Representación de los números en la computadora.

Sólo hay 10 tipos de personas:
las que saben binario y las que no.

Pablo Santamaría
Facultad de Ciencias Astronómicas y Geofísicas
Universidad Nacional de La Plata
email: pablo@fcaglp.unlp.edu.ar

v1.0 (Septiembre 2009)

Introducción

Consideremos el siguiente código escrito en Fortran.

```
program main
implicit none
integer i
i = 2**31 -1
write (*,*) i+1
end
```

Al compilarlo y ejecutar el programa resultante obtenemos la siguiente salida:

```
-2147483648
```

Es decir, ¡al sumar dos números positivos obtenemos un número negativo! Ahora consideremos este otro programa en Fortran.

```
program main
implicit none
real a
a = 2.0
if (1.0/a.eq.0.5) then
     write(*,*) '1/2 es igual a 0.5'
else
     write(*,*) '1/2 es distinto a 0.5'
endif
a = 10.0
if (1.0/a.eq.0.1) then
     write(*,*) '1/10 es igual a 0.1'
else
     write(*,*) '1/10 es distinto a 0.1'
endif
end
```

Al compilarlo y ejecutarlo obtenemos como salida:

```
1/2 es igual a 0.5
1/10 es distinto a 0.1
```

Parece haber algún tipo de error. ¡Los números no se están comportando como deberían!

Los dos ejemplos anteriores exponen ciertas características de la *representación en la computadora* de los números enteros y los números reales. Pensemos un momento, el sistema de número reales constituye un conjunto altamente sofisticado en virtud de las propiedades que posee: (i) no está acotado ni superior ni inferiormente, (ii) es infinitamente *denso*, esto es, entre dos números reales siempre hay un número real, (iii) es *completo*, esto es, el límite de toda sucesión convergente de números reales es un número real, (iv) las operaciones aritméticas definida sobre ellos satisfacen una gran cantidad de propiedades, como ser los *axiomas de cuerpo*, (v) es un conjunto ordenado. Ahora bien, para poder realizar los cálculos en forma rápida y eficiente, todo dispositivo de cálculo, como una computadora o una calculadora, representa a los números con una cantidad *finita* y *fija* de dígitos. Consecuencia inmediata de ésto es que no todas las propiedas de los números reales se satisfacen en la representación de los mismos. La forma de la representación y sus consecuencias son discutidas en lo largo de estas notas.

Estas notas distan de ser originales. Todo el material, incluso muchos de los ejemplos, han sido extraído de diversas fuentes[*]. Sin embargo, el enfoque si es propio: el propósito de estas notas es tratar de dar a la gente que hace *ciencia con computadoras*, pero *que no son informáticos*, una explicación lo más completa posible de los aspectos a tener en cuenta sobre la representación de los números en la computadora con el fin de que puedan comprender (y evitar) por que suceden ciertas cosas (como las de nuestros programas anteriores) al utilizar software ya sea programado por uno mismo o por terceros. Estas notas están divididas en cuatro secciones principales: primeramente introducimos el *sistema posicional* para una base cualquiera, luego discutimos la *representación de punto fijo*, utilizada para representar números enteros, y posteriormente la *representación de punto flotante*, utilizada para representar los números reales. Cierran estas notas una discusión de la *propagación de errores de redondeo* en los algoritmos numéricos.

El sistema posicional

Cotidianamente, para representar los números utilizamos un *sistema posicional* de *base* 10: el *sistema decimal*. En este sistema los números son representados usando diez diferentes caracteres, llamados *dígitos decimales*, a saber, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. La magnitud con la que un dado dígito a contribuye al valor del número depende de su posición en el número de manera tal que, si el dígito ocupa la posición n a la izquierda del punto decimal, el valor con que contribuye es $a\,10^{n-1}$, mientras que si ocupa la posición n a la derecha del punto decimal, su contribución es $a\,10^{-n}$. Por ejemplo, la secuencia de dígitos 472.83 significa

$$472.83 = 4 \cdot 10^2 + 7 \cdot 10^1 + 2 \cdot 10^0 + 8 \cdot 10^{-1} + 3 \cdot 10^{-2}.$$

En general, la representación decimal

$$(-1)^s(a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots)$$

corresponde al número

$$(-1)^s(a_n 10^n + a_{n-1}10^{n-1} + \cdots + a_1 10^1 + a_0 10^0 + a_{-1}10^{-1} + a_{-2}10^{-2} + \ldots),$$

donde s depende del signo del número ($s = 0$ si el número es positivo y $s = 1$ si es negativo). De manera análoga se puede concebir otros sistemas posicionales con una base distinta de 10. En

[*] Ver sección de referencias.

principio, cualquier número natural $\beta \geq 2$ puede ser utilizado como base. Entonces, fijada una base, todo número real admite una *representación posicional* en la base β de la forma

$$(-1)^s(a_n\beta^n + a_{n-1}\beta^{n-1} + \cdots + a_1\beta^1 + a_0\beta^0 + a_{-1}\beta^{-1} + a_{-2}\beta^{-2} + \dots),$$

donde los coeficientes a_i son los "dígitos" en el sistema con base β, esto es, enteros positivos tales que $0 \leq a_i \leq \beta - 1$. Los coeficientes $a_{i\geq 0}$ se consideran como los dígitos de la *parte entera*, en tanto que los $a_{i<0}$, son los dígitos de la *parte fraccionaria*. Si, como en el caso decimal, utilizamos un punto para separar tales partes, el número es representado en la base β como

$$(-1)^s(a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots)_\beta,$$

donde hemos utilizado el subíndice β para evitar cualquier ambigüedad con la base escogida.

Una de las grandes ventajas de los sistemas posicionales es que se pueden dar reglas generales simples para las operaciones aritméticas[*]. Además tales reglas resultan más simples cuanto más pequeña es la base. Esta observación nos lleva a considerar el sistema de base $\beta = 2$, o *sistema binario*, en donde sólo tenemos los dígitos 0 y 1. Pero existe otra razón. Una computadora, en su nivel más básico, sólo puede registrar si fluye o no electricidad por cierta parte de un circuito. Estos dos estados pueden representar entonces dos dígitos, convencionalmente, 1 cuando hay flujo de electricidad, 0 cuando no lo hay. Con una serie de circuitos apropiados una computadora puede entonces contar (y realizar operaciones aritméticas) en el sistema binario. El sistema binario consta, pues, sólo de los dígitos 0 y 1, llamados *bits* (del inglés *binary digits*). El 1 y el 0 en notación binaria tienen el mismo significado que en notación decimal

$$0_2 = 0_{10}, \quad 1_2 = 1_{10},$$

y las tablas de adición y multiplicación toman la forma

$$\begin{array}{lll} 0+0=0, & 0+1=1+0=1, & 1+1=10, \\ 0\cdot 0=0, & 0\cdot 1=1\cdot 0=0, & 1\cdot 1=1. \end{array}$$

Otros números se representan con la notación posicional explicada anteriormente. Así, por ejemplo, 1101.01 es la representación binaria del número 13.25 del sistema decimal, esto es,

$$(1101.01)_2 = (13.25)_{10},$$

puesto que,

$$1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 1\cdot 2^{-2} = 13 + 0.25 = 13.25.$$

Además del sistema binario, otros dos sistemas posicionales resultan de interés en el ámbito computacional, a saber, el sistema con base $\beta = 8$, denominado *sistema octal*, y el sistema con base $\beta = 16$, denominado *sistema hexadecimal*. El sistema octal usa dígitos del 0 al 7, en tanto que el sistema hexadecimal usa los dígitos del 0 al 9 y las letras A, B, C, D, E, F[**]. Por ejemplo,

$$(13.25)_{10} = (1101.01)_2 = (15.2)_8 = (\mathrm{D}.4)_{16}$$

La gran mayoría de las computadoras actuales (y efectivamente *todas* las computadoras personales, o PC) utilizan internamente el sistema binario ($\beta = 2$). Las calculadoras, por su parte, utilizan el sistema decimal ($\beta = 10$). Ahora bien, cualquiera sea la base β escogida, todo dispositivo de cálculo sólo puede almacenar un número *finito* de dígitos para representar un número. En particular, en una computadora sólo se puede disponer de un cierto número finito fijo N de

[*] Por el contrario, en un sistema no-posicional como es la notación de números romanos las operaciones aritméticas resultan de gran complejidad.

[**] Para sistemas con base $\beta > 10$ es usual reemplazar los dígitos $10, 11, \dots, \beta - 1$ por las letras A, B, C, ...

posiciones de memoria para la representación de un número. El valor de N se conoce como *longitud de palabra* (en inglés, *word length*). Además, aún cuando en el sistema binario cualquier número puede representarse tan sólo con los dígitos 1 y 0, el signo $-$ y el punto, la representación interna en la computadora no tiene la posibilidad de disponer de los símbolos signo y punto. De este modo una de tales posiciones debe ser reservada de algún modo para indicar el signo y cierta distinción debe hacerse para representar la parte entera y fraccionaria. Esto puede hacerse de distintas formas. En las siguientes secciones discutiremos, en primer lugar, la *representación de punto fijo* (utilizada para representar los números enteros) y, en segundo lugar, la *representación de punto flotante* (utilizada para representar los números reales).

Conversión entre sistemas numéricos. Ocasionalmente puede presentarse el problema de obtener la representación de un número, dado en un sistema, en otro, por cálculos a mano[*]. Es fácil convertir un número de notación en base β a decimal. Todo lo que se necesita es multiplicar cada dígito por la potencia de β apropiada y sumar los resultados. A continuación veremos como puede convertirse la representación decimal de un número a un sistema de base β. El procedimiento trata a las partes entera y fraccionaria por separado, así que supongamos, en primer lugar, que el número p dado es un entero positivo. Entonces, en la base $\beta \geq 2$, este puede ser escrito en la forma

$$p = a_n\beta^n + a_{n-1}\beta^{n-1} + \cdots + a_1\beta^1 + a_0 = \sum_{j=0}^{n} a_j\beta^j,$$

donde a_j son los dígitos que queremos determinar. La igualdad anterior puede ser escrita en la forma

$$p = a_0 + \beta\left(\sum_{j=1}^{n} a_j\beta^{j-1}\right),$$

de donde se sigue que el dígito a_o se identifica con el resto de la *división entera* de p por la base β:

$$a_0 = \mathrm{mod}(p, \beta).$$

Considerando ahora el cociente entero de tal división

$$c_1 = \sum_{j=1}^{n} a_j\beta^{j-1} = a_1 + \beta\left(\sum_{j=2}^{n} a_j\beta^{j-2}\right),$$

resulta que el dígito a_1 es el resto de la división entera de c_1 por β:

$$a_1 = \mathrm{mod}(c_1, \beta).$$

Procediendo sucesivamente tenemos que, si c_i denota el cociente de la división entera por β del paso anterior (definiendo $c_0 = p$), tenemos que

$$c_i = a_i + \beta\left(\sum_{j=i+1}^{n} a_j\beta^{j-i}\right), \qquad i = 0, 1, \ldots, n,$$

con lo cual

$$a_i = \mathrm{mod}(c_i, \beta).$$

El proceso se detiene en n, puesto que le dividendo en el paso $(n+1)$ es cero. En efecto, siendo $c_n = a_n = a_n + 0 \cdot \beta$, es $c_n = 0$. El procedimiento puede ser resumido esquemáticamente como sigue.

Para convertir un número entero positivo de base 10 a base $\beta \geq 2$.

Paso 1. Realice la división entera por β del cociente obtenido en el paso anterior, comenzando con el número dado.

Paso 2. Guarde el resto de tal división.

Paso 3. Continúe con el paso 1 hasta que el dividendo sea cero.

Paso 4. Lea los restos obtenidos, desde el último al primero, para formar la representación buscada.

Ejemplo: Convertir 29 a binario.

[*]O al menos nos interesa saber como podría hacerse.

29		
$29/2 = 14$	resto 1	
$14/2 = 7$	resto 0	
$7/2 = 3$	resto 1	
$3/2 = 1$	resto 1	
$1/2 = 0$	resto 1	$\uparrow$

Entonces $(29)_{10} = (11101)_2$.

Consideremos ahora la parte fraccional del número en su representación decimal. Podemos entonces asumir que el número p dado es un número real positivo entre 0 y 1. Entonces, en la base $\beta \geq 2$, puede ser escrito en la forma

$$p = a_{-1}\beta^{-n} + a_{-2}\beta^{-2} + \cdots = \sum_{j=1}^{n} a_{-j}\beta^{-j},$$

donde a_{-j} son los dígitos que queremos determinar. Multiplicando la igualdad anterior por β tenemos que

$$q_1 = \beta p = a_{-1} + \sum_{j=2}^{n} a_{-j}\beta^{-j+1},$$

de donde se sigue que el dígito a_{-1} se identifica con la *parte entera* de la multiplicación de p por β:

$$a_{-1} = [q_1]\,.$$

Consideremos ahora la *parte fraccionaria* de tal producto, si ésta no es nula podemos volver a multiplicar por β para obtener

$$q_2 = \beta(q_1 - [q_1]) = a_{-2} + \sum_{j=3}^{n} a_{-j}\beta^{-j+2},$$

de donde,

$$a_{-2} = [q_2]\,.$$

Procediendo sucesivamente, si q_i denota el producto de β por la parte fraccionaria del paso anterior (definiendo $q_1 = \beta p$), tenemos que

$$q_i = a_{-i} + \sum_{j=i+1}^{n} a_{-j}\beta^{-j+i}, \qquad i = 1, 2, \ldots,$$

de donde,

$$a_{-i} = [q_i]\,.$$

Notemos que el procedimiento continúa *indefinidamente*, a menos que para algún q_k su parte fraccionaria sea nula. En tal caso $q_{k+1} = \beta(q_k - [q_k]) = 0$ y, en consecuencia, $a_{-(k+1)} = a_{-(k+2)} = \cdots = 0$, obteniéndose así una cantidad finita de dígitos fraccionarios respecto de la base β. Resumimos el procedimiento como sigue.

Para convertir un número decimal entre 0 y 1 a la base $\beta \geq 2$.

Paso 1. Realice la multiplicación por β de la parte fraccionaria obtenida en el paso anterior, comenzando con el número dado.

Paso 2. Guarde la parte entera de tal producto.

Paso 3. Continúe con el paso 1 hasta que la parte fraccionaria sea cero, o hasta obtener el suficiente número de dígitos requeridos para su representación.

Paso 4. Lea las partes enteras obtenidas, del principio hacia el final, para formar la representación buscada.

Ejemplo: Convertir 0.625 a binario.

0.625		
2*0.625 = 1.25	[1.25] = 1	$\downarrow$
2*0.25 = 0.5	[0.5] = 0	
2*0.5 = 1	[1] = 1	

Nos detenemos puesto que $1 - [1] = 0$. Así, $(0.625)_{10} = (0.101)_2$.

Ejemplo: Convertir 0.1 a binario.

0.1		
2*0.1 = 0.2	[0.2] = 0	↓
2*0.2 = 0.4	[0.4] = 0	
2*0.4 = 0.8	[0.8] = 0	
2*0.8 = 1.6	[1.6] = 1	
2*0.6 = 1.2	[1.2] = 1	
2*0.2 = 0.4	[0.4] = 0	
2*0.4 = 0.8	[0.8] = 0	
2*0.8 = 1.6	[1.6] = 1	
2*0.6 = 1.2	[1.2] = 1	
2*0.2 = 0.4	[0.4] = 0	...

En este caso la representación es infinita, $(0.1)_2 = (0.0001100110\ldots)_2$.

Finalmente, veamos como podemos convertir un número binario en su representación hexadecimal, sin necesidad de pasar por su representación decimal. Puesto que $2^4 = 16$, la representación hexadecimal puede ser obtenida de la representación binaria agrupando los bits de la misma en grupos de cuatro a partir del punto binario tanto a la derecha e izquierda del mismo, agregando tantos ceros como sea necesario. Luego, cada grupo de cuatro bits es trasladado directamente en un dígito hexadecimal de acuerdo a la siguiente tabla.

Binario	Hexadecimal	Binario	Hexadecimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Ejemplo: Convertir el número binario 11101.011 a hexadecimal. Simplemente agrupamos en grupo de cuatro bits (agregando los ceros necesarios) y hacemos la conversión.

$$11101.011 = 0001\ 1101\ .\ 0110 = 1D.6$$

Un procedimiento semejante permite obtener directamente la representación octal a partir de la representación binaria. Puesto que $2^3 = 8$, el agrupamiento de los bits es, en este caso, en grupos de a tres.

Representación de enteros: números de punto fijo

Supongamos una longitud de palabra N, es decir, se dispone de N posiciones para guardar un número real respecto de cierta base β. Una manera de disponer tales posiciones consiste en utilizar la primera para indicar el signo, las $N - k - 1$ posiciones siguientes para los dígitos de la parte entera y las k posiciones restantes para la parte fraccionaria. De esta forma, la secuencia de N dígitos

$$a_{N-1}\ \underbrace{a_{N-2}\, a_{N-3} \cdots a_k}_{N-k-1}\ \underbrace{a_{k-1} \cdots a_0}_{k},$$

donde $a_{N-1} = s$ (0 ó 1), corresponde al número

$$(-1)^s(a_{N-2}\, a_{N-3}\cdots a_k\,.\,a_{k-1}\cdots a_0)_\beta = (-1)^s\beta^{-k}\sum_{j=0}^{N-2} a_j\beta^j.$$

La representación de los números en esta forma se conoce como *representación de punto fijo*. El nombre hace referencia al hecho de que al fijar en k el número de dígitos de la parte fraccionaria, el punto en la base β resulta fijo a la derecha del dígito a_k.

Ejemplo. Supongamos $\beta = 10$, $N = 11$ y $k = 6$. Entonces, disponemos de $k = 6$ dígitos para la parte fraccionaria y $N - k - 1 = 4$ dígitos para la parte entera. Por ejemplo,

-30.412 :	1	0030	421000
0.0437 :	0	0000	000437

Para una longitud palabra N y con k dígitos fijos para la parte fraccionaria, el rango de valores de los números reales que pueden representarse se encuentra dentro del intervalo $[-\beta^{N-k}, \beta^{N-k}]$. Este rango es muy restrictivo: los números muy grandes y las fracciones muy pequeñas no pueden representarse (a menos que una longitud de palabra N muy grande sea utilizada). Por este motivo la representación de punto fijo *no* es utilizada para implementar la representación de números reales. Sin embargo, con la convención $k = 0$, la representación de punto fijo resulta útil para representar números enteros. En lo que resta de esta sección discutiremos como se implementa en una computadora donde la base $\beta = 2$.

Considerando la base $\beta = 2$, dada una palabra de N bits existen 2^N combinaciones distintas que pueden generarse. Con ellas queremos representar los enteros positivos, negativos y el cero. Existen varias maneras de proceder, pero todas ellas tratan al bit *más significativo* (el más de la izquierda) de la palabra como un bit de signo. Si dicho bit es 0, el entero es positivo, y si es 1, es negativo. Ahora, la representación más sencilla que puede considerarse es la *representación signo-magnitud*. En una palabra de N bits, mientras que el bit más significativo indica el signo del entero, los restantes $N - 1$ bits a la derecha representan la *magnitud* del entero. Así, la secuencia de N bits

$$a_{N-1}\, a_{N-2}\cdots a_1\, a_0,$$

corresponde, en la representación signo-magnitud, al número entero

$$(-1)^s\sum_{j=0}^{N-2} a_j 2^j,$$

con $s = 0$ si $a_{N-1} = 0$, ó $s = 1$ si $a_{N-1} = 1$. Esta representación, aunque directa, posee varias limitaciones. Una de ellas es que las operaciones aritméticas de suma y resta requieren separar el bit que representa el signo de los restantes para llevar a cabo la operación en cuestión. Otra limitación es que hay dos representaciones del número cero 0, a saber,

$$000\cdots00, \qquad 100\cdots00.$$

Como indicamos, el número de combinaciones diferentes de una palabra de N bits es 2^N, un número par. Puesto que tenemos dos representaciones del cero, en la representación signo-magnitud se representan la misma cantidad de enteros positivos que negativos. El rango de enteros que puede representarse comprende entonces al intervalo $[-(2^{N-1}-1), 2^{N-1}-1]$. En efecto, el entero positivo más grande tiene un bit de signo 0 y todos los bits de magnitud son 1, y corresponde pues al entero

$$\sum_{j=0}^{N-2} 2^j = \frac{2^{N-1}-1}{2-1} = 2^{N-1}-1.$$

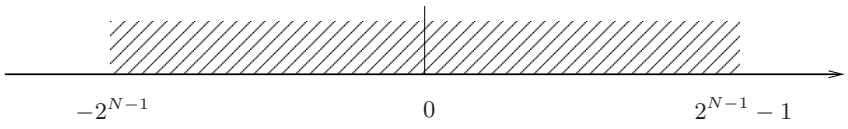


Figura 1. Enteros representables en complemento a dos para una longitud de palabra de N bits.

Aquí hemos uso de la fórmula para la suma (finita) de una progresión geométrica*. Por su parte, el entero negativo de mayor magnitud tiene 1 por bit de signo y todos los bits de magnitud igual a 1, y es, por tanto,

$$-\sum_{j=0}^{N-2} 2^j = -\frac{2^{N-1}-1}{2-1} = -\,(2^{N-1}-1).$$

La implementación de las operaciones aritméticas de suma y resta de enteros en una computadora resulta mucho más simple y eficiente si el bit de signo puede ser tratado de la misma forma que los bits de magnitud. La representación que permite ésto es la denominada *representación complemento a dos*. Puesto que en esta representación la implementación de dichas operaciones resulta mucho más simple, *la representación complemento a dos es utilizada casi universalmente como representación de los números enteros en la computadora*. Al igual que la representación signo-magnitud, la representación complemento a dos utiliza el bit más significativo como bit de signo (lo cual facilita la comprobación del signo de un entero), pero difiere en la forma de interpretar los bits restantes. Específicamente, para una longitud de palabra de N bits, una secuencia de N dígitos binarios

$$a_{N-1}\, a_{N-2} \cdots a_1\, a_0,$$

corresponde, en la representación complemento a dos, al entero

$$-\,a_{N-1}\, 2^{N-1} + \sum_{j=0}^{N-2} a_j 2^j.$$

Puesto que para los enteros positivos $a_{N-1} = 0$, el término $-\,a_{N-1}\, 2^{N-1} = 0$. Así la representación complemento a dos y signo-magnitud coinciden para los enteros positivos. Sin embargo, para los enteros negativos ($a_{N-1} = 1$) el bit más significativo es ponderado con un factor de peso -2^{N-1}. En esta representación el 0 se considera positivo y es claro que hay una *única* representación del mismo, la cual corresponde a un bit de signo 0 al igual que los restantes bits. Puesto que, como ya indicamos, para una longitud de palabra de N bits existe un número par de combinaciones posibles distintas, si hay una única representación del 0 entonces existe un número desigual de números enteros positivos que de números enteros negativos representables. El rango de los enteros positivos en esta representación va desde 0 (todos los bits de magnitud son 0) hasta $2^{N-1}-1$ (todos los bits de magnitud 1, como en el caso de la representación signo-magnitud). Ahora, para un número entero negativo, el bit de signo, a_{N-1}, es 1. Los $N-1$ bits restantes pueden tomar cualquiera de las 2^{N-1} combinaciones diferentes. Por lo tanto, el rango de los enteros negativos que pueden representarse va desde -1 hasta -2^{N-1}. Es claro de la fórmula que define a la representación que -1 tiene una representación complemento a dos consistente de un bit de signo igual a 1 al igual que los restos de los bits, mientras que -2^{N-1}, el menor entero representable, tiene un representación complemento a dos consistente en el bit de signo igual a 1 y el resto de los bits igual a 0. En definitiva, en la representación complemento a dos, el rango de enteros que pueden representarse comprende el intervalo $[-2^{N-1}, 2^{N-1}-1]$. Esto implica, en particular, que *no existe una representación complemento a dos para el opuesto del menor entero representable*. La figura 1 ilustra el rango de los enteros representables complemento a dos sobre la recta real, mientras que la tabla 1 compara, para una longitud de palabra de $N = 4$ bits, las representaciones en signo-magnitud y en complemento a dos.

*Si $r \neq 1$, $\sum_{j=0}^{n-1} r^j = \frac{r^n-1}{r-1}$

Representación decimal	Representación signo-magnitud	Representación complemento a dos
$+7$	0111	0111
$+6$	0110	0110
$+5$	0101	0101
$+4$	0100	0100
$+3$	0011	0011
$+2$	0010	0010
$+1$	0001	0001
$+0$	0000	0000
-0	1000	—
-1	1001	1111
-2	1010	1110
-3	1011	1101
-4	1100	1100
-5	1101	1011
-6	1110	1010
-7	1111	1001
-8	—	1000

Tabla 1. Representaciones alternativas de los enteros de 4 bits.

Las operaciones aritméticas de suma, resta, multiplicación y división (entera) entre dos enteros pueden ser implementadas en la computadora de manera tal que su resultado siempre es *exacto*, es decir, sin error de redondeo. Sin embargo, puede ocurrir que el resultado de tales operaciones esté fuera del rango de la representación complemento a dos. Esta situación es conocida como *desbordamiento entero* (*integer overflow*, en inglés). En la mayoría de las implementaciones está condición no se considera un error que detendrá al programa, sino que simplemente se almacenan los dígitos binarios posibles. El entero correspondiente a la representación complemento a dos resultante del overflow *no* es el resultado que uno esperaría. Por ejemplo, considerando una longitud de palabra de $N = 4$ bits, al intentar sumar 1 al mayor entero representable se produce una condición de overflow y obtenemos el menor entero representable (¡un número negativo!):

$$\begin{array}{rrcl} & 7 & 0111 & \\ + & \underline{1} & \underline{0001} & \\ & 8 & 1000 & = \ -8. \end{array}$$

El ejemplo anterior nos permite explicar el desconcertante resultado del primer programa presentado en la introducción. *En las computadoras personales los números enteros son implementados como enteros binarios con signo en complemento a dos con una longitud de palabra de $N = 32$ bits.* En consecuencia, una variable entera en Fortran, declarada con la sentencia `integer` podrá representar exactamente a un número entero si éste se encuentra dentro del intervalo $[-2^{31}, 2^{31} - 1]$. Así, al intentar sumar una unidad al mayor entero representable, $2^{31} - 1$, se produce el desbordamiento y el entero almacenado corresponde, en complemento a dos, al menor entero representable, el cual es -2^{31}. Claramente, el desbordamiento entero puede originar problemas en aquellas secciones del programa que asuman que los enteros presentes son siempre positivos.

Aritmética con enteros complemento a dos. En la representación complemento a dos, al tratar por igual todos los bits, la suma procede como se ilustra a continuación en el caso particular de una longitud de palabra de $N = 4$ bits. Como vemos, las operaciones son correctas, obteniéndose el correspondiente número positivo o negativo en forma de complemento a dos. Obsérvese que si hay un bit de *acarreo* más

allá del final de la palabra (el dígito sombreado), éste se ignora.

$$\begin{array}{r} 1001 \\ +\ 0101 \\ \hline 1110 \end{array} = -2 \qquad \begin{array}{r} 1100 \\ +\ 0100 \\ \hline 1\,0000 \end{array} = 0 \qquad \begin{array}{r} 0011 \\ +\ 0100 \\ \hline 1111 \end{array} = 7 \qquad \begin{array}{r} 1100 \\ +\ 1111 \\ \hline 1\,1011 \end{array} = -5$$

(a) $(-7)+(+5)$ (b) $(-4)+(+4)$ (c) $(+3)+(+4)$ (d) $(-4)+(-1)$

Por otra parte, los siguientes ejemplos muestran situaciones de desbordamiento entero.

$$\begin{array}{r} 0101 \\ +\ 0100 \\ \hline 1001 \end{array} = -7 \qquad \begin{array}{r} 1001 \\ +\ 1010 \\ \hline 1\,0011 \end{array} = 3$$

(e) $(+5)+(+4)$ (f) $(-7)+(-6)$

La operación aritmética de sustracción entre dos números puede considerarse como la suma del primer número y el opuesto del segundo. En la representación de signo-magnitud, el opuesto de un entero se obtiene sencillamente invirtiendo el bit del signo. En la representación complemento a dos, el opuesto de un entero se obtiene siguiendo los siguientes pasos:

1. Obtener el *complemento booleano* de cada bit (incluyendo el bit del signo). Es decir, cambiar cada 1 por 0 y cada 0 por 1. El patrón de bits obtenido se conoce como *complemento a 1* de la secuencia original.
2. Sumar 1 al binario obtenido (tratando al bit de signo como un bit ordinario e ignorando cualquier acarreo de la posición del bit más significativo que exceda el límite de la palabra).

El patrón de bits obtenido, interpretado ahora como un entero complemento a dos, es el opuesto del entero representado por la secuencia original. Este proceso de dos etapas para obtener el opuesto se denomina *transformación complemento a dos*. La validez de este procedimiento puede demostrarse como sigue. Consideremos una secuencia de N dígitos binarios

$$a_{N-1}\, a_{N-2} \cdots a_1\, a_0,$$

que corresponde, en la representación complemento a dos, al entero A de valor

$$A = -\, a_{N-1}\, 2^{N-1} + \sum_{j=0}^{N-2} a_j 2^j.$$

Ahora se construye el complemento a uno

$$\overline{a_{N-1}}\,\overline{a_{N-2}} \cdots \overline{a_1}\,\overline{a_0},$$

y, tratando todos los bits por igual, se le suma 1. Finalmente, se interpreta la secuencia de bits resultante como un entero complemento a dos, tal que su valor es

$$B = -\,\overline{a_{N-1}}\, 2^{N-1} + 1 + \sum_{j=0}^{N-2} \overline{a_j} 2^j.$$

Tenemos que mostrar que $A = -B$, o, equivalentemente, que $A + B = 0$. Esto se comprueba fácilmente,

$$\begin{aligned} A + B &= -\left(a_{N-1} + \overline{a_{N-1}}\right) 2^{N-1} + 1 + \left(\sum_{j=0}^{N-2} (a_j + \overline{a_j}) \right) \\ &= -2^{N-1} + 1 \sum_{j=0}^{N-2} 2^j \\ &= -2^{N-1} + 1 + 2^{N-1} - 1 \\ &= 0. \end{aligned}$$

Por ejemplo, para una longitud de palabra de $N = 4$ bits,

$$\begin{array}{rcl} +5 & = & 0101 \\ \text{complemento a 1} & = & 1010 \\ & & + \ \underline{0001} \\ & & 1011 \ = \ -5. \end{array}$$

Como era de esperarse, el opuesto del opuesto es el propio número,

$$\begin{array}{rcl} -5 & = & 1011 \\ \text{complemento a 1} & = & 0100 \\ & & + \ \underline{0001} \\ & & 0101 \ = \ +5. \end{array}$$

Hay dos situaciones especiales a considerar. En primer lugar, para el 0, en una representación con 4 bits,

$$\begin{array}{rcl} 0 & = & 0000 \\ \text{complemento a 1} & = & 1111 \\ & & + \ \underline{0001} \\ & & 1\,0000 \ = \ 0. \end{array}$$

El bit de acarreo de la posición más significativa debe ser ignorado. De esta forma el opuesto de 0 es 0, como debe ser. El segundo caso especial involucra el menor entero representable en complemento a dos. Para una longitud de palabra de 4 dígitos, éste es -8. Su opuesto, formado con la regla anterior es

$$\begin{array}{rcl} -8 & = & 1000 \\ \text{complemento a 1} & = & 0111 \\ & & + \ \underline{0001} \\ & & 1000 \ = \ -8. \end{array}$$

Esta anomalía debería ser evitada, puesto que, como hemos indicado, no existe una representación complemento a dos del opuesto del menor entero representable. Sin embargo, en la mayoría de las implementaciones, al calcular el opuesto del menor entero representable *no* se genera una situación de error sino que se obtiene el mismo número. Esta situación puede resultar confusa, pero es así.

Conociendo el opuesto de un número, la operación aritmética de resta se trata entonces como sigue: para substraer un número (el substraendo) de otro (minuendo), se forma el complemento a dos del substraendo y se le suma el minuendo. Los siguientes ejemplos, ilustran la situación para una longitud de palabra de $N = 4$ bits.

$$\begin{array}{rcl} & 0010 & \\ + & \underline{1001} & \\ & 1011 \ = \ -5 & \end{array}$$

(a) $(+2) - (+7)$

$$\begin{array}{rcl} & 0101 & \\ + & \underline{1110} & \\ & 1\,0011 \ = \ 3 & \end{array}$$

(b) $(+5) - (+2)$

$$\begin{array}{rcl} & 1101 & \\ + & \underline{0010} & \\ & 0111 \ = \ 7 & \end{array}$$

(c) $(+5) - (-2)$

$$\begin{array}{rcl} & 1011 & \\ + & \underline{1110} & \\ & 1\,1001 \ = \ -7 & \end{array}$$

(d) $(-5) - (-2)$

En tanto que los siguientes ejemplos ilustran situaciones de desbordamiento.

$$\begin{array}{rcl} & 0111 & \\ + & \underline{0111} & \\ & 1110 \ = \ -2 & \end{array}$$

(e) $(+7) - (-7)$

$$\begin{array}{rcl} & 1010 & \\ + & \underline{1100} & \\ & 1\,0110 \ = \ 6 & \end{array}$$

(f) $(-6) - (+4)$

Las operaciones de multiplicación y división de enteros son más complejas de implementar y no serán tratadas aquí.

Representación de números reales: números de punto flotante

La representación del sistema de números reales en una computadora basa su idea en la conocida *notación científica*. La notación científica nos permite representar números reales sobre un

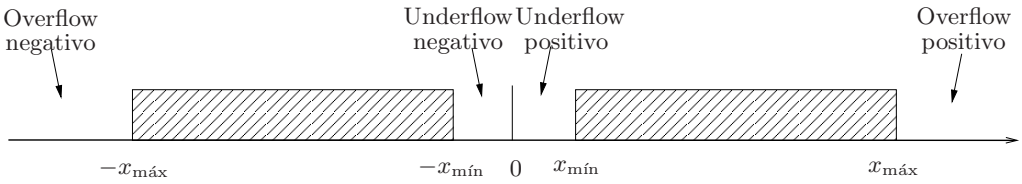


Figura 2. Números de punto flotante $\mathbb{F}(\beta, t, L, U)$.

amplio rango de valores con sólo unos pocos dígitos. Así 976 000 000 000 000 se representa como 9.76×10^{14} y 0.0000000000000976 como 9.76×10^{-14}. En esta notación el punto decimal se mueve dinámicamente a una posición conveniente y se utiliza el exponente de 10 para registrar la posición del punto decimal. En particular, todo número real *no nulo* puede ser escrito en forma única en la notación científica *normalizada*

$$(-1)^s\, 0.a_1a_2a_3\cdots a_ta_{t+1}a_{t+2}\cdots \times 10^e,$$

siendo el dígito $a_1 \neq 0$. De manera análoga, todo número real no nulo puede representarse en forma única, respecto la base β, en la *forma de punto flotante normalizada*:

$$(-1)^s\, 0.a_1a_2a_3\cdots a_ta_{t+1}a_{t+2}\cdots \times \beta^e,$$

donde los "dígitos" a_i respecto de la base β son enteros positivos tales que $1 \leq a_1 \leq \beta - 1$, $0 \leq a_i \leq \beta - 1$ para $i = 1, 2, \ldots$ y constituyen la parte fraccional o *mantisa* del número, en tanto que e, el cual es un número entero llamado el *exponente*, indica la posición del punto correspondiente a la base β. Si m es la fracción *decimal* correspondiente a $(0.a_1a_2a_3\cdots)_\beta$ entonces el número representado corresponde al número decimal

$$(-1)^s\, m \cdot \beta^e, \qquad \text{siendo } \beta^{-1} \leq m < 1.$$

Ahora bien, en todo dispositivo de cálculo, ya sea una computadora o una calculadora, el número de dígitos posibles para representar la mantisa es *finito*, digamos t dígitos en la base β, y el exponente puede variar sólo dentro de un rango finito $L \leq e \leq U$ (con $L < 0$ y $U > 0$). Esto implica que sólo un conjunto *finito* de números reales pueden ser representados, los cuales tienen la forma

$$(-1)^s\, 0.a_1a_2a_3\cdots a_t \times \beta^e.$$

Tales números se denominan *números de punto flotante* con t dígitos (o de precisión t) en la base β y rango (L, U). Denotaremos a dicho conjunto de números por $\mathbb{F}(\beta, t, L, U)$.

Un sistema de números de punto flotante $\mathbb{F}(\beta, t, L, U)$ es, pues, discreto y finito. El número de elementos del conjunto, esto es, la cantidad de números de puntos flotantes de $\mathbb{F}$, es

$$2(\beta - 1)\beta^{t-1}(U - L + 1),$$

dado que existen dos posibles elecciones del signo, $\beta - 1$ elecciones posibles para el dígito principal de la mantisa, β elecciones para cada uno de los restantes dígitos de la mantisa, y $U - L + 1$ posibles valores para el exponente. Debido a la normalización el cero *no* puede ser representado como un número de punto flotante y por lo tanto está excluido del conjunto $\mathbb{F}$. Por otra parte, es claro que si $x \in \mathbb{F}$ entonces su opuesto $-x \in \mathbb{F}$. Más aún, el conjunto $\mathbb{F}$ está acotado tanto superior como inferiormente:

$$x_{\text{mín}} = \beta^{L-1} \leq |x| \leq x_{\text{máx}} = \beta^U(1 - \beta^{-t}),$$

donde $x_{\text{mín}}$ y $x_{\text{máx}}$ son el menor y mayor número de punto flotante positivo representable, respectivamente.

De las consideraciones anteriores se sigue, entonces, que en la recta de los números reales hay cinco regiones excluidas para los números de $\mathbb{F}$, tal como se ilustra en la figura 2,

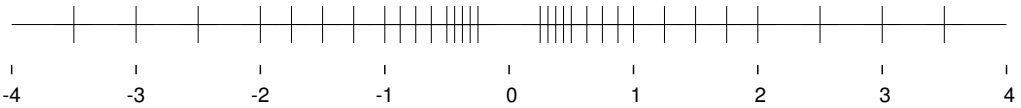


Figura 3. Números de punto flotante del conjunto $\mathbb{F}(2, 3, -1, 2)$.

a) Los números negativos menores que $-x_{\text{máx}}$, región denominada *desbordamiento* (*overflow*) *negativo*.

b) Los números negativos mayores que $-x_{\text{mín}}$, denominada *desbordamiento a cero* (*underflow*) *negativo*.

c) El cero.

d) Los números positivos menores que $x_{\text{mín}}$, denominada *desbordamiento a cero* (*underflow*) *positivo*.

e) Los números positivos mayores que $x_{\text{máx}}$, denominada *desbordamiento* (*overflow*) *positivo*.

Nótese que los números de punto flotante no están uniformemente distribuidos sobre la recta real, sino que están más próximos cerca del origen y más separados a medida que nos alejamos de él. Con más precisión: dentro de un intervalo fijo de la forma $[\beta^e, \beta^{e+1}]$ los números de punto flotante presentes están igualmente espaciados con una separación igual a β^{e+1-t}. Conforme e se incrementa, el espaciado entre los mismos crece también. Una medida de este espaciamiento es dado por el llamado *epsilon de la máquina*:

$$\epsilon_M = \beta^{1-t},$$

el cual representa la distancia entre el número 1 y el número de punto flotante siguiente más próximo. Esto es, ϵ_M es el número de punto flotante más pequeño para el cual $1 + \epsilon_M > 1$.

Ejemplo. Los números de punto flotante positivos del conjunto $\mathbb{F}(2, 3, -1, 2)$ son:

$$(0.100)_2 \times 2^{-1} = \frac{1}{4}, \quad (0.101)_2 \times 2^{-1} = \frac{5}{16}, \quad (0.110)_2 \times 2^{-1} = \frac{3}{8}, \quad (0.111)_2 \times 2^{-1} = \frac{7}{16},$$

$$(0.100)_2 \times 2^{0} = \frac{1}{2}, \quad (0.101)_2 \times 2^{0} = \frac{5}{8}, \quad (0.110)_2 \times 2^{0} = \frac{3}{4}, \quad (0.111)_2 \times 2^{0} = \frac{7}{8},$$

$$(0.100)_2 \times 2^{1} = 1 \quad (0.101)_2 \times 2^{1} = \frac{5}{4}, \quad (0.110)_2 \times 2^{1} = \frac{3}{2}, \quad (0.111)_2 \times 2^{1} = \frac{7}{4},$$

$$(0.100)_2 \times 2^{2} = 2, \quad (0.101)_2 \times 2^{2} = \frac{5}{2}, \quad (0.110)_2 \times 2^{2} = 3, \quad (0.111)_2 \times 2^{2} = \frac{7}{2}.$$

Junto con los respectivos opuestos, tenemos un total de $2(\beta-1)\beta^{t-1}(U-L+1) = 2(2-1)2^{3-1}(2+1+1) = 32$ de números de punto flotante en este sistema. Aquí $x_{\text{mín}} = \beta^{L-1} = 2^{-2} = 1/4$ y $x_{\text{máx}} = \beta^U(1 - \beta^{-t}) = 2^2(1 - 2^{-3}) = 7/2$, en tanto que $\epsilon_M = \beta^{1-t} = 2^{-2} = 1/4$. La figura 3 muestra nuestro sistema sobre la recta real. Aunque éste es un sistema de punto flotante muy pequeño *todos* lucen como nuestro ejemplo a un grado de magnificación suficiente.

El hecho de que en una computadora sólo un subconjunto $\mathbb{F}$ de los números reales es representable implica que, dado un número real x, éste debe ser aproximado por un número de punto flotante de $\mathbb{F}$ al que designaremos por $fl(x)$. La manera usual de proceder consiste en considerar el *redondeo al más próximo*, ésto es, $fl(x)$ es el número de punto flotante más próximo a x. Tal número de punto flotante resulta del redondeando a t dígitos de la mantisa correspondiente a la representación de punto flotante normalizada (infinita) de x. Esto es, siendo

$$x = (-1)^s \; 0.a_1a_2 \ldots a_t a_{t+1} a_{t+2} \cdots \times \beta^e,$$

$fl(x)$ está dado por

$$fl(x) = (-1)^s \; 0.a_1 a_2 \ldots \widetilde{a}_t \times \beta^e, \qquad \widetilde{a}_t = \begin{cases} a_t & \text{si } a_{t+1} < \beta/2 \\ a_t + 1 & \text{si } a_{t+1} \geq \beta/2. \end{cases}$$

siempre que el exponente e esté dentro del rango $-L \leq e \leq U^*$. Claramente, $fl(x) = x$ si $x \in \mathbb{F}$, y un mismo número de punto flotante puede representar a muchos números reales distintos.

El error que resulta de aproximar un número real por su forma de punto flotante se denomina *error de redondeo*. Una estimación del mismo está dado en el siguiente resultado: *Todo número real x dentro del rango de los números de punto flotante puede ser representado con un error relativo que no excede la unidad de redondeo* $\mathbf{u}$:

$$\frac{|x - fl(x)|}{|x|} \leq \mathbf{u} \equiv \frac{1}{2}\,\epsilon_M = \frac{1}{2}\,\beta^{1-t}.$$

Se sigue de este resultado que existe un número real δ, que depende de x, tal que

$$fl(x) = x(1 + \delta), \qquad \text{siendo } |\delta| \leq \mathbf{u}.$$

Por otra parte, debido a que los números de punto flotante no están igualmente espaciados, el error absoluto cometido en la representación no es uniforme, sino que está dado por

$$|x - fl(x)| \leq \frac{1}{2}\beta^{-t+e}.$$

Además de dar una representación inexacta de los números, la aritmética realizada en la computadora no es exacta. Aún si x e y son números de punto flotante, el resultado de una operación aritmética sobre ellos no necesariamente es un número de punto flotante. En consecuencia, debe definirse una aritmética en $\mathbb{F}$ que sea lo más semejante posible a la aritmética de los números reales. Así, si $\circ$ denota una operación aritmética (suma, resta, multiplicación o división) la correspondiente operación de punto flotante, denotada por $\circledcirc$, entre dos números de punto flotante x e y es definida como:

$$x \circledcirc y = fl(x \circ y), \quad x, y \in \mathbb{F}.$$

en tanto no se produzca desbordamiento. Esta aritmética consiste, pues, en efectuar la operación exacta en las representaciones de punto flotante y luego redondear el resultado a su representación de punto flotante[**]. Se sigue entonces que en una operación aritmética con números de punto flotante, el error relativo cometido no excede a la unidad de redondeo, y por lo tanto, existe un número real δ tal que

$$x \circledcirc y = (x \circ y)(1 + \delta), \qquad \text{siendo } |\delta| \leq \mathbf{u}.$$

Un punto importante a notar es que *no todas las leyes de la aritmética real son preservadas al considerar la aritmética de punto flotante*. En particular, la suma y multiplicación de punto flotante son conmutativas, pero *no* son asociativas. La tabla 2 muestra que propiedades de los números reales son retenidas, y cuales no, en el sistema de números de punto flotante. Algunas propiedades que son consecuencia de los axiomas básicos son aún válidas, por ejemplo:

$$x \oplus y = 0 \text{ equivale a } x = 0 \text{ ó } y = 0,$$
$$x \leq y \text{ y } z > 0 \text{ implica que } x \oplus z \leq y \oplus z,$$

mientras que $x = y$ es equivalente a $(x - y) = 0$ sólo si se utilizan números de punto flotante *denormalizados*, los cuales serán discutidos más adelante.

[*] Aquí estamos suponiendo que la base β es par, la cual es válido para las situaciones de interés $\beta = 2, 6, 8, 10$.

[**] La implementación efectiva de estas operaciones en una computadora no procede exactamente de esta forma pero el resultado final se comporta como hemos indicado. En particular la operación de resta requiere de la presencia de *dígitos de guarda*, situación que discutiremos más adelante.

Propiedades topológicas		
Conectividad	No	Todos los puntos están aislados
Completitud	No	No todas las sucesiones convergen
Axiomas de cuerpo		
Cerrado para la suma	No	Puede haber desbordamiento
Asociatividad de la suma	No	$(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$
Conmutatividad de la suma	Si	$x \oplus y = y \oplus x$
Unicidad del elemento neutro	Si	$x \oplus 0 = x$
Unicidad del inverso aditivo	Si	$x \oplus -x = 0$
Cerrado para la multiplicación	No	Puede haber desbordamiento
Asociatividad de la multiplicación	No	$(x \odot y) \odot z \neq x \odot (y \odot z)$
Conmutatividad de la multiplicación	Si	$x \odot y = y \odot x$
Unicidad del elemento unidad	Si	$x \odot 1 = x$
Unicidad del inverso multiplicativo	Si	$x \odot (1/x) = x$
Distributividad de la multiplicación en la suma	No	$x \odot (y \oplus z) \neq x \odot y \oplus x \odot z$
Axiomas de orden		
Transitividad	Si	$x \geq y,\, y \geq z \Rightarrow x \geq z$

Tabla 2. Propiedades de la aritmética de punto flotante.

Estándar del IEEE para la representación binaria en punto flotante. La precisión t de un sistema de números de punto flotante en una computadora estará limitada por la longitud de palabra N disponible para representar un número. Con el fin de evitar la proliferación de diversos sistemas de puntos flotantes incompatibles entre sí, a fines de la década de 1980 se desarrolló la norma o estándar IEEE754/IEC559[*], la cual es implementada en todas las computadoras personales actuales y aplicado en casi todos los otros sistemas. Esta norma define dos formatos de punto flotante con base $\beta = 2$:

a) *precisión simple*: $\mathbb{F}(2, 24, -125, 128)$ implementado en una longitud de palabra $N = 32$ bits,

b) *precisión doble*: $\mathbb{F}(2, 53, -1021, 1024)$ implementado en una longitud de palabra $N = 64$ bits.

Para el formato de precisión simple tenemos que

$$\begin{aligned} x_{\text{mín}} &= 2^{-126} \approx 10^{-38}, \\ x_{\text{máx}} &= 2^{128}(1 - 2^{-24}) \approx 10^{38}, \\ \epsilon_M &= 2^{-23} \approx 10^{-7}, \end{aligned}$$

siendo la unidad de redondeo

$$\mathbf{u} = \frac{1}{2}\epsilon_M = 2^{-24} \approx 6 \cdot 10^{-8},$$

lo cual implica unos 7 dígitos *decimales* significativos.

Para el formato de precisión doble, tenemos,

$$\begin{aligned} x_{\text{mín}} &= 2^{-1022} \approx 10^{-308}, \\ x_{\text{máx}} &= 2^{1024}(1 - 2^{-53}) \approx 10^{308}, \\ \epsilon_M &= 2^{-52} \approx 10^{-16}, \end{aligned}$$

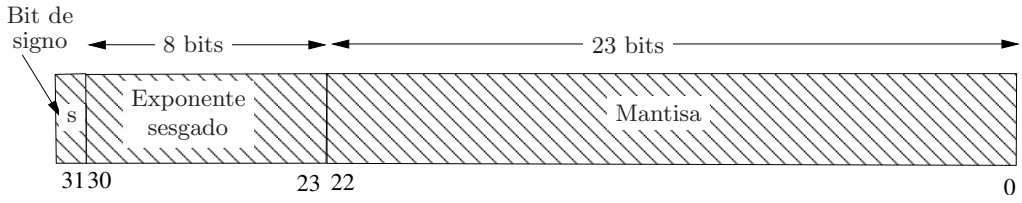
[*]IEEE = Institute of Electrical and Electronics Engineeers, IEC = International Electronical Commission.

con una unidad de redondeo

$$\mathbf{u} = \frac{1}{2}\epsilon_M = 2^{-53} \approx 10^{-16},$$

lo cual implica unos 16 dígitos *decimales* significativos.

Anatomía del formato de precisión simple. En el formato de precisión simple, el conjunto de números de punto flotante tiene por parámetros $\beta = 2$, $t = 24$, $L = -125$, $U = 128$. Los $N = 32$ bits disponibles en la palabra, numerados de 0 a 31 de derecha a izquierda, son utilizados como sigue:



- El bit más significativo (el bit más a la izquierda de la palabra, el 31) es utilizado como bit de signo, esto es, su valor es 0 si el número es positivo o 1 si es el número es negativo.
- Los siguientes 8 bits (bits 30 a 23) son utilizados para almacenar la representación binaria del exponente que varía en el rango $L = -125 \leq e \leq U = 128$. La representación utilizada para el exponente se conoce como "sesgada", ya que se introduce un nuevo exponente sumando 126 al original: $E = e + 126$. El exponente sesgado varía entonces en el rango $1 \leq E \leq 254$ que puede ser representado por un binario entero de 8 bits. Más aún, podemos incluir los valores del exponente para $L - 1 = -126$ ($E = 0$) y $U + 1 = 129$ ($E = 255$), ya que todos los enteros en el rango $0 \leq E \leq 255$ pueden ser representados como un binario entero sin signo de 8 bits. En efecto, con 8 bits tenemos $2^8 = 256$ combinaciones distintas, una para cada uno de los 256 números enteros del intervalo $[0, 255]$. El límite inferior, el 0, corresponde a todos los bits igual a 0, mientras que el límite superior, el 255, corresponde a todos los dígitos igual a 1 ($\sum_{i=0}^{7} 2^i = \frac{2^8-1}{2-1} = 255$). Estos dos valores del exponente *no* representan números de punto flotante del sistema, pero serán utilizados para almacenar números especiales, como veremos más adelante.
- Los restantes 23 bits (bits 22 a 0) de la palabra son utilizados para almacenar la representación binaria de la mantisa. En principio, esto parecería implicar que sólo pueden almacenarse $t = 23$ dígitos binarios de la mantisa y no 24. Sin embargo, para la base $\beta = 2$, el primer dígito de la mantisa en la representación normalizada siempre es igual a 1 y por lo tanto no necesita ser almacenado (tal bit se denomina *bit implícito – hidden bit*, en inglés). Así pues, un campo de 23 bits permite albergar una mantisa efectiva de 24 bits.

Como ejemplo, consideremos la representación de formato simple del número cuya representación decimal es -118.625. Su representación binaria es (¡verifíquelo!)

$$(-1)^1(1110110.101)_2,$$

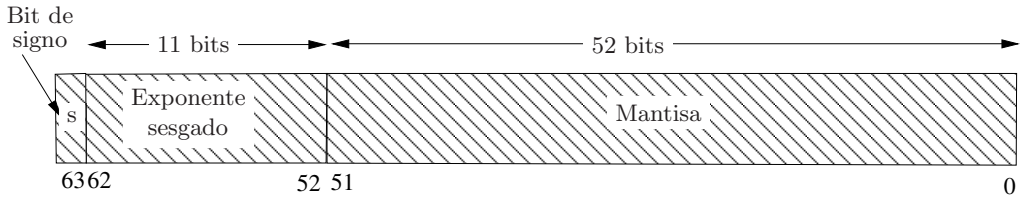
con lo que, su representación binaria normalizada es

$$(-1)^1\, 0.1110110101 \times 2^7.$$

El bit de signo es 1, y el exponente sesgado es $E = 7 + 126 = 133$ cuya representación binaria de 8 bits es 10000101. El bit implícito de la mantisa no es almacenado, por lo que tenemos 9 dígitos binarios provenientes de la representación normalizada, el resto de los $23 - 9 = 14$ dígitos binarios son ceros. Así pues, la representación de formato simple del número propuesto es:

1	1	0	0	0	0	1	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Anatomía del formato de precisión doble. En el formato de precisión doble, el conjunto de números de punto flotante tiene por parámetros $\beta = 2$, $t = 53$, $L = -1021$, $U = 1024$. Los $N = 64$ bits disponibles en la palabra, numerados de 0 a 63 de derecha a izquierda, son utilizados como sigue:



- El bit más significativo (el bit más a la izquierda de la palabra, el 63) es utilizado como bit de signo, esto es, su valor es 0 si el número es positivo o 1 si es el número es negativo.
- Los siguientes 11 bits (bits 62 a 52) son utilizados para almacenar la representación binaria del exponente que varía en el rango $L = -1021 \leq e \leq U = 1024$. La representación sesgada corresponde al exponente: $E = e + 1022$. El exponente sesgado varía entonces en el rango $1 \leq E \leq 2046$ que puede ser representado por un binario entero de 11 bits. Incluyendo los valores del exponente para $L - 1 = -1022$ ($E = 0$) y $U + 1 = 1025$ ($E = 2047$), todos los enteros en el rango $0 \leq E \leq 2047$ pueden ser representados como un binario entero sin signo de 11 bits ($2^{11} = 2048$)
- Los restantes 52 bits (bits 51 a 0) de la palabra son utilizados para almacenar la representación binaria de la mantisa con el bit implícito no almacenado. De este modo el campo de 52 bits se corresponde con una mantisa efectiva de 53 dígitos binarios.

Números especiales. La condición de normalización sobre la mantisa de los números de punto flotante impide la representación del cero, por lo tanto debe disponerse de una representación separada del mismo. Por otra parte, en la aritmética de punto flotante pueden presentarse las tres siguientes condiciones excepcionales:

a) Una operación puede conducir a un resultado fuera del rango representable, ya sea porque $|x| > x_{\text{máx}}$ (*overflow*) o porque $|x| < x_{\text{mín}}$ (*underflow*),

b) el cálculo puede ser una operación matemática indefinida, tal como la división por cero,

c) el cálculo corresponde a una operación matemática ilegal, por ejemplo $0/0$ ó $\sqrt{-1}$.

Antes de la implementación de la norma IEEE754, frente a tales situaciones excepcionales las computadoras abortaban el cálculo y detenían el programa. Por el contrario, la norma IEEE754 define una aritmética *cerrada* en $\mathbb{F}$ introduciendo ciertos números especiales. De esta manera, en las computadoras actuales cuando un cálculo intermedio conduce a una de las situaciones excepcionales el resultado es asignado al número especial apropiado y los cálculos continúan. Así la norma IEEE754 implementa una *aritmética de no detención*.

- **Ceros.** En la norma IEEE754 el cero es representado por un número de punto flotante con una mantisa nula y exponente $e = L - 1$, pero, como ninguna condición es impuesta sobre el signo, existen dos ceros: $+0$ y -0 (con la salvedad de que en una comparación se consideran iguales en vez de $-0 < +0$). Un cero con signo es útil en determinadas situaciones, pero en la mayoría de las aplicaciones el cero del signo es invisible.

Valor	Exponente	Mantisa
normalizados	$L \leq e \leq U$	$\neq 0$
denormalizados	$L-1$	$\neq 0$
± 0	$L-1$	0
$\pm$`Infinity`	$U+1$	0
`NaN`	$U+1$	$\neq 0$

Tabla 3. El sistema de números de punto flotante extendido en la norma IEEE754.

- **Infinitos.** Cuando un cálculo produce un desbordamiento (*overflow*) positivo el resultado es asignado al número especial denominado *infinito positivo*, codificado como `+Infinity`*. De la misma manera, el cálculo de un desbordamiento negativo es asignado al número especial *infinito negativo*, codificado como `-Infinity`. Los infinitos permiten considerar también el caso excepcional de la división de un número no nulo por cero: el resultado es asignado al infinito del signo apropiado. Los infinitos son representados en el estándar por los números de punto flotante con mantisa nula y exponente $e = U + 1$ con el correspondiente signo.

- **Números denormalizados.** Tradicionalmente si una operación producía un valor de magnitud menor que $x_{\text{mín}}$ (desbordamiento a cero, o *underflow*), el resultado era asignado a cero. Ahora bien, la distancia entre cero y $x_{\text{mín}} = \beta^{L-1}$ (el menor número de punto flotante positivo representable) es mucho mayor que la distancia entre este número y el siguiente por lo que la asignación a cero de una condición de *underflow* produce errores de redondeo excepcionalmente grandes. Para cubrir esta distancia y reducir así el efecto de desbordamiento a cero a un nivel comparable con el redondeo de los números de punto flotante se implementa el *desbordamiento a cero gradual* (*gradual underflow*) introduciendo los *números de punto flotante denormalizados*. Los números denormalizados son obtenidos removiendo en la representación de punto flotante la condición de que el dígito a_1 sea no nulo *sólo* para los números que corresponden al mínimo exponente $e = L$. De esta manera la unicidad de la representación es mantenida y ahora es posible disponer de números de punto flotante en el intervalo $(-\beta^{L-1}, \beta^{L-1})$. La magnitud del más pequeño de estos números denormalizados es igual a β^{L-t}. De este modo, cuando el resultado de una operación tiene magnitud menor que $x_{\text{mín}}$ el mismo es asignado al correspondiente número de punto flotante denormalizado más próximo. Esto permite que la propiedad $x = y \Leftrightarrow (x - y) = 0$ se mantenga válida en la representación de punto flotante. En el estándar, los números denormalizados son representados como números de punto flotante con mantisa no nula y exponente $e = L - 1$.

- **NaN.** Operaciones matemáticamente ilegales, como $0/0$ ó $\sqrt{x}$ para $x < 0$, son asignadas al número especial denominado *Not a Number* (literalmente, no es un número), codificado como `NaN`. En el estándar un `NaN` es representado por un número de punto flotante con mantisa no nula y exponente $e = U + 1$. Nótese que, puesto que la mantisa no está especificada, no existe un único `NaN` sino un conjunto finito de ellos los cuales pueden utilizarse para especificar situaciones de excepción particulares.

La tabla 3 resume esta representación de los números en la norma IEEE754.Las operaciones aritméticas que involucran a los números especiales están definidas de manera de obtener resultados

*Nótese que "infinito" no significa necesariamente que el resultado sea realmente ∞, sino que significa "demasiado grande para representar".

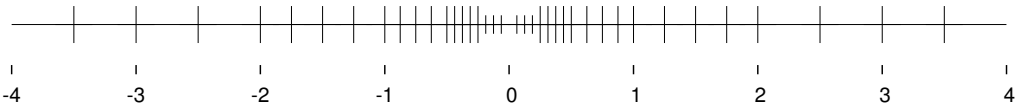


Figura 4. Números de punto flotante, incluyendo los denormalizados, del conjunto $\mathbb{F}(2, 3, -1, 2)$.

razonables, tales como

$$(\pm\texttt{Infinity}) + (+1) = \pm\texttt{Infinity}$$
$$(\pm\texttt{Infinity}) \cdot (-1) = \mp\texttt{Infinity}$$
$$(\pm\texttt{Infinity}) + (\pm\texttt{Infinity}) = \pm\texttt{Infinity}$$
$$(\pm\texttt{Infinity}) + (\mp\texttt{Infinity}) = \texttt{NaN}$$
$$1/(\pm 0) = \pm\texttt{Infinity} \qquad 1/(\pm\texttt{Infinity}) = \pm 0$$
$$0/0 = \texttt{NaN} \qquad (\pm\texttt{Infinity})/(\pm\texttt{Infinity}) = \texttt{NaN}$$
$$0 \cdot (\pm\texttt{Infinity}) = \texttt{NaN}$$

Además un `NaN` se propaga agresivamente a través de las operaciones aritméticas: *en cualquier operación donde un* `NaN` *participe como operando el resultado será un* `NaN`.

Ejemplo. Consideremos nuevamente el sistema de punto flotante $\mathbb{F}(2, 3, -1, 2)$. Los números de punto flotante denormalizados positivos corresponden a mantisas no normalizadas para el exponente $e = L = -1$ y son, pues,

$$0.001 \times 2^{-1} = \frac{1}{16}, \quad 0.010 \times 2^{-1} = \frac{1}{8}, \quad 0.011 \times 2^{-1} = \frac{3}{16}.$$

El más pequeño de estos números es, en efecto, $\beta^{L-t} = 2^{-1-3} = 1/16$. La inclusión de los mismos, junto con sus opuestos, en el sistema de puntos flotantes cubre el desbordamiento a cero en forma uniforme, tal como se ilustra en la figura 4.

Números de punto flotante y Fortran. Todos los compiladores del lenguaje Fortran admiten, al menos, dos *clases* para los tipos de datos reales: simple precisión y doble precisión los cuales se corresponden con los formatos de precisión simple y doble de la norma IEEE754. En Fortran 77, las variables reales de simple y doble precisión son declarados con las sentencias

`REAL` *nombre de variable*
`DOUBLE PRECISION` *nombre de variable*

respectivamente[*]. Las constantes literales reales de simple precisión siempre llevan un punto decimal, por ejemplo `34.0`. Si el punto decimal no está presente entonces la constante literal es un valor entero. Por otra parte, las constantes de doble precisión llevan además del punto decimal el sufijo `D0`, como ser, `34.D0`. Debería quedar claro ahora que las constantes `34`, `34.0` y `34.D0` se corresponden a tres *diferentes* tipos de números en la computadora y *no* son intercambiables. El primero constituye un entero representado en complemento a dos mientras que los otros dos son números de punto flotante en dos sistemas distintos, teniendo el último un mayor número de dígitos binarios. Muchos errores de programación (*bugs*, en la jerga) tienen su origen en no utilizar correctamente el tipo de dato. Por ejemplo, la asignación de un número de simple precisión a una variable de doble precisión *no* incrementa el número de dígitos significativos, como se ejemplifica en el siguiente programa:

[*]Es usual ver también estas declaraciones en la forma, no estándar, `REAL*4` y `REAL*8`, respectivamente. Notando que 1 byte = 8 bits, vemos que estas sentencias significan que la longitud de palabra utilizada para almacenar un dato real es de 4 bytes = 32 bits y 8 bytes = 64 bits, respectivamente.

```
program wrong0
double precision d
d = 1.66661
write(*,*) d
end
```

La ejecución del programa conduce a

```
1.66610002517700
```

Esto se debe a que la conversión de tipo de simple a doble precisión se hace simplemente agregando ceros a la representación binaria de simple precisión. Otro error típico involucra el pasaje incorrecto de tipos de datos a subprogramas. Consideremos, por ejemplo, el siguiente código,

```
program wrong1
call display(1)
end

subroutine display(x)
real x
write(*,*) x
end
```

Debido a que Fortran 77 no controla la consistencia de tipo entre los argumentos formales y los argumentos realmente pasados a un subprograma, el código anterior compilará. Al ejecutarlo obtenemos

```
1.4012985E-45
```

El problema se debe a que, mientras que el valor pasado en la llamada de la subrutina es un entero, éste es asignado a un tipo real de simple precisión debido a que el argumento formal `x` es declarado en la subrutina como tal. Puesto que la computadora no convierte el tipo de los argumentos cuando la sentencia `call` es ejecutada, la secuencia de 32 bits $0\cdots01$ correspondiente al entero 1 es interpretada como un número de punto flotante de simple precisión. Dado que el exponente sesgado correspondiente a esta secuencia de bits es 0, el número en cuestión corresponde a un número denormalizado. A saber, al número $0.0\cdots01\times 2^{-125} = 2^{-24}\cdot 2^{-125} = 2^{-149} \approx 1.4\cdot 10^{-45}$. La situación es aún más problemática si pasamos un dato de simple precisión a un subprograma que espera un dato de doble precisión:

```
program wrong2
call display(1.0)
end

subroutine display(x)
double precision x
write(*,*) x
end
```

El código compilará, ¡pero su ejecución arrojará un valor diferente en cada corrida! Esto se debe a que la subrutina imprime el número de punto flotante correspondiente a la secuencia de bits almacenados en una longitud de palabra de 64 bits, pero dado que el dato pasado está representado por una secuencia de 32 bits, los 32 bits faltantes son leídos de posiciones de memoria aledañas cuyos valores no están especificados por el programa. El número de doble precisión así construido cambia, entonces, con cada ejecución del programa.

En Fortran 90 la declaración del tipo de clase de una variable real utiliza la siguiente sintaxis:

`REAL(KIND=`*número de clase*`) ::` *nombre de variable*

donde el *número de clase* es un número entero que identifica la clase de real a utilizar. Tal número, para el compilador `gfortran`, es 4 en el caso de simple precisión y 8 para doble precisión. Si no se especifica la clase, entonces se utiliza la clase por omisión, la cual es la simple precisión. Dado que *el número de clase es dependiente del compilador*, es recomendable asignar los mismos a constantes con nombres y utilizar éstas en todas las declaraciones de tipo. Esto permite asegurar la portabilidad del programa entre distintas implementaciones cambiando simplemente el valor de las mismas.

```
INTEGER, PARAMETER :: SP = 4 ! Valor de la clase de simple precisión
INTEGER, PARAMETER :: DP = 8 ! Valor de la clase de doble precisión
...
REAL(KIND=SP) :: variables
REAL(KIND=DP) :: variables
```

Además, las constantes reales en el código son declaradas de una dada clase agregando a las mismas el guión bajo seguida del número de clase. Por ejemplo:

```
34.0             ! Real de clase por omisión
34.0_SP          ! Real de clase SP
124.5678_DP      ! Real de clase DP
```

Una manera alternativa de especificar la clase de los tipos de datos reales y que resulta *independiente del compilador y procesador utilizado* consiste en seleccionar el número de clase vía la función intrínseca `SELECTED_REAL_KIND`. Esta función selecciona automáticamente la clase del tipo real al especificar la precisión y rango de los números de punto flotante que se quiere utilizar. Para el formato de simple y doble precisión de la norma IEEE754, la asignación apropiada es como sigue:

```
INTEGER, PARAMETER :: SP = SELECTED_REAL_KIND(6,37)   ! Clase de simple precisión
INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(15,307) ! Clase de doble  precisión
```

Una manera simple y eficiente, en Fortran 90, de escribir un programa que pueda ser compilado con variables reales ya sea de una clase u otra según se requiera, consiste en utilizar un *módulo* para definir la precisión de los tipos reales y luego invocarlo, en el programa, especificando el tipo de clase vía un *alias* como ser `WP` (por *working precisión*, precisión de trabajo), la cual es utilizada para declarar los tipos de datos reales (variables y constantes). Específicamente, definimos el módulo `precision` como sigue:

```
MODULE precision
    ! ------------------------------------------------
    ! SP : simple precision de la norma IEEE 754
    ! DP : doble  precision de la norma IEEE 754
    !
    ! Uso: USE precision, WP => SP ó USE precision, WP => DP
    ! ------------------------------------------------
    INTEGER, PARAMETER :: SP = SELECTED_REAL_KIND(6,37)
    INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(15,307)
END MODULE precision
```

Entonces podemos escribir un programa que se compile ya sea con reales de simple o doble precisión escogiendo apropiadamente la sentencia que importa el módulo. Por ejemplo:

```
PROGRAM main
    USE precision, WP => SP   ! ó WP => DP
    IMPLICIT NONE
    REAL(KIND=WP) :: a
    a = 1.0_WP/3.0_WP
    WRITE (*,*) a
END PROGRAM main
```

Mientras que en Fortran 77 no existen procedimientos estándar para determinar las propiedades de la representación de punto flotante implementada en una clase de tipo real, Fortran 90, por su parte, dispone de un conjunto de funciones intrínsecas para ello, las cuales presentamos en el siguiente código que hace uso del módulo `precisión` discutido para asignar la clase de los tipos reales.

```
PROGRAM machar
  USE precision, WP => SP  ! ó WP => DP
  IMPLICIT NONE
  INTEGER :: i
  REAL(KIND=WP) :: x

  WRITE(*,*) ' base  = ', RADIX(i)
  WRITE(*,*) ' t     = ', DIGITS(x)
  WRITE(*,*) ' L     = ', MINEXPONENT(x)
  WRITE(*,*) ' U     = ', MAXEXPONENT(x)
  WRITE(*,*) ' x_max = ', HUGE(x)
  WRITE(*,*) ' x_min = ', TINY(x)
  WRITE(*,*) ' eps_M = ', EPSILON(x)

END PROGRAM machar
```

La aritmética de no detención de la norma IEEE754 permite simplificar la programación cuando los cálculos involucran los puntos singulares del sistema de punto flotante extendido. Por ejemplo, considérese el siguiente código usual para calcular el error relativo:

```
program test1
real x, y, ZERO
parameter (ZERO = 0.0E0)
write (*,*) 'Ingrese dos numeros reales: '
read (*,*) x, y
if (y .ne. ZERO) then
  write (*,*) 'El error relativo es: ', (x-y) / y
else
  write (*,*) 'El error relativo no puede ser calculado.'
endif
end
```

Con la aritmética extendida no es necesario escribir una sentencia condicional para verificar si el segundo número ingresado es nulo. Si `y` es nulo, entonces el resultado de `(x-y)/y` será un infinito (del signo apropiado) y la sentencia de escritura procederá en consecuencia.

```
program test2
real x, y
write (*,*) 'Ingrese dos numeros reales: '
read (*,*) x, y
write (*,*) 'El error relativo es: ', (x-y) / y
end
```

En efecto, la ejecución de este programa conduce a:

```
 Ingrese dos números reales:
1.0 0.0
 El error relativo es: +Infinity
```

Muchos usuarios, sin embargo, encuentran a la aritmética de no detención algo confusa y prefieren que los cálculos sean abortados con un apropiado mensaje de error. El compilador `gfortran` permite anular el comportamiento del estándar en las situaciones excepcionales utilizando la opción `-ffpe-trap=invalid,zero,overflow,underflow,denormal` en el comando de compilación. Así la ejecución del código anterior, compilado con esta opción, conduce a:

```
 Ingrese dos números reales:
1.0 0.0
Floating point exception
```

deteniéndose inmediatamente la ejecución del programa.

Dígitos de guarda. Implementación de las operaciones de punto flotante en las computadoras personales. La implementación de la operación de resta en el sistema de punto flotante, para poder satisfacer efectivamente el requisito impuesto $x \ominus y = (x + y)(1 + \delta)$, siendo $|\delta| < \mathbf{u}$, requiere la presencia de uno o dos dígitos *extra* en las mantisas de los números a restar. Tales dígitos extra se conocen como *dígitos de guarda* o *de respaldo*. Para ejemplificar la importancia de los mismos consideremos, en el sistema $\mathbb{F}$ con $\beta = 10$ y $t = 2$, la resta de los números $x = 1$ e $y = 0.99$. Para restar el menor de ellos del mayor, debe desplazarse el punto (decimal en este caso) de y un dígito a la derecha para igualar los exponentes. En este proceso, y pierde un dígito significativo y el resultado que se obtiene es 0.10, el cual difiere del resultado exacto por un factor de 10.

$$\begin{array}{rl} & 0.10 \times 10^1 \\ - & \underline{0.09 \times 10^1,} \\ & 0.01 \times 10^1 \end{array} \quad = \quad 0.10 \times 10^0.$$

Consideremos ahora la misma operación, pero con un dígito de guarda. Ahora, el dígito menos significativo no se pierde al alinear, y el resultado obtenido de la operación es exacto.

$$\begin{array}{rl} & 0.10\,0 \times 10^1 \\ - & \underline{0.09\,9 \times 10^1,} \\ & 0.00\,1 \times 10^1 \end{array} \quad = \quad 0.01 \times 10^0.$$

En la implementación en una computadora, como paso previo a una operación de punto flotante, el exponente y la mantisa de los números a operar se cargan en ciertas regiones de memoria de la unidad central de procesamiento (CPU, o simplemente *procesador*) conocidas como *registros*. En general, el tamaño del registro es siempre mayor que la longitud de la mantisa. Los bits adicionales, que se añaden a la derecha de la mantisa en forma de ceros, pueden, entonces, actuar como bits de guarda durante una resta.

Todas las computadoras actuales poseen procesadores basados en la arquitectura de 32 bits diseñada por la empresa Intel (arquitectura designada como IA-32, o más informalmente *x86*)*. En esta arquitectura el tamaño de los registros destinados a las operaciones de puntos flotante es de una longitud de palabra de $N = 80$ bits. De este modo, aunque tengamos datos reales de simple (32 bits) o doble (64 bits) precisión en las variables del programa, cuando las mismas son leídas de la memoria principal a los registros del procesador se convierten a 80 bits. El procesador efectúa entonces las operaciones requeridas en 80 bits y, después del cálculo, el resultado es guardado nuevamente en la variable de 32 ó 64 bits correspondiente. Específicamente, el formato de punto flotante implementado en los 80 bits es un *formato extendido* donde 64 bits se utilizan para la mantisa, 15 bits para el exponente (representado en forma sesgada, con un sesgo de 16383) y un bit para el signo. Por razones históricas este formato no tiene bit implícito. Una consecuencia de este diseño es que el resultado final de un cálculo puede ser más preciso de lo esperado debido a que los

*Aún los procesadores más modernos de las computadoras personales, que extienden este diseño a 64 bits, mantienen la compatibilidad. Esta arquitectura es conocida como *x86-64* o simplemente *x64*.

cálculos intermedios involucrados son efectuados en precisión extendida. En general esto es beneficioso, pero es el responsable del comportamiento observado en el segundo programa presentado en la introducción de estas notas. En efecto, notemos primeramente que el número (decimal) 0.1 tiene una representación binaria infinita y, por lo tanto, su representación normalizada en simple precisión sólo dispone de 24 dígitos binarios. Por otra parte, la operación división entre los números 1.0 y 10.0 es efectuada en precisión extendida lo que equivale a redondear a 64 dígitos binarios la representación binaria infinita de 0.1 La comparación entre este resultado y la representación de simple precisión de 0.1 *es efectuada en el registro en precisión extendida comparando bit a bit* completado con ceros los bits restantes del formato simple del 0.1. Claramente la representación de punto flotante en precisión extendida de dichas cantidades es diferente y, por lo tanto, la comparación de igualdad en la sentencia condicional arroja el valor falso. Incidentalmente, si el resultado de la división es guardado en una variable real, el resultado en precisión extendida es convertido a precisión simple y ahora la comparación arroja el valor verdadero.

```
program main
implicit none
real a, b
a = 10.0
b = 1.0/a
if (b.eq.0.1) then
     write(*,*) '1/10 es igual a 0.1'
else
     write(*,*) '1/10 es distinto a 0.1'
endif
end
```

```
1/10 es igual a 0.1
```

En realidad, este comportamiento no depende solamente de la arquitectura *x86*, sino también del compilador utilizado. En particular, el compilador `gfortran` (y toda la suite de compiladores GNU) generan código sobre la arquitectura *x86* que trabaja internamente en precisión extendida. Otros compiladores, tales como el `ifort` de Intel, siguen estrictamente el estándar IEEE754 y la situación presentada no se da. En cualquier caso, testear la igualdad de dos números de puntos flotantes no es recomendable[*].

Una consideración final, ¿cuándo conviene utilizar doble precisión? La respuesta corta es *siempre*. Para las aplicaciones científicas la precisión de los resultados es generalmente crucial, con lo cual debe utilizarse la mejor representación de punto flotante disponible en la computadora. Puesto que los cálculos intermedios se realizan internamente con 80 bits independientemente de la precisión de los datos a ser procesados *no existe una diferencia de velocidad sustancial entre los cálculos en doble y simple precisión.* Por supuesto, la cantidad de memoria utilizada para los datos de doble precisión es el doble que para sus contrapartes de simple precisión pero, en tanto la disponibilidad de memoria no sea un límite, el uso de datos de doble precisión es, en general, la mejor elección.

Redondeo en la norma IEEE754. La política de redondeo implícitamente contemplada en el estándar es el redondeo al más próximo. Sin embargo, el estándar IEEE754 enumera cuatro posibles alternativas para redondear el resultado de una operación:

- *Redondeo al más próximo*: el resultado se redondea al número más próximo representable.

[*]Para aquellos (raros) programas que requieren trabajar a la precisión dictada por el estándar, en vez de modificar el código de manera de almacenar los resultados intermedios en variables, el compilador `gfortran` dispone de la opción `-ffloat-store` para mitigar el exceso de precisión.

- *Redondeo hacia* 0: el resultado se redondea hacia cero.
- *Redondeo hacia* $+\infty$: el resultado se redondea por exceso hacia más infinito.
- *Redondeo hacia* $-\infty$: el resultado se redondea por exceso hacia menos infinito.

El redondeo al más próximo ya ha sido discutido. El redondeo hacia cero consiste en un simple truncamiento donde los dígitos siguientes al $t-$ésimo son ignorados. Esta es, ciertamente, una política de redondeo muy simple de implementar, pero la unidad de redondeo correspondiente es $\mathbf{u} = \epsilon_M$, un factor dos veces mayor que la correspondiente al redondeo al más próximo. Las dos siguientes opciones, redondeo a más o menos infinito, son útiles en la implementación de una técnica conocida como *aritmética de intervalos*. La aritmética de intervalos proporciona un método eficiente para monitorizar y controlar errores en los cálculos en punto flotante, produciendo dos valores por cada resultado. Los dos valores se corresponden con los límites inferior y superior de un intervalo que contiene al resultado exacto. La longitud del intervalo, que es la diferencia de los límites superior e inferior, indica la precisión del resultado. Si los extremos del intervalo no son representables, se redondean por defecto y por exceso, respectivamente. Aunque la anchura del intervalo puede variar de unas implementaciones a otras, se han diseñado diversos algoritmos que permiten conseguir intervalos estrechos. Si dicho intervalo es suficientemente estrecho se obtiene un resultado bastante preciso. Si no, al menos lo sabremos y podemos considerar realizar análisis adicionales.

Propagación del error de redondeo.

Una cuestión de suma importancia es determinar el efecto de la *propagación* de un error introducido en algún punto de un cálculo, es decir, determinar si su efecto aumenta o disminuye al efectuar operaciones subsiguientes. Si los errores de redondeo se mantuvieran acotados después de cualquier número de operaciones entonces el contenido de estas notas no tendría ninguna importancia. La cuestión es que los errores de redondeo pueden acumularse de manera tal que el resultado numérico obtenido es una pobre aproximación al resultado buscado. El hecho de que los errores de redondeo en el cálculo de una cantidad puedan amplificarse o no dependen de los cálculos realizados, esto es, del algoritmo numérico utilizado o, incluso, puede depender de la propia naturaleza del problema que se está resolviendo. En lo que sigue ejemplificamos algunas de las situaciones que pueden, y suelen, ocurrir.

En la sección anterior hemos definido una aritmética en $\mathbb{F}$ asumiendo que los números involucrados eran números de punto flotante. Si ahora x e y son dos números reales que no son exactamente representados en $\mathbb{F}$, la operación de punto flotante $\circledcirc$ correspondiente a la operación aritmética $\circ$ procede como sigue:

$$x \circledcirc y = fl(fl(x) \circ fl(y))$$

Es decir, la operación se corresponde a efectuar la aritmética exacta en las representaciones de punto flotante de x e y, y luego convertir el resultado a su representación de punto flotante. En particular, cuando $\circ$ es el operador suma, se sigue que

$$x \oplus y = (x(1+\delta_1) + y(1+\delta_2))(1+\delta),$$

para $|\delta_1|, |\delta_2|, |\delta| \leq \mathbf{u}$. El error relativo cometido en la suma está acotado, entonces, por

$$\frac{|x \oplus y - (x+y)|}{|x+y|} \leq \mathbf{u} + \frac{|x|+|y|}{|x+y|}(1+\mathbf{u})\mathbf{u}.$$

Además del error originado por la operación en sí, vemos que existe un término adicional proveniente de la representación inexacta de los sumandos. Este término será pequeño en tanto $x+y$

no sea pequeño. Ahora bien, si los números a sumar son aproximadamente iguales en magnitud, pero de signos opuestos, el error relativo puede ser muy grande, aún cuando el error proveniente de la operación en sí no lo es. Esta situación, conocida como *fenómeno de cancelación de dígitos significativos* debe ser evitada tanto como sea posible.

Por ejemplo, la fórmula cuadrática nos dice que las raíces de $ax^2 + bx + c = 0$ cuando $a \neq 0$ son

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Si $b^2 \gg 4ac$ entonces, cuando $b > 0$ el cálculo de x_1 involucra en el numerador la sustracción de dos números casi iguales, mientras que, si $b < 0$, esta situación ocurre para el cálculo de x_2. "Racionalizando el numerador" se obtienen las siguientes fórmulas alternativas que no sufren de este problema,

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}}, \qquad x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}},$$

siendo la primera adecuada cuando $b > 0$ y la segunda cuando $b < 0$.

Ejemplo. Consideremos la ecuación

$$0.05x^2 - 98.78x + 5.015 = 0$$

cuyas raíces, redondeadas a diez dígitos significativos, son

$$x_1 = 1971.605916, \qquad x_2 = 0.05077069387.$$

Supongamos que efectuamos los cálculos utilizando aritmética decimal a cuatro dígitos, con redondeo al más próximo. Primero tenemos que, en esta aritmética,

$$\begin{aligned} \sqrt{b^2 - 4ac} &= \sqrt{9757 - 1.005} \\ &= \sqrt{9756} \\ &= 98.77 \end{aligned}$$

así que,

$$x_1 = \frac{98.78 + 98.77}{0.1002} = 1972, \quad x_2 = \frac{98.78 - 98.77}{0.1002} = 0.0998.$$

La aproximación obtenida para x_1 es correcta a cuatro dígitos, pero la obtenida para x_2 es completamente equivocada, con un error relativo del orden del 100 %. La resta de $-b$ y la raíz cuadrada del determinante expone aquí los errores de redondeo previos en el cálculo del discriminante y, dominando el error, conducen a una drástica cancelación de dígitos significativos en x_2. Utilizando la fórmula alternativa para x_2 encontramos que, a cuatro dígitos decimales,

$$x_2 = \frac{10.03}{98.78 + 98.77} = 0.05077,$$

la cual es una aproximación precisa de dicha raíz.

En general, el fenómeno de cancelación puede ser evitado reescribiendo en forma apropiada la fórmula que origina el problema. Por ejemplo, la expresión $\sqrt{x + \delta} - \sqrt{x}$ puede ser reescrita como

$$\sqrt{x + \delta} - \sqrt{x} = \frac{x + \delta - x}{\sqrt{x + \delta} + \sqrt{x}} = \frac{\delta}{\sqrt{x + \delta} + \sqrt{x}},$$

evitándose así la cancelación cuando $|\delta| \ll x$. En general, si no puede encontrarse una manera exacta de reescribir la expresión $f(x+\delta) - f(x)$, donde f es una función continua, entonces puede resultar apropiado utilizar uno o más términos de la serie de Taylor de f

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

Un problema donde el fenómeno de cancelación juega un papel relevante es la estimación de la derivada de una función f en un punto x a través de una fórmula de *diferenciación numérica*.

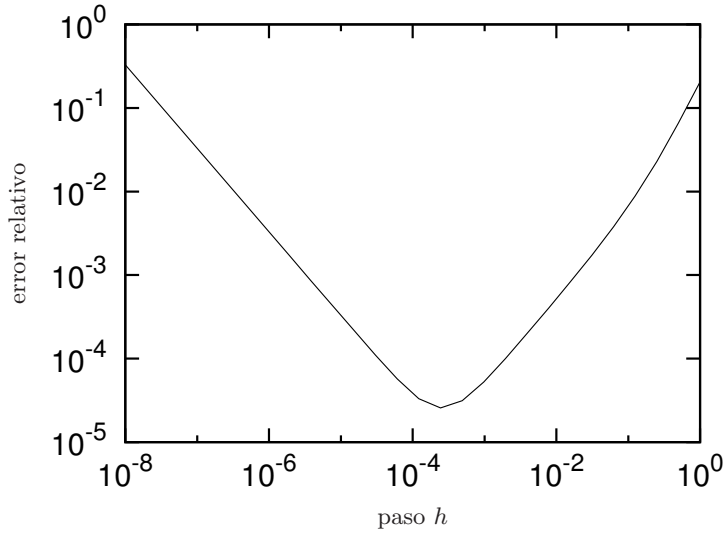


Figura 5. Error relativo en una fórmula de diferenciación numérica como función del paso h (escalas logarítmicas en ambos ejes).

Estas fórmulas utilizan los valores de f en puntos próximos a x equispaciados en una distancia, o paso, h. En principio, parece ser que podemos calcular la derivada con cualquier precisión deseada tomando el paso h suficientemente pequeño. Sin embargo estas fórmulas adolecen de cancelación debido a la necesidad de dividir por una potencia de h la resta de dos cantidades que tienden a ser iguales conforme $h \to 0$. Como consecuencia de ésto el paso h no puede ser tomado tan pequeño como se desee, sino que existe un valor a partir del cual el error en la estimación numérica de la derivada comienza a incrementarse nuevamente.

Ejemplo. Consideremos la más simple fórmula de diferenciación numérica para estimar $f'(x)$, a saber

$$\frac{f(x+h)-f(x)}{h}.$$

Supongamos que al evaluar $f(x+h)$ y $f(x)$ tenemos errores (absolutos) de redondeo $e(x+h)$ y $e(x)$, esto es, en el cálculo utilizamos los números $\hat{f}(x+h)$ y $\hat{f}(x)$ siendo

$$f(x+h) = \hat{f}(x+h) + e(x+h) \quad \text{y} \quad f(x) = \hat{f}(x) + e(x).$$

Entonces, el error en la aproximación *calculada* está dado por:

$$f'(x) - \frac{\hat{f}(x+h)-\hat{f}(x)}{h} = f'(x) - \frac{f(x+h)-f(x)}{h} + \frac{e(x+h)-e(x)}{h}.$$

Suponiendo que f es derivable con continuidad al menos hasta la derivada segunda, el desarrollo de Taylor de f en torno a x nos dice que

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(\xi)h^2,$$

donde ξ es un punto localizado entre x y $x+h$. Luego,

$$f'(x) - \frac{\hat{f}(x+h)-\hat{f}(x)}{h} = -\frac{h}{2}f''(\xi) + \frac{e(x+h)-e(x)}{h}.$$

Vemos, pues, que el error consta de dos partes: una debida a la discretización introducida por la fórmula numérica (*error de truncamiento o discretización*) y la otra debida a los errores de redondeo. Si $M > 0$ es una cota de f'' y los errores de redondeo están acotados por un número $\delta > 0$, tenemos que

$$\left| f'(x) - \frac{\hat{f}(x+h)-\hat{f}(x)}{h} \right| \leq \frac{Mh}{2} + \frac{2\delta}{h}.$$

Así, mientras que el error de truncamiento tiende a cero cuando $h \to 0$, el error de redondeo puede crecer sin límite. Luego, para h suficientemente pequeño, el error de redondeo dominará por sobre el de truncamiento. El valor *óptimo* de h será aquel para el cual la cota del error total toma su valor mínimo, a saber (¡verifíquelo!)

$$h_{\text{ópt}} = 2\sqrt{\delta/M}.$$

Escribiendo la cota del error (absoluto) de redondeo como $\delta = \epsilon|f(x)|$, siendo ϵ una cota para el error relativo, y asumiendo que f y f'' tienen el mismo orden de magnitud, $h_{\text{ópt}} \approx \sqrt{\mathbf{u}}$. En precisión simple, esto implica que el error alcanza su mínimo en $h \approx 10^{-4}$ tal como se observa en la figura 5 para una función particular.

Debido a la acumulación de los errores de redondeo, un algoritmo que es matemáticamente correcto puede *no* serlo numéricamente. En estas circunstancias, los errores de redondeo involucrados en los cálculos "conspiran" de manera tal que se obtienen resultados sin ningún sentido. Este desagradable fenómeno es conocido como *inestabilidad numérica*.

Ejemplo. Consideremos el problema de calcular las integrales

$$I_n = \int_0^1 x^n e^{x-1} \mathrm{d}x, \quad \text{para } n = 1, 2 \ldots$$

Es claro de la definición que

$$I_1 > I_2 > \cdots > I_{n-1} > I_n > \cdots > 0.$$

Integrando por partes resulta que

$$I_n = 1 - \int_0^1 n x^{n-1} e^{x-1} \mathrm{d}x = 1 - n\, I_{n-1}.$$

y dado que

$$I_1 = 1 - \int_0^1 e^{x-1} \mathrm{d}x = e^{-1},$$

obtenemos el siguiente procedimiento *recursivo* para calcular las integrales:

$$\begin{aligned} I_1 &= e^{-1}, \\ I_n &= 1 - I_{n-1}, \quad n = 2, 3, \ldots \end{aligned}$$

Implementado este algoritmo en simple precisión encontramos los siguientes valores

$$\begin{aligned} I_1 &= 0.3678795 \\ I_2 &= 0.2642411 \\ &\vdots \\ I_{10} &= 0.0506744 \\ I_{11} &= 0.4425812 \quad \text{¡los valores deberían decrecer!} \\ I_{12} &= -4.310974 \quad \text{¡los valores exactos son positivos!} \\ &\vdots \\ I_{20} &= -0.2226046 \times 10^{11} \quad \text{¡los valores exactos están entre 0 y 1!} \end{aligned}$$

Para comprender lo que sucede utilizaremos una aproximación del análisis del error conocida como *análisis inverso del error*. Hemos visto que cualquiera sea la operación aritmética $\circ$, la correspondiente operación de punto punto flotante, denotada por $\circledcirc$, está definida y satisface la siguiente relación:

$$x \circledcirc y = fl(x \circ y) = (x \circ y)(1 + \delta), \qquad \text{siendo } |\delta| \leq \mathbf{u}.$$

Esta expresión permite interpretar una operación aritmética de punto flotante como el resultado de la operación aritmética exacta sobre operandos ligeramente perturbados. Así, por ejemplo, la

multiplicación de punto flotante, $fl(x \cdot y) = x \cdot y\,(1+\delta)$, puede interpretarse como el producto exacto de los números $x' = x$ e $y' = y\,(1+\delta)$, los cuales difieren relativamente poco de x e y, respectivamente. Este punto de vista del error se denomina análisis inverso del error. Aplicando tal interpretación, paso a paso, en las operaciones de un algoritmo numérico podemos concluir que, independientemente de cuan complicado sea el mismo, *los resultados que un algoritmo produce, bajo la influencia de los errores de redondeo, son el resultado exacto de un problema del mismo tipo en el cual los datos de entrada están perturbados por cantidades de magnitud (relativa) del orden de la unidad de redondeo* $\mathbf{u}$ *de la máquina*. Aplicando el análisis inverso del error transferimos el problema de estimar los efectos del redondeo durante los cálculos de un algoritmo, al problema de estimar los efectos de pequeñas perturbaciones en los datos de entrada. Esto permite, a su vez, establecer la siguiente definición: un algoritmo numérico se dice *numéricamente estable* si pequeños cambios en los datos iniciales producen, en correspondencia, pequeños cambios en los resultados finales. De lo contrario se dice que el algoritmo es *numéricamente inestable*.

Consideremos, pues, a la luz del análisis inverso del error, la fórmula recursiva que calcula I_n. Supongamos, entonces, que comenzamos el proceso iterativo con $\hat{I}_n = I_1 + \delta$ y efectuamos todas las operaciones aritméticas en forma exacta. Entonces,

$$\begin{aligned}
\hat{I}_2 &= 1 - 2\hat{I}_1 = 1 - 2I_1 - 2\,\delta = I_2 - 2\,\delta \\
\hat{I}_3 &= 1 - 3\hat{I}_2 = 1 - 3I_2 - 6\,\delta = I_3 - 3!\,\delta \\
&\vdots \\
\hat{I}_n &= I_n \pm n!\,\delta.
\end{aligned}$$

Vemos, pues, que un pequeño cambio en el valor inicial I_1 origina un gran cambio en los valores posteriores de I_n. De hecho, en este caso, el efecto es devastador: puesto que los valores exactos de I_n decrecen conforme n aumenta, a partir de cierto n el error será tan o más grande que el valor a estimar. Claramente, el algoritmo recursivo propuesto es numéricamente inestable.

Considérese ahora la fórmula recursiva escrita en forma inversa,

$$I_{n-1} = \frac{1 - I_n}{n}, \quad n = \ldots, N, N-1, \ldots, 3, 2, 1.$$

Si conocieramos una aproximación $\hat{I}_N$ a I_N para algún N, entonces la fórmula anterior permitiría calcular aproximaciones a $I_{N-1}, I_{N-2}, \ldots, I_1$. Para determinar la estabilidad del proceso recursivo consideremos, en forma análoga a lo anterior, que $\hat{I}_N = I_N + \delta$, entonces

$$\begin{aligned}
\hat{I}_{N-1} &= \frac{1 - \hat{I}_N}{N} = \frac{1 - _N}{N} - \frac{\delta}{N} = I_{N-1} - \frac{\delta}{N} \\
\hat{I}_{N-2} &= I_{N-2} - \frac{\delta}{N(N-1)} \\
&\vdots \\
\hat{I}_1 &= I_1 \pm \frac{\delta}{N!}.
\end{aligned}$$

Vemos, pues, que el error introducido en I_N es rápidamente reducido en cada paso. Más aún, en este caso, puesto que en la dirección de la recursión los valores de I_N se incrementan conforme n disminuye, la magnitud del error tiende a ser mucho menor respecto del valor a estimar. Esto último permite que, aún con una pobre aproximación $\hat{I}_N$ para algún N suficientemente grande, podamos obtener una aproximación precisa para el valor I_n de interés. Por ejemplo, la desigualdad

$$0 < I_n < \int_0^1 x^n \mathrm{d}x = \frac{1}{n+1},$$

nos dice que, si tomamos $N = 20$, la (cruda) aproximación $\hat{I}_{20} = 0$ tiene un error absoluto a lo más $1/21$

en magnitud. Las aproximaciones obtenidas por el proceso recursivo son, entonces,

$$\begin{aligned}
\hat{I}_{20} &= 0 \\
\hat{I}_{19} &= 0.05 \\
&\vdots \\
\hat{I}_{15} &= 0.059017565 \\
&\vdots \\
\hat{I}_{1} &= 0.3678795
\end{aligned}$$

La aproximación para I_1 coincide con el resultado exacto a seis decimales. Claramente, el algoritmo construido es numéricamente estable.

Nótese que algunos algoritmos pueden ser estables para cierto grupo de datos iniciales y no para otros. Nótese también que la pérdida de precisión puede tener su origen no en el algoritmo escogido sino en el problema numérico en si mismo. Esto es, los resultados pueden ser muy sensibles a las perturbaciones de los datos de entrada *independientemente* de la elección del algoritmo. En esta situación se dice que *el problema está mal condicionado* o bien que es *matemáticamente inestable*.

Ejemplo. Consideremos nuevamente la ecuación cuadrática $ax^2 + bx + c = 0$. Si el discriminante $b^2 - 4ac$ es pequeño respecto de los coeficientes entonces las dos raíces x_1 y x_2 están muy próximas una de otra y su determinación constituye un problema mal condicionado. En general, la determinación de la raíz α de un polinomio f para el cual $|f'(\alpha)|$ es próximo a cero es un problema mal condicionado. Un ejemplo drástico es dado por el polinomio

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

cuyas raíces son $1, 2, \ldots, 19, 20$. Si el coeficiente de x^{19} es perturbado por 2^{-23}, muchas de las raíces se apartan enormemente de las del polinomio original, incluso cinco pares de ellas resultan complejas con partes imaginarias de magnitud sustancialmente no nulas.

Debería quedar claro ahora que, debido a los errores de redondeo, leves cambios en el orden de las operaciones pueden cambiar los resultados. Más aún, ya que diferentes compiladores y procesadores almacenan los cálculos intermedios con diferentes precisiones, tales resultados variarán dependiendo de la implementación. Así dos números obtenidos a partir de cálculos independientes que se esperan sea iguales, no lo serán, en general, dentro de la aritmética de punto flotante. Una línea de código que testee la igualdad en la forma

```
if (x .eq. y) ...
```

arrojará, en general, el valor lógico falso. Por lo tanto al testear la igual de dos números debemos permitir cierta tolerancia en la comparación:

```
if ( abs(x-y) .le. eps ) ...
```

o bien, en un sentido relativo,

```
if ( abs(x-y) .le. eps*max(abs(x),abs(y)) ) ...
```

donde `eps` es un número pequeño, pero mayor que la unidad de redondeo **u**, adecuadamente escogido. La elección de `eps` depende de los errores de redondeo esperados en la evaluación de `x` e `y` y no hay, por lo tanto, un valor único que se adapte a todas las situaciones.

Colofón

Finalizamos estas notas con el siguiente extracto del libro *The Art of Computer Programming* de Donald E. Knuth.

> Los cálculos de punto flotante son, por naturaleza, inexactos, y no es difícil que un mal uso de los mismos conduzca a resultados erroneos. Uno de los principales problemas del análisis numérico es determinar cuan precisos serán los resultados de los métodos numéricos. Un problema de "falta de credibilidad" está involucrado aquí: no sabemos cuánto debemos creer en los resultados obtenidos con la computadora. Los usuarios novatos resuelve este problema confiando implícitamente en la computadora como si se tratara de una autoridad infalible, tienden a creer que todos los dígitos del resultado obtenido son significativos. Usuarios desilusionados toman el enfoque opuesto, están constantemente temerosos de que sus resultados no tenga mayormente ningún sentido.

Referencias

- William Stallings, *Organización y Arquitectura de Computadores*, Quinta Edición, Prentice Hall, 2000.
- David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Vol 23, No 1, March 1991.
- User Notes On FORTRAN Programming, An open cooperative practical guide, `http://sunsite.icm.edu.pl/fortran/unfp.html`
- Alfio Quarteroni, Riccardo Sacco, Fausto Saleri, *Numerical Mathematics*, Springer, 2000.
- L. F. Shampine, Rebecca Chan Allen, S. Pruess, R. C. Allen, *Fundamentals of Numerical Computing*, Wiley, 1997.
- Michael T Heath, *Scientific Computing*, McGraw-Hill, 2001.
- H.M. Antia, *Numerical Methods for Scientists and Engineers*, Tata McGraw-Hill India, 1991.
- Germun Dahlquist, Åke Björck, Numerical Methods, Prentice-Hall, 1974.