TEMA 2. ARITMÉTICA FLOTANTE Y ANÁLISIS DE ERRORES

2 Aritmética de ordenadores y análisis de errores **29**

2.1 ¿Por qué son importantes los errores numéricos? 29

2.1.1 El fallo de un misil Patriot 30

2.1.2 La explosión del cohete Ariane 5 31

2.2 Representación de números en el ordenador 31

2.2.1 Números naturales y la regla de Horner 32

2.2.2 Números enteros en binario 33

2.2.3 Números reales en punto flotante 35

2.2.4 El formato en punto flotante IEEE-754 36

2.3 Errores, condicionamiento y estabilidad numérica 42

2.3.1 Errores relativos de truncado y redondeo 42

2.3.2 Operaciones aritméticas básicas en punto flotante 45

2.3.3 Errores progresivos, regresivos y estabilidad numérica 47

2.3.4 Número de condicionamiento 48

2.3.5 Cancelación catastrófica 51

2.4 Análisis de la propagación de los errores 57

2.4.1 Modelo estándar para la aritmética flotante 58

2.4.2 Análisis de errores para el producto interior y el exterior 59
2.4.3 El propósito del análisis de errores 64
2.4.4 Unos lemas útiles en análisis de errores 65
2.4.5 Análisis de errores para el producto matricial 67
2.4.6 Análisis de errores para la suma 68
2.5 Relación entre condicionamiento y estabilidad 72
2.6 Pseudocódigo utilizado en los algoritmos 77

Bibliografía **83**

18 de octubre de 2002

© Francisco R. Villatoro, Carmen M. García, Juan I. Ramos. Estas notas están protegidas por derechos de copyright y pueden ser distribuidas libremente sólo con propósitos educativos sin ánimo de lucro. *These notes are copyright-protected, but may be freely distributed for instructional nonprofit purposes.*

CAPÍTULO 2

ARITMÉTICA DE ORDENADORES Y ANÁLISIS DE ERRORES

2.1 ¿Por qué son importantes los errores numéricos?

Turing[1] demostró en 1936 que el conjunto de los números calculables o computables mediante ordenador (tales que existe un programa capaz de calcular sus dígitos uno a uno) es de cardinal numerable, como lo es el de los números algebraicos (los que son raíces de polinomios), los racionales (cociente de enteros) o los mismos naturales. El conjunto de números reales es no numerable, por lo que elegido un número real aleatoriamente, éste es imposible de calcular mediante un ordenador. Este hecho quizás parezca sorprendente, teniendo en cuenta que se utilizan los ordenadores para resolver todo tipo de problemas prácticos sin que este hecho sea relevante. Afortunadamente, el conjunto de los números calculables (igual que el de los racionales) es denso en el de los reales, es decir, dado un real, existe un número calculable tan próximo a éste como queramos. Ese es el secreto de que los ordenadores sean útiles en computación científica. Aún así, y por ello, los ordenadores siempre cometen errores a la hora de representar un número.

Los ordenadores trabajan con una aritmética que utiliza un número finito de dígitos. Un número real tiene, salvo pocas excepciones, infinitos dígitos. Para representar este número en un ordenador tenemos que elegir un número finito de dígitos, lo que introduce un error en la representación de dicho número. Tener en cuenta estos errores introducidos por el ordenador es muy importante, sobre todo en aplicaciones en las que se realizan muchas operaciones aritméticas. Conforme realizamos estas operaciones el error se propaga, se acumula y en algunos casos puede llegar a crecer exponencialmente, provocando grandes errores en el resultado final. En este tema

[1] Turing realizó en 1948 uno de los primeros análisis de propagación de errores para el método de Gauss para resolver sistemas lineales que estudiaremos en el tema 4.

estudiaremos qué errores comete un ordenador, cómo se puede analizar su propagación, cómo detectar cuando van a ser perniciosos y, sólo en algunos casos, cómo evitar sus efectos negativos.

Para empezar, presentamos dos problemas de la vida real que se han debido a la propagación de errores numéricos. Mencionaremos algunos conceptos, como representación en punto flotante, redondeo, aritmética binaria, etc., que serán introducidos más adelante en este tema. El lector puede relegar la lectura de estos problemas hasta el final del tema.

2.1.1 El fallo de un misil Patriot

El 25 de febrero de 1991, durante la guerra del Golfo, una batería de misiles Patriot americanos en Dharan (Arabia Saudí) no logró interceptar un misil Scud iraquí. Murieron 28 soldados americanos. La causa: los errores numéricos por utilizar truncado en lugar de redondeo en el sistema que calcula el momento exacto en que debe ser lanzado el misil [6].

Los ordenadores de los Patriot que han de seguir la trayectoria del misil Scud, la predicen punto a punto en función de su velocidad conocida y del momento en que fue detectado por última vez en el radar. La velocidad es un número real. El tiempo es una magnitud real pero el sistema la calculaba mediante un reloj interno que contaba décimas de segundo, por lo que representaban el tiempo como una variable entera. Cuanto más tiempo lleva el sistema funcionando más grande es el entero que representa el tiempo. Los ordenadores del Patriot almacenan los números reales representados en punto flotante con una mantisa de 24 bits. Para convertir el tiempo entero en un número real se multiplica éste por 1/10, y se trunca el resultado (en lugar de redondearlo). El número 1/10 se almacenaba truncado a 24 bits. El pequeño error debido al truncado, se hace grande cuando se multiplica por un número (entero) grande, y puede conducir a un error significativo. La batería de los Patriot llevaba en funcionamiento más de 100 horas, por lo que el tiempo entero era un número muy grande y el número real resultante tenía un error cercano a 0.34 segundos.

Veamos el cálculo en detalle. El número 1/10 es $1/2^4 + 1/2^5 + 1/2^8 + 1/2^9 + 1/2^{12} + 1/2^{13} + \cdots$, es decir, $(0.0001100110011001100110011001100\cdots)_2$, que almacenado en un registro de 24 bits conduce al número $(0.00011001100110011001100)_2$ que introduce un error de $(0.0000000000000000000000011001100\cdots)_2$, igual en decimal a 0.000000095.

En 100 horas este pequeño error se multiplica y amplifica hasta alcanzar $0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$. Como un misil Scud viaja a unos 1676 m/s, es decir, unos 6000 km/hora, en 0.34 segundos recorre más de medio kilómetro. Esta distancia fue suficiente para que el misil Patriot no pudiera alcanzar al misil Scud y destruirlo.

2.1.2 La explosión del cohete Ariane 5

El 4 de junio de 1996, el cohete Ariane 5 de la Agencia Europea del Espacio (ESA) explotó 40 segundos después de su despegue a una altura de 3.7 km. tras desviarse de la trayectoria prevista [7]. Era su primer viaje tras una década de investigación que costó más de 7000 millones de euros. El cohete y su carga estaban valorados en más de 500 millones de euros. La causa del error fue un fallo en el sistema de guiado de la trayectoria provocado 37 segundos después del despegue. Este error se produjo en el software que controlaba el sistema de referencia inercial (SRI). En concreto, se produjo una excepción software debido al intento de convertir un número en punto flotante de 64 bits, relacionado con la velocidad horizontal del cohete respecto de la plataforma de lanzamiento, en un entero con signo de 16 bits. El número más grande que se puede representar de esta forma es 32768. El intento de convertir un número mayor causó la excepción que provocó que el software de seguimiento de la trayectoria dejara de funcionar y en última instancia el accidente.

2.2 Representación de números en el ordenador

Para que un ordenador pueda manejar números naturales, enteros, racionales, reales o, incluso, complejos, es necesario representar estos números en memoria en un formato bien definido y suficientemente flexible. Además, todos los ordenadores tienen unidades aritmético-lógicas y/o coprocesadores aritméticos que realizan las operaciones numéricas que el ordenador necesita. La implementación física o hardware mediante circuitos electrónicos de estos dispositivos requiere una representación numérica adecuada de los números. Normalmente se utiliza una representación estática, que utiliza una cantidad fija de memoria, siempre la misma, para representar cada tipo de número. Esta representación facilita el diseño electrónico de estos circuitos. Nosotros estudiaremos en este curso sólo representaciones estáticas. Los lenguajes de programación de alto nivel, como Fortran o C, y la mayoría de los lenguajes matemáticos, como Matlab y también Mathematica, utilizan este sistema de representación de números.

Sin embargo, existen también representaciones dinámicas que utilizan una cantidad de memoria variable en función de las necesidades de cada número concreto, como números naturales o números racionales, de longitud arbitraria. El programa matemático Mathematica, y Matlab a través de la Toolbox simbólica (basada en Maple), permiten aritmética exacta (con números racionales) de longitud arbitraria, sólo limitada por la memoria de la máquina. Sin embargo, su uso se reserva para aplicaciones muy específicas. Prácticamente no se utilizan en computación científico-técnica.

Estudiaremos la representación y el almacenamiento de números naturales, enteros y reales, éstos últimos en punto flotante. También se presentará el estándar IEEE-754 de representación de números reales.

2.2.1 Números naturales y la regla de Horner

Para escribir números mediante símbolos en una hoja de papel se utiliza una representación en un sistema numérico. La forma habitual de representar números utiliza el sistema decimal o base 10. En esta representación un número natural se representa por una cadena de dígitos de la forma

$$a_n\, a_{n-1} \dots\, a_1\, a_0 = a_0 + a_1\, 10^1 + \cdots + a_n\, 10^n,$$

donde $a_i \in \{0, 1, \dots 9\}$ y cada dígito se ve afectado por un factor de escala que depende de su posición. Por ejemplo,

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0.$$

Los ordenadores digitales actuales no utilizan la base 10 para representar números y realizar operaciones aritméticas sino que prefieren el sistema binario o de base 2. Los dígitos en este sistema son 0 y 1, se denominan *bits*, y se representan físicamente mediante los dos estados de conducción (*on*) y corte (*off*) de un transistor funcionando como conmutador. Un ejemplo sencillo de representación de números enteros en base 2 es

$$(1101)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13.$$

Utilizaremos notación $(\cdot)_b$ para indicar la base b utilizada en la representación de un número, aunque omitiremos dicha notación para los números decimales ($b = 10$). Como vemos, para representar números en sistema binario son necesarios más dígitos que en sistema decimal.

También se pueden utilizar representaciones en otros sistemas de numeración como el sistema octal (base 8, $a_i \in \{0, 1, \dots, 7\}$, donde cada dígito representa tres *bits*) y el hexadecimal (base 16, $a_i \in \{0, 1, \dots, 9, \text{A, B, C, D, E, F}\}$, donde cada dígito representa cuatro *bits*). Los sistemas octal y hexadecimal, sobre todo éste último, facilitan la escritura de números binarios grandes y su conversión al sistema decimal, ya que reducen significativamente el número de dígitos del número; por ejemplo

$$\begin{aligned} 123 = (1111011)_2 &= (001\,111\,011)_2 = (173)_8 \\ &= (0111\,1011)_2 = (7\text{B})_{16}. \end{aligned}$$

La utilidad de los sistemas octal y hexadecimal es muy limitada en computación científica y en análisis numérico.

Hemos visto como pasar un número binario a decimal. La operación inversa, escribir un número decimal en binario, se realiza dividiendo reiteradamente el número entre dos hasta que el cociente sea la unidad; escribiendo este cociente y todos los restos en orden inverso a como los hemos obtenidos se obtienen los dígitos del número en binario. Por ejemplo, pasemos 123 y 124 a base dos,

123	61	30	15	7	3	1	$\Longrightarrow (1111011)_2$,
1	1	0	1	1	1		

124	62	31	15	7	3	1	$\Longrightarrow (1111100)_2$,
0	0	1	1	1	1		

donde la primera fila de la tabla representa los cocientes y la segunda línea los restos de dividir entre dos.

Para evaluar un número en una base b mediante un ordenador el procedimiento más rápido es utilizar la regla de Horner (que también se utiliza para evaluar polinomios). Escribiremos el número de la siguiente forma

$$\begin{aligned} & a_n\, b^n + a_{n-1}\, b^{n-1} + \cdots + a_2\, b^2 + a_1\, b + a_0 \\ &= a_0 + b\,(a_1 + b\,(a_2 + \cdots + b\,\underbrace{(a_{n-2} + b\,\overbrace{(a_{n-1} + b\,a_n)}^{c_{n-1}})}_{c_{n-2}}\cdots)), \end{aligned}$$

y calculamos de forma sucesiva los coeficientes c_i de la forma

$$c_n = a_n, \qquad c_i = b\, c_{i+1} + a_i, \qquad i = n-1, n-2, \ldots, 0.$$

De este forma se necesitan solamente n multiplicaciones y n adiciones, y no hay que calcular ninguna potencia de b de forma explícita.

El algoritmo 2.1 presenta el pseudocódigo para la aplicación de la regla de Horner. En la sección 2.6 se describe en detalle el pseudocódigo que utilizaremos a lo largo de este texto.

2.2.2 Números enteros en binario

Para representar un número entero en decimal basta añadir, si es negativo, el signo ortográfico menos; por ejemplo, −23. En binario se puede hacer lo mismo, dedicando un bit para el signo;

Algoritmo 2.1 *Aplicación de la regla de Horner.*

```
function c = horner (a,b)
% a es un vector de dígitos de un número
% b es la base
% c es el resultado decimal de evaluar dicho número
%
  n=length(a)
  c = a(n)
  for i=n-1:-1:0
    c=b*c+a
  end
end
```

sin embargo, entonces el cero tiene dos representaciones equivalentes $+0$ y -0. Para evitar esta duplicidad, en los ordenadores los números enteros de n dígitos se presentan en complemento a dos o por exceso.

En complemento a dos, el bit más significativo representa el signo del número entero. Si es 0, el número es positivo, si es 1 es negativo. El módulo de un número negativo se calcula, complementado a uno el número, es decir, cambiando los bits 1 por 0, y los 0 por 1, y luego sumándole 1. Por ejemplo, utilizando 4 bits, el número $(0111)_2$ representa el número positivo 7, y el número $(1010)_2$ representa el valor -6, ya que sumándole uno a su complemento a uno obtenemos $(0101)_2 + 1 = (0110)_2$.

Es fácil verificar que en complemento a dos el cero tiene una única representación, todos los *bits* a cero; además, el intervalo de números enteros representables en complemento a dos con n *bits* es $[-2^{n-1}, 2^{n-1} - 1]$; finalmente, en orden creciente de representación, los números están ordenados de la forma $0,\ 1,\ 2,\ \ldots,\ 2^{n-1} - 1,\ -2^{n-1},\ -2^{n-1} + 1,\ \ldots,\ -2,\ -1$. Como vemos, la ventaja de esta representación es que el cero se representa como tal, sin embargo, tiene la desventaja de que cambia el orden de los números enteros ya que los negativos son "más grandes" que los positivos.

En casi todos los ordenadores los números enteros se representan y almacenan en complemento a dos. La longitud n en bits de un número entero suele ser un múltiplo (par) de 8, y según ésta se denominan entero corto (*short integer*), entero (*integer*), entero largo (*long integer*), y entero muy largo (*very long integer*), cuando $n = 8$, 16, 32 y 64, respectivamente.

En algunos casos el orden "incorrecto" de los números representados en complemento a dos

es un problema grave que hay que resolver. La solución más simple es utilizar la representación en exceso a z, que consiste en sumar al número decimal la cantidad z y luego representarlo en binario con n *bits*. Normalmente, sólo se utilizan $z = 2^{n-1}$ o $z = 2^{n-1} - 1$. Por ejemplo, con cuatro bits y $z = 2^{n-1} = 2^3 = 8$, el número -8 se representa como $(0000)_2$, el -7 como $(0001)_2$, el 0 como $(1000)_2$, el 7 como $(1111)_2$, y el 8 no es representable con sólo cuatro bits. Animamos al lector a determinar cómo se representarían estos números con $z = 2^{n-1} - 1$.

Tomando $z = 2^{n-1}$, es fácil verificar que el cero tiene representación única, el bit más significativo a uno y el resto a cero, y que el intervalo de números enteros con n *bits* es $[-2^{n-1}, 2^{n-1} - 1]$, el mismo que en complemento a dos, pero ahora en orden creciente encontramos el orden usual, primero los negativos (cuyo bit más significativo es 0) y luego los positivos (en los que es 1), es decir, -2^{n-1}, $-2^{n-1} + 1$, $\ldots$, -2, -1, 0, 1, 2, $\ldots$, $2^{n-1} - 1$.

La otra posibilidad es tomar $z = 2^{n-1} - 1$, con lo que el cero se representa con el bit más significativo a cero y el resto a uno, el intervalo representable es $[-2^{n-1} + 1, 2^{n-1}]$, y el orden es el usual, o sea, $-2^{n-1} + 1$, $\ldots$, -2, -1, 0, 1, 2, $\ldots$, $2^{n-1} - 1$, 2^{n-1}.

2.2.3 Números reales en punto flotante

Convencionalmente, para representar los números reales se introduce el punto decimal[2]. De esta forma, los números positivos menores que la unidad se representan como

$$0.a_1\, a_2 \ldots\, a_n = a_1 10^{-1} + a_2 10^{-2} + \cdots + a_n 10^{-n}.$$

Por ejemplo,

$$0.123 = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 3 \cdot 10^{-3},$$

y de igual forma

$$456.123 = 4 \cdot 10^2 + 5 \cdot 10 + 6 + 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 3 \cdot 10^{-3}.$$

Dado que los números reales pueden tener un número infinito de dígitos, los números reales se representan en un ordenador mediante un formato denominado de punto flotante, que utiliza sólo un número finito de dígitos.

[2] En castellano se suele utilizar la coma decimal, reservando el punto para los millares, por lo que se suele hablar de números en coma flotante. Sin embargo, dado que la mayoría de los programas y lenguajes de ordenador utilizados para problemas numéricos utiliza el punto decimal, hemos preferido utilizar el punto decimal en este curso. Además, no utilizaremos la coma para los millares, para evitar confusiones.

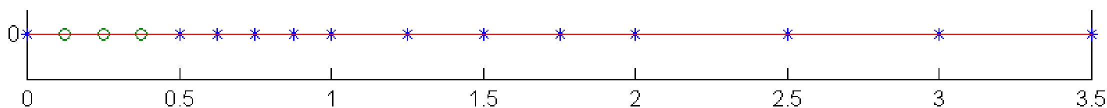


Figura 2.1. Representación de los números flotantes positivos en un sistema $(m, e) = (3, 2)$. Los números con asterisco y círculo son normales y subnormales, respectivamente.

En general, la representación de un número x en punto flotante en una base general b toma la forma

$$x = \pm(0.d_1\, d_2 \ldots d_n)_b\, b^E = \pm \left(\frac{d_1}{b} + \frac{d_2}{b^2} + \cdots + \frac{d_n}{b^n} \right) b^E, \qquad d_1 \neq 0,$$

donde $0.d_1\, d_2 \ldots d_n$ es la mantisa M y E es el exponente entero del número en punto flotante. La condición $d_1 \neq 0$, o de normalización del número, se impone para asegurar la representación única de cada número en punto flotante. Dado un número real $x \in \mathbb{R}$ escribiremos su representación en punto flotante como $fl(x)$, y denotaremos al conjunto (finito) de todos los números flotantes como $\mathbb{F}$, de forma que $fl(x) \in \mathbb{F} \subset \mathbb{R}$.

La precisión, también llamada número de dígitos significativos, de un número flotante viene determinada por el número de dígitos m de su mantisa. Por ejemplo, una mantisa de 24 dígitos binarios corresponde a unos 7 dígitos decimales de precisión, y una de 52 a unos 16. Hay que tener en cuenta que cuanto mayor es la precisión utilizada mayor es el almacenamiento necesario para guardar el número y más tiempo de cómputo es necesario para realizar operaciones con él.

El conjunto de números flotantes no está distribuido de forma uniforme entre el flotante máximo y el mínimo. En la figura 2.1 se representan todos los números flotantes positivos con una mantisa de tres dígitos binarios y un exponente de dos dígitos binarios, es decir, $-1 \leq E \leq 2$. Como se puede ver en la Figura 2.1, la densidad de los números se reduce conforme nos alejamos del origen.

2.2.4 El formato en punto flotante IEEE-754

Existen varios formatos para la representación de números flotantes en un ordenador, aunque el estándar, y por ello el utilizado en la mayoría de los ordenadores, pero no en todos, es el formato ANSI/IEEE standard 754-1985 [1, 2][3], que llamaremos IEEE-754 para abreviar. Este

[3]ANSI significa *American National Standards Institute* y es la organización nacional de estándares de EEUU. IEEE significa *Institute of Electrical and Electronic Engineers* y es la organización científica más importante en

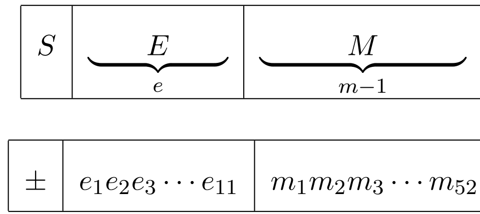


Figura 2.2. Representación de un número flotante en binario con $e + m$ bits (arriba) y representación de un número en doble precisión o de $e + m = 64$ bits (abajo).

estándar ANSI/IEEE fue preconizado por W. Kahan y utiliza un sistema de representación desarrollado previamente por I.B. Goldberg [3]. En este formato los números flotantes se representan en sistema binario (base 2), con m y e bits para la mantisa M y para el exponente E, respectivamente, como aparece en la figura 2.2. De esta forma la representación de un número flotante requiere de $m + e$ bits, número que normalmente es un múltiplo par de 16 (2 *bytes*). Se pueden representar números en precisión simple (*float*), doble (*double*) y cuádruple (*quadruple*) que tienen 32 ($e = 8$), 64 ($e = 11$) y 128 ($e = 15$) bits de longitud, respectivamente[4].

Para concretar estudiaremos el formato de doble precisión, hoy en día el más utilizado, que tiene $m = 53$ bits de mantisa y $e = 11$ bits de exponente. Como el primer dígito de la mantisa, que está normalizada, debe ser necesariamente $d_1 = 1$, se aprovecha este bit para almacenar en su lugar el signo de la mantisa. Este formato permite representar $+0$ y -0, lo que a veces puede ser ventajoso. Se reservan $e = 11$ dígitos binarios para el exponente y su signo. Este número entero se representa en exceso a $2^{e-1} - 1 = 1023$, por lo que se pueden representar sólo los números enteros en el rango $[-2^{e-1} + 1, 2^{e-1}] = [-1022, 1024]$. El exponente máximo 1024, se reserva para representar los números excepcionales $\pm\infty$ y `NaN` (*Not a Number*). Los primeros se representan cuando la mantisa es 0 y se producen cuando una operación aritmética genera un número más grande que el máximo representable, es decir, se produce un desbordamiento por exceso u *overflow*. `NaN` se genera en operaciones aritméticas de resultado no determinado, como $0/0$, $\infty - \infty$, ∞/∞, etc.

De esta forma, dados el bit de signo S, los 52 dígitos binarios de la mantisa M y el exponente E representado como decimal positivo, el número flotante resultante es

$$(-1)^S \times (1.M)_2 \times 2^{E-1023}. \tag{2.1}$$

Es fácil observar que hay números binarios de 64 bits que representan números flotantes con

el mundo de la ingeniería eléctrica.

[4] En el sistema IEEE-754 existen otros tipos de números, los extendidos, que no estudiaremos.

E en binario	E	Valor numérico
$(00000000000)_2$	0	± 0, si $m_i = 0, \forall i$
$(00000000000)_2$	0	$\pm(0.m_1 m_2 m_3 \cdots m_{52})_2 \times 2^{-1022}$
$(00000000001)_2$	1	$\pm(1.m_1 m_2 m_3 \cdots m_{52})_2 \times 2^{-1022}$
$(00000000010)_2$	2	$\pm(1.m_1 m_2 m_3 \cdots m_{52})_2 \times 2^{-1021}$
$\vdots$	$\vdots$	$\vdots$
$(01111111111)_2$	1023	$\pm(1.m_1 m_2 m_3 \cdots m_{52})_2 \times 2^{0}$
$(10000000000)_2$	1024	$\pm(1.m_1 m_2 m_3 \cdots m_{52})_2 \times 2^{1}$
$\vdots$	$\vdots$	$\vdots$
$(11111111110)_2$	2046	$\pm(1.m_1 m_2 m_3 \cdots m_{52})_2 \times 2^{1023}$
$(11111111111)_2$	2047	$\pm\infty$ si $m_i = 0, \forall i$,
$(11111111111)_2$	2047	`NaN` si $\exists i$, $m_i \neq 0$.

Tabla 2.1. Formato binario de todos los números flotantes de doble precisión.

Parámetro	simple	doble	cuádruple
posibles números (2^{m+e})	$\approx 2^{32} = 4.295 \times 10^9$	$\approx 2^{64} = 1.845 \times 10^{19}$	$\approx 2^{128} = 3.403 \times 10^{38}$
precisión (m bits)	24	53	113
épsilon máquina	$2^{-23} = 1.192 \times 10^{-7}$	$2^{-52} = 2.220 \times 10^{-16}$	$2^{-112} = 1.926 \times 10^{-34}$
dígitos decimales	≈ 7	≈ 16	≈ 34
exponente (e bits)	8	11	15
exponentes posibles	$[-126, 127]$	$[-1022, 1023]$	$[-16382, 16383]$
máximo normal	3.403×10^{38}	1.798×10^{308}	1.190×10^{4932}
mínimo normal	1.175×10^{-38}	2.225×10^{-308}	3.362×10^{-4932}
mínimo subnormal	1.401×10^{-45}	4.941×10^{-324}	6.475×10^{-4966}
en FORTRAN	`REAL*4`	`REAL*8`	`REAL*16`
en C	`float`	`double`	`long double`
en Matlab	NO	SI	NO

Tabla 2.2. Resumen de los parámetros de los números simple, doble y cuádruple en formato IEEE-754. ©Copyright 1985 by The Institute of Electrical and Electronics Engineers, Inc. [1]

exponente 0 que no se utilizan en el sistema descrito hasta ahora, ya que la mantisa siempre está normalizada. Para aprovechar dichos valores se definen los números subnormales, y se denomina normales a los que tienen la representación (2.1). Los números subnormales no tienen la mantisa normalizada y permiten representar números mucho más pequeños, en valor absoluto, que los que se obtienen con los números normales. El valor representado por un número subnormal con signo S, mantisa en binario M y exponente decimal siempre $E = 0$ es

$$(-1)^S \times (0.M)_2 \times 2^{-1022}.$$

En la tabla 2.1 se muestra la representación en binario de todos los números flotantes en doble precisión, incluyendo cómo calcular su valor numérico en decimal.

Se denomina épsilon de la máquina ε al valor asociado con al último dígito representable en

la mantisa cuando el exponente es cero, es decir, se define como

$$\varepsilon = min\{\epsilon \in \mathbb{F} : fl(1+\epsilon) \neq 1\}.$$

La representación binaria (no normalizada) de este número es la siguiente

0	$(0\overbrace{11\cdots 11}^{10})_2$	$(\overbrace{000\cdots 000}^{51}1)_2$
$+$	$E=0$	$M=(1)_2$

que nos da su valor en doble precisión $\varepsilon = 2^{-52} = 2.22 \times 10^{-16}$. El lector puede determinar como ejercicio la expresión normalizada del épsilon de la máquina en doble precisión.

En la tabla 2.2 se presenta un resumen de los parámetros más significativos de la representación flotante tanto de los números en precisión doble, como simple y cuádruple. En concreto se presenta una estimación del número total de posibles números, la precisión de éstos o número de bits de la mantisa, el épsilon de la máquina, la precisión en decimal aproximada, el número de bits del exponente, el rango de exponentes enteros posibles, el número normal más grande, el número normal más pequeño, el número subnormal más pequeño, y cómo se representan estos tres formatos en los lenguajes FORTRAN y C. En Matlab sólo se puede utilizar el formato de doble precisión, que de hecho es el más utilizado de todos.

Algunos ejemplos de números representados en flotante aparecen en la tabla 2.3. Con objeto de abreviar la tabla, así como de practicar con el formato IEEE-754 de simple precisión, hemos utilizado éste en lugar del de precisión doble. Es un ejercicio recomendable para el lector el pasar dichos números a dicho formato.

Dado que existe un número finito de números flotantes, como indican las tablas 2.2 y 2.1, y que existe un número flotante más pequeño, mínimo subnormal en la tabla, cuando en una operación aritmética se produce un número aún más pequeño, y por tanto no representable, se dice que se ha producido una excepción por desbordamiento por defecto o de tipo *underflow*. Por otro lado también existe un número flotante mayor, máximo normal en la tabla, y cuando se produce un número aún mayor se eleva una excepción por desbordamiento por exceso o de tipo *overflow*. Para evitar abortar la ejecución del programa cuando se producen excepciones por *overflow*, en el formato IEEE-754 se utilizan los números $\pm\infty$ para representar los estados de desbordamiento por exceso, y para evitarlo en excepciones por *underflow*, en el formato IEEE-754 se generan números subnormales cada vez más pequeños que finalmente alcanzan el valor ± 0.

Número	Signo 1 bit	Exponente 8 bits	Mantisa 23 bits
$7/4$	0	0 1 1 1 1 1 1 1	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-34.432175	1	1 0 0 0 0 1 0 0	0 0 0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0
-959818	1	1 0 0 1 0 0 1 0	1 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0
$+0$	0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-0	1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ε	0	0 1 1 0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
`real_min`	0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
`real_max`	0	1 1 1 1 1 1 1 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
$+\infty$	0	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
`NaN`	0	1 1 1 1 1 1 1 1	1 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0
2^{-128}	0	0 0 0 0 0 0 0 0	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Tabla 2.3. Algunos números y su formato binario en simple precisión [8].

2.3 Errores, condicionamiento y estabilidad numérica

Los números reales representados en punto flotante son los más utilizados, con diferencia, en computación científica. Como hemos observado al normalizar un número real se comete un error. Incluso números con una representación finita en decimal, como 0.1, o 27.9, tienen una representación binaria infinita, tienen decimales binarios periódicos. Por ello, cuando estos números se almacenan en punto flotante se debe "cortar" este número a una cantidad finita de bits, y se incurre en un error de representación flotante. Estos errores, de truncado o redondeo, son inevitables en toda computación científica. Tenemos que estudiar cuánto valen estos errores y cómo medirlos. Además, cuando un ordenador realiza operaciones elementales como sumar o multiplicar números también incurre en un error adicional, que como veremos es similar al de normalización.

Un método numérico conlleva la realización de gran número de operaciones aritméticas, todas y cada una de ellas incurren en errores pequeños. Estos pequeños errores se pueden acumular y crecer catastróficamente conforme aumentan el número de operaciones conduciendo a un resultado final de muy baja exactitud. Estudiaremos como medir los errores de un método numérico como un todo, así como de qué manera determinar si un método sufre crecimiento de los errores (es numéricamente inestable) o no (es numéricamente estable).

Una manera de medir si un método es inestable es estudiando su sensibilidad o condicionamiento a los errores. Veremos cómo se puede calcular una medida de este concepto. A veces, operaciones muy sencilas, como la suma o la resta, pueden incurrir en una gran pérdida de exactitud en el resultado. Como ejemplo presentaremos el caso de la cancelación catastrófica.

El condicionamiento de un algoritmo numérico para resolver un problema dado, su estabilidad numérica, está relacionado con la propia estabilidad del problema original. Un problema inestable conducirá a métodos numéricos inestables. Pero para un problema estable también es posible que algunos métodos sean inestables. De ahí que el condicionamiento y la estabilidad numérica de un método sean factores importantes a estudiar.

2.3.1 Errores relativos de truncado y redondeo

Un punto importante a no olvidar es la diferencia entre precisión y exactitud. Al representar un número real en un ordenador se comete un error. Este error se denomina precisión del número y depende sólo del formato de representación que se utilice. Cuando se obtiene un resultado tras aplicar un método numérico, el error de este resultado nos da la exactitud del mismo. El resultado puede, como número representado, tener una precisión alta aunque su exactitud sea

baja. La exactitud depende de cómo se han acumulado los errores en el resultado. Además, se puede obtener un resultado de mayor exactitud que la precisión con la que se opera, por ejemplo, cuando se utiliza para simular aritmética de mayor precisión, o se utiliza aritmética de precisión infinita.

Para medir este error hay dos procedimientos posibles, calcular su error absoluto y su error relativo (también llamado porcentual).

El error absoluto e_a de un número b como aproximación a otro número a es la diferencia entre ellos, $e_a = b-a$, luego $b = a+e_a$. A veces se define este error en valor absoluto, $e_a = |b-a|$. Por ejemplo, si b aproxima a $a = 22.43$ con un error absoluto $e_a = 0.01221$, entonces $b = 22.44$ coincide con a en aproximadamente 4 dígitos. Sin embargo, con ese error absoluto, pero con $a = 0.02243$, el número $b = 0.03464$ no coincide con a en ninguno de sus dígitos. Aunque un error de 0.01 pueda parecer pequeño, lo será sólo si el número original es grande. Por ello, el error absoluto no nos permite decidir correctamente si un error es grande o no lo es.

El error relativo e_r de un número b como aproximación a otro a se define como

$$e_r = \frac{b-a}{a}, \qquad b = a + e_r\, a = a\,(1+e_r).$$

A veces se define utilizando valores absolutos. El lector observará que para $a = 0$, el error relativo es infinito, lo que es correcto, ya que todo número, por muy pequeño que sea, es infinitamente grande comparado con cero, y todos sus (infinitos) dígitos pueden ser diferentes de cero.

Ilustremos el error relativo con el ejemplo anterior. Sea b un número que aproxima con un error relativo $e_r = 0.01221$ a $a = 22.43$, entonces $b = 22.70$ cuyos dos dígitos más significativos coinciden con a. Para $a = 0.02243$, con el mismo error relativo, tenemos $b = 0.02270$, que también coincide en dos dígitos significativos con a. De esta forma vemos que el error relativo nos indica claramente el número de dígitos en los que coinciden el número original y su aproximación.

Podemos interpretar el error relativo de forma porcentual, si lo multiplicamos por 100. Por ejemplo, el error relativo $e_r = 0.01$ significa un error del 1%. Cuando decimos que dos números se aproximan hasta en un 5% queremos decir que su error relativo es de 0.05.

Para representar un número real en un ordenador hay que elegir un número en punto flotante adecuado. A priori, se debería elegir el más cercano, sin embargo esto no es posible siempre salvo que conozcamos todos los dígitos de dicho número y eso no es posible en muchos casos. En la práctica necesitamos un procedimiento que sólo requiera, como mucho, un único dígito más de los necesarios para representar el número. Utilizando esta condición tenemos que hay dos posibilidades, el truncado y el redondeo.

Supongamos que queremos representar un número real x con una mantisa de m dígitos. El truncado consiste en escoger los m dígitos más significativos de x. No se necesita conocer ningún dígito más del número real a representar. El redondeo, por otro lado, necesita conocer el dígito siguiente, el $(m+1)$-ésimo, y según este dígito esté en $\{0,1,2,3,4\}$, o en $\{5,6,7,8,9\}$, el resultado es diferente. En el primer caso se cogen los m primeros dígitos directamente, como en el truncado. En el segundo caso, se le suma una unidad al dígito menos significativo, el m-ésimo, a los m primeros dígitos, y el resultado es lo que se representa. Consideremos los números reales $2/3 = 0.6666\cdots$, y $0.25977\cdots$, que tiene un número infinito de dígitos en el sistema decimal. Consideremos una representación en punto flotante sencilla con una mantisa de $m = 3$. El redondeo y el truncado conducen a

$$\text{redondeo}(2/3) = (0.666\,6\ldots) = 0.667, \qquad \text{redondeo}(0.25977\ldots) = (0.259\,7\ldots) = 0.260,$$

$$\text{truncado}(2/3) = (0.666\,6\ldots) = 0.666, \qquad \text{truncado}(0.25977\ldots) = (0.259\,7\ldots) = 0.259.$$

¿Cuál de los dos procedimientos es el mejor? Obviamente, el redondeo que, para $2/3$ incurre en un error aproximado de 3×10^{-4} cuando el de truncado es de -6×10^{-4}, aproximadamente el doble, y para $0.25977\ldots$ son de -2×10^{-4} y 7×10^{-4}, respectivamente. El redondeo incurre en un error menor por lo que es la operación usada habitualmente.

Consideremos una mantisa con $m = 3$. El error de truncado máximo ocurre para los números cuya expresión decimal acaba en 9 periódico, como $0.667 = 0.6669999\cdots$, y es igual a $0.9999\cdots\times10^{-4} = 10^{-4}$. Por ello, el error relativo máximo de truncado es igual al último dígito representable (0.001 en el ejemplo previo).

El error de redondeo máximo se produce con los números que tienen una expresión decimal acabada en un 4 más infinitos dígitos 9, como $0.6665 = 0.66649999\cdots$. Estos números, como no se pueden calcular sus infinitos dígitos, son redondeados con el error máximo $4.999\cdots\times10^{-4} = 5\times10^{-4}$. Por ello, el error relativo máximo de redondeo es igual a la mitad del último dígito (0.0005).

Estas conclusiones obtenidas en el sistema decimal son igualmente ciertas en el sistema binario. Considerando de nuevo $m = 3$ pero en base 2, el error de truncado máximo ocurre para los números que en binario acaban en $(1)_2$ periódico, como $(0.01011111\cdots)_2$, que se trunca a $(0.010)_2$, con un error máximo de $(0.00011111\cdots)_2$, es decir, $(0.001)_2$, que es igual al último dígito representable. El truncado de números representados en el sistema IEEE tiene como error máximo exactamente el épsilon de la máquina. Con m dígitos es igual a 2^{-m}.

En binario, cuando redondeamos el número $(0.01011111\cdots)_2$, obtenemos $(0.011)_2$, que es un valor exacto. El error de redondeo máximo lo podemos observar al redondear el número

$(0.01001111\cdots)_2$, que nos da $(0.010)_2$, con error $(0.00001111\cdots)_2 = (0.0001)_2$, que es la mitad del valor del último dígito. Para una mantisa con m dígitos, será 2^{-m-1}.

En resumen, hemos mostrado cómo el error relativo máximo cometido al truncar un número es igual al épsilon de la máquina, ε, y al redondearlo es la mitad, $\varepsilon/2$. Este último valor se denota u, y se denomina unidad de redondeo, $u \equiv \varepsilon/2$. Es la cantidad más útil asociada a un formato de representación flotante, y la más ubicua en el mundo del análisis de errores de redondeo, por ello aparecerá en todos los análisis de errores que presentaremos en este curso. En el sistema IEEE-754 de doble precisión el error máximo de redondeo o unidad de redondeo es igual a 2^{-53}.

2.3.2 Operaciones aritméticas básicas en punto flotante

Las operaciones básicas de la aritmética tienen su equivalente en los ordenadores en las llamadas operaciones en punto flotante, que tienen en cuenta los errores en la representación de los números. Ilustraremos estas operaciones en un formato de representación numérica en decimal. Lo más importante a observar es que estas operaciones siempre incurren en error, que habrá de ser tenido en cuenta en los análisis de errores que realizaremos a lo largo de este curso.

Suma (adición) y resta (sustracción). Para sumar (restar) números flotantes se igualan los exponentes, se suman (restan) las mantisas y luego se normaliza el resultado (se representa como número flotante).

Veamos un ejemplo del modus operandi. Sean $x = 0.235 = 0.235 \cdot 10^0$, e $y = 0.00123 = 0.123 \cdot 10^{-2}$, entonces

$$fl(x+y) = fl(0.235 \cdot 10^0 + 0.00123 \cdot 10^0) = 0.236 \cdot 10^0 \neq 0.23623 = x + y.$$

Como vemos, el resultado tiene error, en este caso, un error absoluto de 0.00023, que corresponde a un error relativo de 9.7×10^{-4}, que indica que todos los dígitos del resultado son correctos.

Tomemos otro ejemplo, sean $x = 1867 = 0.1867 \cdot 10^4$, e $y = 0.32 = 0.3200 \cdot 10^0$, se tiene

$$fl(x+y) = fl(0.1867 \cdot 10^4 + 0.000032 \cdot 10^4) = 0.1867 \cdot 10^4 \neq 1867.32 = x + y,$$

que como vemos también tiene error.

De hecho, aunque no haya error en la representación flotante de los operandos, $x = fl(x)$ e $y = fl(y)$, como ilustran los anteriores ejemplos, casi siempre hay error en el resultado de su suma $x + y \neq fl(x+y)$. Por ejemplo, en el segundo ejemplo el número más pequeño no se ha tenido en cuenta; sin embargo, el resultado es tan exacto como los operandos.

Como último ejemplo tomemos dos números próximos entre sí y consideremos su resta. Sean dos números de 6 dígitos de precisión, $x = 0.467546$, e $y = 0.462301$, que podemos representar sólo con cuatro dígitos como $fl(x) = 0.4675$, y $fl(y) = 0.4623$, y que al ser restados nos dan $fl(x - y) = 0.0052$. Como vemos aunque el resultado es exacto para la suma de los números flotantes, de cuatro dígitos de precisión, no así con respecto a los números originales, que tenían seis dígitos. De hecho el resultado ha perdido dígitos, sólo tiene dos dígitos de precisión, cuando los operandos tenían cuatro. Este fenómeno de pérdida de dígitos de precisión se denomina cancelación, o diferencia cancelativa, y puede ser peligroso (cancelación catastrófica) si el resultado de la resta es utilizado posteriormente en otras operaciones, pues para ellas, uno de los operandos tiene sólo dos dígitos de precisión. Estudiaremos el fenómeno de la cancelación en detalle más adelante.

Producto (multiplicación) y cociente (división). Para multiplicar (dividir) números flotantes se multiplican (dividen) las mantisas, se suman (restan) los exponentes y luego se normaliza el resultado.

Veamos un ejemplo sencillo. Sean $x = 1867 = 0.1867 \cdot 10^4$, e $y = 0.201 = 0.201 \cdot 10^0$, entonces

$$fl(x\,y) = fl(0.1867 \cdot 10^4 \times 0.2010 \cdot 10^0) = fl(0.0375267 \cdot 10^4) = 0.3753 \cdot 10^3 \neq x\,y,$$

y

$$fl(x/y) = fl(0.1867 \cdot 10^4 / 0.2010 \cdot 10^0) = fl(0.928855\ldots \cdot 10^4) = 0.9289 \cdot 10^4 \neq x/y.$$

En relación a las operaciones aritméticas en coma flotante es importante notar varios puntos. Primero, siempre se cometen errores cuando se realizan operaciones aritméticas con ordenadores, es decir,

$$fl(x) \neq x, \quad fl(x \pm y) \neq x \pm y, \quad fl(x\,y) \neq x\,y, \qquad fl(x/y) \neq x/y.$$

Aunque el resultado sea exacto, se puede haber producido una pérdida de dígitos significativos, como en el caso de las diferencias cancelativas, que a todos los efectos podemos interpretar como un error.

Segundo, todos los coprocesadores matemáticos están diseñados para cometer un error máximo igual a la unidad de redondeo, $u = \varepsilon/2$, al realizar cualquiera de las operaciones elementales (incluidas la evaluación de raíces cuadradas, logaritmos, trigonométricas, etc.). Para ello, muchos procesadores utilizan los llamados dígitos de reserva, dígitos adicionales que se utilizan internamente durante los cálculos que permiten garantizar que el resultado de una operación con números flotantes tiene como resultado un número flotante redondeado de forma exacta (por ejemplo, los coprocesadores Intel ix87 utilizan internamente 80 dígitos para operar con números

en doble precisión de sólo 64 dígitos). Un estudio de los efectos de los dígitos de reserva en los errores está fuera de los objetivos de este curso.

Tercero, las operaciones de suma y multiplicación flotantes son conmutativas, pero no son asociativas ni distributivas. Por ejemplo, aunque $x + y + z = (x + y) + z = x + (y + z)$, se tiene que

$$fl(fl(x + y) + z) \neq fl(x + fl(y + z)),$$

y por tanto el orden con el que se opera es importante por cuanto afecta a cómo se acumulan los errores. Un orden adecuado puede llegar a reducir los errores en el resultado final. Y por el contrario, un orden inadecuado puede incrementar éstos.

Y cuarto, en algunos libros se denotan las operaciones flotantes rodeando su símbolo con un círculo, es decir, $\oplus$, $\ominus$, $\otimes$ y $\oslash$ representarán la suma, resta, multiplicación y cociente, respectivamente, de números flotantes, de forma tal que

$$x \oplus y \equiv fl(x + y), \qquad x \otimes y \equiv fl(x\, y),$$

Esta notación puede ser más clara en algunos casos, pero en este curso seguiremos utilizando la notación *fl* que se debe a Wilkinson [14].

2.3.3 Errores progresivos, regresivos y estabilidad numérica

Sea $\hat{y}$ una aproximación numérica a $y = f(x)$ calculada con una aritmética flotante de precisión (unidad de redondeo) u, donde f es una función escalar de variable escalar. ¿Cómo podemos determinar la "calidad" de $\hat{y}$? [9] Lo más directo es estudiar los errores absoluto y relativo de $\hat{y}$,

$$\hat{y} = y + \Delta y, \qquad e_r(\hat{y}) = \frac{|y - \hat{y}|}{|y|}.$$

A estos errores se les denomina errores hacia adelante o progresivos (*forward errors*). En el caso ideal desearíamos que $\hat{y}$ tuviera un error relativo muy pequeño, $e_r(\hat{y}) \approx u$, pero esto en la práctica es muy difícil de conseguir.

Podemos interpretar $\hat{y}$ como la evaluación exacta de la función f sobre un dato inicial perturbado, $\hat{y} = f(x + \Delta x)$. En general, si existe tal Δx, nos gustaría que fuera lo más pequeño posible. El mínimo de todos estos valores se denomina error hacia atrás o regresivo (*backward error*). El error regresivo fue introducido por Wilkinson [14] en la década de 1960 para analizar la propagación de errores en problemas de álgebra lineal numérica [9].

Aunque puede parecer que el error regresivo es más difícil de calcular que el progresivo, Wilkinson demostró que en realidad se da todo lo contrario, y en muchos problemas es más fácil estimar este error.

Se dice que un método para calcular $y = f(x)$ es estable a errores regresivos (*backward stable*) si para todo x produce un valor calculado $\hat{y}$ con un pequeño error hacia atrás, es decir, $\hat{y} = f(x + \Delta x)$ para algún Δx pequeño. La definición de "pequeño" depende del contexto. No todos los métodos para resolver un problema son estables a errores hacia atrás.

En algunos problemas no es posible determinar el error regresivo, pero se puede determinar un error mixto progresivo-regresivo de la forma

$$\hat{y} + \Delta y = f(x + \Delta x), \qquad |\Delta y| \le \epsilon\, |y|, \quad |\Delta x| \le \eta\, |x|,$$

donde ϵ y η son suficientemente pequeños. También se puede definir el error mixto progresivo-regresivo mediante errores relativos

$$\hat{y}(1 + \delta y) = f(x(1 + \delta x)), \qquad |\delta y| \le \epsilon, \quad |\delta x| \le \eta. \tag{2.2}$$

Estas definiciones incluyen como casos particulares al error progresivo y al regresivo.

En general, se dice que un método numérico es numéricamente estable[5] si es estable en el sentido del error mixto progresivo-regresivo. Por ello, un método estable a errores regresivos es numéricamente estable.

2.3.4 Número de condicionamiento

La relación entre los errores progresivos y regresivos en un problema está gobernada por el condicionamiento de éste, es decir, por la sensibilidad de la solución a cambios en los datos iniciales.

Consideremos el problema de evaluar $y = f(x)$ mediante un método que conduce a una solución aproximada $\hat{y}$ que es numéricamente estable a errores mixtos progresivos-regresivos, definimos el número de condición o condicionamiento de dicho método como el cociente entre los errores relativos del resultado y de los datos,

$$\kappa\{f(x)\} = \frac{\delta y}{\delta x}, \qquad \hat{y}(1 + \delta y) = f(x(1 + \delta x)). \tag{2.3}$$

Es importante notar que el número de condición se refiere a un método para evaluar la función, y que varios métodos distintos pueden tener números de condición muy diferentes. A veces, estimaremos el número de condición utilizando cotas de los errores relativos (2.2), $\kappa\{f(x)\} \approx \epsilon/\eta$.

[5]La palabra estable y el concepto de estabilidad numérica se utilizan en métodos numéricos, y por ello aparecerán en este curso, con varios sentidos y significados ligeramente diferentes. El lector debe tener cuidado en diferenciarlos claramente según el contexto.

El concepto de número de condición también se puede aplicar a funciones vectoriales de variable vectorial, dejamos al lector como ejercicio la formalización de esta definición.

Cuando el método para evaluar la función $f(x)$ es estable a errores regresivos, $\hat{y} = f(x+\Delta x)$, y además $f(x)$ es doblemente diferenciable[6], podemos escribir por Taylor

$$\hat{y} - y = f(x+\Delta x) - f(x) = f'(x)\,\Delta x + f''(x+\theta\,\Delta x)\,\frac{\Delta x^2}{2!}, \qquad \theta \in (0,1),$$

que acotando el miembro derecho de esta expresión, el error absoluto cometido, da

$$\frac{\hat{y}-y}{y} = \left(\frac{x\,f'(x)}{f(x)}\right)\frac{\Delta x}{x} + O\big(\Delta x^2\big)\,,$$

de donde obtenemos número de condición como

$$\kappa\{f(x)\} = \left|\frac{x\,f'(x)}{f(x)}\right|.$$

También podríamos haber escrito

$$\hat{y} - y = f(x+\Delta x) - f(x) = f'(x+\mu\,\Delta x)\,\Delta x, \qquad \mu \in (0,1),$$

para obtener

$$\kappa\{f(x)\} = \left|\frac{x\,f'(x+\mu\,\Delta x)}{f(x)}\right| \approx \left|\frac{x\,f'(x)}{f(x)}\right|.$$

En general, para un método numérico que no sea numéricamente estable, se puede definir el número de condición dado un error relativo, acotado por ϵ, como el valor absoluto del cociente entre los errores relativos del resultado y los datos, cuando éstos son perturbado con error relativo acotado por ϵ. En nuestro caso, perturbaremos el argumento de la función con ϵ, lo que nos dará el número de condición,

$$\kappa_\epsilon\,\{f(x)\} = \left|\frac{\dfrac{f(x+\epsilon)-f(x)}{f(x)}}{\dfrac{x+\epsilon-x}{x}}\right| = \left|\frac{f(x+\epsilon)-f(x)}{\epsilon\,f(x)}\,x\right|.$$

De nuevo podemos calcular el número de condicionamiento de una función doblemente diferenciable, $f(x) \in \mathbf{C}^2$, utilizando su desarrollo de Taylor

$$f(x+\epsilon) = f(x) + f'(x)\,\epsilon + O\big(\epsilon^2\big)\,,$$

estimando su número de condicionamiento mediante la expresión aproximada,

$$\kappa\{f(x)\} = \left|\frac{f'(x)\,\epsilon + O\big(\epsilon^2\big)}{\epsilon\,f(x)}\,x\right| \approx \left|\frac{x\,f'(x)}{f(x)}\right|,$$

[6] También se llama función de clase dos.

es decir, que el número de condicionamiento es el cociente entre el cambio en la función, medido por su derivada, y el cambio relativo en el argumento, medido por el cociente $f(x)/x$.

Un número de condición grande indica que la función cambia mucho al experimentar pequeñas variaciones en sus argumentos, y por tanto será muy sensible a los errores de redondeo cometidos al sustituir un número concreto por su representación en punto flotante. Cuando el número de condicionamiento es mucho mayor que la unidad se dice que el problema está mal condicionado; en caso contrario, se dice que el problema está bien condicionado. Como el número de condición depende de x, el condicionamiento de un método para evaluar una función depende también de su argumento.

Ejemplo 2.1 *Considere la función* $f(x) = \log x$.

Su número de condición es

$$\kappa\{f(x)\} = \left|\frac{1}{\log x}\right|,$$

que es muy grande para $x \approx 1$. Ello significa que un pequeño cambio en x produce un cambio muchísimo mayor en $\log x$ para x cercanos a la unidad.

Ejemplo 2.2 . *Considere la suma de* n *números [11],*

$$s = x_1 + x_2 + \cdots + x_n.$$

Su condicionamiento se puede estudiar perturbando ligeramente cada uno de los datos, sea con un error relativo de $|\eta_i| < \epsilon$. Operando con aritmética exacta obtendremos para la suma aproximada

$$\widehat{s} = x_1\,(1+\eta_1) + x_2\,(1+\eta_2) + \cdots + x_n\,(1+\eta_n).$$

Para calcular el número de condición de la suma es necesario estudiar el error relativo de la suma respecto del error relativo de los datos, sea su cota, ϵ. El error absoluto de la suma es

$$\widehat{s} - s = x_1\,\eta_1 + x_2\,\eta_2 + \cdots + x_n\,\eta_n.$$

Tomando valores absolutos y acotando los errores relativos de los sumandos, obtenemos

$$|\widehat{s} - s| \le (|x_1| + |x_2| + \cdots + |x_n|)\,\epsilon,$$

que conduce a la siguiente expresión para el error relativo, y con él para el número de condicionamiento de la suma,

$$\frac{|\widehat{s} - s|}{|s|} \le \kappa\,\epsilon, \qquad \kappa = \frac{|x_1| + |x_2| + \cdots + |x_n|}{|x_1 + x_2 + \cdots + x_n|}.$$

Este número de condicionamiento puede ser grande, indicando una pérdida de exactitud en el resultado de la suma. Las diferencias cancelativas son un claro ejemplo de número de condición grande, ya que para ellas, $|x_1| + |x_2| \gg |x_1 + x_2|$, con $x_2 < 0 < x_1$.

La importancia del número de condición no está sólo en que nos acota (muchas veces de forma muy pesimista) el error relativo del resultado en función del de los datos, sino que nos indica posibles causas por las cuales la evaluación de la función es susceptible de sufrir grandes errores. Estas causas, a veces, pueden ser evitadas con una reescritura conveniente de la función a evaluar, con el correspondiente incremento en exactitud para el resultado. Por ejemplo, para la suma, sumar por un lado todos los operandos positivos, y por otro los negativos, y luego restar el resultado conduce a una reducción en el número de condicionamiento y por tanto un menor error en el resultado final.

Hay una regla empírica observada casi siempre entre el error progresivo, el regresivo y el número de condicionamiento cuando éstos se definen de una forma consistente entre sí [9],

$$\text{error progresivo} \lesssim \text{número de condición} \times \text{error regresivo},$$

donde muchas veces se da la igualdad. Esta regla empírica significa que la solución calculada para un problema mal condicionado puede tener un gran error progresivo, incluso si tiene un error regresivo pequeño.

2.3.5 Cancelación catastrófica

En secciones anteriores hemos observado que al sumar (restar) números de diferente (igual) signo, y de módulo muy parecido, se produce una pérdida de dígitos significativos en el resultado, incluso cuando éste es exacto. Este fenómeno se denomina cancelación. En principio, la cancelación no es algo malo. Sin embargo, si el resultado de ésta se utiliza en operaciones sucesivas, la pérdida de dígitos significativos puede generar la propagación de errores que pueden reducir fuertemente la exactitud del resultado final, y se produce una cancelación catastrófica.

Ejemplo 2.3 *Ilustremos la diferencia entre cancelación y cancelación catastrófica.*

Sean $x = 0.732112$, $y = 0.732110$. Con aritmética de 5 dígitos de precisión (tamaño de la mantisa), éstos y su diferencia se representan como sigue

$$fl(x) = 0.73211, \qquad fl(y) = 0.73211,$$

$$fl(x-y) = 0 \neq x - y = 0.000002 = 0.2 \times 10^{-5}.$$

Como vemos en este caso límite, se han perdido todos los dígitos significativos en el resultado, sin embargo, éste es el valor exacto de la resta de los números flotantes. Esta cancelación no es realmente peligrosa. Sin embargo, si utilizamos este resultado en una operación posterior, como

$$fl(fl(x-y)*y) = 0 \neq (x-y)*y = 0.14642 \times 10^{-5},$$

el resultado es completamente inexacto. Todos los dígitos son erróneos, se ha producido una cancelación catastrófica.

Ejemplo 2.4 *Consideremos la función* $f(x) = 1 - \cos(x)$.

Incluso en la evaluación de una función tan sencilla como ésta, se pueden producir cancelaciones catastróficas. Como $f(0) = 0$, para números pequeños, $|x| \approx 0$, se producirá una pérdida de dígitos significativos. El número de condición de esta función nos permite comprobar esta situación, ya que

$$\kappa\{f(x)\} = \left|\frac{x\,f'(x)}{f(x)}\right| = \left|\frac{x\,\sin(x)}{1-\cos(x)}\right|,$$

que para $|x| \ll 1$ nos da un número de condición que crece hacia infinito.

El lector observará que para $|x| \to 0$, podemos aplicar Taylor para evaluar el número de condición, con lo que obtenemos

$$\kappa\{f(x)\} = 2 - \frac{x^2}{6} + O\left(x^4\right),$$

que indica que el problema está bien condicionado. ¿Cómo es posible ésto? En realidad, al aplicar Taylor estamos cambiando el método numérico para evaluar la función $f(x)$ lo que implica que el número de condición puede cambiar. Si evaluamos la función $f(x)$ cerca de 0 mediante serie de Taylor obtenemos

$$f(x) = 1 - \left(1 - \frac{x^2}{2} + O\left(x^4\right)\right) = \frac{x^2}{2} + O\left(x^4\right),$$

que claramente no tiene diferencia cancelativa alguna. El número de condición que hemos obtenido mediante el desarrollo en serie de Taylor nos informa que aplicar Taylor a la función nos permite evitar la diferencia cancelativa.

Este ejemplo muestra como, en muchos casos, es posible evitar las diferencias cancelativas con una re-escritura adecuada de la función a evaluar para los valores para los que éstas se dan. También ilustra que para evitar diferencias cancelativas es necesario determinar para qué valores del argumento se producen éstas, y aproximar la función para valores cercanos utilizando desarrollo en serie de Taylor. En algunas ocasiones es posible evitar el carácter aproximado del

desarrollo de Taylor utilizando una reescritura apropiada de la función, sin embargo, esto no es posible siempre y, en su caso, requiere cierta práctica e intuición, como ilustra el siguiente ejemplo.

Ejemplo 2.5 *Cálculo de las raíces de una ecuación cuadrática.*

Las raíces de una ecuación cuadrática (o de segundo grado) se pueden obtener mediante la fórmula clásica

$$a\,x^2 + b\,x + c = 0, \qquad x_\pm = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2\,a}.$$

Para $b^2 > 4\,a\,c$ hay dos raíces reales. Sin embargo, para $b^2 \gg 4\,a\,c$, se puede producir una cancelación catastrófica para la raíz x_+ si $b > 0$, o para la raíz x_- si $b < 0$; en cada caso la otra raíz no sufre problema alguno.

Ilustremos este fenómeno con un ejemplo [13]. Para $a = 1$, $b = -10^5$ y $c = 1$, que cumple que $b^2 \gg 4\,a\,c$, las raíces exactas redondeadas a 11 dígitos decimales son

$$x_+ = 99999.999990, \qquad x_- = 0.000010000000001,$$

sin embargo si utilizamos la fórmula y una calculadora con 11 dígitos decimales obtendremos, como el lector puede comprobar fácilmente a mano,

$$x_+ = 100000.00, \qquad x_- = 0,$$

donde vemos que la primera raíz se ha obtenido con gran exactitud, pero la segunda es completamente inexacta. Esta raíz ha sufrido una cancelación catastrófica.

Consideremos el caso $b > 0$. Como en el ejemplo anterior, utilizando Taylor adecuadamente, podemos evitar este problema,

$$x_+ = \frac{-b + \sqrt{b^2 - 4\,a\,c}}{2\,a} = \frac{-b + b\sqrt{1 - \frac{4\,a\,c}{b^2}}}{2\,a}$$
$$\approx \frac{1}{2\,a}\left(-b + b\left(1 - \frac{2\,a\,c}{b^2} + \cdots\right)\right) = -\frac{c}{b} + O\left(\frac{1}{a}\left(\frac{a\,c}{b^2}\right)^2\right).$$

Más aún, incluso sin utilizar Taylor, operando de la siguiente forma, también podemos evitar la cancelación,

$$x_+ = \frac{-b + \sqrt{b^2 - 4\,a\,c}}{2\,a} = \frac{\left(-b + \sqrt{b^2 - 4\,a\,c}\right)\left(-b - \sqrt{b^2 - 4\,a\,c}\right)}{2\,a\left(-b - \sqrt{b^2 - 4\,a\,c}\right)} = \frac{-2\,c}{b + \sqrt{b^2 - 4\,a\,c}}.$$

El caso $b < 0$ se trata exactamente igual. De hecho, se puede utilizar la fórmula siguiente

$$x_1 = \frac{-(b + \operatorname{sign}(b)\sqrt{b^2 - 4\,a\,c})}{2\,a}. \tag{2.4}$$

Una vez determinada una de las raíces de la ecuación, la otra se puede obtener fácilmente, ya que

$$a\,(x - x_1)\,(x - x_2) = a\,x^2 - a\,(x_1 + x_2)\,x + a\,x_1\,x_2 = 0,$$

conduce a

$$x_2 = \frac{c}{a\,x_1}. \tag{2.5}$$

Aplicando las expresiones (2.4) y (2.5) al ejemplo numérico anterior, obtenemos

$$x_1 = 100000.00, \qquad x_- = \frac{c}{a\,x_1} = 0.00001000000,$$

donde ahora las dos expresiones tienen una exactitud bastante buena.

Además de la cancelación que hemos observado, existe otra para $b^2 \approx 4\,a\,c$, que se da para el caso de raíces reales dobles. Sin embargo, esta cancelación no es catastrófica, como el lector puede observar fácilmente. El hecho de que no todas las diferencias cancelativas sean catastróficas, es decir, correspondan a problemas mal condicionadas, es algo a lo que el lector debe prestar especial atención. Por ello, mostraremos un ejemplo más.

Ejemplo 2.6 *Diferencia cancelativa no catastrófica.*

Como indicado previamente, el carácter catastrófico de una cancelación sólo aparece cuando el resultado de ésta se utiliza en operaciones sucesivas; si no es así, no hay problemas con el uso de la misma. Consideremos la evaluación de la función

$$f(x) = \sqrt{1 + x} - \sqrt{x},$$

para $|x| \gg 1$. Se produce claramente una cancelación, que se puede evitar aplicando Taylor,

$$f(x) = \sqrt{x}\left(\sqrt{1 + \frac{1}{x}} - 1\right) = \sqrt{x}\left(1 + \frac{1}{2\,x} + O\left(\frac{1}{x^2}\right) - 1\right)$$

$$= \frac{\sqrt{x}}{2\,x} + O\left(\frac{\sqrt{x}}{x^2}\right),$$

o si se prefiere, sin aplicar Taylor, como

$$f(x) = \frac{(\sqrt{1+x} - \sqrt{x})(\sqrt{1+x} + \sqrt{x})}{\sqrt{1+x} + \sqrt{x}} = \frac{1}{\sqrt{1+x} + \sqrt{x}}.$$

Calculando el número de condición de la función $f(x)$, para lo que requerimos su derivada

$$f'(x) = \frac{1}{2\sqrt{1+x}} - \frac{1}{2\sqrt{x}} = \frac{\sqrt{x}-\sqrt{1+x}}{2\sqrt{x\,(1+x)}}$$

obtenemos

$$\approx \left| \frac{\sqrt{x}-\sqrt{1+x}}{2\sqrt{x\,(1+x)}} \, \frac{x}{\sqrt{1+x}-\sqrt{x}} \right| = \left| \frac{x}{2\sqrt{x\,(1+x)}} \right| = \left| \frac{1}{2}\sqrt{\frac{x}{1+x}} \right|.$$

Como vemos, el número de condicionamiento para $x \gg 1$ es

$$\kappa\{f(x)\} \approx \frac{1}{2},$$

con lo que este problema está bien condicionado, a pesar de que se produce una diferencia cancelativa. Ello indica que esta diferencia cancelativa no es catastrófica, aunque implica una pérdida de dígitos significativos en el resultado.

En general, debemos indicar que la evaluación de funciones mal condicionadas puede producir pérdidas significativas de dígitos en el resultado, una reducción en la exactitud del mismo; sin embargo, a diferencia de muchas cancelaciones catastróficas, normalmente no es posible reescribir la expresión de la función de forma tal que se evite este mal condicionamiento.

Ejemplo 2.7 *Condicionamiento del polinomio de Wilkinson.*

Como ejemplo final vamos a considerar el polinomio debido a Wilkinson [14],

$$p(x) = (x-1)\,(x-2)\,\cdots\,(x-19)\,(x-20) = x^{20} - 210\,x^{19} + \cdots,$$

cuyos ceros son los números 1, 2, ..., 19, 20. Wilkinson utilizando aritmética binaria con una mantisa de 90 dígitos de precisión, encontró que las raíces del polinomio ligeramente perturbado $p(x) + 2^{-23}\,x^{19}$, tenían un gran error (de hecho algunas eran pares de números complejos conjugados). Esto es indicativo de que el cálculo de las raíces de un polinomio es un problema mal condicionado.

Para calcular el número de condición de un método para calcular las raíces de este polinomio, supondremos que sólo se produce error en el segundo coeficiente,

$$p(x,\alpha) = x^{20} - \alpha\,x^{19} + \cdots,$$

cuyas raíces son la función que queremos evaluar, por tanto su número de condición es

$$\kappa\{x(\alpha)\} = \frac{\alpha}{x}\,\frac{\partial x}{\partial \alpha}.$$

Derivando la ecuación $p(x,\alpha)=0$ con respecto a α, obtenemos

$$\frac{\partial p(x,\alpha)}{\partial x}\,\frac{\partial x}{\partial \alpha}+\frac{\partial p(x,\alpha)}{\partial \alpha}=0, \qquad \frac{\partial x}{\partial \alpha}=-\frac{\partial p/\partial \alpha}{\partial p/\partial x}.$$

Necesitamos conocer la derivada del polinomio $\partial p/\partial x$. Deduciremos la fórmula inductivamente, comenzando con $n=2$, la derivada del polinomio es

$$p_2(x)=(x-1)\,(x-2), \qquad p_2'(x)=(x-1)+(x-2),$$

y para $n=3$,

$$p_3(x)=(x-1)\,(x-2)\,(x-3), \qquad p_3'(x)=(x-2)\,(x-3)+(x-1)\,(x-3)+(x-1)\,(x-2).$$

Estas expresiones se pueden generalizar fácilmente para el caso general,

$$p_n(x)=\prod_{i=1}^{n}(x-j), \qquad \frac{\partial p(x)}{\partial x}=\sum_{i=1}^{n}\prod_{\substack{j=1\\ j\neq i}}^{n}(x-j).$$

Para nuestro polinomio con $n=20$, obtenemos finalmente

$$\frac{\partial x}{\partial \alpha}=-\frac{\partial p/\partial \alpha}{\partial p/\partial x}=\frac{x^{19}}{\displaystyle\sum_{i=1}^{20}\prod_{\substack{j=1\\ j\neq i}}^{20}(x-j)},$$

que evaluado en cada raíz x_i de $p(x_i,\alpha)=0$, nos da el número de condición para cada una de ellas,

$$\kappa\{x_i=i\}=\frac{\alpha}{i}\,\frac{i^{19}}{\displaystyle\prod_{\substack{j=1\\ j\neq i}}^{20}(i-j)}, \qquad i=1,2,\ldots,19,20.$$

Evaluando sólo la derivada $\partial x/\partial \alpha$ en las raíces nos podemos hacer una idea del tamaño de este número,

raíz	$\partial x/\partial \alpha\|_{x=i}$
1	-8.2×10^{-18}
2	8.2×10^{-11}
$\vdots$	$\vdots$
16	2.4×10^{9}
$\vdots$	$\vdots$
19	-3.1×10^{8}
20	4.3×10^{7}

con lo que observamos que el número de condición es grande para las raíces de mayor módulo. Ello implica que cualquier método numérico para determinar las raíces de polinomios conducirá a grandes errores cuando se aplique a polinomios mal condicionados como el de Wilkinson.

2.4 Análisis de la propagación de los errores

Dado que en todas las operaciones computacionales con ordenadores se cometen errores, es necesario analizar cómo se propagan éstos cuando se aplican sucesivamente muchas operaciones.

Para estimar los errores cometidos en un algoritmo numérico se pueden utilizar cualquiera de los siguientes procedimientos:

1. Usar un análisis de propagación de errores hacia adelante o progresivo.
2. Usar un análisis de propagación de errores hacia atrás o regresivo.
3. Desde un punto de vista práctico, realizar el cálculo con precisión simple y luego con precisión doble (o cuádruple) y comparar los diferencias (errores) entre los resultados obtenidos.
4. Usar aritmética por intervalos en la que se sustituyen los números x por intervalos $[x - \epsilon, x + \epsilon]$ y realizar todas las operaciones aritméticas con los intervalos en lugar de los números.

5. Utilizar aritmética "infinita" (números racionales), en la que se preservan todos los dígitos significativos en las operaciones aritméticas simples.

6. Usar métodos estadísticos en los que se asume que los errores son variables aleatorias independientes.

En este curso estudiaremos sólo las técnicas básicas para el análisis de la propagación de los errores, el análisis hacia adelante y el análisis hacia atrás. La aritmética por intervalos, la aritmética "infinita" y los métodos estadísticos están fuera de los contenidos de este curso.

2.4.1 Modelo estándar para la aritmética flotante

A la hora de realizar una análisis de la propagación de errores es necesario realizar una serie de hipótesis sobre la exactitud de las operaciones aritméticas básicas. Estas hipótesis son dependientes del tipo de formato de representación flotante que se utilice y de los detalles de la implementación del coprocesador en punto flotante de la máquina en la que se ejecute el algoritmo.

El modelo más habitual para describir el comportamiento del error en las operaciones básicas se denomina modelo estándar y nos dice que para todo $x, y \in \mathbb{F}$,

$$\mathit{fl}(x \text{ op } y) = (x \text{ op } y)\,(1+\delta), \qquad |\delta| \leq u, \quad \text{op } \in \{+,-,*,/\}, \tag{2.6}$$

donde u es la unidad de redondeo, la mitad del épsilon de la máquina. Normalmente se asume que (2.6) se cumple también para la operación de extracción de la raíz cuadrada. Este modelo es válido cuando se utiliza aritmética con el estándar IEEE-754, de ahí su nombre.

El modelo estándar nos dice que el error relativo del valor calculado de x op y es igual al error relativo de redondear el valor exacto. El lector notará que este modelo no requiere que el error relativo sea $\delta = 0$ cuando el resultado exacto sea representable correctamente, x op $y \in \mathbb{F}$.

A veces se facilita el análisis de errores si se usa una versión ligeramente modificada pero equivalente del modelo estándar,

$$\mathit{fl}(x \text{ op } y) = \frac{x \text{ op } y}{1+\delta}, \qquad |\delta| \leq u. \tag{2.7}$$

El lector puede comprobar fácilmente que esta expresión es equivalente a (2.6), por ello, durante este curso utilizaremos ambas expresiones indistintamente.

Algoritmo 2.2 *Producto interior o escalar de dos vectores.*

```
function s = productoInterior (x, y)
% x, y, son vectores de números flotantes
% s es su producto interior
%
  n=length(x)
  s = 0
  for i=1:n
    s = s + x(i)*y(i)
  end
end
```

2.4.2 Análisis de errores para el producto interior y el exterior

Como ejemplo de análisis de errores vamos a presentar los análisis de propagación de errores progresivo (o hacia adelante) y regresivo (o hacia atrás) para la operación de calcular el producto interior de dos vectores. Sean $x, y \in \mathbb{R}^n$, queremos calcular

$$s_n = x^\top y = \sum_{i=1}^{n} x_i\, y_i = x_1\, y_1 + x_2\, y_2 + \cdots + x_n\, y_n.$$

Para calcular esta expresión procederemos paso a paso, siguiendo el método,

$$s_1 = x_1\, y_1, \qquad s_i = s_{i-1} + x_i\, y_i, \quad i = 2, 3, \ldots, n,$$

que aparece en pseudocódigo en el algoritmo 2.2.

Utilizando el modelo estándar (2.6), obtenemos para el primer sumando

$$\widehat{s}_1 = fl(x_1\, y_1) = x_1\, y_1\,(1 + \delta_1), \qquad |\delta_1| \le u,$$

para el segundo

$$\begin{aligned}
\widehat{s}_2 &= fl(\widehat{s}_1 + x_2\, y_2) = (\widehat{s}_1 + x_2\, y_2\,(1 + \delta_2))\,(1 + \delta_{s,1}) \\
&= (x_1\, y_1\,(1 + \delta_1) + x_2\, y_2\,(1 + \delta_2))\,(1 + \delta_{s,1}) \\
&= x_1\, y_1\,(1 + \delta_1)\,(1 + \delta_{s,1}) + x_2\, y_2\,(1 + \delta_2)\,(1 + \delta_{s,1}), \qquad |\delta_2|, |\delta_{s,1}| \le u,
\end{aligned}$$

para el tercero

$$\widehat{s}_3 = fl(\widehat{s}_2 + x_3\, y_3) = (\widehat{s}_2 + x_3\, y_3\,(1 + \delta_3))\,(1 + \delta_{s,2})$$

$$
\begin{aligned}
&= x_1\, y_1\, (1+\delta_1)\,(1+\delta_{s,1})\,(1+\delta_{s,2}) \\
&\qquad + x_2\, y_2\, (1+\delta_2)\,(1+\delta_{s,1})\,(1+\delta_{s,2}) \\
&\qquad + x_3\, y_3\, (1+\delta_3)\,(1+\delta_{s,2}), \qquad |\delta_3|, |\delta_{s,2}| \le u,
\end{aligned}
$$

y así sucesivamente se obtiene finalmente

$$
\begin{aligned}
\widehat{s}_n &= x_1\, y_1\, (1+\delta_1) \prod_{i=1}^{n-1} (1+\delta_{s,i}) \\
&\quad + x_2\, y_2\, (1+\delta_2) \prod_{i=1}^{n-1} (1+\delta_{s,i}) \\
&\quad + x_3\, y_3\, (1+\delta_3) \prod_{i=2}^{n-1} (1+\delta_{s,i}) + \cdots \\
&\quad + x_{n-1}\, y_{n-1}\, (1+\delta_{n-1})\,(1+\delta_{s,n-2})\,(1+\delta_{s,n-1}) \\
&\quad + x_n\, y_n\, (1+\delta_n)\,(1+\delta_{s,n-1}), \qquad |\delta_1|, |\delta_i|, |\delta_{s,i}| \le u, \quad i = 2, 3, \ldots, n.
\end{aligned}
$$

Para interpretar correctamente esta expresión es conveniente simplificarla un poco. Introduzcamos los coeficientes

$$
\begin{aligned}
1+\theta_n &= (1+\delta_1)\,(1+\delta_{s,1})\,(1+\delta_{s,2}) \,\cdots\, (1+\delta_{s,n-1}), \\
1+\theta'_n &= (1+\delta_2)\,(1+\delta_{s,1})\,(1+\delta_{s,2}) \,\cdots\, (1+\delta_{s,n-1}), \\
1+\theta_{n-1} &= (1+\delta_3)\,(1+\delta_{s,2}) \,\cdots\, (1+\delta_{s,n-1}), \\
&\ \ \vdots \\
1+\theta_3 &= (1+\delta_{n-1})\,(1+\delta_{s,n-2})\,(1+\delta_{s,n-1}) \\
1+\theta_2 &= (1+\delta_n)\,(1+\delta_{s,n-1}),
\end{aligned}
$$

con lo que obtenemos

$$
\widehat{s}_n = x_1\, y_1\, (1+\theta_n) + x_2\, y_2\, (1+\theta'_n) + x_3\, y_3\, (1+\theta_{n-1}) + \cdots + x_n\, y_n\, (1+\theta_2). \tag{2.8}
$$

Esta es una expresión del error regresivo o hacia atrás en el producto interior, y se interpreta como sigue: el producto interior calculado es igual al obtenido utilizando aritmética exacta pero con un conjunto de datos perturbado, es decir, x_1, $y_1\,(1+\theta_n)$, x_2, $y_2\,(1+\theta'_n)$, x_3, $y_3\,(1+\theta_{n-1})$,

$\ldots$, x_n, $y_n\,(1+\theta_2)$. También se podría haber perturbado los valores de x_i y dejar los y_i sin perturbar, sean $x_1\,(1+\theta_n)$, y_1, $x_2\,(1+\theta_n')$, y_2, $x_3\,(1+\theta_{n-1})$, y_3, $\ldots$, $x_n\,(1+\theta_2)$, y_n. Estos dos casos son equivalentes.

Para interpretar correctamente el significado de la expresión (2.8) es necesario acotar el tamaño de los números θ_i, que se denominan errores relativos hacia atrás. Esta acotación nos dirá qué términos introducen más error en el resultado final, y además nos indicará si son pequeños, lo que garantizará la estabilidad numérica de la operación.

Para intuir qué tipo de cota esperamos obtener, vamos a considerar el caso más sencillo

$$1+\theta_2 = (1+\delta_n)\,(1+\delta_{s,n-1}) = 1 + (\delta_n + \delta_{s,n-1}) + \delta_n\,\delta_{s,n-1}.$$

Como los errores relativos $|\delta_i|, |\delta_{s,i}| \le u$, y $u^2 \ll u$ (en doble precisión $u \approx 10^{-16}$ y $u^2 \approx 10^{-32}$), podemos acotar de forma aproximada

$$|\theta_2| \le |\delta_n + \delta_{s,n-1}| + |\delta_n\,\delta_{s,n-1}| \le 2\,u + O\big(u^2\big)\,,$$

donde u es la unidad de redondeo. De forma del todo similar, el lector observará que

$$|\theta_3| \le 3\,u + O\big(u^2\big)\,, \qquad \cdots, \qquad |\theta_n| \le n\,u + O\big(u^2\big)\,, \qquad |\theta_n'| \le n\,u + O\big(u^2\big)\,.$$

Como vemos el subíndice indica el número de las unidades de redondeo que aparecen en la cota (no rigurosa) que hemos obtenido. Esta cota nos indica que los primeros sumandos introducen más error que los últimos, por lo que el error final es menor si se ordenan los sumandos de la forma $x_i\,y_i \le x_{i+1}\,y_{i+1}$. En la práctica, ordenar los sumandos de esta forma no es fácil.

Dado que el producto interior forma parte de muchos algoritmos numéricos más complejos, muchos coprocesadores numéricos implementan directamente en hardware esta operación, de tal forma que el resultado tenga un error menor que u. Para ello, estos procesadores tienen que retener internamente un número suficiente de bits adicionales (llamados bits de reserva) para la mantisa. En muchos coprocesadores con unidad aritmética encauzada (*pipeline*) se opera internamente con registros de 80 bits y se redondea el resultado final a la mantisa de 53 bits que se utiliza en doble precisión. En estos procesadores el modelo estándar que hemos utilizado como base de nuestro análisis no es aplicable. Omitiremos un análisis de este tipo en este curso.

El error relativo θ_n de cada sumando en la expresión (2.8) es muy pequeño, lo que indica que el producto interior es estable numéricamente en el sentido de los errores regresivos.

Podemos obtener una cota rigurosa de una manera sencilla. Como para $0 < x \ll 1$, se tiene por el teorema de Taylor que

$$1+x \le \exp(x) = 1 + x + \frac{\eta^2}{2!}, \qquad 0 < \eta < x,$$

y $|1+\delta_i| \le (1+u)$, haciendo $\delta_{s,n} \equiv \delta_1$ para abreviar, tenemos que

$$|\theta_n| = \left|\prod_{i=1}^{n}(1+\delta_{s,i}) - 1\right| \le (1+u)^n - 1 \le \exp(n\,u) - 1$$

$$\le n\,u\,\left(1 + n\,u + (n\,u)^2 + (n\,u)^3 + \cdots\right) = \frac{n\,u}{1-n\,u}.$$

En resumen, se cumple la cota rigurosa

$$\theta_n \le \frac{n\,u}{1-n\,u} \equiv \gamma_n,$$

siempre y cuando $n\,u < 1$. Esta última hipótesis se cumple virtualmente siempre que se utiliza aritmética IEEE tanto en doble como en simple precisión. En este curso, utilizaremos la notación θ_n y γ_n en reiteradas ocasiones.

De hecho, se puede obtener una cota rigurosa más próxima a nuestro análisis no riguroso en el que se desprecian términos cuadráticos. Imponiendo la condición $n\,u < 0.01$, podemos escribir [9]

$$\prod_{i=1}^{n}(1+\delta_{s,i}) = 1 + \theta_n, \qquad |\theta_n| \le 1.01\,n\,u = n\,\tilde{u},$$

donde $\tilde{u}$ se denomina unidad de redondeo ajustada. La demostración de este resultado es sencilla,

$$|\theta_n| = \left|\prod_{i=1}^{n}(1+\delta_{s,i}) - 1\right| \le (1+u)^n - 1 \le \exp(n\,u) - 1$$

$$\le n\,u\left(1 + \frac{n\,u}{2} + \left(\frac{n\,u}{2}\right)^2 + \left(\frac{n\,u}{2}\right)^3 + \cdots\right) = \frac{n\,u}{1-n\,u/2}$$

$$< \frac{n\,u}{0.995} < 1.01\,n\,u.$$

Note que este resultado es más fuerte que la cota correspondiente que habíamos obtenido previamente[7]

$$|\theta_n| \le \frac{n\,u}{1-n\,u} < \frac{n\,u}{0.99} = 1.0101\cdots\,n\,u.$$

Estas acotaciones, rigurosas y no rigurosas, nos indican, en resumen, que en todos los análisis de errores se asume tácitamente que el tamaño del problema es suficientemente pequeño ($n\,u < 0.01$) como para que las cotas no rigurosas de la forma $\theta_n \lesssim n\,u$ se puedan sustituir rigurosamente por $\theta_n \le n\,\tilde{u}$ donde $\tilde{u}$ es la unidad de redondeo ajustada, $\tilde{u} = 1.01\,u$.

[7] Como nota histórica hemos de indicar que Wilkinson en 1965 introdujo la unidad de redondeo ajustada con $n\,u < 0.1$ como $\tilde{u} = 1.06\,u$. Nos parece más acertada la versión que presentamos, debida a Higham [9].

Volvamos de nuevo al producto interior. Hemos observado que la expresión (2.8) se cumple para cualquier ordenación posible de los sumandos. Utilizando notación vectorial, podemos escribir sintéticamente

$$fl(x^\top y) = (x + \Delta x)^\top y = x^\top (y + \Delta y), \qquad |\Delta x| \leq \gamma_n |x|, \qquad |\Delta y| \leq \gamma_n |y|, \tag{2.9}$$

donde $x^\top$ la traspuesta matricial del vector x, es decir, un vector columna, $|x|$ denota el vector cuyos elementos son $|x_i|$, y las desigualdades entre vectores (más tarde utilizaremos también matrices) se cumplen componente a componente.

Una cota de error progresivo o hacia adelante se puede obtener fácilmente a partir de (2.9) como

$$|x^\top y - fl(x^\top y)| \leq \gamma_n |x|^\top |y| = \gamma_n \sum_{i=1}^{n} |x_i y_i|.$$

Este resultado muestra que para el cálculo de $\|x\|^2 = x^\top x$ (es decir, $y = x$), el resultado obtenido tiene un gran exactitud relativa. Sin embargo, ese resultado no es cierto siempre, ya que no está garantizada la gran exactitud relativa si $|x^\top y| \ll |x|^\top |y|$.

Hemos observado que el producto interior de vectores es numéricamente estable. Podemos estudiar qué pasa con el producto exterior $A = x y^\top$, donde $x, y \in \mathbb{R}^n$, y el resultado es una matriz cuadrada. El análisis es muy sencillo. Los elementos del resultado cumplen

$$a_{ij} = x_i y_j, \qquad \widehat{a}_{ij} = x_i y_j (1 + \delta_{ij}), \qquad |\delta_{ij}| \leq u,$$

con lo que, en notación matricial con desigualdades componente a componente,

$$\widehat{A} = x y^\top + \Delta A, \qquad |\Delta A| \leq u |x y^\top|.$$

Como vemos el resultado es muy exacto, sin embargo este cálculo no es estable a errores regresivos, ya que no existen Δx y Δy pequeños tales que $\widehat{A} = (x + \Delta x)(y + \Delta y)^\top$, como el lector puede comprobar fácilmente.

La distinción entre los productos interior y exterior ilustra un principio general del análisis de errores: un proceso numérico es más posible que sea numéricamente estable a errores regresivos cuando el tamaño del resultado es pequeño comparado con el tamaño de los datos de entrada, de tal manera que hay un gran número de datos entre los que repartir el error hacia atrás. El producto interior tiene el tamaño mínimo posible como salida (un número) para su entrada ($2n$ números), mientras que el producto exterior tiene el máximo posible (dentro del álgebra lineal) como salida (n^2).

2.4.3 El propósito del análisis de errores

Antes de presentar análisis de errores de otros problemas es necesario considerar la utilidad de dicho tipo de análisis, cuyo propósito es mostrar la existencia de cotas a priori para medidas apropiadas del efecto de los errores de redondeo en un método numérico. Lo importante es que dicha cota exista, no así, el valor exacto de las constantes implicadas. Muchos analistas numéricos calculan dichas cotas de forma no rigurosa, utilizando la notación O de Landau. El valor de dichas constantes suele ser una estimación muchos órdenes de magnitud por encima de su valor exacto.

En el caso ideal esperamos que las cotas de error sean pequeñas para cualquier elección de los datos de entrada al algoritmo numérico. Cuando no lo son, estas cotas revelan características del problema que caracterizan su potencial inestabilidad, y sugieren como dicha inestabilidad puede ser corregida o al menos evitada. Por supuesto, hay algoritmos inestables para los que no existen acotaciones útiles del error. Detectar estos algoritmos es importante para evitar su uso en aplicaciones prácticas.

Cuando se desean acotaciones "buenas" del error cometido hay que trabajar a posteriori, es decir, calcular estas cotas junto con la aplicación del algoritmo, conforme el resultado se va calculando. A este tipo de análisis se le denomina análisis de error a posteriori. Como ejemplo, estudiaremos este tipo de análisis para el producto interior, basado en el algoritmo 2.2. Escribiendo las sumas parciales calculadas como $\widehat{s}_i = s_i + e_i$, y utilizando el modelo estándar modificado para los errores, ecuación (2.7), obtenemos

$$\widehat{z}_i = fl(x_i\, y_i) = \frac{x_i\, y_i}{1+\delta_i}, \qquad |\delta_i| \le u,$$

de donde

$$\widehat{z}_i = x_i\, y_i - \delta_i\, \widehat{z}_i, \qquad (1+\delta_{s,i})\, \widehat{s}_i = \widehat{s}_{i-1} + \widehat{z}_i, \qquad |\delta_{s,i}| \le u,$$

que se puede escribir como

$$s_i + e_i + \delta_{s,i}\, \widehat{s}_i = s_{i-1} + e_{i-1} + x_i\, y_i - \delta_i\, \widehat{z}_i,$$

que nos permite calcular el error cometido

$$e_i = e_{i-1} - \delta_{s,i}\, \widehat{s}_i - \delta_i\, \widehat{z}_i,$$

que podemos acotar fácilmente

$$|e_i| \le |e_{i-1}| + u\, |\widehat{s}_i| + u|\widehat{z}_i|,$$

Algoritmo 2.3 *Producto interior con estimación del error progresivo.*

```
function s = productoInteriorErrorProgresivo (x, y)
% x, y, son vectores de números flotantes
% s es su producto interior más su error progresivo
%
  n=length(x)
  s = 0;   mu=0
  for i=1:n
    z = x(i)*y(i)
    s = s + z
    mu = mu + |s| + |z|
  end
  s = s + mu
end
```

luego, como $e_0 = 0$, tenemos $|e_n| \leq u\,\mu_n$, donde

$$\mu_i = \mu_{i-1} + |\widehat{s}_i| + |\widehat{z}_i|, \qquad \mu_0 = 0.$$

De esta forma podemos obtener una estimación a posteriori del error progresivo $|fl(x^\top y) - x^\top y| \leq u\,\mu$, en la que se basa el algoritmo 2.3, que da como resultado $\widehat{s} = s_n + e_n$ con error progresivo acotado por $|e_n| \leq u\,\mu_n$.

Una cota de error calculada a posteriori normalmente es mucho más pequeña que la cota obtenida a priori, y nos permite estudiar el comportamiento de un algoritmo desde una perspectiva "práctica". De hecho, la idea general en este análisis es utilizar la siguiente forma modificada del modelo estándar (2.6),

$$|x \text{ op } y - fl(x \text{ op } y)| \leq u\,|fl(x \text{ op } y)|.$$

Este tipo de análisis es muy fácil de incluir en un algoritmo numérico y nos permite estudiar empíricamente el comportamiento de sus errores y su estabilidad numérica. Wilkinson, el padre del análisis de errores, utilizó mucho esta práctica en los primeros años de la computación.

2.4.4 Unos lemas útiles en análisis de errores

Un lema muy utilizado en análisis de errores, que será referenciado varias veces en el resto de este curso, y que escribimos siguiendo a [9], es el siguiente.

Lema 2.8 *Sean* $|\delta_i| \le u$ *y* $\rho_i = \pm 1$, *para todo* $i = 1, 2, \ldots, n$. *Si* $n\,u < 1$, *entonces*

$$\prod_{i=1}^{n} (1+\delta_i)^{\rho_i} = 1 + \theta_n,$$

donde

$$|\theta_n| \le \frac{n\,u}{1 - n\,u} \equiv \gamma_n.$$

Además, si $n\,u < 0.01$, *se cumple que*

$$|\theta_n| \le 1.01\, n\,u = n\,\tilde{u}.$$

Demostración. Vamos a demostrar este teorema por inducción en n. El caso base $n = 1$ es obvio. Supongamos como hipótesis de inducción que se cumple para $n-1$ y estudiemos el caso n-ésimo. Tenemos dos posibilidades claramente diferenciadas, que $\rho_n = +1$ o que $\rho_n = -1$. En el primer caso,

$$|\theta_n| \le \left| (1+\delta_n) \prod_{i=1}^{n-1} (1+\delta_i)^{\rho_i} - 1 \right| = \left| \theta_{n-1} + \delta_n \,(1 + \theta_{n-1}) \right|,$$

aplicando la hipótesis de inducción obtenemos

$$|\theta_n| \le \frac{(n-1)\,u}{1-(n-1)\,u} + u \left(1 + \frac{(n-1)\,u}{1-(n-1)\,u} \right) = \frac{(n-1)\,u}{1-(n-1)\,u} + \frac{u}{1-(n-1)\,u} = \frac{n\,u}{1-(n-1)\,u},$$

de donde obtenemos el resultado deseado

$$|\theta_n| \le \frac{n\,u}{1-(n-1)\,u} \le \frac{n\,u}{1-n\,u} \equiv \gamma_n.$$

Para el segundo caso,

$$|\theta_n| \le \left| \frac{1}{1+\delta_n} \prod_{i=1}^{n-1} (1+\delta_i)^{\rho_i} - 1 \right| = \left| \frac{1+\theta_{n-1}}{1+\delta_n} - 1 \right|.$$

Las siguientes relaciones

$$1 - u \le 1 + \delta_n \le 1 + u, \qquad \frac{1}{1+u} \le \frac{1}{1+\delta_n} \le \frac{1}{1-u},$$

$$1 - \gamma_{n-1} \le 1 + \theta_{n-1} \le 1 + \gamma_{n-1}, \qquad \frac{1-\gamma_{n-1}}{1+u} \le \frac{1+\theta_{n-1}}{1+\delta_n} \le \frac{1+\gamma_{n-1}}{1-u},$$

implican que

$$|\theta_n| \le \left| \frac{1+\theta_{n-1}}{1+\delta_n} - 1 \right| \le \left| \frac{1+\gamma_{n-1}}{1-u} - 1 \right| = \left| \frac{1}{1-(n-1)\,u} \, \frac{1}{1-u} - 1 \right|.$$

Ahora bien, como se cumple

$$\frac{1}{(1-(n-1)\,u)\,(1-u)} \leq \frac{1}{1-n\,u}, \qquad 1-n\,u \leq 1-n\,u+(n-1)\,n\,u^2,$$

finalmente obtenemos el resultado deseado (dado que $n\,u<1$)

$$|\theta_n| \leq \left|\frac{1}{1-n\,u}-1\right| \leq \frac{n\,u}{1-n\,u} = \gamma_n.$$

La segunda parte del teorema, el caso $n\,u<0.01$, se lo dejamos como ejercicio al lector.

Lema 2.9 *Sean k y j números naturales, y $\{\theta_k\}$ un conjunto de números reales acotados por $|\theta_k| \leq \gamma_k = k\,u/(1-k\,u)$. Se cumplen las siguientes relaciones [9]*

$$(1+\theta_k)\,(1+\theta_j) = 1+\theta_{k+j},$$

$$\frac{1+\theta_k}{1+\theta_j} = \begin{cases} 1+\theta_{k+j}, & j \leq k, \\ 1+\theta_{k+2\,j}, & j > k, \end{cases}$$

y además la aritmética de γ_k cumple

$$\begin{aligned} \gamma_k + u &\leq \gamma_{k+1}, \\ j\,\gamma_k &\leq \gamma_{j\,k}, \\ \gamma_k\,\gamma_j &\leq \gamma_{\min\{k,j\}}, \\ \gamma_k+\gamma_j+\gamma_k\,\gamma_j &\leq \gamma_{k+j}. \end{aligned}$$

La demostración de este lema se le ofrece como ejercicio al lector, quien notará que a pesar de la regla segunda del lema anterior, se cumple siempre que

$$\frac{\prod_{i=1}^{k}(1+\delta_i)^{\pm 1}}{\prod_{i=1}^{j}(1+\delta_i)^{\pm 1}} = 1+\theta_{k+j},$$

pero, que bajo las hipótesis del lema, sólo se puede demostrar el resultado que aparece en el enunciado.

2.4.5 Análisis de errores para el producto matricial

Conocido el análisis de errores del producto interior es fácil estudiar los productos matriz-vector y matriz-matriz. Sea $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, e $y = A\,x$. El vector y está formado por m productos interiores $y_i = a_i^\top\,x$, $i = 1:m$, donde $a_i^\top$ es la i-ésima fila de A. Aplicando (2.9) obtenemos

$$\widehat{y}_i = (a_i+\Delta a_i)^\top\,x, \qquad |\Delta a_i| \leq \gamma_n\,|a_i|,$$

y por tanto, la acotación del error regresivo o hacia atrás

$$\widehat{y} = (A + \Delta A)\, x, \qquad |\Delta A| \leq \gamma_n\, |A|, \qquad A \in \mathbb{R}^{m\times n},$$

que implica la cota de error progresivo o hacia adelante

$$|y - \widehat{y}| \leq \gamma_n\, |A|\, |x|.$$

En el análisis de errores es más fácil obtener acotaciones componente a componente que mediante normas[8].

Consideremos ahora el producto de dos matrices, $C = A\,B$, donde $A \in \mathbb{R}^{m\times n}$ y $B \in \mathbb{R}^{n\times p}$. Podemos calcular cada columna de C utilizando $c_j = A\, b_j$, donde b_j es la j-ésima columna de B. De esta forma,

$$\widehat{c}_j = (A + \Delta A)\, b_j, \qquad |\Delta A| \leq \gamma_n\, |A|.$$

Como vemos la j-ésima columna de C tiene un error regresivo pequeño, es la columna de una matriz obtenida perturbando ligeramente la matriz A. Sin embargo, no se puede decir lo mismo de $\widehat{C}$ como matriz completa[9].

Se puede obtener una cota de error progresivo también "pequeña" de la forma

$$|C - \widehat{C}| \leq \gamma_n\, |A|\, |B|.$$

El lector notará que este resultado no está lejos del resultado ideal $|C - \widehat{C}| \leq \gamma_n\, |C|$, que indica que cada una de las componentes de C se calcula con una exactitud en error relativo alta.

2.4.6 Análisis de errores para la suma

Una de las operaciones más importantes y utilizadas en computación numérica es la suma de n números,

$$s = \sum_{i=1}^{n} x_i,$$

que podemos calcular mediante el algoritmo 2.4, llamado suma recursiva,

$$s_0 = 0, \qquad s_i = s_{i-1} + x_i, \qquad i = 1, 2, \ldots, n.$$

[8] El concepto de norma de un vector y de una matriz será introducido en el próximo tema.

[9] Se puede demostrar que $\widehat{C} = (A + \Delta A)\, B$ donde $|\Delta A| \leq \gamma_n\, |A|\, |B|\, |B^{-1}|$, que no es una cota pequeña si $|B|\, |B^{-1}|$ es muy grande. El mismo resultado se puede escribir simétricamente $\widehat{C} = A\, (B + \Delta B)$. Omitimos la demostración de estos resultados [9]

Algoritmo 2.4 *Algoritmo recursiva para la suma de números.*

```
function s = sumaRecursiva (x)
% x, es un vector de números flotantes
% s es su suma
%
  n=length(x)
  s = 0;
  for i=1:n
    s = s + x(i)
  end
end
```

La exactitud de la suma (y su error) depende del orden de los sumandos. De hecho existen otros algoritmos que ordenan los sumandos antes de sumarlos. Por ejemplo, el resultado de la suma es diferente si se ordenan los sumandos de forma creciente, $|x_1| \leq |x_2| \leq \cdots \leq |x_n|$, o decreciente, $|x_1| \geq |x_2| \geq \cdots \geq |x_n|$.

Usando el modelo de aritmética estándar (2.7), podemos realizar un análisis de errores para la suma. La i-ésima suma acumulada tiene el error

$$\widehat{s}_i = \frac{\widehat{s}_{i-1} + x_i}{1 + \delta_i}, \qquad |\delta_i| \leq u, \quad i = 1, 2, \ldots, n-1,$$

con lo que el error final es

$$E_n = s_n - \widehat{s}_n = \sum_{i=1}^{n-1} \delta_i \, \widehat{s}_i,$$

y la cota más pequeña de este error es

$$|E_n| \leq u \sum_{i=1}^{n-1} \widehat{s}_i.$$

Es fácil comprobar que

$$|\widehat{s}_i| \leq \sum_{j=1}^{i} |x_j| + O(u), \qquad i = 1, 2, \ldots, n-1,$$

con lo que obtenemos como cota de error progresivo

$$|E_n| \leq (n-1)\, u \sum_{i=1}^{n} |x_j| + O\big(u^2\big).$$

A partir de aquí podemos obtener fácilmente una cota de error regresivo, sumando de forma exacta los números $x_i\,(1+\theta_{i-1})$ donde $|\theta_{i-1}| \le \gamma_{i-1}$.

El resultado que hemos obtenido es fácil de interpretar. Debemos sumar de manera que los sumandos intermedios $|s_i|$ sean lo más pequeños posibles. Aunque aparentemente esto es fácil de conseguir, no siempre es así. Para números no negativos, $x_i \ge 0$ el mejor orden es el orden creciente en el sentido de que conduce a la menor cota de error progresivo posible. Sin embargo, cuando los números son positivos y negativos, el mejor orden es difícil de obtener. De hecho, el orden creciente puede llegar a ser el peor. Consideremos los números $\{1, M, 2\,M, -3\,M\}$ donde $fl(1+M) = M$, es decir, $M > 1/u$, con u la unidad de redondeo. Sumando en orden creciente obtenemos

$$fl(1 + M + 2\,M - 3\,M) = 0,$$

con un error relativo máximo, y en orden decreciente

$$fl(-3\,M + 2\,M + M + 1) = 1,$$

sin error. La razón son las diferencias cancelativas catastróficas que se producen en los sumandos intermedios. El orden decreciente es el mejor cuando se produce un gran números de diferencias cancelativas en la suma, es decir, cuando

$$\left|\sum_{i=1}^{n} x_i\right| \ll \sum_{i=1}^{n} |x_i|.$$

Como vemos, elegir el orden más adecuado para sumar números es una decisión que no es fácil, salvo en los casos más simples.

Existe un algoritmo descubierto por Gill en 1951, la suma compensada, que estima el error cometido durante la suma, y lo añade al final de ésta, permitiendo reducir fuertemente el error final. Presentaremos la versión desarrollada por Kahan en 1965. La idea es sencilla, y la vamos a ilustrar con un ejemplo práctico. Sean $x = 0.235 = 0.235 \cdot 10^0$, e $y = 0.00134 = 0.134 \cdot 10^{-2}$; su suma flotante toma la forma,

$$fl(x + y) = fl(0.235 \cdot 10^0 + 0.00134 \cdot 10^0) = fl(0.23634 \cdot 10^0) = 0.236 \cdot 10^0,$$

con un error de $e = 0.34 \times 10^{-2}$. Si restamos a la suma, sucesivamente, el primer argumento y luego el segundo

$$fl(fl(x + y) - x) = fl(0.236 \cdot 10^0 + 0.235 \cdot 10^0) = fl(0.001 \cdot 10^0) = 0.100 \cdot 10^{-2},$$

$$fl(fl(fl(x + y) - x) - y) = fl(0.100 \cdot 10^{-2} - 0.134 \cdot 10^{-2}) = fl(-0.034 \cdot 10^{-2}) = -0.340 \cdot 10^{-2},$$

Algoritmo 2.5 *Algoritmo de suma compensada para la suma de números.*

```
function s = sumaRecursiva (x)
% x, es un vector de números flotantes
% s es su suma
%
s=0
e=0
for i=1:n
   temp=s
   y=x(i)+e
   s=temp+y
   e=(temp-s)+y    % debe evaluarse en este orden
end
s=s+e
```

obtenemos exactamente $-e$. De hecho, podemos obtener e exactamente utilizando la expresión

$$e = fl(fl(a - fl(a + b)) + b) = ((a \ominus (a \oplus b)) \oplus b),$$

que en nuestro ejemplo nos da

$$fl(x - fl(x + y)) = fl(0.235 \cdot 10^0 - 0.236 \cdot 10^0 +) = -fl(0.001 \cdot 10^0) = -0.100 \cdot 10^{-2},$$

$$fl(fl(x - fl(x + y)) + b) = fl(-0.100 \cdot 10^{-2} + 0.134 \cdot 10^{-2}) = fl(0.034 \cdot 10^{-2}) = 0.340 \cdot 10^{-2}.$$

Aunque omitimos los detalles de la demostración se puede demostrar que el error calculado $\widehat{e}$ es una buena estimación al error verdadero $(a + b) - fl(a + b)$. Más aún, en una aritmética en base 2 con redondeo se tiene que $a + b = \widehat{s} + \widehat{e}$ de forma exacta. La demostración para la aritmética IEEE se encuentra en el libro de Knuth [10].

El algoritmo de suma compensada de Kahan emplea la corrección e en cada paso de la suma recursiva, lo calcula y corrige la suma acumulada antes de añadir el siguiente término. Este algoritmo aparece descrito en detalle en algoritmo 2.5.

El error estimado $\widehat{e}$ obtenido en el algoritmo de suma compensada no coincide con el error real e; sin embargo, se puede demostrar un análisis de errores conduce a una cota de error regresivo mucho más pequeña [10]

$$\widehat{s}_n = \sum_{i=1}^{n} (1 + \eta_i)\, x_i, \qquad |\eta_i| \le 2\,u + O\big(n\,u^2\big)\,,$$

que es casi la cota de error regresivo ideal. Si además acabamos la suma con `s=s+e`, entonces se puede probar que

$$|\eta_i| \leq 2\,u + O\big((n-i+1)\,u^2\big)\,.$$

La cota de error progresiva correspondiente a las cotas regresivas anteriores es

$$|E_n| \leq (2\,u + O\big(n\,u^2\big))\,\sum_{i=1}^{n} |x_i|,$$

que indica que para $n\,u \leq 1$, la constante obtenida es independiente de n, lo que supone una mejora muy significativa respecto a las anteriores cotas que dependen linealmente de n. Sin embargo, el lector debe notar que la suma compensada no garantiza que el error relativo del resultado sea pequeño cuando

$$\sum_{i=1}^{n} |x_i| \gg \left|\sum_{i=1}^{n} x_i\right|.$$

2.5 Relación entre condicionamiento y estabilidad

El lector, probablemente, ha estudiado en cursos anteriores la estabilidad de soluciones de ecuaciones diferenciales ordinarias y de ecuaciones en diferencias finitas, y su aplicación, por ejemplo, al diseño de sistemas de control en tiempo continuo y en tiempo discreto. En esos contextos, se entiende por estabilidad de un problema en el sentido de Hadamard, cuando pequeños cambios en los datos del problema, sean los coeficientes de la ecuación, o sus condiciones iniciales o de contorno, conduce a pequeños cambios en su solución.

La estabilidad del problema está muy relacionada con su estabilidad numérica, y ésta con su condicionamiento numérico. La estabilidad numérica describe la sensibilidad del problema a errores en sus datos, que en el caso del problema de evaluación de funciones son los argumentos de la función, y los únicos errores que se cometen son errores de redondeo. Por ello, estabilidad y condicionamiento numéricos son conceptos equivalentes en un sentido amplio. Un problema bien condicionado es estable, y uno inestable está mal condicionado. En muchos casos se habla de problemas bien (mal) condicionados en el sentido de Hadamard.

Ejemplo 2.10 *El concepto de estabilidad de Hadamard para ecuaciones diferenciales ordinarias.*

Recordaremos el concepto de estabilidad para ecuaciones diferenciales ordinarias considerando el problema de valores iniciales para la ecuación diferencial lineal

$$\frac{d^2y}{dx^2} - y = 1, \qquad y(0) = 0, \qquad y'(0) = -1,$$

cuya solución general es fácil de obtener

$$y(x) = -1 + A\,e^{x} + B\,e^{-x},$$

y aplicando las condiciones iniciales

$$y(0) = -1 + A + B = 0, \qquad y'(0) = A - B = -1,$$

permite obtener

$$-1 + 2\,A = y(0) + y'(0) = -1 \quad\Rightarrow\quad A = 0,$$

$$-1 + 2\,B = y(0) - y'(0) = 1 \quad\Rightarrow\quad B = 1,$$

lo que indica, finalmente, que la solución es

$$y(x) = -1 + e^{-x}.$$

Consideremos ahora el mismo problema pero añadiendo pequeños errores en las condiciones iniciales,

$$\frac{d^2y}{dx^2} - y = 1, \qquad y(0) = \epsilon_1, \qquad y'(0) = -1 + \epsilon_2.$$

En este caso, la aplicación de las condiciones iniciales a la solución permite obtener

$$-1 + 2\,A = \epsilon_1 - 1 + \epsilon_2 \quad\Rightarrow\quad A = \frac{\epsilon_1 + \epsilon_2}{2},$$

$$-1 + 2\,B = \epsilon_1 + 1 - \epsilon_2 \quad\Rightarrow\quad B = 1 + \frac{\epsilon_1 - \epsilon_2}{2},$$

con lo que la nueva solución es

$$y(x) = -1 + \underbrace{\frac{\epsilon_1 + \epsilon_2}{2} e^{x}} + \left(1 + \frac{\epsilon_1 - \epsilon_2}{2}\right) e^{-x}.$$

En este caso se puede observar que el término subrayado es exponencialmente creciente al menos que $\epsilon_1 + \epsilon_2 = 0$. Si esta condición no se cumple, el problema será inestable y se dice que el problema no está bien condicionado en el sentido de Hadamard.

La inestabilidad de un problema no se puede evitar siempre. Hay problemas físicamente inestables, que implicarán modelos matemáticos inestables y, a su vez, métodos numéricos inestables. En estos últimos no es posible evitar su inestabilidad intrínseca. Sin embargo, estos métodos no son necesariamente inútiles, aunque requieren un tratamiento especial, que en este curso no podemos estudiar en detalle.

Ejemplo 2.11 *Una explosión atómica como ejemplo de problema físico inestable intrínsecamente.*

La ley fundamental de la desintegración radioactiva indica que el número de núcleos que se desintegran es proporcional al número existente y al intervalo de tiempo,

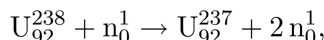
$$dN = -\lambda \, N \, dt,$$

donde $\lambda > 0$ es la constante de desintegración que depende de la vida media del núcleo en cuestión. Integrando esta expresión diferencial obtenemos como solución

$$N = N_0 \, e^{-\lambda \, t},$$

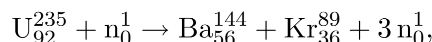
que nos muestra que la desintegración radiactiva es un proceso físicamente estable.

Sin embargo, hay isótopos que sufren una reacción de fisión en cadena mediada por neutrones (n_0^1), proceso que es físicamente inestable, predicho originalmente por Zeldovich y Yariton en 1939. Por ejemplo, para el isótopo U_{92}^{238}, el más abundante del uranio en la naturaleza, el proceso de fisión puede tomar la forma



$$\mathrm{U}_{92}^{238} + \mathrm{n}_0^1 \rightarrow \mathrm{U}_{92}^{237} + 2 \, \mathrm{n}_0^1,$$

donde el U_{92}^{237}, también inestable, se desintegrará a su vez. En las bombas y en los reactores nucleares se prefiere el isótopo U_{92}^{235}, sólo presente en el 0.95% del uranio natural, que puede seguir reacciones en las que se generan hasta 3 neutrones, como en



$$\mathrm{U}_{92}^{235} + \mathrm{n}_0^1 \rightarrow \mathrm{Ba}_{56}^{144} + \mathrm{Kr}_{36}^{89} + 3 \, \mathrm{n}_0^1,$$

donde los productos resultado también son radiactivos. De esta forma, en una reacción en cadena, un neutrón produce 3, que a su vez producen otros 3, y así sucesivamente.

El proceso de reacción en cadena se modela como la desintegración radiactiva natural pero introduciendo el coeficiente de multiplicación $k = -\lambda > 0$, lo que conduce a la ecuación diferencial

$$\frac{dN}{dt} = k \, N, \qquad N(0) = N_0, \tag{2.10}$$

para el número de núcleos que se desintegra, y cuya solución es

$$N = N_0 \, e^{k \, t}.$$

La energía liberada es proporcional a este número y, por tanto, también crece exponencialmente.

Hemos de indicar que este proceso físico inestable no puede durar indefinidamente, ya que el número de núcleos inicial N_0 se consume rápidamente, por lo que nuestro modelo matemático

(inestable) sólo describe el comportamiento inicial de la reacción en cadena, que eventualmente acabará parándose.

Aproximando numéricamente la ecuación (2.10) mediante el método de Euler hacia adelante (véase el tema 1) se tiene, conocido N_0,

$$\frac{N_{n+1} - N_n}{\Delta t} = a\, N_n, \tag{2.11}$$

donde Δt es el paso de tiempo. Esta ecuación se puede escribir como

$$N_{n+1} = N_n\,(1 + a\,\Delta t),$$

luego iterando su solución es

$$N_n = N_0\,(1 + a\,\Delta t)^n.$$

Para que esta solución sea estable es necesario que

$$|1 + a\,\Delta t| < 1,$$

es decir,

$$-1 < 1 + a\,\Delta t < 1 \quad \Rightarrow \quad a\,\Delta t < 0 \quad \Rightarrow \quad a < 0,$$

la misma condición que para el problema físico. Pero además, si $a = -b$, para tener $b > 0$, también

$$-1 < 1 - b\,\Delta t \quad \Rightarrow \quad b\,\Delta t < 2 \quad \Rightarrow \quad \Delta t < \frac{2}{b},$$

el paso de tiempo sufre una restricción adicional de origen estrictamente numérico. Es decir, que incluso cuando el problema matemático es estable (desintegración radiactiva), el método numérico puede ser inestable si no se elige adecuadamente el paso de tiempo.

Cuando el problema matemático es inestable (reacción en cadena), el problema numérico también lo es. Sin embargo, esto no impide que la solución numérica se aproxime a la solución analítica cuando el paso de tiempo se reduce a cero, es decir, cuando para un tiempo $t \equiv t_n = n\,k$ fijado se introduce un número de pasos de tiempo n que tiende a infinito, en cuyo caso

$$\lim_{n\to\infty} N_n = N_0 \lim_{n\to\infty} (1 + a\,k)^n = N_0 \lim_{n\to\infty} \left(1 + \frac{a\,t}{n}\right)^n$$

$$= N_0 \lim_{n\to\infty} \left(1 + \frac{1}{\left(\frac{n}{a\,t}\right)}\right)^{\frac{n}{a\,t}\,a\,t} = N_0\, e^{a\,t}.$$

Ejemplo 2.12 *Calcularemos las siguientes integrales definidas [13]*

$$E_n = \int_0^1 x^n\, e^{x-1}\, dx, \qquad n = 1, 2, \ldots.$$

Este problema está matemáticamente bien condicionado, ya que pequeños errores de redondeo en el integrando afectan muy poco al resultado, como el lector puede comprobar fácilmente. Para obtener un algoritmo numérico iterativo para evaluar estas integrales podemos usar integración por partes

$$\int_0^1 x^n\, e^{x-1}\, dx = x^n\, e^{x-1}\Big|_0^1 - \int_0^1 n\, x^{n-1}\, e^{x-1}\, dx$$

lo que conduce a la siguiente ecuación en diferencias

$$E_n = 1 - n\, E_{n-1}, \qquad n = 2, 3, \ldots,$$

donde fácilmente se calcula $E_1 = 1/e$. Si se usa aritmética flotante con 6 dígitos decimales, se obtienen los siguientes resultados tras 9 iteraciones

$$E_1 \approx 0.367879, \qquad E_2 \approx 0.264242,$$

$$E_3 \approx 0.207274, \qquad E_4 \approx 0.170904,$$

$$E_5 \approx 0.145480, \qquad E_6 \approx 0.127120,$$

$$E_7 \approx 0.110160, \qquad E_8 \approx 0.118720,$$

$$E_9 \approx -0.0684800.$$

Se observa que E_9 es negativo, lo que es un resultado sorprendente, ya que el integrando $x^n\, e^{x-1}$ es positivo en todo el intervalo $(0, 1)$ y por tanto debe ser siempre $E_n > 0$.

La causa de este error tan grande ha sido la propagación del error cometido en E_1 al aproximar $1/e$ con 6 dígitos decimales. Este error se ha propagado catastróficamente, ya que ha sido multiplicado por $(-2)(-3)\cdots(-9) = 9! = 362880$, por lo que el error en E_1, aproximadamente 4.4×10^{-7} se ha convertido en un error de $3.6 \times 10^6 \times 4.4 \times 10^{-7} \approx 0.16$. Esta estimación del error cometido es tan buena que nos permite corregir el valor obtenido para E_9 y obtener $0.16 - 0.06848 = 0.092$, que es el valor correcto con dos dígitos decimales.

Para evitar la inestabilidad numérica de este algoritmo es necesario encontrar otro algoritmo que sea estable. Por ejemplo, la iteración hacia atrás

$$E_{n-1} = \frac{1 - E_n}{n}, \qquad n = \ldots, 3, 2,$$

tiene la ventaja que el error de redondeo original decrece conforme se itera. Para obtener un valor inicial para esta iteración se puede usar,

$$E_n = \int_0^1 x^n\, e^{x-1}\, dx \le \int_0^1 x^n\, dx = \frac{1}{n+1}. \tag{2.12}$$

Con lo que se observa que E_n tiende a cero cuando $n \to \infty$. Si se considera $E_{20} \approx 0$ y se calcula E_9 con el algoritmo estable

$$\begin{aligned} E_{20} &\approx 0.0000000, \qquad & E_{19} &\approx 0.0500000, \\ E_{18} &\approx 0.0500000, \qquad & E_{17} &\approx 0.0527778, \\ E_{16} &\approx 0.0557190, \qquad & E_{15} &\approx 0.0590176, \\ E_{14} &\approx 0.0627322, \qquad & E_{13} &\approx 0.0669477, \\ E_{12} &\approx 0.0717733, \qquad & E_{11} &\approx 0.0773523, \\ E_{10} &\approx 0.0838771, \qquad & E_{9} &\approx 0.0916123. \end{aligned}$$

A partir de E_{15}, el error inicial en E_{20} se ha disipado gracias a la estabilidad del algoritmo y, por tanto, los valores obtenidos para $E_{15}, \ldots, E_9$ son exactos hasta los 6 dígitos de precisión con los que han sido calculados, salvo errores de redondeo en el último dígito. De hecho, tomando la cota (2.12), por ejemplo, para E_{14}, podemos calcular fácilmente E_9 mediante el algoritmo estable

$$\begin{aligned} E_{14} &\approx 0.0666667, \qquad & E_{13} &\approx 0.0666667, \\ E_{12} &\approx 0.0717949, \qquad & E_{11} &\approx 0.0773504, \\ E_{10} &\approx 0.0838772, \qquad & E_{9} &\approx 0.0916123, \end{aligned}$$

obteniendo de nuevo el resultado correcto con 6 dígitos.

Se anima al lector a realizar como ejercicio un análisis de errores progresivos y regresivos de los dos algoritmos presentados en este ejemplo, con objeto de confirmar los resultados que hemos comentado.

En este ejemplo hemos visto como un problema matemáticamente bien condicionado (estable) requiere un algoritmo numérico también estable, ya que los resultados de un algoritmo inestable no son fiables en manera alguna.

2.6 Pseudocódigo utilizado en los algoritmos

En análisis numérico es necesario presentar algoritmos que expresen de forma clara y sencilla la aplicación de los métodos numéricos a la hora de resolver un problema. En lugar de elegir un

lenguaje de alto nivel, como Fortran, C o C++, o un lenguaje de propósito específico, como Matlab o Mathematica, en este curso utilizaremos un lenguaje en forma de pseudocódigo. Suponemos que el lector ha estudiado previamente algún curso de programación en algún lenguaje de alto nivel, incluyendo la especificación de algoritmos en algún tipo de pseudocódigo.

El pseudocódigo que utilizaremos en este curso es similar al utilizado en muchos cursos de análisis numérico [12, 9, 11], y se caracteriza básicamente por la notación de índices que Moler incorporó al lenguaje Matlab y que también usan otros lenguajes, como Fortran90. Esta notación es actualmente estándar para la descripción de técnicas numéricas.

Las variables en nuestros algoritmos serán utilizadas sin declaración previa y sin indicación explícita de su tipo. Para los vectores y matrices utilizaremos la notación `v(i)`, y `A(i,j)`, respectivamente. Los vectores, sin indicación previa de lo contrario, se suponen vectores columna (matrices de $n \times 1$). El tamaño de los vectores y matrices vendrá determinado por el contexto, entre los parámetros de las funciones o calculada mediante la función `length` que devolverá la dimensión de un vector o de una matriz cuadrada. Por ejemplo, `length(A)` devolverá la dimensión de la matriz cuadrada `A`.

Para facilitar el acceso a componentes de un vector o de una matriz, incluyendo el acceso a sus filas y columnas, utilizaremos la notación de "`:`" para rangos de Matlab. Un <rango> se especifica como `i:j`, que representa la sucesión de números `i, i+1, i+2, ..., j`, salvo que `i>j`, en cuyo caso representa el rango vacío (vector vacío). Excepcionalmente utilizaremos la notación `i:d:j`, que incluye un incremento o salto, de forma que representa la sucesión `i, i+d, i+2*d, ..., i+k*d`, donde `i+k*d<=j` pero `i+(k+1)*d>j`. Permitiremos que `d` sea negativo si `i>=j`.

Sea `v` un vector de n componentes, podemos acceder a sus primeras 3 componentes con `v(1:3)`, a sus componentes impares con `v(1:2:n)`, y a su última componente con `v(n)`.

Sea `A` una matriz de $n \times m$ componentes, podemos acceder a un elemento mediante `A(i,j)`, a toda la fila i-ésima con `A(i,:)` (que es un vector fila, matriz de $1 \times n$), a toda la columna j-ésima con `A(:,j)` (que es un vector columna, matriz de $n \times 1$), y con `A(1:3,2:5)` a la submatriz formada por las componentes de las columnas 2 a 5 y de las filas 1 a 3 de la matriz.

Nuestro pseudocódigo tiene las sentencias de control más usuales. La sentencia de condición `if`, la sentencia de iteración `for` y la sentencia de iteración condicional `while`. Para indicar comentarios se utilizará `%` que indicará el comienzo de un comentario.

La sintaxis de la sentencia `if` es la siguiente:

```
if (<condicion>)
```

```
  <bloque de sentencias>
else
  <ultimo bloque de sentencias>
end
```

donde el bloque `else` es opcional.

Observe los dos ejemplos a continuación

```
if (s=0) s=s+1 else s=s-1 end
a=32
if (a<45)
  a=a-3
end
```

La sentencia `for` tiene la siguiente sintaxis

```
for i=<rango>
  <bloque de sentencias>
end
```

donde para `<rango>` utilizaremos la notación de dos puntos de Matlab. Cuando éste corresponda a un rango (vector) vacío no se ejecutará el bloque de sentencias.

Veamos un par de ejemplos

```
n=3
for i=n:-1:2
  x(1)=x(1)-x(i)   % resta a x(1) la suma de x(2:n)
end
for i=1:n
  for j=1:i
    B(i,j)=A(j,i)/x(i)
  end
end
```

Finalmente, la sentencia `while` tiene como sintaxis

```
while (<condicion>)
  <bloque de sentencias>
end
```

y ejecuta el bloque de sentencias mientras se cumpla la `<condicion>`.

Para describir procedimientos o funciones utilizaremos una notación similar a la de Matlab [12], cuya sintaxis es la siguiente

```
function [<variable>] = <nombre> (<argumentos>)
  <bloque de sentencias>
end
```

donde `<nombre>` es el nombre de la función, `<argumentos>` es una sucesión de variables separadas por comas, y `[<variable>]` es la serie de variables de salida que retornará los valores de la función y que van entre corchetes. Cuando sólo se devuelve una única variable, los corchetes se omitirán. En el bloque de sentencias se deberá asignar un valor a las variables de salida.

Veamos una función que devuelve el producto escalar o interno de dos vectores

```
function c = productoInterior (x, y)
  c=0
  n=length(x)
  for i=1:n
    c=c+x(i)*y(i)
  end
end
```

Una función que calcula el producto de una matriz por un vector y devuelve el vector de salida y su módulo, tomará la forma

```
function [y, ymod] = productoInterior (A, x)
  n=length(A)
  y(1:n)=0
  ymod=0
  for i=1:n
    for j=1:n
      y(i)=y(i)+A(i,j)*x(j)
```

```
      ymod = ymod + y(i)^2
    end
  end
  ymod = sqrt(ymod)
end
```

En los algoritmos a veces utilizaremos el producto de matrices a través de la notación ":". También utilizaremos algoritmos previamente definidos en el curso. Por ejemplo, el algoritmo anterior se puede reescribir como

```
function [y, ymod] = prodMatrizVector (A, x)
  n=length(A)
  y(1:n)=0
  for i=1:n
    y(i)=y(i)+A(i,:)*x(:)
  end
  ymod = sqrt( productoInterior (y,y) )
end
```

aunque intentaremos no abusar de estas características.

BIBLIOGRAFÍA

[1] "*IEEE standards 754 and 854 for Floating-Point Arithmetic*," Institute of Electrical and Electronics Engineers (1985).

[2] W. Kahan, "*Lecture Notes on the Status of the IEEE Standard 754 for Binary Floating-Point Arithmetic*," University of California, Berkeley (1996). Downloadable at `http://cch.loria.fr/documentation/IEEE754/wkahan/ieee754.ps`, or at `http://HTTP.CS.Berkeley.EDU/~wkahan/ieee754status/ieee754.ps`.

[3] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, pp. 153–230 (1991). Reproduced at `http://citeseer.nj.nec.com/goldberg91what.html`.

[4] M.L. Overton, "*Numerical Computing with IEEE Floating Point Arithmetic*," SIAM, Philadelphia (2001). Michael L. Overton, traducido por Alejandro Casares M., "*Cómputo Numérico con Aritmética de punto flotante IEEE*," Sociedad Matemática Mexicana (2002).

[5] M.L. Overton, "*Floating Point Representation*," Courant Institute of Mathematical Sciences, New York University (1996). Reproduced at `http://www.cs.nyu.edu/cs/faculty/overton/g22_nm/ieee.ps`, or at `www.cs.nyu.edu/courses/spring01/ V22.0421-001/extract.pdf`.

[6] Douglas N. Arnold, "*The Patriot Missile Failure*," Institute for Mathematics and its Applications, Minneapolis (2000). Reproduced at `http://www.ima.umn.edu/~arnold/disasters/patriot.html`.

[7] Douglas N. Arnold, "*The Explosion of the Ariane 5 Rocket*," Institute for Mathematics and its Applications, Minneapolis (2000). Reproduced at
`http://www.ima.umn.edu/~arnold/disasters/ariane.html`.

[8] S.-Sum Chow, "*IEEE Floating Point Standard*," Brigham Young University, Provo, Utah (2001). Reproduced at
`http://www.math.byu.edu/~schow/work/IEEEFloatingPoint.htm`

[9] Nicholas J. Higham, "*Accuracy and Stability of Numerical Algorithms*," SIAM, Philadelphia (1996).

[10] Donald E. Knuth, "*The Art of Computer Programming. Volume 2, Seminumerical Algorithms*," Second Edition, Addison-Wesley, Reading, MA, USA (1981).

[11] G.W. Stewart, "*Matrix Algorithms. Volume 1, Basic Decompositions*," SIAM, Philadelphia (1998).

[12] Gene H. Golub and Charles F. Van Loan, "*Matrix Computations*," Second Edition, The Johns Hopkins University Press, Baltimore, Maryland, USA (1989).

[13] George E. Forsythe, Michael A. Malcolm and Cleve B. Moler, "*Computer Methods for Mathematical Computations*," Prentice-Hall, Englewood Cliffs, New Jersey, USA (1977).

[14] James H. Wilkinson, "*Rounding Errors in Algebraic Processes*," Prentice-Hall, Englewood Cliffs, New Jersey, USA (1963).