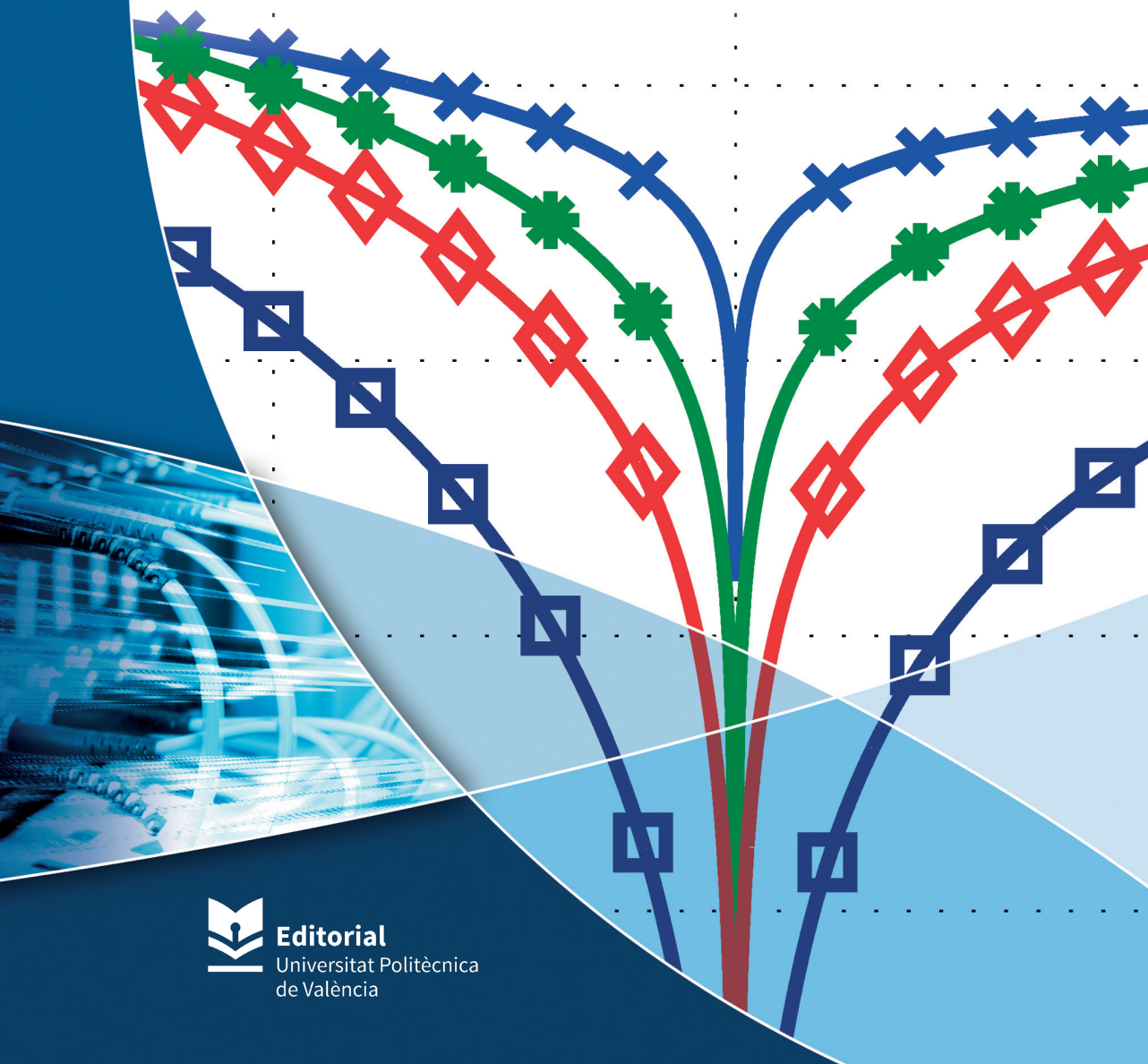
académica

Errores, optimización y resolución numérica de sistemas

2ª edición

Valentín Gregori Gregori
Bernardino Roig Sala



Editorial
Universitat Politècnica
de València

Valentín Gregori Gregori
Bernardino Roig Sala

Errores, optimización y resolución numérica de sistemas

2ª edición

Editorial
Universitat Politècnica
de València

Colección *Académica*

Para referenciar esta publicación utilice la siguiente cita: Gregori Gregori, Valentín; Roig Sala, Bernardino (2019*). Errores, optimización y resolución numérica de sistemas.* Valencia: Editorial Universitat Politècnica de València

Venta: www.lalibreria.upv.es / Ref.: 0246_04_02_01

Imprime: Byprint Percom, sl

ISBN: 978-84-9048-784-6

Presentación

El presente libro es un texto sobre la resolución numérica de problemas esenciales para los alumnos de ingeniería. Básicamente el contenido corresponde a un curso que los autores han impartido en la Escola Politècnica Superior de Gandia de la Universitat Politècnica de València en anteriores años académicos. El poco tiempo de que se dispone para su impartición queda patente, en cierta manera, en la ausencia de demostraciones que sólo aparecen esporádicamente en letra pequeña si éstas permiten entender mejor el capítulo. Ello, sin embargo, permite una lectura más fluida del texto.

Este texto encuentra un hueco entre los textos matemáticos de investigación numérica en los que se estudia la teoría con detalle y se realizan algunas aplicaciones, y aquellos otros más centrados en la ingeniería que sin apenas justificación alguna se dedican a enfatizar las aplicaciones. Sirve de base para primeros cursos de grado centrándose en la parte algorítmica y computacional y para cursos superiores en donde ya es posible incidir con detalle en todo el contenido del libro. En esta segunda edición se han realizado correcciones, mejoras en el texto, algunos añadidos interesantes sin cambiar la estructura del libro y se han reescrito los ejercicios.

Para la comprensión del texto sólo se requieren nociones del análisis matemático y matricial, así como una base de algorítmica y programación. Todos los requisitos teóricos utilizados se han explicitado previamente de forma breve. Los algoritmos o comandos presentados se han escrito de la forma que se ha creído más sencilla y compacta en las versiones actuales de Matlab y Octave. Obviamente dichos algoritmos se pueden escribir de forma más general y abarcando más casuísticas, ahora bien, ello desvía el enfoque de este libro que incide más en la comprensión de los métodos y conceptos tratados. En http://personales.upv.es/broig/ se pueden encontrar todos los comandos incluidos en el texto. Para los cálculos simbólicos en *Octave* se requiere instalar *Python* y sus librerías *Mpmath* y *SymPy* y cargarla dentro de *Octave* con los comandos `pkg install -forge symbolic` y `pkg load symbolic`. Se han recuadrado las ecuaciones más esenciales desde la perspectiva numérica sobre todo para facilitar su uso a estudiantes de primeros cursos.

Los autores agradecerán cualquier sugerencia tendente a mejorar el presente texto en ediciones sucesivas.

Los autores.

NOTACIÓN:

En este texto se ha evitado un lenguaje excesivamente simbólico. No obstante, el lector debe conocer la siguiente terminología básica que se usa en matemáticas y ciencias tecnológicas:

$\forall$	Cuantificador universal. Se lee "para todo" o "para cada"
$\exists$	Cuantificador existencial. Se lee "existe"
$\Longleftrightarrow$	Equivalencia proposicional. Se lee "si y sólo si"
sii	Abreviatura de "si y sólo si"
$\Rightarrow$	Implicación proposicional. La proposición de la izquierda implica la de la derecha. Se lee "implica"
$\mid$	Se lee "tal (tales) que"
:	Se lee "tal (tales) que"
i.e.	En latín *id est* y se lee "es decir"
$\in$	Símbolo de pertenencia
$\subset$	Símbolo de inclusión
$\cup$	Símbolo de unión
$\cap$	Símbolo de intersección
$\mathbb{N}$	Conjunto de los números naturales (incluye al cero)
$\mathbb{N}^*$	El conjunto $\mathbb{N}$ sin el cero
$\mathbb{Z}$	El anillo de los números enteros
$\mathbb{Q}$	El cuerpo de los números racionales
$\mathbb{R}$	El cuerpo de los números reales
$\mathbb{C}$	El cuerpo de los números complejos

Sumario

Capítulo 1

REPRESENTACIÓN NUMÉRICA Y TEORÍA DE ERRORES

1.1 INTRODUCCIÓN

Los ordenadores no pueden representar todos los números reales de forma exacta con un número finito de bits, en particular si tienen infinitos decimales no nulos como ocurre con los números irracionales (π, $\sqrt{2}$, etc.) o los periódicos (1/3, 1/11, ...). Ello obliga a limitar la precisión numérica en la representación. Generalmente, en un ordenador se asigna un número fijo de bits para representar un número. Esta limitación también define el **rango de representación**, o sea, el máximo y mínimo número representable.

La anterior consideración implica un error de representación que se añade a otros posibles errores de medida o de cualquier otro origen. Conforme se realizan operaciones el error se propaga, se acumula y en algunos casos puede llegar a crecer *exponencialmente*, provocando grandes errores en el resultado final.

Por ejemplo, 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 y murieron 28 soldados. La causa fue debida a los errores numéricos provenientes de utilizar truncamiento en lugar de redondeo en el sistema que calcula el momento exacto en que debe ser lanzado el misil [4]. 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 veloci-

dad 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 truncamiento 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.

Otro ejemplo ocurrió el 4 de junio de 1996 cuando el cohete Ariane 5 de la Agencia Europea del Espacio (ESA) explotó 40 segundos después de su despegue a una altura de 3.7 kilómetros tras desviarse de la trayectoria prevista [5]. 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 de 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.

En este capítulo se ofrece una introducción a la representación computacional de números, a la precisión y al estudio de errores y su propagación.

1.2 REPRESENTACIÓN DE NÚMEROS AL 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. La implementación física (o hardware) mediante circuitos electrónicos de estos dispositivos requiere una representación numérica adecuada. 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. Los lenguajes de programación de alto nivel, como Fortran o C, y la mayoría de los paquetes matemáticos, como Matlab, Mathematica y Octave utilizan este sistema de representación de números.

Aunque se verá más adelante con más detalle (ver sección 1.4.1), se recuerda ahora que el **error absoluto** de aproximar un número x por otro x^* es $E_a = |x - x^*|$, mientras que el **error relativo** es $E_r = \frac{E_a}{|x|}$. Habitualmente se trabaja con **cotas de** estos **errores** que son límites superiores del valor que éstos pueden tomar (en la sección 1.4.4 se verá con más detalle).

1.2.1 El sistema decimal y el binario

Aunque no es interés de este libro estudiar con detalle los sistemas de representación numérica y, aunque el lector ya sabrá transformar de una representación decimal a binaria y viceversa, aquí se presenta de modo sucinto dichos algoritmos para una presentación autocontenida de los conceptos necesarios en este capítulo.

Sea x un número natural representado en el **sistema decimal** por los dígitos $d_m d_{m-1} \dots d_2 d_1 d_0$ con $d_i \in \{0,1,2,3,4,5,6,7,8,9\}$ $i = 0, \dots, m$, y que denotaremos $x = (d_m d_{m-1} \dots d_2 d_1 d_0)_{10}$. A d_0 se le llama unidades, a d_1 decenas, a d_2 centenas,... y el valor de x es $x = d_m\ 10^m + d_{m-1}\ 10^{m-1} + \cdots + d_2\ 10^2 + d_1\ 10^1 + d_0\ 10^0$.

Se considera ahora x representado en el **sistema binario** por los dígitos $r_n r_{n-1} \dots r_2 r_1 r_0$ con $r_i \in \{0,1\}$, $i = 0, \dots, n$, y que denotaremos con $x = (r_n r_{n-1} \dots r_2 r_1 r_0)_2$. El valor de x en el sistema decimal es $x = r_n\ 2^n + r_{n-1}\ 2^{n-1} + \cdots + r_2\ 2^2 + r_1\ 2^1 + r_0\ 2^0$.

Para pasar del sistema decimal al binario se presenta a continuación el método de las divisiones euclídeas enteras sucesivas. Sea x un número natural en el sistema decimal. Se divide x entre 2 y se obtiene un cociente c_0 y un resto r_0. Seguidamente se toma el cociente como dividendo y se vuelve a dividir entre 2 obteniendo un nuevo cociente c_1 y un nuevo resto r_1. Se repite el proceso anterior hasta que el cociente es 0. En resumen, el algoritmo es:

$$\left.\begin{array}{rl} x = & 2 \cdot c_0 + r_0,\ 0 \le r_0 < 2 \\ c_0 = & 2 \cdot c_1 + r_1,\ 0 \le r_1 < 2 \\ c_1 = & 2 \cdot c_2 + r_2,\ 0 \le r_2 < 2 \\ \vdots & \\ c_{n-2} = & 2 \cdot c_{n-1} + r_{n-1},\ 0 \le r_{n-1} < 2 \\ c_{n-1} = & 2 \cdot 0 + r_n,\ 0 \le r_n < 2 \end{array}\right\} \Rightarrow x = (r_n r_{n-1} \dots r_2 r_1 r_0)_2\,.$$

Con una argumentación que omitimos se puede sistematizar la conversión de un decimal menor que uno a un número binario como se describe a continuación. Si x es un número decimal menor que 1, entonces x admite la escritura obvia $x = 0.d$ donde d es la parte decimal de x. Se halla el producto (decimal) $2x$ cuyo resultado se puede escribir como $2x = 2 \cdot 0.d = a_1.d_1$, donde a_1 es entero y d_1 la parte decimal del producto obtenido. Se halla ahora el producto $2 \cdot 0.d_1$ que se podrá escribir, al igual que antes, como $a_2.d_2$ donde a_2 es un entero y d_2 la parte decimal del producto obtenido. Se halla ahora el producto $2 \cdot 0.d_2$ que se escribirá como $a_3.d_3$ como antes y así se va repitiendo el proceso. La expresión de x en binario es entonces $0.a_1a_2a_3\ldots$. Esta expresión se termina obteniendo una representación finita cuando aparece un d_i nulo o cuando se alcanza el máximo número de dígitos representables.

Ejemplo

Escribamos en binario 14.375 y 14.5627.

El algoritmo anterior permite obtener la expresión binaria del entero 14 como 1110. Su parte decimal 0.375 se convierte en binario como sigue:

$$\begin{aligned} 2 \cdot 0.375 &= 0.75 \quad \text{por lo que } a_1 = 0,\ d_1 = 75, \\ 2 \cdot 0.75 &= 1.5 \quad \text{por lo que } a_2 = 1,\ d_2 = 5, \\ 2 \cdot 0.5 &= 1 \quad \text{por lo que } a_3 = 1,\ d_3 = 0. \end{aligned}$$

Por tanto 0.375 se representa de forma exacta en binario como 0.011. En consecuencia, el número decimal 14.375 en binario se escribe como 1110.011.

La parte decimal 0.5627 del segundo número se convierte en binario como sigue:

$$\begin{aligned} 2 \cdot 0.5627 &= 1.1254 \quad \text{por lo que } a_1 = 1,\ d_1 = 1254, \\ 2 \cdot 0.1254 &= 0.2508 \quad \text{por lo que } a_2 = 0,\ d_2 = 2508, \\ 2 \cdot 0.2508 &= 0.5016 \quad \text{por lo que } a_3 = 0,\ d_3 = 5016, \\ 2 \cdot 0.5016 &= 1.0032 \quad \text{por lo que } a_4 = 1,\ d_4 = 0032 \end{aligned}$$

y así sucesivamente. Por tanto, 0.5627 es $0.1001\ldots$ en binario. En consecuencia el número decimal 14.5627 se escribe en binario como $1110.1001\ldots$

1.2.2 Números naturales

Los números se almacenan en *variables*. Una variable representa un trozo de la memoria del computador. La memoria está formada por una gran cantidad de bytes y cada byte está constituido por 8 bits. Un bit puede almacenar un 1 o un 0. Si un computador tiene, por ejemplo, 32 mega bytes

de memoria (MB) esto significa que tiene $32 \cdot 1024 \cdot 1024$ bytes (es decir, $32 \cdot 1024 \cdot 1024 \cdot 8$ bits).

Una variable x natural o entera positiva (uint -unsigned integer-) está formada por 4 bytes, es decir 32 bits que denotaremos x_{31}, x_{30}, ... , x_2, x_1 y x_0. El valor numérico decimal representado por la variable x en binario viene determinado por:

$$\text{valor}(x) = x_{31} \cdot 2^{31} + x_{30} \cdot 2^{30} + ... + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0.$$

1.2.3 Números enteros

Una variable x entera (int -integer-) está formada por 4 bytes, es decir 32 bits que denotaremos x_{31}, x_{30}, ... , x_2, x_1 y x_0 pero en este caso el bit de más a la izquierda se utiliza para indicar el signo. Por tanto, el valor máximo representable ahora es inferior al caso anterior (uint). El valor numérico decimal representado por la variable x en binario viene determinado por:

- Si x_{31} es 0, el valor es positivo o cero y se calcula como:

$$\text{valor}(x) = x_{30} \cdot 2^{30} + ... + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0.$$

- Si x_{31} es 1, entonces x representa un entero negativo y se calcula construyendo una nueva variable y, tal que $y_i = 1$ si $x_i = 0$ e $y_i = 0$ si $x_i = 1$ para $i = 0, \ldots, 31$. Se dice que y es el complemento de x respecto 2^{32}. De este modo,

$$\text{valor}(x) = -(\text{valor}(y) + 1).$$

Ejemplo

- $\text{valor}(000...001001) = 1 \cdot 2^3 + 1 \cdot 2^0 = 9.$
- $\text{valor}(111...111010) = -(\text{valor}(000...000101) + 1) = -(5 + 1) = -6.$

Una variable entera (int) siempre utiliza 32 bits, aun cuando el número sea pequeño. Por otra parte, no es capaz de almacenar números demasiado grandes en valor absoluto. El valor máximo y mínimo representable por este tipo de variable es:

$$\text{Máximo valor} = \text{valor}(011...111111) = 2^{31} - 1.$$
$$\text{Mínimo valor} = \text{valor}(100...000000) = -2^{31}.$$

Por lo tanto, con una variable entera (int) se pueden almacenar números de 9 dígitos aproximadamente.

Observación

Usualmente se usa el tipo int para almacenar números enteros, pero también existen otros tipos. La Tabla 1.1 muestra los tipos enteros habituales y su **rango de representación**, es decir, los valores mínimo y máximo representables.

Tipo	Bits	Rango
int	32	$[-2^{31}, 2^{31}-1]$
short	16	$[-2^{15}, 2^{15}-1]$
long	64	$[-2^{63}, 2^{63}-1]$

Tabla 1.1: Tipos de representación habituales de números enteros

El rango de representación de los enteros es limitado. Si el resultado de una operación aritmética (como la multiplicación) excede el rango de representación se producirá un error en la magnitud del valor almacenado.

1.2.4 Números reales

Una variable real *normal* (float) está formada por 8 bytes, es decir el doble de un entero. En ella se almacenan números reales en formato binario. Por ejemplo, el número binario 1101.1010 representa en decimal al número

$$1\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0+1\cdot 2^{-1}+0\cdot 2^{-2}+1\cdot 2^{-3}+0\cdot 2^{-4} = 8+4+1+0.5+0.125\,.$$

Los primeros ordenadores almacenaban los números reales en una formato llamada **representación en punto fijo**, en donde se destinaba una cantidad fija de bits para la parte entera (por ejemplo 32 bits) y el resto a la parte fraccionaria. Por lo tanto, el número anterior sería representado en 64 bits como:

00000000000000000000000000001101.10100000000000000000000000000000

En esta representación, el punto que separa la parte entera de la parte fraccionaria siempre se ubica en una posición fija y de ahí su nombre. El problema de esta representación es que el rango de valores representables es el mismo de los números enteros. Hace falta un mecanismo que aumente el rango de números representables. Para ello se destinó una parte de los bits a indicar la posición del punto, y este formato pasó a llamarse **representación**

en punto flotante. La notación comercial habitual suele utilizar la coma decimal para separar la parte entera de la fraccionaria reservando el punto para los millares, por lo que también se suele hablar de **representación 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 para separar la parte entera de la fraccionaria. Además, no utilizaremos la coma para los millares para evitar confusiones.

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

$$\boxed{x = \pm(0.d_1 d_2 \dots d_m)_b \cdot b^E = \pm\left(d_1 \cdot b^{-1} + d_2 \cdot b^{-2} + \cdots + d_m \cdot b^{-m}\right) \cdot b^E}$$

donde $d_1 \neq 0$, $0 \leq M = 0.d_1 d_2 \dots d_m < 1$ es la mantisa y E es el exponente entero de x 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.

La mantisa es una secuencia de bits que siempre comienza en 1 en binario. El exponente indica la posición donde colocar en la mantisa el punto que separa la parte entera de la fraccionaria. La **precisión** viene determinada por el número de dígitos m de su mantisa M en punto flotante. 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 un número real y también es necesario más tiempo de cómputo para realizar operaciones con él.

Ejemplos

Como ya se ha visto, un número real se almacena en una variable especificando 3 componentes: el signo, la mantisa y el exponente. Por ejemplo, el número decimal $11 = 2^3 + 2^1 + 2^0$ que se corresponde en binario a 1011_2, se representa como $+0.1011000_2 \cdot 2^{+4}$, es decir,

- signo: $+$.
- mantisa: 10110000... (tantos 0 como determine la precisión).
- exponente: $+4$.

A continuación se consideran otros casos.

- Con el signo $-$, la mantisa 110010000... y el exponente 7 se obtiene en binario -1100100.00... que se corresponde con $-(64+32+4) = -100$ en decimal.
- Con el signo $+$, la mantisa 1000... y el exponente 100 se obtiene en binario $\overbrace{100\ldots0}^{100}.00_2$ que se corresponde con 2^{99} en decimal.
- Con el signo $+$, la mantisa 101000... y el exponente 0 se obtiene en binario 0.101_2 que se corresponde con $0.5 + 0.125 = 0.625$ en decimal.
- Con el signo $+$, la mantisa 1000... y el exponente -3 se obtiene en binario 0.0001_2 que se corresponde con $2^{-4} = 0.0625$ en decimal.

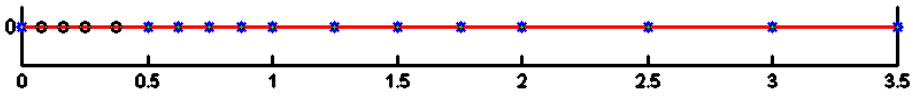


Figura 1.1: Representación decimal de los números flotantes positivos en un sistema con 3 bits en la mantisa y el exponente entre -1 y 2.

El conjunto de números flotantes no está distribuido de forma uniforme entre el máximo y el mínimo. En la Figura 1.1 se representan todos los números flotantes positivos con una mantisa de tres dígitos binarios (es decir, 100, 101, 110, 111) y un exponente de dos dígitos binarios, es decir, $-1 \leq E \leq 2$. Se observa que la densidad de los números se reduce conforme nos alejamos del origen y, por tanto, no todos los números se opueden representar con la misma precisión.

Observación

Normalmente se usa el tipo punto flotante (float) para almacenar números reales, pero también existen otros tipos. La Tabla 1.2 muestra los tipos enteros habituales y su rango de representación.

Los números reales se almacenan con una precisión limitada. Esto hace que al realizar multiplicaciones y divisiones, los valores que se obtienen poseen un error de precisión que no suele ser relevante para la mayoría de las aplicaciones numéricas.

1.2.5 Ejemplo con 4 decimales en binario

Se considera ahora una representación de números reales en binario mediante 4 decimales en la mantisa y un exponente de la base entre -3 y 4,

Tipo	**Bits**	**Bits M**	**Bits E**	**Rango aprox.**	**Precisión**
float	32	23	8	$[-10^{30}, 10^{30}]$	7
double	64	52	11	$[-10^{300}, 10^{300}]$	15
quadruple	128	112	15	$[-10^{3000}, 10^{3000}]$	34

Tabla 1.2: Tipos de representación habituales de números reales.

	$n=-3$	$n=-2$	$n=-1$	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$
0.1000_2	0.0625	0.125	0.25	0.5	1	2	4	8
0.1001_2	0.0703125	0.140625	0.28125	0.5625	1.125	2.25	4.5	9
0.1010_2	0.078125	0.15625	0.3125	0.625	1.25	2.5	5	10
0.1011_2	0.0859375	0.171875	0.34375	0.6875	1.375	2.75	5.5	11
0.1100_2	0.09375	0.1875	0.375	0.75	1.5	3	6	12
0.1101_2	0.1015625	0.203125	0.40625	0.8125	1.625	3.25	6.5	13
0.1110_2	0.109375	0.21875	0.4375	0.875	1.75	3.5	7	14
0.1111_2	0.1171875	0.234375	0.46875	0.9375	1.875	3.75	7.5	15

Tabla 1.3: Conjunto de todos los números reales representables en binario de la forma $0.d_1d_2d_3d_4 \cdot 2^n$, $n \in \{-3,-2,-1,0,1,2,3,4\}$.

es decir, todos los números representables de la forma $0.d_1d_2d_3d_4 \cdot 2^n$ (con $d_1 \neq 0$), $n \in \{-3,-2,-1,0,1,2,3,4\}$. Todos los posibles números representables de este modo se encuentran en la Tabla 1.3.

Supongamos que ahora queremos calcular $(1/10 + 1/5) + 1/6$ con esta representación binaria de los números reales en nuestro ordenador y que, como es habitual, aproxima todos los números reales al número binario más próximo que dispone.

Primero representamos los números reales en el sistema considerado: $\frac{1}{10} \simeq 0.1101_2 \cdot 2^{-3}$, $\frac{1}{5} \simeq 0.1101_2 \cdot 2^{-2}$, $\frac{1}{6} \simeq 0.1011_2 \cdot 2^{-2}$. A continuación se realiza la primera suma según el orden dado por los paréntesis. Para ello primero se igualan los exponentes y después se suma la mantisa:

$$\begin{aligned} \tfrac{1}{10} + \tfrac{1}{5} &\simeq 0.01101_2 \cdot 2^{-2} + 0.1101_2 \cdot 2^{-2} = 1.00111_2 \cdot 2^{-2} \\ &= 0.0100111_2 = 0.304687 \text{ en decimal} \end{aligned}$$

que en la Tabla 1.3 se aproxima por $0.1010_2 \cdot 2^{-1}$. Seguidamente se procede a la segunda suma:

$$\begin{aligned}\left(\tfrac{1}{10}+\tfrac{1}{5}\right)+\tfrac{1}{6} &= 0.1010_2\cdot 2^{-1}+0.01011_2\cdot 2^{-1}=0.11111_2\cdot 2^{-1}\\ &= 0.011111_2\simeq 0.48437 \text{ en decimal}\end{aligned}$$

que en la Tabla 1.3 se aproxima por $0.1000_2 \cdot 2^0$. Por lo tanto, la solución al cálculo pedido es $0.1000_2 \cdot 2^0$ que conlleva un error absoluto

$$E_a\left(\left(\frac{1}{10}+\frac{1}{5}\right)+\frac{1}{6}\right)=\left|\frac{7}{15}-0.1000_2\right|\simeq|0.4667-0.5000|=0.0333.$$

En consecuencia, al sumar las tres fracciones anteriores de la forma indicada se ha cometido un error relativo

$$E_r\left(\left(\frac{1}{10}+\frac{1}{5}\right)+\frac{1}{6}\right)\simeq\frac{0.0333}{0.4667}\simeq 0.0714,$$

es decir, se ha obtenido un error de representación del 7%.

Se deja al lector verificar el resultado de $1/10 + (1/5 + 1/6)$ en este sistema de representación y observará que el resultado difiere un poco con el anterior. Ello significa que la suma de números reales no es asociativa en este sistema de representación.

Para efectuar el producto de dos números, primero se representan en binario, luego se multiplican las mantisas y se suman los exponentes.

1.2.6 El formato en punto flotante IEEE-754

Existen varios formatos para la representación de números flotantes en un ordenador aunque el habitual, y por ello el utilizado en la mayoría de los ordenadores, es el formato ANSI/IEEE standard 754-1985 [14, 16], que llamaremos **IEEE-754** para abreviar. En este formato los números flotantes se representan en sistema binario (base 2), con 1 bit para el signo S (0 para positivo y 1 para negativo), e bits para el exponente E y m bits para la mantisa M, en este orden (véase la Figura 1.2). Generalmente el número total de bits es un múltiplo de 16 (2 bytes). Se pueden representar números en **precisión simple** (float), **precisión doble** (double) y **precisión cuádruple** (quadruple) como ya se ha visto en la Tabla 1.2.

La representación de un número flotante de precisión simple en IEEE-754 se realiza en palabras de 32 bits como se indica a continuación.

El primer bit de la izquierda corresponde al signo S, como se ha dicho anteriormente.

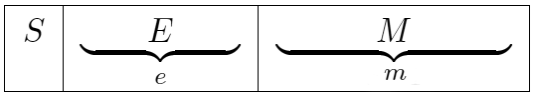


Figura 1.2: Representación de un número flotante en binario con $1 + e + m$ bits.

El rango del exponente está entre -126 y 127. Ahora bien, el exponente E se expresa como un entero positivo en binario con 8 dígitos. Por tal razón, tras desplazarlo 127 unidades, el rango de E se expresa entre 1 y 254. A modo de ejemplo, el exponente 0 se convierte en el 127. Los exponentes 0 y 255 están reservados para significados especiales del tipo *overflow*, *NaN* y 0.

Finalmente, para representar la mantisa M se procede como sigue. El número real x en cuestión se escribe en sistema binario. A continuación, este número binario se escribe en forma exponencial de modo que sólo haya un 1 (inmediatamente) a la izquierda del punto decimal (se dice que está normalizado). Los dígitos a la derecha del punto decimal, hasta 23 (completando con ceros a la derecha si es necesario), es la mantisa M.

Ejemplo

Vamos a representar el número -13.4375 en el formato IEEE-754 de precisión simple.

Como 13 en binario es 1101 y $0.4375 = 0{\cdot}0.5{+}1{\cdot}0.25{+}1{\cdot}0.125{+}1{\cdot}0.0625$, se tiene que 13.4375 en binario es $1101 + 0.0111 = 1101.0111$. En forma exponencial se escribe, según se ha indicado arriba, $1.1010111\ E{+}3$, con lo que los primeros dígitos de la mantisa M son 1010111 que habrá que completar con ceros a la derecha hasta 23 dígitos. El exponente $+3$ desplazado 127 unidades es 130 que en binario se escribe 10000010. Por tanto, el formato del número -13.4375 en el formato IEEE-754 es

S	E	M
1	10000010	10101110000000000000000

1.2.7 El formato punto flotante de precisión doble

La representación de un **número flotante de precisión doble en IEEE-754** se realiza en palabras de 64 bits de manera similar al caso de precisión simple con las siguientes diferencias:

- El rango del exponente E está entre -1022 y 1023. Como éste se expresa como entero positivo en binario con 11 dígitos, se desplaza en 1023 unidades, con lo que el rango de E se expresa entre 1 y 2046.

- La mantisa M se escribe como con el caso de precisión simple usando ahora 52 bits.

Como el primer dígito de la mantisa normalizada debe ser un 1 (en binario), se aprovecha este bit para almacenar en su lugar el signo de la mantisa. El exponente máximo 1024, se reserva para representar los números excepcionales $\pm\infty$ y *NaN* (*Not a Number*). Los valores $\pm\infty$ se presentan cuando se produce una operación aritmética que genera un número más grande que el máximo representable, es decir, se produce un desbordamiento por exceso. *NaN* se genera en operaciones aritméticas de resultado no determinado, como $0/0$, $\infty - \infty$, ∞/∞, etc.

A los números con la representación anterior se les llama *normales*, y se pueden extender para representar números mucho más pequeños en valor absoluto eliminando la normalización de la mantisa.

Se denomina **épsilon** de la máquina al valor asociado con el último dígito representable en la mantisa cuando el exponente es cero cuyo valor en doble precisión es $\varepsilon = 2^{-52} \simeq 2.22 \cdot 10^{-16}$.

Dado que existe una cantidad finita de números flotantes, 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 tiene una excepción por desbordamiento por exceso o de tipo ***overflow***.

Ejemplo

La función seno se define de $\mathbb{R}$ a $[-1, 1]$, mientras que la función arcoseno habitualmente se define de $[-1, 1]$ a $[-\pi/2, \pi/2]$. Es por ello que se obtiene que $\text{asen}(\,\text{sen}(\pi)) = 0$ mientras que si se realizan los cálculos anteriores en Matlab con los comandos `asin(sin(pi))` no se obtiene `0`, sino que se obtiene `1.2246e-016`, es decir, el épsilon en doble precisión (aproximadamente). Podemos afirmar que obtener este valor u obtener cero es lo mismo desde una perspectiva computacional cuando se usa precisión doble.

Observación

Como consecuencia de los párrafos anteriores se tiene que si x es una variable real con doble precisión, cualquier valor inferior a 10^{-16} en valor absoluto debe ser considerado como 0. Esta cota de 0 para x puede tomar

valores más elevados como 10^{-14} o superiores cuando esta variable tiene errores acumulados como se verá más adelante.

El exponente d en 10^{-d} de una cota superior cercana al valor del épsilon de la máquina viene dado en la última columna de la Tabla 1.2 para distintas precisiones. Se puede obtener de forma más exacta como 2^{-M} donde M es la tercera columna de dicha Tabla.

Para trabajar con la variable $x = 3$ en punto flotante en precisión simple en Matlab y Octave basta definir `x=single(3)`. Para hacerlo en precisión doble se utiliza `x=double(3)` aunque ello no es necesario ya que, por defecto, se trabaja en precisión doble, basta con `x=3`.

Ejemplo

En este ejemplo, dado el número real $x = 123.456$ se calculará la representación binaria IEEE754 en precisión doble a partir de la decimal y recíprocamente así como los comandos Matlab u Octave utilizados para ello. En los comandos siguientes primero se separa el signo, la parte entera y la parte decimal de x. Seguidamente se pasa a binario la parte entera y la decimal siguiendo los algoritmos indicados en la sección 1.2.1. A continuación se eliminan los ceros que hayan podido quedar a la izquierda para obtener el exponente en binario para la representación en punto flotante. Finalmente se pasa a binario el exponente utilizando 11 dígitos y se juntan todos los valores obteniendo

0 10000000101 1110110111010010111100011010100111111011111001110111.

Los comandos Matlab u Octave utilizados para ello son los que se muestran a continuación. Se coloca un 1 al final de `xbi` para asegurar la finalización del bucle que no altera el resultado final. Se calculan 1022+52+1 decimales en binario para asegurar una representación correcta de números cercanos al épsilon.

```
x=123.456,signo=(sign(x)==-1);x=abs(x);xent=floor(x);xdec=x-xent;
%% Parte entera a binario:
res=mod(xent,2);xentb=mod(xent,2);xent=(xent-xentb)/2;
while(xent>1)res=mod(xent,2);xentb=[mod(xent,2),xentb];
xent=(xent-xentb(1))/2;end; if(xent>0)xentb=[xent,xentb];end;
%% Parte decimal a binario:
xdecb=[];for i=1:1075, xdec=xdec*2;xdecb=[xdecb,floor(xdec)];
xdec=xdec-floor(xdec); end;
%%Numero en binario y calculo del exponente:
xbi=[xentb,xdecb,1];
nzero=0;while(xbi(1)==0)xbi=xbi(2:length(xbi));nzero=nzero+1;end;
```

Para seguir leyendo haga click aquí