Representación Computacional de Números Reales

Diego Feroldi
feroldi@fceia.unr.edu.ar

Arquitectura del Computador*
Departamento de Ciencias de la Computación
FCEIA-UNR

*Actualizado 12 de septiembre de 2024 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

1. Introducción

Para representar números reales es necesario utilizar una coma o punto para separar la parte entera de la parte fraccionaria, independientemente del sistema de representación con que se trabaje (decimal, binario, etc.). Existen dos formas de resolver el problema:

- **Punto fijo:** Se considera la coma o punto en cierta posición fija.
- **Punto flotante:** La posición del punto decimal dentro del número es variable y se almacena esa posición mediante un exponente.

2. Representación de números reales con punto fijo

Este método supone que el punto (o coma) está siempre en una posición fija. Aunque el punto o coma no se almacena explícitamente, su presencia se asume implícitamente en una posición determinada. Los dígitos a la izquierda de esta posición se consideran como un entero (potencias positivas), mientras que los dígitos a la derecha se consideran como una fracción (potencias negativas).

Ventajas: La aritmética de punto fijo es relativamente simple.

Desventajas: El rango y la precisión de representación, es decir, la cantidad de valores que se pueden representar y la menor diferencia que se puede distinguir, son muy limitados[1].

Ejemplo

Veamos un sistema simple de representación con 8 bits con la convención magnitud y signo, de los cuales hemos reservado 1 bit para el signo, 4 para la parte entera y 3 para la fraccionaria. Por lo tanto:

$$(11011.011)_2 = -(1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) = (-13.375)_{10}$$

[1] En un sistema de representación de punto fijo, la precisión se refiere a la diferencia entre un número representable y el siguiente. Más adelante, veremos cómo en los sistemas de punto flotante el concepto de precisión se vuelve más complejo.

- ***Mayor número representable:***
  $(01111.111)_2 = 2^4 - 2^{-3} = (15.875)_{10}$
- ***Menor número representable:***
  $(11111.111)_2 = -(2^4 - 2^{-3}) = (-15.875)_{10}$
- ***Precisión:***
  $(00000.001)_2 = 2^{-3} = (0.125)_{10}$

3. Representación de números reales con punto flotante

Como hemos visto en la sección anterior, la representación con punto fijo tiene importantes limitaciones en cuanto al rango de representación y precisión. A continuación, veremos cómo estas limitaciones pueden superarse utilizando la notación científica.

3.1. Notación científica normalizada

Para la representación de números reales sobre un amplio rango de valores se emplea la notación científica. Por ejemplo, el número 976000000000000 se puede representar como $9.76{\times}10^{14}$ mientras que el número 0.0000000000000000976 se puede representar como 9.76×10^{-16}. En esta notación la coma o punto decimal se mueve dinámicamente a una posición conveniente y se utiliza el exponente en base 10 para registrar la posición del punto decimal.

Sin embargo, observemos que este tipo de notación tiene cierta ambigüedad dado que un mismo número puede ser representado de diferentes maneras. En el ejemplo anterior el número 976000000000000 también puede escribirse como 0.976×10^{15}, etc. Por lo tanto es necesario definir una normalización.

Definición 1. Todo número real no nulo se puede escribir en forma única en la **notación científica normalizada** como:

$$N = \pm(a_0.a_{-1}a_{-2}a_{-3}\ldots a_{-t}) \times 10^e,$$

siendo el dígito $a_0 \neq 0$ y $1 \leq a_{-i} \leq 9$.

Por lo tanto, en el ejemplo anterior 9.76×10^{14} es la forma normalizada.

Definición 2. De forma general, todo número real no nulo puede representarse de manera única en una base β de la siguiente forma:

$$N = (-1)^s \times (a_0.a_{-1}a_{-2}a_{-3}\ldots a_{-t})_\beta \times \beta^e,$$

donde los dígitos a_i son enteros positivos tales que $1 \leq a_0 \leq \beta - 1$, y $0 \leq a_i \leq \beta - 1$ para $i = -1, -2, \ldots$, constituyendo los dígitos del **significante**. El valor e es el **exponente**, que indica la posición del punto en relación con la base β. Finalmente, s es el **signo**, con la convención de que $s = 0$ si N es positivo y $s = 1$ si es negativo. Por lo tanto,

$$N = (-1)^s \times \left(a_0 \times \beta^0 + \sum_{i=1}^{t} a_{-i} \times \beta^{-i} \right) \times \beta^e.$$

En el caso particular del sistema binario, donde $\beta = 2$, el único valor posible para el dígito a la izquierda del punto es $a_0 = 1$. Así, el significando resulta $m = (1.a_{-1}a_{-2}\ldots a_{-t})_2$. Esto se puede expresar como:

$$m = 1 + a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \ldots + a_{-t} \times 2^{-t}$$

Por lo tanto, el número representado corresponde al número decimal $N = (-1)^s \times m \times 2^e$.

Ejemplo

Dado el número en formato decimal $N = (3.375)_{10}$, se puede representar en binario como $(11.011)_2$. Normalizando, tenemos:

$$N = (3.375)_{10} = (-1)^0 \times (1.1011)_2 \times 2^1$$

Observación

Al realizar la normalización se debe tener en cuenta la corrección del exponente.

3.2. Características de los números en notación científica normalizada

Sea el conjunto $\mathbb{F}(\beta, t, L, U)$ de números en notación científica normalizada, donde t es la cantidad de dígitos significativos en la parte fraccionaria, y $(L \leq 0)$ y $(U > 0)$ son las cotas inferior y superior, respectivamente, del exponente e, cumpliendo con $L \leq e \leq U$:

1. El número de elementos del conjunto $\mathbb{F}$ es $2(\beta - 1)\beta^t(U + L + 1)$.

2. El cero no puede ser representado como punto flotante debido a la normalización.

3. Si $x \in \mathbb{F}$, entonces su opuesto $-x \in \mathbb{F}$.

4. El conjunto está acotado tanto superior como inferiormente:

$$x_{min} \leq |x| \leq x_{max},$$

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

5. Hay cinco regiones excluidas para los números del conjunto $\mathbb{F}$:

   - Los números negativos menores que $-x_{max}$ ("*overflow*" negativo).
   - Los números negativos mayores que $-x_{min}$ ("*underflow*" negativo).
   - El cero.
   - Los números positivos menores que x_{min} ("*underflow*" positivo).
   - Los números positivos mayores que x_{max} ("*overflow*" positivo).

6. Los números en en el conjunto $\mathbb{F}$ no están igualmente espaciados sobre la recta real, si no que están más próximos cerca del origen y más espaciados a medida que nos alejamos del origen (Este tema lo veremos en detalle en la Sección 5.4).

3.3. Redondeo de un número real

Dado que solo los números reales dentro del conjunto $\mathbb{F}$ pueden ser representados, en general, los números reales deben aproximarse a un valor perteneciente a este conjunto, denotado como $fl(x)$. La manera usual de proceder consiste en aplicar el redondeo simétrico a t dígitos en la parte fraccionaria del número real x. Esto es, a partir de

$$x = (-1)^s \times (a_0.a_{-1}a_{-2}\ldots a_{-t}\ldots)_\beta \times \beta^e$$

obtenemos $fl(x)$ como

$$fl(x) = (-1)^s \times (a_0.a_{-1}a_{-2}\ldots \tilde{a}_{-t})_\beta \times \beta^e$$

con

$$\tilde{a}_t = \begin{cases} a_t & \text{si} \quad a_{t+1} < \beta/2, \\ a_t + 1 & \text{si} \quad a_{t+1} \geq \beta/2. \end{cases}$$

El error que resulta se denomina **error de redondeo**.

Ejemplos

Supongamos que queremos redondear con 5 dígitos significativos:

$$N_1 = (4.567689010\ldots)_{10} \quad \rightarrow \quad (4.5677)_{10} \quad \rightarrow \quad \textbf{error} \approx 6.9 \times 10^{-4}$$
$$N_2 = (1.10101010\ldots)_2 \quad \rightarrow \quad (1.1011)_2 \quad \rightarrow \quad \textbf{error} = 0.0234$$

En la Sección 5.6 analizaremos en detalle el error que se comete al utilizar el estándar IEEE 754 para números en punto flotante.

4. Estándar IEEE 754 para números en punto flotante

Para evitar la proliferación de diversos sistemas de punto flotante, el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) creó un comité en 1985 para formular una norma para números en punto flotante: el Estándar IEEE 754. Un número en formato IEEE 754 puede ser expresado a partir de la definición de notación científica normalizada de la siguiente manera:

$$\boxed{N = (-1)^s \times 2^e \times (1.f)}, \tag{1}$$

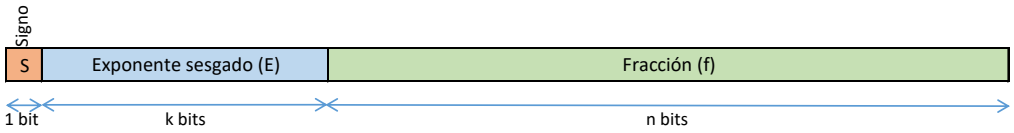
donde:

- s indica el signo del número con la siguiente convención:

$$\mathrm{s} = \begin{cases} 0 & \text{si el número es positivo,} \\ 1 & \text{si el número es negativo.} \end{cases} \tag{2}$$

- e es el exponente del número.
- $(1.f)$ es el significante, donde f es la parte fraccionaria[2].

La representación a nivel bits de un número de punto flotante formato IEEE 754 se divide en tres campos de bits a partir de la Ecuación (1):



- **Bit de signo (s)**: El bit más a la izquierda codifica directamente el signo s con la convención expresada en la Ecuación (2).
- **Bits del exponente sesgado (E)**: Los siguientes k bits codifican el exponente sesgado E. En la Sección 4.2 veremos en detalle el concepto de exponente sesgado.
- **Bits de la fracción (f)**: Los últimos n bits codifican la parte fraccionaria del significante. En la Sección 4.3 veremos en detalle el concepto de significante.

4.1. Formatos

La norma IEEE 754 define 3 formatos estándar para números en punto flotante:

1. **Precisión simple**

   Los números en precisión simple tienen 32 bits con la siguiente distribución de bits:

[2]El término "significante (o "significando") ha reemplazado generalmente al término más antiguo "mantisa".

2. **Precisión doble**

   Los números en precisión doble tienen 64 bits con la siguiente distribución de bits:

3. **Precisión cuádruple**

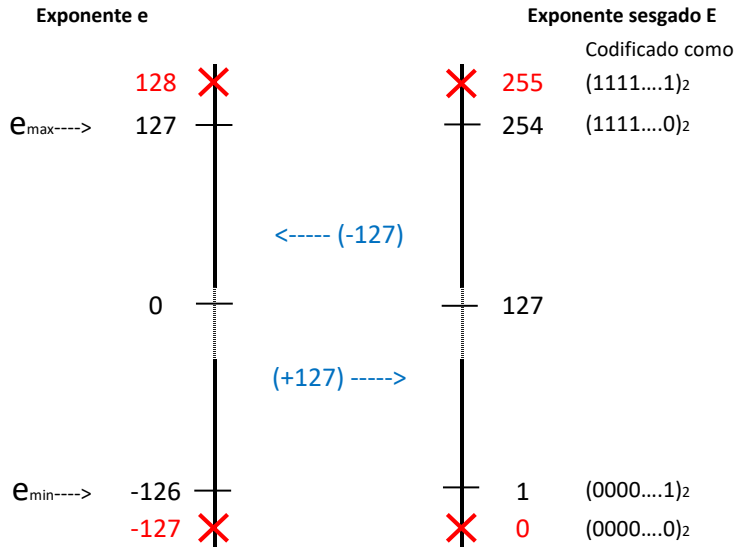
   Los números en precisión cuádruple tienen 128 bits con la siguiente distribución de bits:

4.2. Exponente sesgado

La norma IEEE 754 almacena un exponente con k bits, denominado exponente sesgado (E), cuyo rango es $[0, 2^k - 1]$ y que se relaciona con el exponente e en la Ecuación (1) de la siguiente manera:

$$\boxed{e = E - sesgo}.$$

El valor del exponente sesgado E es siempre positivo y de esta manera el exponente e puede adoptar tanto valores positivos como negativos. En la siguiente figura se muestra esta idea para el formato simple precisión:

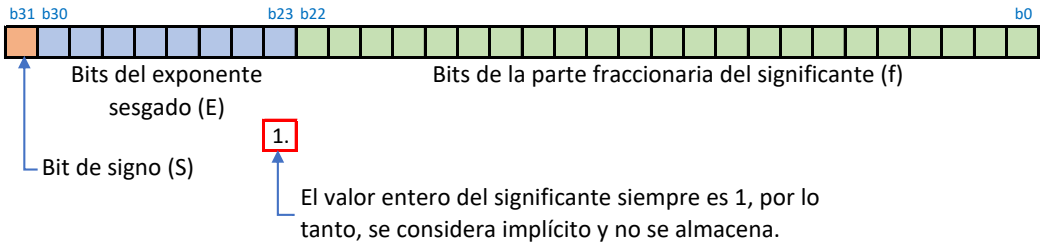


Los valores mínimos ($E = 0$) y máximo del exponente sesgado ($E = 255$, $E = 2047$ y $E = 32767$, según el formato) no se utilizan para números normalizados sino que tienen usos especiales como se verá más adelante. Por lo tanto, los valores mínimos y máximos de exponente para números normalizados en la Ecuación (1) resultan:

- Precisión simple:
  - Cantidad de bits: $k = 8$
  - Sesgo: $2^k/2 - 1 = 127$
  - $e_{min} = 1 - 127 = -126$
  - $e_{max} = 254 - 127 = 127$
- Precisión doble:
  - Cantidad de bits: $k = 11$
  - Sesgo: $2^k/2 - 1 = 1023$
  - $e_{min} = 1 - 1023 = -1022$
  - $e_{max} = 2046 - 1023 = 1023$
- Precisión cuádruple:
  - Cantidad de bits: $k = 15$
  - Sesgo: $2^k/2 - 1 = 16383$
  - $e_{min} = 1 - 16383 = -16382$
  - $e_{max} = 32766 - 16383 = 16383$

4.3. Significante

La definición de notación científica presentada en la Sección 3.1 indica que el significante se compone de un bit igual a uno (parte entera), seguido de un punto y luego el resto de los bits del significante (23, 52 o 112 bits, según el formato), que constituyen la parte fraccionaria. Entonces, ese bit, que siempre es igual a 1, no necesita ser almacenado, ya que se puede asumir su presencia. La siguiente figura muestra esta idea para el formato simple precisión:



Por lo tanto, la representación para números normalizados se puede formalizar de la siguiente manera:

$$\boxed{N = (-1)^s \times (1.f) \times 2^e},$$

con $e_{min} \leq e \leq e_{max}$, donde $(1.f)$ representa el significante de manera tal que:

$$(1.f) = (1.b_{-n} \cdots b_0)_2 = 1 + b_{-n} \times 2^{-1} + \ldots + b_0 \times 2^{-n},$$

donde n es la cantidad de bits en la parte fraccionaria del significante.

Si todos los bits del significante son ceros, su magnitud es 1. En cambio, si todos los bits son unos, el valor resultante es ligeramente inferior a 2:

$$1 \leq (1.f) < 2 - 2^{-n}.$$

Observación

Notar que esta normalización en principio deja sin representación al cero. Sin embargo, en la Sección 4.8 veremos una de las excepciones a esta regla y que por lo tanto el número cero se puede representar como caso especial.

4.4. Conversión de decimal a IEEE 754

A continuación mostramos el procedimiento de conversión de decimal a IEEE 754 mediante un ejemplo en simple precisión. Para otras precisiones el procedimiento es análogo con los cambios necesarios en el sesgo y el número de bits.

Ejemplo

Se quiere convertir el número decimal $N = (2.625)_{10}$ a IEEE 754 simple precisión. El procedimiento se compone de los siguientes pasos:

1. *Convertir la parte entera a binario:* $(2)_{10} = (10)_2$

2. *Convertir la parte fraccional a binario:*

$$\begin{array}{lcll} 0.625 \times 2 & = & \boldsymbol{1}.25 & \longrightarrow b_{-1} = 1 \\ 0.25 \times 2 & = & \boldsymbol{0}.5 & \longrightarrow b_{-2} = 0 \\ 0.5 \times 2 & = & \boldsymbol{1}.0 & \longrightarrow b_{-3} = 1 \end{array}$$

Por lo tanto:

$$(0.625)_{10} = (0.101)_2$$

3. *Ya tenemos el número convertido a binario:*

$$N = (2.625)_{10} = (10.101)_2$$

4. *Agregar exponente:* $N = (10.101)_2 = (10.101)_2 \times 2^0$

5. *Normalizar según lo visto previamente:*

$$N = (10.101)_2 \times 2^0 = (1.0101)_2 \times 2^1$$

6. *Por lo tanto, el significante resulta* $(1.01010000000000000000000)_2$ *donde el primer* 1 *y el punto están implícitos, es decir no se almacenan.*

7. *Corregir el exponente sumando el sesgo correspondiente (sesgo* $= 127$*):* $E = 1 + 127 = 128$

8. *Convertir el exponente a binario:* $E = (128)_{10} = (10000000)_2$

9. *El número es positivo. Por lo tanto el bit de signo es* $s = (0)_2$

10. *Finalmente, el número* $N = (2.625)_{10}$ *convertido a IEEE 754 simple precisión resulta:*

$$\underbrace{0}_{S} \quad \underbrace{10000000}_{\text{exponente sesgado}} \quad \underbrace{01010000000000000000000}_{\text{fracción del significante}}$$

A esta secuencia de bits la podemos expresar como $(0|10000000|01010000000000000000000)_2$*, utilizando el símbolo* | *como separador para los tres campos del números.*

11. *Es usual representar esta secuencia de bits en formato hexadecimal para tener una forma más comprimida de representación. Por lo tanto,* $N =$ `0x40280000`.

4.5. Conversión de IEEE 754 a decimal

A continuación mostramos el procedimiento de conversión de IEEE 754 a decimal mediante un ejemplo en simple precisión. Para otras precisiones el procedimiento es análogo con los cambios necesarios en el sesgo y el número de bits.

Ejemplo

Se quiere convertir el número expresado en IEEE 754 simple precisión $N = (0|10000010|10110100000000000000000)_2$ *a decimal. El procedimiento consiste en los siguientes pasos:*

1. *Separar el número en sus diferentes partes:*

Signo	*Exponente sesgado*	*Fracción del significante*
1 bit	*8 bits*	*23 bits*
0	10000010	10110100000000000000000

2. *El bit de signo es cero por lo tanto el número es positivo.*

3. *Convertir el exponente a decimal:*

$$E = (10000010)_2 = (130)_{10}$$

4. *Restar al exponente el sesgo correspondiente (sesgo* $= 127$*):*

$$e = 130 - 127 = 3$$

5. *Convertir el significante a decimal:*

$$(1.10110100000000000000000)_2 = 1 + 1 \times 2^{-1} + 1 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-6} = (1.703125)_{10}$$

6. *Finalmente, el número convertido a decimal resulta:*

$$N = (-1)^0 \times 1.703125 \times 2^3 = (13.625)_{10}$$

4.6. Características principales

En la Tabla 1 se muestra un resumen de las características de los números en punto flotante de la norma IEEE 754 en precisión simple, doble y cuádruple. La norma IEEE 754 va más allá de la simple definición de formatos, detallando cuestiones prácticas y procedimientos para que la aritmética en punto flotante produzca resultados uniformes y predecibles independientemente de la plataforma utilizada. Con este fin, el estándar IEEE 754 agrega a los números normalizados cuatro tipos numéricos que se describen en la Tabla 2 y se describen en las siguientes secciones.

4.7. Números denormalizados

El menor número normalizado en simple precisión es 1.0×2^{-126}. Antes de la definición del estándar si se presentaban problemas de "*underflow*" debía redondearse a cero. Los números denormalizados del estándar tienen una mejor aproximación al problema. Estos números tienen un exponente E igual a cero (no permitido para los números normalizados) y una fracción de 23, 52 o 112 bits, según la precisión, distinta de cero. El "1" implícito del significante es cero para los números denormalizados. Por lo tanto, el significante de los números denormalizados es mayor a cero y menor a uno. Por lo tanto, la representación para números denormalizados se puede formalizar de la siguiente manera:

$$N = (-1)^s \times (0.f) \times 2^{e_{min}},$$

donde $e_{min} = -126$ para simple precisión y $e_{min} = -1022$ para doble precisión.

De esta manera se pueden representar números más pequeños que los números normalizados. Por ejemplo, trabajando en simple precisión el rango que cubren los números desnormalizados (que se suma al rango de los números normalizados) es $[2^{-23} \times 2^{-126}, (1 - 2^{-23}) \times 2^{-126}]$. En la Figura 1 se observa el desbordamiento a cero gradual ("*gradual underflow*") introduciendo los números de punto flotante desnormalizados. Un caso especial dentro del conjunto de los números denormalizados es el número cero, el cual tiene un patrón de bits de todo ceros: el bit de signo es 0, el campo de exponente es todo ceros (lo que indica un valor denormalizado) y el campo de fracción es todo ceros. Curiosamente, cuando el bit de signo es 1, pero los demás campos son todos ceros, obtenemos el valor -0.0. Con el formato de punto flotante IEEE, los valores -0.0 y $+0.0$ se consideran diferentes en algunos aspectos e iguales en otros.

Tabla 1: Características de los números en punto flotante (Norma IEEE 754).

	Precisión Simple	Precisión Doble	Precisión Cuádruple
Bits de signo	1	1	1
Bits de exponente	8	11	15
Bits de fracción	23	52	112
Bits totales	32	64	128
Sesgo del exponente	+127	+1023	+16383
Rango del exponente	[-126,127]	[-1022,1023]	[-16382,16383]
Valor más chico (normalizado)	2^{-126}	2^{-1022}	2^{-16382}
Valor más grande (normalizado)	$\approx 2^{128}$	$\approx 2^{1024}$	$\approx 2^{16382}$
Rango decimal	$[\approx 10^{-38}, \approx 10^{38}]$	$[\approx 10^{-308}, \approx 10^{308}]$	$[\approx 10^{-4932}, \approx 10^{4932}]$
Valor más chico (desnormalizado)	$2^{-126}.2^{-23} \cong 10^{-45}$	$2^{-1022}.2^{-52} \cong 10^{-324}$	$2^{-16382}.2^{-112} \cong 10^{-4966}$

Tabla 2: Tipos numéricos del standard IEEE 754

Signo	Exponente	Fracción	Representa	Denominación
$s = 0/1$	$e = e_{min} - 1$ $E = (00\ldots0)_2$	$f = 0$	± 0	Ceros
		$f \neq 0$	$(-1)^s \times (0.f) \times 2^{e_{min}}$	Núm. denormal.
	$e_{min} \leq e \leq e_{max}$ $E = (\ldots\ldots)_2$	$0 \leq f < 1$	$(-1)^s \times (1.f) \times 2^e$	Núm. normaliz.
	$e = e_{max} + 1$ $E = (11\ldots1)_2$	$f = 0$	$\pm\infty$	Infinitos
		$f \neq 0$	$\pm$ NaN	*Not a Number*

Observación

Notar que en la representación de los números denormalizados el exponente en la representación es $e = e_{min}$ y no $e = e_{min} - 1$. Es decir, el exponente del número representado es $e = -126$ para simple precisión ($e = -1022$ para doble precisión) a pesar de que en el registro realmente se almacena el exponente sesgado $E = (0000\ 0000)_2$ que correspondería al exponente $e = e_{min} - 1 = -127$ ($e = -1023$ para doble precisión). De lo contrario quedaría un "hueco" entre 2^{-127} y 2^{-126}.

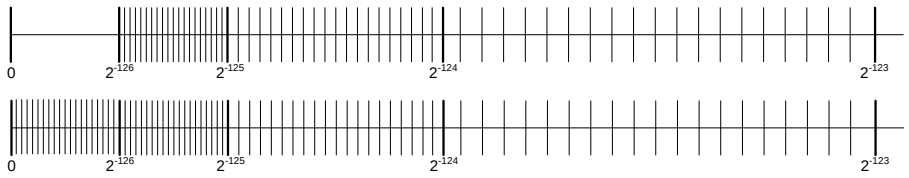


Figura 1: Desbordamiento a cero gradual mediante números de punto flotante denormalizados. Ejemplo con simple precisión: el rango entre 0 y 2^{-126} se divide en 2^{23} valores.

4.8. Ceros

El cero se representa por el exponente $e_{min} - 1$ (es decir, se codifica con un exponente sesgado con todos ceros: $E = (0\ldots0)_2$) y un significante cero (es decir, todos ceros en la parte fraccionaria). Dado que el bit de signo puede tomar dos valores diferentes, hay dos ceros, +0 y –0.

4.9. Infinitos

El estándar IEEE 754 provee una representación para los números infinitos ($\pm\infty$). Estos números pueden ser utilizados como operador.

Ejemplos

A continuación se enumeran algunas operaciones que dan como resultado un valor de tipo infinito:

1. $N + (+\infty) = +\infty$
2. $N - (+\infty) = -\infty$
3. $N + (-\infty) = -\infty$
4. $N - (-\infty) = +\infty$
5. $(+\infty) + (+\infty) = +\infty$
6. $(-\infty) + (-\infty) = -\infty$
7. $(-\infty) - (+\infty) = -\infty$
8. $(+\infty) - (-\infty) = +\infty$

4.10. Formato NaN

El formato NaN (*Not a Number*) es una entidad simbólica utilizada en punto flotante. Sirve para representar valores de variables no inicializadas y tratamiento de tipo aritmético que no están contempladas en el estándar.

Ejemplos

A continuación se enumeran algunas operaciones que dan como resultado un valor de tipo NaN:

1. $0/0 = NaN$
2. $\pm\infty/\pm\infty = NaN$
3. $0 * \pm\infty = NaN$

4. $\infty - \infty = NaN$

Ejemplo

```
// Ejemplo usando la cabecera math.h
#include <stdio.h>
#include <math.h>

int main(){
    float a=NAN, b=INFINITY;
    printf("%f %f\n",a,b);
    return 0;
}
```

Ejemplo

Usando la cabecera `ieee754.h`*:*

```
#include <stdio.h>
#include <math.h>
#include <ieee754.h>
int main(){
    union ieee754_float myfloat;
    myfloat.f = INFINITY;
    printf("Signo: %x\n", myfloat.ieee.negative);
    printf("Exponente: %x\n", myfloat.ieee.exponent);
    printf("Mantisa: %x\n", myfloat.ieee.mantissa);

    union ieee754_double mydouble;
    mydouble.d = NAN;
    printf("Signo: %x\n", mydouble.ieee.negative);
    printf("Exponente: %x\n", mydouble.ieee.exponent);
    printf("Mantisa (b51 a b32): %x\n", mydouble.ieee.mantissa0);
    printf("Mantisa (b31 a b0): %x\n", mydouble.ieee.mantissa1);
    return 0;
```

```
}
```

5. Operaciones con números en punto flotante

5.1. Suma/resta

Para sumar o restar números en punto flotante hay que igualar los dos exponentes desplazando el significante. Luego, se pueden sumar o restar los significantes. Concretamente, la secuencia de acciones que debe realizarse para sumar o restar es la siguiente:

1. Verificar `NaN`: Si alguno de los operandos es `NaN`, el resultados es `NaN`.
2. Verificar `INF-INF`: En este caso el resultado es `NaN`.
3. Verificar `+INF`: Si alguno de los operandos es `+INF`, el resultado es `+INF`.
4. Verificar `-INF`: Si alguno de los operandos es `-INF`, el resultado es `-INF`.
5. Chequear por ceros: Si alguno de los operandos es 0, el resultado es el otro operando ($X + 0 = X$).
6. Verificar diferencia de exponentes: Si la diferencia entre los exponentes de los argumentos es mayor a los bits del significante, el resultado es el mayor argumento ya que el argumento menor no logra afectar el resultado.
7. Igualar exponentes (si es necesario): Para poder realizar la suma/resta los exponentes deben ser iguales. Se debe desplazar hacia la derecha el punto del significante con menor exponente la cantidad de bits necesarios hasta igualar los exponentes.
8. Sumar o restar los significantes.
9. Normalizar el resultado (si es necesario): El resultado se debe volver a normalizar teniendo en cuenta que que el significante debe tener la forma $1.b_{-1} \cdots$. En caso de tener que ajustar el punto, hay que ajustar el exponente del resultado.

Ejemplo

Supongamos que queremos hacer la suma $S = 123 + 456 = 579$. *El número* 123 *se puede expresar como* $(-1)^0 \times 2^6 \times 1.921875$ *que a su vez equivale a la secuencia de bits en formato IEEE 754* $(0|10000101|11101100000000000000000)_2$. *Por otra parte, el número 456 se puede expresar como* $(-1)^0 \times 2^8 \times 1.78125$ *que equivale a la secuencia de bits* $(0|10000111|11001000000000000000000)_2$.

Como ambos números tienen diferentes exponentes lo primero que hay que hacer es igualar los exponentes (el menor hacia el mayor) ajustando al mismo tiempo los significantes. En este ejemplo tenemos que correr la coma dos lugares hacia a izquierda. Luego podemos hacer la suma de los significantes:

$$\begin{array}{rr} & 1.11001000000000000000000 \\ + & 0.01111011000000000000000 \\ \hline & 10.01000011000000000000000 \end{array}$$

Una vez realizada la suma debemos ajustar la coma y el exponente para que el significante quede normalizado:

$$Significante = (1.00100001100000000000000)_2$$

y el exponente de la suma resulta $e = 9$. *Por lo tanto, el resultado es:*

$$S = (-1)^0 \times 2^9 \times 1.130859375 = 579$$

que equivale a la secuencia de bits $(0|10001000|00100001100000000000000)_2$. *Este resultado también lo podemos expresar usando notación hexadecimal para que quede más compacto:* `0x4410c000`.

5.2. Multiplicación

Para multiplicar, se multiplican los significantes y se suman los exponentes. Concretamente, la secuencia de acciones que debe realizarse para multiplicar es la siguiente:

1. Verificar `NaN`: si algunos de los operandos en `NaN`, el resultado es `NaN`.
2. Verificar `0×INF`: si un operando es 0 y el otro es `INF`, el resultado es `NaN`.
3. Verificar por ceros: si alguno de los operando es cero, el resultado es cero.

4. Sumar los exponentes.

5. Multiplicar los significantes.

6. Normalizar el producto (si es necesario).

Ejemplo

Supongamos que queremos hacer la multiplicación $M = 120 \times 12 = 1440$ *en norma IEEE 754. Primero convertimos ambos números:*

$$\begin{aligned}(120)_{10} &= 2^6 \times 1.875 &&= (0|10000101|11100000000000000000000)_2\\ (12)_{10} &= 2^3 \times 1.5 &&= (0|10000010|10000000000000000000000)_2\end{aligned}$$

Luego podemos multiplicar los significantes y sumar los exponentes:

$$M = 2^9 \times 2.8125 \tag{3}$$

Como el significante en (3) no está normalizado, es necesario normalizarlo aumentando en uno el exponente y corriendo un lugar el punto del significante:

$$(2.8125)_{10} = (10.11010000000000000000000)_2$$

Notar que correr el punto binario un lugar hacia la izquierda equivale a dividir por dos su equivalente en decimal. Por lo tanto, el resultado de la multiplicación es:

$$M = 2^{10} \times 1.40625 = 1440$$

que equivale a la secuencia de bits $(0|10001001|01101000000000000000000)_2$ *o a* `0x44b40000` *en notación hexadecimal.*

5.3. División

Para dividir, se dividen los significantes y se restan los exponentes. Concretamente, la secuencia de acciones que debe realizarse para dividir es la siguiente:

1. Verificar `NaN`: Si algunos de los operandos en `NaN`, el resultado es `NaN`.

2. Verificar `0/0`: Si ambos operandos son ceros, el resultado es `NaN`.

3. Verificar `INF/INF`: Si ambos operandos son `INF`, el resultado es `NaN`.

4. Verificar `INF/X`: Si el operando dividendo es `INF`, el resultado es `INF`.

5. Verificar `X/INF`: Si el operando divisor es `INF`, el resultado es 0.

6. Chequear por ceros: Si el argumento divisor es `0`, el resultados es `INF` con el signo del dividendo. Si argumento dividendo es 0, el resultado es 0 ($0/X = 0$, con $X \neq 0$).

7. Restar los exponentes.

8. Dividir los significantes.

9. Normalizar (si es necesario).

Ejemplo

Supongamos que queremos hacer la división $D = N_1/N_2 = 96/15 = 6.4$. *Primero convertimos ambos números:*

$$\begin{aligned}(96)_{10} &= 2^6 \times 1.5 &&= (0|10000101|10000000000000000000000)_2\\ (15)_{10} &= 2^3 \times 1.875 &&= (0|10000010|11100000000000000000000)_2\end{aligned}$$

Luego podemos dividir los significantes y restar los exponentes:

$$D = 2^3 \times 0.8$$

Como el significante no está normalizado, es necesario normalizarlo disminuyendo en uno el exponente y corriendo un lugar el punto del significante. Notar que correr el punto binario un lugar hacia la derecha equivale a multiplicar por dos su equivalente en decimal. Por lo tanto, el resultado de la división es:

$$M = 2^2 \times 1.6 = (6.4)_{10}$$

En realidad, el número 6.4 *no tiene representación exacta y el número más cercano es* 6.400000095367431640625 *que equivale a la secuencia de bits* $(0|10000001|10011001100110011001101)_2$ *o a* `0x40cccccd` *en notación hexadecimal.*

5.4. Densidad de los números en punto flotante

Al contrario que con los números en punto fijo, en punto flotante la distribución en la recta numérica no es uniforme y a medida que nos alejamos del origen se van separando. Cómo se verá a continuación, existe un compromiso entre rango y precisión.

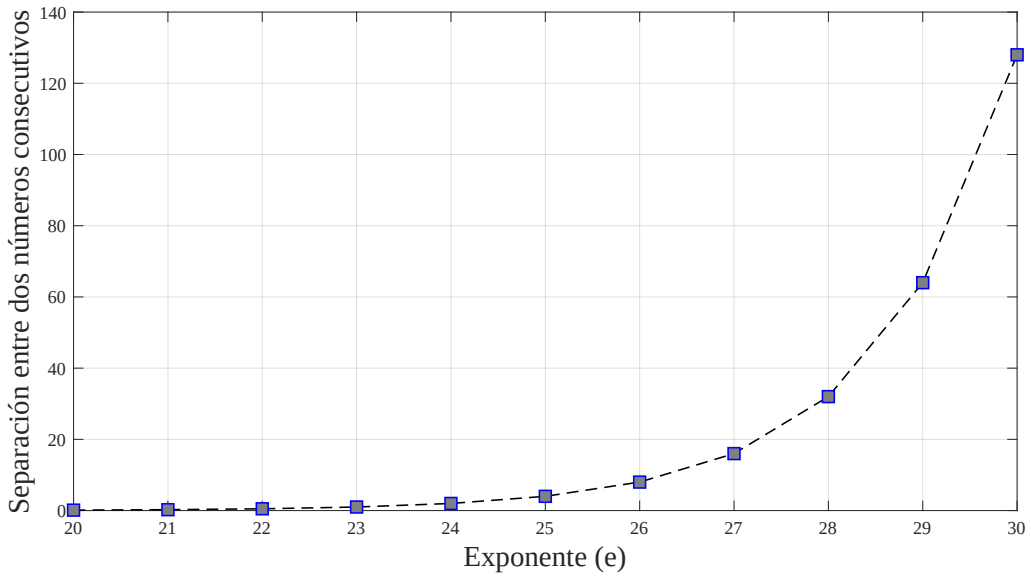


Figura 2: Evolución de la separación entre dos números consecutivos en punto flotante en función del exponente e.

Supongamos que P (el número de bits de la parte fraccionaria del significante) sea 23. En el intervalo $[1, 2)$ (con exponente $e = 0$) es posible representar 2^{23} números equiespaciados y separados con una distancia $1/2^{23}$. De modo análogo, en cualquier intervalo $[2^e, 2^{e+1})$ habrá 2^{23} números equiespaciados pero la densidad de este caso será $2^e/2^{23}$.

Por ejemplo, entre $2^{20} = 1048576$ y $2^{21} = 2097152$ hay $2^{23} = 8388608$ números pero el espaciado es de solo 0.125. De este modo se deriva una regla práctica que cuando es necesario comparar dos números en punto flotante relativamente grandes es preferible comparar la diferencia relativa entre esos dos números en lugar de las magnitudes absolutas de los mismos dado que la precisión es menor cuanto más grandes sean los números. En la Figura 2 se puede apreciar cómo aumenta la separación entre dos números consecutivos en función del exponente e en el rango $e = [20, 30]$.

5.5. Precauciones al operar con números en punto flotante

La alineación o ajuste se consigue desplazando a la derecha el número más pequeño (incrementando su exponente) o desplazando a la izquierda el más grande (decrementando su exponente). Dado que cualquiera de estas operaciones puede ocasionar que se pierdan dígitos, conviene desplazar el

número más pequeño ya que los dígitos que se pierden tienen una importancia relativa menor.

Ejemplo

Queremos sumar $S = N_1 + N_2$ *con* $N_1 = (1230.015625)_{10}$ *y* $N_2 = (4.56)_{10}$ *en formato IEEE 754 simple precisión. Convirtiendo los números* N_1 *y* N_2 *obtenemos:*

$$\begin{aligned} N_1 = &\quad (0|10001001|00110011100000010000000)_2 \\ N_2 = &\quad (0|10000001|00100011110101110000101)_2 \end{aligned}$$

El exponente del primer número es $e_1 = 10$ *mientras que el exponente del segundo número es* $e_2 = 2$*. Por lo tanto, tenemos que ajustar los exponentes corriendo el punto del segundo número 8 posiciones. El problema es que en este caso perdemos los 8 bits menos significativos del significante de* N_2*. Como conclusión, el error cometido es* $2^2 \times (2^{-16} + 2^{-21} + 2^{-23}) = 0.6342 \times 10^{-4}$*. Por el contrario, si hubiéramos ajustado el exponente de* N_1 *hacia* N_2 *el error cometido hubiera sido* $2^{10} \times 2^{-16} = 0.1563 \times 10^{-1}$*.*

Como resultado de lo anterior se pueden establecer los siguientes enunciados:

1. Se tiene más precisión si se suman números de magnitud similares.
2. Al restar dos números muy próximos el resultado que se obtendrá puede ser todo error de redondeo.
3. El orden de evaluación puede afectar la precisión: $A \oplus (B \oplus C)$ puede ser distinto a $(A \oplus B) \oplus C$[3].
4. Cuando se restan dos números del mismo signo o se suman dos números con signo opuesto, la precisión del resultado puede ser menor que la precisión disponible en el formato.
5. Cuando se realiza una cadena de cálculos tratar de realizar primero los productos y cocientes. En general: $x \otimes (y \oplus z) \neq x \otimes y \oplus x \otimes z$
6. Cuando se multiplica y divide un conjunto de números, tratar de juntarlos de manera que se multipliquen números grandes y pequeños y por otro lado se dividan números de magnitud similares.

[3]La notación $\oplus$ y $\otimes$ se emplea para notar la suma y el producto en punto flotante, respectivamente.

7. Cuando se comparan dos números en punto flotante, siempre comparar con respecto a un valor de tolerancia pequeño. Por ejemplo, en lugar de comparar `x=y`, realizar la evaluación `IF ABS((x-y)/y)) <= tol`. De todas maneras, además hay que tener precaución con los casos límites: $x = y = 0$, $y = 0$.

5.6. Épsilon de máquina

Una cantidad de gran importancia es la denominada **Épsilon de máquina** (ϵ_m), que representa la distancia entre el número 1 y el siguiente número mayor en punto flotante. Es decir, ϵ_m es el menor número tal que $1 + \epsilon_m > 1$. Por lo tanto, teniendo en cuenta la estructura d elos números en formato punto flotante se puede calcular que:

$$\epsilon_m = 2^{-t},$$

donde t es la cantidad de dígitos en la parte fraccionaria del significante. A esta expresión se puede llegar teniendo en cuenta que $(1)_{10} = 2^0$ y el siguiente número representables es $(1 + 2^{-t}) \times 2^0$. Por lo tanto, para simple precisión es $\epsilon_m = 2^{-23}$ y para doble precisión es $\epsilon_m = 2^{-52}$.

¿Por qué es importante el Épsilon de máquina? La respuesta es porque indica la precisión para un determinando formato. En primer lugar, hay que tener en cuenta que la distancia entre un número representable y el siguiente es variable a lo largo de la recta real como hemos visto en la Sección 5.4. Por lo tanto, el error por representación aumenta a medida que nos alejamos de cero. Sin embargo, el error relativo de representación (ρ_r) se mantiene constante y es igual a la mitad del Épsilon de máquina. Es decir, teniendo en cuenta que en cualquier intervalo $[2^e, 2^{e+1})$ habrá 2^t números equiespaciados cuya distancia entre un número y el siguiente es $2^{-t} \times 2^e$, si x es el número real a representar y $fl(x)$ su valor efectivamente representado:

$$\rho_r = \left| \frac{fl(x) - x}{x} \right| \leq \frac{1}{2} \left(\frac{2^{-t} \times 2^e}{2^e} \right) = \frac{1}{2} \epsilon_m$$

La Figura 3 ilustra esta idea, donde se puede observar como de manera general un número real quedará ubicado entre dos números representables con un error de representación acotado por la mitad de la distancia entre dichos números.

Ejemplo

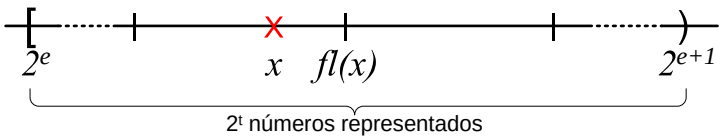


Figura 3: Error de representación de números reales. Se muestra un fragmento de la recta real para un determinado exponente e, es decir, entre 2^e y 2^{e+1} sin incluir. El verdadero valor x se aproxima a $fl(x)$, que es el valor más cercano dentro del conjunto $\mathbb{F}$.

Empleando el método de conversión se puede llegar a que $(3.45)_{10}$ *no tiene representación exacta. Su valor aproximado en el formato simple precisión es* $(3.450000047683715820312 5)_{10}$ *y el error cometido por representación es aproximadamente* 4.77×10^{-8}*. Por lo tanto, el error relativo es aproximadamente* 1.38×10^{-08}*, el cual está acotado por* $\frac{1}{2}\epsilon_m = 5.96 \times 10^{-08}$.

Código

Este es un código para calcular el Épsilon de máquina tanto para simple precisión como para doble precisión:

```
#include <stdio.h>
int main(){
    float x = 1.0;
    double y = 1.0;
    int n = 0;

    while ((float)1.0 + (x * (float)0.5) > (float)1.0){
        ++n;
        x *= 0.5;
    }
    printf("Float: Épsilon de máquina: 2^(-\%d)=\%G\n",n,x);

    n = 0;
    while (1.0 + (y * 0.5) > 1.0){
        ++n;
        y *= 0.5;
    }
    printf("Float: Épsilon de máquina: 2^(-\%d)=\%G\n",n,y);
```

```
    return 0;
}
```

Referencias

[1] Mano, M.M., *Computer system architecture*, Prentice-Hall, 1993.

[2] Hyde, R., *The art of assembly language*, No Starch Pr, 2003.

[3] Goldberg, D, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys (CSUR), **(23)**, 5-48, 1991.

[4] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, 2008.

[5] Carter, P., *PC Assembly Language*, 2006.