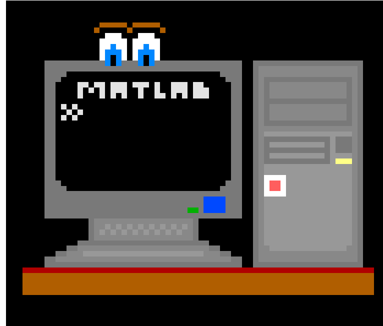
MATLAB como ayuda al estudiante de Ciencias Matemáticas

Mª Dolores Cárdenas Luque
lornacl@iname.com



Copyright © 2.000, por Mª Dolores Cárdenas Luque

MATLAB como ayuda al estudiante de Ciencias Matemáticas puede distribuirse libremente (fotocopiado, impreso o en un archivo PS) de acuerdo a las siguientes restricciones:

1. El documento no puede distribuirse incompleto.

2. El documento completo, o parte de él, ni puede modificarse sin el consentimiento escrito de la autora ni puede usarse con fines lucrativos.

3. Cualquier sugerencia al mismo deberá ser comunicada a la autora, quien se reserva el derecho de llevarlas a cabo.

Índice General

1 Introducción **1**

2 Empezamos: línea de comandos, ¿qué es eso? **3**

3 Mi primer ejemplo (y no es un "Hola Mundo"). Primeros pasitos **5**
3.1 Variables reservadas 8
3.2 Notas sobre la precisión 9
3.3 Notas sobre cadenas 10
3.4 Cositas sobre funciones 11
3.5 MATLAB sabe operar con números complejos 12
3.6 ¿Cómo puedo trabajar con polinomios? 13

4 Sí, ya hemos llegado a las matrices **19**

5 Porque todos llevamos dentro un artista... **25**
5.1 Si eres un artista 2D, esta es tu sección 25
5.2 Y si eres un artista 3D, este es tu lugar 31
5.2.1 Las curvas también tienen su encanto 31
5.2.2 Aunque hay quien prefiere las superficies 32

6 Cuando las funciones (y funcionalidades) básicas se quedan cortas, aprende a hacerlas tú mismo **37**
6.1 Un primer vistazo a los ficheros m. Declaración de funciones 37
6.2 Programación con MATLAB 41
6.2.1 Bucles 41
6.2.2 Condicionales 42

7 Unas palabras para finalizar: Ejemplos **47**
7.1 Primer ejemplo: Un problema de álgebra lineal; la descomposición LU . . . 47
7.2 Segundo ejemplo: Integración definida 49
7.3 Ejemplo final: El atractor saltarín 51

Capítulo 1

Introducción

Cuando un estudiante llega a primero de Matemáticas, usualmente éste lo hace sin conocimientos sobre cómo manejar un ordenador. Por tanto, no sabe que hay programas con los que puede trabajar usándolos de herramientas para sus cálculos. Así, el estudiante se concentra demasiado en hacer las operaciones impecables quitando parte de su atención en entender lo que está haciendo. En las primeras semanas de su peregrinación por la licenciatura (o incluso mejor antes, en algún curso de verano preparatorio) sería recomendable presentar al alumno herramientas de cálculo que le faciliten su labor de cara a saber si está haciendo bien las cosas, enfocando el uso de estas herramientas, primero, a mostrarle cómo hacer con ellas todo lo que en el instituto se hacía a mano, segundo, a mostrarle cómo emplearlas en tareas ya más propias de lo que vea en la licenciatura. Es tan necesario enseñar al alumno el manejo de estas herramientas para poder facilitar el desarrollo de su trabajo, como necesario resulta (y nadie lo discute) que sepa leer y escribir.

Por ello, en esta pequeña memoria voy a realizar un recorrido de las operaciones más usuales que hace un estudiante en su viaje por el bachillerato (antes "unificado y polivalente", ahora, Dios dispondrá) utilizando para ello la herramienta MATLAB.

MATLAB viene de "MATrix LABoratory"; se trata de un programa especializado en los cálculos con matrices que puede llegar a ser muy útil en cuanto se sabe cómo trabajar con él, pues nos permite, con poco esfuerzo, desde hacer los cálculos rutinarios con matrices, a escribir pequeños programas para hacer tareas más complejas de forma muy sencilla.

Dado que los escalares son matrices 1×1, MATLAB también trabaja con ellos sin ningún problema. Podemos hacer con matrices cualquier operación de la que nos enseñaron: sumar, restar, multiplicar, potencias, ... simplemente hemos de tener en cuenta que las matrices tendrán que ser de las dimensiones adecuadas, aunque en caso de no ser así, el programa nos lo avisará con un mensaje de error.

Dejamos pues de lado la introducción para meternos de lleno en lo que queremos que los estudiantes aprendan: el manejo de MATLAB como herramienta para su vida cotidiana matemática.

Capítulo 2

Empezamos: línea de comandos, ¿qué es eso?

Supongamos que estamos con un ordenador que tiene MATLAB instalado (recuerda que MATLAB es un programa por el que hay que pagar, si tu MATLAB es gratuito entonces es que tienes un OCTAVE, lo que, para el caso, viene a ser lo mismo, y de paso fomentas el uso de Linux). Es más, supongamos que ya has escrito `matlab`, con lo que has ejecutado el programa, y de pronto te ha salido una pantalla negra más plana que plana, en modo texto, con dos líneas en la parte superior en las que te invita a pedir ayuda si la necesitas usando el comando `help`. A lo mejor estamos suponiendo demasiado al nombrar la palabra *comando* y resulta que no sabes lo que es. A *"grosso modo"*, debes saber que un ordenador es una máquina tonta que no hace más que lo que tú le digas. Si le dices `matlab`, pues él va y ejecuta MATLAB, pero no esperes que haga nada más a partir de ese momento. Al entrar en MATLAB verás una línea en la que pone

```
>>
```

eso significa que MATLAB está esperando que le introduzcas un comando. Un comando es una *orden* que le das al programa, así, si le dices

```
>> pintame la casa
```

observarás que MATLAB se enfada y te da un mensaje de error: no ha sido diseñado para tales menesteres (lo siento, créeme que a mí también me gustaría), y además, te da un mensaje de error para avisarte que no puede hacer lo que le dices. Si no puede hacerlo, será por uno de los dos motivos siguientes:

1. Has introducido un comando que no forma parte de los que él es capaz de entender.
2. Has cometido algún error al introducir un comando, una matriz u otro elemento. Debes tener presente que no es lo mismo `eye(5)` que `eey(5)`, si tú te equivocas

> escribiendo lo que no puedes esperar es tener a MATLAB detrás de ti adivinando tus intenciones. De aquí aprenderás una importante lección: responsabilízate de lo que escribes u obtendrás resultados no deseados. Y cuando el trabajo a realizar es largo y complicado, un resultado no deseado puede terminar destrozando todo tu trabajo. Eso enseña otra valiosa lección: confía en la máquina, pero nunca al 100% (recuerda que han sido diseñadas por el hombre).

Ya sabemos qué es un comando, ya sabemos qué es la línea de comandos (por si acaso, la línea que comienza con `>>` a la espera de que introduzcas, pues qué va a ser, comandos). Hay que hacer notar una cosa: MATLAB es uno de esos programas que se llama *case-sensitive*, eso, en cristiano, significa que **distingue** una mayúscula de una minúscula. No es porque tengamos especiales manías a la hora de escribir, pero si él tiene un comando que se llama `help`, significa que si tú le pones `HeLp` no va a entenderlo y, en consecuencia, te dará un error.

Perfecto. Ya ha quedado todo claro, y ya hemos hecho notar que MATLAB no tiene sentido del humor, así que no le escribas en mayúsculas lo que para él debe ir en minúsculas. Ahora, lo que hay que saber es qué comandos se pueden usar con MATLAB para poder trabajar.

Capítulo 3

Mi primer ejemplo (y no es un "Hola Mundo"). Primeros pasitos

Lo primero que se le ocurre hacer a uno en cuanto le ponen un programa de matemáticas delante es (no se sabe por qué) escribir la siguiente línea:

```
>> 2+2
```

Pruébalo. Comprobarás que obtienes por respuesta:

```
ans =
 4
```

Es decir, con MATLAB, realizar las operaciones ordinarias con números son como uno se espera que sean: escribes números, pones operaciones entre ellos, y al pulsar `RETURN` obtienes el resultado de esa operación. MATLAB no fue diseñado para usarlo como calculadora, pero siempre es una forma de empezar a aprender a usarlo. Así, si ahora queremos realizar la operación

$$2 + 3^2 - \frac{4 \times 5^2}{3}$$

lo introducimos de la misma forma en MATLAB

```
>> 2+3^2-(4*5^2)/3
```

y él nos da su respuesta:

```
ans =
 -22.3333
```

lo que nos muestra, además, que MATLAB no trabaja con precisión exacta, sino que lo hace usando precisión aproximada. Si en vez de elevar al cuadrado queremos hacer una raíz cúbica, efectivamente, se tratará de escribir:

```
>> 8^(1/3)
```

y MATLAB nos responde, como era de esperar

```
ans =
 2
```

Además, en vez de hacer operaciones con números directamente, podemos asignar esos valores a unas variables y luego trabajar con dichas variables. Por ejemplo, haz lo siguiente:

```
>> pepe=34
>> juan=42
>> luisa=-27
>> pepe*juan+luisa
```

¿Qué sale? ¿1401? ¿Y si haces directamente `34*42+(-27)`? ¿Sale lo mismo? Entonces lo has hecho bien. Comprobarás que cuando aprietas la tecla `RETURN` de tu teclado, MATLAB repite en pantalla lo que tú le has escrito. Puedes evitar que te muestre por pantalla lo que le acabas de introducir poniendo ; al final del comando que estés escribiendo. Por ejemplo, si escribimos

```
>> pepe=40;
```

ya no nos vuelve a salir, inmediatamente después

```
>> pepe
40
```

Podemos, incluso, asignar a una variable el resultado de una operación realizada a partir de otras variables, con el fin de conservar este resultado. Siguiendo con el ejemplo anterior, podemos escribir

```
>> luis=pepe*juan+luisa
```

y así podremos usar este resultado repetidas veces simplemente escribiendo el nombre de la variable, sin necesidad de escribir todas las operaciones cada vez.

Ahora no verás mucha utilidad a las variables, pero un poco más adelante, cuando ya describamos cómo introducir matrices, sí que se ve la utilidad, pues no es lo mismo escribir

A para operar con una cierta matriz que escribirla siempre entera (imaginad que fuera una matriz grande, menuda paliza, ¿no?).

Puede llegar un momento en que queramos eliminar una variable de la memoria; para ello, no tendremos más que hacer lo siguiente:

```
>> clear variable
```

donde `variable` es la variable que queremos eliminar. Por ejemplo, podemos querer eliminar la variable *luis* porque ya no nos vaya a resultar de utilidad; entonces, escribiremos:

```
>> clear luis
```

o podemos, incluso, querer eliminar todas las variables de la memoria, para ello basta con escribir simplemente `clear`, y a partir de ahí MATLAB no recordará nada de lo hecho anteriormente (y si intentamos que lo recuerde nos dará un error).

Ahora que decimos que MATLAB se sumirá en el olvido, ¿no sería mejor que pudiéramos grabar nuestro trabajo en algún fichero para poder tener esos resultados listos para incluir en alguna memoria que tengamos que entregar? Pues sí, podemos, y es tan sencillo como escribir

```
>> diary nombre_fichero
```

donde `nombre_fichero` es el nombre del fichero en el que queremos guardar nuestro resultados. Es un fichero de texto en el que se muestra todo lo que hayamos hecho. Para "activar" este "diario" pondremos

```
>> diary on
```

y para desactivarlo

```
>> diary off
```

Fácil, ¿o no? No obstante, esto tiene un problema y es que, vale, tenemos un fichero de texto en el que están nuestros resultados, pero luego no podemos recuperarlos en MATLAB para volver a usarlos. ¿Cómo grabamos las variables para luego poder volver a cargarlas y usarlas más veces? Para grabarlas, escribimos

```
>> save nombre_fichero
```

y en este fichero MATLAB grabará todas nuestras variables con los valores que tuvieran. Este fichero tendrá la extensión `.mat`. Después, bien tras un `clear`, bien simplemente cuando se necesiten, escribimos

```
>> load nombre_fichero
```

y ya volvemos a tener en memoria las variables que habíamos empleado antes.

Además, siempre podemos preguntar a MATLAB por el nombre de todas las variables que tengamos en memoria, sin más que escribir `who`. Por ejemplo, vamos a introducir dos variables A y B, la primera un vector y la segunda una matriz, para probar esto (aunque explicaremos más adelante cómo introducir con vectores y matrices, esto nos da una primera pista):

```
>> A=[3 4 5];
>> B=[1 2;3,4];
>> who

Your variables are:

A          B
```

MATLAB puede devolvernos más información sobre nuestras variables. Para ello, en lugar de `who`, usaremos el comando `whos`, como se ve a continuación:

```
>> whos
 Name          Size         Elements      Bytes     Density   Complex
   A         1 by 3                3         24        Full       No
   B         2 by 2                4         32        Full       No

Grand total is 7 elements using 56 bytes
```

Estos comandos de manejo genérico del programa puedes consultarlos en la ayuda escribiendo `help general`. Otro de estos comandos que puede resultarnos de utilidad es `clc`. Pruébalo. Simplemente, limpia la pantalla.

3.1 Variables reservadas

MATLAB es un programa de matemáticas, así que en alguna parte debe tener escondido al número π. Concretamente (en un alarde de originalidad), si escribimos

```
>> pi
```

ya sabremos dónde se lo guardaba el muy sinvergüenza. ¿Qué quiere decir esto? Pues que si ahora en un descuido escribimos

```
>> pi=4
```

perderemos el valor de π, pues este será sustituido por 4, el nuevo valor que le hemos asignado. Si queremos recuperar π, tendremos que borrar de la memoria la variable `pi` que hemos introducido previamente. Para ello, sí, hacemos `clear pi`. Sin embargo, si ahora escribimos

```
>> pi
```

en lugar de obtener un error (si eliminamos una variable y después vamos a usarla, MATLAB nos da un error porque esa variable ya no existe para él), obtenemos de nuevo el valor de π. Esto es así porque `pi` es una variable reservada de MATLAB. Podemos redefinirla si queremos, pero es mejor no hacerlo, a no ser que estemos seguros de que no dará lugar a confusión.

¿Podemos encontrarnos con más variables de este tipo? Sí, concretamente tenemos `i`, `j` (ambas se refieren a la unidad imaginaria compleja, $\sqrt{-1}$), `eps` (precisión en operaciones de coma flotante, esto es, el mayor número positivo tal que `1+eps=1`, ¿o es que pensabas que a un ordenador le caben todos los decimales de π?), `Inf` (definido como el resultado de $1/0$) y `NaN`[1] (definido como $0/0$). Al igual que lo dicho para `pi`, podemos redefinirlas con otros valores si sabemos que no nos vamos a equivocar. Lo que no hay que olvidar es que, tanto si hablamos de variables reservadas como de variables normales, el valor que tienen guardado es el **último** que le hayamos dado.

3.2 Notas sobre la precisión

MATLAB trabaja con mucha precisión, sin embargo, al mostrarnos resultados en pantalla, observamos que sólo nos pone cuatro decimales, ¿cómo hacemos para ver más decimales? Pues escribir lo siguiente

```
>> format tipo_formato
```

Algunos de los formatos disponibles son `short` (formato por defecto), `long`, `short e` (formato corto con notación científica), `long e` (igual, pero formato largo), `rational`.

Este último formato puede resultar interesante: nos muestra el resultado en forma de fracción, dicha fracción es la que más se aproxima al resultado decimal de nuestro número. Por ejemplo:

```
>> format rational
>> pi
```

[1]Not a Number: no es un número

```
>> ans =
      355/113
```

pues $355/113 = 3.141592...$

También tenemos el formato `compact`; éste evita que MATLAB muestre líneas vacías entre el nombre de la variable y el resultado, entre la línea que hemos introducido y el resultado, ... Este formato nos es útil por si queremos ver más información en la pantalla de la que cabría si dejamos que MATLAB inserte las líneas en blanco que inserta por defecto.

3.3 Notas sobre cadenas

Se entiende por cadena una estructura de datos cuyos elementos, en lugar de ser números, son letras, por ejemplo, una palabra o una frase (tenga o no sentido). Para poder introducir una cadena en MATLAB, hemos de ponerla entre comillas simples. Por ejemplo

```
>> 'Muestra de cadena en MATLAB'
```

será una cadena para MATLAB, mientras que

```
>> Esto no es una cadena
```

y además MATLAB nos lo ratificará con uno de sus mensajes de error (recordad que no tenía sentido del humor; sigue sin tenerlo). ¿Y para qué nos pueden servir las cadenas? Ahora mismo para nada, pero seguro que cuando aprendamos a hacer dibujos con MATLAB, querremos ponerles un título. O puede que necesitemos mostrar un texto al usuario ofreciéndole opciones en algún programa que hagamos.

Podemos mostrar una cadena de texto en MATLAB con la función `disp('cadena')`, por ejemplo, si hacemos `disp('Me repito como el ajoaceite');`, en cuanto pulsemos `RETURN` MATLAB imprimirá como resultado la línea `Me repito como el ajoaceite`.

Si lo que necesitamos es asignar un valor a una variable tras pedir al usuario que lo introduzca, necesitamos la función `input('cadena')`. Por ejemplo, si escribimos

```
h = input('Introduzca valor del paso de integracion  ');
```

nos aparece en pantalla la frase `Introduzca valor del paso de integracion`, y vemos que MATLAB se está quieto: está esperando que introduzcamos algo y pulsemos la tecla `RETURN`. En cuanto pulsemos `RETURN`, lo que hayamos escrito será asignado a la variable `h`. Por ejemplo, si introducimos `0.01`, entonces tendremos `h=0.01`. Pero si introducimos

`[0 1]`, entonces será `h=[0 1]`. A lo mejor no es eso lo que pretendíamos, pero queda como ejemplo de que podemos introducir lo que queramos, ya que éso será asignado a la variable que hayamos indicado.

3.4 Cositas sobre funciones

Trabajar con MATLAB sería más difícil si no fuera porque lleva incorporadas una gran cantidad de funciones con las que se pueden hacer bastantes cosas. Como ya sabemos, las funciones son máquinas a las que les introducimos un "alimento" con el que trabajar, y que nos devuelven el resultado de trabajar ese alimento, sin necesidad de saber cómo hacen sus tareas. Concretamente, en MATLAB esto se escribe así:

```
variable_resultado=nombre_funcion(variable_entrada)
```

Por ejemplo:

```
Cerito_lindo=sin(0)
```

A lo largo de tus andaduras por el instituto ya conocerás unas cuantas funciones matemáticas que, por supuesto, puedes usar en MATLAB sin complicarte la vida, así que veamos una pequeña lista de ellas y algún ejemplo de cómo usarlas así como del resultado que se obtiene:

Función	Sintaxis	Ejemplo	Resultado
seno	`sin(x)`	`sin(pi/2)`	`1`
coseno	`cos(x)`	`cos(-pi/4)`	`0.7071`
tangente	`tan(x)`	`tan(pi/2)`	`Inf`
secante	`sec(x)`	`sec(2*pi)`	`1`
cosecante	`csc(x)`	`csc(pi/2)`	`1`
cotangente	`cot(x)`	`cot(0)`	`Inf`
arcoseno	`asin(x)`	`asin(1)`	`1.5708`
arcocoseno	`acos(x)`	`acos(0)`	`1.5708`
arcotangente	`atan(x)`	`atan(pi)`	`1.2626`
arcosecante	`asec(x)`	`asec(pi)`	`1.2469`
arcocosecante	`acsc(x)`	`acsc(pi)`	`0.3239`
arcocotangente	`acot(x)`	`acot(0)`	`1.5708`
exponencial	`exp(x)`	`exp(1)`	`2.7183`
logaritmo neperiano	`log(x)`	`log(1)`	`0`
logaritmo decimal	`log10(x)`	`log10(100)`	`2`
logaritmo base 2	`log2(x)`	`log2(8)`	`3`
raíz cuadrada	`sqrt(x)`	`sqrt(16)`	`4`

Estas no son todas las funciones de las que disponemos, pero para empezar ya son unas cuantas. Si consultamos la ayuda (recordemos que MATLAB nos ofrecerá ayuda en cualquier momento escribiendo `help`), podemos escribir `help elfun` (`elfun`: *elementary functions*) para que nos informe de más funciones que tiene disponibles. También nos ofrecerá ayuda sobre otras funciones escribiendo `help specfun` (*special functions*), sin embargo, la mayoría de estas funciones no serán de utilidad al recién llegado a la licenciatura. De todas formas, nunca viene mal saber que están ahí. Para cuando sí se necesiten.

Otras funciones que pueden resultar interesantes son estas:

- `ceil(x)` redondea el número x que le hayamos introducido al entero más cercano hacia más infinito. Es decir, si, por ejemplo, calculamos `ceil(3.8)`, el resultado será `4`, como cabe esperar, pero si calculamos `ceil(3.1)`, el resultado también será `4`.
- `floor(x)` redondea el número x al entero más cercano, pero esta vez hacia menos infinito. Por, ejemplo, si calculamos `floor(3.2)` obtenemos como resultado `3`, y si calculamos `floor(3.9)` también obtenemos como resultado `3`.
- `gcd(x,y)`, `lcm(x,y)` devuelven, respectivamente, el máximo común divisor y el mínimo común múltiplo de los enteros x e y. Hay que remarcar dos cosas, la primera es que la función `lcm` sólo admitirá que los enteros sean *positivos*, y la segunda es que se ha tomado la convención siguiente: `gcd(0,0)=0`.
- `rem(x,y)` nos da el resto de la división de x entre y siendo el cociente entero. Por ejemplo, si calculamos `rem(5.6,1.5)`, nos da como resultado `1.1`. Si dividimos, tenemos que $5.6/1.5 = 3.733$, luego $5.6 - 1.5 \times 3 = 5.6 - 4.5 = 1.1$
- `round(x)` redondea x al entero más cercano. Si ponemos `round(0.5)` obtendremos `1`, mientras que si ponemos `round(0.4999)` obtendremos `0`. Igualmente, si escribimos `round(-0.9)` da como resultado `-1` y si escribimos `round(-1.1)` obtenemos `-1`

Como nota final; cuando trabajamos con funciones que devuelven más de un valor, la forma de escribirlo es la siguiente

```
>> [valor_devuelto1, ..., valor_devueltoN]=funcion(arg1, ..., argM)
```

Más adelante, cuando nos encontremos con funciones que devuelven más de un valor, veremos más claramente cómo se usan.

3.5 MATLAB sabe operar con números complejos

Ya sabemos hacer bastantes cosas con los números "normales y corrientes", pero es posible que muchos de los recién llegados hayan tenido la oportunidad de conocer los números complejos. ¿Cómo se introducen y cómo se opera con ellos en MATLAB? Pues exactamente de la misma forma que lo haríamos con los números reales. Por ejemplo:

```
>> 1+3i
```

sin necesidad de poner * entre el 3 e i (cosa que no pasa con pi; hay que escribir 2*pi si se quiere 2π y no un error). O si queremos sumar/restar/multiplicar/dividir números complejos:

```
>> (1-i)+(2+4i)
 ans =
    3.0000 + 3.0000i
>> (1-i)-(2+4i)
 ans =
    -1.0000 - 5.0000i
>> (1-i)*(2+4i)
 ans =
    6.0000 + 2.0000i
>> (1-i)/(2+4i)
 ans =
    -0.1000 - 0.3000i
```

Las funciones que hemos descrito brevemente en la tabla de la sección anterior (recordar: sección 3.4, *"Cositas sobre funciones"*) también pueden operar con números complejos. Además de éstas, tenemos las funciones conocidas que se usan en cuanto nos enseñan un poco de variable compleja:

Función	Sintaxis	Ejemplo	Resultado
Parte real	real(x)	real(70+i)	70
Parte imaginaria	imag(x)	imag(1-7i)	-7
Conjugado	conj(x)	conj(1-i)	1+i
Módulo	abs(x)	abs(3-4i)	5
Argumento	angle(x)	angle(1-i)	-0.7854

Y, bueno, no es que MATLAB sepa operar con números complejos si se lo pides, es que siempre lo hace así, sólo que hasta ahora no lo sabías.

3.6 ¿Cómo puedo trabajar con polinomios?

MATLAB utiliza vectores para operar con polinomios, así que será mejor que sepamos primero cómo trabajar con vectores para después ver cómo usar los polinomios.

Como sabeis, un vector de n componentes lo representamos así, $(x_1, \ldots, x_n)$. Pues en MATLAB, la única diferencia que hay para introducir un vector es que, en lugar de con paréntesis, delimitamos las componentes entre corchetes y separadas por espacios o por comas. Así, por ejemplo

```
>> [0 1 4 6 -4]
```

denota al vector en $\mathbb{R}^5$, $(0, 1, 4, 6, -4)$. Si lo que queremos es que el vector esté en columna, podemos hacerlo de dos formas, bien separando cada componente por punto y coma, bien escribiendo cada componente en una línea, como vemos en los ejemplos siguientes:

```
>> X = [1;2;3];
>> Y = [1
        2
        3];
```

representan, respectivamente, a los vectores $X = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$ e $Y = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$ de $\mathbb{R}^3$ (obviamente, son el mismo vector).

Más adelante, a lo largo de la carrera, sobre todo en asignaturas de análisis numérico, necesitarás vectores cuyas componentes son unos ciertos nodos donde tendrás que evaluar unas ciertas funciones. Lo más común, además, será que estos nodos sean equiespaciados. Por ejemplo, puedes necesitar una partición del intervalo $[-1, 5]$ cuyos puntos estén separados entre sí 0.1 (es decir, es una partición de $\frac{5-(-1)}{0.1} = 60$ puntos, sin contar uno de los extremos). Esto lo puedes almacenar en un vector de la siguiente forma:

```
>> X = -1:0.1:5;
```

Si no le pones el punto y coma al final, verás aparecer ante ti todos los puntos de la partición que forman las componentes del vector X. A partir de ahora tienes un vector de 61 componentes con todos los nodos que necesitas. En general, lo que se hace es esto:

```
>> nombre_vector = x_inicial : distancia_entre_nodos : x_final
```

Autoexplicativo, ¿no?

Otra función de que disponemos para crear vectores es la siguiente:

```
linspace(x_inicial,x_final,cuantos)
```

que crea un vector de `cuantos` términos en progresión aritmética, desde `x_inicial` hasta `x_final`. Por ejemplo, si escribimos

```
>> linspace(3,9,12)
```

MATLAB nos responde lo siguiente

```
ans =
Columns 1 through 7
3.0000    3.5455    4.0909    4.6364    5.1818    5.7273    6.2727
Columns 8 through 12
6.8182    7.3636    7.9091    8.4545    9.0000
```

Si no especificamos `cuantos`, `linspace` creará por defecto un vector con 100 componentes equiespaciadas entre `x_inicial` y `x_final`

¿Qué operaciones recuerdas que podías hacer con vectores? Por ejemplo, sumarlos, restarlos o multiplicarlos por un número, ¿no es así? Pues estas operaciones son tan intuitivas de llevar a cabo en MATLAB como sigue:

```
>> X = [1 2 3];
>> Y = [-1 -2 -3];
>> X+Y
ans =
   0     0     0
>> X-Y
ans =
   2     4     6
>> 3*X
ans =
   3     6     9
```

También sabíamos calcular la norma euclídea de un vector (recuerda que si $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, entonces $||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$ era su norma euclídea). Esto con MATLAB es tan sencillo como lo siguiente:

```
>> X = [1 2 3];
>> norm(X)
ans =
   3.7417
```

Verás más adelante (en la carrera) que se pueden definir otras normas de vectores, las llamadas *normas "p"*, definidas como sigue, para $1 \leq p \leq +\infty$:

$$||x||_p = \begin{cases} \left(\sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}} & p \neq +\infty \\ \max\limits_{i=1,\ldots,n} \{x_i\} & p = +\infty \end{cases}$$

Estas normas puedes calcularlas muy fácilmente de la forma siguiente:

```
>> norm(vector,p)        % En el caso p distinto de infinito
>> norm(vector,Inf)      % En el caso p igual a infinito
```

por ejemplo, `norm(X,4)` da como resultado `3.1463` y `norm(X,Inf)` da como resultado `3`.

Podemos querer multiplicar dos vectores componente a componente. Si tratamos de escribir en MATLAB `X*Y`, nos dará un error, puesto que intentará multiplicarlo como si fueran matrices, y no podrá al no coincidir las dimensiones (¿que por qué lo intenta?, pues porque, como se dijo muy al principio, para MATLAB **todo** son matrices). Para poder operar componente a componente tendremos que usar la notación *punto*. Veamos un ejemplo. Si queremos multiplicar componente a componente los vectores X e Y introducidos más arriba, no tendremos más que postponer un punto a X antes de poner el símbolo de la multiplicación, así:

```
>> X.*Y
```

y entonces obtenemos

```
ans =
   -1    -4    -9
```

Sirve lo mismo para las operaciones `/` (haríamos el cociente componente a componente, por ejemplo con `X./Y` obtendríamos `-1 -1 -1`) y `^` (potencia de cada componente de un vector, por ejemplo, `X.^5` da como resultado `1 32 243`, precisamente 1^5, 2^5, 3^5).

Vamos a finalizar este recorrido por el mundo de los vectores con una tabla en la que se recogen algunas de las funciones más útiles para trabajar con ellos en MATLAB. Después de esta tabla, veremos cómo usar polinomios (y, con ello, nos daremos cuenta de por qué era necesario saber esto primero).

Función	Sintaxis	Ejemplo	Resultado
Transposición	`X'`	`[1 2 3]'`	`[1;2;3]`
Longitud del vector	`length(X)`	`length([1 2 5])`	`3`
Suma de componentes	`sum(X)`	`sum([1 2 3])`	`6` $(= 1 + 2 + 3)$
Producto de componentes	`prod(X)`	`prod([3 4])`	`12` $(= 3 \cdot 4)$
Producto escalar	`dot(X,Y)`	`dot([1 2],[3 4])`	`11` $(= 1 \cdot 3 + 2 \cdot 4)$
Producto vectorial	`cross(X,Y)`	`cross([1 2 3],[1 0 1])`	`2 2 -2`

Con estas funciones podemos hacer cosas más complicadas, como por ejemplo, calcular la suma de los cuadrados de los cien primeros números naturales. Una forma de hacerlo es esta:

```
>> X = linspace(1,100);
>> Y = X.^2;
>> sum(Y)
```

Primero creamos el vector X con términos en progresión aritmética desde 1 hasta 100, siendo el vector de 100 componentes (el tercer valor es por defecto 100 para la función `linspace`, por eso no lo ponemos), así que las 100 componentes son los 100 primeros números naturales. Después calculamos el vector Y como el producto componente a componente de X por sí mismo, esto nos da el cuadrado de sus componentes. Finalmente, con la función `sum` sumamos todos estos cuadrados. No nos hacía falta el vector Y intermedio, también podíamos haber hecho

```
>> X = linspace(1,100);
>> sum(X.^2)
```

e, incluso, más directamente

```
>> sum((1:100).^2)
```

pero ya queda a la elección personal cómo de legible se desea que quede el trabajo realizado.

Y ahora ya, **polinomios**. Para introducir un polinomio en MATLAB, hemos de introducir un vector cuyas componentes son los coeficientes que multiplican a las indeterminadas, ordenados de mayor grado a menor grado. Por ejemplo, si queremos usar el polinomio $P(x) = 3x^5 - x^3 + x^2 - 1$, tendremos que introducir el vector `P=[3 0 -1 1 0 -1]` que, a partir de ahora, será para nosotros el polinomio $P(x)$.

Este polinomio podemos evaluarlo en un punto, por ejemplo, podemos calcular $P(7)$, escribiendo `polyval(P,7)`, con lo que obtenemos `50126`, así que ya sabemos que $P(7) = 50126$. También podemos querer calcular sus raíces, cosa que haremos escribiendo `roots(P)`, y a la que MATLAB nos responde con

```
ans =
   0.3493 + 0.7412i
   0.3493 - 0.7412i
  -0.7393 + 0.3001i
  -0.7393 - 0.3001i
   0.7800
```

Como ya sabemos cómo introducir polinomios, y cómo sumar/restar vectores, ya sabemos cómo sumar/restar polinomios, pero, ¿y cómo los multiplicamos?. Pues con la función `conv(P,Q)`, siendo P, Q los vectores con los coeficientes de dos polinomios $P(x)$, $Q(x)$. Por ejemplo, si queremos multiplicar los polinomios $P(x) = x^3 + 3x - 1$, $Q(x) = 2x^2 - x + 4$ haremos:

```
>> P = [ 1 0 3 -1 ];
>> Q = [ 2 -1 4 ];
>> conv(P,Q)
ans =
   2    -1    10    -5    13    -4
```

lo que nos dice que $P(x)\cdot Q(x) = 2x^5 - x^4 + 10x^3 - 5x^2 + 13x - 4$. Ahora, si queremos dividirlos usaremos la función `[Cociente,Resto]=deconv(P,Q)`, donde `Cociente` y `Resto` son dos vectores en los que se almacenan los coeficientes del cociente y el resto, respectivamente, de la división euclídea de $P(x)$ entre $Q(x)$. Si escribimos `[C,R]=deconv(P,Q)`, obtenemos

```
C =
 0.5000    0.2500
R =
 0          0    1.2500   -2.0000
```

es decir, el cociente de $P(x)/Q(x)$ es $0.5x + 0.25$, y el resto es $1.25x - 2$. Esto lo podemos comprobar haciendo la *prueba* de la división: si ahora calculamos `conv(Q,C)+R`, obtenemos:

```
ans =
   1     0     3    -1
```

que representa al vector $x^3 + 3x - 1$, justo el que habíamos dicho antes que era $P(x)$.

MATLAB no está pensado para el análisis matemático, sin embargo, con los polinomios estamos de suerte, ya que implementa una función que deriva un polinomio, esta es la función `polyder(P)`, que nos devuelve un vector con los coeficientes del polinomio derivada del que nosotros le introduzcamos como argumento. Por ejemplo, con nuestro polinomio $P(x)$ de antes, si escribimos `polyder(P)`, obtenemos el vector `3 0 3`, que corresponde al polinomio $3x^2 + 3$, y que es justo la derivada de $P(x) = x^3 + 3x - 1$.

Una última función que vamos a comentar en este apartado es la función `poly(P)`. Lo que hace esta función es construir un polinomio (con coeficiente director la unidad) a partir de sus raíces, que nosotros le pasamos como argumento en el vector `P`. Por ejemplo, si hacemos

```
>> P = [ 1 -1 ];
>> poly(P)
ans =
   1     0    -1
```

la respuesta corresponde al polinomio $x^2 - 1 = (x + 1) \cdot (x - 1)$.

Capítulo 4

Sí, ya hemos llegado a las matrices

Recordemos que, dicho a lo bruto, una matriz es una colección de elementos dispuestos por filas (o por columnas) de la misma longitud. Es decir:

$$\begin{pmatrix} 4 & -3 & 7 \\ -1 & 0 & -4 \\ 2 & 3 & \pi \end{pmatrix}$$

se considera una matriz, mientras que:

$$\begin{pmatrix} 4 & -3 & \\ -1 & 0 & -4 \\ 2 & & \end{pmatrix}$$

no se considera una matriz, ya que hay filas (o columnas, lo que prefieras) con distinto número de elementos.

El siguiente paso lógico es preguntarse cómo introducir una matriz en MATLAB. Es más, es lógico incluso preguntarse si no será de forma parecida a como se introducen vectores. Pues mira, sí que es lógico planteárselo y, lo que es mejor, obtener una respuesta afirmativa. Para ello, tenemos dos posibilidades. La primera es introducirla "de seguida" separando las distintas filas por punto y coma, de la forma siguiente:

```
>> A = [ 1 2 3 ; 0 2 -3 ; 0 0 1 ];
```

simplemente teniendo en cuenta que los elementos hay que separarlos por espacios. También podemos introducir la matriz escribiendo cada fila por separado, de la siguiente manera:

```
>> A = [ 1 2  3
         0 2 -3
         0 0  1 ];
```

No hay que ser riguroso con la forma en que lo escribes; lo hemos puesto así por claridad, pero vale igualmente si lo quieres poner así:

```
>> A = [ 1 2 3
0 2 -3
0 0 1 ];
```

Lo que importa es que separes las filas, bien entre líneas, o bien con un punto y coma, como en el primer ejemplo. Date cuenta de que hemos puesto el punto y coma tras la definición de la matriz, esto era para evitar que volviera a salirnos por pantalla tal como la introducíamos.

Para referirnos a un elemento concreto de la matriz, escribiremos `A(i,j)`, donde i se refiere a la fila y j a la columna en la que se encuentra el elemento. En la matriz que hemos introducido antes, si escribimos `A(1,3)` obtenemos `3`, que es el elemento que se encuentra en la fila 1, columna 3. Si queremos referirnos a toda una fila, escribiremos `A(i,:)`, por ejemplo, con `A(1,:)` obtenemos `1 2 3`, que es la primera fila de la matriz. Si queremos referirnos a toda una columna, escribimos `A(:,j)`, por ejemplo, con `A(:,3)` MATLAB nos dice

```
ans =
     3
    -3
     1
```

que es la tercera columna de la matriz.

Podemos sumar, restar o multiplicar matrices sin más que interponer entre ellas el símbolo correspondiente a la operación (`+`, `-`, `*`), siempre teniendo en cuenta que las dimensiones sean adecuadas, de lo contrario, MATLAB dará un error. Igualmente, podemos multiplicar la componente (i,j) de una matriz A por la componente (i,j) de otra matriz B usando la notación `.` (punto). Por ejemplo, introducimos las matrices y hacemos la operación

```
>> A=[1 2 3;0 2 -3;0 0 1];
>> B=[4 5 6;1 2 3;0 0 -1];
>> A.*B
```

¿Qué esperas que salga como resultado? ¿Esperas que MATLAB haga lo siguiente?

$$A. \cdot B = \begin{pmatrix} 1\cdot 4 & 2\cdot 5 & 3\cdot 6 \\ 0\cdot 1 & 2\cdot 2 & (-3)\cdot 3 \\ 0\cdot 0 & 0\cdot 0 & 1\cdot(-1) \end{pmatrix}$$

Pues acertaste, mira la respuesta de MATLAB:

```
ans =
     4    10    18
     0     4    -9
     0     0    -1
```

En MATLAB tenemos, además, una forma muy sencilla de definir algunas matrices que son bastante usadas, mediante unas funciones que a continuación detallamos:

- `eye(n)` crea una matriz identidad de tamaño $n \times n$ (recuerda que la matriz identidad es una matriz cuadrada cuyos elementos son todo ceros salvo los de la diagonal principal, que son todo unos). También podemos usar `eye(m,n)` para crear una matriz no necesariamente cuadrada (de tamaño $m \times n$) con unos en la diagonal principal y ceros en el resto de posiciones.
- `ones(n)` crea una matriz cuadrada $n \times n$ cuyas entradas son todos unos. También podemos usar `ones(m,n)` para crear una matriz de tamaño $m \times n$ con unos en todas las entradas.
- `zeros(n)` y `zeros(m,n)` crean, respectivamente, una matriz nula de tamaño $n \times n$ y de tamaño $m \times n$ (recuerda que una matriz nula es aquella que tiene ceros en todas sus entradas).
- `rand(n)` y `rand(m,n)` crean matrices cuyas entradas son aleatorias. Estas entradas son elegidas de una distribución uniforme en el intervalo $[0, 1]$.

No son éstas las únicas funciones de las que disponemos, pero sí nos serán suficientes por el momento. De todas formas, para más información puedes escribir `help elmat`, MATLAB te mostrará (como es su costumbre) la lista de funciones sobre las que le puedes preguntar.

Podemos crear matrices de más formas. Una de estas formas es a partir de otras matrices, es decir, construir una matriz a base de bloques (estos bloques son,a su vez, matrices). Vamos a ilustrar esto con un ejemplo. Definimos las matrices

```
>> A=[1 2 3;0 7 6;3 5 6];
>> B=[-1 4 6;0 0 1;1 1 0];
```

Si ahora queremos construir una matriz en la que aparezcan A seguida de B, es decir, en la que cada fila sea la correspondiente fila de A seguida de la correspondiente fila de B, escribimos `M=[A B]`, con lo que obtenemos

```
M =
      1     2     3    -1     4     6
      0     7     6     0     0     1
      3     5     6     1     1     0
```

o podemos escribir `[A B;B A]`; como sabemos, el punto y coma delimita las filas, así que el resultado es el siguiente:

```
M =
      1     2     3    -1     4     6
      0     7     6     0     0     1
      3     5     6     1     1     0
     -1     4     6     1     2     3
      0     0     1     0     7     6
      1     1     0     3     5     6
```

También podemos crear matrices extrayendo submatrices de una mayor y asignándolo a alguna variable. Para ello, debemos indicar las filas y columnas que vamos a extraer. Por ejemplo, de nuestra matriz M podemos extraer las filas 2, 3 y las columnas 4, 5, haciendo lo siguiente:

```
>> M_2345=M(2:3,4:5)
M_2345 =
      0     0
      1     1
```

o las filas 1 a 4 y las columnas 2 a 6

```
>> M_1426=M(1:4,2:6)
M_1426 =
      2     3    -1     4     6
      7     6     0     0     1
      5     6     1     1     0
      4     6     1     2     3
```

Como vemos, para numerar filas (o columnas) consecutivamente basta con poner `inicio:fin`, sin embargo, no es obligado que las filas (o columnas) sean consecutivas, es más, ni siquiera es obligado que sean ordenadas. Para ello, en lugar de especificar un rango, lo que podemos es pasar un vector con las filas (o columnas) que queremos extraer de la matriz. Por ejemplo:

```
>> filas=[2 4 1];
>> columnas=[6 4 1 3];
```

```
>> M_fc=M(filas,columnas)
M_fc =
     1     0     0     6
     3     1    -1     6
     6    -1     1     3
```

A partir de aquí, ya depende de la imaginación y/o de las necesidades que uno tenga a la hora de extraer submatrices.

Si en algún momento necesitásemos saber el tamaño de una matriz, podemos averiguarlo mediante la función `size(matriz)`, que nos da como resultado un vector de dos componentes, en la primera tenemos el número de filas de la matriz `matriz` y en la segunda el número de columnas. Por ejemplo, con `size(M_fc)` obtenemos `3 4`, es decir, la matriz tiene 3 filas y 4 columnas. Existe una función análoga para vectores, la función `length(vector)`, que devuelve el número de componentes del vector `vector`. Por ejemplo, `length([1,2,3,4,5])` devuelve `5`

En nuestro viaje por el instituto también nos enseñaron otras cosas que podíamos calcular con matrices, tales como su rango, el determinante (si es cuadrada), su inversa (si es cuadrada y su determinante es no nulo), la transpuesta, ... así que vamos a ir viendo cómo hacerlo con MATLAB. Prácticamente su nombre nos lo dice, pero de todas formas aquí tenemos una pequeña referencia:

- `rank(matriz)` nos dice el número de filas o columnas linealmente independientes que tiene una matriz (su rango). Por ejemplo, con la matriz M de antes, si calculamos `rank(M)` obtenemos `6`.
- `det(matriz)` calcula el determinante de la matriz cuadrada `matriz`. Si la matriz que introducimos no es cuadrada, MATLAB dará un error. Por ejemplo, con `det(M)` obtenemos `-5544`.
- `inv(matriz)` calcula la matriz inversa de `matriz` cuando esto sea posible, en caso contrario, MATLAB nos avisará dando un error. Obtendríamos el mismo resultado si hiciéramos `matriz^(-1)`
- Transposición: si escribimos `matriz'` obtendremos la matriz transpuesta de `matriz`, en caso de que esta sea real. Si la matriz a transponer es compleja, al escribir `matriz'`, MATLAB calcula la transpuesta de la matriz conjugada. Por ejemplo, si `A=[1+i 2+i;3+i 4+i]`, entonces `A'=[1-i 3-i;2-i 4-i]`. Si queremos que simplemente calcule la transpuesta, sin conjugar, tendremos que escribir `A.'` (nótese de nuevo el uso de la notación . *punto*).
- `trace(matriz)` calcula la suma de los elementos de la diagonal de la matriz `matriz`. Por ejemplo, con `trace([1 2 3;3 2 1])` obtenemos como resultado `1+2=3`.

Una nota: MATLAB interpreta `A/B` (A, B matrices cuadradas del mismo orden) como $A \cdot B^{-1}$, e interpreta `A\B` como $A^{-1} \cdot B$. Por tanto, si queremos resolver un sistema de n

ecuaciones con n incógnitas, suponiendo que sea compatible determinado, A la matriz de coeficientes y b el vector *columna* de los términos independientes, sabemos que la solución será $x = A^{-1} \cdot b$, por tanto, en MATLAB nos bastará con poner `A\b` (aunque también podríamos poner `inv(A)*b`; si b fuera un vector fila habría que transponerlo primero, y escribiríamos `inv(A)*(b')`).

Vamos a aplicar esto a resolver el sistema

$$\left\{ \begin{array}{rcl} x+2y+3z & = & 1 \\ 4x+5y+6z & = & 0 \\ 7x+y+4z & = & -1 \end{array} \right.$$

escribiendo b como vector fila (o como columna) y viendo cómo resolverlo en cada caso.

```
>> A=[1,2,3; 4,5,6; 7,1,4];
>> b=[1,0,-1];
>> det(A)
ans =
  -27
>> % Distinto de cero: estupendo! Se puede resolver.
>> A\(b')       % b es vector fila, hay que transponer
 ans =
    -0.6296
    -0.7407
     1.0370
>> inv(A)*(b')  % b es vector fila, hay que transponer
 ans =
    -0.6296
    -0.7407
     1.0370
>> b=[1;0;-1];  % Escribimos b ahora en columna
>> A\b
 ans =
    -0.6296
    -0.7407
     1.0370
>> inv(A)*b
 ans =
    -0.6296
    -0.7407
     1.0370
```

P.D: El carácter % significa que a partir de él, MATLAB ignora lo que escribes (siempre que no empieces línea nueva con RETURN), así que lo podemos usar para comentarios.

Capítulo 5

Porque todos llevamos dentro un artista...

5.1 Si eres un artista 2D, esta es tu sección

Seguramente recordarás del instituto el esfuerzo que requiere dibujar la gráfica de una función: había que hacer un estudio concienzudo de dominio, simetrías, periodicidad, monotonía, puntos singulares, etc, tras el cual, con algo de pulso, se podía pintar algo parecido a lo que queríamos. Afortunadamente, cuando uno se enfrenta con una máquina que está ejecutando un programa de matemáticas, este esfuerzo se reduce considerablemente para nosotros, pues no tenemos más que indicarle la función que queremos dibujar, y ya se las arregla él como puede para darnos el resultado.

MATLAB no está pensado para hacer gráficas como Derive, y eso a lo mejor nos desilusiona, pero a su manera también es útil, puesto que es mucho más apto para dibujar a partir de datos[1], y en esto sí que le saca ventaja. Para dibujar una curva en 2D, no tenemos más que escribir

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

donde x e y son dos vectores de n componentes que representan, respectivamente, los puntos sobre el eje X y sus correspondientes imágenes. Por defecto, MATLAB une estos puntos con segmentos. Si añadimos un tercer argumento al llamar a la función, podemos cambiar el color y estilo del trazo. Este tercer argumento debe ser de tipo cadena. Si consultamos la ayuda de MATLAB, vemos que nos dice:

[1]Por datos entendemos un vector x de abscisas y un vector y de ordenadas (o más vectores, si la dimensión es mayor que 2), obtenidas a partir de x, bien como resultado de un experimento, cálculos de un programa ...

```
Various line types, plot symbols and colors may be obtained with
PLOT(X,Y,S) where S is a character string made from one element
from any or all the following 3 colunms:

       y     yellow        .     point              -     solid
       m     magenta       o     circle             :     dotted
       c     cyan          x     x-mark             -.    dashdot
       r     red           +     plus               --    dashed
       g     green         *     star
       b     blue          s     square
       w     white         d     diamond
       k     black         v     triangle (down)
                           ^     triangle (up)
                           <     triangle (left)
                           >     triangle (right)
                           p     pentagram
                           h     hexagram

For example, PLOT(X,Y,'c+:') plots a cyan dotted line with a plus
at each data point; PLOT(X,Y,'bd') plots blue diamond at each data
point but does not draw any line.
```

Como se ve, la ayuda es, en muchos casos, bastante buena. Y de paso ya tienes la lista de los estilos disponibles para pintar.

Si tenemos un vector cuyas componentes tienen parte imaginaria no nula (llamémosle `X`), al hacer `plot(X)`, veremos que MATLAB hace lo mismo que si le hubiéramos dicho `plot(real(X),imag(X))`. En todos los demás usos que podamos hacer de la función `plot`, la parte imaginaria de las componentes de los vectores es ignorada.

Podemos pintar simultáneamente varias curvas, escribiendo

```
>> plot(X1,Y1,S1,X2,Y2,S2,...,X_n,Y_n,S_n)
```

donde cada triplete `(X_i,Y_i,S_i)` hace referencia a la curva obtenida por `(X_i,Y_i)` usando el estilo `S_i`. No es necesario que indiquemos en cada curva un estilo de trazo si queremos que se dibuje con el estilo por defecto, podemos escribir, por ejemplo `plot(x_1,y_1,x_2,y_2,'*')` y MATLAB pintará la primera curva con el estilo por defecto y la segunda con el estilo dado por `'*'`

Cuando tenemos una partición suficientemente pequeña, al pintar la gráfica nosotros no nos daremos cuenta de los segmentos que los unen debido a que la distancia entre los puntos es muy pequeña. Vamos a aclarar todo esto con unos dibujos, resultado de aplicar las siguientes órdenes:

```
>> x=linspace(-1,1,5);
```

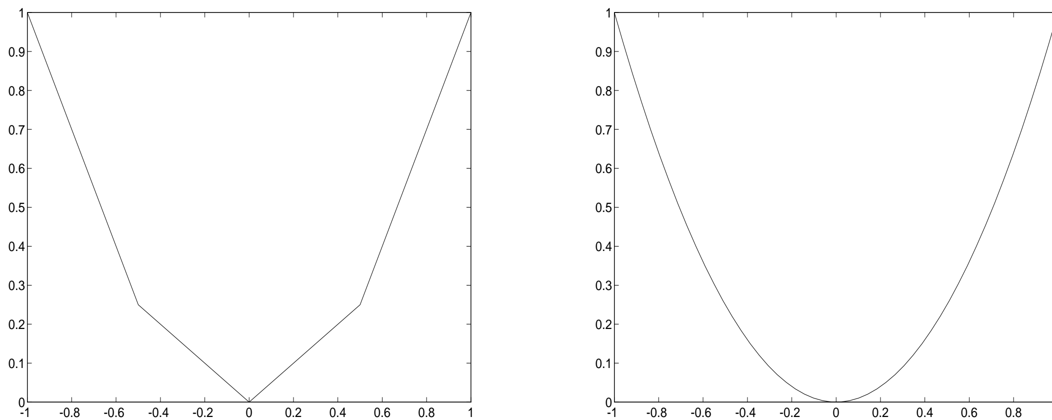


Figura 5.1: Gráficas con 5 y 50 puntos, respectivamente

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

por una parte, y por otra parte

```
>> x=linspace(-1,1,50);
>> y=x.^2;
>> plot(x,y)
```

Ambas dibujan la función $f(x) = x^2$ en el intervalo $[-1, 1]$, la diferencia está en que la primera sólo toma 5 valores del intervalo, y la segunda toma 50. En embas se unen esos puntos mediante segmentos para dar sensación de continuidad, pero por ser pocos puntos en el primer caso la vemos tan angulosa.

Cuando ponemos un gráfico sin más, no aparecen títulos ni nada que se le parezca, pero estaría bien poder ponerle un título que diga algo sobre el gráfico, alguna leyenda en el eje X o en el eje Y... para ello, disponemos de las funciones `title(cadena)`, `xlabel(cadena)` e `ylabel(cadena)`, donde `cadena` es el texto que nosotros queremos poner. Como es una cadena, este texto debe ir entre `'  '`. La función `title` es la que pone el título, la función `xlabel` es la que pone la leyenda en el eje X y la función `ylabel` pone la leyenda en el eje Y. Sin embargo, si pones un título y luego dibujas una función, por ejemplo:

```
>> x=linspace(-3,5);
>> y=1./(1+x.*x);
>> title('Mi primera grafica');
>> plot(x,y)
```

verás que primero se abre una ventana sin gráfico pero con el título, y después se dibuja la gráfica pero sin el título, ¿cómo hacemos para que pasen las dos cosas? Escribiendo

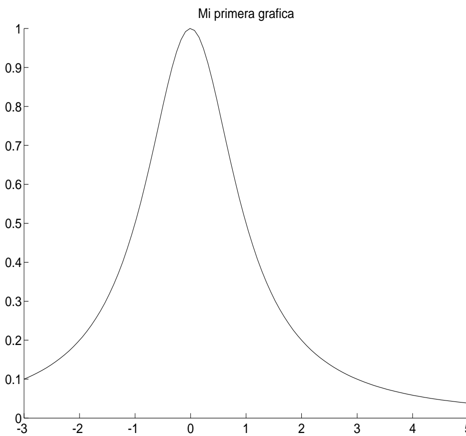


Figura 5.2: Gráfica 'sujeta' con `hold`

`hold` antes de pintar nada (ni títulos, ni gráficas). Al escribir `hold`, lo que hacemos es que, cuando dibujemos algo, se "sujete", de manera que al ir poniendo más elementos gráficos, se añaden al que teníamos en lugar de borrarlo. Una vez que ya tenemos el gráfico a nuestro gusto, podemos grabarlo en formato PS (PostScript) escribiendo `print nombre_fichero` (si en `nombre_fichero` no ponemos una ruta, MATLAB lo grabará en su directorio `bin`). Para "des-sujetar" los gráficos, escribimos de nuevo `hold`, pues éste actua como si fuera un interruptor ("sujeta", "des-sujeta", "sujeta", "des-sujeta", ...).

Podemos también escribir texto en una posición muy específica de la gráfica, con la función `text(x,y,cadena)`, donde x e y son la posición en la que queremos que aparezca el texto, y `cadena` es el texto que queremos escribir. O podemos situarlo con el ratón, escribiendo `gtext(cadena)`. Cuando escribimos esto, MATLAB nos pone en la pantalla gráfica y espera a que pulsemos con el ratón sobre algún punto; cuando lo hemos hecho, coloca el texto en esa posición. Esta característica puede aprovecharse con la función `ginput(n)`, que sirve para extraer información de un gráfico. `ginput(n)` devuelve una matriz de n filas y dos columnas, cuyas entradas (que serán pares (x, y)) son las coordenadas de los n puntos del gráfico sobre los que pinchemos. Esto puede servirnos, por ejemplo, para obtener de forma aproximada los extremos relativos de una función, como vamos a ver en el siguiente ejemplo:

```
>> R=[-2 1 4 7];
>> P=poly(R);
>> x=linspace(-3,10,1000);
>> y=polyval(x,y);
>> plot(x,y)
>> Extremo=ginput(3)
Extremo =
   -0.9032  -83.3333
    2.4516   40.9357
    5.8710  -83.3333
```

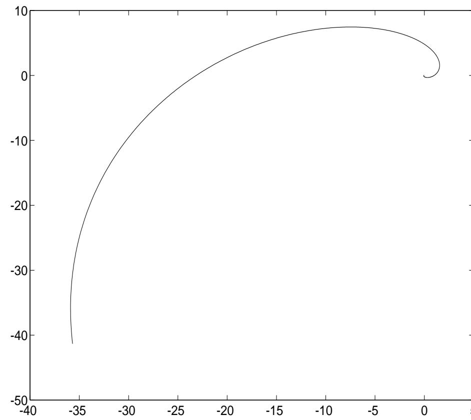


Figura 5.3: Curva en paramétricas $(e^t \cos t, e^t \sin t)$

MATLAB se espera a que pulses sobre la gráfica las veces que le has dicho en `ginput`, y al terminar te muestra las coordenadas de los puntos en los que has pinchado, que en nuestro ejemplo son, aproximadamente, los tres extremos relativos que tiene el polinomio construido a partir de las cuatro raíces dadas en el intervalo $[-3, 10]$. Estas coordenadas, además, han sido almacenadas en la matriz `Extremo` y las podemos usar posteriormente si las necesitamos sin más que hacer referencia a esta matriz.

Con esta misma idea, pintar una curva dada en paramétricas es igualmente sencillo. Sabemos que una curva en paramétricas tiene la forma $(f(t), g(t))$, donde t toma valores en un cierto intervalo de $\mathbb{R}$. Con el siguiente ejemplo, vamos a ver cómo se pintan estas curvas. Se trata de la curva $\alpha(t) = (\cos(t)e^t, \sin(t)e^t)$, para valores de t tales que $t \in [-4, 4]$:

```
>> t=linspace(-4,4,1000);
>> x=cos(t).*exp(t);
>> y=sin(t).*exp(t);
>> plot(x,y)
```

Por último, para finalizar este apartado sobre curvas planas vamos a ver cómo representar curvas dadas en coordenadas polares. Para ello, tenemos que usar la función `polar(angulo,radio)`. Por ejemplo, si queremos dibujar la función dada en polares $\rho(\theta) = \sin(5\theta)$ con $\theta \in [0, 2\pi]$, hacemos:

```
>> angulo=0:pi/600:2*pi;
>> radio=sin(5*angulo);
>> polar(angulo,radio)
```

y ante nosotros tendremos una hermosa flor de cinco pétalos.

Por si no te acuerdas: `angulo` es un vector cuya primera componente vale 0, la última

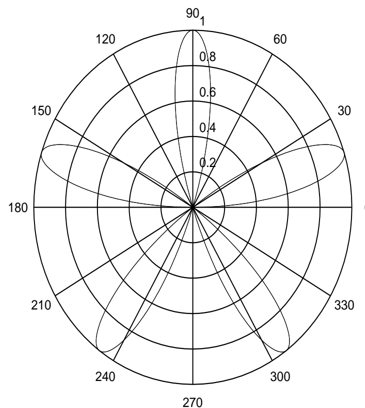


Figura 5.4: Curva en polares

vale 2π, y el incremento con el que se calcula el resto es $\pi/600$. Es decir, lo hemos definido de la forma `v=inicio:paso:fin`.

Con el fin de ejercitar el uso de la notación *punto* para las operaciones con vectores que queremos que se hagan componente a componente, y también para que practiques un poco de dibujo, te proponemos que representes con MATLAB las siguientes funciones en el intervalo $[-5, 5]$:

1. $f(x) = \dfrac{x^2\sqrt{15 - x^2}}{10}$
2. $f(x) = x \cdot \sin\left(\dfrac{2}{x}\right)$
3. $f(x) = -\sqrt{|\cot(0.8x)|}$
4. $f(x) = 2\sqrt{\sin x}$

P.D. Si no se te ocurre, prueba a escribir esto:

```
>> X=linspace(-5,5,500);
>> Y=( (X.^2).*sqrt(15-X.^2) )/10;
>> plot(X,Y)
>> Y=X.*sin(2./X);
>> plot(X,Y)
>> Y=-sqrt(abs(0.8.*X));
>> plot(X,Y)
>> Y=2.*sqrt(abs(sin(X)));
>> plot(x,y)
```

5.2 Y si eres un artista 3D, este es tu lugar

En 3D tenemos más posibilidades de expresar nuestras inquietudes, ya que podemos dibujar curvas o superficies, siendo esta última opción la que resulta más espectacular.

Pero el aprendiz de brujo debe primero saber barrer, así es que vamos a explicar ahora cómo representar curvas en 3 dimensiones, dejando para después el caso de las superficies.

5.2.1 Las curvas también tienen su encanto

Cuando queremos pintar una curva en paramétricas, no tenemos más que usar la función `plot3(x,y,z)`. Esta es la forma más sencilla en que podemos usar esta función. Le pasamos los vectores x, y, z (que deben tener la misma dimensión) y entonces representa la curva que en el eje X tomará los valores de x, en el eje Y los de y, y en el eje Z los de z. Lo más lógico es que estos x, y, z correspondan a funciones de t, donde t es la variable con la que parametrizamos la curva. Por ejemplo, si queremos representar la curva $\alpha(t) = (\sin t, \cos t, t)$ para $t \in [0, 4\pi]$, haremos, como hacíamos en 2D, una partición del intervalo $[0, 4\pi]$ donde tomará valores t, y evaluaremos las funciones en t, asignando el resultado a los vectores x, y, z:

```
>> t=linspace(0,4*pi,200);
>> x=sin(t);
>> y=cos(t);
>> z=t;
>> plot3(x,y,z)
```

Y he ante nuestros ojos nuestra conocida helicoide.

Podemos pintar varias curvas a la vez usando la función `plot3`. Si llamamos `X1`, `Y1`, `Z1`, . . . , `Xn`, `Yn`, `Zn` a los vectores con las correspondientes parametrizaciones en los ejes X, Y, Z de n curvas, escribiendo `plot3(X1,Y1,Z1,...,Xn,Yn,Zn)` obtendremos la representación simultánea de las n curvas. Si, además, queremos cambiar el estilo/color del dibujo, podemos poner también tras cada triplete una cadena con la correspondiente definición del estilo que queremos modificar, por ejemplo:

```
>> t=linspace(0,6*pi,300);
>> X1=sin(t); Y1=sin(2*t); Z1=t.*sin(t);
>> X2=sin(t); Y2=cos(t); Z2=t.*cos(t);
>> X3=cos(t); Y3=sin(t); Z3=t;
>> plot3(X1,Y1,Z1,X2,Y2,Z2,'*',X3,Y3,Z3)
```

Pintaremos las curvas primera y tercera con el estilo por defecto, pero el de la segunda lo hemos modificado con `*`, como puede verse en la figura 5.5. Los estilos son los mismos que los de la función `plot`.

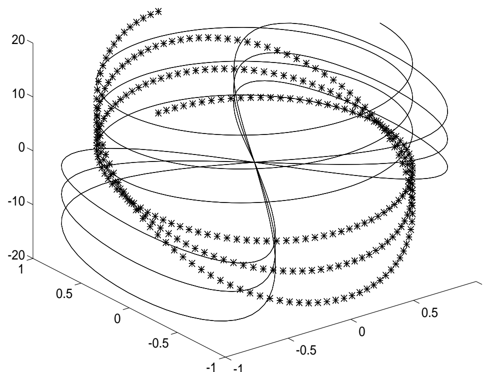


Figura 5.5: Tanta curva junta termina siendo confusa

5.2.2 Aunque hay quien prefiere las superficies

Y como gustos hay para todos, los interesados en las superficies tienen aquí su rinconcito particular. Hay varias opciones para los gráficos de superficies: podemos representar la superficie, o tan sólo las curvas de nivel, o cortes de ella, ... De todas ellas, vamos a empezar viendo cómo representar la superficie propiamente dicha. Para ello, usaremos lo que se conoce como gráficos de malla. Estas mallas se dibujan por medio de la función `mesh(matriz)`. Esta función lo que hace es crear un gráfico a partir de una matriz. Las filas y columnas denotan las coordenadas de un rectángulo, y el valor que toman las entradas es el valor de la función y el que será representado para el punto de coordenadas `(fila, columna)`.

Como todo esto suena un poco complicado, vamos a ver un primer ejemplo. Si escribimos `mesh(rand(12))`, MATLAB toma como rectángulo XY el $[0,12]\times[0,12]$, puesto que la matriz es 12×12. Ahora, sobre cada punto de coordenadas (i,j), siendo éstos los índices de la matriz, pinta un punto a altura `rand(i,j)`, es decir, el valor aleatorio que le haya dado en esa posición. Luego, como no le hemos especificado nada por defecto, une esos puntos por líneas, y de ahí sale esa figura tan caótica.

Lo que queremos es hacer una partición más fina del intervalo que nosotros vayamos a representar, y entonces evaluar una cierta función sobre el rectángulo, creando con ello la matriz que necesita la función `mesh`. Para ello tenemos la función `meshdom(x,y)`. Esta función devuelve dos matrices, una primera matriz en la que cada fila es igual a x, y una segunda matriz en la que cada columna es igual a y. Con estas dos matrices (digamos X e Y) ya podemos calcular la matriz z, sin más que aplicar $z=f(x,y)$ sobre cada entrada de X e Y.

Como ejemplo, vamos a representar la función $z=e^{-x^2-y^2}$ en el rectángulo $[-2,2]\times[-2,2]$ siendo el paso entre puntos en cada intervalo 0.1:

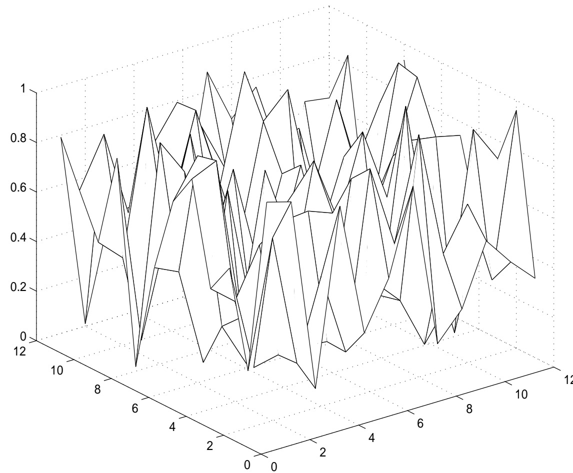


Figura 5.6: Una representación un tanto aleatoria

```
>> x = -2:0.1:2;
>> y = -2:0.1:2;
>> [X,Y] = meshdom(x,y);
>> z = exp(-X.^2-Y.^2);
>> mesh(z)
```

completamente equivalente a hacer

```
>> [x,y] = meshdom( -2:0.1:2, -2:0.1:2);
>> z = exp(-x.^2-y.^2);
>> mesh(z);
```

No te preocupes por el warning (advertencia) que da MATLAB sobre la función; simplemente es que esta función ya es obsoleta y recomienda que uses en su lugar la función `meshgrid`. Esta función se utiliza esencialmente igual que la antigua `meshdom`, lo que sucede es que incorpora más utilidades que podeis consultar haciendo `help meshgrid`. Puedes comprobar que el uso básico que obtenemos con `meshdom` también puedes obtenerlo con `meshgrid` sin más que cambiar el nombre la función. Pruébalo con el ejemplo anterior. Te saldrá lo mismo.

Otra forma de representar las superficies es con la gráfica *"de placas"*, que se obtiene con la función `surf(x,y,z)`. Cuando x, y y z son matrices de la misma dimensión, obtenemos la gráfica "a placas" de la superficie que queremos dibujar, como se ve en el ejemplo:

```
>> [x,y] = meshgrid( -2:0.1:2, -2:0.1:2);
>> z = exp(-x.^2-y.^2);
>> surf(x,y,z);
```

Cuando queremos hacer un gráfico 3D también podemos ponerle título y leyendas, al igual que hacíamos en 2D, usando las mismas funciones (recordando que había que poner `hold`).

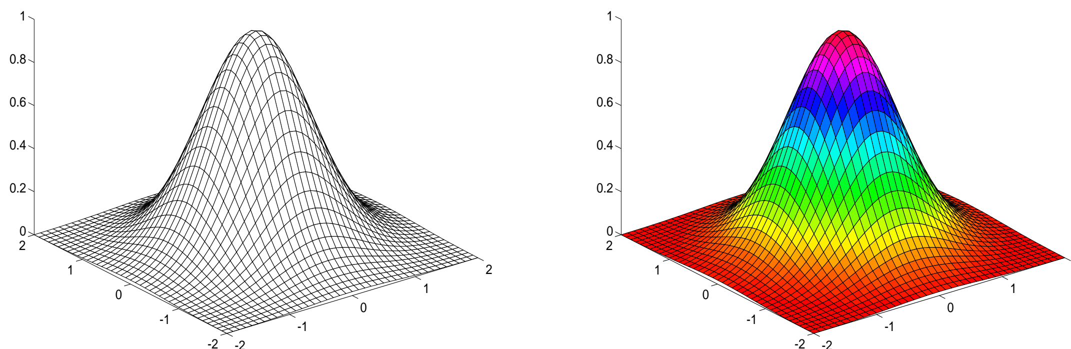


Figura 5.7: Una bonita gaussiana, mallada (izqda.) y a placas (dcha.)

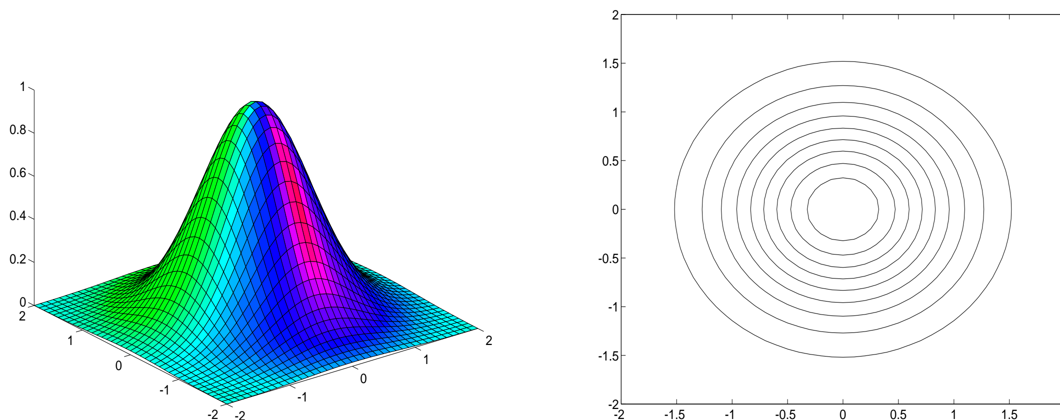


Figura 5.8: Superficie suavizada (izqda.) y curvas de nivel (dcha.)

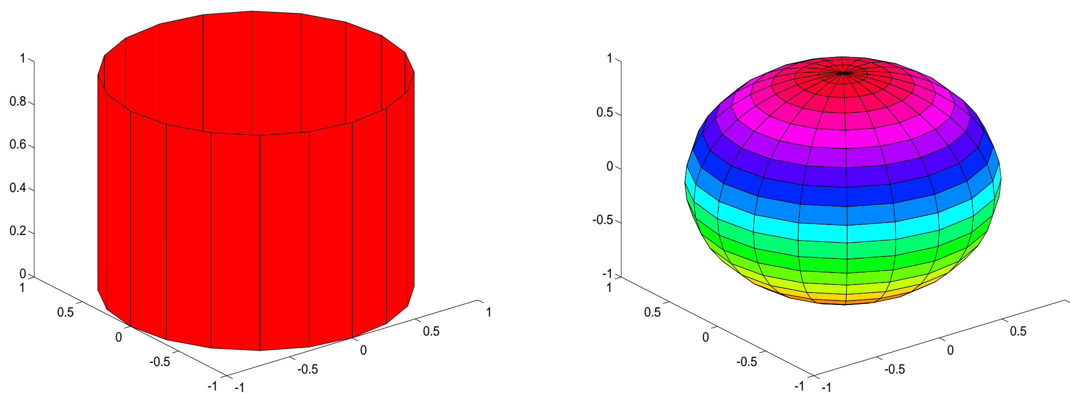


Figura 5.9: Un cilindro y una esfera

Tan sólo añadimos una que es obvia: una función para poner una leyenda en el eje Z: `zlabel(cadena)` donde `cadena` es un texto delimitado por comillas simples.

Tenemos más funciones para representar superficies: `surfl(x,y,z)`, que nos representa la superficie suavizada, `contour(x,y,z)` dibuja las curvas de nivel, ... no hay más que consultar la ayuda con `help plotxyz`.

Sin lugar a dudas, las dos superficies que más fácilmente podemos representar con MATLAB son un cilindro y una esfera: no hay más que escribir `cylinder` o bien `sphere`, sin argumentos. Los resultados son los de las figuras. Y con todo esto terminamos el capítulo dedicado a los gráficos.

Capítulo 6

Cuando las funciones (y funcionalidades) básicas se quedan cortas, aprende a hacerlas tú mismo

6.1 Un primer vistazo a los ficheros m. Declaración de funciones

Cuando hablamos de guardar nuestro trabajo, dijimos que al hacer `save` tan sólo guardábamos las variables existentes en memoria, pudiendo recuperarlas más tarde. También dijimos que podíamos guardar los pasos realizados en nuestro trabajo con `diary`, pero que luego no podíamos recuperar este trabajo para que MATLAB volviera a seguir todos los pasos que hicimos la otra vez. Entonces surge una pregunta obvia: ¿es que no hay manera de grabar una secuencia de operaciones para que MATLAB las realice cuando nosotros queramos cargarlas de un fichero?

Afortunadamente, la respuesta es sí, y pasa por grabar lo que se hace, o por editar previamente, lo que se conoce como *ficheros punto m (.m)*.Estos ficheros no son más que ficheros ASCII que pueden contener una secuencia de órdenes MATLAB, puede haber variables definidas en él junto con esos comandos ... Para crear uno, sólo necesitamos un editor de texto que sea capaz de grabar un fichero como texto plano (ASCII), y darle la extensión `.m` al grabarlo.

Luego, para ejecutarlo desde MATLAB, tenemos que incluirlo en su "camino" haciendo uso de la función `path()`. Podemos ejecutarla con uno o dos paramétros. Si la ejecutamos con un parámetro (que debe ser una cadena que contenga la ruta en la que estarán nuestros archivos `.m`), a partir de ese momento, nuestro directorio pasa a ser el único directorio en el que MATLAB buscará las funciones que intentemos ejecutar. Esto no es lo que queremos si sólo pretendemos *añadir* un directorio a los que ya hubiera definidos en su "camino".

Si simplemente escribimos `path`, vemos que MATLAB nos muestra una lista con todos

los directorios en los que él busca las funciones. Si consultamos la ayuda, vemos que nos dice que al ejecutar `path(P1,P2)`, siendo `P1` y `P2` cadenas, lo que hace es añadir a la cadena P1 la cadena P2, y poner los directorios que haya en la cadena P1 como "camino" en el que buscar funciones. Pues ya lo tenemos: si escribimos `path(path,'unidad:\mi_directorio')` (en el caso de un sistema MS-DOS, en el caso de un UNIX pondríamos `'/mi_directorio'`), el directorio `mi_directorio` se añadirá a la lista que hay en `path` (todos los que tenía MATLAB al iniciar el programa), y la lista completa será el nuevo "camino" en el que MATLAB buscará una función que queramos ejecutar. Con esto conseguimos nuestro objetivo: añadir nuestro directorio a la lista de búsqueda.

Una vez que hemos añadido el directorio en el que tengamos nuestros ficheros `.m` con las funciones que nosotros hayamos ido creando, ya podemos usarlas con la misma tranquilidad con la que usábamos las que MATLAB nos da en cuanto entramos al programa, simplemente escribiendo su nombre y los argumentos de llamada y, si es el caso, asignando a una o más variables la salida de dicha función.

Vamos a ver un ejemplo en el que crearemos una función, y con ello explicaremos cómo se crean. Lo primero es abrir un editor ASCII y escribir lo siguiente:

```
function [TriSup,TriInf,Diagon]=trimat(A)
% Entrada: Una matriz A
% Salida: Tres matrices. TriSup contiene la parte triangular superior
%         de la matriz A, TriInf la parte triangular inferior, y Diagon
%         la diagonal de la matriz A
TriSup = triu(A);
TriInf = tril(A);
Diagon = diag(A);
```

Lo grabamos como `trimat.m` (es muy recomendable que lo grabemos con el mismo nombre que le hemos dado a la función), introducimos en MATLAB una matriz `A` cuadrada, y a continuación escribimos `[A_sup,A_inf,A_diag]=trimat(A);`. A partir de este momento tenemos tres matrices, `A_sup`, `A_inf` y `A_diag`, que contienen, respectivamente, la parte triangular superior, parte triangular inferior y parte diagonal, de la matriz `A` que introduzcamos como argumento. Las líneas que están escritas como comentarios sirven para dos cosas: la primera es que, cuando revisemos el fichero tres meses después de haber escrito la función, nos ayudarán a saber qué habíamos hecho. La segunda utilidad la descubrirás inmediatamente si escribes `help trimat`. Además, si escribes `type trimat` verás que aparece todo el código de la función. Prueba a escribir `type vander` (`vander` es una función que viene con MATLAB). Puedes aprender mucho viendo cómo están hechas algunas funciones.

Al escribir en la primera línea del archivo la palabra reservada `function`, estamos diciendo a MATLAB que el fichero es un fichero de función, así que no tenemos que especificar de ninguna forma especial el fin de la función. Simplemente, cuando ya no haya más código, MATLAB devolverá las variables correspondientes con lo que haya calculado para ellas en la función. La sintaxis general para definir una función es:

```
function [valor_dev1, ... , valor_devN]=nombre_fun(arg1, ... , argM)
```

Podemos escribir nuestras funciones usando comandos (o funciones) de MATLAB de forma secuencial, pero con esto no estamos aprovechando toda la potencia que nos brinda. En el punto siguiente veremos qué más elementos podemos utilizar para programar nuestras funciones.

MATLAB nos da un comando básico, el comando `!`, tras el cual, si escribimos algo y pulsamos `RETURN`, MATLAB tratará de ejecutar el comando del sistema operativo que le hayamos escrito. Esto es muy útil cuando estamos programando, porque no tenemos que estar continuamente abriendo y cerrando ventanas del editor de texto o, incluso, si el sistema operativo no es multitarea, saliendo de MATLAB, entrando en el editor, escribiendo el programa, saliendo del editor y volviendo a entrar en MATLAB. Bastará con que escribamos `!nombre_editor ruta\nombre_fichero`; entonces, MATLAB abre el editor, escribimos, grabamos y cerramos el editor, y automáticamente ya estamos de vuelta en MATLAB. Si tu sistema operativo es un UNIX y tienes costumbre de usar *vi*, bastará con que escribas `!vi /ruta/archivo.m`. Si tu editor es *emacs*, pues en vez de `vi` pones `emacs` (obviamente). Si tu sistema operativo es un Windows y tienes costumbre de usar el *edit*, pues escribes `!edit unidad:\ruta\archivo.m` y a programar.

Los ficheros `.m` también sirven para definir variables, como vectores y matrices que sean muy grandes, o que sean resultado de un programa. Vamos a poner un ejemplo en el que, de alguna manera, a partir de los datos del fichero `x.m`, se han obtenido unos datos que están grabados en el fichero `y.m`, y queremos representarlos. Por ejemplo, vamos a poner que en el fichero `x.m` tenemos los datos

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

y en el fichero `y.m` tenemos los datos

```
5
6
3
8
7
-1
0
4
```

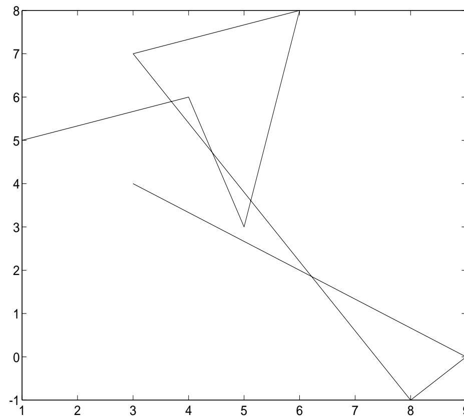


Figura 6.1: Una figura un tanto esotérica que muestra el uso de ficheros

cargamos esos datos con `load x.m` y `load y.m`, y ahora hacemos `plot(x,y)`: veremos como resultado una figura un tanto esotérica que corresponde a la gráfica de los pares (x, y) unidos por segmentos. Queda claro que el dibujo está hecho un tanto al azar por ser un ejemplo, pero queda igualmente claro que esto nos será muy útil para representar gráficas de procesos que hayamos obtenido de forma numérica, así como que lo que hemos dicho para 2D vale para 3D, sin más que añadir un tercer fichero `z.m` en el que tuviéramos las correspondientes coordenadas, con lo que tendríamos una curva 3D.

Es interesante lo que nos dice MATLAB cuando le pedimos ayuda sobre la función `load`:

```
LOAD Retrieve variables from disk.
  LOAD fname  retrieves the variables from the MAT-file 'fname.mat'.
  LOAD, by itself, loads from the file named 'matlab.mat'.
  LOAD xxx.yyy  reads the ASCII file xxx.yyy, which must contain a
  rectangular array of numeric data, arranged in m lines with n values
  in each line.  The result is an m-by-n matrix named xxx.

  To load an ASCII file that does not have a filename extension, use
  LOAD fname -ascii. Otherwise MATLAB adds the extension '.mat' and
  tries to load it as a MAT-file. To load a MAT-file that does NOT
  have a '.mat' extension, use LOAD fname.ext -mat.
```

Nosotros hemos hecho `load x.m`. Como los datos estaban dispuestos uno por fila, MATLAB ha creado una variable con el mismo nombre que el fichero, pero sin la extensión, es decir, `x`, le ha dado dimensiones `num_lineas_fichero`$\times 1$ y con ello ya tenemos el vector `x`. Igualmente hemos obtenido el vector `y.m`. Si en lugar de vectores queremos que sean matrices, el fichero deberá contener más de un dato por fila. Si queremos que la matriz sea 3×2, tendremos que crear un fichero con 3 líneas que contenga cada una dos datos. Por ejemplo, supongamos que tenemos los siguientes datos en un fichero `a.m`:

```
1 4
3 -7
4 8
```

si ahora hacemos `load a.m`, MATLAB creará una variable `a` de dimensiones 3×2 cuyos elementos serán los del fichero, es decir, tendremos que es la matriz

$$\begin{pmatrix} 1 & 4 \\ 3 & -7 \\ 4 & 8 \end{pmatrix}$$

6.2 Programación con MATLAB

Entramos de lleno en lo que hace de MATLAB una herramienta poderosa: un lenguaje de programación muy sencillo con el que abordar tareas complejas y cuyos elementos más básicos pasamos a describir aquí. Para poder seguir este punto se presuponen ciertas nociones sobre programación en general, nociones que no vamos a exponer aquí pues no forman parte del propósito de esta introducción a MATLAB. Así que vamos a empezar con la descripción de los elementos del lenguaje de los que dispone MATLAB para programar.

6.2.1 Bucles

Los bucles en MATLAB tienen la forma

```
for variable_contador = INICIO : PASO : FIN
    Accion_1
      ...
    Accion_M
end
```

Además, podemos anidarlos según nuestras necesidades, teniendo en cuenta que cada bucle debe finalizar con su correspondiente `end`. En caso de no especificar `PASO`, se sobreentiende que es 1. Los bucles pueden sernos útiles para definir matrices, por ejemplo:

```
A=zeros(10);
for i=1:10
  for j=1:10
    A(i,j)=rem( i+j, 10);
  end;
end
```

Notar que previamente hemos inicializado la matriz `A` haciendo que todas sus componentes sean ceros; esto puede sernos útil porque así dimensionamos previamente la matriz y puede evitarnos futuros errores en el programa.

6.2.2 Condicionales

De las tres posibles estructuras condicionales que se dan en la programación, tenemos disponibles dos de ellas en MATLAB (realmente, con dos hay suficiente), el condicional **if** y el condicional **while**, cuya sintaxis exponemos:

Condicional IF

La estructura del *if* simple es la siguiente:

```
if condicion
  Acciones a realizar si es cierta la condicion
end
```

y la estructura del *if* con alternativa es:

```
if condicion
  Acciones a realizar si es cierta la condicion
else
  Acciones a realizar si es falsa la condicion
end
```

También tenemos *if* con alternativa `elseif`:

```
if condicion1
  Acciones a realizar si es cierta condicion1
elseif condicion2
  Acciones a realizar si es cierta condicion2
else
  Acciones a realizar si no son ciertas condicion1 y condicion2
end
```

La condición debe ser una expresión lógica que MATLAB reconoce, pero de forma vectorial, asociándole el valor 1 si es cierta y el valor 0 si es falsa. Vamos a ver unos ejemplos tomando los vectores `x=[-1,0,1]` e `y=[0,1,-1]`.

- `x==0` devuelve `[0,1,0]`. `==` es el operador de igualdad; MATLAB compara si cada una de las componentes del vector `x` es igual a 0, poniendo en el resultado un 0 en el lugar de las componentes que no lo son, y un 1 en el lugar de las componentes que sí lo son. Si hubiéramos puesto `x==pi`, MATLAB nos hubiera devuelto `[0,0,0]` porque el vector `x` no tiene componente alguna cuyo valor sea `pi`.

- `x>0` devuelve `[0,0,1]` mientras que `x>=0` devuelve `[0,1,1]`. De igual forma tenemos los operadores de desigualdad `<` y `<=`. La no igualdad la conseguimos con `~=`. Si escribimos `x~=0`, MATLAB nos devolverá `[1,0,1]`, pues las componentes primera y tercera son distintas de cero, mientras que la segunda no es distinta de cero.

La expresiones lógicas pueden ser compuestas, usando para ello los nexos conjuntivos, disyuntivos y negativos, como vemos a continuacion:

- La conjunción `AND` se escribe `&` en MATLAB, así, si hacemos `(x>=0)&(y>0)`, MATLAB nos responde `[0,1,0]`, pues `x>=0` es `[0,1,1]` mientras que `y>0` es `[0,1,0]`, así que ambas son ciertas cuando es `[0,1,0]`.

- La disyunción `OR` se escribe `|` en MATLAB, por tanto, `(x>=0)|(y>0)` devuelve `[0,1,1]`. También tenemos la disyunción exclusiva `XOR`, que en MATLAB se escribe `xor`, y se usa como en el siguiente ejemplo: `xor((x==1),(y>=-1))`, cuyo resultado es `[1,1,0]`

- Por último, podemos negar toda una proposición sin más que delimitarla entre paréntesis y anteponer el símbolo de la negación `~`. Por ejemplo, con `~(x==1)` obtenemos `[1,1,0]`, pues `x==1` es `[0,0,1]`

Por último sólo queda señalar que, al igual que con los bucles `for`, el condicional `if` puede anidarse según la complejidad que necesitemos.

Veamos un ejemplo ilustrativo aunque un poco simplón de *if-elseif-else*:

```
function mayor(a,b)
% Funcion tonta que te dice si, dados dos argumentos de entrada
% A y B, son iguales, uno mayor que el otro o menor
% El resultado solo tiene sentido cuando A,B son numeros reales

if a==b disp('a=b');
 elseif a>b disp('a>b');
 else disp('a<b');
end
```

Si lo probamos con números (por ejemplo, prueba a hacer `mayor(3,4)`, verás que el resultado es `'a<b'`), el programa nos dice si a es mayor, menor o igual que b, siendo a y b los números que hemos introducido. Ahora, si lo probamos, por ejemplo, con matrices, observarás que el resultado no tiene mucho sentido, puesto que ¿cuándo es una matriz mayor o menor que otra si no lo son componente a componente?

Condicional WHILE

La estructura del condicional *while* es igual de sencilla:

```
while condicion
   Instrucciones
end
```

donde `condicion` es una expresión lógica construída de acuerdo a cómo lo hemos expuesto arriba.

Vamos a ver un ejemplo en el que definimos una función que haga lo siguiente: dado $\varepsilon > 0$, encontrar el menor n_0 natural tal que $\left|\frac{\pi^2}{6} - \sum_{k=1}^{n} \frac{1}{n^2}\right| < \varepsilon$.

```
function n0=existe_n(epsilon)
% Entrada: Un cierto epsilon positivo
% Salida: El menor n_0 natural tal que la diferencia entre
%         (pi^2)/6 y la suma parcial n-esima de 1/n^2 es
%         menor que epsilon
Sum = 0;
Cont = 1;
ExactSum = (pi^2)/6;

while abs(ExactSum-Sum) > epsilon
   Sum = Sum + 1/(Cont^2);
   Cont = Cont + 1;
end

n0 = Cont - 1;
```

Si ejecutamos esta función escribiendo `n=existe_n(0.1);`, obtendremos en `n` el valor 10; si escribimos `n=existe_n(0.004)` en `n` tendremos el valor 250, etc.

La palabra clave BREAK

A lo mejor queremos controlar en un condicional que si se da una ocasión muy excepcional, como algún posible error que tengamos previsto, el programa no continue en el condicional, sino que queremos que salga fuera y se muestre por pantalla algún mensaje informando del error. Pues para ello tenemos la palabra reservada **break**, como vemos en el siguiente ejemplo:

```
function mayorC(a,b)
```

```
% Funcion tonta similar a mayor, pero excluye el caso complejo dando error
if (imag(a)~=0)|(imag(b)~=0)
  disp('Error: no existe orden en C');
  break
elseif a==b disp('a=b');
 elseif a>b disp('a>b');
  else disp('a<b');
end
```

y pruébala con `mayorC(i,1)` o con `mayorC(3,i)`.

Capítulo 7

Unas palabras para finalizar: Ejemplos

Con todo lo visto a lo largo de este tutorial, uno ya tiene herramientas más que suficientes como para desenvolverse con el uso de MATLAB. Se han descrito todas las funcionalidades básicas y algunos aspectos un poco más avanzados, con el fin de que a partir de aquí el propio estudiante sea capaz de encontrar (y hacer) aquello que necesita. Para finalizar con esta introducción, vamos a realizar unos ejemplos un poco más complejos en los que haremos uso de algunas funciones nuevas.

7.1 Primer ejemplo: Un problema de álgebra lineal; la descomposición LU

Cuando tenemos un sistema de n ecuaciones con n incógnitas $Ax = b$, donde A es la matriz de coeficientes (cuadrada de orden $n \times n$), sabemos que el sistema tiene solución única si y sólo si el determinante de la matriz es no nulo. Entonces, la idea para resolver el sistema es multiplicar a izquierda, ambos lados de la ecuación, por la inversa de la matriz A. Sin embargo, calcular la inversa de la matriz es un proceso costoso, más costoso cuanto más grande sea la matriz. En lugar de eso, lo que se hace es descomponer la matriz A como el producto de dos matrices que llamamos L (*lower*) y U (*upper*). Es decir, $A = L \cdot U$. La matriz L es una matriz triangular inferior con unos en la diagonal, y la matriz U es una matriz triangular superior.

Vereis en la carrera que esta descomposición es única (así como unos requisitos que deben darse para poder calcularla). ¿Por qué nos interesa? Pues porque una vez tenemos la descomposición, si $Ax = b$ y $A = L \cdot U$, entonces $L \cdot Ux = b$. Ahora, multiplicando por la izquierda por la inversa de L a ambos lados de la ecuación, se tiene que $Ux = L^{-1}b$, y multiplicando a izquierda por la inversa de U tenemos finalmente $x = U^{-1}L^{-1}b$. La ventaja que tiene esto es que calcular la inversa de una matriz triangular (inferior o superior) es bastante más sencillo (y menos costoso) que calcular la inversa de la matriz A.

MATLAB nos ofrece una función que calcula directamente estas dos matrices: la función

`lu(matriz)`. Esta función devuelve dos matrices que son, respectivamente, la matriz L y la matriz U de la descomposición mencionada. Para usarla, tendremos que escribir `[L,U]=lu(matriz)`. Vamos a resolver el siguiente sistema usando esta nueva técnica:

$$\left\{\begin{array}{rcl} x+2y+3z &=& 1 \\ 5y+6z &=& 0 \\ y+z &=& 0 \end{array}\right.$$

La matriz de coeficientes será $A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 1 & 1 \end{pmatrix}$, así que escribimos en MATLAB

```
>> A=[1,2,3;0,5,6;0,1,1];
>> [L,U]=lu(A)
L =
   1.00000000000000                  0                  0
                  0   1.00000000000000                  0
                  0   0.20000000000000   1.00000000000000
U =
   1.00000000000000   2.00000000000000   3.00000000000000
                  0   5.00000000000000   6.00000000000000
                  0                  0  -0.20000000000000
```

Y ahora, para resolver el sistema, escribimos:

```
>> b=[1;0;0];
>> sol=U\(L\b)
sol =
   1
   0
   0
```

Obtenemos exactamente lo mismo que si hacemos

```
>> sol=A\b
sol =
     1
     0
     0
```

¿Dónde está la diferencia para preferir el método de la descomposición LU? La diferencia está en el número de operaciones. MATLAB tiene una función que cuenta el número de operaciones que ha realizado desde que se ejecutó el programa, es la función `flops`. Si escribimos `flops` sin más, MATLAB nos dirá cuántas operaciones ha hecho desde que lo

ejecutamos en la línea de comandos de nuestro sistema operativo. Lo que queremos es saber cuántas operaciones nos lleva hacer un cierto proceso. Pues ponemos el contador de operaciones a cero justo antes de empezar con nuestro proceso, y cuando haya terminado contamos las operaciones que ha realizado con `flops`. ¿Cómo lo ponemos a cero? Escribiendo `flops(0)`. Vamos a usar esto para comparar los métodos de la descomposición LU con el de invertir la matriz A, y veamos qué resultados obtenemos:

```
% Primero inicializamos la matriz y el vector
>> A=[1,2,3;0,5,6;0,1,1];
>> b=[1;0;0];
% Descomposicion LU
>> flops(0);
>> [L,U]=lu(A);
>> U\(L\b)
ans =
     1
     0
     0
>> flops
ans =
    43
% Metodo "tradicional"
>> flops(0);
>> A\b
ans =
     1
     0
     0
>> flops
ans =
    75
```

El método tradicional lleva más operaciones que el de la descomposición LU, y hay que hacer notar que en el caso de la descomposición, con el número de operaciones van contadas las que lleva hacer la descomposición más las dos inversas por el vector, mientras que con el método tradicional sólo se ha hecho una inversión. La ventaja queda clara.

7.2 Segundo ejemplo: Integración definida

El problema del cálculo del área encerrada por una función $f(x)$ entre dos puntos de abscisa a y b puede abordarse de diversas formas. La idea básica consiste en tomar una partición $\{x_i\}$ del intervalo $[a,b]$ suficientemente pequeña, calcular el área de cada uno de los rectángulos de base $x_i - x_{i-1}$ y altura $f(x_i)$ y luego sumar todas estas áreas; este valor

es una aproximación al valor del área. Se trata de un método lento que funciona bien si la partición es bastante grande (tiene muchos puntos). En el curso de la carrera verás métodos que funcionan bastante mejor que esta primera idea. Uno de ellos es el método de Simpson. Si llamamos $I(f)$ al valor de la integral definida de la función f en el intervalo $[a, b]$, con el método de Simpson este valor es

$$I(f) = \frac{h}{6} \cdot \left(f(a) + 2 \cdot \sum_{i=1}^{n-1} f(x_i) + 4 \cdot \sum_{i=0}^{n-1} f\left(\frac{x_i + x_{i+1}}{2}\right) + f(b) \right)$$

siendo h el tamaño de la partición y n el número de subintervalos en que dividimos el intervalo inicial $[a, b]$. Siempre se cumple la relación $\frac{b-a}{n} = h$ y viceversa, $n = \frac{b-a}{h}$. Fácilmente podemos programar una función que calcule integrales definidas usando este método, pero MATLAB nos lo pone más fácil aún: usando la función `quad`. En su forma más simple, no tenemos más que llamarla usando tres parámetros: `quad('f',a,b)`, y MATLAB calculará la integral de la función que queremos (ahora explicamos cómo introducirla) entre los puntos a y b, con una tolerancia de 10^{-3}.

Está claro que `a` y `b` son, respectivamente, el extremo inferior y superior del intervalo. 'f' es una cadena, cuyo contenido debe ser el nombre de una función. Esta función debe devolver un vector de valores de salida si le introducimos un vector de valores de entrada. Vamos a ver un ejemplo de uso en el que calcularemos, mediante esta función, $\int_0^4 \frac{1}{1+x^2}\,dx$. En primer lugar, escribimos el fichero `fun1.m`, cuyo contenido es este:

```
function resul=fun1(x)
 resul=1./(1+x.^2);
```

Ahora que tenemos el fichero donde la función está definida, simplemente escribimos `quad('fun1',0,4)`, y MATLAB nos dice `1.3258`. Notad que en las operaciones que se hacen en la función hemos usado la notación *punto*. Recordad que antes habíamos dicho que la función debía ser capaz de devolver un vector si le introducían un vector, así que como queremos que las operaciones sean componente a componente debemos poner los puntos.

Si ejecutamos la función con cuatro parámetros, `quad('f',a,b,tol)`, integra la función especificada por `'f'` entre a y b con una tolerancia relativa `tol` (es decir, se la decimos nosotros).

Existe otra función que calcula integrales definidas, pero con un método de orden mayor que la que hemos expuesto: es la función `quad8` y se usa exactamente igual que la función `quad`. De todas formas, si teneis alguna duda siempre podeis decirle a MATLAB `help quad` o `help quad8`

7.3 Ejemplo final: El atractor saltarín

Hopalong es el nombre que recibe un atractor cuya peculiaridad es la siguiente: con la misma implementación del algoritmo, en dos máquinas diferentes hay una probabilidad muy alta de no obtener la misma figura. Este atractor se define por recurrencia:

$$\begin{aligned} x(0) &= 0 \\ y(0) &= 0 \\ x(n+1) &= y(n) - \text{signo}(x(n)) \cdot \sqrt{|b \cdot x(n) - c|} \\ y(n+1) &= a - x(n) \end{aligned}$$

donde a, b y c son tres parámetros que debemos especificar previamente. Lo que vamos a hacer es un programa con MATLAB que dibuje este atractor. Este programa va a ser una función a la que le pasaremos cuatro parámetros: los parámetros naturales del atractor, a, b y c, y el número de iteraciones que queremos que realice (este número será multiplicado por 1000). Incluiremos el directorio en el que grabemos la función en el `path` de MATLAB, y la podremos llamar directamente para ver cómo es la forma del atractor según los valores de los parámetros.

Un comando que aparece en el listado de la función y que no hemos comentado previamente es el comando `return`: nos permite salir de la función en cualquier punto de la ejecución. Si no vamos a permitir hacer más de 20 iteraciones, aparte de dar el mensaje de error tendremos que salir de la función. Ese es el motivo por el cual hemos puesto el comando.

Ponemos también la gráfica que hemos obtenido ejecutando `hopalong(0.4,1,0,3);` y te invitamos a que la pruebes, a ver si sale o no sale lo mismo.

```
function hopalong(a,b,c,n_iter)
% Hop-Along: El atractor saltarin
% Entrada:   Los parametros a,b,c y el numero de iteraciones
% Salida:    El dibujo del atractor saltarin
%
% La forma de la iteracion es    x(n+1)=y(n)-sign(x(n))*sqrt(abs(b*x(n)-c))
%                                y(n+1)=a-x(n)
% Comenzando por x(0)=0=y(0)
% Numero maximo de iteraciones: 20 (*1000=20000)

if n_iter>20
   disp('El numero de iteraciones debe ser menor o igual que 20');
   disp('Para mas informacion, consulta la ayuda');
   return
end

n_iter = n_iter*1000;
```

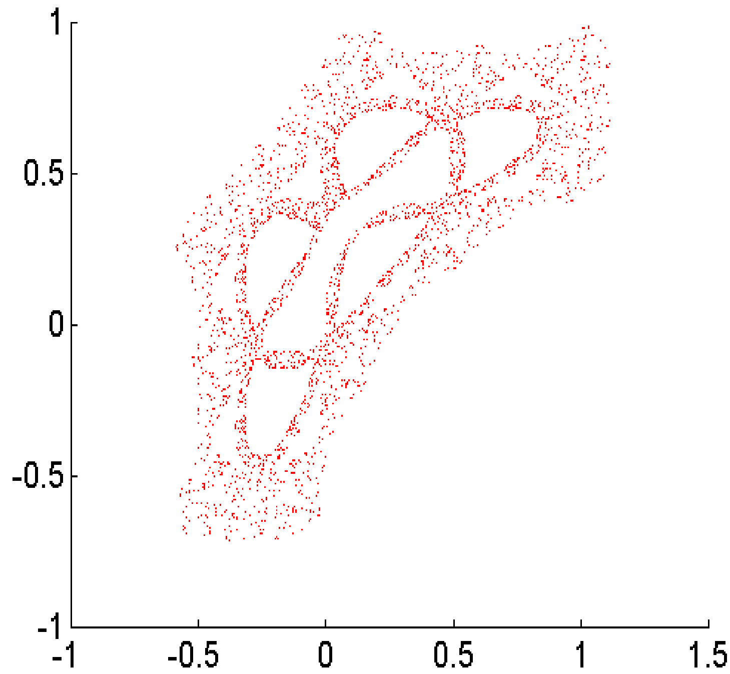


Figura 7.1: Hopalong con a=0.4, b=1, c=0, n_iter=3 (*1000=3000)

```
estilo = 'r';
x0 = 0;
y0 = 0;
hold on;

for cont=0:1:n_iter-1
    x1 = y0 - sign(x0)*sqrt(abs(b*x0-c));
    y1 = a - x0;
    x0 = x1;
    y0 = y1;
    plot(x1,y1,estilo);
end;

hold off;
```

Bibliografía

[1] Varios Autores. *Primeras jornadas docentes del Departamento de Matemática Aplicada.* Universidad Politécnica de Valencia, 2000.

[2] Ayuda on-line del programa MATLAB.