitud.

El uso de `abs(det(A)) < epsilon` tampoco es recomendable ya que es muy difícil elegir el `epsilon` adecuado (véase el Ejemplo 5.4).

Lo aconsejable para testar la singularidad de una matriz es usar su **número de condición** `cond(A)`.

Ejemplo 5.4 (Matriz no singular, con det. muy pequeño, bien condicionada)

Se considera la matriz 10×10 siguiente, que no es singular, ya que es múltiplo de la identidad,

```
>> A = 0.0001 * eye(10);
>> det(A)
ans =
1.0000e-40
```

Vemos que su determinante es muy pequeño. De hecho, el test `abs(det(A)) < epsilon` etiquetaría esta matriz como singular, a menos que se eligiera `epsilon` extremadamente pequeño. Sin embargo, esta matriz no está mal condicionada:

```
>> cond(A)
ans =
1
```

Ejemplo 5.5 (Matriz singular, con `det(A)` muy grande)

Se considera la matriz 13×13 construida como sigue, que es singular y de diagonal dominante:

```
>> A = diag([24, 46, 64, 78, 88, 94, 96, 94, 88, 78, 64, 46, 24]);
>> S = diag([-13, -24, -33, -40, -45, -48, -49, -48, -45, -40, -33, -24], 1);
>> A = A + S + rot90(S,2);
```

La matriz que se obtiene es

24	-13	0	0	0	0	0	0	0	0	0	0	0
-24	46	-24	0	0	0	0	0	0	0	0	0	0
0	-33	64	-33	0	0	0	0	0	0	0	0	0
0	0	-40	78	-40	0	0	0	0	0	0	0	0
0	0	0	-45	88	-45	0	0	0	0	0	0	0
0	0	0	0	-48	94	-48	0	0	0	0	0	0
0	0	0	0	0	-49	96	-49	0	0	0	0	0
0	0	0	0	0	0	-48	94	-48	0	0	0	0
0	0	0	0	0	0	0	-45	88	-45	0	0	0
0	0	0	0	0	0	0	0	-40	78	-40	0	0
0	0	0	0	0	0	0	0	0	-33	64	-33	0
0	0	0	0	0	0	0	0	0	-24	46	-24	
0	0	0	0	0	0	0	0	0	0	-13	24	

A es singular (la suma de todas sus filas da el vector nulo). Sin embargo, el cálculo de su determinante con la función `det` da

```
>> det(A)
ans =
1.0597e+05
```

cuando debería dar como resultado cero! Esta (enorme) falta de precisión es debida a los errores de redondeo que se cometan en la implementación del método *LU*, que es el que MATLAB usa para calcular el determinante. De hecho, vemos que el número de condición de A es muy grande:

```
>> cond(A)
ans =
2.5703e+16
```

5.3 La factorización LU

La factorización *LU* de una matriz se calcula con MATLAB con la orden

 $[L, U, P] = \text{lu}(A)$

El significado de los distintos argumentos es el siguiente:

L es una matriz triangular inferior con unos en la diagonal.

U es una matriz triangular superior.

P es una matriz de permutaciones, que refleja los intercambios de filas realizados durante el proceso de eliminación gaussiana sobre las filas de la matriz A , al aplicar la técnica del pivot.

LU=PA es la factorización obtenida.

Entonces, para calcular la solución del sistema lineal de ecuaciones

$$Ax = b$$

utilizando la factorización anterior sólo hay que plantear el sistema, equivalente al anterior,

$$PAx = Pb \iff LUx = Pb \iff \begin{cases} Lv = Pb \\ Ux = v \end{cases}$$

El primero de estos dos sistemas se resuelve fácilmente mediante un algoritmo de bajada, ya que la matriz del mismo es triangular inferior. Una vez calculada su solución, v , se calcula x como la solución del sistema $Ux = v$, que se puede calcular mediante un algoritmo de subida, al ser su matriz triangular superior.

Ejercicio 5.6 Escribir una M-función `function [x] = Bajada(A, b)` para calcular la solución x del sistema $Ax = b$, siendo A una matriz cuadrada triangular inferior.

Algoritmo de bajada

n = dimensión de A

Para cada $i = 1, 2, \dots, n$,

$$x_i = \frac{1}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j \right)$$

Fin

Para comprobar el funcionamiento del programa, construir una matriz 20×20 (por ejemplo) y un vector columna b de números generados aleatoriamente (con la función `rand` o bien con `randi`) y luego extraer su parte triangular inferior con la función `tril`(*). (Consultar en el help de MATLAB la utilización de estas funciones).

(*) Aunque, en realidad esto no es necesario. Obsérvese que en el programa `Bajada` que sigue, no se utiliza la parte superior de la matriz A .

```

function [x] = Bajada(A, b)
%
% Bajada(A, b) es la solucion del sistema lineal de
%     matriz triangular inferior Ax = b
%
%----- tolerancia para el test de singularidad
tol = 1.e-10;
%----- inicializaciones
n = length(b);
x = zeros(n,1);
for i = 1:n
    Aii = A(i,i);
    if abs(Aii) < tol
        warning(' La matriz A es singular ')
    end
    suma = 0;                                % en version vectorial, sin usar for,
    for j = 1:i-1                            % se puede calcular la suma con:
        suma = suma + A(i,j)*x(j);          %
    end                                         % suma = A(i,1:i-1)*x(1:i-1)
    x(i) = (b(i) - suma)/Aii;
end

```

Ejercicio 5.7 Escribir una M-función `function [x] = Subida(A, b)` para calcular la solución `x` del sistema $Ax = b$, siendo A una matriz cuadrada triangular superior.

Algoritmo de subida

n = dimensión de A

Para cada $i = n, \dots, 2, 1$

$$x_i = \frac{1}{A_{ii}} \left(b_i - \sum_{j=i+1}^n A_{ij} x_j \right)$$

Fin

Para comprobar el funcionamiento del programa, construir una matriz A y un vector b de números generados aleatoriamente (como en el ejercicio anterior) y luego extraer su parte triangular inferior con la función `triu`.

Ejercicio 5.8 Escribir una M-función `[x, res] = LU(A, b)` que calcule la solución `x` del sistema $Ax = b$ y el residuo `res` = $Ax - b$ siguiendo los pasos siguientes:

- Calcular la factorización LU mediante la función `lu` de MATLAB
- Calcular la solución del sistema $Lv = Pb$ mediante la M-función **Bajada**
- Calcular la solución del sistema $Ux = v$ mediante la M-función **Subida**

Utilizar la M-función `LU` para calcular la solución del sistema

$$\begin{pmatrix} 1 & 1 & 0 & 3 \\ 2 & 1 & -1 & 2 \\ 3 & -1 & -1 & 2 \\ -1 & 2 & 3 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

Observación: Llamamos `LU` (con mayúsculas) a esta función para que no se confunda con la función `lu` (minúsculas) de MATLAB.

Ejercicio 5.9 Las matrices de Hilbert definidas por $H_{ij} = \frac{1}{i + j - 1}$ son un ejemplo notable de matrices mal condicionadas, incluso para dimensión pequeña.

Utilizar la M-función `LU` para resolver el sistema $Hx = b$, donde H es la matriz de Hilbert de dimensión $n = 15$ (por ejemplo) y b es un vector de números generados aleatoriamente. Comprobar que el residuo es grande. Comprobar que la matriz H está mal condicionada calculando su número de condición.

La función `hilb(n)` construye la matriz de Hilbert de dimensión n .

5.4 La factorización de Cholesky

La factorización de Cholesky de una matriz simétrica se calcula en MATLAB con alguna de las órdenes

```
U = chol(A) % se obtiene una matriz triang. superior
L = chol(A, 'lower') % se obtiene una matriz triang. inferior
```

de manera que se tiene $U^t U = A$ con la primera opción y $L L^t = A$ con la segunda.

Cuando se usa en la forma `chol(A)`, la función `chol` sólo usa la parte triangular superior de la matriz A para sus cálculos, y asume que la parte inferior es la simétrica. Es decir, que si se le pasa una matriz A que no sea simétrica no se obtendrá ningún error. Por el contrario, si se usa en la forma `chol(A, 'lower')`, sólo se usará la parte triangular inferior de A , asumiendo que la parte superior es la simétrica.

La matriz A tiene que ser definida positiva. Si no lo es, se obtendrá un error.

Una vez calculada la matriz L , para calcular la solución del sistema lineal de ecuaciones

$$Ax = b$$

utilizando la factorización anterior sólo hay que plantear el sistema, equivalente al anterior,

$$Ax = b \iff LL^t x = b \iff \begin{cases} L v = b \\ L^t x = v \end{cases}$$

que, de nuevo, se resuelven fácilmente utilizando sendos algoritmos de bajada + subida.

Si, por el contrario, se calcula la factorización en la forma $U^t U = A$, entonces se tiene

$$Ax = b \iff U^t U x = b \iff \begin{cases} U^t v = b \\ U x = v \end{cases}$$

que se resuelve aplicando en primer lugar el algoritmo de subida y a continuación el de bajada.

Ejercicio 5.10 Escribir una M-función `[x, res] = CHOL(A, b)` que calcule la solución x del sistema con matriz simétrica definida positiva $Ax = b$ y el residuo $res = Ax - b$ siguiendo los pasos siguientes:

- Calcular la matriz L de la factorización de Cholesky LL^t de A mediante la función `chol` de MATLAB
- Calcular la solución del sistema $Lv = b$ mediante la M-función **Bajada**
- Calcular la solución del sistema $L^t x = v$ mediante la M-función **Subida**

Para comprobar el funcionamiento del programa, se puede generar alguna de las matrices definida positivas siguientes:

- `M = gallery('moler', n)`
- `T = gallery('toeppd', n)`
- `P = pascal(n)`

Observación: Por la misma razón que en el ejercicio 5.8, llamamos **CHOL** (mayúsculas) a esta función para que no se confunda con la función `chol` (minúsculas) de MATLAB.

5.5 Resolución de sistemas con características específicas

En la resolución numérica de sistemas lineales, sobre todo si son de grandes dimensiones, resulta imprescindible aprovechar propiedades concretas que pueda tener la matriz del sistema para utilizar un método *ad hoc* que reduzca el número de operaciones a realizar y, en consecuencia, el tiempo de cálculo y los errores de redondeo.

MATLAB dispone de una gran cantidad de funciones dedicadas a este problema. Para detalles se debe consultar la documentación.

Sin entrar en muchos detalles, la función `linsolve` proporciona un poco más de control que el operador `\` sobre el método de resolución a utilizar. La orden

```
x = linsolve(A, b)
```

resuelve el sistema lineal $\mathbf{Ax} = \mathbf{b}$ por factorización LU si \mathbf{A} es una matriz cuadrada y regular, y por factorización QR si no.

La orden

```
x = linsolve(A, b, opts)
```

resuelve el sistema lineal $\mathbf{Ax} = \mathbf{b}$ por el método que resulte más apropiado, dadas las propiedades de la matriz \mathbf{A} , que se pueden especificar mediante el argumento opcional **opts** (véase la documentación).

5.6 Matrices huecas (*sparse*)

En muchas aplicaciones prácticas, aparecen matrices que sólo tienen un pocos elementos distintos de cero. Estas matrices se denominan *huecas* (*sparse* en la terminología en inglés, *creuse* en la terminología francesa). Por ejemplo, en simulación de circuitos y en implementación de métodos de elementos finitos, aparecen matrices que tienen menos de un 1% de elementos no nulos.

Para tales matrices, es un enorme desperdicio de memoria almacenar todos sus ceros, y es un enorme desperdicio de tiempo realizar operaciones aritméticas con ceros. Por esta razón se han desarrollado un buen número de estrategias de almacenamiento que evitan el uso de memoria para guardar ceros, y se han adaptado los algoritmos para no realizar operaciones con elementos nulos.

MATLAB ofrece una estrategia general para este tipo de matrices: el almacenamiento como matriz **sparse**, consistente en almacenar sólo los elementos no nulos, junto con su posición en la matriz, esto es, su fila y su columna.

Ejemplo 5.11 (Almacenamiento *sparse*)

En forma *sparse*, de la matriz 5×5

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{matrix}$$

se almacenarían sólo los siguientes datos

$$\begin{matrix} 2 & 1 & 1 \\ 3 & 5 & -1 \\ 5 & 3 & 3 \end{matrix}$$

Para crear matrices *sparse*, MATLAB tiene la función **sparse**, que se puede utilizar de varias formas.

La orden

```
AS = sparse(A)
```

convierte la matriz *llena* **A** en la matriz *sparse* **AS**.

La orden

```
A = full(AS)
```

hace la operación contraria, i.e., convierte la matriz *sparse* **AS** en la matriz *llena* **A**.

Ejemplo 5.12

```
>> A = [0, 0, 1; 0, 2, 0; -1, 0, 0];
>> AS = sparse(A)
AS =
    (3,1)      -1
    (2,2)       2
    (1,3)       1

>> A = full(AS)
A =
    0     0     1
    0     2     0
   -1     0     0
```

La orden

```
AS = sparse(i, j, s)
```

crea la matriz *sparse* **AS** cuyos elementos no nulos son **AS(i(k), j(k)) = s(k)**

Ejemplo 5.13

```
>> i = [2, 3, 5]
>> j = [1, 5, 3]
>> s = [1, -1, 3]
>> AS = sparse(i, j, s)
AS =
(2,1)      1
(5,3)      3
(3,5)     -1
```

Las operaciones con matrices *sparse* funcionan, con MATLAB, con la misma sintaxis que para las matrices llenas. En general, las operaciones con matrices llenas producen matrices llenas y las operaciones con matrices *sparse* producen matrices *sparse*. Las operaciones mixtas generalmente producen matrices *sparse*, a menos que el resultado sea una matriz con alta densidad de elementos distintos de cero. Por ejemplo, con la misma matriz **AS** del ejemplo anterior:

Ejemplo 5.14

```
>> AS(5,3)
ans =
(1,2)      1
(5,3)     -1
(3,5)      3
```

```
>> AS'
ans =
(1,2)      1
(5,3)     -1
(3,5)      3
```

```
>> 10*AS
ans =
(2,1)      10
(5,3)      30
(3,5)     -10
```

```
>> AS(3,5) = 7
AS =
(2,1)      1
(5,3)      3
(3,5)      7
```

```
>> AS(1,4) = pi
AS =
(2,1)      1.0000
(5,3)      3.0000
(1,4)      3.1416
(3,5)      7.0000
```


6

Operaciones de lectura y escritura con ficheros en MATLAB



Hasta ahora se ha visto:

- Cómo leer valores que el usuario escribe en el teclado (orden `input`).
- Cómo imprimir información en la ventana de comandos de MATLAB (órdenes `disp` y `fprintf`).

En este capítulo veremos operaciones avanzadas de lectura y escritura, concretamente, que operan sobre el contenido de un fichero.

6.1 Ficheros de texto y ficheros binarios

Existen muchos tipos distintos de ficheros, que normalmente se identifican por distintas extensiones;

.txt	.dat	.m	.mat
.jpg	.xls	.pdf	etc.

Una de las cosas que diferencia unos ficheros de otros es el hecho de que su contenido sea o no legible con un editor de texto plano, también llamado **editor ascii**¹. El editor de MATLAB es un editor *ascii* y también lo es el Bloc de Notas de Windows y cualquier editor orientado a la programación. Word no es un editor *ascii*.

Desde el punto de vista anterior, existen dos tipos de ficheros:

- Ficheros **formateados o de texto**: son aquéllos cuyo contenido está formado por texto plano. Esto significa que su contenido se reduce a una secuencia de caracteres (los permitidos en el código de representación de caracteres que se esté utilizando: ASCII, UTF, Unicode, ...) y que no contienen ningún tipo de información sobre el aspecto del texto, como tipos de letra, tamaño, grosor, color, etc. Los códigos de programa en cualquier lenguaje de programación se escriben normalmente en ficheros de texto.
- Ficheros **no formateados ó binarios**: contienen información de cualquier tipo codificada en binario, es decir, tal y como se almacena en la memoria del ordenador, utilizando ceros y unos. Ejemplos de ficheros binarios son los que almacenan texto formateado (por ejemplo,

¹ASCII = American Standard Code for Information Interchange (Código Estándar Estadounidense para el Intercambio de Información).

de Word), ficheros que contienen libros de hojas de cálculo (Excel), ficheros de imágenes, ficheros que contienen programas ejecutables (es decir, escritos en código-máquina), etc. Sólo se pueden utilizar con programas adecuados para cada fichero.

6.2 Lectura y escritura en ficheros automática: órdenes save y load

6.2.1 Grabar en fichero el espacio de trabajo: orden save

La utilización más sencilla de la orden **save** es para guardar en un fichero todo el contenido del **workspace** (espacio de trabajo), es decir, el conjunto de las variables almacenadas en memoria:

```
save nombredefichero  
save('nombredefichero') % también se puede escribir así
```

Esta orden guarda todo el contenido del workspace (nombres de variables y sus valores) en un fichero de nombre **nombredefichero.mat**. Esto puede ser útil si se necesita abandonar una sesión de trabajo con MATLAB sin haber terminado la tarea: se puede salvar fácilmente a un fichero el workspace en su estado actual, cerrar MATLAB y luego recuperar todas las variables en una nueva sesión. Esto último se hace con la orden

```
load nombredefichero  
load('nombredefichero') % equivalente a la anterior
```

Ejemplo 6.1 (Uso de save para salvar el workspace)

1. Crea unas cuantas variables en tu espacio de trabajo. Por ejemplo:

```
A = rand(5,5);  
B = sin(4*A);  
x = 1:15;  
frase = 'Solo se que no se nada';
```

2. Comprueba con la orden **who** qué variables tienes en el workspace:

```
who
```

3. Salva el contenido del workspace a un fichero de nombre **memoria.mat**:

```
save memoria
```

Comprueba que en tu carpeta de trabajo aparece un fichero con este nombre.

4. Borra todas las variables de la memoria:

```
clear
```

y comprueba con `who` que el workspace está vacío.

5. Recupera tus variables desde el fichero `memoria.mat`:

```
load memoria
```

6. Comprueba de nuevo, con `who`, que vuelves a tener todas tus variables.

También es posible salvar sólo parte de las variables:

```
save nombredefichero var1 var2 var3           (sin comas de separación)
```

Ejemplo 6.2 (Uso de save para salvar algunas variables)

1. Comprueba con `who` que tienes tus variables en el workspace. Si no es así, recuérdalas usando la orden

```
load memoria
```

2. Salva, a otro fichero de nombre `memo.mat` el contenido de dos de las variables que tengas, por ejemplo:

```
save memo A frase
```

Comprueba que en tu carpeta de trabajo aparece un fichero de nombre `memo.mat`.

3. Borra las variables salvadas:

```
clear A frase
```

y comprueba con `who` que ya no existen en el workspace.

4. Recupera tus variables:

```
load memo
```

y comprueba con `who` que ahora están de nuevo en el workspace.

Los ficheros `.mat` creados de esta forma son ficheros binarios. Si se intentaran abrir con un editor de texto plano cualquiera, sólo se verían un montón de signos extraños. Solo MATLAB “sabe” cómo interpretar los datos binarios almacenados dentro.

Sin embargo, la orden `save` también puede ser usada para guardar datos numéricos en un fichero de texto, usando la opción `-ascii`:

```
save nombrefichero var1 var2 -ascii
```

En este caso, el fichero no tiene la extensión `.mat`, ya que la misma está reservada para los ficheros binarios de MATLAB. Se debe especificar la extensión junto con el nombre del fichero. Además, no se guardan en el fichero los nombres de las variables, solamente sus valores.

Ejemplo 6.3 (Uso de save para salvar datos en un fichero de texto)

1. Comprueba con `who` que tienes tus variables en el workspace. Si no es así, recuérdalas usando la orden

```
load memoria
```

2. Salva, a un fichero de texto de nombre `memo.dat` el contenido de algunas de las variables que tengas, por ejemplo:

```
save memo.dat A B -ascii
```

El cualificador `-ascii` indica que se salvaguarden los datos en un fichero de texto. Comprueba que en tu carpeta de trabajo aparece un fichero de nombre `memo.dat`.

3. Puedes usar el editor de MATLAB para «ver» y modificar el contenido de este fichero: con el cursor sobre el ícono del fichero, pulsa el botón derecho del ratón y elige **Open as Text**.

Es importante hacer notar que los datos guardados de esta manera en un fichero de texto no pueden ser recuperados en la memoria con la orden `load`, a menos que se trate de una sola matriz de datos numéricos, es decir que cada línea del fichero contenga la misma cantidad de datos. En ese caso hay que usar la versión `A=load('nombrefichero.extension')` de la orden para guardar los valores de la matriz en la variable `A`.

Ejemplo 6.4 (Cargar datos binarios desde un fichero)

Se dispone de ciertos ficheros de datos de tipo `.mat` (es decir, binarios), cada uno de los cuales contiene tres variables, `x`, `y` y `titulo`, que han sido almacenadas previamente con la orden `save`. Los ficheros son `flor3.mat`, `flor4.mat`, `flor5.mat` y `flor7.mat`.

Se desea escribir un *script* que lea del teclado el nombre del fichero que se desea procesar, cargue las variables en la memoria y dibuje la gráfica definida por los datos `x` e `y`, y le añada el título correspondiente.

Vamos a escribir un *script* que llamaremos, por ejemplo, `Plot_datos.m`, que «cargue» las variables del fichero y genere la gráfica correspondiente. .

```
%  
% Plot_Datos: lee datos de un fichero binario y hace una grafica  
%  
fichero = input('Nombre del fichero a cargar : ','s');  
load(fichero)  
clf  
plot(x,y, 'LineWidth',2)  
title(titulo)  
shg
```

Salva el script y ejecútalo con los distintos ficheros antes mencionados.

Ejemplo 6.5 (Cargar una matriz desde un fichero de texto)

Se dispone de un fichero **temperaturas.dat** en el que se almacenan datos meteorológicos de cierta ciudad. Las temperaturas están expresadas en grados Celsius, con un decimal. Concretamente, el fichero contiene la temperatura a lo largo del día, medida cada hora, desde las 0 hasta las 23 horas (en total 24 datos por día). Cada linea contiene las temperaturas de un día. El fichero contiene cuatro líneas.

Se desea realizar una gráfica que represente, mediante 4 curvas, la evolución de las temperaturas a lo largo de cada día.

1. Puesto que el fichero **temperaturas.dat** es de texto, se puede editar. Ábrelo para ver su estructura. Cierra el fichero.
2. Comienza por «cargar» en memoria los datos del fichero **temperaturas.dat**, que es un fichero de texto. Puesto que todas las filas del fichero contienen el mismo número de datos, esto se puede hacer con la orden **load**:

```
Temp = load('temperaturas.dat');
```

que guarda en la variable **Temp** una matriz con 4 filas y 24 columnas conteniendo los datos del fichero. Compruébalo. Cada fila de **Temp** contiene las temperaturas de un día.

3. Observa que disponemos de los valores de 4 funciones, pero no disponemos de un vector de abscisas. Sabemos que las temperaturas corresponden a las horas de un día, desde 0 hasta 23. En consecuencia vamos a crear un vector con estos valores, para usarlo como abscisas:

```
hh = 0:23;
```

4. Se quiere construir una curva con cada fila de la matriz **Temp**. Esto se puede hacer con la orden

```
plot(hh,Temp(1,:),hh,Temp(2,:),hh,Temp(3,:),hh,Temp(4,:),)
```

Pero también se puede hacer mediante la orden

```
plot(hh,Temp)
```

Consulta en el Help de MATLAB este uso de la función `plot` en el que el segundo argumento es una matriz.

5. Añade un título, una etiqueta en el eje OX y una leyenda, para la correcta comprensión de esta gráfica.

6.3 Lectura y escritura en ficheros avanzada

Las órdenes `save` y `load` son fáciles de usar, aunque apenas permiten diseñar su resultado y sólo sirven en algunos casos.

Por ejemplo, con la orden `save` no se pueden escribir datos en un fichero con un formato personalizado, como una tabla, mezclando texto y datos numéricos, etc.

Con la orden `load` no podemos cargar en memoria datos de un fichero de texto, a menos que se trate de una sola matriz de datos numéricos.

Con frecuencia es necesario utilizar ficheros procedentes de otros usuarios o creados por otros programas y cuya estructura y formato no se puede elegir o, por el contrario, es preciso generar un fichero con una estructura determinada.

Para todo ello es necesario recurrir a órdenes de lectura/escritura de «bajo nivel». Estas órdenes son más complicadas de usar, ya que requieren más conocimiento del usuario pero, a cambio, permiten elegir todas las características de la operación que se realiza.

Los pasos en el trabajo «de bajo nivel» con ficheros son:

Abrir el fichero e indicar qué tipo de operaciones se van a efectuar sobre él. Esto significa que el fichero queda listo para poder ser utilizado por nuestro programa.

Leer datos que ya están en el fichero, o bien **Escribir** datos en el fichero o **Añadir** datos a los que ya hay.

Cerrar el fichero.

Sólo hablamos aquí de operaciones con ficheros formateados. No obstante, en algunas ocasiones, como por ejemplo cuando hay que almacenar grandes cantidades de datos, puede ser conveniente utilizar ficheros binarios, ya que ocupan menos espacio y las operaciones de lectura y escritura sobre ellos son más rápidas.

6.3.1 Apertura y cierre de ficheros

Para **abrir** un fichero sólo para lectura con MATLAB se utiliza la orden

```
fid=fopen('NombreFichero');
```

NombreFichero es el nombre del fichero que se desea utilizar.

fid es un número de identificación que el sistema asigna al fichero, si la operación de apertura ha sido exitosa. Si no, **fopen** devolverá el valor **fid=-1**. En adelante se usará ese número para cualquier operación sobre el fichero que se quiera realizar.

De forma más general, la orden para abrir un fichero es:

```
fid=fopen('NombreFichero', 'permiso');
```

permiso es un código que se utiliza para indicar si se va a utilizar el fichero para leer o para escribir. Posibles valores para el argumento **permiso** son:

r indica que el fichero se va a utilizar sólo para lectura. Es el valor por defecto.
(En Windows conviene poner **rt**).

w indica que el fichero se va a utilizar para escribir en él. En este caso, si existe un fichero con el nombre indicado, se abre y se sobre-escribe (el contenido previo, si lo había, se pierde). Si no existe un fichero con el nombre indicado, se crea.
(En Windows conviene poner **wt**)

a indica que el fichero se va a utilizar para añadir datos a los que ya haya en él.
(En Windows conviene poner **at**)

Para otros posibles valores del argumento **permiso**, ver en el Help de MATLAB la descripción de la función **fopen**.

Una vez que se ha terminado de utilizar el fichero, hay que cerrarlo, mediante la orden

```
fclose(fid)
```

donde **fid** es el número de identificación que **fopen** le dió al fichero. Si se tienen varios ficheros abiertos a la vez se pueden cerrar todos con la orden

```
fclose('all')
```

6.3.2 Leer datos de un fichero con `fscanf`

La orden

```
A = fscanf(fid, 'formato', dimension);
```

lee los datos de un fichero de acuerdo con el `formato` especificado, hasta completar la `dimension` dada, o hasta que se encuentre un dato que no se puede leer con dicho formato, o hasta que se termine el fichero.

`fid` es el número de identificación del fichero, devuelto por la orden `fopen`.

`formato` es una descripción del formato que hay que usar para leer los datos. Se utilizan los mismos códigos que con la orden `fprint`.

`A` es un vector columna en el que se almacenan los datos leídos, en el orden en que están escritos.

`dimension` es la dimensión de la variable a la que se asignarán los datos (`A`). Puede ser:
(i) un número entero `n`; (ii) `Inf`; (iii) un vector `[n, m]` o `[n, Inf]`.

Los datos se leen del fichero en el orden natural en que están escritos en el fichero (por filas) y se almacenan en la matriz `A` en orden de columnas.

Ejemplo 6.6

El fichero `datos3.dat` contiene dos columnas de números reales, de longitud desconocida. Se desea leer dichos datos y guardarlos en una matriz con dos columnas y el número necesario de filas, como están en el fichero.

1. Abre el fichero `datos3.dat` para lectura:

```
fid = fopen('datos3.dat','r');
```

En realidad, la opción `'r'` es la opción por defecto, luego no sería necesario indicarlo.

2. Lee los datos del fichero con la orden:

```
mat = fscanf(fid, '%f');
```

El formato `'%f'` indica que se van a leer datos numéricos reales, esto es con una parte entera, un punto decimal y una parte fraccionaria. El número de dígitos de la parte entera y de la parte fraccionaria vendrá determinado por lo que se lee.

3. Tras la orden anterior, la variable `mat` contiene un vector columna. Compruébalo. Para que los datos formen una matriz con 2 columnas como en el fichero hay que reorganizarlos con la función `reshape` (ver su descripción más abajo):

```
mat = reshape(mat,2,[]);
```

Observa el resultado. Ahora `mat` es una matriz con dos filas. Lo que deseamos obtener es la transpuesta de esta matriz:

```
mat = mat';
```

4. Cierra el fichero

```
fclose(fid);
```

La orden `reshape` utilizada en el Ejemplo anterior sirve para organizar del datos de una matriz almacenada en memoria como una matriz con otras dimensiones:

```
B = reshape(A, nf, nc);
```

asigna los datos de la matriz `A` a la matriz `B` de con `nf` filas y `nc` columnas. Es necesario que el producto de `nf × nc` sea igual al número total de elementos de la matriz `A`.

También se puede dejar sin especificar una de las dos dimensiones:

```
B = reshape(A, nf, []); % con las columnas que salgan  
B = reshape(A, [], nc); % con las filas que salgan
```

Naturalmente, en este caso es necesario que el número total de elementos de `A` sea un múltiplo de `nf` en el primer caso o de `nc` en el segundo.

Observación: en el Ejemplo 6.6 se podrían haber leído los datos del fichero usando la orden `load`, ya que todas las filas están formadas por el mismo número de datos. Sin embargo la orden `fscanf` sirve para otros casos que no se podrían leer con `load`, como en el Ejemplo siguiente.

Ejemplo 6.7

Se dispone de un fichero `contaminacion.dat` que contiene datos sobre contaminación de las provincias andaluzas. Cada linea del fichero contiene el nombre de la provincia y tres datos numéricos.

Se quieren recuperar los datos numéricos del fichero en una matriz con tres columnas, es decir, ignorando el texto de cada línea.

El fichero `contaminacion.dat` tiene el siguiente contenido:

Almeria	3.7	1.4	2.0
Cadiz	9.5	8.5	3.3
Almeria	0.9	4.2	9.9
etc.			

1. Intenta «cargar» el fichero con la orden `load`. Recibirás un mensaje de error, ya que, como hemos dicho, `load` no puede leer ficheros de texto con datos de distintos tipos:

```
mat = load('contaminacion.dat'); % producira errores
```

2. Sin embargo, el fichero se puede leer con la orden **fscanf**. Abre el fichero para lectura:

```
fid = fopen('contaminacion.dat','r');
```

3. Escribe la orden siguiente:

```
mat = fscanf(fid, '%f');
```

que indica que se va a leer el contenido del fichero hasta que se termine o bien hasta que los datos a leer no coincidan con el formato especificado. El formato especifica leer 1 dato real. Si después de éste siguen quedando datos en el fichero, se «re-visita» el formato, es decir, es como si pusiéramos **%f %f %f %f %f %f ...** hasta que se acabe el fichero.

Verás que el resultado es una matriz vacía. ¿Cuál es el error? Que se han encontrado datos que no se pueden leer con el formato **'%f'**.

4. Escribe ahora la orden:

```
frewind(fid);
```

Esta orden «rebobina» el fichero, es decir, coloca el apuntador de lectura al principio del mismo, ya que, tras la orden anterior puede no estar ahí.

5. Escribe ahora la orden

```
mat = fscanf(fid, '%s %f %f %f');
```

Esta orden indica que se han de leer del fichero 1 dato tipo *string* (cadena de caracteres), luego 3 reales y repetir. Ahora obtendrás una matriz con una sola columna. Observa sus valores. ¿Es lo que esperabas? Intenta comprender lo que has obtenido. Quizá te ayude consultar la Tabla de los caracteres imprimibles del Código ASCII, al final de este Capítulo.

6. Rebobina de nuevo el fichero y escribe la orden:

```
mat = fscanf(fid, '*s %f %f %f')
```

Esta orden indica que se **ignore** (***s**) un dato de tipo *string*, que luego se lean 3 reales y luego repetir. Observa el resultado.

7. Para dar a la matriz **mat** la estructura deseada hay que usar la orden

```
mat = reshape(mat,3,[]);
```

que indica que se re-interprete **mat** como una matriz con 3 filas y el número de columnas que salgan y luego se transpone.

6.3.3 Escribir datos en un fichero con fprintf

La orden

```
fprintf(fid, 'formato', lista_de_variables);
```

escribe los valores de la variables de la `lista_de_variables` en el fichero con numero de identificación `fid` de acuerdo con el `formato` especificado.

`fid` es el número de identificación del fichero, devuelto por la orden `fopen`.

`formato` es una descripción del formato que hay que usar para escribir los datos.

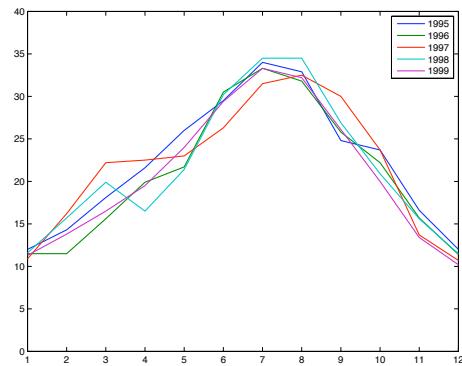
`lista_de_variables` son las variables o expresiones cuyos valores hay que escribir en el fichero.

Ejercicio 6.8 El siglo pasado era costumbre escribir los números de los años con sólo dos dígitos: 76, 77, 78, etc.

Se dispone de un fichero `temp_anuales.dat` en donde se almacena la temperatura máxima media de cada mes del año, para varios años del siglo XX. Los años vienen expresados con dos dígitos y las temperaturas vienen expresadas con una cifra decimal.

Se necesita generar a partir de este fichero otro que contenga los mismos datos, pero con los años expresados con 4 dígitos, para poder añadir los de este siglo. Las temperaturas deben ser escritas, también, con un decimal.

Genera también una gráfica que muestre la evolución de las temperaturas a lo largo del año (una curva para cada año). Dibuja cada curva de un color y añade a cada curva una leyenda con el número del año al que corresponde, como en la gráfica siguiente:



1. Puesto que el contenido del fichero `temp_anuales.dat` es una matriz de números, se puede leer con la orden `load`:

```
datos = load('temp_anuales.dat');
```

2. Los números de los años forman la primera columna de la matriz datos, así que un vector conteniendo los años con la numeración de 4 cifras se obtiene con:

```
anyos = datos(:,1)+1900;
```

Observación: para escribir los datos en un nuevo fichero no podemos usar la orden **save**, ya que los escribiría con formato exponencial (p.e. 9.500000e+01) y queremos escribir los años sin decimales y los datos con solo un decimal.

Hay que usar órdenes de «bajo nivel», concretamente la orden **fprintf**.

3. Abrimos un nuevo fichero para escritura, en el que grabaremos los nuevos datos.

```
fid = fopen('temp_anuales_new.dat','w');
```

4. Recuperamos las dimensiones de la matriz **datos**, para saber cuántas filas tiene:

```
[nfils,ncols] = size(datos); % nfils = numero de filas
```

5. Vamos a realizar un bucle con un índice **k** variando desde **1** hasta **nfils**. En cada iteración del bucle escribiremos una fila del nuevo fichero, mediante las órdenes:

```
fprintf(fid,'%4i',anyos(k)); % el año con formato %4i  
fprintf(fid,'%6.1f',datos(k,2:13)); % los datos con formato %6.1f  
fprintf(fid,'\n'); % al final un salto de linea
```

Prueba a escribir el nuevo fichero mediante una estrategia distinta de esta.

6. Cerramos el fichero

```
fclose(fid);
```

7. La gráfica pedida se puede hacer con las órdenes

```
plot(1:12,datos(:,2:13),'LineWidth',1.1);  
legend(num2str(anyos));  
axis([1,12,0,40])
```

Ejercicio 6.9 Modifica alguno de los scripts de cálculo de series (por ejemplo **SumaExp.m**) para que la tabla de las iteraciones se escriba en un fichero en lugar de en la pantalla.

Caracteres ASCII imprimibles			
32 espacio	64 @	96 `	
33 !	65 A	97 a	
34 "	66 B	98 b	
35 #	67 C	99 c	
36 \$	68 D	100 d	
37 %	69 E	101 e	
38 &	70 F	102 f	
39 '	71 G	103 g	
40 (72 H	104 h	
41)	73 I	105 i	
42 *	74 J	106 j	
43 +	75 K	107 k	
44 ,	76 L	108 l	
45 -	77 M	109 m	
46 .	78 N	110 n	
47 /	79 O	111 o	
48 0	80 P	112 p	
49 1	81 Q	113 q	
50 2	82 R	114 r	
51 3	83 S	115 s	
52 4	84 T	116 t	
53 5	85 U	117 u	
54 6	86 V	118 v	
55 7	87 W	119 w	
56 8	88 X	120 x	
57 9	89 Y	121 y	
58 :	90 Z	122 z	
59 ;	91 [123 {	
60 <	92 \	124 }	
61 =	93]	125 }	
62 >	94 ^	126 ~	
63 ?	95 _		

Figura 6.1: Tabla de los caracteres imprimibles del código ASCII

7

Interpolación y ajuste de datos en MATLAB



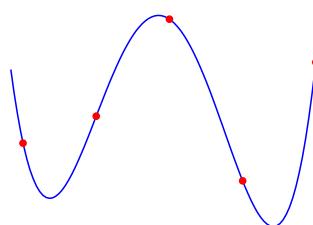
7.1 Introducción

En Física y otras ciencias con frecuencia es necesario trabajar con conjuntos discretos de valores de alguna magnitud que depende de otra variable. Pueden proceder de muestreos, de experimentos o incluso de cálculos numéricos previos.

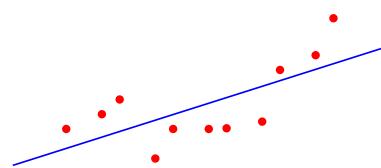
En ocasiones, para utilizar estos valores en cálculos posteriores es preciso «darles forma» de función, es decir: es preciso disponer de una función dada por una expresión matemática que «coincida» con dichos valores.

Existen básicamente dos enfoques para conseguir esto:

Interpolación es el proceso de determinar una función que tome exactamente los valores dados para los valores adecuados de la variable independiente, es decir que pase exactamente por unos puntos dados. Por ejemplo, determinar un polinomio de grado 4 que pase por 5 puntos dados, como en la figura de la derecha.



Ajuste de datos es el proceso de determinar la función, de un tipo determinado, que mejor se aproxime a los datos («mejor se ajuste»), es decir tal que la distancia a los puntos (medida de alguna manera) sea lo menor posible. Esta función no pasará necesariamente por los puntos dados. Por ejemplo, determinar un polinomio de grado 1 que aproxime lo mejor posible unos datos, como se muestra en la figura adjunta.



7.2 Interpolación polinómica global

Si la función a construir es un polinomio de un determinado grado, se habla de interpolación polinómica.

Interpolación lineal. Por dos puntos dados del plano pasa una sola línea recta.

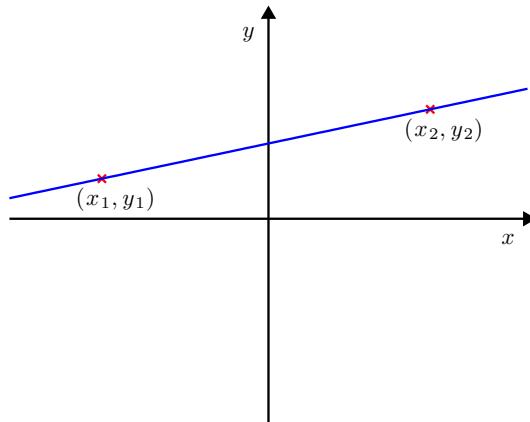
Más concretamente, dados dos puntos en el plano (x_1, y_1) y (x_2, y_2) , con $x_1 \neq x_2$ se trata de determinar una función polinómica de grado 1

$$y = ax + b$$

que tome el valor y_1 para $x = x_1$ y el valor y_2 para $x = x_2$, es decir

$$\begin{cases} y_1 = ax_1 + b \\ y_2 = ax_2 + b \end{cases}$$

La solución de este sistema lineal de dos ecuaciones con dos incógnitas proporciona los valores adecuados de los coeficientes a y b .



Interpolación cuadrática. En general, tres puntos del plano determinan una única parábola (polinomio de grado 2).

Dados (x_1, y_1) , (x_2, y_2) y (x_3, y_3) con x_1 , x_2 y x_3 distintos dos a dos, se trata de determinar una función de la forma

$$y = ax^2 + bx + c$$

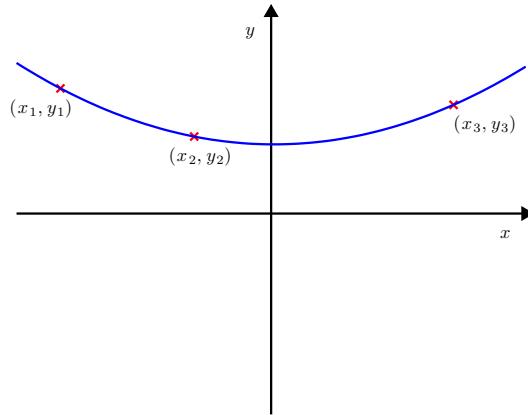
que pase por dichos puntos, es decir tal que

$$\begin{cases} y_1 = ax_1^2 + bx_1 + c \\ y_2 = ax_2^2 + bx_2 + c \\ y_3 = ax_3^2 + bx_3 + c \end{cases}$$

Este sistema lineal se escribe en forma matricial

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

y su solución (única) proporciona los coeficientes que determinan la función interpolante.



Interpolación global. En general, dados N puntos (x_k, y_k) , $k = 1, \dots, N$, con x_k todos distintos, existe un único polinomio de grado $N - 1$ que pasa exactamente por estos puntos. Este polinomio se puede expresar de la forma

$$p(x) = c_1 x^{N-1} + c_2 x^{N-2} + \cdots + c_{N-1} x + c_N$$

y verifica que $p(x_k) = y_k$ para $k = 1, \dots, N$, es decir:

$$\left\{ \begin{array}{l} y_1 = c_1 x_1^{N-1} + c_2 x_1^{N-2} + \cdots + c_{N-1} x_1 + c_N \\ y_2 = c_1 x_2^{N-1} + c_2 x_2^{N-2} + \cdots + c_{N-1} x_2 + c_N \\ \vdots \\ y_N = c_1 x_N^{N-1} + c_2 x_N^{N-2} + \cdots + c_{N-1} x_N + c_N \end{array} \right.$$

En forma matricial, el sistema anterior se escribe:

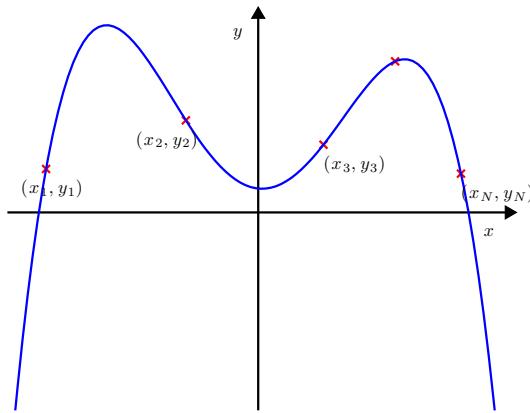
$$\begin{pmatrix} x_1^{N-1} & x_1^{N-2} & \cdots & x_1 & 1 \\ x_2^{N-1} & x_2^{N-2} & \cdots & x_2 & 1 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ x_N^{N-1} & x_N^{N-2} & \cdots & x_N & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}$$

La matriz de este sistema es conocida como *matriz de Vandermonde*.¹ La solución (única) de este sistema proporciona los coeficientes del polinomio (único) que pasa por los N puntos dados.

Este procedimiento se conoce como interpolación global de Lagrange.²

¹Alexandre-Théophile Vandermonde (1735-1796) fue un músico, químico y matemático francés. Aunque en realidad fue Henri-Léon Lebesgue quien en 1940 dió, en su honor, este nombre a las matrices.

²Joseph Louis Lagrange (1736–1813), fue un matemático, físico y astrónomo italiano nacido en Turín, aunque la mayor parte de su vida transcurrió entre Prusia y Francia.



Interpolación con MATLAB. Dados N puntos (x_k, y_k) , $k = 1, 2, \dots, N$, con todos los x_k distintos, la instrucción

```
c=polyfit(x,y,N-1)
```

calcula los coeficientes del polinomio de grado N-1 que pasa por los N puntos.

x, y son dos vectores de longitud N conteniendo respectivamente las abscisas y las ordenadas de los puntos.

El vector **c** devuelto por **polyfit** contiene los coeficientes del polinomio de interpolación:

$$p(x) = c_1 x^{N-1} + c_2 x^{N-2} + \cdots + c_{N-1} x + c_N.$$

Ejercicio 7.1 Calcular el polinomio de interpolación de grado 2 que pasa por los puntos

$$(1, -3), (2, 1), (3, 3)$$

Representar su gráfica en el intervalo $[0, 7]$, señalando con marcadores los puntos interpolados y dibujando también los ejes coordenados.

1. Crea dos vectores **x** e **y** conteniendo respectivamente las abscisas y las ordenadas de los puntos:

```
x = [1,2,3];
y = [-3,1,3];
```

2. Calcula con la orden **polyfit** los coeficientes de la parábola que pasa por estos puntos:

```
c = polyfit(x,y,2);
```

Obtendrás el resultado **c=[-1, 7, -9]**, que significa que el polinomio de interpolación de grado 2 que buscamos es $p(x) = -x^2 + 7x - 9$.

3. Para dibujar la gráfica de esta función comenzamos por crear un vector `z` de puntos en el intervalo $[0, 9]$:

```
z = linspace(0,7);
```

4. Necesitamos ahora calcular los valores del polinomio $p(x)$ en todos estos puntos. Para ello usamos la función `polyval`:

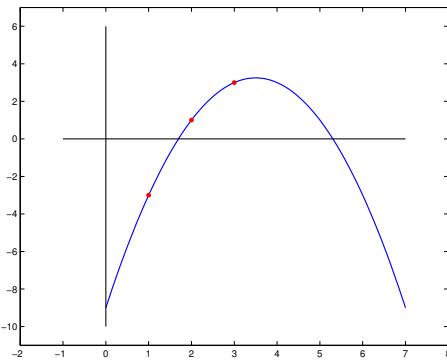
```
p = polyval(c,z);
```

5. Dibuja la parábola:

```
plot(z,p)
```

6. Añade ahora los ejes de coordenadas y los marcadores de los puntos del soporte de interpolación:

```
hold on
plot([-1,7],[0,0], 'k','LineWidth',1.1)
plot([0,0],[-10,6], 'k','LineWidth',1.1)
plot(x,y,'r.','MarkerSize',15)
axis([-2,8,-11,7])
hold off
```



Ejercicio 7.2 La temperatura del aire cerca de la tierra depende de la concentración K del ácido carbónico (H_2CO_3) en él. En la tabla siguiente se recoge, para diferentes latitudes L sobre la tierra y para el valor de $K = 0.67$, la variación δ_K de la temperatura con respecto a una cierta temperatura de referencia:

L	65	35	5	-25	-55
δ_K	-3.1	-3.32	-3.02	-3.2	-3.25

Calcular y dibujar el polinomio de interpolación global de estos datos. Usando el polinomio de interpolación construido, calcular la variación de la temperatura para $L = 10$ (valor de la latitud que no está entre las mediciones de la tabla de datos).

1. Crea dos vectores con los datos de la tabla:

```
x = [-55,-25,5,35,65];
y = [-3.25,-3.2,-3.02,-3.32,-3.1];
```

2. Calcula y dibuja el polinomio de interpolación:

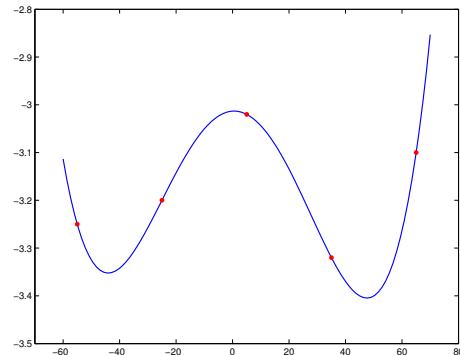
```
c = polyfit(x,y,4);
z = linspace(-60,70);
p = polyval(c,z);
plot(z,p,'Color',[0,0,1],'LineWidth',1.2)
```

3. Dibuja también marcadores de los puntos:

```
hold on
plot(x,y,'r.', 'MarkerSize',15)
```

4. Usa el polinomio que acabas de obtener para calcular δ_K para $L = 10$ e imprime su valor:

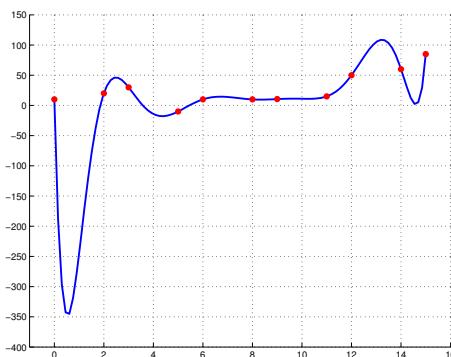
```
L = 10;
delta=polyval(c,10);
fprintf('Para L =%3i delta = %6.3f \n ',L, delta)
```



Ejercicio 7.3 (Propuesta) Calcula el polinomio de grado 10 que interpola los valores:

x	0	2	3	5	6	8	9	11	12	14	15
y	10	20	30	-10	10	10	10.5	15	50	60	85

Dibuja su gráfica y observa las inestabilidades cerca de los extremos:



Esta práctica pretende mostrar que el procedimiento de **interpolación global** es, en general inestable, ya que los polinomios tienden a hacerse oscilantes al aumentar su grado y eso puede producir grandes desviaciones sobre los datos.

Ejercicio 7.4 (Propuesta) Calcula el polinomio de grado 5 que interpola los valores:

x	0.01	2.35	1.67	3.04	2.35	1.53
y	6	5	4	3	2	3

Recibirás un **Warning** y unos resultados.

Intenta comprender lo que dice el mensaje del **Warning** y analizar los resultados. ¿Ves algo raro? ¿Llegas a alguna conclusión?

7.3 Interpolación lineal a trozos

Como se ha visto en la Práctica 7.3, la interpolación polinómica global es inestable cuando el número de puntos es elevado. En la práctica se usan procedimientos de **interpolación a trozos**, que se explica en lo que sigue.

Consideramos N puntos (x_k, y_k) , $k = 1, \dots, N$, con los valores de x_k todos diferentes y ordenados en orden creciente o decreciente. Se llama **interpolante lineal a trozos** a la poligonal que sobre cada intervalo formado por dos valores de x consecutivos: $[x_k, x_{k+1}]$, $k = 1, \dots, N - 1$ está definida por el segmento que une los puntos (x_k, y_k) y (x_{k+1}, y_{k+1}) , como en la Figura 7.1.

La instrucción MATLAB:

```
s1=interp1(x,y,z)
```

calcula el valor en el/los punto(s) z de la interpolante lineal a trozos que pasa por los puntos (x, y) .

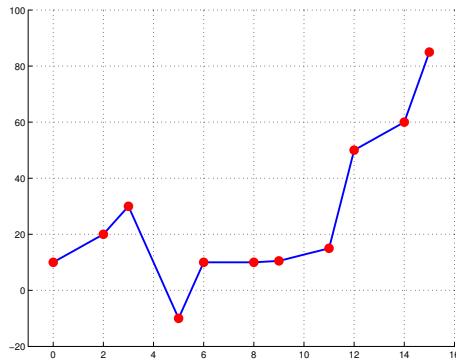


Figura 7.1: Interpolante lineal a trozos.

x **y** son dos vectores de la misma dimensión que contienen las abscisas y las ordenadas de los puntos dados.

z son las abscisas de los puntos a interpolar, es decir, los puntos en los cuales queremos evaluar la interpolante. Puede ser una matriz.

s1 son los valores calculados, es decir, los valores de la función interpolante en **z**. **s1** tendrá las mismas dimensiones que **z**.

Ejercicio 7.5 Se consideran los mismos valores de la Práctica 7.3:

$$x = (0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15),$$

$$y = (10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85)$$

Calcula, a partir del polinomio de interpolación lineal a trozos, el valor interpolado para $x = 1$ y compáralo con el obtenido mediante un polinomio de interpolación global de grado 10 (el de la Práctica 7.3)

Dibuja juntas las gráficas del polinomio de interpolación lineal a trozos y del polinomio de interpolación de grado 10.

- Crea dos variables **x** e **y** con las abscisas y las ordenadas que vas a interpolar:

```
x = [0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y = [10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

- Calcula a partir del polinomio de interpolación lineal a trozos el valor interpolado para $x = 1$:

```
s1 = interp1(x,y,1)
```

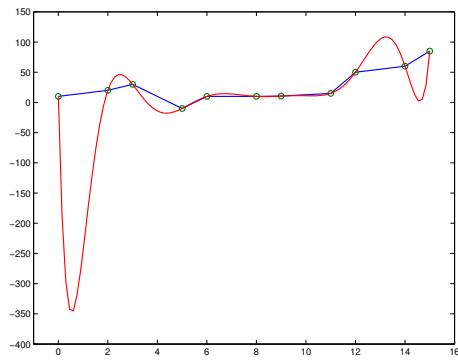
Obtendrás el valor **s1=15**, mientras que mediante el polinomio de interpolación de grado 10 de la Práctica 3 se obtendría:

```
c = polyfit(x,y,10);
s = polyval(c,1)      % = -245.7853
```

3. Dibuja el polinomio de interpolación lineal a trozos y el polinomio de grado 10 junto con los puntos a interpolar:

```
t = linspace(0,15);
p = polyval(c,t);
plot(x,y,x,y,'o',t,p)
axis([-1,16,-400,150])
```

Observa la gráfica y compara los valores los valores **s1** y **c1** ¿Qué puedes decir? ¿Cuál de los dos valores da una aproximación *a priori* más acertada?



Ejercicio 7.6 (Propuesta para valientes) Escribe tu propia M-función

```
function [yi] = interpol(xs,ys,xi)
```

que haga lo mismo que `interp1(xs,ys,xi)`.

7.4 Interpolación por funciones *spline*

Con frecuencia se necesita interpolar un conjunto de datos con funciones «suaves» (sin picos), como por ejemplo en la creación de gráficos por ordenador.

Obsérvese que la función de interpolación lineal a trozos es sólo continua y para obtener funciones «suaves» necesitamos que tengan al menos una derivada continua. Esto se consigue usando, por ejemplo, interpolación a trozos como la que hemos explicado antes, pero con polinomios cúbicos en vez de líneas rectas. Las funciones así construidas se conocen como **splines cúbicos**

En MATLAB podemos usar la instrucción

```
s = spline(x,y,z)
```

Los argumentos de entrada `x`, `y` y `z` tienen el mismo significado que en el comando `interp1` (ver la Sección 7.3), el argumento de salida almacena en el vector `s` los valores de la función spline en los puntos arbitrarios guardados en el vector `z`.

Ejercicio 7.7 Calcula el spline cúbico que interpola los datos de la Práctica 7.3. Dibuja en la misma ventana el spline calculado junto con el polinomo de interpolación lineal a trozos.

1. Crea dos variables `x` e `y` con las abscisas y las ordenadas que vas a interpolar:

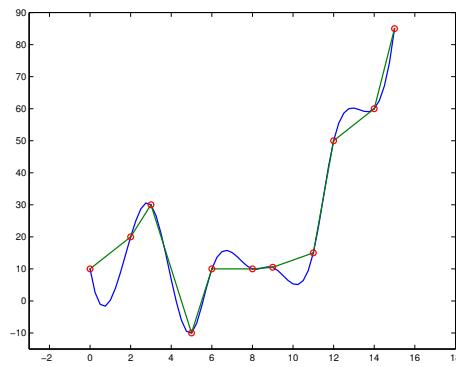
```
x=[0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];  
y=[10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Calcula el spline cúbico

```
z=linspace(0,15);  
s=spline(x,y,z);
```

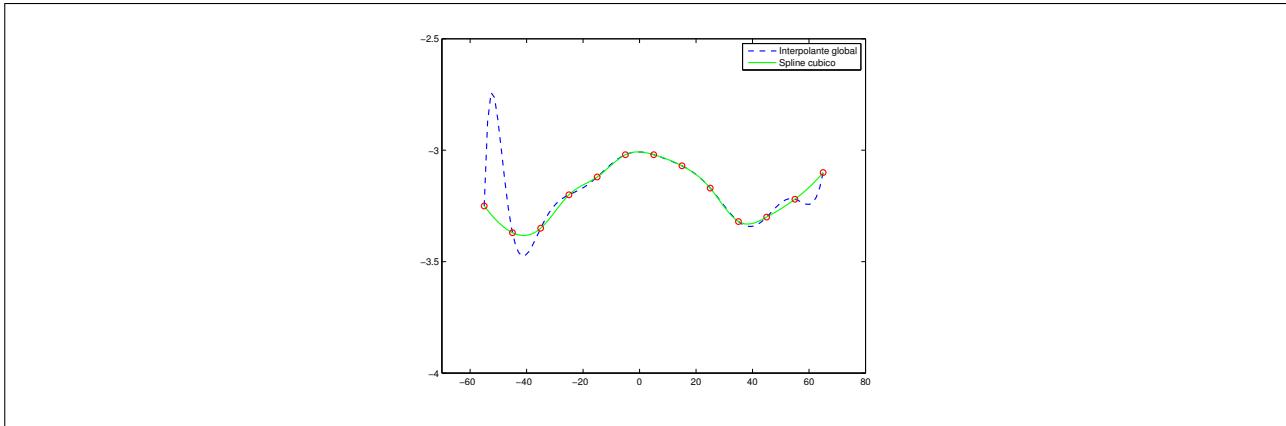
3. Dibuja el spline cúbico y la interpolante lineal a trozos junto con los datos:

```
plot(x,y,'or',z,s,x,y,'LineWidth',2)
```



Ejercicio 7.8 (Propuesta) El fichero `Datos7.dat` contiene una matriz con dos columnas, que corresponden a las abscisas y las ordenadas de una serie de datos.

Hay que leer los datos del fichero, y calcular y dibujar juntos el polinomio de interpolación global y el spline cúbico que interpolan dichos valores, en un intervalo que contenga todos los puntos del soporte.



Existen distintos tipos de splines que se diferencian en la forma que toman en los extremos. Para más información consulta en el **Help** de MATLAB.

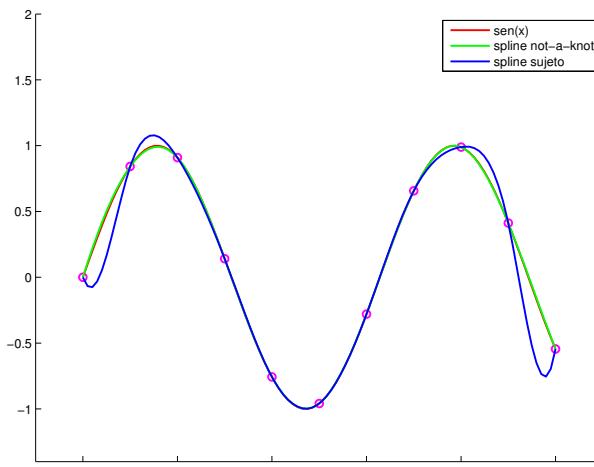
Ejercicio 7.9 (Para ampliar conocimientos) Cuando se calcula un spline cúbico con la función **spline** es posible cambiar la forma en que éste se comporta en los extremos. Para ello hay que añadir al vector **y** dos valores extra, uno al principio y otro al final. Estos valores sirven para imponer el valor de la pendiente del spline en el primer punto y en el último. El spline así construido se denomina *sujeto*.

Naturalmente, todos los procedimientos de interpolación antes explicados permiten aproximar funciones dadas: basta con interpolar un soporte de puntos construido con los valores exactos de una función.

En esta práctica se trata de calcular y dibujar una aproximación de la función $\sin(x)$ en el intervalo $[0, 10]$ mediante la interpolación con dos tipos distintos de spline cúbico y comparar estos resultados con la propia función. Hay por lo tanto que dibujar tres curvas en $[0, 10]$:

1. La curva $y = \sin(x)$.
2. El spline que calcula MATLAB por defecto (denominado *not-a-knot*).
3. El spline *sujeto* con pendiente $= -1$ en $x = 0$ y pendiente $= 2$ en $x = 10$.

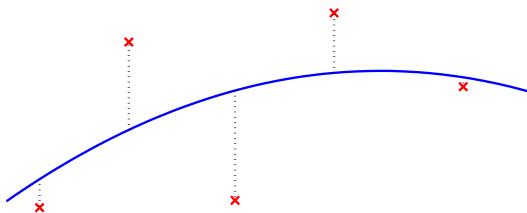
Observación: Este ejercicio no es imprescindible.



7.5 Ajuste de datos

La técnica de interpolación que hemos explicado antes requiere que la función que interpola los datos pase exactamente por los mismos. En ocasiones esto no da resultados muy satisfactorios, por ejemplo si se trata de muchos datos. También sucede con frecuencia que los datos vienen afectados de algún error, por ejemplo porque provienen de mediciones. No tiene mucho sentido, pues, obligar a la función que se quiere construir a «pasar» por unos puntos que ya de por sí no son exactos.

Otro enfoque diferente es construir una función que no toma exactamente los valores dados, sino que «se les parece» lo más posible, por ejemplo minimizando el error, medido éste de alguna manera.



Cuando lo que se minimiza es la suma de las distancias de los puntos a la curva (medidas como se muestra en la figura) hablamos de **ajuste por mínimos cuadrados**. La descripción detallada de este método se escapa de los objetivos de esta práctica. Veremos solamente cómo se puede hacer esto con MATLAB en algunos casos sencillos.

7.5.1 Ajuste por polinomios

La función **polyfit** usada ya para calcular el polinomio de interpolación global sirve también para ajustar unos datos por un polinomio de grado dado:

```
c=polyfit(x,y,m)
```

x **y** son dos vectores de la misma dimensión que contienen respectivamente las abscisas y las ordenadas de los N puntos.

m es el grado del polinomio de ajuste deseado

c es el vector con los coeficientes del polinomio de ajuste

Si $m = 1$, el polinomio resultante es una recta, conocida con el nombre de **recta de regresión**, si $m = 2$ es una parábola, y así sucesivamente. Naturalmente, cuando $m = N - 1$ el polinomio calculado es el polinomio de interpolación global de grado $N - 1$ que pasa por todos los puntos.

Ejercicio 7.10 Calcula y dibuja los polinomios de ajuste de grado 1, 2, 3 y 6 para los siguientes datos:

$$(0.9, 0.9) \quad (1.5, 1.5) \quad (3, 2.5) \quad (4, 5.1) \quad (6, 4.5) \quad (8, 4.9) \quad (9.5, 6.3)$$

Una vez calculados, escribe las expresiones analíticas de los polinomios que has obtenido.

1. Construye los vectores **x** e **y** a partir de los datos:

```
x=[0.9, 1.5, 3, 4, 6, 8, 9.5];  
y=[0.9, 1.5, 2.5, 5.1, 4.5, 4.9, 6.3];
```

2. Calcula la recta de regresión e imprime los coeficientes calculados:

```
p1=polyfit(x,y,1);
```

Obtendrás los valores: **0.57** y **1.00**. La recta de regresión es $y = 0.57x + 1$

3. Dibuja la recta de regresión (recuerda que para dibujar una recta bastan dos puntos):

```
subplot(2,2,1)  
plot(x,y,'o',[0,10],[polyval(p1,0),polyval(p1,10)])  
title('Ajuste por una recta')
```

4. Calcula ahora la parábola que ajusta los datos:

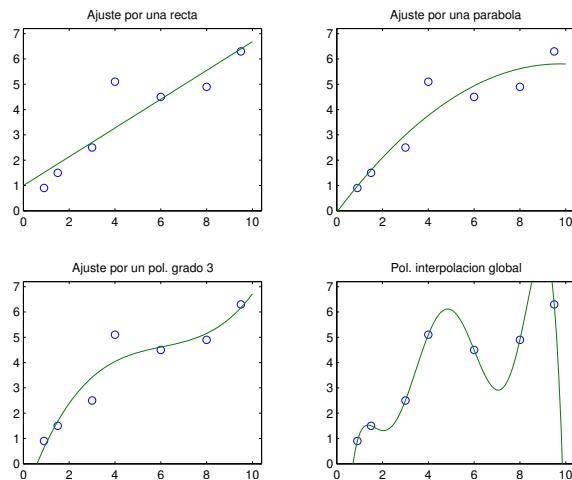
```
p2=polyfit(x,y,2);
```

Obtendrás los valores **-0.0617**, **1.2030** y **-0.0580**. La parábola es por tanto:
 $y = -0.062x^2 + 1.2x - 0.06$ (redondeando a dos decimales).

5. Dibuja la parábola

```
xp=linspace(0,10);  
subplot(2,2,2)  
plot(x,y,'o',xp,polyval(p2,xp));  
title('Ajuste por una parabola')
```

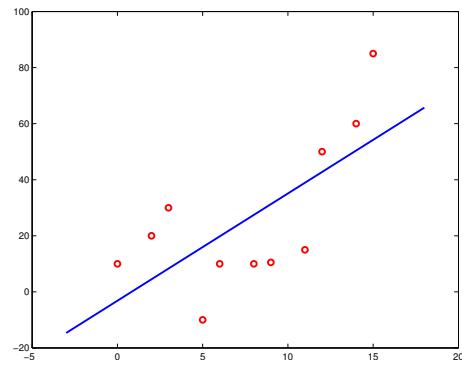
6. Termina la práctica construyendo y dibujando los polinomios de grado 3 y 6. Observa que el polinomio de grado 6 es el polinomio de interpolación que pasa por todos los puntos.



Al realizar esta práctica dibujando cada curva en un cuadro distinto, como se hace aquí, debes tener cuidado de fijar en cada `subplot` los mismos ejes con el comando `axis`, para poder comparar bien. Si no lo haces los resultados te pueden resultar confusos.

Ejercicio 7.11 (Propuesta) Calcula y representa gráficamente la recta de regresión asociada a los siguientes datos:

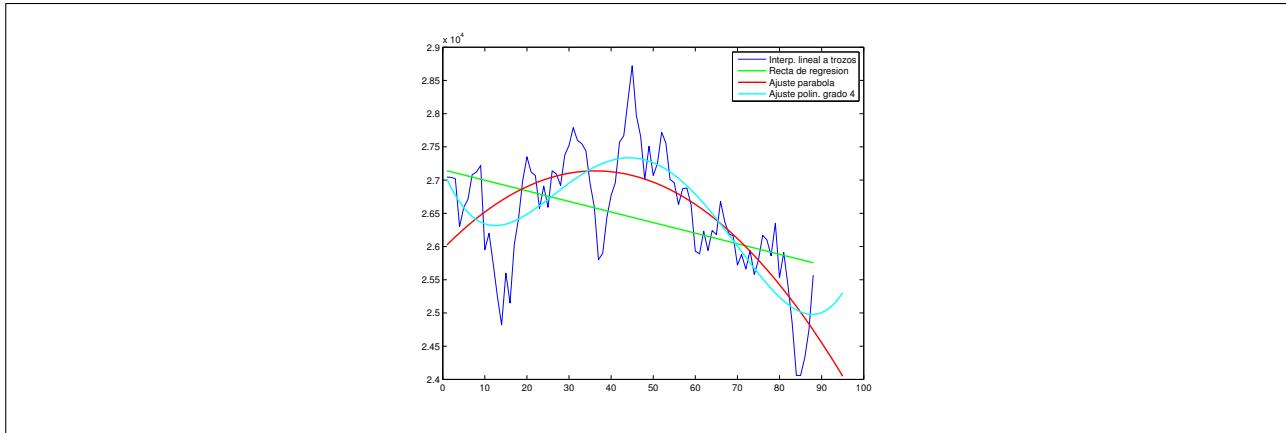
x	0	2	3	5	6	8	9	11	12	14	15
y	10	20	30	-10	10	10	10.5	15	50	60	85



Ejercicio 7.12 (Propuesto) En el fichero `Finanzas.dat` está recogido el precio de una determinada acción de la Bolsa española a lo largo de 88 días, tomado al cierre de cada día.

Queremos ajustar estos datos por una función que nos permita predecir el precio de la acción para un corto intervalo de tiempo más allá de la última cotización.

Lee los datos del fichero y represéntalos. Calcula los polinomios de ajuste de grados 1, 2 y 4 y represéntalos también.



7.5.2 Otras curvas de ajuste

En ocasiones hace falta usar funciones distintas a las polinómicas para ajustar datos. Desde el punto de vista teórico se puede utilizar cualquier función. Las que se utilizan habitualmente son las siguientes:

Función potencia: $y = b x^m$ Se trata de encontrar b y m de forma que la función $y = b x^m$ se ajuste lo mejor posible a unos datos. Observando que

$$y = b x^m \Leftrightarrow \ln(y) = \ln(b) + m \ln(x)$$

se ve que se pueden encontrar $\ln(b)$ y m ajustando los datos $\ln(x)$, $\ln(y)$ mediante un polinomio de grado 1.

Función exponencial: $y = b e^{mx}$ Nuevamente, tomando logaritmos se tiene:

$$y = b e^{mx} \Leftrightarrow \ln(y) = \ln(b) + mx$$

de donde se pueden encontrar $\ln(b)$ y m ajustando los datos x y $\ln(y)$ mediante una recta.

Función logarítmica: $y = m \ln(x) + b$ Está claro que hay ajustar los datos $\ln(x)$ e y mediante una recta.

Función $y = \frac{1}{mx + b}$ La anterior relación se puede escribir también:

$$y = \frac{1}{mx + b} \Leftrightarrow mx + b = \frac{1}{y}$$

lo que muestra que x y $\frac{1}{y}$ se relacionan linealmente. Por lo tanto se pueden calcular m y b ajustando x y $\frac{1}{y}$ mediante una recta de regresión.

Ejercicio 7.13 Ajustar los datos

x	2.0	2.6	3.2	3.8	4.4	5.0
y	0.357	0.400	0.591	0.609	0.633	0.580

mediante una función potencial $y = bx^m$ y dibujar la función obtenida así como los datos con marcadores.

Puesto que, si los x e y son positivos,

$$y = b x^m \Leftrightarrow \ln(y) = \ln(b) + m \ln(x)$$

podemos calcular la recta de regresión $y = \alpha x + \beta$ para los datos $(\ln(x), \ln(y))$ y luego tomar $b = e^\beta$ y $m = \alpha$.

1. Creamos los dos vectores:

```
x = [2.0, 2.6, 3.2, 3.8, 4.4, 5.0];
y = [0.357, 0.400, 0.591, 0.609, 0.633, 0.580];
```

2. Calculamos los coeficientes de la recta de regresión:

```
c = polyfit(log(x), log(y), 1);
```

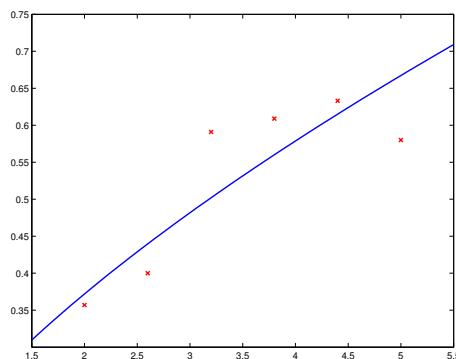
Obtendrás $c = [0.6377, -1.4310]$, lo que significa, por lo dicho antes, que la función que buscamos es:

$$y = e^{-1.4310} \cdot x^{0.6377} = 0.2391 \cdot x^{0.6377}$$

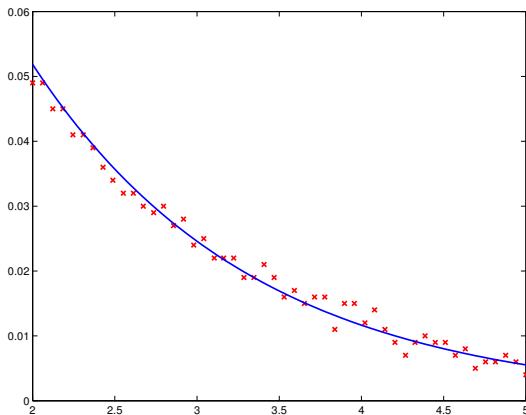
3. Dibujamos ahora la función obtenida y los datos:

```
plot(x,y,'rx','LineWidth',1.5)
hold on

xs = linspace(1.5,5.5);
ys = exp(c(2)) * xs.^c(1));
plot(xs,ys,'LineWidth',1.5)
hold off
```



Ejercicio 7.14 (Propuesta) Ajustar los datos contenidos en el fichero **datos13.dat** mediante una función exponencial $y = b e^{mx}$ y dibujar la función obtenida así como los datos.



Ejercicio 7.15 Determinar una función, de las indicadas antes, que se ajuste lo mejor posible (a simple vista) a los datos de la siguiente tabla:

x	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
y	6	4.83	3.7	3.15	2.41	1.83	1.49	1.21	0.96	0.73	0.64

Escribe la expresión de la función que has calculado.

1. En primer lugar representamos los datos que se quieren ajustar.

```
x=0:0.5:5;
y=[6, 4.83, 3.7, 3.15, 2.41, 1.83, 1.49, 1.21, 0.96, 0.73, 0.64];
plot(x,y,'rx','LineWidth',1.2)
```

Observa, mirando la gráfica, que una función lineal no proporcionaría el mejor ajuste porque los puntos claramente no siguen una línea recta. De las restantes funciones, la logarítmica también se excluye, ya que el primer punto es $x = 0$. Lo mismo sucede con la función potencia ya que ésta se anula en $x = 0$ y nuestros datos no.

Vamos a realizar el ajuste con las funciones exponencial y inversa. A la vista de la gráfica que vamos a obtener veremos cuál de ellas se ajusta mejor a los datos.

2. Comienza por calcular la función exponencial $y = b e^{mx}$. Utiliza la función **polyfit** con **x** y **log(y)** para calcular los coeficientes b y m :

```
c=polyfit(x,log(y),1);
m1=c(1);
b1=exp(c(2));
```

Escribe la expresión de la función que has calculado.

3. Determina ahora la función inversa $y = 1/(mx + b)$, utilizando de nuevo **polyfit** con **x** y **1./y**:

```
p=polyfit(x,1./y,1);
m2=p(1);
b2=p(2);
```

Escribe la expresión de la función que has calculado.

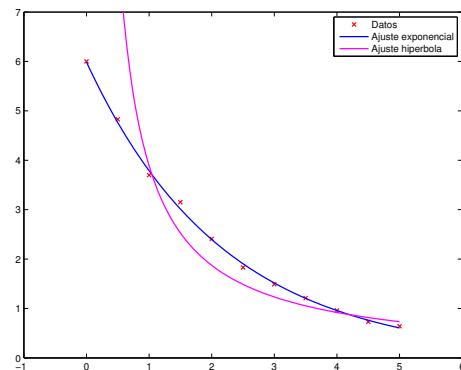
4. Para dibujar las dos funciones escribimos:

```
t1=linspace(0,5);
t2=linspace(0.1,5)
y1=b1*exp(m1*t1);
y2=1./(m2*t2+b2);
```

¿Porqué se ha creado el vector **t2**? ¿Es necesario hacerlo?

5. Realizamos la representación gráfica de ambas funciones

```
axis([-1,6,0,7])
plot(t1,y1,t2,y2)
legend('Datos','Ajuste por exponencial','Ajuste por hipérbola')
```



A la vista de la gráfica obtenida, ¿cuál de las dos funciones se ajusta mejor a los datos?

8

Resolución de ecuaciones no lineales



8.1 Introducción

Dada $f : [a, b] \subset \mathbb{R} \mapsto \mathbb{R}$, continua, se plantea el problema de encontrar soluciones de la ecuación

$$f(x) = 0. \quad (8.1)$$

A las soluciones de esta ecuación, también se les suele llamar **ceros** de la función f , por ser los puntos en los que $f = 0$.

Desde el punto de vista geométrico, las soluciones de la ecuación 8.1 son los puntos en los que la gráfica de la función f «toca» al eje OX , aunque no necesariamente lo atraviesa (véanse las figuras)

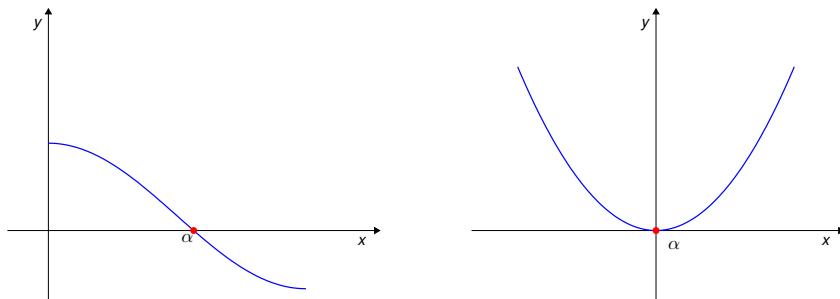


Figura 8.1: La gráfica «toca» al eje de abscisas en el punto α , lo que significa que α es un cero de la función f , aunque no en ambos casos la función cambia de signo en α .

En las aplicaciones surgen frecuentemente problemas que conducen a ecuaciones del tipo (8.1) cuyas soluciones no se pueden calcular explícitamente.

Por ejemplo, la ecuación

$$e^x + e^{-x} = \frac{2}{\cos x}$$

que aparece, por ejemplo, cuando se quieren determinar las frecuencias de las oscilaciones transversales de una viga con extremos empotrados y sometida a un golpe. Las soluciones de esta ecuación no se pueden calcular por métodos analíticos.

Incluso para ecuaciones tan aparentemente sencillas como las polinómicas

$$a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = 0 \quad (8.2)$$

es bien conocido que para $n \geq 5$, no existe una fórmula explícita de sus soluciones.

8.2 Resolución de ecuaciones polinómicas

En el caso de ecuaciones polinómicas, es posible calcular, de una vez, todas sus soluciones, gracias al comando, ya conocido,

```
s = roots(p)
```

que calcula todas las raíces (reales y complejas) de un polinomio y puede ser utilizado para calcular las soluciones reales de la ecuación 8.2.

Ejemplo 8.1

Calcular las soluciones de la ecuación polinómica:

$$x^3 - 9x^2 - x + 5 = 0.$$

-
1. Comenzamos introduciendo el polinomio p que aparece en el primer miembro de la ecuación homogénea anterior. Recuerda que se introduce como un vector fila cuyas componentes son los coeficientes del polinomio, ordenados de mayor a menor grado. En este caso

$$p = [1, -9, -1, 5];$$

Recuerda que hay que incluir los coeficientes nulos, si los hay.

2. Calculamos las raíces

```
roots(p)
```

obtendrás las raíces: **9.0494**, **-0.7685** y **0.7190** que, puesto que son todas reales, son las soluciones de la ecuación.

Recuerda también, siempre, que estamos realizando cálculos numéricos con el ordenador y que, por consiguientes, **todo** lo que calculemos es **aproximado**.

Ejemplo 8.2

Calcular las soluciones de la ecuación polinómica:

$$2x^2(x + 2) = -1.$$

1. Comenzamos por escribir la ecuación en forma homogénea (con segundo miembro cero) y desarrollar el polinomio, para disponer de todos sus coeficientes:

$$2x^2(x + 2) = -1 \Leftrightarrow 2x^3 + 4x^2 + 1 = 0$$

2. Los coeficientes del polinomio son:

$$p = [2, 4, 0, 1];$$

3. Calculamos las raíces

$$\text{roots}(p)$$

obtendrás las raíces: -2.1121 , $0.0560 + 0.4833i$ y $0.0560 - 0.4833i$. Por consiguiente, en el campo real, la única solución de la ecuación es $x = -2.1121$.

Ejercicio 8.3 Se quiere determinar el volumen V ocupado por un gas a temperatura T y presión p a partir de la ecuación de estado

$$\left[p + a \left(\frac{N}{V} \right)^2 \right] (V - Nb) = kNT,$$

siendo N el número de moléculas, k la constante de Boltzmann y a y b constantes que dependen del gas.

Escribir una M-función

```
function [V] = volumen(T, p, N, k, a, b)
```

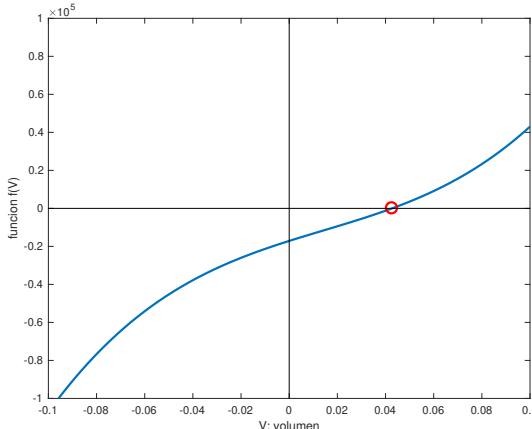
que, a partir de los valores de T , p , N , k , a y b , calcule la solución de la ecuación polinómica

$$f(V) = pV^3 - (pbN + kNT)V^2 + aN^2V - abN^3 = 0$$

(obtenida de la ecuación de estado multiplicando por V^2) y realice una gráfica de la función $f(V)$ en un intervalo adecuado, marcando en ella la solución.

Datos para comprobación: Para el dióxido de carbono (CO_2) se tiene $k = 1.3806503 \cdot 10^{-23}$ Joule/K, $a = 0.401$ Pa m⁶ y $b = 42.7 \cdot 10^{-6}$ m³.

El volumen ocupado por 1000 moléculas de CO₂ a temperatura $T = 300$ K y presión $p = 3.5 \cdot 10^7$ Pa debe ser $V = 0.0427$.



8.3 La función fzero

Los algoritmos numéricos para resolver el problema

$$\text{Hallar } x \in [a, b] \text{ tal que } f(x) = 0 \quad (8.3)$$

cuando la ecuación $f(x) = 0$ es no lineal (en el caso lineal su resolución es inmediata) son en general **algoritmos iterados**.

Esto significa que, a partir de un punto o intervalo iniciales (dependiendo del algoritmo), se construye una sucesión de aproximaciones (calculadas una a partir de la anterior) cuyo límite es la solución buscada. Estos algoritmos, cuando convergen, lo hacen a **una** de las soluciones de la ecuación. Si ésta tuviera más de una solución, habría que utilizar el algoritmo una vez para cada una, cambiando el punto inicial.

En la práctica, lógicamente, sólo se efectúan un número finito de iteraciones y el algoritmo se detiene cuando se verifica algún criterio, previamente establecido.

Para resolver el Problema 8.4, MATLAB dispone de la función

```
solucion = fzero(funcion, xcero)
```

donde

funcion es un manejador de la función que define la ecuación, f . Puede ser el nombre de una **función anónima** dependiente de una sola variable, o también un manejador de una **M-función**, en cuyo caso se escribiría **@funcion**. Ver los ejemplos a continuación.

xcero puede ser:

- un valor «cercano» a la solución, a partir del cual el algoritmo iterado de búsqueda de la solución comenzará a trabajar.
- un intervalo que contenga a una solución (y solo una).

solucion es el valor (aproximado) de la solución encontrado por el algoritmo.

Ejemplo 8.4

La ecuación:

$$x + \ln\left(\frac{x}{3}\right) = 0$$

tiene una solución cerca de $x = 1$. Calcularla.

1. Comienza por definir una función anónima que evalúe la expresión del primer miembro:

```
fun = @(x) x + log(x/3);
```

2. A continuación usa el comando **fzero** tomando **x=1** como valor inicial:

```
fzero(fun,1)
```

obtendrás el resultado **1.0499**.

Tambien podríamos haber usado una M-función para definir la función, en lugar de una función anónima. Mostramos a continuación cómo se haría.

1. Escribimos, en un fichero de nombre **mifuncion.m**, las órdenes siguientes:

```
function [y] = mifuncion(x)
y = x + log(x/3);
```

Salvamos el fichero y lo cerramos. Recuerda que el fichero tiene que llamarse igual que la M-función.

2. En la ventana de comandos, para calcular el cero de la función escribimos:

```
fzero(@mifuncion,1)
```

Como se ha dicho antes, los algoritmos de aproximación de raíces de ecuaciones no lineales necesitan que el punto inicial que se tome esté «cerca» de la solución. ¿Cómo de cerca? Esta pregunta no tiene una respuesta fácil.

Hay condiciones matemáticas que garantizan la convergencia del algoritmo si el punto inicial se toma en un entorno adecuado de la solución. Estas condiciones no son inmediatas de comprobar y requieren un análisis de cada caso.

El ejemplo siguiente pone de manifiesto la necesidad de elegir un punto cercano a la solución.

Ejemplo 8.5

Analizar la influencia del punto inicial en la convergencia de **fzero** al aproximar la solución de

$$x + \ln\left(\frac{x}{3}\right) = 0$$

1. Ya en la práctica anterior se ha definido la función anónima y se ha comprobado la eficacia de **fzero** eligiendo $x = 1$ como punto inicial:

```
fun = @(x) x + log(x/3);
fzero(fun, 1)
```

2. Prueba ahora eligiendo un punto inicial más alejado de la solución, por ejemplo:

```
fzero(fun, 15)
```

Recibirás un mensaje diciendo que se aborta la búsqueda de un cero porque durante la búsqueda se han encontrado valores complejos de la función (la función logaritmo no está definida para argumentos negativos en el campo real, pero sí lo está en el campo complejo). En la búsqueda de un intervalo que contenga un cambio de signo de la función, el algoritmo se ha topado con valores de x negativos, en los que la función **log** toma valores complejos.

El algoritmo utilizado por **fzero** comienza por «localizar» un intervalo en el que la función cambie de signo. No funcionará, pues, con ceros en los que no suceda esto, como pasa en el ??.

8.4 Gráficas para localizar las raíces y elegir el punto inicial

La determinación teórica de un punto inicial cercano a la solución puede ser una tarea difícil. Sin embargo, en muchos casos, un estudio gráfico previo puede resultar de gran ayuda.

Ejemplo 8.6

Calcular, si existe, una solución positiva de la ecuación

$$\sin(x) - 2 \cos(2x) = 2 - x^2$$

determinando un punto inicial a partir de la gráfica de la función.

1. Comienza por escribir la ecuación en forma homogénea:

$$\sin(x) - 2 \cos(2x) + x^2 - 2 = 0$$

2. A continuación, representa gráficamente la función.

```
fun = @(x) sin(x) -2*cos(2*x) + x.^2 - 2;
x = linspace(-5,5);
plot(x,fun(x));
```

La gráfica te mostrará que esta función tiene dos ceros: uno positivo cerca de $x = 1$ y otro negativo cerca de $x = -1$.

3. Utiliza ahora **fzero** para intentar aproximar el cero positivo:

```
fzero(fun,1)
```

Obtendrás la solución **x = 0.8924**

Como se ha dicho antes, en los casos en que la ecuación tenga varias soluciones, habrá que utilizar **fzero** una vez para cada solución que interese calcular.

Ejemplo 8.7

Calcular las soluciones de la ecuación

$$\sin\left(\frac{x}{2}\right) \cos(\sqrt{x}) = \frac{1}{5} \quad \text{en } [0, \pi].$$

1. El primer paso es escribir la ecuación en forma homogénea ($f(x) = 0$). Por ejemplo

$$f(x) = \sin\left(\frac{x}{2}\right) \cos(\sqrt{x}) - \frac{1}{5} = 0$$

2. Comienza por definir una función anónima y hacer la gráfica:

```
fun = @(x) sin(x/2).*cos(sqrt(x))- 0.2;
x = linspace(0,pi);
plot(x,fun(x));
grid on
```

Comprobarás que hay una solución en el intervalo $[0.4, 0.7]$ y otra en el intervalo $[1.5, 2]$.

3. Utiliza ahora **fzero** para intentar aproximar cada una de ellas:

fzero(fun, [0.4, 0.7])	% sol. x = 0.5490
fzero(fun, [1.5, 2])	% sol. x = 1.6904

Cuando se utiliza la gráfica por ordenador para «localizar» las soluciones de una ecuación es preciso prestar especial atención a los factores de escala de los ejes en el dibujo y hacer sucesivos dibujos «acerándose» a determinadas zonas de la gráfica para comprender la situación. Asimismo, siempre es recomendable hacer previo análisis teórico de la función. Como ejemplo de esta situación, véase el ?? y el ?? al final del capítulo.

8.5 Ceros de funciones definidas por un conjunto discreto de valores

En ocasiones, como ya se ha visto antes, es preciso trabajar con funciones de las que sólo se conocen sus valores en un conjunto finito de puntos.

Para calcular (de forma aproximada) en qué punto(s) se anula una tal función se pueden interpolar sus valores por alguno de los procedimientos estudiados en el Tema anterior, definir una función anónima con la interpolante y calcular con **fzero** el cero de esta función.

Ejercicio 8.8 El fichero **Datos.dat** contiene, en dos columnas, las abscisas y las ordenadas de un conjunto de puntos correspondientes a los valores de una función.

Se desea interpolar dichos valores mediante una interpolante lineal a trozos y calcular el valor para el que dicha interpolante toma el valor 0.56.

1. Comienza por leer los datos del fichero:

```
mat = load('Datos.dat');
xs = mat(:,1);
ys = mat(:,2);
```

2. Si denotamos por $g(x)$ a la interpolante, lo que tenemos que hacer es resolver la ecuación

$$g(x) = 0.56 \Leftrightarrow g(x) - 0.56 = 0$$

Necesitamos definir una función anónima que evalúe el primer miembro de la anterior ecuación

```
fun = @(x) interp1(xs,ys,x) - 0.56;
```

3. Ahora utilizamos **fzero** para calcular un cero de esta función partiendo, por ejemplo, del punto medio del intervalo ocupado por los **xs** (se podría elegir cualquier otro).

```
fzero(fun,mean(x))
```

Obtendrás el valor $x = 0.6042$.

Ejercicio 8.9 Repite la práctica anterior, pero interpolando mediante un spline cúbico.

1. Igual que antes, lee los datos del fichero:

```
mat = load('Datos.dat');
xs = mat(:,1);
ys = mat(:,2);
```

2. La función anónima ahora se podría escribir

```
fun = @(x) spline(xs,ys,x)-0.56;
```

Sin embargo, sería mas adecuado lo siguiente:

```
coef = spline(xs,ys); % calcula coeficientes del spline
fun = @(x) ppval(coef,x)-0.56; % evalúa el spline en x
```

Esta segunda opción es mejor porque con ella los coeficientes se calculan una sola vez.

3. Ahora utiliza **fzero** igual que antes

```
fzero(fun,mean(x))
```

Obtendrás el valor $x = 0.6044$.

8.6 Algoritmos numéricos para la resolución de ecuaciones

Los algoritmos numéricos para resolver el problema

$$\text{Hallar } x \in [a, b] \text{ tal que } f(x) = 0 \quad (8.4)$$

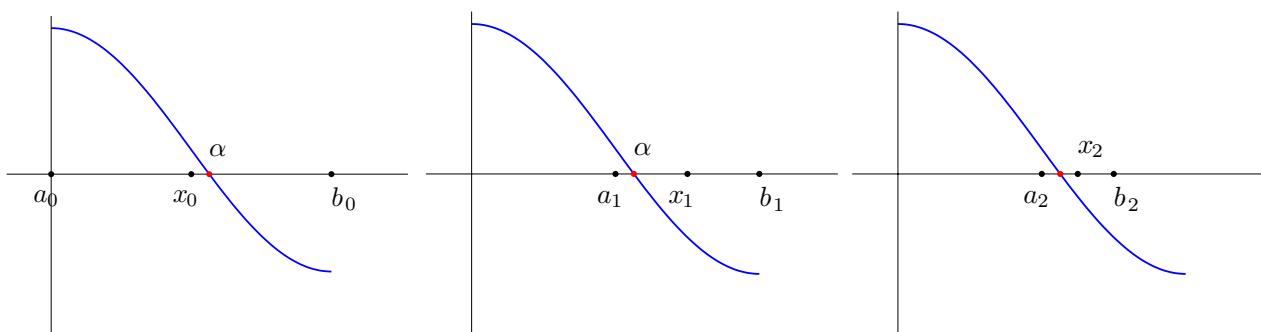
cuando la ecuación $f(x) = 0$ es no lineal (en el caso lineal su resolución es inmediata) son en general **algoritmos iterados**. Esto significa que, a partir de un punto o intervalo iniciales (dependiendo del algoritmo), se construye una sucesión de aproximaciones (calculadas una a partir de la anterior) cuyo límite es la solución buscada. Estos algoritmos, cuando convergen, lo hacen a **una** de las soluciones de la ecuación. Si ésta tuviera más de una solución, habría que utilizar el algoritmo una vez para cada una, cambiando el punto inicial. En la práctica, lógicamente, sólo se efectúan un número finito de iteraciones y el algoritmo se detiene cuando se verifica algún criterio, previamente establecido.

8.6.1 El algoritmo de bisección o dicotomía

Si f es una función continua en el intervalo $[a, b]$ y tiene signos distintos en los extremos, $f(a)f(b) < 0$, entonces posee al menos un cero en el intervalo, es decir, $\exists \alpha \in (a, b)$ tal que $f(\alpha) = 0$. Dicho cero no tiene porqué ser único (salvo que se tengan condiciones adicionales como, por ejemplo, la monotonía estricta de la función).

Sin mucha precisión, el método de bisección consiste en lo siguiente:

- 1) Subdividir en dos partes el intervalo en que se sabe que la función cambia de signo y tiene una sola raíz.
- 2) Averiguar, utilizando el Teorema de Bolzano, en cual de las dos mitades se encuentra la raíz y descartar la otra mitad del intervalo.
- 3) Reiniciar el proceso con el subintervalo elegido.
- 4) Continuar este proceso hasta que el subintervalo elegido tenga una longitud lo suficientemente pequeña como para que cualquiera de sus puntos sea una aproximación aceptable de la solución. La elección óptima como aproximación es, entonces, el punto medio del intervalo.



Algoritmo de bisección (versión 1)

- a) Datos de entrada: a, b y $tol > 0$.
Hacer $e = (b - a)/2$.
- b) Dados a, b y e , hacer $x = \frac{a + b}{2}$
 - b.1) Si $|e| \leq tol$, parar y devolver x como aproximación de la solución.
 - b.2) Si $f(a)f(x) < 0$, hacer $b = x$.
 - b.3) Si no, hacer $a = x$
- c) Hacer $e = e/2$ y volver al paso b).

En el algoritmo anterior, cabe plantearse la posibilidad de que, en alguna de las iteraciones, sea $f(x) = 0$. En ese caso, lo lógico sería detener el algoritmo y dar x como solución.

Habría, pues, que introducir, después del paso b1), un test para detectar si $f(x) = 0$ y, en caso afirmativo, parar.

Observación: cuando este algoritmo se implementa en un programa, para ser ejecutado en un ordenador, el test mencionado antes **no debe consistir en preguntar si $f(x)$ es igual a 0** ya que, debido a los errores de redondeo, es prácticamente imposible que un número real que resulte de hacer cálculos sea exactamente igual a cero: será **muy pequeño en valor absoluto**.

Estas consideraciones dan lugar a la siguiente versión (mejorada) del algoritmo:

Algoritmo de bisección (versión 2)

- a) Datos de entrada: $a, b, \varepsilon > 0$ y $tol > 0$.
Hacer $e = (b - a)/2$.
- b) Dados a, b y e , calcular $x = \frac{a + b}{2}$
 - b.1) Si $|e| \leq tol$ o bien $|f(x)| < \varepsilon$, parar y devolver x como aproximación de la solución.
 - b.2) Si $f(a)f(x) < 0$, hacer $b = x$.
 - b.3) Si no, hacer $a = x$
- c) Hacer $e = e/2$ y volver al paso b).

Ejercicio 8.10 (Método de bisección) Escribir una M-función que aproxime la solución de $f(x) = 0$ en el intervalo $[a, b]$ utilizando el método de bisección.

```
function [x] = Biseccion(fun, a, b, epsi, tol)
%
% Biseccion(fun,a,b,epsi,tol) devuelve una aproximacion del
% unico cero de la funcion fun entre a y b
% La funcion fun debe ser continua y tener signos
% distintos en a y b
%
% Metodo utilizado: biseccion
% Argumentos de entrada:
% fun      : un handle a una funcion
% a, b     : extremos de un intervalo con un cero de fun
% epsi     : se detiene el algoritmo si |fun(x)| < epsi
% tol      : se detiene el algoritmo si |x-sol_exacta| < tol
% Argumentos de salida:
% x        : la aproximacion de la solucion
%

fa = fun(a);
fb = fun(b);
if ( fa*fb > 0 )
    error('La funcion debe cambiar de signo en el intervalo');
end

e = (b-a)*0.5;
x = (a+b)*0.5;

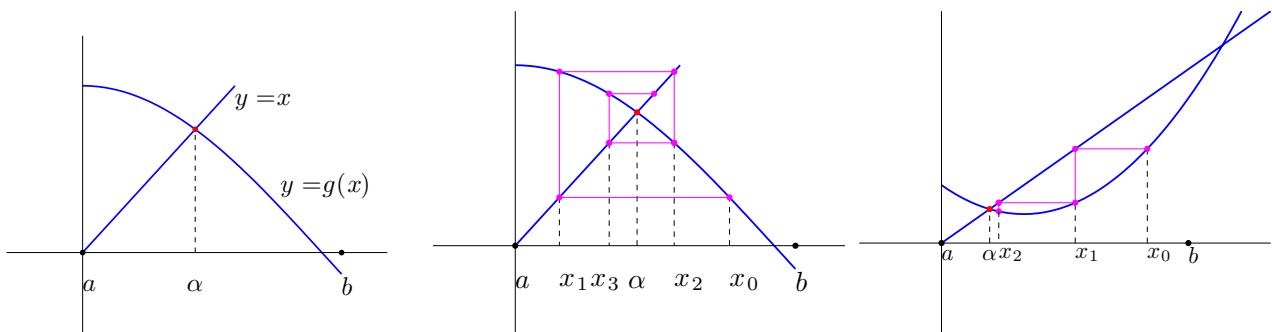
while ( abs(e) >= tol )
    fx = fun(x);
    if ( abs(fx) < epsi )
        return
    end
    if (fa*fx > 0)
        a = x;
        fa = fx;
    else
        b = x;
        fb = fx;
    end
    x = (a+b)*0.5;
    e = e*0.5;
end
```

8.6.2 El método de aproximaciones sucesivas

Dada $g : [a, b] \subset \mathbb{R} \rightarrow \mathbb{R}$ continua, el método de aproximaciones sucesivas sirve para aproximar soluciones de una **ecuación de punto fijo**

$$x = g(x), \quad x \in [a, b].$$

A las soluciones de $x = g(x)$ se les llama **puntos fijos** de la función g . Geométricamente hallar un punto fijo de g es determinar la abscisa del punto de corte de las gráficas de $y = g(x)$ e $y = x$ en $[a, b]$.



El **método de aproximaciones sucesivas** es un método iterativo que consiste en tomar una aproximación inicial $x_0 \in [a, b]$ y calcular los demás términos de la sucesión $\{x_n\}_{n \geq 0}$ mediante la relación

$$x_{n+1} = g(x_n).$$

Es obvio que cualquier ecuación $f(x) = 0$ puede escribirse en la forma $x = g(x)$ (por ejemplo $x = x + f(x)$, pero también de muchas otras formas). Bajo ciertas condiciones sobre la función g el método de aproximaciones sucesivas es convergente con orden de convergencia lineal (orden 1).

Cuando se utiliza el método de aproximaciones sucesivas para calcular una aproximación del punto fijo, se suelen terminar las iteraciones cuando el valor absoluto de la diferencia entre dos puntos sucesivos sea menor que una tolerancia pre-establecida, ε :

$$|x_n - x_{n-1}| \leq \varepsilon.$$

Por otra parte, para prevenir el caso en que el algoritmo no converja por alguna razón, es preciso imponer que el programa se detenga tras realizar un número máximo prefijado de iteraciones sin que se haya obtenido la convergencia.

Algoritmo de aproximaciones sucesivas (AASS) (para aproximar la solución de $x = g(x)$)

- Elegir $x_0 \in [a, b]$ y $\varepsilon > 0$. Hacer $n = 0$.
- Dados $n \geq 0$ y x_n .
 - Calcular $x_{n+1} = g(x_n)$
 - Si $|x_{n+1} - x_n| \leq \varepsilon$, parar y devolver x_{n+1} como aproximación.
 - Hacer $n = n + 1$ y repetir el paso b).

Ejercicio 8.11 (Método de aproximaciones sucesivas) Escribir una M-función que aproxime una solución de $x = g(x)$ utilizando el método de aproximaciones sucesivas.

```

function [x] = AASS(fun, xcero, epsi, Nmax)
%
% x = AASS(fun,xcero,tol,Nmax) devuelve una aproximacion de
%       un punto fijo de fun(x), i.e. una solucion de x=fun(x)
%
% Metodo utilizado: aproximaciones sucesivas
% Argumentos:
% xcero      : punto para iniciar el algoritmo
% epsi       : se detiene el algoritmo si |x_{n+1}-x_n| < epsi
% Nmax       : numero maximo de iteraciones a realizar
%

x = xcero;
for k = 1:Nmax
    x0 = x;
    x = fun(x);
    if ( abs(x-x0) < epsi )
        return
    end
end
%

```

8.6.3 El método de Newton

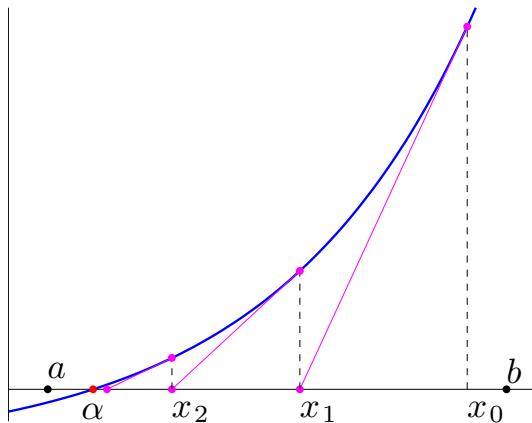
El método de Newton para aproximar la solución de la ecuación

$$f(x) = 0$$

consiste en generar una sucesión $\{x_n\}_{n \geq 0}$ construida a partir de un valor inicial x_0 mediante el método iterado:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Obviamente, el método de Newton necesita del conocimiento de la derivada $f'(x)$ y que esta no se anule en ningún término de la sucesión. La interpretación geométrica del método de Newton es la siguiente: x_{n+1} es la abscisa del punto de intersección con el eje OX de la tangente a la curva $y = f(x)$ en el punto $(x_n, f(x_n))$



Tomando el punto inicial x_0 cercano a la solución, bajo ciertas condiciones sobre la función f , el método de Newton es convergente con orden de convergencia cuadrático (de orden 2).

La forma de detener las iteraciones de este método para obtener una aproximación de la raíz es similar al método de aproximaciones sucesivas:

$$|x_{n-1} - x_n| < \varepsilon.$$

Por otra parte, para prevenir el caso en que el algoritmo no converja por alguna razón, es preciso imponer que el programa se detenga tras realizar un número máximo prefijado de iteraciones sin que se haya obtenido la convergencia.

Algoritmo de Newton (para aproximar la solución de $f(x) = 0$)

- a) Elegir $x_0 \in [a, b]$, $tol > 0$ y $\varepsilon > 0$.
- b) Dados $n \geq 0$ y x_n .
 - b.1) Hacer $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.
 - b.2) Si $|x_{n+1} - x_n| < tol$ o bien $|f(x_{n+1})| < \varepsilon$, parar y devolver x_{n+1} como aproximación.

Ejercicio 8.12 Escribir una M-función que aproxime la solución de $f(x) = 0$ utilizando el método de Newton.

```
function [x] = Newton(fun, dfun, xcero, tol, epsi, Nmax)
%
%   x = Newton(fun, dfun, xcero, tol, epsi, Nmax) devuelve una
%       aproximacion de la solucion de fun(x) = 0
%
% Metodo utilizado: Newton
% Argumentos:
%   fun      : un handle a una funcion
%   dfun     : un handle a la derivada de fun
%   xcero    : punto para iniciar el algoritmo
%   tol      : se detiene el algoritmo si |x_{n+1}-x_n| < tol
%   epsi     : se detiene el algoritmo si |fun(x)| < epsi
%   Nmax    : numero maximo de iteraciones a realizar
%
```

9

Integración numérica



Nos planteamos aquí el cálculo de integrales definidas

$$\int_a^b f(x) dx$$

Si se conoce una primitiva, F , de la función f , es bien sabido que el valor de la integral definida se puede calcular mediante la *Regla de Barrow*:

$$\int_a^b f(x) dx = F(b) - F(a).$$

En la mayoría de los casos, sin embargo, no se puede utilizar esta fórmula, ya que no se conoce dicha primitiva. Es posible, por ejemplo, que no se conozca la expresión matemática de la función f , sino sólo sus valores en determinados puntos. Pero también hay funciones (de apariencia sencilla) para las que se puede demostrar que no tienen ninguna primitiva que pueda escribirse en términos de funciones elementales, como por ejemplo $f(x) = e^{-x^2}$.

La **integración numérica** es una herramienta de las matemáticas que proporciona **fórmulas y técnicas** para calcular aproximaciones de integrales definidas. Gracias a ella se pueden calcular, aunque sea de forma aproximada, valores de integrales definidas que no pueden calcularse analíticamente y, sobre todo, se puede realizar ese cálculo en un ordenador.

9.1 La función integral (función quad en versiones anteriores)

MATLAB dispone de la función **integral** para calcular integrales definidas¹

```
integral(fun,a,b);
```

a, b son los límites de integración

¹En antiguas versiones de MATLAB, la función básica para el cálculo numérico de integrales definidas era **quad**, que ha sido sustituida por **integral**, que, aunque tiene más funcionalidades, funciona igual que **quad** en los casos en que esta última funcionaba.

fun es la función a integrar y puede ser especificada de dos formas:

integral(fun,a,b) mediante una función anónima

integral(@fun,a,b) mediante una M-función

Ejercicio 9.1 Calcular la integral definida

$$\int_{0.2}^3 x \sin(4 \ln(x)) dx$$

```
f = @(x) x.*sin(4*log(x));
q = integral(f,0.2,3)
```

Es preciso que la función del integrando esté vectorizada. Debes obtener el valor -0.2837. También se podría escribir, directamente,

```
q = integral(@(x) x.*sin(4*log(x)), 0.2, 3)
```

Ejercicio 9.2 Calcular la siguiente integral definida utilizando una M-función para representar el integrando

$$\int_0^8 (x e^{-x^{0.8}} + 0.2) dx$$

Usando la función **area**, representar gráficamente el área por debajo de la curva de la función a integrar, las rectas $x = 0$ y $x = 8$ y el eje OX . Consultar para ello el help de MATLAB.

1. Escribe, en un fichero de nombre **mifun.m**, una M-función que evalúe la función a integrar:

```
function [y] = mifun(x)
y = x.*exp(-x.^0.8) + 0.2;
```

2. Para calcular la integral hay que usar la orden:

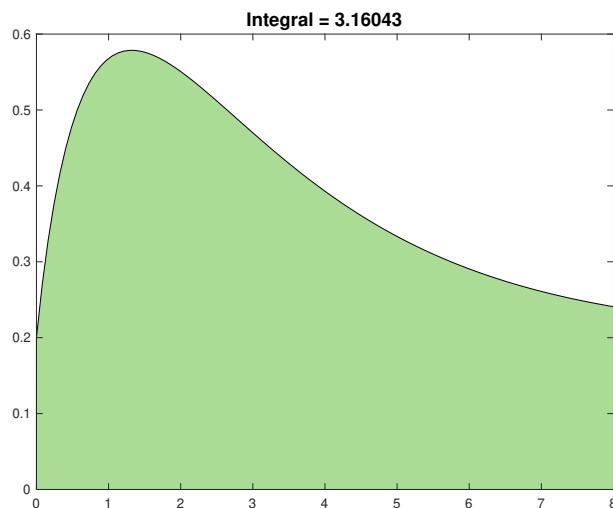
```
q = integral(@mifun, 0,8);
```

Recuerda que, puesto que el integrando viene definido mediante una M-función, es necesario poner el símbolo **@** delante del nombre al transmitirlo a otra función.

3. La gráfica pedida se puede hacer con las órdenes siguientes. Incluimos un título que utilizamos para mostrar el valor calculado de la integral.

La función $f(x) = x e^{-x^{0.8}} + 0.2$ es positiva en el intervalo $[0, 8]$, por lo tanto el valor de la integral calculada es el área de la región mostrada en la figura.

```
x = linspace(0,8);
y = mifun(x);
area(x, y, 'FaceColor', [0.1,0.8,0])
title(['Integral=' , num2str(q,'%12.5f')], 'FontSize', 14);
```



Ejercicio 9.3 (Propuesta) Calcular la integral definida del Ejercicio 9.1, usando una M-funcióñ

Ejercicio 9.4 Escribir una M-funcióñ

```
function [v] = ValorInt(k)
```

que calcule y devuelva el valor de la integral

$$\int_{0.5}^7 x \sin(4 \ln(kx)) dx$$

9.2 Aplicaciones de la integral definida

9.2.1 Cálculo de áreas

Como es bien sabido, si $f : [a, b] \mapsto \mathbb{R}$, es continua y $f > 0$ en $[a, b]$, entonces el área A de la región delimitada por la curva de ecuación $y = f(x)$, el eje OX y las rectas verticales $x = a$ y $x = b$ viene dada por

$$A = \int_a^b f(x) dx$$

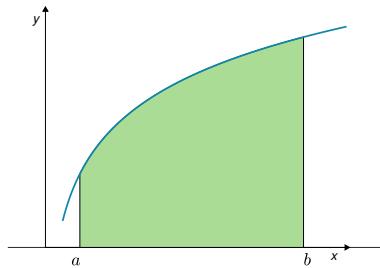
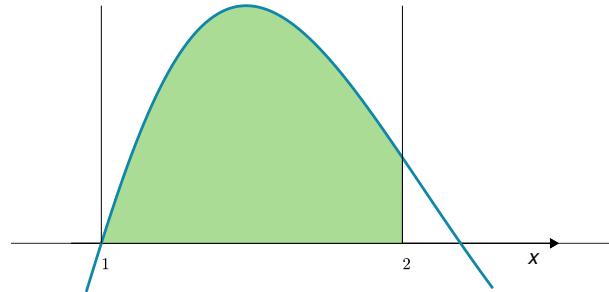


Figura 9.1: Región delimitada por la gráfica de la función $y = f(x)$, el eje de abscisas y las rectas $x = a$ y $x = b$.

Ejercicio 9.5 Calcular el área de la región plana delimitada por la curva de ecuación $y = \sin(4 \ln(x))$, el eje OX y las rectas verticales $x = 1$ y $x = 2$.

1. Comenzamos analizando y representando gráficamente la función $y = \sin(4 \ln(x))$ para asegurarnos de que es positiva en el intervalo $[1, 2]$:

```
f = @(x) sin(4*log(x));
x1 = linspace(0.95,2.3);
y1=f(x1);
plot(x1,y1)
grid on
```



2. Puesto que f es positiva en $[1, 2]$, el área de la región mencionada es:

$$A = \int_1^2 f(x) dx$$

y se calcula con la orden

```
A = integral(f,1,2)
```

Obtendrás el valor: $A = 0.7166$.

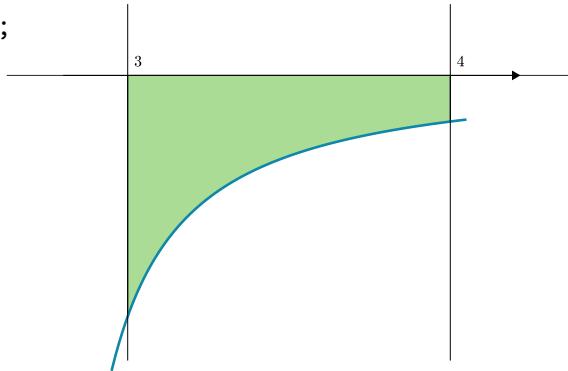
Si la función f es negativa en $[a, b]$, entonces el área de la región delimitada por la curva $y = f(x)$, el eje OX y las rectas verticales $x = 1$ y $x = b$ es

$$A = - \int_a^b f(x) dx$$

Ejercicio 9.6 Calcular el área de la región plana delimitada por la curva de ecuación $y = \frac{1}{x(1 - \ln(x))}$, el eje OX y las rectas verticales $x = 3$ y $x = 4$.

1. De nuevo comenzamos representando gráficamente la función para analizar su signo:

```
f = @(x) 1./(x.*(1-log(x)));
x1 = linspace(3, 4);
y1 = f(x1);
plot(x1, y1)
grid on
```



2. Puesto que f es negativa en $[3, 4]$, el área de la región mencionada es:

$$A = - \int_3^4 f(x) dx$$

y se calcula con la orden

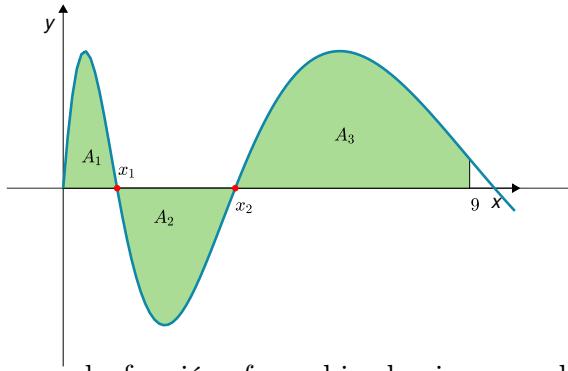
```
A = - integral(f,3,4)
```

Obtendras el valor: **area = 1.3654.**

Ejercicio 9.7 Calcular el área total de la región plana delimitada por la curva de ecuación $y = \sin(4 \ln(x+1))$ y el eje OX entre los puntos de abscisas $x = 0$ y $x = 9$.

1. Comenzamos por representar gráficamente la función para analizar su signo:

```
f = @(x) sin(4*log(x+1));
x1 = linspace(0,10);
y1 = f(x1);
plot(x1,y1)
grid on
```



2. La observación de la gráfica nos muestra que la función f cambia de signo en dos puntos del intervalo $[0, 9]$, x_1 y x_2 , y que es positiva en $[0, x_1]$, negativa en $[x_1, x_2]$ y de nuevo positiva en $[x_2, 9]$.

Tenemos que calcular las tres áreas por separado y para ello lo primero es calcular los valores x_1 y x_2 :

```
xcero1 = fzero(f,1); % xcero1 = 1.1933
xcero2 = fzero(f,4); % xcero2 = 3.8105
```

3. Calculamos ahora las tres áreas:

$$A_1 = \int_0^{x_1} f(x) dx \quad A_2 = - \int_{x_1}^{x_2} f(x) dx \quad A_3 = \int_{x_2}^9 f(x) dx$$

```
A1 = integral(f,0,xcero1); % A1 = 0.7514
A2 = - integral(f,xcero1,xcero2); % A2 = 1.6479
A3 = integral(f,xcero2,9); % A3 = 3.5561
```

4. El área total buscada es finalmente:

```
A = A1 + A2 + A3; % A = 5.9554
```

Ejercicio 9.8 Calcular el área de la región del primer cuadrante delimitada por la curva $y = x \operatorname{sen}\left(6 \log\left(\frac{x}{2}\right)\right)$, el eje OX y las rectas verticales $x = 1$ y $x = 10$.

Ejercicio 9.9 Calcular el área de la región delimitada por la curva

$$y = \frac{1}{\sin x},$$

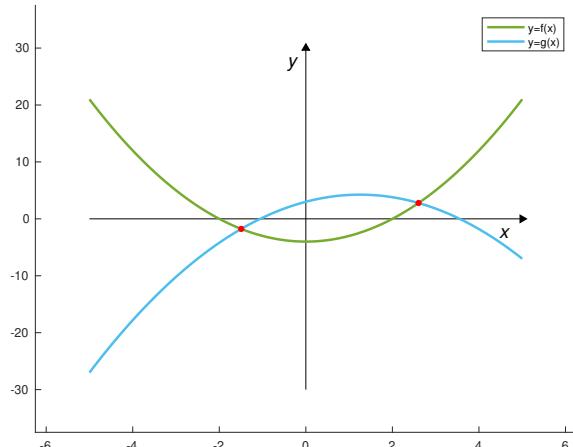
el eje OX , el eje OY , la recta horizontal $y = 5$ y la recta vertical $x = 1.5$.

Ejercicio 9.10 Calcular los puntos de corte de las curvas siguientes, así como el área de la región plana encerrada entre ellas

$$y = x^2 - 4 = f(x) \quad \text{y} \quad y = 2x - 0.8x^2 + 3 = g(x)$$

1. Tenemos que comenzar por identificar la zona en cuestión y hacernos una idea de la localización de los puntos de corte. Para ello vamos a dibujar ambas curvas, por ejemplo, entre $x = -5$ y $x = 5$ (podrían ser otros valores):

```
f = @(x) x.^2-4;
g = @(x) 2*x - 0.8*x.^2 +3;
x = linspace(-5,5);
yf = f(x);
yg = g(x);
plot(x,yf,x,yg)
grid on
```



2. La observación de la gráfica nos muestra que ambas curvas se cortan en dos puntos, (x_1, y_1) y (x_2, y_2) .

Para calcularlos comenzamos por calcular sus abscisas x_1 y x_2 , que son las soluciones de la ecuación

$$f(x) = g(x) \quad \text{equivalente a} \quad f(x) - g(x) = 0$$

```
fg = @(x) f(x) - g(x);
x1 = fzero(fg,-1); % x1 = -1.4932
x2 = fzero(fg, 3); % x2 = 2.6043
```

Las ordenadas de los puntos de corte son los valores en estos puntos de la función f (o g , ya que toman el mismo valor en ellos):

$y_1 = f(x_1);$	$\% \text{ o bien } y_1 = g(x_1);$	$y_1 = -1.7703$
$y_2 = f(x_2);$	$\% \text{ o bien } y_2 = g(x_2);$	$y_2 = 2.7826$

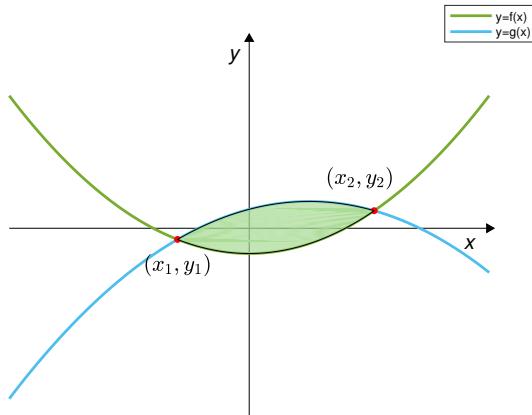
Así pues, las dos curvas se cortan en los puntos $(-1.4932, -1.7703)$ y $(2.6043, 2.7826)$.

3. El área de la región que queda encerrada entre las dos curvas es, ya que en $[x_1, x_2]$ se tiene $g(x) \geq f(x)$:

$$A = \int_{x_1}^{x_2} (g(x) - f(x)) dx = - \int_{x_1}^{x_2} (f(x) - g(x)) dx$$

y se puede calcular con la orden:

```
A = - integral(fg,x1,x2); % A = 20.6396
```



Ejercicio 9.11 Calcular los puntos de corte de las curvas siguientes, así como el área de la región plana encerrada entre ellas

$$y = \frac{\cos x}{x^2 + 1} = f(x) \quad \text{y} \quad y = 0.3 \left| x - \frac{1}{2} \right| = g(x)$$

Ejercicio 9.12 Calcular el área de la región plana delimitada por las rectas verticales $x = 2$ y $x = 3$ y las gráficas de las funciones

$$f(x) = \operatorname{tg}(x) \quad \text{y} \quad g(x) = -\frac{x}{3}$$

Ejercicio 9.13 Las funciones

$$f(t) = \frac{t}{1 + \ln(4 + t^2)} \quad \text{y} \quad g(t) = \frac{t}{6}$$

se cortan en el punto $(0, 0)$ y en otro punto (\bar{x}, \bar{y}) del primer cuadrante. Se pide calcular las coordenadas de dicho punto y el área de la región encerrada entre ambas curvas.

Ejercicio 9.14 Las funciones

$$f(t) = t \operatorname{sen}(\sqrt{5t}) \quad \text{y} \quad g(x) = 0.5 - t$$

se cortan en dos puntos del intervalo $[\pi, 2\pi]$. Se pide calcular las coordenadas de dichos puntos así como el área de la región encerrada entre ambas curvas.

Ejercicio 9.15 Calcula el área de la o las regiones planas delimitadas por las curvas

$$y = \cos^2(x) \quad \text{e} \quad y = \operatorname{sen}(x)$$

y las rectas verticales $x = 0$ y $x = 3$.

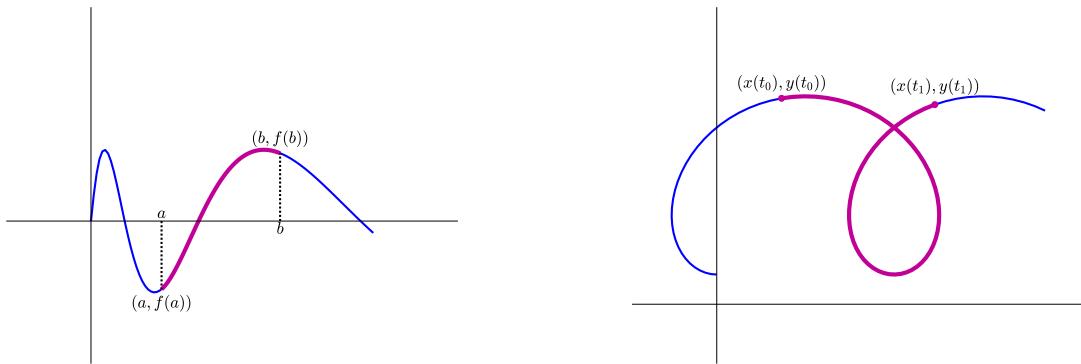
9.2.2 Cálculo de longitudes de arcos de curva

Consideremos el arco de curva dado por

$$y = f(x) \quad x \in [a, b].$$

La longitud de este arco de curva desde $(a, f(a))$ hasta $(b, f(b))$ viene dada por

$$L = \int_a^b \sqrt{1 + (f'(x))^2} dx \tag{9.1}$$



Si la curva viene definida por sus ecuaciones paramétricas

$$\begin{cases} x = f(t) \\ y = g(t) \end{cases} \quad t \in [t_0, t_1]$$

entonces la longitud del arco de curva desde $(x(t_0), y(t_0))$ hasta $(x(t_1), y(t_1))$ viene dada por

$$L = \int_{t_0}^{t_1} \sqrt{(f'(t))^2 + (g'(t))^2} dt \tag{9.2}$$

Ejercicio 9.16 Calcular la longitud del arco de la curva $y = \operatorname{sen}(4 \ln(x+1))$ entre $x = 0$ y $x = 9$.

Denotando $f(x) = \operatorname{sen}(4 \ln(x+1))$, se tiene

$$f'(x) = \frac{4}{x+1} \cos(4 \ln(x+1))$$

La longitud del arco indicado se puede calcular con las órdenes:

```
df    = @(x) 4*cos(4*log(x+1))./(x+1);
fint = @(x) sqrt(1+df(x).^2);
L = integral(fint, 0, 9)
```

Ejercicio 9.17 Calcular la longitud del arco de la curva definida por las ecuaciones paramétricas

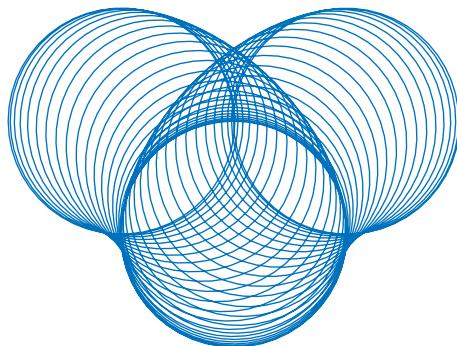
$$\begin{cases} x = \cos(80t) - \cos^3(t) \\ y = \operatorname{sen}(80t) - \operatorname{sen}^3(t) \end{cases} \quad t \in [0, \pi]$$

Denotando $f(t) = \cos(80t) - \cos^3(t)$ y $g(t) = \operatorname{sen}(80t) - \operatorname{sen}^3(t)$, se tiene

$$\begin{cases} f'(t) = -80 \operatorname{sen}(80t) + 3 \cos^2(t) \operatorname{sen}(t) \\ g'(t) = 80 \cos(80t) - 3 \operatorname{sen}^2(t) \cos(t) \end{cases}$$

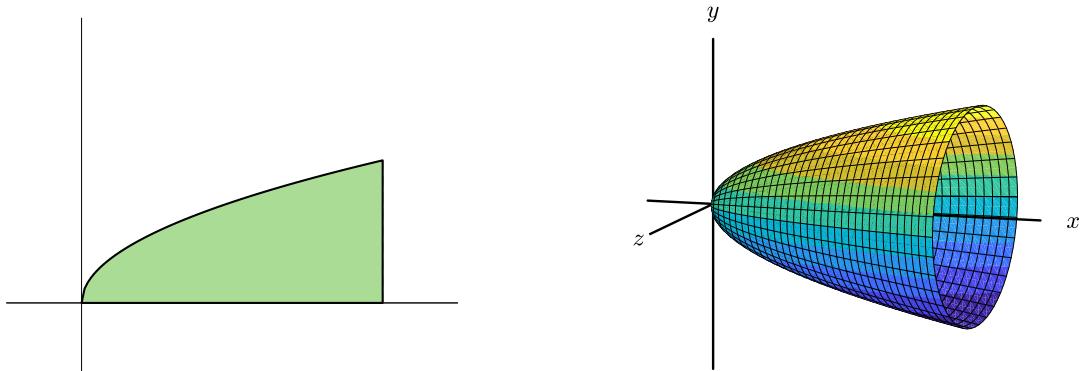
La longitud del arco indicado se puede calcular con las órdenes:

```
df    = @(t) -80*sin(80*t) + 3*cos(t).^2.*sin(t);
dg    = @(t) 80*cos(80*t) - 3*sin(t).^2.*cos(t);
fint = @(x) sqrt(df(x).^2 + dg(x).^2);
L = integral(fint, 0, pi)
```



9.3 Cálculo de volúmenes y superficies de revolución

Una figura que se genera por la rotación de una región plana alrededor de un eje se llama sólido de revolución. La rotación de una curva plana genera una superficie. Esta superficie junto con su interior es un sólido de revolución. Por ejemplo, la superficie de un cilindro puede ser obtenida por la rotación de un segmento paralelo al eje; la de un cono, por la rotación de su generatriz alrededor del eje o la de una esfera por la rotación de una semicircunferencia en torno a su diámetro.



El volumen del sólido de revolución generado por rotación de la región determinada por la curva $y = f(x)$, las rectas verticales $x = a$ y $x = b$ y el eje OX en torno a dicho eje viene dado por

$$V = \pi \int_a^b f(x)^2 dx$$

El área de la superficie de revolución generada al girar el arco de curva $y = f(x)$, $x \in [a, b]$ alrededor del eje OX viene dada por

$$S = 2\pi \int_a^b f(x) \sqrt{1 + (f'(x))^2} dx$$

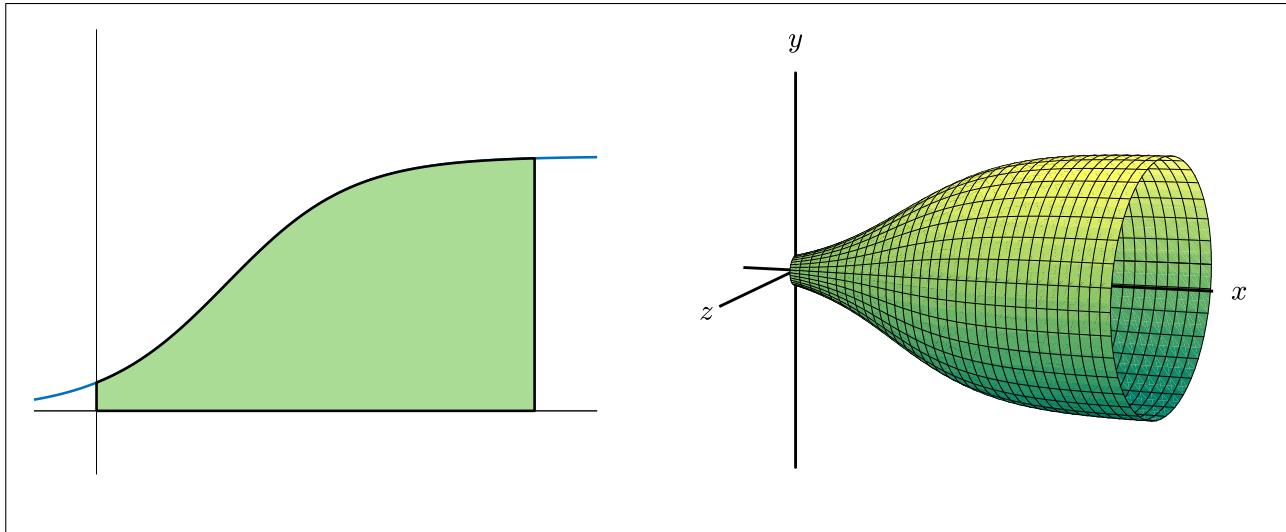
Ejercicio 9.18 Calcular el volumen del sólido de revolución generado por rotación alrededor del eje OX de la región plana delimitada por la curva

$$y = \frac{4}{1 + 8e^{-t}}, \quad t \in [0, 7],$$

las rectas verticales $x = 0$ y $x = 7$ y el eje OX .

El volumen pedido se puede calcular con las órdenes

```
f = @(x) 4./(1+8*exp(-x));
V = pi*integral(@(x) f(x).^2, 0, 7); % V = 197.4628
```



Ejercicio 9.19 Calcular el volumen del sólido de revolución (elipsoide) generado por rotación alrededor del eje OX de la región plana encerrada dentro de la elipse de ecuación

$$(x - 3)^2 + 3y^2 = 4.$$

Calcular también el área superficial del elipsoide.

En la ecuación implícita de la elipse, se puede despejar la variable y en función de la variable x :

$$y = \pm \sqrt{\frac{4 - (x - 3)^2}{3}}$$

El sólido mencionado se obtiene por rotación alrededor del eje OX de la parte positiva

$$y = + \sqrt{\frac{4 - (x - 3)^2}{3}}$$

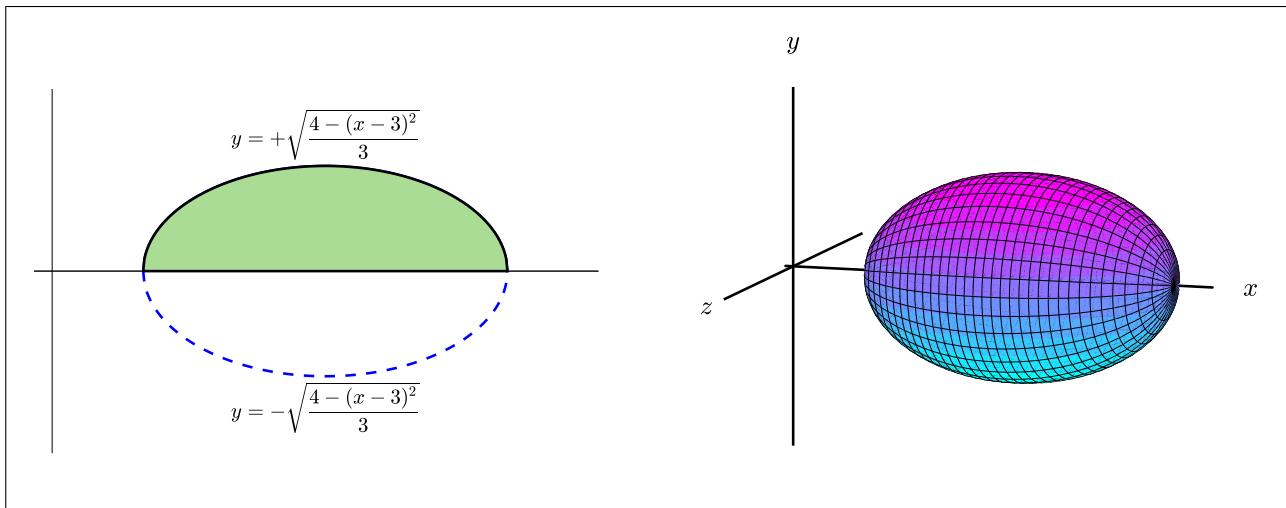
Se puede calcular su volumen con las órdenes

```
f = @(x) sqrt((4 - (x-3).^2)/3);
V = pi * integral(@(x) f(x).^2, 1, 5) % V = 11.1701
```

Para calcular el área superficial necesitamos la derivada de la función f :

$$y = -\frac{1}{\sqrt{3}} \frac{x - 3}{\sqrt{4 - (x - 3)^2}}$$

```
df = @(x) -r3 * (x-3) ./ sqrt(4-(x-3).^2);
fint = @(x) f(x).*sqrt( 1 + df(x).^2);
S = 2*pi*integral(fint, 1, 5) % S = 25.3550
```



Ejercicio 9.20 Calcular el volumen del toroide generado por rotación alrededor del eje OX de la región plana encerrada dentro de la elipse de ecuación

$$x^2 + 3(y - 3)^2 = 4.$$

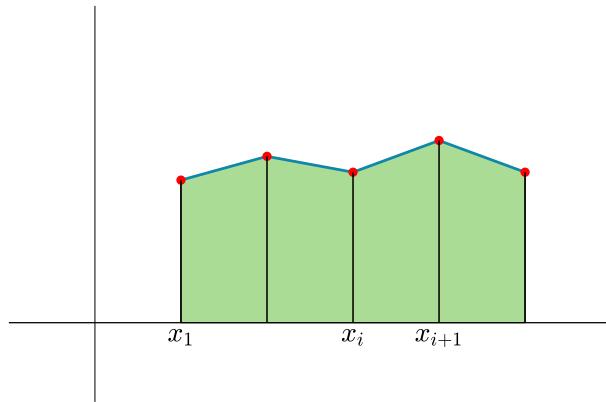
Calcular también su área superficial.

9.4 Cálculo de integrales de funciones definidas por un conjunto discreto de valores

Como hemos mencionado antes, muchas veces es preciso trabajar con funciones de las que sólo se conocen sus valores en un conjunto de puntos. Para calcular (de forma aproximada) la integral definida de una tal función se pueden interpolar sus valores por alguno de los procedimientos vistos anteriormente y luego integrar dicha interpolante.

9.4.1 Integrar una poligonal: La función trapz

La posibilidad más sencilla es hacer una interpolación lineal a trozos. Entonces la integral en todo el intervalo es la suma de las integrales en cada uno de los subintervalos, que es fácil de hacer, ya que en cada uno de ellos la función a integrar es un polinomio de grado uno.



La función MATLAB `trapz` permite realizar este cálculo:

```
q = trapz(x,y);
```

`x,y` son dos vectores de la misma dimensión y representan las coordenadas de los puntos.

`q` es el valor de la integral

Ejercicio 9.21 Calcula la integral definida de una función que viene dada a través del siguiente conjunto de puntos:

x	0	2	3	5	6	8	9	11	12	14	15
y	10	20	30	-10	10	10	10.5	15	50	60	85

1. Define dos vectores que contienen las abscisas y las ordenadas de los puntos:

```
x = [0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y = [10, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Calcula el valor de la integral usando `trapz`

```
q = trapz(x,y)
```

Debes obtener el valor 345.7500.

9.4.2 Integrar otros interpolantes

Si se lleva a cabo otro tipo de interpolación, el procedimiento a seguir es definir una función (por ejemplo anónima) y luego utilizar `integral`.

Ejercicio 9.22 Se consideran los siguientes datos correspondientes a los valores de una función:

x	0	2	3	5	6	8	9	11	12	14	15
y	10	20	30	-10	10	10	10.5	15	50	60	85

Se desea interpolar dichos valores mediante un *spline* cúbico $s = s(x)$, representarlo gráficamente y calcular la integral definida:

$$\int_{10}^{14} s(x) dx$$

1. Crea dos variables x e y con las abscisas y las ordenadas que vas a interpolar:

```
x = [0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y = [0, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Calcula con la orden **spline** los coeficientes del *spline* que pasa por estos puntos:

```
c = spline(x,y);
```

3. Define una función anónima $s = s(x)$ que evalúa el *spline* en los puntos x :

```
s = @(x) ppval(c,x);
```

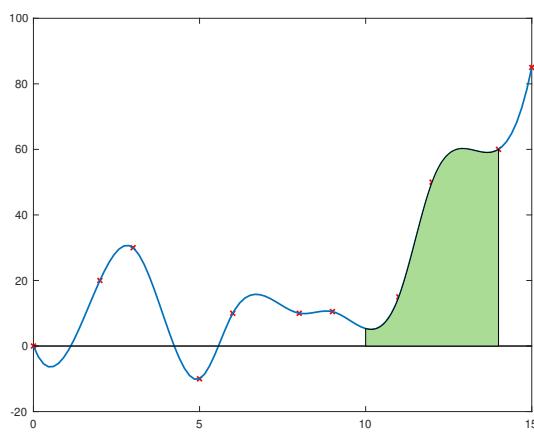
4. Calcula usando **quad** la integral definida requerida

```
a = integral(s,10,14)
```

Obtendrás el valor $a = 157.0806$.

5. Dibuja los puntos y el *spline*:

```
plot(x,y,'rx','LineWidth',1.5)
hold on
xs = linspace(min(x),max(x));
ys = s(xs);
plot(xs,ys,'LineWidth',1.5)
hold off
```



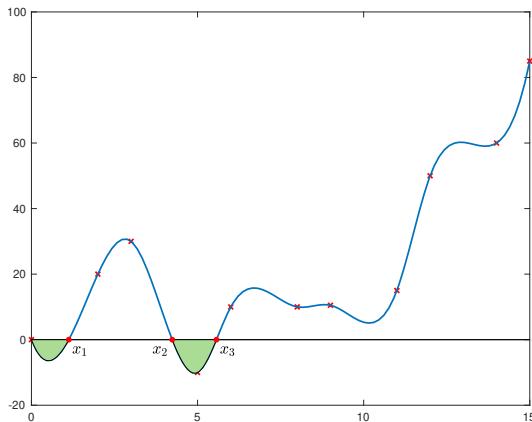
Ejercicio 9.23 Usando los datos del Ejercicio 9.22, calcular el área de la región del cuarto cuadrante ($x \geq 0$, $y \leq 0$) delimitada por el *spline* cúbico que interpola estos datos y el eje OX .

1. Crea dos variables x e y con las abscisas y las ordenadas de los puntos a interpolar:

```
x = [0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15];
y = [0, 20, 30, -10, 10, 10, 10.5, 15, 50, 60, 85];
```

2. Comienza por representar gráficamente el *spline* para analizar su signo:

```
c = spline(x,y);
s = @(x) ppval(c,x);
xs = linspace(0,15);
ys = s(xs);
plot(xs,ys)
grid on
```



3. La observación de la gráfica muestra que el *spline* cúbico cambia de signo en tres puntos del intervalo $[0, 15]$: x_1 , x_2 y x_3 y que las regiones que nos interesan son las que quedan entre $x = 0$ y $x = x_1$ por un lado, y entre $x = x_2$ y $x = x_3$ por otro. Lo primero es calcular los valores x_1 , x_2 y x_3 :

```
x1 = fzero(s,1) % x1 = 1.1240
x2 = fzero(s,4) % x2 = 4.2379
x3 = fzero(s,6) % x3 = 5.5685
```

4. Calcula ahora las dos áreas:

$$A_1 = - \int_0^{x_1} s(x) dx, \quad A_2 = - \int_{x_2}^{x_3} s(x) dx,$$

```
A1 = -integral(s,0,x1) % A1 = 4.7719
A2 = -integral(s,x2,x3); % A2 = 8.6921
```

5. El área total buscada es finalmente:

```
A = A1 + A2 % A = 13.4640
```

Ejercicio 9.24 Se considera el siguiente conjunto de puntos correspondientes a los valores de una función:

x	12	13	14	15	16	17	18	19	20
y	-11.43	-7.30	8.86	13.82	-1.76	-16.73	-8.06	13.87	17.24

Se pide calcular el área de la región delimitada por el *spline* cúbico que interpola estos datos, el eje OX y las rectas verticales $x = 12$ y $x = 20$.

Ejercicio 9.25 El fichero **datos21.dat** contiene una matriz con dos columnas, que corresponden a las abscisas y las ordenadas de una serie de datos.

Se pide leer los datos del fichero, y calcular y dibujar el *spline* cúbico que interpola dichos valores, en un intervalo que contenga todos los puntos del soporte. También se desea calcular el área de la región del primer cuadrante delimitada por el *spline* cúbico.

1. Puesto que el fichero **datos21.dat** es de texto, se puede editar. Ábrelo (como texto) para ver su estructura. Cierra el fichero.
2. Dado que todas las líneas del fichero tienen el mismo número de datos, se puede leer con la orden **load**:

```
datos = load('datos21.dat');
```

que guarda en la variable **datos** una matriz 50×2 cuya primera columna contiene los valores de las abscisas y la segunda los de las ordenadas. Compruébalo.

3. Extrae la primera columna de la matriz **datos** a un vector **x** y la segunda columna a un vector **y**:

```
x = datos(:,1);
y = datos(:,2);
```

4. Calcula los coeficientes del *spline* y guárdalos en una variable **c**

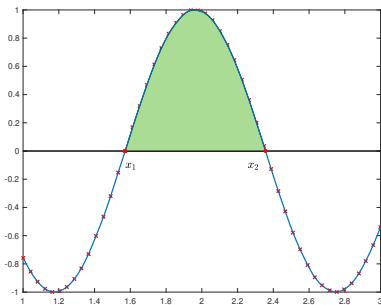
```
c = spline(x,y);
```

5. Define una función anónima que evalúe el *spline* en x :

```
f = @(x) ppval(c,x);
```

6. Dibuja el *spline* en un intervalo que contenga todos los nodos:

```
xs = linspace(min(x),max(x));
ys = f(xs);
plot(xs,ys)
grid on
```



7. En la gráfica se observa que el *spline* cúbico cambia de signo en tres puntos del intervalo $[1, 3]$: x_1 y x_2 . La región del primer cuadrante delimitada por el *spline* es la que queda entre $x = x_1$ y $x = x_2$. Comienza por calcular los valores de x_1 y x_2 :

```
x1 = fzero(f,1.5) % x1 = 1.5713
x2 = fzero(f,2.5) % x2 = 2.3566
```

8. Calcula ahora el área:

$$A_1 = \int_{x_1}^{x_2} f(x) dx$$

```
A = integral(f,x1,x2) % A = 0.5002
```

Ejercicio 9.26

Se considera el siguiente conjunto de datos:

x	0	1	2	3	4	5	6	7	8	9	10
y	0	0.28	-1.67	-2.27	1.63	4.95	0.92	-6.32	-5.33	4.72	9.64

Se desea interpolar dichos valores mediante una interpolante lineal a trozos $s(x)$, calcular la integral definida

$$\int_{\pi/2}^{2\pi} s(x) dx$$

y realizar una representación gráfica de los datos junto con la interpolante lineal a trozos.

Ejercicio 9.27

El fichero **datos22.dat** contiene una matriz con dos columnas, que corresponden a las abscisas y las ordenadas de una serie de datos.

Se pide leer los datos del fichero, y calcular la función logarítmica $f(x) = m \ln(x) + b$ que mejor se ajusta a estos valores, y dibujarla en un intervalo que contenga todos los puntos del soporte.

También se pide calcular el área de la región del plano delimitada por la gráfica de la función $f(x)$, el eje OX y las rectas verticales $x = 1.5$ y $x = 3.5$.

9.5 Funciones definidas a trozos

Los siguientes ejemplos muestran cómo vectorizar el cálculo de una función que no viene definida mediante una fórmula; por ejemplo, que está definida a trozos.

Ejercicio 9.28 Escribir una M-función para calcular la siguiente función definida a trozos:

$$f(x) = \begin{cases} 0.5x^2 & \text{si } x \geq 0 \\ x^3 & \text{si no} \end{cases}$$

y calcular el área de la región delimitada por la curva $y = f(x)$, el eje OX y las rectas verticales $x = -2$ y $x = 2.5$.

1. En un primer intento, escribiríamos probablemente algo como lo que sigue para calcular la función f :

```
function [y] = mifun(x)
if (x>=0)
    y = 0.5*x^2;
else
    y = x^3;
end
```

en un fichero de nombre **mifun.m**.

Esto sería correcto si sólo fuéramos a utilizarla siendo el argumento x un escalar. Pero no funcionaría si x fuera un vector, ya que el **if** no se lleva a cabo componente a componente.

Intenta comprender, mediante algún ejemplo y leyendo el **help, cuál sería el funcionamiento del **if** anterior si **x** fuese un vector.**

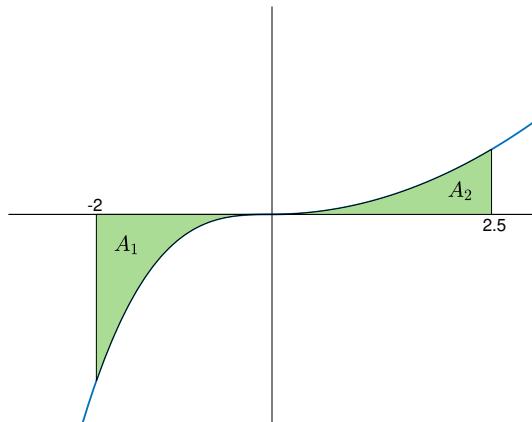
2. Para que una M-función como la anterior sea «vectorial» hay que calcular los valores de **y** componente a componente, como se muestra a continuación:

```
function [y] = mifun(x)
y = zeros(size(x));
for k = 1:length(x)
    if x(k) >= 0
        y(k) = 0.5*x(k)^2;
    else
        y(k) = x(k)^3;
    end
end
```

Escribe el código anterior en un fichero y sálvalo con el nombre **mifun.m**. Comprueba, desde la ventana de comandos, que el programa funciona correctamente, dando distintos valores a la variable de entrada.

3. Dibuja la función en el intervalo $[-3, 3]$ (por ejemplo):

```
x = linspace(-3, 3);
y = mifun(x);
plot(x, y, 'LineWidth', 1.5)
```



4. Calcula ahora las dos áreas:

$$A_1 = - \int_{-2}^0 f(x) dx, \quad A_2 = \int_0^{2.5} f(x) dx,$$

```
A1 = -integral(@mifun,-2,0) % A1 = 4
A2 = integral(@mifun,0,2.5) % A2 = 2.6042
```

5. Luego el área buscada es:

```
A = A1 + A2; % A = 6.6042
```

Observación 1

Las órdenes para realizar esta práctica se pueden escribir todas en un solo fichero de nombre, por ejemplo, **practica23.m**, de la siguiente manera (para ejecutarlo hay que invocar el programa principal, es decir, el que lleva el nombre del fichero):

```
function practica23
%-----
% Practica numero 23 : funcion "principal"
%
x = linspace(-2.5, 3);
y = mifun(x);
plot(x, y, 'LineWidth', 1.5)
axis([-2.5, 3, -15, 10])
```

```
grid on
%
A1 = - integral(@mifun, -2, 0);
A2 = integral(@mifun, 0, 2.5);
A = A1 + A2;
fprintf(' El area calculada es: %12.6f \n',A)

%-----
% mifun : funcion auxiliar utilizada por practica23
%           No es "visible" fuera de este programa
%
function [y] = mifun(x)
y = zeros(size(x));
for k = 1:length(x)
    xk = x(k);
    if xk>=0
        y(k) = 0.5*xk^2;
    else
        y(k) = xk^3;
    end
end
```

Observación 2

En realidad, utilizando adecuadamente las operaciones vectoriales de Matlab y los operadores de comparación, es posible escribir la función $f(x)$ mediante una sola expresión y, por lo tanto, como una función anónima, de la siguiente manera:

```
f = @(x) (0.5*x.^2).* (x>=0) + (x.^3).* (x<0);
```

En esta expresión se hace uso del hecho de que, cuando se combinan en una expresión aritmética datos numéricos y datos lógicos, Matlab interpreta estos últimos como ceros y unos (números).

10

Resolución de ecuaciones y sistemas diferenciales ordinarios



Una **ecuación diferencial** es una ecuación en que la incógnita es una función y que, además, involucra también las derivadas de la función hasta un cierto orden. La incógnita no es el valor de la función en uno o varios puntos, sino la función en sí misma.

Cuando la incógnita es una función de una sola variable se dice que la ecuación es ordinaria, debido a que la o las derivadas que aparecen en ella son derivadas ordinarias (por contraposición a las derivadas parciales de las funciones de varias variables).

Comenzamos con la resolución de ecuaciones diferenciales ordinarias (edo) de primer orden, llamadas así porque en ellas sólo aparecen las derivadas de primer orden.

Más adelante trataremos la resolución de sistemas de ecuaciones diferenciales ordinarias (sdo) de primer orden. Con ello podremos, en particular, abordar la resolución de ecuaciones diferenciales de orden superior a uno, ya que éstas siempre se pueden reducir a la resolución de un sistema de primer orden.

10.1 Ecuaciones diferenciales ordinarias de primer orden

Una ecuación diferencial ordinaria

$$y' = f(t, y), \quad f: \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}$$

admite, en general, infinitas soluciones. Si, por ejemplo, $f \in C^1(\Omega; \mathbb{R})$ ¹, por cada punto $(t_0, y_0) \in \Omega$ pasa una única solución $\varphi: I \rightarrow \mathbb{R}$, definida en un cierto intervalo $I \subset \mathbb{R}$, que contiene a t_0 .

Se denomina Problema de Cauchy (PC) o Problema de Valor Inicial (PVI)

$$\begin{cases} y' = f(t, y), \\ y(t_0) = y_0 \end{cases} \quad (10.1)$$

al problema de determinar, de entre todas las soluciones de la ecuación diferencial $y' = f(t, y)$, aquélla que pasa por el punto (t_0, y_0) , es decir, que verifica $y(t_0) = y_0$.

Sólo para algunos (pocos) tipos muy especiales de ecuaciones diferenciales es posible encontrar la expresión de sus soluciones en términos de funciones elementales. En la inmensa mayoría de los casos prácticos sólo es posible encontrar aproximaciones numéricas de los valores de una solución en algunos puntos.

¹En realidad basta con que f sea continua y localmente Lipschitziana con respecto de y en Ω .

Así, una **aproximación numérica** de la solución del problema de Cauchy

$$\begin{cases} y' = f(t, y), & t \in [t_0, t_f] \\ y(t_0) = y_0 \end{cases} \quad (10.2)$$

consiste en una sucesión de valores de la variable independiente:

$$t_0 < t_1 < \dots < t_n = t_f$$

y una sucesión de valores y_0, y_1, \dots, y_n tales que

$$y_k \approx y(t_k), \quad k = 0, 1, \dots, n,$$

es decir, y_k es una aproximación del valor en t_k de la solución del problema (10.2).

$$y_k \approx y(t_k), \quad k = 0, 1, \dots$$

10.2 Resolución de problemas de Cauchy para ecuaciones diferenciales ordinarias de primer orden

MATLAB dispone de toda una familia de funciones para resolver (numéricamente) ecuaciones diferenciales:

```
ode45, ode23, ode113
ode15s, ode23s, ode23t, . . .
```

Cada una de ellas implementa un método numérico diferente, siendo adecuado usar unas u otras en función de las dificultades de cada problema en concreto. Para problemas no demasiado «difíciles» será suficiente con la función **ode45** ó bien **ode23**.

Exponemos aquí la utilización de la función **ode45**, aunque la utilización de todas ellas es similar, al menos en lo más básico. Para más detalles, consultese la documentación.

Para dibujar la gráfica de la solución (numérica) de (10.2) se usará la orden:

```
ode45(@odefun, [t0,tf], y0)
```

donde:

odefun es un manejador de la función que evalúa el segundo miembro de la ecuación, $f(t, y)$. Puede ser el nombre de una función anónima dependiente de dos variables, siendo t la primera de ellas e y la segunda, o también una M-función, en cuyo caso se escribiría **@odefun**. Ver los ejemplos a continuación.

[t0,tf] es el intervalo en el que se quiere resolver la ecuación, i.e. $\mathbf{t0} = t_0$, $\mathbf{tf} = t_f$.

y0 es el valor de la condición inicial, $\mathbf{y0} = y_0$.

Ejercicio 10.1 Calcular la solución del Problema de Cauchy:

$$\begin{cases} y' = 5y \quad \text{en } [0, 1] \\ y(0) = 1 \end{cases}$$

Comparar (gráficamente) con la solución exacta de este problema, $y = e^{5t}$

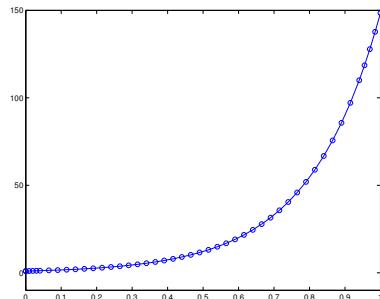
1. Comenzamos por definir una función anónima que represente el segundo miembro de la ecuación $f(t, y)$:

```
f = @(t,y) 5*y;
```

2. Calculamos y dibujamos la solución numérica:

```
ode45(f, [0,1], 1)
```

Obsérvese que no se obtiene ningún resultado numérico en salida, únicamente la gráfica de la solución.



3. Para comparar con la gráfica de la solución exacta, dibujamos en la misma ventana, sin borrar la anterior:

```
hold on
t = linspace(0,1);
plot(t, exp(5*t), 'r')
shg
```

4. Para hacer lo mismo utilizando una M-función en vez de una función anónima, tendríamos que escribir, en un fichero (el nombre **odefun** es sólo un ejemplo):

```
function [dy] = odefun(t,y)
% Segundo miembro de la ecuacion diferencial
dy = 5*y;
```

y guardar este texto en un fichero de nombre **odefun.m** (el mismo nombre que la función). Este fichero debe estar en la carpeta de trabajo de MATLAB.

Después, para resolver la ecuación, usaríamos **ode45** en la forma:

```
ode45(@odefun, [0,1], 1)
```

Si se desean conseguir los valores t_k e y_k de la aproximación numérica para usos posteriores, se debe usar la función `ode45` en la forma:

```
[t, y] = ode45(odefun, [t0,tf], y0);
```

obteniéndose así los vectores `t` e `y` de la misma longitud que proporcionan la aproximación de la solución del problema:

$$y(k) \approx y(t(k)), \quad k = 1, 2, \dots, \text{length}(y).$$

Si se utiliza la función `ode45` en esta forma, **no se obtiene ninguna gráfica**.

Ejercicio 10.2 Calcular el valor en $t = 0.632$ de la solución del problema

$$\begin{cases} y' = t e^{t/y} \\ y(0) = 1 \end{cases}$$

Calculamos la solución en el intervalo $[0, 1]$ y luego calculamos el valor en $t = 0.632$ utilizando una interpolación lineal a trozos:

```
f      = @(t,y) t.*exp(t/y);
[t,y] = ode45(f, [0,1], 1);
v      = interp1(t, y, 0.632)
```

Obsérvese que `t` e `y`, las variables de salida de `ode45`, son dos vectores columna de la misma longitud.

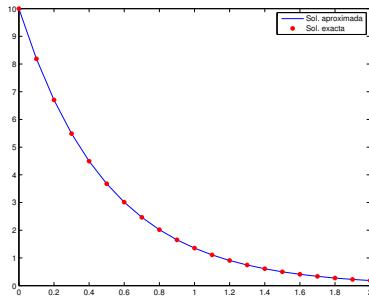
En ocasiones puede ser necesario calcular las aproximaciones y_k en unos puntos t_k pre-determinados. Para ello, en lugar de proporcionar a `ode45` el intervalo `[t0,tf]`, se le puede proporcionar un vector con dichos valores, como en el ejemplo siguiente:

Ejercicio 10.3 Calcular (aproximaciones de) los valores de la solución del problema

$$\begin{cases} y' = -2y \\ y(0) = 10 \end{cases}$$

en los puntos: 0, 0.1, 0.2, ..., 1.9, 2. Comparar (gráficamente) con la solución exacta $y = 10 e^{-2t}$.

```
f = @(t,y) -2*y;
t = 0:0.1:2;
[t,y] = ode45(f, t, 10);
plot(t, y, 'LineWidth', 1.1)
hold on
plot(t, 10*exp(-2*t), 'r.')
legend('Sol. aproximada', 'Sol. exacta')
shg
```



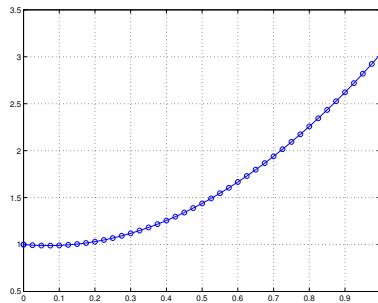
Ejercicio 10.4 Calcular el instante en que la solución del problema

$$\begin{cases} y' = 0.5(10t - \ln(y + 1)) \\ y(0) = 1 \end{cases}$$

alcanza el valor $y = 1.5$.

1. Comenzamos por visualizar la gráfica de la solución, para obtener, por inspección, una primera aproximación:

```
f = @(t,y) 0.5*(10*t-log(y+1));
ode45(f, [0,1], 1)
grid on
shg
```



Vemos así que el valor $y = 1.5$ se produce para un valor de t en el intervalo $[0.5, 0.6]$.

2. Usaremos ahora la función `fzero` para calcular la solución de la ecuación $y(t) = 1.5$ escrita en forma homogénea, es decir, $y(t) - 1.5 = 0$, dando como aproximación inicial (por ejemplo) $t = 0.5$. Para ello, necesitamos construir una función que interpole los valores que nos devuelva `ode45`. Usamos interpolación lineal a trozos:

```
[ts,ys] = ode45(f, [0,1], 1);
fun = @(t) interp1(ts, ys, t) - 1.5;
fzero(fun, 0.5)
```

Obtendremos el valor $t \approx 0.5292$.

10.3 Resolución de problemas de Cauchy para sistemas diferenciales ordinarios de primer orden

Nos planteamos ahora la resolución de sistemas diferenciales ordinarios:

$$\begin{cases} y'_1 = f_1(t, y_1, y_2, \dots, y_n), \\ y'_2 = f_2(t, y_1, y_2, \dots, y_n), \\ \dots \\ y'_n = f_n(t, y_1, y_2, \dots, y_n), \end{cases} \quad (10.3)$$

y, concretamente, de Problemas de Cauchy asociados a ellos, es decir, problemas consistentes en calcular la solución de (10.3) que verifica las condiciones iniciales:

$$\begin{cases} y_1(t_0) = y_1^0, \\ y_2(t_0) = y_2^0, \\ \dots \\ y_n(t_0) = y_n^0. \end{cases} \quad (10.4)$$

Gracias a las características vectoriales del lenguaje de MATLAB, estos problemas se resuelven con las mismas funciones (`ode45`, `ode23`, etc.) que los problemas escalares. Basta escribir el problema en forma vectorial. Para ello, denotamos:

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}, \quad F(t, Y) = \begin{pmatrix} f_1(t, y_1, y_2, \dots, y_n) \\ f_2(t, y_1, y_2, \dots, y_n) \\ \dots \\ f_n(t, y_1, y_2, \dots, y_n) \end{pmatrix}, \quad Y_0 = \begin{pmatrix} y_1^0 \\ y_2^0 \\ \dots \\ y_n^0 \end{pmatrix}, \quad (10.5)$$

Con esta notación el problema planteado se escribe:

$$\begin{cases} Y' = F(t, Y) \\ Y(t_0) = Y_0 \end{cases} \quad (10.6)$$

y se puede resolver (numéricamente) con la función `ode45`, de forma similar al caso escalar, con las adaptaciones oportunas, como en el ejemplo siguiente.

Ejercicio 10.5 Calcular (una aproximación de) la solución del problema

$$\begin{cases} y'_1 = y_2 y_3, \\ y'_2 = -0.7 y_1 y_3, & t \in [0, 5\pi] \\ y'_3 = -0.51 y_1 y_2, \\ y_1(0) = 0 \\ y_2(0) = 1 \\ y_3(0) = 1 \end{cases}$$

1. Comenzamos por escribir el sistema en notación vectorial:

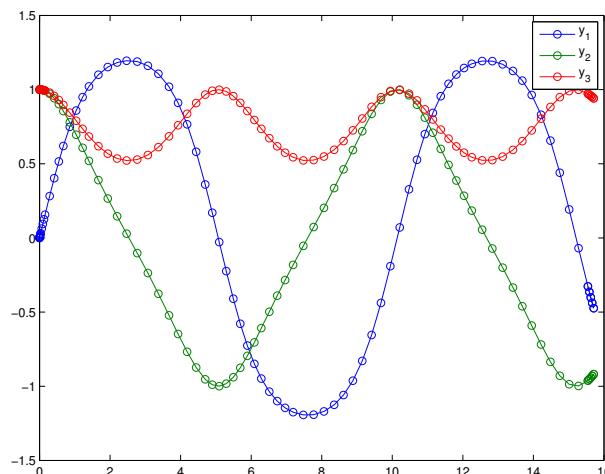
$$Y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \quad F(t, Y) = \begin{pmatrix} y_2 y_3 \\ -0.7 y_1 y_3 \\ -0.51 y_1 y_2 \end{pmatrix}, \quad Y_0 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \quad (10.7)$$

2. Definimos una función anónima para $F(t, Y)$ y el dato inicial (ambos son vectores columna con tres componentes):

```
f = @(t,y) [ y(2)*y(3); -0.7*y(1)*y(3); -0.51*y(1)*y(2)];  
y0 = [ 0; 1; 1];
```

3. Utilizamos `ode45` para calcular y dibujar la solución. Añadimos una leyenda para identificar correctamente la curva correspondiente a cada componente: $y_1(t)$, $y_2(t)$, $y_3(t)$.

```
ode45(f, [0, 5*pi], y0)  
legend('y_1', 'y_2', 'y_3')
```



4. Vamos ahora a utilizar `ode45` recuperando las variables de salida de la aproximación numérica:

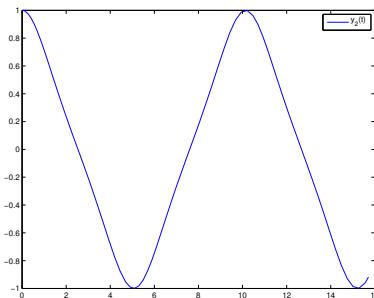
```
[t, y] = ode45(f, [0, 5*pi], y0);
```

Obsérvese que `t` es un vector columna y que `y` es una matriz con tantas filas como `t` y tres columnas: cada columna es una de las componentes de la solución Y . Es decir:

$$y(k, i) \approx y_i(t_k)$$

5. Si, por ejemplo, sólo quisiéramos dibujar la gráfica de la segunda componente $y_2(t)$ usaríamos la orden `plot` con la segunda columna de `y`:

```
plot(t, y(:,2))
legend('y_2(t)')
shg
```



En el análisis de las soluciones de sistemas diferenciales autónomos (que no dependen explícitamente del tiempo) con dos ecuaciones, interesa con frecuencia dibujar las soluciones en el *plano de fases*, es decir, en el plano (y_1, y_2) , como en el ejemplo siguiente.

Ejercicio 10.6 Calcular (una aproximación de) la solución del sistema siguiente para $t \in [0, 15]$ y dibujar la solución en el plano de fases:

$$\begin{cases} y'_1 = 0.5y_1 - 0.2y_1y_2 \\ y'_2 = -0.5y_2 + 0.1y_1y_2 \\ y_1(0) = 4 \\ y_2(0) = 4 \end{cases}$$

1. Comenzamos por escribir el sistema en notación vectorial:

$$Y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad F(t, Y) = \begin{pmatrix} 0.5y_1 - 0.2y_1y_2 \\ -0.5y_2 + 0.1y_1y_2 \end{pmatrix}, \quad Y_0 = \begin{pmatrix} 4 \\ 4 \end{pmatrix},$$

2. Para ver cómo se haría, vamos a escribir la función del segundo miembro como una M-función. También se podría hacer como una función anónima, como en el ejercicio anterior.

```
function [dy] = odefun(t,y)
dy = zeros(2,1);
dy(1) = 0.5*y(1) - 0.2*y(1)*y(2);
dy(2) = -0.5*y(2) + 0.1*y(1)*y(2);
```

Observaciones:

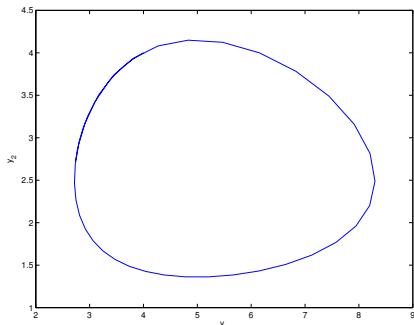
- a) Esta función debe ser guardada en un fichero de nombre `odefun.m`, que debe estar en la carpeta de trabajo de MATLAB para poder ser encontrada.
- b) El nombre `odefun` es sólo un ejemplo.
- c) La orden `dy = zeros(2,1);` en la función `odefun` tiene el objetivo de crear la variable `dy` desde un principio con las dimensiones adecuadas: un vector columna con dos componentes.

3. Calculamos la solución:

```
[t, y] = ode45(@odefun, [0, 15], [4; 4]);
```

4. Dibujamos la órbita de la solución en el plano de fases:

```
plot(y(:,1), y(:,2))
xlabel('y_1')
ylabel('y_2')
```



10.4 Resolución de problemas de Cauchy para ecuaciones diferenciales ordinarias de orden superior

En esta sección nos planteamos la resolución numérica de problemas de Cauchy para ecuaciones diferenciales ordinarias de orden superior a uno, es decir, problemas como:

$$\begin{cases} y^{(n)} = f(t, y, y', \dots, y^{(n-1)}) \\ y(t_0) = y_0 \\ y'(t_0) = y_1 \\ \dots \\ y^{(n-1)}(t_0) = y_{n-1} \end{cases} \quad (10.8)$$

La resolución de este problema se puede reducir a la resolución de un problema de Cauchy para un sistema diferencial de primer orden mediante el cambio de variables:

$$z_1(t) = y(t), \quad z_2(t) = y'(t), \quad \dots, \quad z_n(t) = y^{(n-1)}.$$

En efecto, se tiene:

$$\begin{cases} z'_1 = y' = z_2 \\ z'_2 = y'' = z_3 \\ \dots \\ z'_n = y^{(n)} = f(t, y, y', \dots, y^{(n-1)}) = f(t, z_1, z_2, \dots, z_{n-1}) \end{cases} \quad \begin{cases} z_1(t_0) = y(t_0) = y_0 \\ z_2(t_0) = y'(t_0) = y_1 \\ \dots \\ z_n(t_0) = y^{(n-1)}(t_0) = y_{n-1} \end{cases}$$

Para escribir este sistema en notación vectorial, denotamos:

$$Z = \begin{pmatrix} z_1 \\ z_2 \\ \dots \\ z_n \end{pmatrix}, \quad F(t, Z) = \begin{pmatrix} z_2 \\ z_3 \\ \dots \\ f_n(t, z_1, z_2, \dots, z_{n-1}) \end{pmatrix}, \quad Z_0 = \begin{pmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \end{pmatrix},$$

Con esta notación el problema planteado se escribe:

$$\begin{cases} Z' = F(t, Z) \\ Z(t_0) = Z_0 \end{cases} \quad (10.9)$$

y puede ser resuelto usando **ode45** como se ha mostrado en la sección anterior.

Ejercicio 10.7 Calcular y dibujar la solución del problema

$$\begin{cases} y'' + 7 \operatorname{sen} y + 0.1 \cos t = 0, & t \in [0, 5] \\ y(0) = 0 \\ y'(0) = 1 \end{cases}$$

Siguiendo los pasos antes indicados, este problema se puede reducir al siguiente:

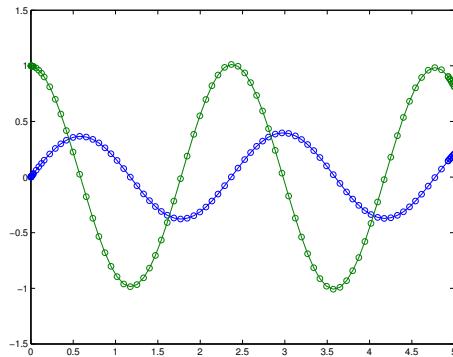
$$\begin{cases} Z' = F(t, Z) \\ Z(0) = Z_0 \end{cases}$$

con

$$F(t, Z) = \begin{pmatrix} z_2 \\ -7 \operatorname{sen} z_1 - 0.1 \cos t \end{pmatrix}, \quad Z_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

1. Comenzamos por usar **ode45** para dibujar la solución del sistema (2 componentes)

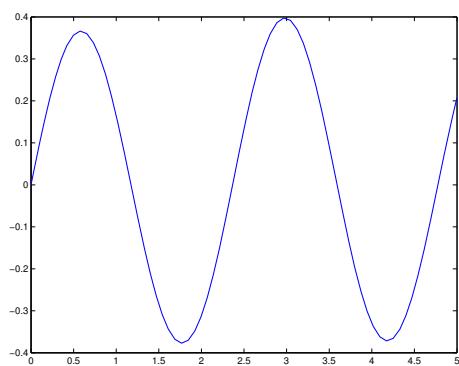
```
f = @(t,z) [z(2); -7*sin(z(1))-0.1*cos(t)];
ode45(f, [0,5], [0;1])
```



En esta gráfica están representadas las dos componentes de la solución $Z(t) = \begin{pmatrix} z_1(t) \\ z_2(t) \end{pmatrix}$

2. A nosotros sólo nos interesa la primera de las componentes de $Z(t)$, ya que es $y(t) = z_1(t)$. Por lo tanto usaremos **ode45** para recuperar los valores de la solución y dibujaremos únicamente la primera componente:

```
[t, z] = ode45(f, [0,5], [0;1]);
y = z(:, 1);
plot(t, y) % o bien, directamente, plot(t, z(:, 1))
```



11

Resolución de problemas de contorno para sistemas diferenciales ordinarios



Exponemos aquí la forma de utilizar las herramientas de MATLAB para resolver **problemas de contorno** relativos a sistemas diferenciales ordinarios de primer orden. En estos problemas se busca encontrar una solución del sistema, definida en un intervalo $[a, b]$, que verifique unas condiciones adicionales que involucran los valores de la solución en ambos extremos del intervalo. Concretamente, queremos calcular una solución del problema

$$\begin{cases} y' = f(x, y), & x \in [a, b] \\ g(y(a), y(b)) = 0. \end{cases} \quad (11.1)$$

Como es conocido, el problema (11.1) puede no tener solución, tener un número finito de soluciones o incluso tener infinitas soluciones.

Las mismas herramientas que permiten calcular (cuando la haya) una solución de (11.1) permiten también resolver problemas de contorno relativos a ecuaciones diferenciales ordinarias de orden superior, ya que, mediante un cambio de variable adecuado, estos problemas se pueden reducir al problema (11.1).

11.1 La función `bvp4c`

La función `bvp4c` utiliza, para resolver (11.1), un método de colocación: Se busca una solución aproximada $S(x)$ que es continua (de hecho, es de clase C^1) y cuya restricción a cada subintervalo $[x_i, x_{i+1}]$ de la malla determinada por la partición $a = x_0 < x_1 < \dots < x_N = b$ es un polinomio cúbico.

A esta función $S(x)$ se le impone que verifique las condiciones de contorno

$$g(S(a), S(b)) = 0,$$

y, también, la ecuación diferencial en los extremos y el punto medio de cada subintervalo $[x_i, x_{i+1}]$. Esto da lugar a un sistema (no lineal) de ecuaciones que es resuelto por métodos iterados.

Por esta razón, y también porque el problema (11.1) puede tener más de una solución, es necesario proporcionar ciertas aproximaciones iniciales.

El uso básico de la función `bvp4c` es el siguiente:

```
sol = bvp4c(odefun, ccfun, initsol)
```

donde:

odefun es un manejador de la función que evalúa el segundo miembro de la ecuación, $f(x, y)$.

Como en otros casos, puede ser el nombre de una función anónima o un manejador de una M-función.

ccfun es un manejador de la función que evalúa las condiciones de contorno $g(y_a, y_b)$

initsol es una estructura que contiene las aproximaciones iniciales. Esta estructura se crea mediante la función **bvpinit**, cuya descripción está más adelante.

sol es una **estructura**¹ que contiene la aproximación numérica calculada, así como algunos datos informativos sobre la resolución. Los campos más importantes de la estructura **sol** son:

sol.x contiene la malla final del intervalo $[a, b]$.

sol.y contiene la aproximación numérica de la solución $y(x)$ en los puntos **sol.x**.

sol.yp contiene la aproximación numérica de la derivada de la solución $y'(x)$ en los puntos **sol.x**.

sol.stats contiene información sobre las iteraciones realizadas por **bvp4c**.

```
initsol = bvpinit(xinit, yinit)
```

xinit es un vector que describe una malla inicial, i.e., una partición inicial $a = x_0 < x_1 < \dots < x_N = b$ del intervalo $[a, b]$. Normalmente bastará con una partición con pocos puntos creada, por ejemplo, con **xinit = linspace(a,b,5)**. La función **bvp4c** refinará esta malla inicial, en caso necesario. Sin embargo, en casos difíciles, puede ser necesario proporcionar una malla que sea más fina cerca de los puntos en los que se prevea que la solución cambia rápidamente.

yinit es una aproximación inicial de la solución. Puede ser, o bien un vector, o bien una función. Se usará la primera opción cuando se desee proporcionar una aproximación inicial de la solución que sea constante. Si se desea proporcionar, como aproximación inicial, una función $y = y_0(x)$, se suministrará el nombre de una función que evalúe $y_0(x)$.

Ejercicio 11.1 Calcular una aproximación numérica de la solución de:

$$\begin{cases} y'_1 = 3y_1 - 2y_2 \\ y'_2 = -y_1 + \frac{1}{2}y_2 \\ y_1(0) = 0, \quad y_2(1) = \pi \end{cases}$$

¹En MATLAB, una **estructura** es una *colección* de datos, organizados en *campos*, que se identifican mediante un *nombre de campo*. Ello permite manipular de forma compacta conjuntos de datos de distintos tipos, a diferencia de las matrices, en que todas las componentes deben ser del mismo tipo.

1. Comenzamos por definir una función anónima que represente el segundo miembro de la ecuación $f(t, y)$:

```
dydx = @(x,y) [ 3*y(1) - 2*y(2); -y(1) + 0.5*y(2)];
```

2. Definimos también una función anónima que evalúe la función que define las condiciones de contorno

```
condcon = @(ya, yb) [ ya(1); yb(2)-pi];
```

3. Construimos a continuación la estructura para proporcionar la aproximación inicial:

```
xinit = linspace(0,1,5);
yinit = [0; pi];
solinit = bvpinit(xinit, yinit);
```

4. Utilizamos ahora la función `bvp4c` para calcular la solución

```
sol = bvp4c(dydx, condcon, solinit);
```

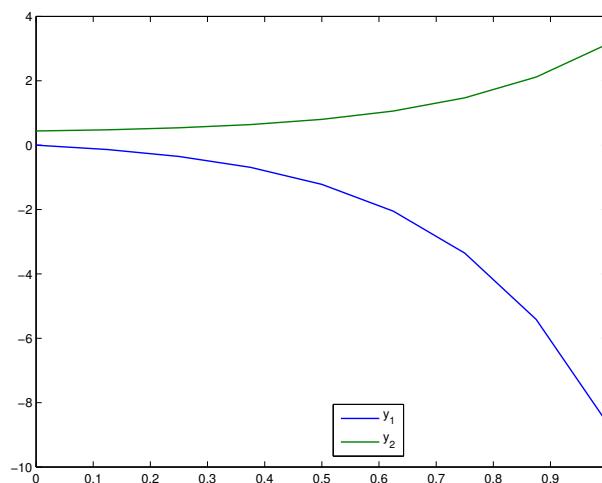
Ahora, para dibujar las curvas de la solución $(y_1(x), y_2(x))$, utilizamos la estructura `sol`:

— `sol.x` es un vector fila que contiene los puntos $a = x_0 < x_1 < \dots < x_N = b$ del soporte de la solución numérica

— `sol.y` es una matriz con dos filas; la primera fila `sol.y(1, :)` son los valores de $y_1(x)$ en los puntos `sol.x`; la segunda fila `sol.y(2, :)` son los valores de $y_2(x)$ en los puntos `sol.x`.

En consecuencia, para obtener la gráfica de la solución ponemos:

```
plot(sol.x, sol.y)
legend('y_1(x)', 'y_2(x)', 'Location', 'Best')
```



Observación: la forma en que se ha construido la gráfica anterior sólo utiliza los valores de la solución numérica en los puntos del soporte `sol.x`. **No hace uso del hecho de que la aproximación obtenida es un polinomio cúbico a trozos.** Para tener esto en cuenta, habría que usar la función `deval`, que se explica a continuación.

Para evaluar la solución obtenida con `bvp4c` en un punto o vector de puntos `xp`, se debe utilizar la función `deval`:

```
yp = deval(sol, xp);
```

donde `sol` es la estructura devuelta por `bvp4c`.

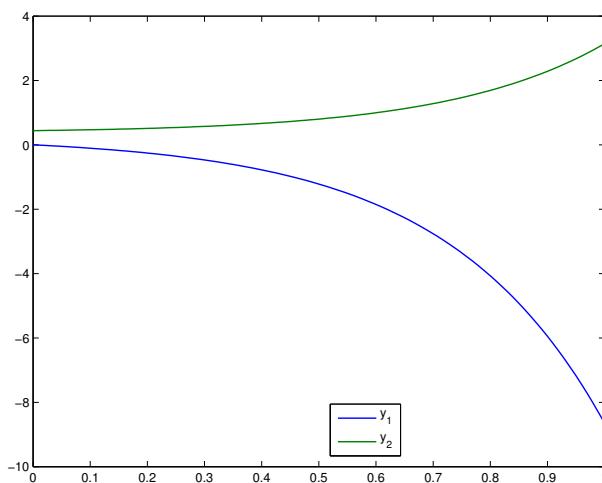
Ejercicio 11.1 (sigue...) Repetimos el ejercicio, utilizando una partición más fina del intervalo $[0, 1]$ para realizar la gráfica, de forma que las curvas se vean más «suaves». Utilizamos la función `deval` para calcular los valores de la solución aproximada en los puntos de la partición.

```
function ejercicio1
%
dydx = @(x,y) [ 3*y(1) - 2*y(2); -y(1) + 0.5*y(2)];
condcon = @(ya, yb) [ ya(1); yb(2)-pi];
xinit = linspace(0,1,5);
yinit = [0; pi];
solinit = bvpinit(xinit, yinit);

sol = bvp4c(dydx, condcon, solinit);

xx = linspace(0,1);
yy = deval(sol, xx);
plot(xx,yy)
legend('y_1', 'y_2', 'Location', 'Best')

shg
```



Ejercicio 11.2 Calcular la solución del problema

$$\begin{cases} -y'' + 100y = 0, & x \in [0, 1] \\ y(0) = 1, & y(1) = 2 \end{cases} \quad (11.2)$$

Comenzamos por escribir el problema en forma de un problema de contorno para un SDO de primer orden. Hacemos el cambio de variable $z_1 = y$, $z_2 = y'$ y el problema (11.2) se transforma en:

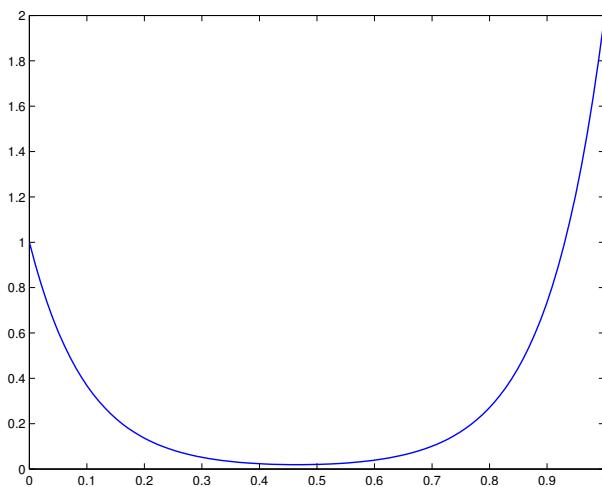
$$\begin{cases} z'_1 = z_2 \\ z'_2 = 100z_1 \\ z_1(0) = 1, & z_1(1) = 2 \end{cases} \quad (11.3)$$

```
function ejercicio2
%
dydx = @(t,y) [ y(2); 100*y(1)];
condcon = @(ya, yb) [ ya(1)-1; yb(1)-2];
xinit = linspace(0,1,5);
yinit = [1.5; 0];
solinit = bvpinit(xinit, yinit);

sol = bvp4c(dydx, condcon, solinit);

xx = linspace(0,1);
yy = deval(sol, xx);
plot(xx, yy(1,:), 'LineWidth', 1.1)

shg
```



La función `bvp4c` permite, mediante un argumento opcional, modificar algunas de los parámetros que utiliza el algoritmo por defecto. Para ello hay que usar la función en la forma

```
sol = bvp4c(odefun, ccfun, initval, options)
```

donde

options es una estructura que contiene los valores que se quieren dar a los parámetros opcionales. Esta estructura se crea mediante la función **bvpset** que se explica a continuación.

```
options = bvpset('nombre1', valor1, 'nombre2', valor2, ...)
```

nombre1, nombre2 son los nombres de las propiedades cuyos valores por defecto se quieren modificar. Las propiedades no mencionadas conservan su valor por defecto.

valor1, valor2 son los valores que se quieren dar a las propiedades anteriores.

Para una lista de las propiedades disponibles, véase el **help** de MATLAB. Los valores actuales se pueden obtener usando la función **bvpset** sin argumentos. Mencionamos aquí la propiedad que vamos a usar en el ejercicio siguiente, que nos permite determinar el nivel de precisión que **bvp4c** utilizará para detener las iteraciones y dar por buena una aproximación:

RelTol es el nivel de tolerancia de error relativo que se aplica a todas las componentes del vector residuo $r_i = S'(x_i) - f(x_i, S(x_i))$. Esencialmente, la aproximación se dará por buena cuando

$$\left\| \frac{S'(x_i) - f(x_i, S(x_i))}{\max\{|f(x_i, S(x_i))|\}} \right\| \leq \text{RelTol}$$

El valor por defecto de **RelTol** es 1.e-3.

Stats si tiene el valor **'on'** se muestran, tras resolver el problema, estadísticas e información sobre la resolución. El valor por defecto es **'off'**.

Ejercicio 11.3 La ecuación diferencial

$$y'' + \frac{3py}{(p+t^2)^2} = 0$$

tiene la solución analítica

$$\varphi(t) = \frac{t}{\sqrt{p+t^2}}.$$

Para $p = 10^{-5}$, resolver el problema de contorno

$$\begin{cases} y'' + \frac{3py}{(p+t^2)^2} = 0 \\ y(-0.1) = \varphi(-0.1), \quad y(0.1) = \varphi(0.1) \end{cases}$$

para el valor por defecto de **RelTol** y para **RelTol=1.e-5** y comparar con la solución exacta.

Comenzamos por escribir el problema en forma de un problema de contorno para un SDO de primer orden. Hacemos el cambio de variable $z_1 = y$, $z_2 = y'$ y el problema (11.2) se transforma en:

$$\begin{cases} z'_1 = z_2 \\ z'_2 = -\frac{3pz_1}{(p+t^2)^2} \\ z_1(-0.1) = \frac{-0.1}{\sqrt{p+0.01}}, \quad z_1(0.1) = \frac{0.1}{\sqrt{p+0.01}} \end{cases} \quad (11.4)$$

```
function ejercicio3

% Parametros y constantes
%
p = 1.e-5;
yenb = 0.1/sqrt(p+0.01);

% Funciones anonimas
%
dydt = @(t,y) [ y(2); -3*p*y(1)/(p+t^2)^2];
condcon = @(ya, yb) [ya(1)+yenb; yb(1)-yenb ];
exacta = @(t) t./sqrt(p+t.^2);

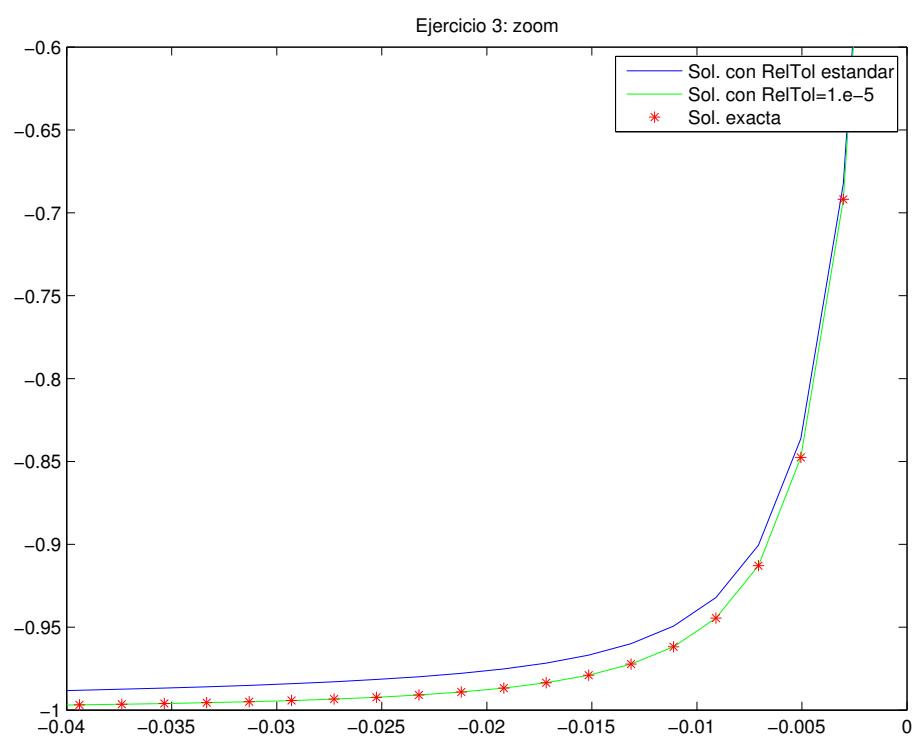
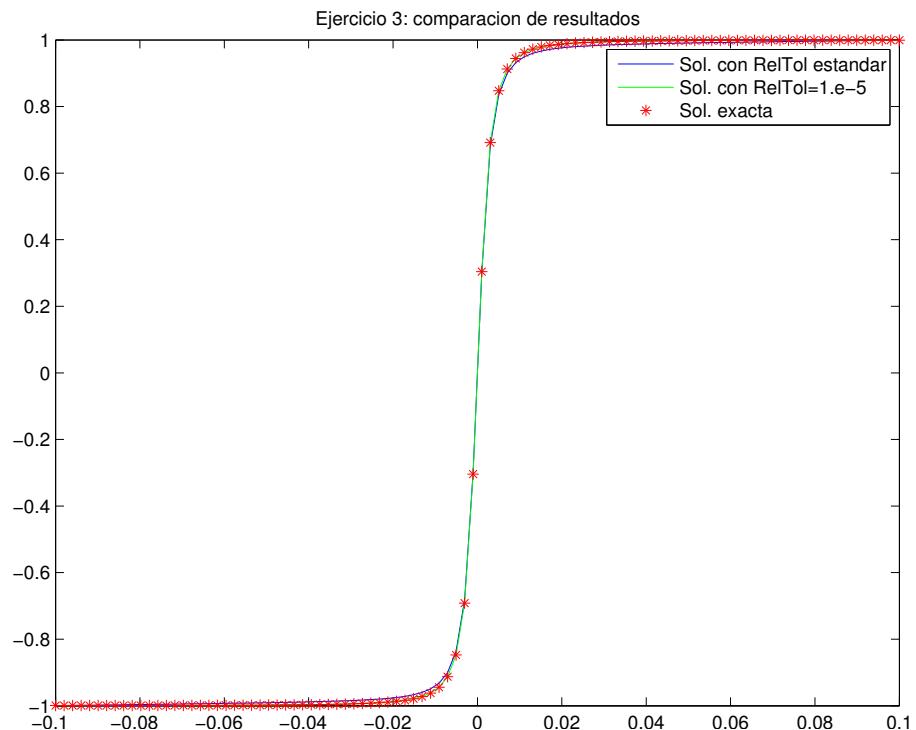
% Aproximaciones iniciales
% La aproximacion inicial se ha obtenido promediando
% las condiciones de contorno
%
tguess = linspace(-0.1, 0.1, 10);
yguess = [0; 10];
solinit = bvpinit(tguess, yguess);

% Resolucion con valor estandar de RelTol
%
options = bvpset('Stats','on');
fprintf('\n')
fprintf('==> Resolucion con valor estandar de RelTol \n')
sol1 = bvp4c(dydt, condcon, solinit,options);

% Resolucion con RelTol = 1.e-5
%
options = bvpset(options, 'RelTol', 1.e-5);
fprintf('\n')
fprintf('==> Resolucion con RelTol =%15.10f\n',bvpget(options,'RelTol'))
sol2 = bvp4c(dydt, condcon, solinit, options);

% Graficas
% Dibujamos solo la primera componente de la solucion del sistema
```

```
%  
xx = linspace(-0.1, 0.1);  
yy1 = deval(sol1, xx, 1);  
yy2 = deval(sol2, xx, 1);  
yye = exacta(xx);  
plot(xx, yy1,'b', xx, yy2,'g', xx, yye,'r*')  
shg
```



12 Resolución de problemas de minimización



Exponemos aquí brevemente la forma de utilizar las herramientas de MATLAB para resolver **problemas de minimización con restricciones**.

Concretamente, queremos calcular una solución del problema

$$\begin{cases} \text{Minimizar } J(x) \\ \text{sujeto a } x \in \mathbb{R}^n \\ G_i(x) = 0 \quad i = 1, \dots, m_e \\ G_i(x) \leq 0 \quad i = m_e + 1, \dots, m \end{cases} \quad (12.1)$$

donde J es una función con valores reales.

12.1 La función fmincon

La función **fmincon**, del **Optimization Toolbox** de MATLAB, se puede utilizar para resolver el problema (12.1) en el caso en que tanto la función a minimizar como las restricciones (o algunas de ellas) son no lineales.

Para resolver el problema (12.1) con **fmincon** lo primero que hay que hacer es clasificar el conjunto de las restricciones ($G_i(x) = 0, i = 1, \dots, m_e, G_i(x) \leq 0, i = m_e + 1, \dots, m$) según los tipos siguientes:

- Cotas: $c_l \leq x \leq c_u$
- Igualdades lineales: $Ax = b$, donde A es una matriz
- Desigualdades lineales: $Cx \leq d$, donde C es una matriz
- Igualdades nolineales: $f(x) = 0$
- Desigualdades no lineales: $g(x) \leq 0$

Cada tipo de restricción habrá que incorporarla de manera distinta a la función **fmincon**.

Los usos más sencillos de la función **fmincon** son los siguientes:

```
x = fmincon(funJ, x0, C, d)
x = fmincon(funJ, x0, C, d, A, b)
x = fmincon(funJ, x0, C, d, A, b, ci, cs)
x = fmincon(funJ, x0, C, d, A, b, ci, cs, resnolin)
x = fmincon(funJ, x0, C, d, A, b, ci, cs, resnolin, options)
```

donde:

funJ es un manejador de la función a minimizar $J(x)$. Como en otros casos, puede ser el nombre de una función anónima o un manejador de una M-función.

x0 es una aproximación inicial de la solución.

C, d son, respectivamente, la matriz y el segundo miembro que definen las restricciones lineales de desigualdad $Cx \leq d$.

A, b son, respectivamente, la matriz y el segundo miembro que definen las restricciones lineales de igualdad $Ax = b$. Si hay que utilizar **A** y **b** y nuestro problema no tiene restricciones con desigualdades lineales, se pone **C=[]** y **b=[]**.

ci, cs son, respectivamente, las cotas inferior y superior $c_i \leq x \leq c_s$. Si no hay cota inferior para alguna de las variables se especificará **-Inf**. Análogamente, si no hay cota superior, se pondrá **Inf**. Como antes, si alguno de los argumentos anteriores (**C, d, A, b**) no se usa, se utilizarán matrices vacías (**[]**) en su lugar.

resnolin es un manejador de la función que define las restricciones no lineales, tanto de igualdad $f(x) = 0$ como de desigualdad $g(x) \leq 0$. Puede ser el nombre de una función anónima o un manejador de una M-función. Esta función debe aceptar como argumento de entrada un vector **x** y devolver dos vectores: **c = g(x)**, **ceq = f(x)**:

```
function [c, ceq] = resnolin(x)
```

options es una **estructura** que permite modificar algunos de los parámetros internos de la función **fmincon**. Esta estructura se crea mediante la función:

```
options = optimset('Opcion1','valor1','Opcion2','valor2', ...)
```

Aunque son numerosas las opciones que se pueden modificar, mencionamos aquí sólo dos:

Algorithm permite elegir entre cuatro algoritmos, en función de las características y/o dificultad del problema a resolver. Los posibles valores son: '**interior-point**', '**sqp**', '**active-set**' y '**trust-region-reflective**'.

Display permite elegir el nivel de *verbosity*, es decir, de información suministrada durante la ejecución. Posibles valores son: '**off**', '**iter**', '**final**' y '**notify**'.

Ejercicio 12.1 Calcular una aproximación de la solución de:

$$\begin{cases} \text{Minimizar } f(x) = -x_1 x_2 x_3 \\ \text{sujeto a} \\ 0 \leq x_1 + 2x_2 + 3x_3 \leq 72 \end{cases}$$

partiendo de la aproximación inicial $x_0 = (10, 10, 10)$.

Re-escribimos las restricciones.

$$\begin{cases} -x_1 - 2x_2 - 3x_3 \leq 0 \\ x_1 + 2x_2 + 3x_3 \leq 72 \end{cases}$$

Se trata en ambos casos de restricciones lineales de desigualdad, que se pueden expresar en la forma

$$Cx \leq d \quad \text{con } C = \begin{pmatrix} -1 & -2 & -3 \\ 1 & 2 & 3 \end{pmatrix}, \quad d = \begin{pmatrix} 0 \\ 72 \end{pmatrix}$$

```
funJ = @(x) - prod(x);

C = [-1, -2, -3; 1, 2, 3];
d = [0; 72];
ci = [-Inf; -Inf; -Inf];
cs = [ Inf; Inf; Inf];
x0 = [10; 10; 10];
options = optimset('Algorithm', 'interior-point', 'Display', 'iter');

x = fmincon(funJ, x0, C, d, [], [], ci, cs, options)
```

Ejercicio 12.2 Calcular una aproximación de la solución de:

$$\begin{cases} \text{Minimizar } f(x) = -x_1 x_2 x_3 \\ \text{sujeto a} \\ 0 \leq x_1 + 2x_2 + 3x_3 \leq 72 \\ x_1^2 + x_2^2 + x_3^2 \leq 500 \end{cases}$$

partiendo de la aproximación inicial $x_0 = (10, 10, 10)$.

Las restricciones lineales son las mismas del ejercicio anterior. La restricción no lineal hay que escribirla en forma homogénea

$$x_1^2 + x_2^2 + x_3^2 - 500 \leq 0$$

y escribir una M-función para definir dichas restricciones:

```
function [c, ceq] = resnolin(x)
%
% funcion definiendo las restricciones no lineales
%
c = sum(x.^2)-500;
ceq = 0;
```

Para resolver el problema,

```
funJ = @(x) - prod(x);
C = [-1, -2, -3; 1, 2, 3];
d = [0; 72];
ci = [-Inf; -Inf; -Inf];
cs = [ Inf; Inf; Inf];
x0 = [10; 10; 10];
options = optimset('Algorithm', 'interior-point', 'Display', 'iter');
x = fmincon(funJ, x0, C, d, [], [], ci, cs, @resnolin, options)
```