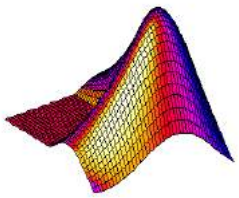
Introducción a MATLAB



Julio Benítez López, José Luis Hueso Pagoaga

Departamento de Matemática Aplicada

Universidad Politécnica de Valencia

Índice.

0. Información General.

Este texto no tretende ser más que unos breves apuntes para presentar el programa MATLAB y su lenguaje de programación. Para una información más completa puede utilizarse los manuales [1] o [2]. Estas notas están elaboradas, en su mayoría, a partir de [3]. La versión de MATLAB usada es la 5.3 para Windows 95.

MATLAB (MATrix LABoratory) es un programa orientado al cálculo con matrices, al que se reducen muchos de los algoritmos que resuelven problemas de Matemática Aplicada e Ingeniería.

MATLAB ofrece un entorno interactivo sencillo mediante una ventana en la que podemos introducir ordenes en modo texto y en la que aparecen los resultados. Los gráficos se muestran en ventanas independientes. Cada ventana dispone de una barra de menús que controla su funcionalidad.

Aprenderemos a asignar, borrar, guardar y recuperar variables, utilizar las funciones incorporadas y, más adelante, a definir funciones nuevas. MATLAB opera directamente con números complejos y con números reales como caso particular.

Lo que distingue a MATLAB de otros sistemas de cálculo es su facilidad para trabajar con vectores y matrices. Las operaciones ordinarias, suma, producto, potencia, operan por defecto sobre matrices, sin más restricción que la compatibilidad de tamaños en cada caso.

Entes matemáticos como los polinomios son tratados por MATLAB como vectores, gracias al hecho de que se suman y multiplican por escalares de la misma forma que éstos. La multiplicación y división entera de polinomios se efectan mediante órdenes específicas, así como la evaluación o derivación de un polinomio.

Una de las características más destacables de MATLAB es su capacidad gráfica. Explicaremos algunos comandos gráficos para representación de funciones de una o dos variables en distintos sistemas de coordenadas.

MATLAB dispone de mandatos propios de un lenguaje de programación para efectuar bucles y bifurcaciones condicionales y puede ejecutar las órdenes contenidas en ficheros grabados en ASCII mediante un editor como el bloc de notas o el editor de ficheros de órdenes incorporado.

Al escribir, por ejemplo, la suma de dos matrices como `A+B` obtenemos un código más claro y conciso que en otro lenguaje de programación, sin bucles innecesarios y de ejecución mucho más rápida.

Desde ahora y en adelante, destacaremos en `Courier` las órdenes que pueden introducirse en MATLAB tal y como aparecen o incorporarse a un fichero de órdenes. En *`Courier cursiva`* indicamos parámetros de algunas órdenes, que deben sustituirse por un dato apropiado en cada caso. Se debe interpretar el inductor `>>` como una invitación a escribir en MATLAB lo que aparece a continuación del mismo.

1. Perdiendo el miedo.

1.1. Mandatos básicos.

Al abrir el programa, MATLAB sugiere comenzar con la `demo` o con órdenes de ayuda.

`help` nos da una lista de temas sobre los que hay información de ayuda. `helpwin` abre una ventana de ayuda que es útil para consultar información sobre rdenes de MATLAB sin interferir con la ventana principal. `help` *`tema`* explica sucintamente el tema elegido. Así, por ejemplo,

```
>> help demo
```

explica brevemente el comando `demo` o

```
>> help cos
```

explica la función `cos` (el coseno de un ángulo). A propósito, MATLAB distingue entre mayúsculas o minúsculas. Por ejemplo:

```
>> Help demo
```

da un mensaje de error pues `Help` no es una orden de MATLAB y `help` sí. Para evitar que MATLAB diferencie entre mayúsculas y minúsculas teclearemos

```
>> casesen off
```

Obviamente, el comando contrario será `casesen on`.

Otro comando útil es `lookfor` *`tema de búsqueda`*. Así, por ejemplo, si queremos averiguar los comandos relacionados con la función seno, teclearemos

```
>> lookfor sine
```

El mandato `dir` lista los ficheros del directorio actual. Pruébese `dir a:`

Uno de los mandatos más útiles de MATLAB es `diary`, que permite guardar en un fichero todo el texto que aparece en la ventana de comandos. Si se tiene un diskette en la unidad a:, se puede escribir

```
>> diary a:taller.txt
```

y todo lo que salga en pantalla se grabará en un fichero **taller.txt** justo cuando se vuelva a introducir el mandato `diary`. Para añadir más texto en una misma sesión al diario

creado se usa `diary on` al principio de lo que se quiera grabar y `diary off` al final (en este momento se graba realmente).

El mandato `%` convierte en comentario lo que se escriba a continuación. Es decir, MATLAB ignora lo que viene a continuación del comando `%`.

```
>> % Esto es un comentario
```

Con las teclas del cursor [↑] y [↓] se recuperan los mandatos antes escritos, evitando así tener que reescribir órdenes iguales o parecidas. También se puede "copiar" con el ratón texto de cualquier sitio y "pegar" en la (única) línea de mandatos activa, eligiendo estas opciones en el menú de edición. Vale usar [Ctrl]+C y [Ctrl]+V con el mismo fin.

Otro comando útil a la hora de manejar funciones definidas por el propio usuario es `path`. Este comando controla los directorios donde MATLAB "puede leer" las funciones que maneja. Inicialmente, MATLAB no "puede manejar" funciones escritas en un diskette insertado en la unidad a: Para que sí las pueda manejar, se escribirá

```
>> path(path,'a:')
```

`path` por sí sólo muestra los directorios en los que MATLAB busca los comandos. También se puede controlar el `path` mediante el menú: **File** y a continuación **Set Path...**

1.2. Variables.

MATLAB puede usarse como calculadora (¡aunque sea infrautilizar MATLAB !). Los signos `+, -, *, /` y `^` denotan las operaciones aritméticas de suma, resta, multiplicación, división y elevación a una potencia. Si el resultado de una operación no es asignado a ninguna variable, MATLAB lo asigna a la variable del sistema **ans**[1]. Así mismo se pueden usar los paréntesis `()` para concatenar expresiones, no así los corchetes `[]` que están reservados para introducir vectores y matrices. Así, por ejemplo:

```
>> 3*((1+3)^(1/2))
```

proporciona, obviamente, `ans = 6`.

En MATLAB las variables se asignan de modo natural. Basta escribir un nombre de variable, a continuacin el signo `=` y luego el valor que toma esa variable. Para aceptar, como siempre, hay que pulsar [Intro]. Escribiendo sólo el nombre de una variable previamente asignada, MATLAB devuelve su valor. Por ejemplo, se puede escribir

```
>> a = 3, b = 4
```

y a continuación

```
>> a+b.
```

Para conservar este resultado se hace

[1] Del inglés answer.

```
>> c = ans
```
o mejor aún
```
>> c = a+b
```
directamente.

Si se pone un "punto y coma", tras la asignación no se muestra el resultado por pantalla. Naturalmente, la asignación no resulta afectada. Esta forma de proceder resulta útil en la elaboración de resultados que requieran cálculos intermedios. Por ejemplo, la orden
```
>> d=(a+b)^2;
```
calcula el valor de d; pero no lo muestra en pantalla.

La orden `who` lista las variables definidas y con la orden `whos` obtenemos además el tipo de variable y su tamaño. Se puede acceder a través del menú e incluso modificar los valores existentes: **File** y a continuación **Show Workspace**. Para modificar las variables una vez abierto el *workspace browser*, no tenemos más que pinchar en cada variable.

`clear` sin argumentos elimina todas las variables. `clear c` elimina la variable c.

Las variables de una sesión se pueden grabar en un fichero en disco. La orden
```
>> save a:taller
```
guarda los nombres y los valores de las variables actualmente definidas en un fichero en a: llamado **taller.mat**. MATLAB añade automáticamente la extensión **mat** y así identifica el formato de estos ficheros. Si se borran las variables:
```
>> clear, who
```
se pueden recuperar ahora o en otra sesión con la orden
```
>> load a:taller
```

De nuevo mediante el menú **File** también se pueden ejecutar los comandos `load` y `save`.

1.3. Constantes y cadenas.

MATLAB utiliza ciertos nombres de variable para fines especiales, como `i` o `j`, que designan ambas a la unidad imaginaria o `pi`, para el número π. El número e, base de los logaritmos neperianos, no está preasignado, pero se obtiene fácilmente como `exp(1)`[2].

[2]El uso de la función `exp` (exponencial) será explicado en la sección siguiente.

La precisión relativa en operaciones de coma flotante se llama `eps`. En realidad `eps` es el mayor positivo que cumple 1+eps = 1. El resultado de 1/0 en MATLAB es `Inf` y el de 0/0, `NaN`[3].

Se pueden utilizar estos nombres de variables preasignadas para almacenar otros valores, prevaleciendo esta última asignación sobre el valor por defecto de MATLAB . Por ejemplo, si no se utilizan números complejos, no hay inconveniente en representar por `i` y `j` los índices de fila y columna de una matriz. Igualmente se podríamos llamar `eps` a una cantidad utilizada como criterio de convergencia, pero en general conviene evitar equívocos empleando otros nombres de variable.

Internamente MATLAB trabaja con mucha precisión, aunque por defecto muestra los resultados con cuatro decimales. La apariencia de los resultados se modifica por medio del menú o con la orden `format`: Por ejemplo, `format long` aumenta el número de decimales visibles.

```
>> format long, pi
```

`format short` vuelve al estado inicial. `format rat` aproxima el resultado por un cociente de enteros pequeños. Se pueden explorar otras opciones con `help format`.

Podemos usar también cadenas de caracteres para manejar texto en funciones de MATLAB . Para introducir una cadena, basta escribir el texto entre comillas.

```
>> 'Esto es una cadena'
```

Un texto sin comillas produce error porque MATLAB lo interpreta como un nombre de variable o función.

```
>> Hola
```

produce el error

```
???  Undefined function or variable Hola.
```

1.4. Funciones.

MATLAB lleva incorporadas una cantidad considerable de *funciones*. Su sintaxis es la siguiente: Para una función de una variable y de una salida es

```
variable_de_salida = nombre_de_funcion(variable_de_entrada)
```

y si $x_1, \cdots, x_n$ son n variables de entrada e $y_1, \cdots y_m$ son m variables de salida:

```
[y1, ...  , ym] = nombre_de_funcion(x1, ..., xn)
```

MATLAB reconoce las funciones matemáticas elementales:

[3]Del inglés Not a Number.

Funciones Trigonométricas

Función	Sintaxis	Ejemplo		
		Si se escribe	equivale	al resultado
seno	`sin(X)`	`sin(pi)`	$\operatorname{sen}\pi$	0
coseno	`cos(X)`	`cos(-pi)`	$\cos(-\pi)$	-1
tangente	`tan(X)`	`tan(pi/4)`	$\operatorname{tg}(\pi/4)$	1
secante	`sec(X)`	`sec(pi/3)`	$\sec(\pi/3)$	2
cosecante	`csc(X)`	`csc(pi/4)`	$\operatorname{cosec}(\pi/4)$	1.4142 ($=\sqrt{2}$)
cotangente	`cot(X)`	`cot(pi/6)`	$\operatorname{cotg}(\pi/6)$	1.7321 ($=\sqrt{3}$)

Funciones Trigonométricas Inversas

Función	Sintaxis	Ejemplo		
		Si se escribe	equivale	al resultado
arcoseno	`asin(X)`	`asin(1)`	$\operatorname{arcsen}1$	1.5708 ($=\pi/2$)
arcocoseno	`acos(X)`	`acos(1)`	$\arccos 1$	0
arcotangente	`atan(X)`	`atan(Inf)`	"$\operatorname{arctg}(\infty)$"	1.5708 ($=\pi/2$)
...	...	...	...	...

Exponenciales

Función	Sintaxis	Ejemplo		
		Si se escribe	equivale	al resultado
exponencial	`exp(X)`	`exp(1)`	$\exp 1$	2.7183 ($=e$)
logaritmo neperiano	`log(X)`	`log(exp(2))`	$\log(e^2)$	2
logaritmo decimal	`log10(X)`	`log10(1000)`	$\log_{10}(1000)$	3
logaritmo en base 2	`log2(X)`	`log2(1/4)`	$\log_2(1/4)$	-2
raíz cuadrada	`sqrt(X)`	`sqrt(9)`	$\sqrt{9}$	3

Hay muchas más funciones que se pueden consultar tecleando `help elfun`. Aparte de estar el nombre en inglés (por ejemplo, el seno es `sin` (del inglés sine)), hay que tener en cuenta unos pequeños detalles para evitar errores al usar funciones:

- Las funciones trigonométricas tienen el argumento en radianes.
- MATLAB reserva `log` para el logaritmo neperiano, coherentemente con el uso habitual en matemática superior, pero en contraste con la práctica en ingeniería en donde son más usados el logaritmo decimal o el de base 2.

Se pueden ver otras funciones disponibles con `help specfun`.

MATLAB dispone de una orden para dibujar fácilmente funciones: es el comando `ezplot`. Se escribe `ezplot` y a continuación la expresión algebraica de la función que se desea dibujar entre comillas. Por ejemplo, si se quiere dibujar la función $f(x) = \frac{x^3+1}{x}$ definida en $[-2, 2]$ se ecribirá:

```
>> ezplot('(x^3+1)/x',[-2,2])
```

Se recomienda escribir `help ezplot` y comprobar los ejemplos aquí propuestos.

La orden `fplot` proporciona más opciones para controlar el aspecto de la gráfica obtenida. Por ejemplo, permite representar varias funciones a la vez[4].

```
>> fplot('[sinh(x),cosh(x),tanh(x)]',[-2,2])
>> grid, axis equal
```

1.5. Números Complejos.

Los números complejos se introducen tal y como se escriben en matemáticas, utilizando `i` o `j` para la unidad imaginaria. Por ejemplo:

```
>> z = 3 + 4i
```

Obsérvese que no es necesario poner el signo de multiplicación, `*`, entre el `4` y la `i`.

MATLAB trabaja por defecto con complejos, incluidas las funciones `exp, sin, cos` que se evalúan sin más: `exp(i*pi)`. Por supuesto tiene implementadas las funciones básicas de los complejos:

Funciones Complejas

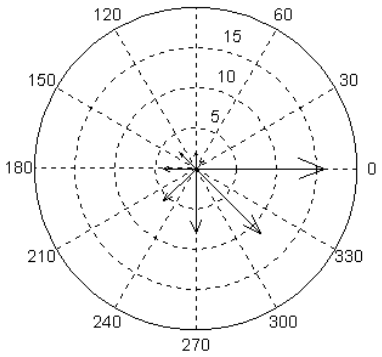
Función	Sintaxis	Ejemplo		
		Si se escribe	equivale	al resultado
Parte real	`real(X)`	`real(1+2i)`	$\mathrm{Re}(1+2i)$	1
Parte imaginaria	`imag(X)`	`imag(1+2i)`	$\mathrm{Im}(1+2i)$	2
Conjugado	`conj(X)`	`conj(1+2i)`	$\overline{1+2i}$	$1-2i$
Módulo	`abs(X)`	`abs(1+2i)`	$\lvert 1+2i\rvert$	2.2361 ($=\sqrt{5}$)
Argumento	`angle(X)`	`angle(1+2i)`	$\arg(1+2i)$	1.1071 ($=\mathrm{arctg}\,2$)

Se pueden dibujar números complejos mediante `plot` y `compass`:

```
>> plot(1+2i,'*')
>> compass(1+2i)
```

Ejercicio 1: Representa las potencias de exponente 1, 2, 3 ... del complejo $z = 1+i$. Usar la orden `hold` depués de obtener el primer gráfico para que los siguientes se dibujen en la misma ventana sin borrar los previos.

[4]El comando `grid` dibuja el "mallado" de la gráfica y el comando `axis` controla los ejes de coordenadas. Se recomienda usar el comando `help` para estas órdenes.



1.6. Vectores.

Para introducir vectores en MATLAB se escriben sus componentes entre corchetes. Separando las componentes con comas o espacios obtenemos un vector fila. Separándolas por punto y coma o por medio de la tecla [Intro], obtenemos un vector columna. Por ejemplo:

```
u = [1 2 3], v = [1,2,3] % Vectores fila
w = [1;2;3] % Vectores columna
z = [1
2
3]
```

Es muy frecuente tener que editar vectores con componentes equiespaciadas, por ejemplo, para crear una tabla de valores de una función. Con `a:h:b` se crea un vector de componentes que van de a hasta b y distan h cada una de la siguiente. La orden `linspace(a,b,n)` crea n términos en progresión aritmética, desde a hasta b.

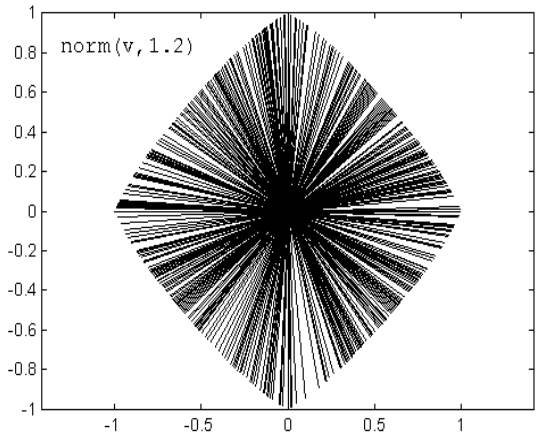
```
>> v= 0:0.1:1
>> w=linspace(0,1,11)
```

`linspace(a,b)` crea un vector con $n = 100$ elementos.

La norma (euclídea) de un vector se calcula con el comando `norm`. Si se introduce

```
>> norm([1,2,3])
```

produce `3.7417` ($=\|(1,2,3)\| = \sqrt{1^2+2^2+3^2}$). Además de esta norma, con MATLAB se pueden manejar otro tipo de normas, como la norma 1, `norm(v,1)`, o la norma infinito o supremo, `norm(v,inf)`.



La función `length` devuelve el número de componentes de un vector.

La suma (o resta) de dos vectores del mismo tamaño se efectúa con MATLAB escribiendo `u + v` (`u-v` para la resta). Para multiplicar un vector u por un escalar a, se pone

```
>> a*u.
```

En ocasiones hay que multiplicar dos vectores elemento a elemento y eso MATLAB lo hace con la versión "punto" del producto. Por ejemplo:

```
>> u=[1 2 3]; v=[4 5 6];
>> u.*v
```

produce el vector `[4 10 18]`. Sin embargo, la orden `u*v`, produce un mensaje de error, ya que MATLAB aplica el producto matricial y los tamaños no son coherentes. El "punto" antepuesto a las operaciones `*, ^, /` significa "hacer dichas operaciones coordenada a coordenada". Por ejemplo, almacenando en un vector los 10 primeros naturales

```
>> N=1:10;
```

podemos[5] calcular los 10 primeros cubos con `N.^3` o las 10 primeras potencias de 2 con `2.^N`.

Ejercicio 2: Obtener 1, 1/2, 1/3, ..., 1/10.

Ejercicio 3: Obtener 10, 9/2, 8/3, ..., 2/9, 1/10. SUGERENCIA: Cuando se introducen vectores mediante `v=a:h:b`, el paso h debe ser negativo, si $a > b$.

Una característica destacable de MATLAB es que evalúa una función sobre todas las componentes de un vector simultáneamente, lo cual es muy práctico, por ejemplo, para construir tablas de funciones o representarlas gráficamente.

[5]Si se introducen vectores mediante `:` y se escribe sólamente `v=a:b`, esto equivale a decir que la separación de las componentes de v es 1, es decir `v=a:1:b`.

```
>> x = -2:0.1:2
>> y = exp(x)
>> plot(x,y)
```

La siguiente tabla muestra algunas de las funciones para vectores más usuales. Los ejemplos son para $u = (1, 2, 3)$ y $v = (4, 5, 6)$.

Funciones Vectoriales

Función	Sintaxis	Ejemplo		
		Si se escribe	equivale	al resultado
Transpuesta	`X'`	`u'`	$(1, 2, 3)^{\mathrm{t}}$	$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$
Suma de componentes	`sum(X)`	`sum(u)`	1+2+3	6
Producto de componentes	`prod(X)`	`prod(v)`	$4 \cdot 5 \cdot 6$	120
Producto escalar	`dot(X,Y)`	`dot(u,v)`	$\langle u, v \rangle$	32
Producto vectorial	`cross(X,Y)`	`cross(u,v)`	$u \wedge v$	(-3,6,-3)
Voltear de derecha a izquierda	`fliplr(X)`	`fliplr(u)`		(3,2,1)
Voltear de arriba a abajo	`flipud(X)`	`(flipud(v'))'`		(6,5,4)

Observar que el producto vectorial sólo está definido para vectores de $\mathbb{R}^3$.

Ejercicio 4: Calcular la suma de los cuadrados de los mil primeros naturales.
Ejercicio 5: Dado un vector v, calcular la media aritmetica de sus componentes.
Ejercicio 6: Calcular la siguiente expresión: $\displaystyle\sum_{n=1}^{15} \operatorname{sen}\left(\frac{2\pi n}{15}\right)$.

1.7. Matrices.

Por defecto, MATLAB trabaja con matrices. Esto supone la ventaja substancial de no tener que declarar tipos de variable ni tamaños de fila o columnas para trabajar tanto con matrices de números reales o complejos como con vectores o escalares, que se consideran casos particulares de matrices.

Las matrices se escriben por filas. Los elementos de una fila se separan por "comas" y las distintas filas por "puntos y comas".

```
>> A = [1,2;3,4]
```

Lo mismo que con vectores, podemos también separar los elementos de una fila con espacios y las filas pulsando la tecla [Intro].

```
>> B = [-1 -2
-3 -4]
```

El elemento en la fila i y la columna j de la matriz A es `A(i,j)`. Se pueden modificar componentes de una matriz. Por ejemplo, para cambiar el elemento $(2,1)$ de A por 0 hacemos:

```
>> A(2,1) = 0
```

`A(i,:)` denota la fila i de la matriz A. Análogamente, `A(:,j)` es la columna j de A.

En ocasiones resulta cómodo construir una matriz a base de bloques. Con tal de que sus tamaños sean coherentes, basta escribir los bloques por filas, como si se tratase de elementos individuales. Por ejemplo:

```
>> M = [A,B;B,A]
```

Para extraer una submatriz, se indican las filas y columnas de que se compone.

```
>> M41 = M(1:3,2:4)
```

Las filas o columnas no tienen por que ser consecutivas:

```
>> fil = [1,2,4]; col = [1,3,4];
>> M32 = M(fil,col)
```

La función `size` devuelve un vector de dos componentes que son el número de filas y el número de columnas de la matriz:

```
>> [filas,columnas] = size(M)
```

El comando `diag` tiene dos comportamientos distintos, según si el argumento es una matriz o un vector. `diag (matriz)` produce un vector cuyas componentes forman la diagonal principal de la matriz. `diag (vector)` produce una matriz diagonal cuya diagonal principal es el vector.

```
>> v=[1 2 3]; A=diag(v)
>> B=[1 2;3 4]; w=diag(B)
```

El comando `diag` en realidad tiene dos argumentos de entrada: `diag(vector, k)` (k es un entero que denota la posición del vector, $k = 0$ es la diagonal principal, $k > 0$ por encima y $k < 0$ por debajo). También se puede usar como `diag(matriz, k)`. Observar como

```
>> v=[1 1 1]; A=diag(v,-1)+2*diag([v 1])+diag(v,1)
```

produce una matriz de bandas.

Ejercicio 7: ¿Qué calcula en realidad el siguiente comando

```
>> prod(diag(A))-prod(diag(flipud(A)))
```

para matrices de orden 2?

La suma, resta, o multiplicación se efectúa de la manera obvia (siempre que los órdenes de las matrices lo permitan):

```
>> A + B, A-B, 5*A, A*B, A^2
```

La inversa de una matriz A se puede escribir o bien `A^(-1)` o usando la función `inv`: `inv(A)`. MATLAB usa métodos numéricos estables (eliminación gaussiana). `det(A)` proporciona el determinante. `rank(A)` estima el rango de A.

MATLAB interpreta `A/B` como AB^{-1} e interpreta `A \ B` como $A^{-1}B$. Asimismo, la solución del sistema $Ax = b$, que formalmente es $x = A^{-1}b$, se obtiene en MATLAB con `A \ b`. Bien entendido que la solución no se obtiene calculando la inversa de A, sino aplicando métodos numéricamente más eficientes (ver `help slash`).

Para multiplicar dos matrices elemento a elemento, se usamos las variantes "punto" de las operaciones correspondientes. Observar la diferencia entre

```
>> A*B, A.*B, A^-1, A.^-1
```

Si A es una matriz real, `A'` es la transpuesta de A. En el caso complejo, `A'` es la transpuesta conjugada. La transpuesta sin conjugar se obtiene con `A.'`

Ejercicio 8: Dada una matriz A, calcular la traza (¡sin usar el comando `trace`!)

MATLAB tiene varias funciones que facilitan la edición de matrices de uso frecuente.

Matrices Especiales

Función	Sintaxis
Matriz identidad de orden n	`eye(n)`
Matriz nula de orden $n \times m$	`zeros(n,m)`
Matriz de unos de orden nula $n \times m$	`ones(n,m)`
Matrices aleatorias de orden $n \times m$	`rand(n,m)`

El comando `rand` produce números aleatorios uniformemente distribuidos en $[0, 1]$. `randn` produce números aleatorios según $N(0, 1)$, la distribución normal de media 0 y desviación típica 1.

Se pueden consultar más tipos de matrices especiales (por ejemplo de Van der Monde, Hilbert, ...) con el comando `help elmat`.

Ejercicio 9: Simular 10 lanzamientos de un dado. Se pueden consultar las funciones de redondeo mediante `lookfor round`, en concreto la parte entera es `floor`.

Ejercicio 10: Si se eligen dos números $x, y \in [0, 1]$ al azar, la probabilidad de que $x^2 + y^2 \leq 1$ es $\frac{\pi}{4}$. Elegir 1000 puntos (x_i, y_i) al azar, contar cuántos de ellos cumplen $x^2 + y^2 \leq 1$, sean n, y aproximar π por $4\frac{n}{1000}$.

Ejercicio 11: Construir el bloque de Jordan de orden 10 asociado al valor propio 5.

Ejercicio 12: Construir la siguiente matriz de Jordan: $\begin{pmatrix} J_1 & O \\ O & J_2 \end{pmatrix}$, donde J_1 y J_2 son dos bloques de Jordan, el primero de orden 10 asociado a 5 y el segundo es de orden 7 y asociado a -2.

MATLAB posee además un número considerable de funciones avanzadas para matrices: Por ejemplo, calcular el espacio nulo, la norma, la exponencial, las factorizaciones LU, de Cholesky, QR, valores y vectores propios, ... Consúltese `help matfun` para una mayor información.

1.8. Polinomios.

MATLAB se vale de vectores para manejar los polinomios. Para ello considera las componentes de un vector como coeficientes de un polinomio, ordenados de mayor a menor grado. Así, por ejemplo, el vector `p = [2 -5 0 -7]`, representa al polinomio $p(x) = 2x^3 - 5x^2 - 7$.

El valor del polinomio p en un punto se halla con `polyval`. Por ejemplo, la orden `polyval(p,3)` evalúa el polinomio en $x_0 = 3$.

Ejercicio 13: Calcular la suma $1 + \frac{1}{3} + \frac{1}{3^2} + \cdots + \frac{1}{3^{10}}$ por lo menos de dos maneras distintas.

Esta función, como muchas de MATLAB , puede evaluarse sobre un vector. Por ejemplo, si, se desea hacer una tabla de 20 valores del polinomio $x^3 - 1$ en el intervalo $[-1, 1]$ se escribirá:

```
>> p=[1 0 0 -1]; x=linspace(-1,1,20); y=polyval(p,x); tabla=[x;y]
```

La gráfica e obtiene con `plot(x,y)`.

Las raíces de un polinomio (reales y complejas e incluso con las multiplicidades correspondientes) se obtienen inmediatamente con `roots`. Para hallar las raíces de $x^4 - 2x^3 + 2x^2 - 2x + 1$ hacemos:

```
>> r = roots([1 -2 2 -2 1])
```

Recíprocamente, dado un vector r, `poly(r)` es el polinomio con coeficiente director unidad cuyas raíces son las componentes del vector.

```
>> poly(r)
```

Las operaciones de suma, resta y producto por un escalar de polinomios corresponden exactamente con las mismas operaciones de vectores. El producto de polinomios se hace con la orden `conv` y la división con resto se hace con

```
[cociente,resto] = deconv( dividendo,divisor)
```

Ejercicio 14: Calcular $(x+1)^4$.

Ejercicio 15: Sean $p(x) = x^4 - x$, $q(x) = x^2 + x + 1$. Hallar el cociente, $C(x)$, y el resto, $R(x)$ de la división euclídea p/q. A continuación comprobar que $p(x) = C(x)q(x) + R(x)$.

1.9. Gráficos.

Hemos utilizado ya algunas órdenes gráficas de MATLAB , como `fplot` o `ezplot` para representar expresiones algebraicas, `compass` y `plot` para representar números complejos. Veremos algún comando más para representar funciones a partir de su expresión en coordenadas cartesianas, polares y paramétricas, valores almacenados en una matriz y funciones de dos variables.

1.9.1. Curvas planas.

La orden `plot(x,y)` representa los pares $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$, siendo $x = (x_1, x_2, \cdots, x_n)$ e $y = (y_1, y_2, \cdots, y_n)$. Hemos usado ya este comando para representar funciones: Por ejemplo, si deseamos representar $p(x) = x^3 - x$ en el intervalo $[-1, 2]$ podemos hacer

```
>> p=[1 0 -1 0]; x=linspace(-1,2); y=polyval(p,x);
>> plot(x,y)
```

Por defecto, `plot` une puntos consecutivos mediante un segmento. Añadiendo un tercer argumento, modificamos el color y el estilo del trazo. Así, con

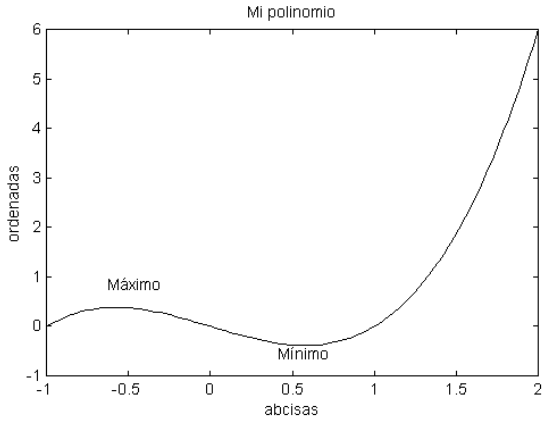
```
>> plot(x,y,'*')
```

visualizamos los puntos que determinan el polinomio del ejemplo anterior.

Con `help plot` vemos las distintas combinaciones de carácteres válidas como tercer argumento de esa función.

La orden `title` sirve para poner un título al gráfico. Igualmente podemos poner títulos en los ejes con `xlabel` e `ylabel`. El argumento de estas funciones ha de ser una cadena o una variable de este tipo.

```
>> title('Mi polinomio')
>> xlabel('Abscisas'), ylabel('Ordenadas')
```



Podemos añadir cualquier texto sobre el gráfico sin más que indicar la posición en que debe aparecer, bien mediante coordenadas o con el ratón. Para escribir la palabra *Máximo* en el punto de coordenadas (1.5, 2) utilizamos

```
>> text(1.5,2,'Máximo')
```

También podemos situar el texto con el ratón

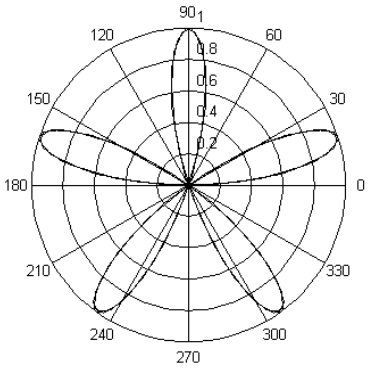
```
>> gtext('Mínimo')
```

Otra posibilidad útil es la de extraer valores del gráfico. `ginput(n)` forma una matriz de 2 columnas y n filas, de modo que cada fila es dónde pinchamos con el ratón en la gráfica.

Ejercicio 16: Dibujar la gráfica de la función $f(x) = \frac{\log x}{x}$ en el intervalo $[1, 4]$ y mediante el comando `ginput` estimar cuál es el extremo.

Para dibujar una curva en coordenadas polares se usa el comando `polar`. Por ejemplo, la ecuación de la *rosa de 5 pétalos* es $\rho(\theta) = \text{sen}(5\theta)$, $\theta \in [0, 2\pi]$. Su representación en MATLAB se obtiene con.

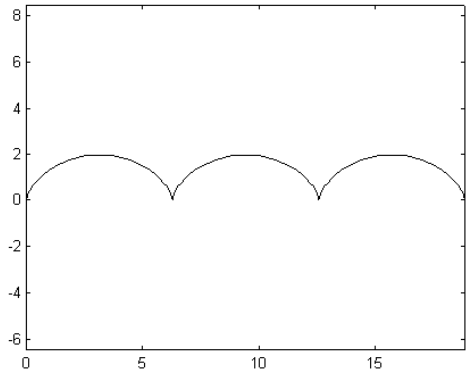
```
>> z = 0:pi/300:pi; r = sin(5*z); polar(z,r)
```



Para representar una curva en paramétricas se procede del modo siguiente: Por ejemplo, la ecuación de la cicloide es $\alpha(t) = (1 - \cos t, t - \operatorname{sen} t)$, $t \in [0, 6\pi]$ se representa por medio de

```
>> t = linspace(0,6*pi);
>> x = t-sin(t); y = 1-cos(t);
>> plot(x,y)
```

Para que vea mejor, se puede escribir `axis equal`.



1.9.2. Curvas en el espacio.

Para representar una curva en el espacio (en paramétricas), $\alpha : [t_0, t_1] \to \mathbb{R}^3$, $\alpha(t) = (x(t), y(t), z(t))$, se usa el comando `plot3`. Para dibujar dos hélices entrelazadas

$$\alpha(t) = (\operatorname{sen} t, \cos t, t), \quad \beta(t) = (-\operatorname{sen} t, -\cos t, t), \quad t \in [0, 4\pi],$$

se escribe

```
>> t=linspace(0,4*pi);
>> x=sin(t);y=cos(t);z=t;
>> plot3(x,y,z,'r',-x,-y,z,'b')
```

Para dibujar una trayectoria helicoidal sobre una superficie cónica:

```
>> t = linspace(0,10*pi);
>> x = t.*sin(t); y = t.*cos(t);
>> plot3(x,y,t)
```

El comando `view` sirve para situar el "punto de vista". La sintaxis es `view(azimut, elevación)` (ambos en grados sexagesimales). ***`Azimut`*** es el ángulo de rotación horizontal y ***`elevación`*** es el vertical. Unos comandos más interactivos (con el ratón) son `rotate3d` y `zoom` en éste último se manejan los dos botones del ratón.

1.9.3. Superficies en el espacio.

La idea básica para representar funciones de dos variables es evaluarlas en un conjunto de puntos del plano XY formando una malla rectangular, almacenar estos valores en una matriz y utilizar órdenes de MATLAB que representan estos valores como alturas sobre el plano y mediante colores o curvas de nivel.

Para representar $z = xy$ en el rectángulo $[-1, 1] \times [-1, 1]$ se procede de la manera siguiente:

```
>> x = -1:0.1:1; y = -1:0.1:1;
>> z = y'*x;                % Columna x Fila = Matriz cuadrada
>> surf(x,y,z)              % Superficie de placas
>> mesh(x,y,z)              % Mallado de la superficie
>> contour(x,y,z)           % Curvas de nivel
>> surfl(x,y,z)             % Superficie suavizada
>> surfc(x,y,z)             % Superficie y curvas de nivel
>> waterfall(x,y,z)         % Cortes verticales
```

Ejercicio 17: Representar la función $z = \operatorname{sen} x \cos y$ en $[0, 2\pi] \times [0, 2\pi]$.

No siempre es tan fácil disponer los valores de la función en una matriz z de modo que la posición en la matriz corresponda con las coordenadas del punto (x, y). MATLAB nos ayuda con la orden `meshgrid`, que a partir de un conjunto de valores de x y otro de y, crea dos matrices, X e Y, con las coordenadas de los puntos de la malla. La matriz z se obtiene evaluando la función que se desea representar sobre el par de matrices obtenidas. Hay que tener cuidado en emplear las versiones "punto" de las operaciones producto, cociente y potencia, pues la función actúa elemento a elemento.

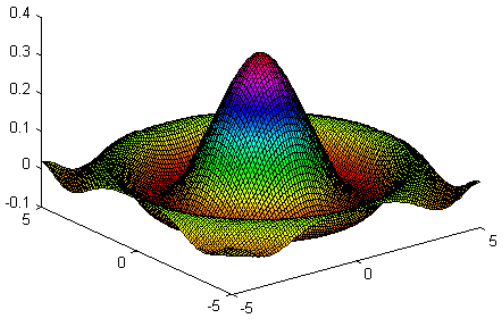
Si se desea representar la cuádrica $z = x^2 - y^2$ en $[-1, 1] \times [-1, 1]$:

```
>> x = -1:0.1:1; y = -1:.1:1;
>> [X,Y] = meshgrid(x,y)                    % Truco!
>> z = X.^2 + Y.^2;                         % !Ojo al punto!
>> surfc(x,y,z)
```

Ejercicio 18: Representar un *sombrero*:

$$f(x, y) = \frac{\cos\left(\frac{x^2+y^2}{4}\right)}{3 + x^2 + y^2}, \qquad f : [-5, 5] \times [-5, 5] \to \mathbb{R}.$$

SUGERENCIA: Para evitar duplicar cálculos utilizar una variable auxiliar, $aux = x^2 + y^2$.



El aspecto de la superficie puede modificarse de varias maneras. Aparte de los comandos vistos para curvas en el espacio, `rotate3d` y `zoom`, están los siguientes: Para crear una malla transparente hacemos

```
>> mesh(x,y,z), hidden off
```

Con `hidden on` volvemos al estado normal en el que se ocultan las líneas posteriores.

Las opciones de sombreado muestran u ocultan la malla sobre la superficie.

```
>> surf(x,y,z), shading flat        % sin malla
>> shading interp                   % color degradado
>> shading faceted                  % losetas con borde
```

El color se cambia con `colormap` que tiene algunas opciones predefinidas (ver la ayuda de `graph3`). Por ejemplo `colormap summer`

La función `pcolor` muestra un rectángulo coloreado según el valor que se representa. `pcolor(x,y,z)`

Jugando con las opciones de sombreado y añadiendo curvas de nivel en negro, creamos un mapa físico:

```
>> hold, shading interp, contour(x,y,z,'k'), hold
```

La letra `k` en la instrucción de `contour` indica el color con el cual queremos dibujar las curvas de nivel: blacK. Si en cambio queremos mostrar las curvas de nivel sobre la superficie, hacemos lo mismo con `surf` y `contour3`

```
>> surf(x,y,z), shading interp, hold contour3(x,y,z,'k'), hold
```

Generar superficies de revolución es muy fácil con MATLAB . Basta crear un vector con los radios de la superficie a intervalos regulares del eje de revolución y utilizar la función `cylinder`. La sintaxis es

`cylinder(`*`vectorderadios, numerodepuntosdelacircunferencia`*`)`

Veamos algunos ejemplos:

```
>> r=2*ones(1,10); cylinder(r)
>> r=2*ones(1,10); cylinder(r,50)
>> r = 1:-.1:0; cylinder(r,100)
```

Ejercicio 19: La ecuación de un toro de radio mayor R y menor r (centrado en el origen), es en ecuaciones cilíndricas: $\rho(z) = R \pm \sqrt{r^2 - z^2}$, $z \in [-r, r]$. Observando que el toro es una superficie de revolución, usar el comando `cylinder` para dibujar un toro cuando $R = 4$, $r = 1$.

Más fácil aún, desde luego, es dibujar una esfera: `sphere`

2. Aprendiendo a programar

2.1. Archivos.m

En estas dos últimas horas aprenderemos (mínimamente) a usar el lenguaje de programació de MATLAB para crear nosotros mismos funciones.

La programación en MATLAB se efectúa mediante **ficheros.m**. Son simplemente ficheros de texto que contienen órdenes de MATLAB . Su uso requiere:

- Editar el fichero con el editor de MATLAB o con un editor ASCII.
- Guardarlo con extensión `.m`.
- Indicar a MATLAB dónde está el archivo con `path(path,'dirección')`.
- Ejecutarlo escribiendo en la línea de órdenes el nombre de fichero y los parámetros de entrada necesarios.

Como ejemplo creamos un fichero que calcula la suma y resta de dos matrices: En el menú **File** se elige **New M-file** para entrar en el editor de MATLAB y en él escribimos el siguiente texto:

```
function [suma,resta] = sumres(A,B)
% Datos de entrada: Dos matrices A y B.
% Salida: Dos matrices: sum = A+B, res = A-B.
suma = A+B;
resta = A-B;
```

- Se graba en **a:** con el nombre **sumres.m**
- Se indica la ruta para llegar al fichero: `path(path,'a:')`
- Se ejecuta: `[M,N] = sumres(eye(2),eye(2))`
- Obsérvese la importancia de las primeras líneas: `help sumres`

El fichero **sumres.m** es un archivo de función, al incluir en la primera línea la palabra clave `function`. La sintaxis es la siguiente: Si $x_1, \cdots, x_m$ son los *argumentos de entrada* e $y_1, \cdots, y_m$ son los *argumentos de salida*

`function [y1, ..., ym]=nombre_de_función (x1, ..., xn)`

Es aconsejable que el nombre de la función coincida con el nombre del fichero `.m`

La orden `type(fichero)` muestra en pantalla el listado del programa. Podemos escribir `type sumres`. Hay muchas funciones implementadas en MATLAB que podemos ver su listado. Por ejemplo, `type hilb` muestra en pantalla cómo formar la matriz de Hilbert.

Veamos un ejemplo más complicado: Aunque ya sepamos dibujar curvas paramétricas en el espacio (`plot3`), veamos otra manera: La *proyección isométrica*, es una aplicación lineal, p, de $\mathbb{R}^3$ a $\mathbb{R}^2$ tal que $p(e_1) = (\cos 210^o, \operatorname{sen} 210^o)$, $p(e_2) = (\cos 330^o, \operatorname{sen} 330^o)$, $p(e_3) = (0, 1)$. Luego la proyección de (x, y, z) se puede calcular

$$\begin{pmatrix} -\cos(\pi/6) & cos(\pi/6) & 0 \\ -\operatorname{sen}(\pi/6) & -\operatorname{sen}(\pi/6) & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

```
function dibuja(v)

% dibuja curvas 3D en proyecci'on isom'etrica.
% Ejemplo: H'elice a(t)=(cost,sent,t/4)
%          t=0:0.05:8*pi;
%          x=cos(t);
%          y=sin(t);
%          z=t/4;
%          v=[x;y;z];
%          dibuja(v)

% Inicializa la matriz proyecci'on
iso=[-cos(pi/6) cos(pi/6) 0;-sin(pi/6) -sin(pi/6) 1];

% Misma escala, borra los ejes y superpone varios gr'aficos.
axis('equal')
axis off
hold on

% Dibuja el eje x
```

```
xmax=max(v(1,:));
xm=[xmax;0;0];
m=1.2*iso*xm;
plot([0,m(1)],[0,m(2)],'b')
text(m(1),m(2),'x')

% Dibuja el eje y
ymax=max(v(2,:));
ym=[0;ymax;0];
m=1.2*iso*ym;
plot([0,m(1)],[0,m(2)],'b')
text(m(1),m(2),'y')

% Dibuja el eje z
zmax=max(v(3,:));
zm=[0;0;zmax];
m=1.2*iso*zm;
plot([0,m(1)],[0,m(2)],'b')
text(m(1),m(2),'z')

% Dibuja la curva en 3d
m=iso*v;
x=m(1,:);
y=m(2,:);
plot(x,y)
```

Una versión matricial (más compacta) del programa anterior es la siguiente. (La forma de ejecutar la función es exactamente la misma que la anterior)

```
function iso(v)
% dibuja curvas 3D en proyecci'on isom'etrica.

iso=[-cos(pi/6) cos(pi/6) 0;-sin(pi/6) -sin(pi/6) 1];
axis('equal')
axis off
hold on

% dibuja ejes
ejes=diag(max(v'));
m=1.2*iso*ejes;
compass(m(1,:),m(2,:),'b')
text(m(1,:),m(2,:),['x';'y';'z'])
```

```
% dibuja la curva
m=iso*v;
plot(m(1,:),m(2,:),'r')
```

Ejercicio 20: Calcular la suma de los n primeros naturales.

Ejercicio 21: Elaborar un archivo `.m` para calcular $S(n) = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2}$. Y comprobar que para valores "grandes" de n, $S(n) \simeq \frac{\pi^2}{6}$.

Ejercicio 22: Sea A una matriz real de orden 2. Se puede medir intuitivamente el condicionamiento de la matriz A observando la elipse $A(S)$, donde S es $\{(x, y) \in \mathbb{R}^2 : \|(x, y)\| = 1\}$ (Esta elipse es la cónica de ecuación $X^t(AA^t)^{-1}X$ supuesta A invertible, si no lo fuese, entonces ¡$A(S)$ es un segmento!). Cuanto más achatada sea esta elipse, peor es el condicionamiento de A. Elaborar un programa que calcule el número de condición de una matriz A (`cond`) y que dibuje la cónica $A(S)$.

Hay una forma rápida en MATLAB para contar el número de operaciones efectuadas. (Lo que nos puede decidir entre varios algoritmos diferentes). Si escribimos

```
>> flops
```

MATLAB nos proporciona el número de operaciones en coma flotante efectuadas desde que hemos iniciado MATLAB . Por tanto la mejor manera para contar el número de operaciones de un cierto número de instruciones es la siguiente

```
operaciones_hasta_ahora = flops
Instrucciones
numero_de_operaciones = flops - operaciones_hasta_ahora
```

Por ejemplo podemos estimar el número de operaciones para elevar una matriz de orden n al cuadrado de la manera siguiente:

```
function operaciones=estima(n)
% estima(n).  Estima el numero de operaciones para calcular el
% cuadrado de una matriz de orden n
a=rand(n);
operaciones=flops;
a^2;
operaciones=flops-operaciones;
```

El siguiente ejercicio muestra un ejemplo sencillo de cómo se obtienen número de operaciones diferentes al calcular lo mismo.

Ejercicio 23: Dado un vector $v = (v_1, v_2, v_3, \cdots, v_n)$ podemos construir el vector de

sumas acumuladas mediante

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \cdots \\ v_n \end{pmatrix}.$$

La matriz se puede implementar usando `tril(ones(n))`[1]. Elaborar un archivo `.m` para calcular el vector de sumas acumuladas y contar el número de operaciones. Repetir lo mismo mediante la orden `cumsum`.

Una de las ventajas de las funciones de MATLAB es que podemos usar funciones implementadas por nosotros mismos para elaborar otro tipo de programas. El siguiente ejemplo es típico:

Se trata de evaluar de forma aproximada por el método del punto medio una integral definida. Para ello escribimos dos programas, uno donde se almacena el integrando y el otro en donde se describe el método. Y así cuando queramos modificar el integrando solo modificamos el fichero donde lo tengamos almacenado y no en donde está implementado el método, ganando considerablemente en claridad: Si queremos calcular de forma aproximada $\int_0^1 \frac{\operatorname{sen} x}{x} dx$, guardamos en el fichero **f.m** el integrando

```
function y=f(x)
% Aqu'i almacenamos una funci'on y=f(x).
y=sin(x)./x;
```

y en otro fichero; por ejemplo **integral.m** el método:

```
function I=integral(a,b,n)
% --- Calcula la integral por el m'etodo del punto medio ---
% integral(a,b,n):
% a:extremo inferior.  b:extremo superior.  n:n'umero de divisiones.
h=(b-a)/n;
x=a+h/2:h:b-h/2;
I=h*sum(f(x));
```

Ejercicio 24: Elaborar un programa para calcular una integral definida por el método de los trapecios. Si se divide $[a, b]$ en la partición equiespaciada por h, $a = x_0 < a + h = x_1 < a + 2h = x_2 < \cdots < x_n = b$, la integral aproximada es

$$I \simeq \frac{h}{2} \left(f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n) \right).$$

[1]Los comandos `tril` y `triu` forman matrices triangulares inferior y superior respectivamente.

Intentar evaluar $\int_0^1 \frac{\operatorname{sen} x}{x} dx$. SUGERENCIA: Al intentar evaluar la función $f(x) = \frac{\operatorname{sen} x}{x}$ en $x = 0$, hay un problema de indeterminación. En este caso se resuelve fácilmente modificando ligeramente la función: `sin(x+eps)./(x+eps)`.

2.2. Bucles.

Un bucle es un conjunto de operaciones, del cual sabemos que se va a ejecutar una cantidad concreta de veces. Por ejemplo, si queremos formar el vector $v = (1, 2, \cdots, n)$ (de una manera poco eficiente) podemos ejecutar

`Hacer desde i=1 hasta n` → `v(i)=i` → `siguiente i`

La sintaxis correcta es (inicializando el vector v)

```
n=100;
v=zeros(1,n);
for i=1:n
   v(i)=i;
end
```

En lo posible se evitará el uso de bucles en MATLAB ; ya que suponen un excesivo gasto de tiempo. Por ejemplo, lo anterior simplemente se implementaría como `v=1:100`. Otro ejemplo: Los dos siguientes calculan la suma de los n primeros cuadrados

```
function s=suma(n)
s=0;
   for i=1:n
   s=s+i^2;
end
```

```
function s=suma(n)
s=sum((1:n).^2);
```

El programa sin usar bucles gana en legibilidad y en tiempo, como se muestra en el siguiente ejercicio:

Ejercicio 25: Usar los comandos `tic` y `toc` para contar el tiempo de ejecución y comprobar que para los bucles el tiempo es mayor. La forma de usar estos comandos es `tic` *operaciones* `toc`.

Un ejemplo un poco más complicado de utilización de bucles es el siguiente que muestra la bola unidad en $\mathbb{R}^2$ para diferentes tipos de normas (¡e incluso la norma infinito!)

```
function bola(p,intentos)
% --- bola(p,intentos) ---
% BOLA  Una demostraci'on de la forma de la bola unidad en R2.
%       La norma es ||(x,y)|| = (x^p+y^p)^(1/p). (p puede ser Inf!)
%       intentos: el n'umero de vectores que se dibujan

clf %sirve para borrar cualquier gr'afico anterior.
axis('square');axis([-1.0 1.0 -1.0 1.0])
plot([-1.0 1.0],[0 0],'b-',[0 0],[-1.0 1.0],'b-') %dibuja los ejes
hold on
axis('equal')
theta=linspace(0,2*pi,intentos+1); %Calcula 'angulos equiespaciados
theta=theta(1:length(theta)-1);
x=cos(theta); %Calcula los vectores
y=sin(theta);
M=[x;y];
norma=zeros(1,length(theta)); %Inicializa el vector de normas
for k=1:length(theta) %Calcula la norma p de cada vector
   norma(k)=norm(M(:,k),p);
end
x=x./norma; y=y./norma; %Normaliza
plot(x,y,'ko') %Dibuja en la punta del vector un c'irculo negro
```

También es posible *anidar* un bucle dentro de otro. En muchas ocasiones esto es especialmente útil para definir matrices. Por ejemplo, la matriz de Hilbert se define como $h_{ij} = \frac{1}{i+j-1}$. Podemos elaborar el siguiente programa:

```
function H=mihilb(n)
H=zeros(n);
for i=1:n
  for j=1:n
    H(i,j)=1/(i+j-1);
   end
end
```

Aunque, repetimos, es preferible, si se puede, evitar el uso de bucles: Escríbase `type hilb` para observar cómo se puede implementar la matriz de Hilbert sin uso de bucles.

El *paso* de los bucles no tiene por qué ser 1, Puede ser cualquier número real (incluso negativo), como muestra el siguiente programa que calcula la solución de $Ax = b$, siendo A una matriz triangular superior de orden n con ningún elemento nulo en la diagonal

principal y b un vector columna. Estos sistemas se resuelven primero hallando x_n, luego x_{n-1} y así sucesivamente. Supuestos hallado hasta x_{k+1}, entonces

$$x_k = \frac{b_k - \sum_{j=k+1}^{n} a_{kj}x_j}{a_{kk}},$$

el sumatorio se puede implementar fácilmente como la fila k de A por el vector x columna.

```
function x=sr(A,b)
% x=sr(A,b)
% resuelve el sistema Ax=b por substituci'on regresiva
% A ha de ser una matriz triangular superior cuadrada
% con ning'un 0 en la diagonal principal.
% b debe ser un vector columna

n=length(A);
x=zeros(n,1);
for k=n:-1:1
  x(k)=(b(k)-A(k,:)*x)/A(k,k);
end
```

A la hora de elaborar programas en MATLAB conviene tener en cuenta una serie de cuestiones para agilizar el rendimiento:

- Elegir algoritmos numéricamente robustos. Hay que tener en cuenta que MATLAB trabaja con precisión finita. Por ejemplo

  ```
  >> (100+1e-15-100)*1e32
  ```

  ```
  >> (100-100+1e-15)*1e32
  ```

  producen ¡0 e 10^{17}! respectivamente.

- Evitar en lo posible el uso de bucles, si se pueden usar funciones de tipo vectorial. (Antes de las dos sentencias se ha ejecutado `clear all`)

  ```
  >> S=0;tic;for k=1:10000;S=S+k;end;toc
  ```

  ```
  >> tic;S=sum(1:10000);toc
  ```

- Dimensionar previamente los vectores. Si ha medida que funciona un programa, un vector se obliga a que cambie la dimensión, MATLAB ralentiza considerablemente el proceso: (Ejecutándose previamente `clear all`)

  ```
  >> tic;for k=1:10000;v(k)=2*k-1;end;toc
  ```

  ```
  >> tic;v=zeros(1,10000);for k=1:10000;v(k)=2*k-1;end;toc
  ```

  ```
  >> tic; v=1:2:20000; toc
  ```

2.3. Condicionales.

Si queremos usar una instrucción, no un número predeterminado de veces (instrucción `for`), sino hasta que se satisfaga determinada condición; hemos de usar los *condicionales* `if` y `while`.

2.3.1. La condición `if`

Una estructura de control útil es la bifurcación condicional `if`. Las instrucciones que contienen `if` se ejecutan si se satisface la condición. Si ésta no se verifica, se produce un salto hasta la orden siguiente a `end`.

La sintaxis de la bifurcación condicional `if` es la siguiente:

```
if condición
   instrucciones
end
```

La instrución `if` se suele usar junto con `else`: Esta última instrucción permite introducir instrucciones que se ejecutan si la condición es falsa. La sintaxis es

```
if condición
   instrucciones si la condición es cierta
else
   instrucciones si la condición es falsa
end
```

Veamos algunos detalles sintácticos a la hora de establecer condiciones. MATLAB reconoce expresiones lógicas asociándoles el valor 1 si son ciertas y 0 si son falsas (¡de manera vectorial!). Por ejemplo, si $a = (-1, 0, 1)$ y si $b = (0, 1, -1)$; entonces

Igualdades: `a==0` devuelve `[0 1 0]`. Observar que la *asignación* se realiza con `=`; mientras que la *comparación* con `==`.

Desigualdades:

- `a>0` devuelve `[0 0 1]`.
- `a>=0` devuelve `[0 1 1]`.
- `a~=0` devuelve `[1 0 1]`. El símbolo `~` se introuduce con [ALT]+126.

Operadores lógicos: **Conjunción** `(a>=0)&(b>0)` devuelve `[0 1 0]`.

Disyunción `(a>0)&(b<=0)` devuelve `[1 0 1]`.

Disyunción exclusiva `xor((a==1),(b>=-1)` devuelve `[1 1 0]`.

Negación `~(a==1)` devuelve `[1 1 0]`.

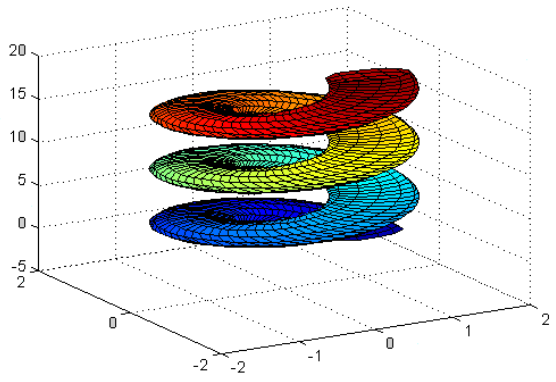
Veamos un ejemplo sencillo: Dado un número r, decidir si es entero o no:

```
function esentero(r)
if r==floor(r)
  disp('es entero')
else
  disp('no es entero')
end
```

Por supuesto los condicionales se pueden anidar uno dentro de otro.

Ejercicio 26: Modificar el programa anterior para decidir si un número es par o no, sólo en el supuesto de que sea entero.

El siguiente fichero dibuja una *superficie tubular* de radio R alrededor de la curva $\alpha(t) = (x(t), y(t), z(t))$.



Las líneas 11 – 13 muestran una utilidad típica del condicional `if` mediante la variable `nargin` (=Number of ARGuments IN; por supuesto, también está la variable `nargout`).

La línea 23 calcula los vectores (almacenados en `v`) que forman la poligonal asociada a la curva; que aproximan a los vectores tangentes a la curva. Ahora hay que construir circunferencias perpendiculares. Las líneas 25 – 32 calculan las coordenadas esféricas (α = latitud; β = longitud) del vector $v/\|v\|$. Si

$$\frac{v}{\|v\|} = (\cos\alpha\cos\beta, \cos\alpha\,\mathrm{sen}\,\beta, \mathrm{sen}\,\alpha),$$

es fácil comprobar que

$$w_1 = (\mathrm{sen}\,\alpha\cos\beta, \mathrm{sen}\,\alpha\,\mathrm{sen}\,\beta, -\cos\alpha), \quad w_2 = (\mathrm{sen}\,\beta, -\cos\beta, 0)$$

forman una base ortonormal del plano ortogonal a v. Por tanto

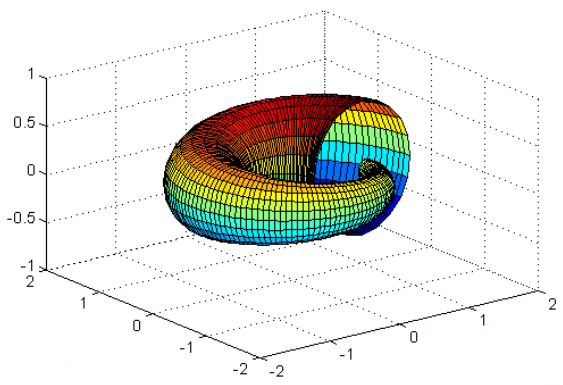
$$w = w_1\cos\theta + w_2\,\mathrm{sen}\,\theta, \qquad \theta \in [0, 2\pi]$$

describe la circunferencia de radio 1 centrada en el origen perpendicular a v. Éste es el objeto de las líneas 33 – 35. El resto del programa es fácil de entender.

```
1   function gusano(xc,yc,zc,R,ntheta)
2   %  function gusano(Xc,Yc,Zc,R,ntheta)
3   %
4   %         --- Dibuja una superficie tubular ---
5   %  sobre la curva [Xc Yc Zc] con radio transversal R.
6   %  La curva debe tener al menos dos puntos.
7   %  ntheta es el numero de puntos de cada circunferencia transversal.
8   %  Por defecto ntheta=12.
9
10  %preparaci'on
11  if nargin==4
12    ntheta=12;
13  end
14  h=2*pi/ntheta;
15  theta=(0:h:2*pi)';
16  ct=cos(theta);
17  st=sin(theta);
18  n=length(xc);
19  X=zeros(n,ntheta+1);Y=X;Z=X;
20
21  %Construye n-1 c'irculos perpendiculares a cada segmento
22  for i=1:n-1
23    v=[xc(i+1)-xc(i),yc(i+1)-yc(i),zc(i+1)-zc(i)];
24    V=norm(v);
25    salfa=v(3)/V;
26    calfa=sqrt(v(1)*v(1)+v(2)*v(2))/V;
27    sbeta=0;
28    cbeta=1;
29    if calfa~=0
30      sbeta=v(2)/V/calfa;
31      cbeta=v(1)/V/calfa;
32    end
33    x=+ct*salfa*cbeta+st*sbeta;
34    y=+ct*salfa*sbeta-st*cbeta;
35    z=-ct*calfa;
36    X(i,:)=x';Y(i,:)=y';Z(i,:)=z';
37  end
38
39  %Mueve y escala los c'irculos
```

```
40  for i=1:n-1
41     X(i,:)=X(i,:)*R+xc(i);
42     Y(i,:)=Y(i,:)*R+yc(i);
43     Z(i,:)=Z(i,:)*R+zc(i);
44  end
45  X(n,:)=X(n-1,:);
46  Y(n,:)=Y(n-1,:);
47  Z(n,:)=Z(n-1,:);
48  surf(X,Y,Z);
```

Ejercicio 27: Modificar el fichero previo para elaborar un programa que dibuje la superficie tubular alrededor de la curva $\alpha(t) = (x(t), y(t), z(t))$, $t \in [a, b]$ con radio variable $R : [a, b] \to \mathbb{R}^+$. Por ejemplo el dibujo para la curva $\alpha(t) = (\cos t, \operatorname{sen} t, 0)$, $t \in [0, 2\pi]$, $R(t) = 0.7 - \frac{0.6}{2\pi}t$, debe ser



2.3.2. La condición `while`

Si se quiere repetir un conjunto de instrucciones no un número determinado de veces, sino hasta que se cumpla una determinada condición, lo más adecuado es usar la sentencia `while`. Las instrucciones que contiene el `while` se repiten hasta que se satisfaga la *`condición`*. Si ésta no se verifica, se produce un salto hasta la orden que sigue al `while`. Para que el bucle no sea indefinido, las instrucciones del `while` modifican la *`condición`*, de modo que en algún momento deje de verificarse. La sintaxis es la siguiente:

```
while condición
   instrucciones
end
```

Veamos un ejemplo sencillo: Dado $\epsilon > 0$, encontrar el menor natural n tal que $\frac{\pi^2}{6} - \sum_{k=1}^{n} \frac{1}{k^2} < \epsilon$

```
function n=mierror(epsilon)
% n=mierror(epsilon)
% Dado epsilon>0 halla el menor natural n tal que
% pi*pi/6-(1+1/4+1/9+...+1/n*n)< epsilon
S=0;
k=1;
sumaexacta=pi^2/6;
while sumaexacta-S>epsilon
   S=S+1/k^2;
   k=k+1;
end
n=k-1;
```

El siguiente ejemplo muestra cómo se puede generar una sucesión de punto fijo. Dado una función f y una estimación a de la raíz de $f(x) = x$, se estudia la convergencia de la sucesión formada por $x_{n+1} = f(x_n)$. Hay que escribir antes en un fichero `.m` la función f.

```
function pfijo(a,precision,itermax)
% calcula y dibuja los primeros iterados de punto fijo de la funcion f
% La sintaxis es:
% pfijo=(a,precision,itermax)
% a = Punto inicial,
% precision = Precisi'on deseada,
% itermax = N'umero m'aximo de iteraciones.

x=[a];
contador=1;
while  abs(a-f(a)) > precision & contador < itermax
  b=f(a);
  x=[x,b];
  a=b;
  contador=contador+1;
end
n=length(x)-1;
if (contador==itermax | abs(a)==inf)
   disp('No converge')
else
   disp('Soluci'on: ')
   a
   disp('Iteraciones: ')
   n
```

```
end
if abs(a)~=Inf
   disp('Pulse una tecla')
   pause
   minx=min(x);
   maxx=max(x);
   rango=maxx-minx;
   u=linspace(minx-0.1*rango,maxx+0.1*rango,100);
   v=f(u);
   hold off
   plot(u,v,'k')
   hold on
   plot(u,u,'r')
   title('Funcion y=f(x) en negro, y=x en rojo.')
   for k=1:n-1
      line([x(k),x(k+1)],[x(k+1),x(k+1)])
      line([x(k+1),x(k+1)],[x(k+1),x(k+2)])
   end
   hold off
end
```

Por ejemplo, para la ecuación $\cos x = x$ se puede ver que el método converge hacia la raíz $\xi = 0.7385$.

Ejercicio 28: Modificar el fichero de punto fijo anterior para implementar el método de Newton.

Ejercicio 29: Comprobar que las sucesiones de punto fijo para las ecuaciones

$$\operatorname{sen}(\pi x) = x; \quad 4x(1-x) = x,$$

con punto inicial $x_0 \in [0,1]$ tienen un carácter caótico.

La sucesión

$$x_0 = 0.5, \qquad x_{n+1} = \lambda x_n(1-x_n), \qquad n \in \mathbb{N}$$

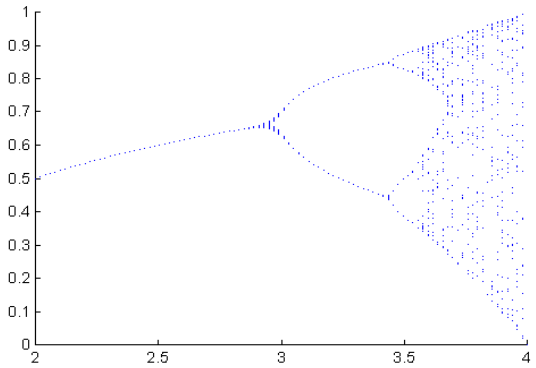
tiene un comportamiento caótico para algunos valores de $\lambda \in [0,4]$. Para valores de λ cercanos a 0, la sucesión converge a un valor, para $\lambda \in [3, 3.4]$ (aproximadamente) la sucesión a la larga oscila hacia dos valores, y el comportamiento para λ cercano a 4 es completamente imprevisible. El comportamiento caótico se puede comprobar mediante el ejercicio siguiente:

Ejercicio 30: Consideremos la sucesión $x_0 = 0.5$, $x_{n+1} = \lambda x_n(1-x_n)$ para $\lambda \in [0,4]$. Para el estudio "a la larga" de la sucesión se va a tomar los términos $(x_{30}, \cdots x_{60})$, y se

dibujarán los puntos (λ, x_i) para $\lambda \in [0,4]$, $i = 30, \cdots, 60$. Elaborar un programa en donde se estudie dicho comportamiento. El pseudocódigo sería el siguiente:

Datos de entrada: a, b ($[a,b]$ subintervalo de [0,4]).
$\lambda = (a, a+h, a+2h, \cdots, a+nh = b)$ (para un h pequeño).
Para $i = 1$ hasta $i = n$ hacer
    $x = 0.5$
    Calcular los 60 primeros términos de la sucesión x_n.
    Dibujar (λ, x_j) para $j = 30, \cdots, 60$.
fin para el bucle de i.

La siguiente figura muestra el comportamiento para $\lambda \in [2,4]$:



También se puede estudiar el comportamiento caótico de la siguiente sucesión: $x_{n+1} = \lambda \operatorname{sen}(\pi x_n)$, $\lambda \in [0,1]$ y $x_0 = 0.5$.

3. Soluciones a algunos ejercicios.

2: `1./(1:10)`

3: `(10:-1:1)./(1:10)`

4: `sum(1:1000)`

5: `sum(v)/length(v)`

6: `sum(sin(linspace(2*pi/15,2*pi,15)))`

8: `sum(diag(A))`

9: `floor(6*rand(1,10))+1`

10: Se pueden generar 1000 pares de números mediante

```
>> A=rand(2,1000);
```

Formemos una matriz correspondiente a $x^2 + y^2$:

```
>> B=A(1,:).^2+A(2,:).^2;
```

Los pares tales que $x^2 + y^2 < 1$ cumplen $[x^2 + y^2] = 0$; los otros cumplen $[x^2 + y^2] = 1$. Luego

```
>> n=1000-sum(floor(B));
```

y así

```
>> aproximapi=4*n/1000
```

11: `v1=ones(1,10); v2=5*ones(1,9); J=diag(v1)+diag(v2,1)`

12:
```
v1=ones(1,10); v2=5*ones(1,9); J1=diag(v1)+diag(v2,1)
w1=ones(1,7); v2=(-2)*ones(1,6); J2=diag(w1)+diag(w2,1)
J=[J1 zeros(10,7);zeros(7,10) J2]
```

13: `p=ones(1,11);suma=polyval(p,1/3)`

o también

`sum((ones(1,11)/3).^(0:1:10))`

14: `p=[1 1]; q=conv(p,p); r=conv(q,q)`

Observar que, aunque correcta, la siguiente expresión es inferior, ya que en ésta $(1 + x)^2$ es calculado 2 veces en vez de 1 sóla vez.

```
p=[1 1]; r = conv(conv(p,p),conv(p,p))
```

15:
```
p=[1 0 0 -1 0]; q=[1 1 1]; [C R] = deconv(p,q)
tienequedar0 = conv(C,q)+R-p
```

18: Se puede representar de esta manera:
```
>> x=-10:0.1:10;
>> y=x;
>> [X,Y]=meshgrid(x,y);
>> aux=X.^2+Y.^2;
>> numerador=cos(aux/4);
>> denominador=3+aux;
>> z=numerador./denominador;
>> surf(x,y,z)
```

19: Usamos dos veces `cylinder` pues hay que representar dos funciones (para el más y para el menos de la raíz):
```
>> z = -1:.1:1;
>> r1 = 4 + sqrt(1-z.^2);
>> r2 = 4 - sqrt(1-z.^2);
>> cylinder(r1), hold, cylinder(r2), hold
```

20: El siguiente programa calcula la suma de los n primeros naturales:
```
function s=suma(n)
% Calcula la suma de los n primeros naturales
v=1:n;
s=sum(v);
```

21: El siguiente programa calcula $s = 1 + \frac{1}{2^2} + \cdots \frac{1}{n^2}$ y $\frac{\pi^2}{6} - s$.
```
function [s,error]=sinv(n)
% Calcula s=1+1/2+...+1/(n*n) y pi*pi/6-s.
v=1./((1:n).^2);
s=sum(v);
error=pi^2/6-s;
```

22: Un punto arbitrario de $A(S)$ es $A(\cos\theta, \operatorname{sen}\theta)$, donde $\theta \in [0, 2\pi]$. Así:

```
function condi(A)
theta=linspace(0,2*pi);
X=[cos(theta);sin(theta)];
Y=A*X;
title(cond(A))
plot(Y(1,:),Y(2,:))
```

23: Se puede implementar por medio de:

```
function [w1,n1,w2,n2]=suma(v)
% w:=sumas acumuladas de las dos maneras.
% n1:=numero de operaciones sin usar cumsum
% n2:=numero de operaciones usando cumsum
op=flops;
n=length(v);
a=tril(ones(n));
w1=a*v;
n1=flops-op;
op=flops;
w2=cumsum(v);
n2=flops-op;
```

24: Una vez creado el archivo **f.m** en donde se almacena el integrando, la fórmula de los trapecios se elabora:

```
function I=trapecio(a,b,n)
h=(b-a)/n;
x=a:h:b;
y=f(x);
I=h*(sum(f(x))+f(a)+f(b));
```

26: Hay que anidar condicionales:

```
function par(r)
if r==floor(r)
   if r==2*floor(r/2)
      disp('es par')
   else
      disp('no es par')
   end
end
```

27: No hay más que modificar las líneas 41–43, por ejemplo la 41 sería

```
X(i,:)=X(i,:)*R(i)+xc(i);
```

30: Si dividimos el intervalo $[a, b]$ en 100 subintervalos:

```
function caos(a,b)
hold on
lambda=linspace(a,b);
for i=1:100
   x=0.5;
   for j=1:60
      x=lambda(i)*x*(1-x);
```

```
        if j>29
           plot(lambda(i),x)
        end
     end
  end
```

A. Mínimos cuadrados.

Éste y el siguiente apéndice constituyen algunos ejemplos más o menos complicados de una utilización (más bien pedagógica) de MATLAB .

El ajuste de datos por el método de los mínimos cuadrados es muy importante, a la par que bastante bien estudiado. MATLAB dispone de una función `polyfit` que implementa dicho método. En el programa adjunto se proporciona una manera alternativa.

Obsérvese el comando `menu` para crear menús interactivos. La orden `strvcat` concatena cadenas verticalmente.

```
function ajuste
% Realiza el ajuste por minimos cuadrados mediante
% Rectas, Hiperplanos, Polinomios, exponenciales.
ayuda=strvcat('Si la tabla de datos es',...
' ',...
'x : x1 x2 ... xn',...
'y : y1 y2 ... yn',...
' ',...
'Cuando le pregunte vector x, introduzca',...
' ',...
'[x1 x2 ... xn]',...
' ',...
'Analogo para y',...
' ',...
'Pulse una tecla');
K = MENU('Elija una opcion', ...
    'Ajuste por rectas: y=a+bx', ...
    'Ajuste por polinomios: y=a0+a1*x+...+am*x^m', ...
    'Ajuste exponencial: y=a*e^(bx)');
if K==1
  clc
  J = MENU('Elija una opcion', ...
    'Ayuda', ...
```

```
    'Calculos');
  if J==1
    clc
    disp(ayuda)
    J=2;
    pause
  end
  if J==2
    clc
    x=input('vector x?  ');
    y=input('vector y?  ');
    flag=0;
    aju1(x,y,flag)
  end
end
if K==2
   clc
   J = MENU('Elija una opcion', ...
    'Ayuda', ...
    'Calculos');
   if J==1
     clc
     disp(ayuda)
     pause
     J=2;
   end
   if J==2
     clc
     x=input('vector x?  ');
     y=input('vector y?  ');
     m=input('grado del polinomio?  ');
     aju2(x,y,m)
   end
end
if K==3
   J = MENU('Elija una opcion', ...
   'Ayuda',...
   'Calculos');
   if J==1
     clc
     disp(ayuda)
   pause
```

```
   end
   clc
   x=input('vector x?  ');
   y=input('vector y?  ');
   y=log(y);
   flag=1;
   aju1(x,y,flag)
end
```

Este fichero "llama" a `aju1.m` que es utilizado para el ajuste por rectas (`flag=0`) y exponenciales (`flag=1`) y también a `aju2.m` que es el ajuste para polinomios.

```
function aju1(x,y,flag)
clc
if length(x)~=length(y),
   error('Las dimensiones de x e y deben coincidir.')
end
mediax=mean(x);
mediay=mean(y);
varianzax=varianza(x);
covarianzaxy=covar(x,y);
pendiente=covarianzaxy/varianzax;
if flag==0
   disp('Si sale [a b c], la recta es y=a+b(x-c)')
end
if flag==1
   disp('Si sale [a b c], la curva es y=a*e^(b(x-c))')
end
[mean(y) pendiente mean(x)]
disp('Pulse una tecla para ver el error y el Indice de determinacion')
pause
varianzay=varianza(y);
disp('Error cuadratico:')
length(x)*(varianzax*varianzay-covarianzaxy^2)/varianzax
disp('Indice de determinacion')
(covarianzaxy^2)/(varianzax*varianzay)
if flag==0
  disp('Pulse una tecla para ver la grafica')
  pause
  hold on
  plot(x,y,'o')
  minimox=min(x);
```

```
  maximox=max(x);
  t=[minimox maximox];
  y=mean(y)*[1 1]+pendiente*(t-mean(x)*[1 1]);
  plot(t,y)
end
if flag==1
   disp('Pulse una tecla para ver la grafica')
   pause
   hold on
   plot(x,exp(y),'o')
   minimox=min(x);
   maximox=max(x);
   t=linspace(minimox,maximox);
   yt=exp(mean(y))*exp(pendiente*(t-mediax*ones(1,length(t))));
   plot(t,yt)
end
```

Para dibujar la recta se aprovecha que basta hallar dos puntos y unirlos. En este archivo se llaman a dos funciones para calcular la varianza y la covarianza:

```
function z = covar(x,y)
xy=x.*y;
z=mean(xy)-mean(x)*mean(y);

function y=varianza(x)
auxiliar=x-mean(x)*ones(1,length(x));
auxiliar=auxiliar.^2;
y=mean(auxiliar);
```

El ajuste por polinomios está implementado en la siguiente función: Es fácil probar que si $p(x) = a_0 + a_1 x + \cdots + a_m x^m$ es el polinomio que mejor ajusta a los datos $x = (x_1, \cdots, x_n)$, $y = (y_1, \cdots, y_n)$, entonces

$$\begin{pmatrix} 1 & \mathsf{E}(x) & \cdots & \mathsf{E}(x^m) \\ \mathsf{E}(x) & \mathsf{E}(x^2) & \cdots & \mathsf{E}(x^{m+1}) \\ \cdots & \cdots & \cdots & \cdots \\ \mathsf{E}(x^m) & \mathsf{E}(x^{m+1}) & \cdots & \mathsf{E}(x^{2m}) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_m \end{pmatrix} = \begin{pmatrix} \mathsf{E}(y) \\ \mathsf{E}(yx) \\ \cdots \\ \mathsf{E}(yx^m) \end{pmatrix};$$

donde $\mathsf{E}(Z)$ indica la media aritmética de la variable Z, que en MATLAB se escribe como `mean(Z)`. Como la matriz suele estar mal condicionada; pero es definida positiva; para resolver el sistema se hace la factorización de Cholesky: `chol`.

```
function aju2(x,y,m)
clc
if length(x)~=length(y),
   error('Las dimensiones de x e y deben coinidir.')
end
if m==length(x) | m>length(x)
   m=length(x)-1;
end
matriz=zeros(m+1,m+1);
for i=1:m+1
   for j=1:m+1
      matriz(i,j)=mean(x.^(i+j-2));
   end
end
b=zeros(m+1,1);
for i=2:m+1
   b(i)=mean(y.*(x.^(i-1)));
end
clc
disp('Si la solucion es [a0 a1 ... am]')
disp(' es a0+a1*x+...+am*x^m')
R=chol(matriz);
u=R'\b;
s=(R\u)';
disp('Pulse una tecla para saber el error cuadratico')
pause
disp('Error= ')
s=fliplr(s);
p=polyval(s,x);
(norm(p-y))^2
disp('Pulse una tecla para ver la grafica')
pause
hold on
plot(x,y,'o')
xmin=min(x);
xmax=max(x);
t=linspace(xmin,xmax);
yt=polyval(s,t);
plot(t,yt)
```

B. Fractales.

En este apéndice usaremos las capacidades gráficas de MATLAB para poder dibujar algunos fractales conocidos, como el triángulo de Sierpinsky, la curva de Koch o el de Mandelbrot. Todos los fundamentos matemáticos de que se compone este apéndice están extraídos de [4].

En primer lugar construiremos algunos fractales clásicos mediante la teoría de los *sistemas de funciones iteradas*. Definimos el siguiente conjunto:

$$\mathcal{K}(\mathbb{R}^n) = \{K \subset \mathbb{R}^n, K \neq \emptyset, K \text{ compacto}\}.$$

Se define

$$\mathrm{d}_H(A,B) = \max\{\max_{a\in A} \mathrm{d}(a,B), \max_{b\in B} \mathrm{d}(b,A)\}.$$

Se puede probar fácilmente que esta aplicación es una métrica, la llamada *métrica de Hausdorff*, y no es tan sencillo probar que $(\mathcal{K}(\mathbb{R}^n), \mathrm{d}_H)$ es completo. Por otra parte, dadas

$$f_i : \mathbb{R}^n \to \mathbb{R}^n, \qquad i = 1, \cdots, n,$$

aplicaciones contractivas de razón $r < 1$, entonces

$$F : \mathcal{K}(\mathbb{R}^n) \to \mathcal{K}(\mathbb{R}^n), \qquad F(K) = \bigcup_{i=1}^{n} f_i(K)$$

es una aplicación contractiva de razón $r < 1$. Por el teorema del punto fijo, la sucesión $(K_n)_{n=1}^{\infty}$ en el espacio $\mathcal{K}(\mathbb{R}^n)$ dada por $K_{n+1} = F(K_n)$ converge hacia un compacto K_F no vacío sin importar el valor inicial de la sucesión. Además se tiene $F(K_F) = K_F$; es decir

$$K_F = \bigcup_{i=1}^{n} f_i(K_F).$$

A $(f_i)_{i=1}^{n}$ le llamaremos en lo sucesivo *sistema de funciones iteradas* y al compacto K_F *atractor* para el sistema de funciones o *fractal autosemejante*.

Utilicemos MATLAB para generar gráficamente fractales de esta manera: Los compactos en $\mathbb{R}^2$ los vamos a discretizar usando matrices cuadradas de ceros y unos. Si k es

el tamaño de una matriz T, querrá decir que el compacto estará incluido en $[0,k]\times[0,k]$, y si $T(i,j)=1$, quiere decir que el punto (i,j) está en dicho compacto y $T(i,j)=0$ en otro caso. El programa siguiente usa la función `sfi.m` que es dónde está implementado el sistema de funciones iteradas:

```
function fractal(T,n)
% Construye fractales mediante un SFI.
% T una matriz cuadrada que representa (0s y 1s) el termino inicial
% n el numero de iterados. (se recomienda n<10)
% Si se quiere modificar el fractal hay que modificar el SFI.
colormap(prism)
k=length(T);
M=zeros(k);
for i=1:n
 titulo=strcat('Iteracion:',num2str(i));
 pcolor(T')
 shading('flat')
 axis('equal')
 title(titulo)
 M=sfi(T);
 T=M;
 pause
end
```

A continuación implementaremos el sistema de funciones iteradas. Como ejemplo, implementaremos el siguiente:

$$f_1(x,y)=\frac{1}{2}(x,y),\quad f_2(x,y)=\frac{1}{2}(x,y)+(\frac{k}{2},0),\quad f_3(x,y)=\frac{1}{2}(x,y)+(\frac{k}{4},\frac{k}{2}).$$

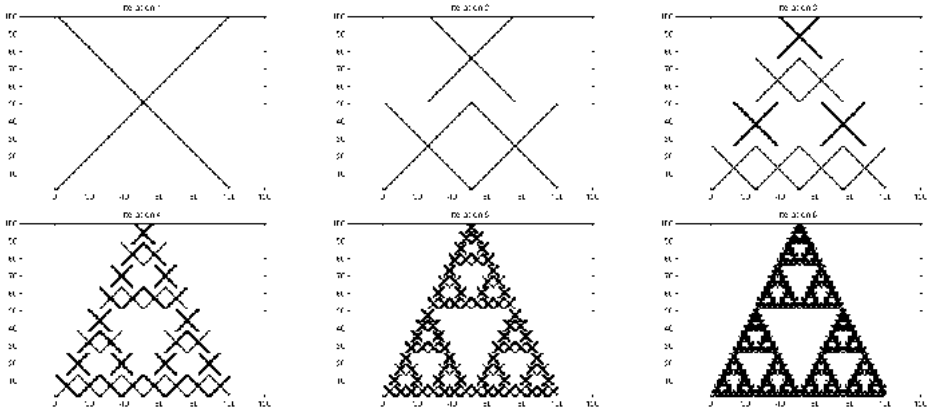
Es evidente que f_i son aplicaciones contractivas de razón $1/2$, y que $f_i(K)\subset[0,k]^2$ para culquier compacto $K\subset[0,k]^2$. Obsérvese el uso de `ceil` para asegurar que los índices de la matriz sean enteros entre 0 y k.

```
function y=sfi(T)
% Implementa un SFI
k=length(T);
y=zeros(k);
for f=1:k
   for c=1:k
      if T(f,c)~=0
         i1=ceil(f/2);
         i2=ceil(c/2);
```

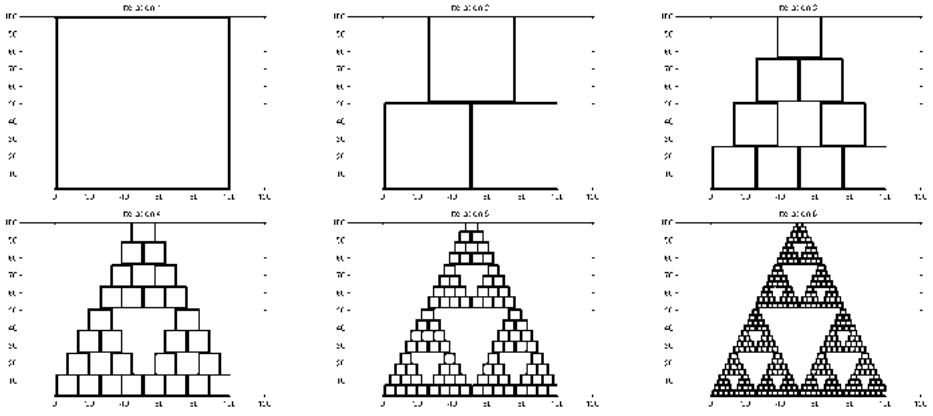
```
            y(i1,i2)=1;
            i1=ceil(f/2+k/2);
            i2=ceil(c/2);
            y(i1,i2)=1;
            i1=ceil(f/2+k/4);
            i2=ceil(c/2+k/2);
            y(i1,i2)=1;
        end
    end
end
```

Se muestra varias gráficas para mostrar, como partiendo de algunos compactos, la sucesión converge siempre al *triángulo de Sierpinski*. Se ejecuta `fractal(T,6)`.

```
T=eye(100)+flipud(eye(100));
```



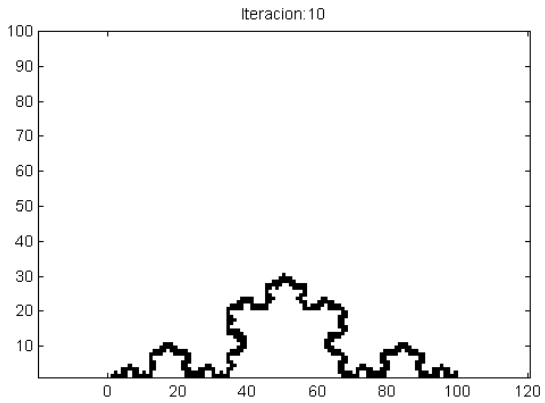
```
a=zeros(100); u=ones(1,100); a(1,:)=u; a(99,:)=u; a(:,1)=u'; a(:,99)=u';
```



Se puede implementar fácilmente el siguiente sistema fundamental de funciones:

$$f_1(v) = \frac{1}{3}v, \quad f_2(v) = \frac{1}{3}G_{\pi/3}(v) + u_2, \quad f_3(v) = \frac{1}{3}G_{-\pi/3}(v) + u_3, \quad f_4(v) = \frac{1}{3}v + u_4,$$

donde G_α indica el giro de ángulo α y $u_2 = (k/3, 0)$, $u_3 = (k/2, k\sqrt{3}/6)$, $u_4 = (2k/3, 0)$. Que como se verá, permite dibujar la *curva de Koch*:



Ahora representaremos el *conjunto de Mandelbrot*. Sea $c \in \mathbb{C}$, y sea $f_c : \mathbb{C} \to \mathbb{C}$ dada por $f_c(z) = z^2 + c$. Consideremos la sucesión $S(c) = (0, f_c(0), f_c^2(0), \cdots)$, es decir la órbita de 0 para el sistema dinámico generado por f_c.

Se puede definir el conjunto de Mandelbrot como

$$\mathcal{M} = \{c \in \mathbb{C} : S(c) \text{ no diverge a } \infty\}.$$

Se puede demostrar que $S(c)$ diverge a ∞ si y sólamente si algún punto de la órbita tien módulo mayor o igual que 2. Este resultado nos permite representar el conjunto de Mandelbrot. Simplemente se calculan los 100 primeros términos de la órbita. El Pseudocódigo es el siguiente:

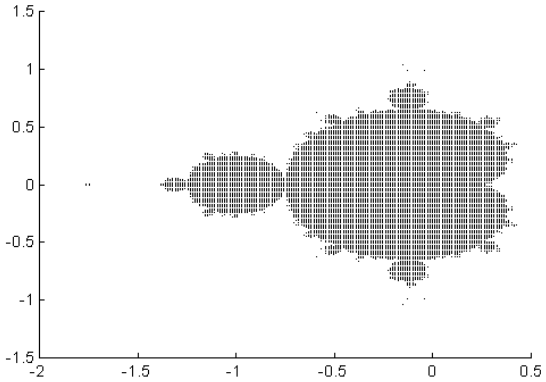
Elegir rectángulo del plano complejo
$z = 0$, $j = 0$.
Tomar $c \in \mathbb{C}$ puntos de este rectángulo.
Mientras $j < 100$ y $|z| < 2$ hacer
    $z_1 = z^2 + c$, $z = z_1$, $j = j + 1$.
Si $|z| < 2$, dibujar c.

El programa (aprovechando que MATLAB posee aritmética compleja) es

```
function mandel(a,b,c,d)
% Representa el conjunto de Mandelbrot en [a,b]x[c,d].
hold on
vreal=linspace(a,b);
vimag=linspace(c,d);
for h=1:100
   for k=1:100
```

```
      z=0;
      j=0;
      c=vreal(h)+i*vimag(k);
      while j<100 & abs(z)<2
         z1=z^2+c;
         z=z1;
         j=j+1;
      end
      if abs(z)<2
         plot(vreal(h)+i*vimag(k))
      end
   end
end
```

La siguiente figura se ha obtenido ejecutando `mandel(-2,0.5,-1.25,1.25)`.



Bibliografía usada.

[1] Duane Hanselman, Bruce Littlefield. The MathWorks Inc. *The student edition of* MATLAB *: Version 5: User's guide*. Prentice Hall. 1997.

[2] The MathWorks Inc. *The student edition of* MATLAB *5.3: for Windows 95/N*. Prentice Hall. 1999.

[3] José Luis Hueso. *Matemática Aplicada; Prácticas con* MATLAB . Universidad Politécnica de Valencia. 1999.

[4] Miguel de Guzmán, Miguel Ángel Martín, Manuel Morán, Miguel Reyes. *Estructuras fractales y sus aplicaciones.* Editorial Labor. 1993.