

Lo que no se puede ignorar

En este libro usaremos sistemáticamente conceptos matemáticos elementales que el lector o la lectora ya debería conocer, aunque podría no recordarlos inmediatamente.

Por consiguiente aprovecharemos este capítulo para refrescarlos y también para introducir nuevos conceptos que pertenecen al campo del Análisis Numérico. Empezaremos explorando su significado y utilidad con la ayuda de MATLAB (MATrix LABoratory), un entorno integrado para la programación y la visualización en cálculo científico. También usaremos GNU Octave (abreviadamente, Octave) que es en su mayor parte compatible con MATLAB. En las Secciones 1.6 y 1.7 daremos una rápida introducción a MATLAB y Octave, que es suficiente para el uso que vamos a hacer aquí. También incluimos algunas notas sobre las diferencias entre MATLAB y Octave que son relevantes para este libro. Sin embargo, remitimos a los lectores interesados al manual [HH05] para una descripción del lenguaje MATLAB y al manual [Eat02] para una descripción de Octave.

Octave es una reimplementación de parte de MATLAB que incluye una gran parte de los recursos numéricos de MATLAB y se distribuye libremente bajo la Licencia Pública General GNU.

A lo largo del texto haremos uso frecuente de la expresión “comando de MATLAB”; en ese caso, MATLAB debería ser entendido como el *lenguaje* que es el subconjunto común a ambos programas MATLAB y Octave.

Hemos procurado asegurar un uso transparente de nuestros códigos y programas bajo MATLAB y Octave. En los pocos casos en los que esto no se aplica, escribiremos una corta nota explicativa al final de la correspondiente sección.

En el presente Capítulo hemos condensado nociones que son típicas de cursos de Cálculo, Álgebra Lineal y Geometría, reformulándolas sin embargo de una forma apropiada para su uso en el cálculo científico.

1.1 Números reales

Mientras que el conjunto de los números reales \mathbb{R} es conocido por todo el mundo, la manera en la que los computadores los tratan es quizás menos conocida. Por una parte, puesto que las máquinas tienen recursos limitados, solamente se puede representar un subconjunto \mathbb{F} de dimensión finita de \mathbb{R} . Los números de este subconjunto se llaman *números de punto flotante*. Por otra parte, como veremos en la Sección 1.1.2, \mathbb{F} está caracterizado por propiedades que son diferentes de las de \mathbb{R} . La razón es que cualquier número real x es truncado, en principio, por la máquina dando origen a un nuevo número (llamado *número de punto flotante*), denotado por $fl(x)$, que no necesariamente coincide con el número original x .

1.1.1 Cómo representarlos

Para conocer la diferencia entre \mathbb{R} y \mathbb{F} , hagamos unos cuantos experimentos que ilustren la forma en que el computador (un PC por ejemplo) trata los números reales. Nótese que utilizar MATLAB u Octave en lugar de otro lenguaje es tan solo una cuestión de conveniencia. Los resultados de nuestro cálculo dependen, en efecto, primariamente de la manera en que el computador trabaja y sólo en menor medida del lenguaje de programación. Consideremos el número racional $x = 1/7$, cuya representación decimal es $0.\overline{142857}$. Ésta es una representación infinita, puesto que el número de cifras decimales es infinito. Para obtener su representación

>> en el computador, introducimos después del *prompt* (el símbolo >>) el cociente $1/7$ y obtenemos

```
>> 1/7
```

```
ans =  
0.1429
```

que es un número con sólo cuatro cifras decimales, siendo la última diferente de la cuarta cifra del número original.

Si ahora considerásemos $1/3$ encontraríamos 0.3333, así que la cuarta cifra decimal sería exacta. Este comportamiento se debe al hecho de que los números reales son *redondeados* por el computador. Esto significa, ante todo, que sólo se devuelve un número fijo a priori de cifras decimales, y además la última cifra decimal se incrementa en una unidad siempre y cuando la primera cifra decimal despreciada sea mayor o igual que 5.

La primera observación que debe hacerse es que usar sólo cuatro cifras decimales para representar los números reales es cuestionable. En efecto, la representación interna del número se hace con 16 cifras decimales, y lo que hemos visto es simplemente uno de los varios posibles formatos de salida de MATLAB. El mismo número puede tomar diferentes expresiones dependiendo de la declaración específica de formato que se haga.

Por ejemplo, para el número $1/7$, algunos posibles *formatos* de salida son:

<code>format long</code>	devuelve	0.14285714285714,	
<code>format short e</code>	"	$1.4286e - 01$,	
<code>format long e</code>	"	$1.428571428571428e - 01$,	
<code>format short g</code>	"	0.14286,	
<code>format long g</code>	"	0.142857142857143.	<code>format</code>

Algunos de ellos son más coherentes que otros con la representación interna del computador. En realidad, un computador almacena, en general, un número real de la forma siguiente

$$x = (-1)^s \cdot (0.a_1a_2 \dots a_t) \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}, \quad a_1 \neq 0 \quad (1.1)$$

donde s es 0 o 1, β (un entero positivo mayor o igual que 2) es la *base* adoptada por el computador específico que estemos manejando, m es un entero llamado *mantisa* cuya longitud t es el máximo número de cifras a_i (con $0 \leq a_i \leq \beta - 1$) que se almacenan, y e es un número entero llamado *exponente*. El formato `long e` es aquél que más se parece a esta representación y `e` representa el exponente; sus cifras, precedidas por el signo, se declaran a la derecha del carácter `e`. Los números cuyas formas se dan en (1.1) se llaman números de punto flotante, porque la posición de su punto decimal no es fija. Las cifras $a_1a_2 \dots a_p$ (con $p \leq t$) suelen llamarse p primeras cifras significativas de x .

La condición $a_1 \neq 0$ asegura que un número no puede tener múltiples representaciones. Por ejemplo, sin ésta restricción el número $1/10$ podría ser representado (en la base decimal) como $0.1 \cdot 10^0$, pero también como $0.01 \cdot 10^1$, etc.

Por consiguiente el conjunto \mathbb{F} está totalmente caracterizado por la base β , el número de cifras significativas t y el rango (L, U) (con $L < 0$ y $U > 0$) de variación del índice e . Por eso se denota por $\mathbb{F}(\beta, t, L, U)$. En MATLAB tenemos $\mathbb{F} = \mathbb{F}(2, 53, -1021, 1024)$ (en efecto, 53 cifras significativas en base 2 corresponden a los 15 cifras significativas que muestra MATLAB en base 10 con el `format long`).

Afortunadamente, el *error de redondeo* que se genera inevitablemente siempre que un número real $x \neq 0$ se reemplaza por su representante $fl(x)$ en \mathbb{F} , es pequeño, porque

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} \epsilon_M \quad (1.2)$$

donde $\epsilon_M = \beta^{1-t}$ proporciona la distancia entre 1 y el número en punto flotante mayor que 1 y más cercano a éste. Nótese que ϵ_M depende de β y t . En MATLAB ϵ_M puede obtenerse mediante el comando `eps`, y `eps`

se tiene $\epsilon_M = 2^{-52} \simeq 2.22 \cdot 10^{-16}$. Señalemos que en (1.2) estimamos el *error relativo* sobre x , que es indudablemente más significativo que el *error absoluto* $|x - fl(x)|$. En realidad, este último no tiene en cuenta el orden de magnitud de x mientras que el primero sí.

El número 0 no pertenece a \mathbb{F} , pues en tal caso tendríamos $a_1 = 0$ en (1.1); por tanto se maneja separadamente. Además, como L y U son finitos, uno no puede representar números cuyo valor absoluto sea arbitrariamente grande o arbitrariamente pequeño. Siendo más concretos, el número real positivo más grande y el más pequeño de \mathbb{F} vienen dados, respectivamente, por

$$x_{min} = \beta^{L-1}, \quad x_{max} = \beta^U (1 - \beta^{-t}).$$

En MATLAB estos valores pueden obtenerse mediante los comandos

`realmin` `realmin` y `realmax`, que producen

$$\begin{aligned} x_{min} &= 2.225073858507201 \cdot 10^{-308}, \\ x_{max} &= 1.7976931348623158 \cdot 10^{+308}. \end{aligned}$$

Un número positivo menor que x_{min} produce un mensaje de *underflow* y se trata como un cero o de una manera especial (véase, por ejemplo, [QSS06], Capítulo 2). Un número positivo mayor que x_{max} origina en cambio un mensaje de *overflow* y se almacena en la variable `Inf` (que es la representación en el computador de $+\infty$).

Los elementos de \mathbb{F} son más densos cerca de x_{min} y menos densos cuando se aproximan a x_{max} . En realidad, los números de \mathbb{F} más cercanos a x_{max} (a su izquierda) y a x_{min} (a su derecha) son, respectivamente,

$$\begin{aligned} x_{max}^- &= 1.7976931348623157 \cdot 10^{+308}, \\ x_{min}^+ &= 2.225073858507202 \cdot 10^{-308}. \end{aligned}$$

De este modo $x_{min}^+ - x_{min} \simeq 10^{-323}$, mientras que $x_{max} - x_{max}^- \simeq 10^{292}$ (!). Sin embargo, la distancia relativa es pequeña en ambos casos, como podemos deducir de (1.2).

1.1.2 Cómo operamos con números de punto flotante

Puesto que \mathbb{F} es un subconjunto propio de \mathbb{R} , las operaciones algebraicas elementales sobre números de punto flotante no gozan de todas las propiedades de las operaciones análogas en \mathbb{R} . Concretamente, la conmutatividad para la suma todavía se verifica (esto es $fl(x+y) = fl(y+x)$) así como para la multiplicación ($fl(xy) = fl(yx)$), pero se violan otras propiedades tales como la asociativa y la distributiva. Además, el 0 ya no es único. En efecto, asignemos a la variable `a` el valor 1, y ejecutemos las instrucciones siguientes:

```
>> a = 1; b=1; while a+b ~= a; b=b/2; end
```

La variable b se divide por dos en cada etapa, en tanto en cuanto la suma de a y b permanezca diferente (\neq) de a . Si operásemos sobre números reales, este programa nunca acabaría, mientras que, en nuestro caso, termina después de un número finito de pasos y devuelve el siguiente valor para b : $1.1102\text{e-}16 = \epsilon_M/2$. Por tanto, existe al menos un número b diferente de 0 tal que $a+b=a$. Esto es posible porque \mathbb{F} consta de números aislados; cuando se suman dos números a y b con $b < a$ y b menor que ϵ_M , siempre obtenemos que $a+b$ es igual a a . El número de MATLAB `a+eps` es el menor número de \mathbb{F} mayor que a . Así, la suma $a+b$ devolverá a para todo $b < \text{eps}$.

La asociatividad se viola siempre que ocurre una situación de *overflow* o *underflow*. Tomemos por ejemplo $a=1.0\text{e}+308$, $b=1.1\text{e}+308$ y $c=-1.001\text{e}+308$, y llevemos a cabo la suma de dos formas diferentes. Encontramos que

$$a + (b + c) = 1.0990\text{e} + 308, \quad (a + b) + c = \text{Inf}.$$

Este es un ejemplo particular de lo que ocurre cuando uno suma dos números con signos opuestos pero de valor absoluto similar. En este caso el resultado puede ser totalmente inexacto y nos referimos a tal situación como de *pérdida*, o *cancelación*, de *cifras significativas*. Por ejemplo, calculemos $((1+x)-1)/x$ (el resultado obvio es 1 para cualquier $x \neq 0$):

```
>> x = 1.e-15; ((1+x)-1)/x
```

```
ans = 1.1102
```

Este resultado es bastante impreciso, ¡el error relativo es superior al 11%!

Otro caso de cancelación numérica se encuentra cuando se evalúa la función

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \quad (1.3)$$

en 401 puntos equiespaciados con abscisas en $[1-2 \cdot 10^{-8}, 1+2 \cdot 10^{-8}]$. Obtenemos la gráfica caótica recogida en la Figura 1.1 (el comportamiento real es el de $(x-1)^7$, que es sustancialmente constante e igual a la función nula en tal diminuto entorno de $x = 1$). El comando de MATLAB que ha generado esta gráfica será ilustrado en la Sección 1.4.

Finalmente, es interesante observar que en \mathbb{F} no hay lugar para formas indeterminadas tales como $0/0$ o ∞/∞ . Su presencia produce lo que se llama *not a number* (`NaN` en MATLAB u Octave), al que no se aplican las reglas normales del cálculo.

Observación 1.1 Si bien es cierto que los errores de redondeo son normalmente pequeños, cuando se repiten dentro de largos y complejos algoritmos, pueden dar origen a efectos catastróficos. Dos casos destacados conciernen a la explosión del cohete Ariane el 4 de Junio de 1996, generada por un *overflow* en el computador de a bordo, y al fracaso de la misión de un misil americano



`NaN`

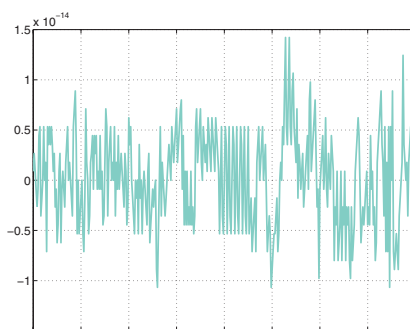


Figura 1.1. Comportamiento oscilatorio de la función (1.3) causado por los errores de cancelación

patriot, durante la guerra del Golfo en 1991, a causa de un error de redondeo en el cálculo de su trayectoria.

Un ejemplo con consecuencias menos catastróficas (pero todavía molestas) lo proporciona la sucesión

$$z_2 = 2, \quad z_{n+1} = 2^{n-1/2} \sqrt{1 - \sqrt{1 - 4^{1-n} z_n^2}}, \quad n = 2, 3, \dots \quad (1.4)$$

que converge a π cuando n tiende a infinito. Cuando se usa MATLAB para calcular z_n , el error relativo encontrado entre π y z_n decrece para las primeras 16 iteraciones, para crecer a continuación debido a los errores de redondeo (como se muestra en la Figura 1.2).



Véanse los ejercicios 1.1-1.2.

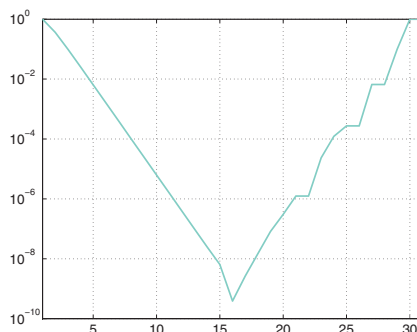


Figura 1.2. Logaritmo del error relativo $|\pi - z_n|/\pi$ frente a n

1.2 Números complejos

Los números complejos, cuyo conjunto se denota por \mathbb{C} , tienen la forma $z = x + iy$, donde $i = \sqrt{-1}$ es la unidad imaginaria (esto es $i^2 = -1$), mientras que $x = \text{Re}(z)$ e $y = \text{Im}(z)$ son las partes real e imaginaria de z , respectivamente. Generalmente se representan en el computador como pares de números reales.

Salvo que se redefinan de otra manera, las variables de MATLAB `i` y `j` denotan la unidad imaginaria. Para introducir un número complejo con parte real `x` y parte imaginaria `y`, uno puede escribir simplemente `x+i*y`; como alternativa, se puede utilizar el comando `complex(x,y)`. Mencionemos también las representaciones exponencial y trigonométrica de un número complejo z , que son equivalentes gracias a la *fórmula de Euler*

$$z = \rho e^{i\theta} = \rho(\cos \theta + i \sin \theta); \quad (1.5)$$

$\rho = \sqrt{x^2 + y^2}$ es el módulo del número complejo (puede obtenerse poniendo `abs(z)`) mientras que θ es su argumento, esto es el ángulo entre el eje x y la línea recta que sale del origen y pasa por el punto de coordenadas (x, y) en el plano complejo. θ puede hallarse tecleando `angle(z)`. Por consiguiente, la representación (1.5) es

$$\text{abs}(z) * (\cos(\text{angle}(z)) + i * \sin(\text{angle}(z))).$$

La representación polar gráfica de uno o más números complejos puede obtenerse mediante el comando `compass(z)`, donde `z` es un solo número complejo o un vector cuyas componentes son números complejos. Por ejemplo, tecleando

```
>> z = 3+i*3; compass(z);
```

se obtiene el gráfico mostrado en la Figura 1.3.

Para un número complejo dado `z`, se puede extraer su parte real con el comando `real(z)` y su parte imaginaria con `imag(z)`. Finalmente, el complejo conjugado $\bar{z} = x - iy$ de z , se puede obtener escribiendo simplemente `conj(z)`.

En MATLAB todas las operaciones se llevan a cabo suponiendo implícitamente que los operandos así como los resultados son complejos. Por tanto podemos encontrar algunos resultados aparentemente sorprendentes. Por ejemplo, si calculamos la raíz cúbica de -5 con el comando de MATLAB `(-5)^(1/3)`, en lugar de $-1.7099 \dots$ obtenemos el número complejo $0.8550 + 1.4809i$. (Anticipamos el uso del símbolo \wedge para el exponente de la potencia). En realidad, todos los números de la forma $\rho e^{i(\theta+2k\pi)}$, con k entero, son indistinguibles de $z = \rho e^{i\theta}$. Al calcular $\sqrt[3]{z}$ hallamos $\sqrt[3]{\rho} e^{i(\theta/3+2k\pi/3)}$, esto es, las tres raíces distintas

$$z_1 = \sqrt[3]{\rho} e^{i\theta/3}, \quad z_2 = \sqrt[3]{\rho} e^{i(\theta/3+2\pi/3)}, \quad z_3 = \sqrt[3]{\rho} e^{i(\theta/3+4\pi/3)}.$$

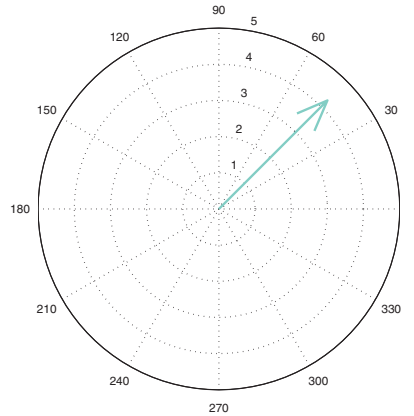


Figura 1.3. Resultado del comando de MATLAB `compass`

MATLAB seleccionará la primera que se encuentre recorriendo el plano complejo en sentido antihorario, empezando desde el eje real. Puesto que la representación polar de $z = -5$ es $\rho e^{i\theta}$ con $\rho = 5$ y $\theta = -\pi$, las tres raíces son (véase la Figura 1.4 para su representación en el plano de Gauss)

$$z_1 = \sqrt[3]{5}(\cos(-\pi/3) + i\sin(-\pi/3)) \simeq 0.8550 - 1.4809i,$$

$$z_2 = \sqrt[3]{5}(\cos(\pi/3) + i\sin(\pi/3)) \simeq 0.8550 + 1.4809i,$$

$$z_3 = \sqrt[3]{5}(\cos(-\pi) + i\sin(-\pi)) \simeq -1.7100.$$

La segunda raíz es la seleccionada.

Finalmente, por (1.5) obtenemos

$$\cos(\theta) = \frac{1}{2} (e^{i\theta} + e^{-i\theta}), \quad \sin(\theta) = \frac{1}{2i} (e^{i\theta} - e^{-i\theta}). \quad (1.6)$$

Octave 1.1 El comando `compass` no está disponible en Octave, sin embargo puede ser emulado con la siguiente función:

```
function octcompass(z) xx = [0 1 .8 1 .8].';
yy = [0 0 .08 0 -.08].';
arrow = xx + yy.*sqrt(-1);
z = arrow * z;
[th,r]=cart2pol(real(z),imag(z));
polar(th,r);
return
```

■

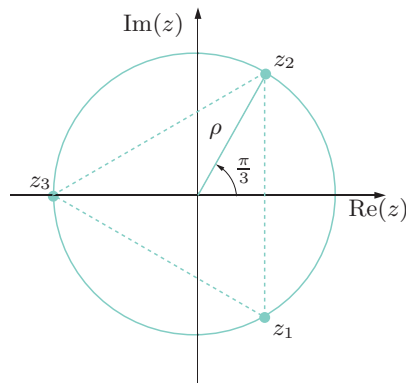


Figura 1.4. Representación en el plano complejo de las tres raíces cúbicas complejas del número real -5

1.3 Matrices

Sean n y m enteros positivos. Una matriz con m filas y n columnas es un conjunto de $m \times n$ elementos a_{ij} , con $i = 1, \dots, m$, $j = 1, \dots, n$, representado mediante la siguiente tabla:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}. \quad (1.7)$$

En forma compacta escribimos $A = (a_{ij})$. Si los elementos de A fuesen números reales, escribiríamos $A \in \mathbb{R}^{m \times n}$, y $A \in \mathbb{C}^{m \times n}$ si fuesen complejos.

Las matrices cuadradas de dimensión n son aquéllas con $m = n$. Una matriz con una sola columna es un *vector columna*, mientras que una matriz con una sola fila es un *vector fila*.

Para introducir una matriz en MATLAB uno tiene que escribir los elementos de la primera fila a la última, introduciendo el carácter `;` para separar las diferentes filas. Por ejemplo, el comando

```
>> A = [ 1 2 3; 4 5 6]
```

devuelve

```
A =
     1     2     3
     4     5     6
```

esto es, una matriz 2×3 cuyos elementos se indican arriba. La matriz $m \times n$ `zeros(m,n)` tiene todos los elementos nulos, `eye(m,n)` tiene todos

`zeros`
`eye`

los elementos nulos salvo a_{ii} , $i = 1, \dots, \min(m, n)$, en la diagonal donde todos son iguales a 1. La matriz identidad $n \times n$ se obtiene con el comando `eye(n)`: sus elementos son $\delta_{ij} = 1$ si $i = j$, 0 en caso contrario, para $i, j = 1, \dots, n$. Finalmente, mediante el comando `A=[]` podemos inicializar una matriz vacía.

Recordamos las siguientes operaciones matriciales:

1. si $A = (a_{ij})$ y $B = (b_{ij})$ son matrices $m \times n$, la *suma* de A y B es la matriz $A + B = (a_{ij} + b_{ij})$;
2. el *producto* de una matriz A por un número real o complejo λ es la matriz $\lambda A = (\lambda a_{ij})$;
3. el *producto* de dos matrices es posible sólo para tamaños compatibles, concretamente, si A es $m \times p$ y B es $p \times n$, para algún entero positivo p . En tal caso $C = AB$ es una matriz $m \times n$ cuyos elementos son

$$c_{ij} = \sum_{k=1}^p a_{ik} b_{kj}, \quad \text{para } i = 1, \dots, m, \quad j = 1, \dots, n.$$

He aquí un ejemplo de suma y producto de dos matrices.

```
>> A=[1 2 3; 4 5 6];
>> B=[7 8 9; 10 11 12];
>> C=[13 14; 15 16; 17 18];
>> A+B
```

```
ans =
      8      10      12
     14      16      18
```

```
>> A*C
```

```
ans =
      94      100
     229      244
```

Nótese que MATLAB devuelve un mensaje diagnóstico cuando uno trata de llevar a cabo operaciones sobre matrices con dimensiones incompatibles. Por ejemplo:

```
>> A=[1 2 3; 4 5 6];
>> B=[7 8 9; 10 11 12];
>> C=[13 14; 15 16; 17 18];
>> A+C
```

```
??? Error using ==> + Matrix dimensions must agree.
```

```
>> A*B
```

??? Error using ==> * Inner matrix dimensions must agree.

Si A es una matriz cuadrada de dimensión n , su *inversa* (si existe) es una matriz cuadrada de dimensión n , denotada por A^{-1} , que satisface la relación matricial $AA^{-1} = A^{-1}A = I$. Podemos obtener A^{-1} mediante el comando `inv(A)`. La inversa de A existe si y sólo si el *determinante* de A , un número denotado por $\det(A)$, es no nulo. La última condición se satisface si y sólo si los vectores columna de A son linealmente independientes (véase la Sección 1.3.1). El determinante de una matriz cuadrada se define mediante la siguiente fórmula recursiva (*regla de Laplace*):

$$\det(A) = \begin{cases} a_{11} & \text{si } n = 1, \\ \sum_{j=1}^n \Delta_{ij} a_{ij}, & \text{para } n > 1, \forall i = 1, \dots, n, \end{cases} \quad (1.8)$$

donde $\Delta_{ij} = (-1)^{i+j} \det(A_{ij})$ y A_{ij} es la matriz obtenida eliminando la i -ésima fila y la j -ésima columna de la matriz A . (El resultado es independiente del índice de la fila i .)

En particular, si $A \in \mathbb{R}^{1 \times 1}$ ponemos $\det(A) = a_{11}$; si $A \in \mathbb{R}^{2 \times 2}$ se tiene

$$\det(A) = a_{11}a_{22} - a_{12}a_{21};$$

si $A \in \mathbb{R}^{3 \times 3}$ obtenemos

$$\begin{aligned} \det(A) &= a_{11}a_{22}a_{33} + a_{31}a_{12}a_{23} + a_{21}a_{13}a_{32} \\ &\quad - a_{11}a_{23}a_{32} - a_{21}a_{12}a_{33} - a_{31}a_{13}a_{22}. \end{aligned}$$

Finalmente, si $A = BC$, entonces $\det(A) = \det(B)\det(C)$.

Para invertir una matriz 2×2 y calcular su determinante podemos proceder como sigue:

```
>> A=[1 2; 3 4];
>> inv(A)

ans =
    -2.0000    1.0000
     1.5000   -0.5000

>> det(A)

ans =
    -2
```

Si una matriz fuese singular, MATLAB devolvería un mensaje diagnóstico, seguido por una matriz cuyos elementos son todos iguales a `Inf`, como se ilustra en el ejemplo siguiente:

```
>> A=[1 2; 0 0];
>> inv(A)
```

```
Warning: Matrix is singular to working precision.
```

```
ans =
     Inf     Inf
     Inf     Inf
```

Para clases especiales de matrices cuadradas, el cálculo de inversas y determinantes es bastante sencillo. En particular, si A es una *matriz diagonal*, es decir, una matriz para la que sólo son no nulos los elementos diagonales a_{kk} , $k = 1, \dots, n$, su determinante viene dado por $\det(A) = a_{11}a_{22} \cdots a_{nn}$. En particular, A es no singular si y sólo si $a_{kk} \neq 0$ para todo k . En tal caso la inversa de A todavía es diagonal con elementos a_{kk}^{-1} .

diag Sea \mathbf{v} un vector de dimensión n . El comando **diag(v)** produce una matriz diagonal cuyos elementos son las componentes del vector \mathbf{v} . El comando más general **diag(v,m)** produce una matriz cuadrada de dimensión $n+\text{abs}(m)$ cuya m -ésima diagonal superior (es decir, la diagonal de los elementos con índices $i, i+m$) tiene elementos iguales a las componentes de \mathbf{v} , mientras que los restantes elementos son nulos. Nótese que esta extensión es válida también cuando m es negativo, en cuyo caso los únicos elementos afectados son los de las diagonales inferiores.

Por ejemplo, si $\mathbf{v} = [1 \ 2 \ 3]$ entonces:

```
>> A=diag(v,-1)
```

```
A =
     0     0     0     0
     1     0     0     0
     0     2     0     0
     0     0     3     0
```

Otros casos especiales son las matrices *triangulares superiores* y las *triangulares inferiores*. Una matriz cuadrada de dimensión n es *triangular inferior* (respectivamente, *superior*) si todos los elementos por encima (respectivamente, por debajo) de la diagonal principal son cero. Su determinante es simplemente el producto de los elementos diagonales.

tril Mediante los comandos **tril(A)** y **triu(A)**, uno puede extraer de la
triu matriz A de dimensión n sus partes inferior y superior. Sus extensiones **tril(A,m)** o **triu(A,m)**, con m recorriendo de $-n$ a n , permiten la extracción de las partes triangulares aumentadas por, o privadas de, m extradiagonales.

Por ejemplo, dada la matriz $A = [3 \ 1 \ 2; -1 \ 3 \ 4; -2 \ -1 \ 3]$, mediante el comando **L1=tril(A)** obtenemos

```
L1 =
      3      0      0
     -1      3      0
     -2     -1      3
```

mientras que, mediante $L2=\text{tril}(A,1)$, obtenemos

```
L2 =
      3      1      0
     -1      3      4
     -2     -1      3
```

Finalmente, recordamos que si $A \in \mathbb{R}^{m \times n}$ su traspuesta $A^T \in \mathbb{R}^{n \times m}$ es la matriz obtenida intercambiando filas y columnas de A . Cuando $A = A^T$ la matriz A se dice *simétrica*. Finalmente, A' denota la traspuesta de A , si A es real, o su traspuesta conjugada, esto es, A^H , si A es compleja. Una matriz cuadrada compleja que coincide con su traspuesta conjugada A^H se llama *hermitiana*.

Se utiliza una notación similar, \mathbf{v}' , para el traspuesto conjugado \mathbf{v}^H del vector \mathbf{v} . Si v_i denota las componentes de \mathbf{v} , el vector adjunto \mathbf{v}^H es un vector fila cuyas componentes son los complejos conjugados \bar{v}_i de v_i .

Octave 1.2 Octave también devuelve un mensaje diagnóstico cuando uno trata de llevar a cabo operaciones sobre matrices que tienen dimensiones incompatibles. Si repetimos los ejemplos de MATLAB previos, obtenemos:

```
octave:1> A=[1 2 3; 4 5 6];
octave:2> B=[7 8 9; 10 11 12];
octave:3> C=[13 14; 15 16; 17 18];
octave:4> A+C
```

```
error: operator +: nonconformant arguments (op1 is
2x3, op2 is 3x2)
error: evaluating binary operator '+' near line 2,
column 2
```

```
octave:5> A*B
```

```
error: operator *: nonconformant arguments (op1 is
2x3, op2 is 2x3)
error: evaluating binary operator '*' near line 2,
column 2
```

Si A es singular, Octave devuelve un mensaje diagnóstico seguido por la matriz a invertir, como se ilustra en el siguiente ejemplo:

```
octave:1> A=[1 2; 0 0];
octave:2> inv(A)
```

```
warning: inverse: singular matrix to machine
precision, rcond = 0
ans =
    1    2
    0    0
```



1.3.1 Vectores

Los vectores serán indicados en negrita; con más precisión, \mathbf{v} denotará un vector columna cuya i -ésima componente es v_i . Cuando todas las componentes son números reales podemos escribir $\mathbf{v} \in \mathbb{R}^n$.

En MATLAB, los vectores se consideran como casos particulares de matrices. Para introducir un vector columna uno tiene que insertar entre corchetes los valores de sus componentes separadas por punto y coma, mientras que para un vector fila basta con escribir los valores de las componentes separados por blancos o comas. Por ejemplo, mediante las instrucciones $\mathbf{v} = [1;2;3]$ y $\mathbf{w} = [1 \ 2 \ 3]$ inicializamos el vector columna \mathbf{v} y el vector fila \mathbf{w} , ambos de dimensión 3. El comando **zeros** `zeros(n,1)` (respectivamente, `zeros(1,n)`) produce un vector columna (respectivamente, fila) de dimensión n con elementos nulos, que denotaremos por $\mathbf{0}$. Análogamente, el comando **ones** `ones(n,1)` genera el vector columna, denotado por $\mathbf{1}$, cuyas componentes son todas iguales a 1.

Un sistema de vectores $\{\mathbf{y}_1, \dots, \mathbf{y}_m\}$ es *linealmente independiente* si la relación

$$\alpha_1 \mathbf{y}_1 + \dots + \alpha_m \mathbf{y}_m = \mathbf{0}$$

implica que todos los coeficientes $\alpha_1, \dots, \alpha_m$ son nulos. Un sistema $\mathcal{B} = \{\mathbf{y}_1, \dots, \mathbf{y}_n\}$ de n vectores linealmente independientes en \mathbb{R}^n (o \mathbb{C}^n) es una *base* para \mathbb{R}^n (o \mathbb{C}^n), esto es, cualquier vector \mathbf{w} en \mathbb{R}^n puede escribirse como combinación lineal de los elementos de \mathcal{B} ,

$$\mathbf{w} = \sum_{k=1}^n w_k \mathbf{y}_k,$$

para una elección única posible de los coeficientes $\{w_k\}$. Estos últimos se llaman *componentes* de \mathbf{w} con respecto a la base \mathcal{B} . Por ejemplo, la base canónica de \mathbb{R}^n es el conjunto de vectores $\{\mathbf{e}_1, \dots, \mathbf{e}_n\}$, donde \mathbf{e}_i tiene su i -ésima componente igual a 1 y todas las otras componentes iguales a 0, y es la que se usa normalmente.

El *producto escalar* de dos vectores $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ se define como

$$(\mathbf{v}, \mathbf{w}) = \mathbf{w}^T \mathbf{v} = \sum_{k=1}^n v_k w_k,$$

siendo $\{v_k\}$ y $\{w_k\}$ las componentes de \mathbf{v} y \mathbf{w} , respectivamente. El correspondiente comando es $\mathbf{w}' * \mathbf{v}$ o también `dot(v,w)`, donde ahora la prima denota trasposición de un vector. La longitud (o módulo) de un vector \mathbf{v} viene dada por

$$\|\mathbf{v}\| = \sqrt{(\mathbf{v}, \mathbf{v})} = \sqrt{\sum_{k=1}^n v_k^2}$$

y puede calcularse mediante el comando `norm(v)`.

El producto vectorial de dos vectores $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $n \geq 3$, $\mathbf{v} \times \mathbf{w}$ o $\mathbf{v} \wedge \mathbf{w}$, es el vector $\mathbf{u} \in \mathbb{R}^n$ ortogonal a ambos, \mathbf{v} y \mathbf{w} , cuyo módulo es $|\mathbf{u}| = |\mathbf{v}| |\mathbf{w}| \sin(\alpha)$, donde α es el ángulo formado por \mathbf{v} y \mathbf{w} . Puede obtenerse por medio del comando `cross(v,w)`.

La visualización de un vector se consigue mediante el comando de MATLAB `quiver` en \mathbb{R}^2 y `quiver3` en \mathbb{R}^3 .

El comando de MATLAB `x.*y` o `x.^2` indica que estas operaciones se llevarían a cabo componente a componente. Por ejemplo si definimos los vectores

```
>> v = [1; 2; 3]; w = [4; 5; 6];
```

la instrucción

```
>> w'*v
```

```
ans =
    32
```

proporciona su producto escalar, mientras que

```
>> w.*v
```

```
ans =
     4
    10
    18
```

devuelve un vector cuya i -ésima componente es igual a $x_i y_i$.

Finalmente, recordamos que un vector $\mathbf{v} \in \mathbb{C}^n$, con $\mathbf{v} \neq \mathbf{0}$, es un *autovector* de una matriz $A \in \mathbb{C}^{n \times n}$ asociado al número complejo λ si

$$A\mathbf{v} = \lambda\mathbf{v}.$$

El número complejo λ se llama *autovalor* de A . En general, el cálculo de autovalores es bastante difícil. Casos excepcionales corresponden a la matrices diagonales y triangulares, cuyos autovalores son sus elementos diagonales.

Véanse los ejercicios 1.3-1.6.



dot

norm

cross

quiver

quiver3

.*
.^

1.4 Funciones reales

fplot Este capítulo tratará de la manipulación de funciones reales definidas sobre un intervalo (a, b) . El comando **fplot(fun,lims)** dibuja la gráfica de la función **fun** (que se almacena como una cadena de caracteres) sobre el intervalo $(\text{lims}(1), \text{lims}(2))$. Por ejemplo, para representar $f(x) = 1/(1+x^2)$ sobre el intervalo $(-5, 5)$, podemos escribir

```
>> fun = '1/(1+x.^2)'; lims=[-5,5]; fplot(fun,lims);
```

o, más directamente,

```
>> fplot('1/(1+x.^2)', [-5 5]);
```

MATLAB obtiene la gráfica muestreando la función sobre un conjunto de abscisas no equiespaciadas y reproduce la verdadera gráfica de f con una tolerancia de 0.2%. Para mejorar la precisión podríamos usar el comando

```
>> fplot(fun,lims,tol,n,'LineStyle',P1,P2,...).
```

donde **tol** indica la tolerancia deseada y el parámetro **n** (≥ 1) asegura que la función será dibujada con un mínimo de **n** + 1 puntos. **LineStyle** es una cadena de caracteres que especifica el estilo o el color de la línea utilizada para hacer la gráfica. Por ejemplo, **LineStyle='--'** se utiliza para una línea discontinua, **LineStyle='r-.'** para una línea roja de puntos y trazos, etc. Para usar valores por defecto para **tol**, **n** o **LineStyle** se pueden pasar matrices vacías (`[]`).

eval Para evaluar una función **fun** en un punto **x** escribimos **y=eval(fun)**, después de haber inicializado **x**. El valor correspondiente se almacena en **y**. Nótese que **x** y **y**, por tanto, pueden ser vectores. Cuando se usa este comando, la restricción es que el argumento de la función **fun** debe ser **x**. Cuando el argumento de **fun** tenga un nombre diferente (este caso es frecuente si este argumento se genera en el interior de un programa), el comando **eval** se reemplazará por **feval** (véase la Observación 1.3).

grid Finalmente señalamos que si se escribe **grid on** después del comando **fplot**, podemos obtener una rejilla de fondo como en la Figura 1.1.

Octave 1.3 En Octave, usando el comando **fplot(fun,lims,n)** la gráfica se obtiene muestreando la función definida en **fun** (ese es el nombre de una *function* o una expresión que contenga a **x**) sobre un conjunto de abscisas no equiespaciadas. El parámetro opcional **n** (≥ 1) asegura que la función será dibujada con un mínimo de **n**+1 puntos. Por ejemplo, para representar $f(x) = 1/(1+x^2)$ usamos los comandos siguientes:

```
>> fun = '1./(1+x.^2)'; lims=[-5,5];
>> fplot(fun,lims)
```



1.4.1 Los ceros

Recordamos que si $f(\alpha) = 0$, α se llama *cero* de f o *raíz* de la ecuación $f(x) = 0$. Un cero es *simple* si $f'(\alpha) \neq 0$, y *múltiple* en caso contrario.

De la gráfica de una función se puede inferir (dentro de cierta tolerancia) cuáles son sus ceros reales. El cálculo directo de todos los ceros de una función dada no siempre es posible. Para funciones que son polinomios de grado n con coeficientes reales, es decir, de la forma

$$p_n(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n = \sum_{k=0}^n a_kx^k, \quad a_k \in \mathbb{R}, \quad a_n \neq 0,$$

podemos obtener el único cero $\alpha = -a_0/a_1$, cuando $n = 1$ (es decir p_1 representa una línea recta), o los dos ceros, α_+ y α_- , cuando $n = 2$ (esta vez p_2 representa una parábola) $\alpha_{\pm} = (-a_1 \pm \sqrt{a_1^2 - 4a_0a_2})/(2a_2)$.

Sin embargo, no hay fórmulas explícitas para los ceros de un polinomio arbitrario p_n cuando $n \geq 5$.

En lo que sigue denotaremos por \mathbb{P}_n el espacio de polinomios de grado menor o igual que n ,

$$p_n(x) = \sum_{k=0}^n a_kx^k \quad (1.9)$$

donde los a_k son coeficientes dados, reales o complejos.

Tampoco el número de ceros de una función puede ser, en general, determinado *a priori*. Una excepción la proporcionan los polinomios, para los cuales el número de ceros (reales o complejos) coincide con el grado del polinomio. Además, si $\alpha = x + iy$ con $y \neq 0$ fuese un cero de un polinomio de grado $n \geq 2$ con coeficientes reales, su complejo conjugado $\bar{\alpha} = x - iy$ también sería un cero.

Para calcular en MATLAB un cero de una función **fun**, cerca de un valor **x0**, real o complejo, se puede utilizar el comando **fzero(fun,x0)**. El resultado es un valor aproximado del cero deseado, y también el intervalo en el que se hizo la búsqueda. Alternativamente, usando el comando **fzero(fun,[x0 x1])**, se busca un cero de **fun** en el intervalo cuyos extremos son **x0,x1**, con tal de que f cambie de signo entre **x0** y **x1**.

Consideremos, por ejemplo, la función $f(x) = x^2 - 1 + e^x$. Observando su gráfica se ve que existen dos ceros en $(-1, 1)$. Para calcularlos necesitamos ejecutar los comandos siguientes:

```
fun=inline('x^2 - 1 + exp(x)','x')
fzero(fun,1)
```

```
ans =
    5.4422e-18
```

```
fzero(fun,-1)
```

```
ans =
    -0.7146
```

Alternativamente, después de observar en la gráfica de la función que existe un cero en el intervalo $[-1, -0.2]$ y otro en $[-0.2, 1]$, podríamos haber escrito

```
fzero(fun, [-0.2 1])
```

```
ans =
    -5.2609e-17
```

```
fzero(fun, [-1 -0.2])
```

```
ans =
    -0.7146
```

El resultado obtenido para el primer cero es ligeramente distinto del obtenido previamente, debido a una diferente inicialización del algoritmo implementado en `fzero`.

En el Capítulo 2 introduciremos y estudiaremos varios métodos para el cálculo aproximado de los ceros de una función arbitraria.

Octave 1.4 En Octave, `fzero` sólo acepta funciones definidas utilizando la palabra clave `function` y su correspondiente sintaxis es como sigue:

```
function y = fun(x)
    y = x.^2 - 1 + exp(x);
end
```

```
fzero("fun", 1)
```

```
ans = 2.3762e-17
```

```
fzero("fun", -1)
```

```
ans = -0.71456
```



1.4.2 Polinomios

polyval Los polinomios son funciones muy especiales y hay una *toolbox*¹ especial en MATLAB, `polyfun`, para su tratamiento. El comando `polyval` es apto para evaluar un polinomio en uno o varios puntos. Sus argumentos de entrada son un vector `p` y un vector `x`, donde las componentes de `p` son los coeficientes del polinomio almacenados en orden decreciente, desde a_n hasta a_0 , y las componentes de `x` son las abscisas donde el polinomio necesita ser evaluado. El resultado puede almacenarse en un vector `y` escribiendo

¹ Una *toolbox* es una colección de funciones MATLAB de propósito especial.

```
>> y = polyval(p,x)
```

Por ejemplo, los valores de $p(x) = x^7 + 3x^2 - 1$, en las abscisas equiespaciadas $x_k = -1 + k/4$ para $k = 0, \dots, 8$, pueden obtenerse procediendo como sigue:

```
>> p = [1 0 0 0 0 3 0 -1]; x = [-1:0.25:1];
>> y = polyval(p,x)
```

```
y =
Columns 1 through 5:

    1.00000    0.55402   -0.25781   -0.81256   -1.00000
Columns 6 through 9:

   -0.81244   -0.24219    0.82098    3.00000
```

Alternativamente, se podría usar el comando `feval`. Sin embargo, en tal caso uno debería dar la expresión analítica entera del polinomio en la cadena de caracteres de entrada, y no simplemente los coeficientes.

El programa `roots` proporciona una aproximación de los ceros de un polinomio y sólo requiere la introducción del vector `p`. roots

Por ejemplo, podemos calcular los ceros de $p(x) = x^3 - 6x^2 + 11x - 6$ escribiendo

```
>> p = [1 -6 11 -6]; format long;
>> roots(p)
```

```
ans =
    3.000000000000000
    2.000000000000000
    1.000000000000000
```

Desafortunadamente, el resultado no siempre tiene tanta precisión. Por ejemplo, para el polinomio $p(x) = (x+1)^7$, cuyo único cero es $\alpha = -1$ con multiplicidad 7, encontramos (de manera bastante sorprendente)

```
>> p = [1 7 21 35 35 21 7 1];
>> roots(p)
```

```
ans =
   -1.0101
  -1.0063 + 0.0079i
  -1.0063 - 0.0079i
  -0.9977 + 0.0099i
  -0.9977 - 0.0099i
  -0.9909 + 0.0044i
  -0.9909 - 0.0044i
```



De hecho, los métodos numéricos para el cálculo de las raíces de un polinomio con multiplicidad mayor que uno están particularmente sujetos a errores de redondeo (véase la Sección 2.5.2).

conv El comando `p=conv(p1,p2)` devuelve los coeficientes del polinomio dado por el producto de dos polinomios cuyos coeficientes están contenidos en los vectores `p1` y `p2`.

deconv Análogamente, el comando `[q,r]=deconv(p1,p2)` suministra los coeficientes del polinomio obtenido dividiendo `p1` por `p2`, es decir, `p1 = conv(p2,q) + r`. En otras palabras, `q` y `r` son el cociente y el resto de la división.

Consideremos por ejemplo el producto y el cociente de los dos polinomios $p_1(x) = x^4 - 1$ y $p_2(x) = x^3 - 1$:

```
>> p1 = [1 0 0 0 -1];
>> p2 = [1 0 0 -1];
>> p=conv(p1,p2)
```

```
p =
     1     0     0    -1    -1     0     0     1
```

```
>> [q,r]=deconv(p1,p2)
```

```
q =
     1     0
r =
     0     0     0     1    -1
```

Por consiguiente encontramos los polinomios $p(x) = p_1(x)p_2(x) = x^7 - x^4 - x^3 + 1$, $q(x) = x$ y $r(x) = x - 1$ tales que $p_1(x) = q(x)p_2(x) + r(x)$.

polyint Los comandos `polyint(p)` y `polyder(p)` proporcionan, respectivamente, los coeficientes de la primitiva (que se anula en $x = 0$) y los de la derivada del polinomio cuyos coeficientes están dados por las componentes del vector `p`.

Si `x` es un vector de abscisas y `p` (respectivamente, `p1` y `p2`) es un vector que contiene los coeficientes de un polinomio p (respectivamente, p_1 y p_2), los comandos previos se resumen en la Tabla 1.1.

polyfit Un comando adicional, `polyfit`, permite el cálculo de los $n + 1$ coeficientes de un polinomio p de grado n una vez que se dispone de los valores de p en $n + 1$ nudos distintos (véase la Sección 3.1.1).

polyderiv **Octave 1.5** Los comandos `polyderiv` y `polyinteg` tienen la misma funcionalidad que `polyder` y `polyfit`, respectivamente. Nótese que el comando `polyder` está disponible también en el repositorio de Octave, véase la Sección 1.6. ■

comando	proporciona
<code>y=polyval(p,x)</code>	y = valores de $p(x)$
<code>z=roots(p)</code>	z = raíces de p tales que $p(z) = 0$
<code>p=conv(p1,p2)</code>	p = coeficientes del polinomio $p_1 p_2$
<code>[q,r]=deconv(p1,p2)</code>	q = coeficientes de q , r = coeficientes de r tales que $p_1 = qp_2 + r$
<code>y=polyder(p)</code>	y = coeficientes de $p'(x)$
<code>y=polyint(p)</code>	y = coeficientes de $\int_0^x p(t) dt$

Tabla 1.1. Comandos de MATLAB para operaciones con polinomios

1.4.3 Integración y diferenciación

Los dos resultados siguientes serán invocados a menudo a lo largo de este libro:

1. *teorema fundamental de integración*: si f es una función continua en $[a, b)$, entonces

$$F(x) = \int_a^x f(t) dt \quad \forall x \in [a, b),$$

es una función diferenciable, llamada una *primitiva* de f , que satisface,

$$F'(x) = f(x) \quad \forall x \in [a, b);$$

2. *primer teorema del valor medio para integrales*: si f es una función continua en $[a, b)$ y $x_1, x_2 \in [a, b)$ con $x_1 < x_2$, entonces $\exists \xi \in (x_1, x_2)$ tal que

$$f(\xi) = \frac{1}{x_2 - x_1} \int_{x_1}^{x_2} f(t) dt.$$

Aun cuando exista, una primitiva podría ser imposible de determinar o difícil de calcular. Por ejemplo, saber que $\ln|x|$ es una primitiva de $1/x$ es irrelevante si uno no sabe cómo calcular eficientemente los logaritmos. En el Capítulo 4 introduciremos varios métodos para calcular la integral de una función continua arbitraria con una precisión deseada, independientemente del conocimiento de su primitiva.

Recordamos que una función f definida en un intervalo $[a, b]$ es diferenciable en un punto $\bar{x} \in (a, b)$ si existe el siguiente límite

$$f'(\bar{x}) = \lim_{h \rightarrow 0} \frac{1}{h} (f(\bar{x} + h) - f(\bar{x})). \quad (1.10)$$

El valor de $f'(\bar{x})$ proporciona la pendiente de la recta tangente a la gráfica de f en el punto \bar{x} .

Decimos que una función, continua junto con su derivada en todo punto de $[a, b]$, pertenece al espacio $C^1([a, b])$. Con más generalidad, una función con derivadas continuas hasta el orden p (un entero positivo) se dice que pertenece a $C^p([a, b])$. En particular, $C^0([a, b])$ denota el espacio de las funciones continuas en $[a, b]$.

Un resultado que será usado a menudo es el *teorema del valor medio*, de acuerdo con el cual, si $f \in C^1([a, b])$, existe $\xi \in (a, b)$ tal que

$$f'(\xi) = (f(b) - f(a)) / (b - a).$$

Finalmente, merece la pena recordar que una función que es continua junto con todas sus derivadas hasta el orden $n + 1$ en el entorno de x_0 , puede ser aproximada en tal entorno por el llamado *polinomio de Taylor de grado n* en el punto x_0 :

$$\begin{aligned} T_n(x) &= f(x_0) + (x - x_0)f'(x_0) + \dots + \frac{1}{n!}(x - x_0)^n f^{(n)}(x_0) \\ &= \sum_{k=0}^n \frac{(x - x_0)^k}{k!} f^{(k)}(x_0). \end{aligned}$$

diff La *toolbox* de MATLAB **symbolic** proporciona los comandos **diff**,
int **int** y **taylor** que nos permiten obtener la expresión analítica de la
taylor derivada, la integral indefinida (es decir una primitiva) y el polinomio de Taylor, respectivamente, de una función dada. En particular, habiendo definido en la cadena de caracteres **f** la función sobre la que estamos interesados en operar, **diff(f,n)** proporciona su derivada de orden **n**, **int(f)** su integral indefinida, y **taylor(f,x,n+1)** el polinomio de Taylor asociado de grado **n** en un entorno de $x_0 = 0$. La variable **x** debe ser declarada *symbolic* usando el comando **syms x**. Esto permitirá su manipulación algebraica sin especificar su valor.

Para hacer esto para la función $f(x) = (x^2 + 2x + 2)/(x^2 - 1)$, procedemos como sigue:

```
>> f = '(x^2+2*x+2)/(x^2-1)';
>> syms x
>> diff(f)

(2*x+2)/(x^2-1)-2*(x^2+2*x+2)/(x^2-1)^2*x

>> int(f)

x+5/2*log(x-1)-1/2*log(1+x)
```

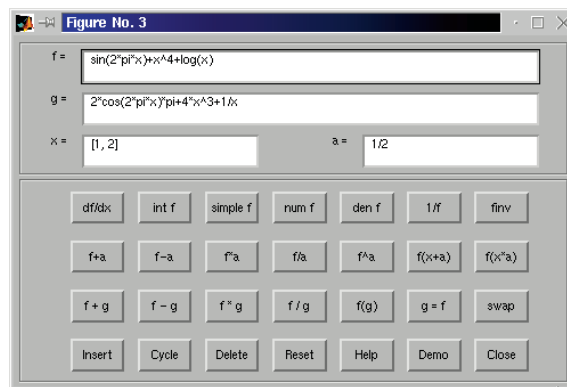


Figura 1.5. Interfaz gráfico del comando funtool

```
>> taylor(f,x,6)
```

```
-2-2*x-3*x^2-2*x^3-3*x^4-2*x^5
```

Observamos que usando el comando `simple` es posible simplificar las expresiones generadas por `diff`, `int` y `taylor` con objeto de hacerlas lo más sencillas posible. El comando `funtool`, mediante el interfaz gráfico

`simple`

`funtool`

Octave 1.6 Los cálculos simbólicos todavía no están disponibles en Octave, aunque existe trabajo en curso al respecto.² ■



Véanse los ejercicios 1.7-1.8.

1.5 Error no sólo es humano

En realidad, parafraseando el lema latino *Errare humanum est*, podríamos decir que en el cálculo numérico el error es incluso inevitable.

Como hemos visto, el simple hecho de usar un computador para representar los números reales introduce errores. Por consiguiente, lo importante es no tanto esforzarse por eliminar los errores, sino más bien ser capaces de controlar sus efectos.

Hablando en general, podemos identificar varios niveles de errores que ocurren durante la aproximación y resolución de un problema físico (véase la Figura 1.6).

² <http://www.octave.org>

En el nivel superior se sitúa el error e_m que ocurre cuando se fuerza la realidad física (PF significa el problema físico y x_f denota su solución) para obedecer a cierto modelo matemático (MM , cuya solución es x). Tales errores limitarán la aplicabilidad del modelo matemático a ciertas situaciones que se sitúan más allá del control del Cálculo Científico.

El modelo matemático (expresado por una integral como en el ejemplo de la Figura 1.6, una ecuación algebraica o diferencial, un sistema lineal o no lineal) generalmente no es resoluble en forma explícita. Su resolución mediante algoritmos computacionales seguramente involucrará la introducción y propagación de errores de redondeo, como mínimo. Llamemos e_a a estos errores.

Por otra parte, a menudo es necesario introducir errores adicionales puesto que cualquier procedimiento de resolución del modelo matemático que involucre una sucesión infinita de operaciones aritméticas no puede realizarse por el computador salvo de manera aproximada. Por ejemplo, el cálculo de la suma de una serie será llevado a cabo necesariamente de manera aproximada, considerando un truncamiento apropiado.

Será por tanto necesario introducir un problema numérico, PN , cuya solución x_n difiere de x en un error e_t que se llama *error de truncamiento*. Tales errores no sólo ocurren en modelos matemáticos que están ya planteados en dimensión finita (por ejemplo, cuando se resuelve un sistema lineal). La suma de errores e_a y e_t constituye el *error computacional* e_c , la cantidad en la que estamos interesados.

El error computacional *absoluto* es la diferencia entre x , la solución del modelo matemático, y \hat{x} , la solución obtenida al final del proceso numérico,

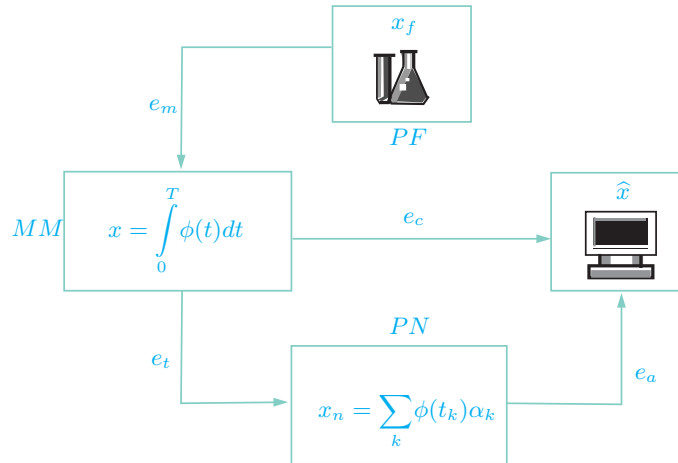


Figura 1.6. Tipos de errores en un proceso computacional

$$e_c^{abs} = |x - \hat{x}|,$$

mientras que (si $x \neq 0$) el error computacional *relativo* es

$$e_c^{rel} = |x - \hat{x}|/|x|,$$

donde $|\cdot|$ denota el módulo, u otra medida del tamaño, dependiendo del significado de x .

El proceso numérico es generalmente una aproximación del modelo matemático obtenido como función de un parámetro de discretización, que será denotado por h y supondremos positivo. Si, cuando h tiende a 0, el proceso numérico devuelve la solución del modelo matemático, diremos que el proceso numérico es *convergente*. Además, si el error (absoluto o relativo) se puede acotar como función de h , de la forma

$$e_c \leq Ch^p \quad (1.11)$$

donde C es un número positivo independiente de h y p , diremos que el método es *convergente de orden p* . A veces es posible incluso reemplazar el símbolo \leq por \simeq , en caso de que, además de la cota superior (1.11), se disponga de una cota inferior $C'h^p \leq e_c$ (siendo C' otra constante ($\leq C$) independiente de h y p).

Ejemplo 1.1 Supongamos que aproximamos la derivada de una función f en un punto \bar{x} por el cociente incremental que aparece en (1.10). Obviamente, si f es diferenciable en \bar{x} , el error cometido reemplazando f' por el cociente incremental tiende a 0 cuando $h \rightarrow 0$. Sin embargo, como veremos en la Sección 4.1, el error puede ser considerado como Ch sólo si $f \in C^2$ en un entorno de \bar{x} . ■

Mientras se estudian las propiedades de convergencia de un procedimiento numérico, a menudo manejaremos gráficas que muestran el error como función de h en escala logarítmica, esto es $\log(h)$, en el eje de abscisas y $\log(e_c)$ en el eje de ordenadas. El propósito de esta representación es fácil de ver: si $e_c = Ch^p$ entonces $\log e_c = \log C + p \log h$. Por tanto, p en escala logarítmica representa la pendiente de la línea recta $\log e_c$, así que si debemos comparar dos métodos, el que presente la mayor pendiente será el de mayor orden. Para obtener gráficas en escala logarítmica sólo se necesita teclear `loglog(x,y)`, siendo x e y los vectores que contiene las abscisas y las ordenadas de los datos que se quiere representar.

`loglog`

A modo de ejemplo, en la Figura 1.7 recogemos las líneas rectas relativas al comportamiento de los errores en dos métodos diferentes. La línea continua representa una aproximación de primer orden, mientras que la línea de trazos representa un método de segundo orden.

Hay una alternativa a la manera gráfica de establecer el orden de un método cuando uno conoce los errores e_i para algunos valores dados h_i

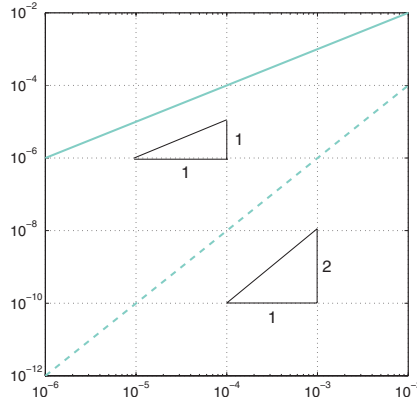


Figura 1.7. Dibujo en escala logarítmica

del parámetro de discretización, con $i = 1, \dots, N$: consiste en suponer que e_i es igual a Ch_i^p , donde C no depende de i . Entonces se puede aproximar p con los valores:

$$p_i = \log(e_i/e_{i-1})/\log(h_i/h_{i-1}), \quad i = 2, \dots, N. \quad (1.12)$$

Realmente el error no es una cantidad computable puesto que depende de la solución desconocida. Por consiguiente es necesario introducir cantidades computables que puedan ser usadas para acotar el error, son los llamados *estimadores de error*. Veremos algunos ejemplos en las Secciones 2.2.1, 2.3 y 4.4.

1.5.1 Hablando de costes

En general un problema se resuelve en el computador mediante un algoritmo, que es una directiva precisa en forma de texto finito, que especifica la ejecución de una serie finita de operaciones elementales. Estamos interesados en aquellos algoritmos que involucran sólo un número finito de etapas.

El *coste computacional* de un algoritmo es el número de operaciones de punto flotante que se requieren para su ejecución. A menudo la velocidad de un computador se mide por el máximo número de operaciones en punto flotante que puede efectuar en un segundo (*flops*). En particular, las siguientes notaciones abreviadas se usan comúnmente: Megaflops, igual a 10^6 *flops*, Gigaflops igual a 10^9 *flops*, Teraflops igual a 10^{12} *flops*. Los computadores más rápidos hoy día alcanzan unos 40 Teraflops.

En general, el conocimiento exacto del número de operaciones requerido por un algoritmo dado no es esencial. En cambio es útil determinar su orden de magnitud como función de un parámetro d que está

relacionado con la dimensión del problema. Por tanto decimos que un algoritmo tiene complejidad *constante* si requiere un número de operaciones independiente de d , es decir $\mathcal{O}(1)$ operaciones, complejidad *lineal* si requiere $\mathcal{O}(d)$ operaciones, o, con más generalidad, complejidad *polinómica* si requiere $\mathcal{O}(d^m)$ operaciones, para un entero positivo m . Otros algoritmos pueden tener complejidad *exponencial* ($\mathcal{O}(c^d)$ operaciones) o incluso *factorial* ($\mathcal{O}(d!)$ operaciones). Recordamos que el símbolo $\mathcal{O}(d^m)$ significa “se comporta, para d grande, como una constante por d^m ”.

Ejemplo 1.2 (producto matriz-vector) Sea A una matriz cuadrada de orden n y sea \mathbf{v} un vector de \mathbb{R}^n . La j -ésima componente del producto $A\mathbf{v}$ está dada por

$$a_{j1}v_1 + a_{j2}v_2 + \dots + a_{jn}v_n,$$

y requiere n productos y $n - 1$ sumas. Por tanto, uno necesita $n(2n - 1)$ operaciones para calcular todas las componentes. De este modo, este algoritmo requiere $\mathcal{O}(n^2)$ operaciones, así que tiene complejidad cuadrática con respecto al parámetro n . El mismo algoritmo requeriría $\mathcal{O}(n^3)$ operaciones para calcular el producto de dos matrices de orden n . Sin embargo, hay un algoritmo, debido a Strassen, que “sólo” requiere $\mathcal{O}(n^{\log_2 7})$ operaciones y otro, debido a Winograd y Coppersmith, que requiere $\mathcal{O}(n^{2.376})$ operaciones. ■

Ejemplo 1.3 (cálculo del determinante de una matriz) Como se ha mencionado anteriormente, el determinante de una matriz cuadrada de orden n puede calcularse usando la fórmula recursiva (1.8). El algoritmo correspondiente tiene complejidad factorial con respecto a n y sólo sería utilizable para matrices de pequeña dimensión. Por ejemplo, si $n = 24$, un ordenador capaz de realizar 1 Petaflops de operaciones (es decir, 10^{15} operaciones en punto flotante por segundo) necesitaría 20 años para llevar a cabo este cálculo. Por tanto uno tiene que recurrir a algoritmos más eficientes. En efecto, existe un algoritmo que permite el cálculo de determinantes mediante productos matriz-matriz, por tanto con una complejidad de $\mathcal{O}(n^{\log_2 7})$ operaciones, aplicando el algoritmo de Strassen mencionado (véase [BB96]). ■

El número de operaciones no es el único parámetro que interviene en el análisis de un algoritmo. Otro factor relevante lo representa el tiempo que se necesita para acceder a la memoria del computador (que depende de la forma en que el algoritmo ha sido codificado). Un indicador de las prestaciones de un algoritmo es, por consiguiente, el tiempo de CPU (CPU significa *unidad central de proceso*), y puede obtenerse usando el comando de MATLAB `cputime`. El tiempo total transcurrido entre las fases de *entrada* y *salida* puede obtenerse con el comando `etime`.

`cputime`
`etime`

Ejemplo 1.4 Para calcular el tiempo necesario para una multiplicación matriz-vector escribamos el siguiente programa:

```
>> n = 4000; step = 50; A = rand(n,n); v = rand(n); T=[];
>> sizeA = [ ]; count = 1;
>> for k = 50:step:n
```

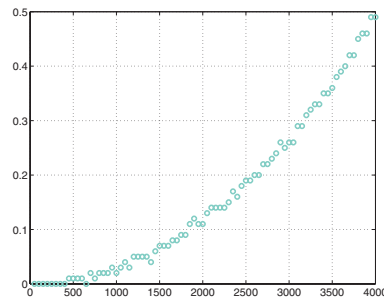


Figura 1.8. Producto matriz-vector: tiempo de CPU (en segundos) frente a la dimensión n de la matriz (en un PC a 2.53 GHz)

```
AA = A(1:k,1:k); vv = v(1:k)';
t = cputime; b = AA*vv; tt = cputime - t;
T = [T, tt]; sizeA = [sizeA,k];
end
```

La instrucción `a:step:b` que aparece en el bucle `for` genera todos los números que son de la forma `a+step*k` donde `k` es un entero que va de 0 al mayor valor `kmax` para el cual `a+step*kmax` no es mayor que `b` (en el presente caso, `a=50`, `b=4000` y `step=50`). El comando `rand(n,m)` define una matriz $n \times m$ de elementos aleatorios. Finalmente, `T` es el vector cuyas componentes contienen el tiempo de CPU necesario para llevar a cabo un producto matriz-vector, mientras que `cputime` devuelve el tiempo de CPU en segundos que ha sido utilizado por el proceso MATLAB desde que MATLAB empezó. El tiempo necesario para ejecutar un solo programa es, por tanto, la diferencia entre el tiempo de CPU y el calculado antes de la ejecución del programa en curso, que se almacena en la variable `t`. La Figura 1.8, obtenida mediante el comando `plot(sizeA,T,'o')`, muestra que el tiempo de CPU crece como el cuadrado del orden de la matriz n . ■

1.6 Los entornos MATLAB y Octave

Los programas MATLAB y Octave, son entornos integrados para el cálculo y la visualización científicos. Están escritos en lenguajes C y C++.

MATLAB está distribuido por The MathWorks (véase el sitio web www.mathworks.com). El nombre significa *MATrix LABoratory* puesto que originalmente fue desarrollado para el cálculo matricial.

Octave, conocido también como GNU Octave (véase el sitio web www.octave.org), es un software que se distribuye libremente. Uno puede redistribuirlo y/o modificarlo en los términos de la Licencia Pública General (GPL) de GNU publicada por la *Free Software Foundation*.

Como se ha mencionado en la introducción de este Capítulo, hay diferencias entre los entornos, lenguajes y *toolboxes* de MATLAB y Octave. Sin embargo, hay un nivel de compatibilidad que nos permite escribir la mayoría de los programas de este libro y ejecutarlos sin dificultad en MATLAB y Octave. Cuando esto no es posible, bien porque algunos comandos se deletrean de manera diferente, o porque operan de forma diferente, o simplemente porque no están implementados, se escribirá una nota al final de cada sección que proporcionará una explicación e indicará qué se podría hacer.

Así como MATLAB tiene sus *toolboxes*, Octave tiene un rico conjunto de funciones disponibles a través del proyecto llamado Octave-forge (véase el sitio web octave.sourceforge.net). Este repositorio de funciones crece continuamente en muchas áreas diferentes tales como álgebra lineal, soporte de matrices huecas (o dispersas) u optimización, por citar algunas. Para ejecutar adecuadamente todos los programas y ejemplos de este libro bajo Octave, es imprescindible instalar Octave-forge.

Una vez instalados, la ejecución de MATLAB y Octave permite el acceso a un entorno de trabajo caracterizado por los *prompt* `>>` y `octave:1>`, respectivamente. Por ejemplo, cuando se ejecuta MATLAB en un computador personal vemos

```
< M A T L A B >
Copyright 1984-2004 The MathWorks, Inc.
Version 7.0.0.19901 (R14)
May 06, 2004
```

```
To get started, select "MATLAB Help" from the Help menu.
>>
```

Cuando ejecutamos Octave en nuestro ordenador personal vemos

```
GNU Octave, version 2.1.72 (x86_64-pc-linux-gnu).
Copyright (C) 2005 John W. Eaton.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.

Additional information about Octave is available at
http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html

Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful
report).

octave:1>
```

1.7 El lenguaje MATLAB

Después de las observaciones introductorias hechas en la sección anterior, estamos preparados para trabajar en los entornos MATLAB y Octave. Además, en adelante, MATLAB debería entenderse como el lenguaje MATLAB que es la intersección de ambos, MATLAB y Octave.

Tras pulsar la tecla *enter* (o *return*), todo lo que se escribe después del *prompt* será interpretado.³ Concretamente, MATLAB comprobará primero si lo que se escribe corresponde a variables que ya han sido definidas o al nombre de uno de los programas o comandos definidos en MATLAB. Si todas esas comprobaciones fallan, MATLAB devuelve un aviso de error. Caso contrario, el comando es ejecutado y posiblemente se visualizará una *salida*. En todos los casos, el sistema devolverá eventualmente el *prompt* para poner de manifiesto que está preparado para un nuevo comando. Para cerrar una sesión de MATLAB uno debería escribir el comando `quit` (o `exit`) y pulsar la tecla *enter*. En adelante se entenderá que para ejecutar un programa o un comando uno tiene que pulsar la tecla *enter*. Además, los términos programa, función o comando se utilizarán de forma equivalente. Cuando nuestro comando coincida con una de las estructuras elementales que caracterizan a MATLAB (por ejemplo un número o una cadena de caracteres que se ponen entre apóstrofes) éstos son inmediatamente devueltos como *salida* en la variable por defecto `ans` (abreviatura de *answer*). He aquí un ejemplo:

`quit`
`exit`

`ans`

```
>> 'casa'
```

```
ans =  
casa
```

Si ahora escribimos una cadena de caracteres (o número) diferente, `ans` asumirá este nuevo valor.

Podemos desactivar la presentación automática de la *salida* escribiendo un punto y coma después de la cadena de caracteres. De este modo, si escribimos `'casa';` MATLAB devolverá simplemente el *prompt* (asignando sin embargo el valor `'casa'` a la variable `ans`).

= Con más generalidad, el comando `=` permite la asignación de un valor (o de una cadena de caracteres) a una variable dada. Por ejemplo, para asignar la cadena `'Bienvenido a Madrid'` a la variable `a` podemos escribir

```
>> a='Bienvenido a Madrid';
```

De este modo no hay necesidad de declarar el *tipo* de una variable, MATLAB lo hará automática y dinámicamente. Por ejemplo, si escribimos `a=5`, la variable `a` contendrá ahora un número y ya no una cadena

³ Así, un programa MATLAB no tiene necesariamente que ser compilado como requieren otros lenguajes, por ejemplo, Fortran o C.

de caracteres. Esta flexibilidad no es gratis. Si ponemos una variable de nombre `quit` igual al número 5 estamos inhibiendo el comando de MATLAB `quit`. Por consiguiente, deberíamos tratar de evitar el uso de variables que tengan el nombre de comandos de MATLAB. Sin embargo, mediante el comando `clear` seguido por el nombre de una variable (por ejemplo `quit`), es posible cancelar esta asignación y restaurar el significado original del comando `quit`.

`clear`

Mediante el comando `save` todas las variables de la sesión (que están almacenadas en el llamado “espacio básico de trabajo”, *base workspace*), se salvan en el archivo binario `matlab.mat`. Análogamente, el comando `load` restaura en la sesión en curso todas las variables almacenadas en `matlab.mat`. Se puede especificar un nombre de fichero después de `save` o `load`. Uno puede también salvar sólo variables seleccionadas, digamos `v1`, `v2` y `v3`, en un fichero llamado, por ejemplo, `area.mat`, usando el comando `save area v1 v2 v3`.

`save``load`

Mediante el comando `help` uno puede ver la totalidad de la familia de comandos y variables predefinidas, incluyendo las llamadas *toolboxes* que son conjuntos de comandos especializados. Entre ellos recordemos aquéllos que definen las funciones elementales tales como seno (`sin(a)`), coseno (`cos(a)`), raíz cuadrada (`sqrt(a)`), exponencial (`exp(a)`).

`help``sin cos
sqrt exp`

Hay caracteres especiales que no pueden aparecer en el nombre de una variable ni tampoco en un comando, por ejemplo los operadores algebraicos (`+`, `-`, `*` y `/`), los operadores lógicos *y* (`&`), *o* (`|`), *no* (`~`), los operadores relacionales *mayor que* (`>`), *mayor o igual que* (`>=`), *menor que* (`<`), *menor o igual que* (`<=`), *igual a* (`==`). Finalmente, un nombre nunca puede empezar con una cifra, un corchete o un signo de puntuación.

`+ -
* / & |
~ > >= <
<==`

1.7.1 Instrucciones de MATLAB

Un lenguaje especial de programación, el lenguaje MATLAB, también está disponible, permitiendo a los usuarios escribir nuevos programas. Aunque no se requiere su conocimiento para entender cómo usar los diversos programas que introduciremos a lo largo de este libro, puede proporcionar al lector la capacidad de modificarlos así como la de producir otros nuevos.

El lenguaje MATLAB incluye instrucciones estándar, tales como condicionales y bucles.

El *if-elseif-else* condicional tiene la siguiente forma general:

```
if condicion(1)
    instruccion(1)
elseif condicion(2)
    instruccion(2)
.
.
.
```

```

else
    instruccion(n)
end

```

donde `condicion(1)`, `condicion(2)`, ... representan conjuntos de expresiones lógicas de MATLAB, con valores 0 o 1 (falso o verdadero) y la construcción entera permite la ejecución de la instrucción correspondiente a la condición que toma el valor igual a 1. Si todas las condiciones fuesen falsas, tendría lugar la ejecución de `instruccion(n)`. De hecho, si el valor de `condicion(k)` fuese cero, el control se movería hacia delante. Por ejemplo, para calcular las raíces de un polinomio cuadrático $ax^2 + bx + c$ uno puede usar las siguientes instrucciones (el comando `disp(.)` simplemente presenta lo que se escribe entre corchetes):

```

>> if a ~= 0
    sq = sqrt(b*b - 4*a*c);
    x(1) = 0.5*(-b + sq)/a;
    x(2) = 0.5*(-b - sq)/a;
elseif b ~= 0
    x(1) = -c/b;
elseif c ~= 0
    disp(' Ecuacion imposible');
else
    disp(' La ecuacion dada es una identidad')
end

```

(1.13)

Nótese que MATLAB no ejecuta la construcción completa hasta que no se teclea la instrucción `end`.

MATLAB permite dos tipos de bucles, un bucle *for* (comparable al bucle *do* de FORTRAN o al bucle *for* de C) y un bucle *while*. Un bucle *for* repite las instrucciones en el bucle mientras el índice toma los valores contenidos en un vector fila dado. Por ejemplo, para calcular los seis primeros términos de la sucesión de Fibonacci $f_i = f_{i-1} + f_{i-2}$, para $i \geq 3$, con $f_1 = 0$ y $f_2 = 1$, uno puede usar las siguientes instrucciones:

```

>> f(1) = 0; f(2) = 1;
>> for i = [3 4 5 6]
    f(i) = f(i-1) + f(i-2);
end

```

Nótese que se puede usar un punto y coma para separar varias instrucciones MATLAB tecleadas en la misma línea. Obsérvese también que podemos reemplazar la segunda instrucción por `for i = 3:6`, que es equivalente. El bucle *while* se repite en tanto en cuanto la condición dada sea cierta. Por ejemplo, el siguiente conjunto de instrucciones puede utilizarse como alternativa al conjunto anterior:

```

>> f(1) = 0; f(2) = 1; k = 3;
>> while k <= 6
    f(k) = f(k-1) + f(k-2); k = k + 1;
end

```


Existen otras instrucciones de uso quizás menos frecuente, tales como *switch*, *case*, *otherwise*. El lector interesado puede tener acceso a su significado a través del comando `help`.

1.7.2 Programación en MATLAB

Expliquemos brevemente cómo escribir programas en MATLAB. Un programa nuevo debe introducirse en un archivo con un nombre dado con extensión `m`, que se llama *m-file*. Estos ficheros deben estar localizados en una de las carpetas en las que MATLAB busca automáticamente los *m-files*; su lista puede obtenerse mediante el comando `path` (véase `help path` para saber cómo añadir una carpeta a esta lista). La primera carpeta escaneada por MATLAB es la “carpeta de trabajo” en curso.

A este nivel es importante distinguir entre *scripts* y *functions*. Un *script* es simplemente una colección de comandos de MATLAB en un *m-file* y puede ser usado interactivamente. Por ejemplo, el conjunto de instrucciones (1.13) puede dar origen a un *script* (que podríamos llamar `equation`) copiándolo en el archivo `equation.m`. Para lanzarlo, se puede escribir simplemente la instrucción `equation` después del *prompt* de MATLAB `>>`. Mostramos a continuación dos ejemplos:

```
>> a = 1; b = 1; c = 1;
>> equation

ans =
    -0.5000 + 0.8660i    -0.5000 - 0.8660i

>> a = 0; b = 1; c = 1;
>> equation

ans =
    -1
```

Puesto que no tenemos interfaz de entrada-salida, todas las variables usadas en un *script* son también las variables de la sesión de trabajo y son, por tanto, borradas solamente bajo un comando explícito (`clear`). Esto no es en absoluto satisfactorio cuando uno intenta escribir programas más complejos involucrando muchas variables temporales y comparativamente menos variables de entrada y salida, que son las únicas que pueden ser efectivamente salvadas una vez terminada la ejecución del programa. Mucho más flexible que los *scripts* son las *functions*.

Una *function* también se define en un *m-file*, por ejemplo `name.m`, pero tiene un interfaz de entrada/salida bien definido que se introduce mediante el comando `function`

```
function [out1,...,outn]=name(in1,...,inm)
```

`function`

donde $\text{out1}, \dots, \text{outn}$ son las variables de salida e $\text{in1}, \dots, \text{inm}$ son las variables de entrada.

El archivo siguiente, llamado `det23.m`, define una nueva función denominada `det23` que calcula, de acuerdo con la fórmula dada en la Sección 1.3, el determinante de una matriz cuya dimensión podría ser 2 o 3:

```
function det=det23(A)
%DET23 calcula el determinante de una matriz cuadrada
% de dimension 2 o 3
[n,m]=size(A); if n==m
    if n==2
        det = A(1,1)*A(2,2)-A(2,1)*A(1,2);
    elseif n == 3
        det = A(1,1)*det23(A([2,3],[2,3]))-...
            A(1,2)*det23(A([2,3],[1,3]))+...
            A(1,3)*det23(A([2,3],[1,2]));
    else
        disp(' Solamente matrices 2x2 o 3x3');
    end
else
    disp(' Solamente matrices cuadradas');
end return
```

... Nótese el uso de los puntos suspensivos `...` que significan que la instrucción continúa en la línea siguiente y el carácter `%` para iniciar comentarios. La instrucción `A([i,j],[k,l])` permite la construcción de una matriz 2×2 cuyos elementos son los de la matriz original `A` que están en las intersecciones de las filas i -ésima y j -ésima con las columnas k -ésima y l -ésima.

Cuando se invoca una *function*, MATLAB crea un espacio de trabajo local (el *espacio de trabajo de la function*). Los comandos dentro de la *function* no se pueden referir a variables del espacio de trabajo global (interactivo) salvo que se pasen como entradas. En particular, las variables utilizadas por una *function* se borran cuando la ejecución termina, salvo que se devuelvan como parámetros de salida.

Observación 1.2 (variables globales) Existe la posibilidad de declarar variables *globales*, y utilizarlas dentro de una *function* sin necesidad de pasarlas como entradas. Para ello, deben declararse como tales en todos los lugares donde se pretenda utilizarlas. A tal efecto, MATLAB dispone del comando `global` (véase, por ejemplo, [HH05]). Si varias funciones, y posiblemente el espacio de trabajo, declaran un nombre particular como variable global, entonces todas comparten una copia de esa variable. Una asignación a esa variable en cualquiera de ellas (o en el espacio de trabajo), queda disponible para todas las demás. •

Generalmente las *functions* terminan cuando se alcanza el `end` de la *function*, sin embargo se puede usar una instrucción `return` para forzar un regreso más temprano (cuando se cumple cierta condición).

Por ejemplo, para aproximar el número de la sección de oro $\alpha = 1.6180339887\dots$, que es el límite para $k \rightarrow \infty$ del cociente de dos

números de Fibonacci consecutivos, f_k/f_{k-1} , iterando hasta que la diferencia entre dos cocientes consecutivos sea menor que 10^{-4} , podemos construir la siguiente *function*:

```
function [golden,k]=fibonacci0
f(1) = 0; f(2) = 1; goldenold = 0;
kmax = 100; tol = 1.e-04;
for k = 3:kmax
    f(k) = f(k-1) + f(k-2);
    golden = f(k)/f(k-1);
    if abs(golden - goldenold) <= tol
        return
    end
    goldenold = golden;
end
return
```

Su ejecución se interrumpe después de `kmax=100` iteraciones o cuando el valor absoluto de la diferencia entre dos iterantes consecutivos sea más pequeña que `tol=1.e-04`. Entonces, podemos escribir

```
[alpha,niter]=fibonacci0

alpha =
    1.618055555555556
niter =
    14
```

Después de 14 iteraciones la *function* ha devuelto un valor aproximado que comparte con α las 5 primeras cifras significativas.

El número de parámetros de entrada y salida de una *function* en MATLAB puede variar. Por ejemplo, podríamos modificar la *function* de Fibonacci como sigue:

```
function [golden,k]=fibonacci1(tol,kmax)
if nargin == 0
    kmax = 100; tol = 1.e-04; % valores por defecto
elseif nargin == 1
    kmax = 100; % valor por defecto solo para kmax
end
f(1) = 0; f(2) = 1; goldenold = 0;
for k = 3:kmax
    f(k) = f(k-1) + f(k-2);
    golden = f(k)/f(k-1);
    if abs(golden - goldenold) <= tol
        return
    end
    goldenold = golden;
end
return
```

La *function* `nargin` cuenta el número de parámetros de entrada. En la nueva versión de la *function* `fibonacci` podemos prescribir el máximo número de iteraciones internas permitidas (`kmax`) y especificar una tolerancia `tol`. Cuando esta información se olvida, la *function* debe suminis-

trar valores por defecto (en nuestro caso, `kmax = 100` y `tol = 1.e-04`). Un ejemplo de ello es el siguiente:

```
[alpha,niter]=fibonacci1(1.e-6,200)
```

```
alpha =
    1.61803381340013
niter =
    19
```

Nótese que utilizando una tolerancia más estricta hemos obtenido un nuevo valor aproximado que comparte con α ocho cifras significativas. La *function* `nargin` puede ser usada externamente a una *function* dada para obtener el número de sus parámetros de entrada. He aquí un ejemplo:

```
nargin('fibonacci1')
```

```
ans =
    2
```

inline **Observación 1.3 (funciones en línea)** El comando `inline`, cuya sintaxis más simple es `g=inline(expr,arg1,arg2,...,argn)`, declara una función `g` que depende de las cadenas de caracteres `arg1,arg2,...,argn`. La cadena `expr` contiene la expresión de `g`. Por ejemplo, `g=inline('sin(r)','r')` declara la función $g(r) = \sin(r)$. El comando abreviado `g=inline(expr)` asume implícitamente que `expr` es una función de la variable por defecto `x`. Una vez que una función *inline* ha sido declarada, puede ser evaluada para cualquier conjunto de variables a través del comando `feval`. Por ejemplo, para evaluar `g` en los puntos `z=[0 1]` podemos escribir

```
>> feval('g',z);
```

Nótese que, contrariamente al caso del comando `eval`, con `feval` el nombre de la variable (`z`) no necesita coincidir con el nombre simbólico (`r`) asignado por el comando `inline`. •

Después de esta rápida introducción, nuestra sugerencia es explorar MATLAB usando el comando *help*, y ponerse al tanto de la implementación de varios algoritmos mediante los programas descritos a lo largo de este libro. Por ejemplo, tecleando `help for` conseguimos no sólo una descripción completa del comando `for` sino también una indicación sobre instrucciones similares a `for`, tales como `if`, `while`, `switch`, `break` y `end`. Invocando sus *help* podemos mejorar progresivamente nuestro conocimiento de MATLAB.

Octave 1.7 Hablando en general, un área con pocos elementos en común es la de los medios para dibujar de MATLAB y Octave. Comprobamos que la mayoría de los comandos de dibujo que aparecen en el libro son reproducibles en ambos programas, pero existen de hecho

muchas diferencias fundamentales. Por defecto, el entorno de Octave para dibujar es GNUPlot; sin embargo el conjunto de comandos para dibujar es diferente y opera de forma diferente de MATLAB. En el momento de escribir esta sección, existen otras bibliotecas de dibujo en Octave tales como `octaviz` (véase el sitio web, <http://octaviz.sourceforge.net/>), `epstk` (<http://www.epstk.de/>) y `octplot` (<http://octplot.sourceforge.net>). La última es un intento de reproducir los comandos de dibujo de MATLAB en Octave. ■

Véanse los Ejercicios 1.9-1.13.



1.7.3 Ejemplos de diferencias entre los lenguajes MATLAB y Octave

Como ya se ha mencionado, lo que se ha escrito en la sección anterior sobre el lenguaje MATLAB se aplica a ambos entornos, MATLAB y Octave, sin cambios. Sin embargo, existen algunas diferencias para el lenguaje en sí. Así, programas escritos en Octave pueden no correr en MATLAB y viceversa. Por ejemplo, Octave soporta cadenas de caracteres con comillas simples y dobles

```
octave:1> a="Bienvenido a Madrid"
a = Bienvenido a Madrid
```

```
octave:2> a='Bienvenido a Madrid'
a = Bienvenido a Madrid
```

mientras que MATLAB sólo soporta comillas sencillas; las comillas dobles originan errores.

Proporcionamos aquí una lista con otras cuantas incompatibilidades entre los dos lenguajes:

- MATLAB no permite un blanco antes del operador trasponer. Por ejemplo, `[0 1]'` funciona en MATLAB, pero `[0 1] '` no. Octave permite ambos casos;
- MATLAB siempre requiere ...,

```
rand (1, ...
      2)
```

mientras que ambas

```
rand (1,
      2)
```

y

```
rand (1, \
      2)
```

funcionan en Octave además de ...;

- para la exponenciación, Octave puede usar `^` o `**`; MATLAB requiere `^`;
- como terminaciones, Octave usa `end` pero también `endif`, `endfor`, ...; MATLAB requiere `end`.

1.8 Lo que no le hemos dicho

Una discusión sistemática sobre los números de punto flotante puede encontrarse en [Übe97], [Hig02] o en [QSS06].

Para cuestiones relativas al tema de la complejidad, remitimos, por ejemplo, a [Pan92].

Para una introducción más sistemática a MATLAB el lector interesado puede consultar el manual de MATLAB [HH05] así como libros específicos tales como [HLR01], [Pra02], [EKM05], [Pal04] o [MH03].

Para Octave recomendamos el manual mencionado al principio de este capítulo.

1.9 Ejercicios

Ejercicio 1.1 ¿Cuántos números pertenecen al conjunto $\mathbb{F}(2, 2, -2, 2)$? ¿Cuál es el valor de ϵ_M para este conjunto?

Ejercicio 1.2 Demostrar que $\mathbb{F}(\beta, t, L, U)$ contiene precisamente $2(\beta-1)\beta^{t-1}(U-L+1)$ elementos.

Ejercicio 1.3 Probar que i^i es un número real; a continuación comprobar este resultado utilizando MATLAB u Octave.

Ejercicio 1.4 Escribir en MATLAB las instrucciones para construir una matriz triangular superior (respectivamente, inferior) de dimensión 10 teniendo 2 en la diagonal principal y -3 en la diagonal superior (respectivamente, inferior).

Ejercicio 1.5 Escribir en MATLAB las instrucciones que permiten el intercambio de las líneas tercera y séptima de las matrices construidas en el Ejercicio 1.3, y luego las instrucciones que permiten el intercambio entre las columnas cuarta y octava.

Ejercicio 1.6 Verificar si los siguientes vectores de \mathbb{R}^4 son linealmente independientes:

$$\mathbf{v}_1 = [0 \ 1 \ 0 \ 1], \quad \mathbf{v}_2 = [1 \ 2 \ 3 \ 4], \quad \mathbf{v}_3 = [1 \ 0 \ 1 \ 0], \quad \mathbf{v}_4 = [0 \ 0 \ 1 \ 1].$$

Ejercicio 1.7 Escribir las siguientes funciones y calcular sus derivadas primera y segunda, así como sus primitivas, usando la *toolbox* de cálculo simbólico de MATLAB:

$$f(x) = \sqrt{x^2 + 1}, \quad g(x) = \operatorname{sen}(x^3) + \cosh(x).$$

Ejercicio 1.8 Para cualquier vector dado \mathbf{v} de dimensión n , usando el comando `c=poly(v)` uno puede construir el $(n+1)$ -ésimo coeficiente del polinomio $p(x) = \sum_{k=1}^{n+1} c(k)x^{n+1-k}$ que es igual a $\prod_{k=1}^n (x - v(k))$. En aritmética exacta, uno encontraría que $\mathbf{v} = \mathbf{roots}(\mathbf{poly}(\mathbf{c}))$. Sin embargo, esto no puede ocurrir debido a los errores de redondeo, como puede comprobarse usando el comando `roots(poly([1:n]))`, donde n va de 2 a 25. poly

Ejercicio 1.9 Escribir un programa para calcular la siguiente sucesión:

$$I_0 = \frac{1}{e}(e-1),$$

$$I_{n+1} = 1 - (n+1)I_n, \quad \text{para } n = 0, 1, \dots$$

Comparar el resultado numérico con el límite exacto $I_n \rightarrow 0$ para $n \rightarrow \infty$.

Ejercicio 1.10 Explicar el comportamiento de la sucesión (1.4) cuando se calcula con MATLAB.

Ejercicio 1.11 Considérese el siguiente algoritmo para calcular π . Genere n pares $\{(x_k, y_k)\}$ de números aleatorios en el intervalo $[0, 1]$, entonces calcule el número m de los que están dentro del primer cuadrante del círculo unidad. Obviamente, π resulta ser el límite de la sucesión $\pi_n = 4m/n$. Escriba un programa en MATLAB para calcular esta sucesión y compruebe el error para valores crecientes de n .

Ejercicio 1.12 Puesto que π es la suma de la serie

$$\pi = \sum_{n=0}^{\infty} 16^{-n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} + \frac{1}{8n+5} + \frac{1}{8n+6} \right),$$

podemos calcular una aproximación de π sumando hasta el término n -ésimo, para un n suficientemente grande. Escribir una *function* MATLAB para calcular sumas finitas de la serie anterior. ¿Cómo debe ser de grande n para obtener una aproximación de π al menos tan precisa como la almacenada en la variable π ?

Ejercicio 1.13 Escribir un programa para el cálculo de los coeficientes binomiales $\binom{n}{k} = n!/(k!(n-k)!)$, donde n y k son dos números naturales con $k \leq n$.

Ejercicio 1.14 Escribir en MATLAB una *function* recursiva que calcule el n -ésimo elemento f_n de la sucesión de Fibonacci. Observando que

$$\begin{bmatrix} f_i \\ f_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{i-1} \\ f_{i-2} \end{bmatrix} \quad (1.14)$$

escribir otra *function* que calcule f_n basado en esta nueva forma recursiva. Finalmente, calcular el tiempo de CPU correspondiente.

Cálculo Científico con MATLAB y Octave

Quarteroni, A.; Saleri, F.

2006, XII, 329 p., Softcover

ISBN: 978-88-470-0503-7