

Números reales

Los lenguajes de programación también permiten manejar números con fracciones —*números reales*, en matemáticas:

3.14159265...

2.71828...

0.000000001 (lo que dura un nanosegundo, en segundos)

... o números más grandes que el que puede ser representado mediante un entero con signo en 32 (o 16 o 64) bits:

3,155,760,000 (el número de segundos en un siglo)

Los lenguajes de programación también permiten manejar números con fracciones — números reales, en matemáticas:

3.14159265...

2.71828...

0.000000001 = 1.0×10^{-9}

notación
científica
normalizada

... o números más grandes que el que puede representarse mediante un entero con signo en 32 (o 16 o 64) bits:

3,155,760,000 = 3.15576×10^9

Notación científica :

un único dígito a la izquierda del punto decimal

... normalizada :

ese único dígito es $\neq 0$

También podemos escribir números binarios en notación científica:

p.ej., 1.0×2^{-1}

La aritmética computacional correspondiente se llama *de punto flotante*:

maneja números en los que la posición del punto decimal (o *punto binario*) no está fija

en el lenguaje C, el tipo de datos correspondiente se llama **float**

Los números son representados a partir de un único dígito $\neq 0$ a la izquierda del punto:

$$1.\text{xxxxxxxx}_2 \times 2^{\text{yyy}}$$

Ventajas:

simplifica el intercambio de datos

simplifica los algoritmos para la aritmética de punto flotante

aumenta la exactitud de los números que pueden almacenarse en una palabra —los 0s iniciales innecesarios son reemplazados por dígitos reales a la derecha del punto

Representación de números de punto flotante

Compromiso entre el tamaño de la fracción —o *mantisa*— y el tamaño del exponente:

el número total de bits es fijo → hay que quitarle un bit a la fracción para agregárselo al exponente, y viceversa

... es decir, entre *precisión* y *rango*:

a mayor tamaño de la fracción, mayor precisión de la fracción

a mayor tamaño del exponente, mayor rango de los números representables

P.ej., en el caso de 32 bits:

un bit para el *signo* del número (1 → negativo)

23 bits para la *fracción*

8 bits para el *exponente* (un número entero: positivo, negativo , o 0)

| 1 | 10000001 | 010000000000000000000000 |

signo
un bit

fracción
23 bits

exponente
8 bits

Representación *signo* y *magnitud*: el signo es un bit separado del resto del número

El valor de un número así representado es

$$(-1)^{\text{signo}} \times \text{fracción} \times 2^{\text{exponente}}$$

Propiedades de la representación

Gran rango de números representables:

aproximadamente, desde 2.0×10^{-38} hasta 2.0×10^{38}

Buena precisión: 2^{-22}

Ahora, además de overflow, puede ocurrir *underflow*:

overflow : cuando el exponente es demasiado grande para ser representado en 8 bits

underflow : cuando el exponente negativo es demasiado grande → el valor de la fracción es muy pequeño (por lo que 0 es una aproximación satisfactoria)

...

...

La recta de números reales queda dividida en 7 regiones (ver pizarra):

1: overflow negativo, 2: números negativos expresables, 3: underflow negativo, 4: cero, 5: underflow positivo, 6: números positivos expresables, y 7: overflow positivo

No todos los números reales en las regiones 2 y 6 pueden ser representados → redondeo

En la práctica, se usa el estándar *IEEE 754*

fracción:

23 bits —los 23 bits menos significativos (del 0 al 22) de la palabra
el 1 a la izquierda del punto es implícito → el *significante* tiene 24
bits

signo:

en el bit más significativo (bit 31) → facilita la comparación con 0

...

...

exponente:

8 bits —bits 23 al 30

inmediatamente a continuación del signo → facilita la ordenación usando instrucciones de comparación de números enteros

... siempre que todos los exponentes tengan el mismo signo

→ en lugar de representarlos como números enteros en complemento de 2

... se los representa como números sin signo desfasados (*biased*) en 127

La forma de un número en punto flotante en el estándar IEEE 754 es

$$(-1)^{\text{signo}} \times (1+\text{fracción}) \times 2^{\text{exponente}-127}$$

P.ej., ¿cuál es el número decimal representado por el siguiente *float*?

11000000101000000000000000000000000000000

Identifiquemos sus partes:

| 1 | 1000001 | 010000000000000000000000000000 |

signo → 1

exponente → 129

fracción → $1 \times 2^{-2} = 1/4 = 0.25$

Por lo tanto, el número es (ver fórmula en diapositiva anterior):

$$\begin{aligned} (-1)^1 \times (1 + 0.25) \times 2^{129-127} &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= \mathbf{-5.0} \end{aligned}$$

P.ej., ¿cuál es la representación en el estándar IEEE 754 de -0.75_{10} ?

$$-0.75_{10} = -3/4_{10} = -3/2^2_{10} = -11_2/2^2_{10} = -0.11_2$$

$$= -0.11_2 \times 2^0 \quad (\text{en notación científica})$$

$$= -1.1_2 \times 2^{-1} \quad (\text{en notación científica normalizada})$$

Por lo tanto, *signo* = 1

significante = 1.1 \rightarrow *fracción* = .1

exponente = $-1 + 127 = 126$

... y la representación es

1 01111110 10000000000000000000000000

El estándar IEEE 754 tiene también un formato de 64 bits → *precisión doble*:

exponente de 11 bits

fracción de 52 bits → *significante* de 53 bits

Rango de números representables:

10^{-38} ($\approx 2^{-126}$) a 10^{38} ($\approx 2^{128}$), en precisión simple

10^{-308} ($\approx 2^{-1022}$) a 10^{308} ($\approx 2^{1024}$), en precisión doble

Combinaciones especiales de exponente y fracción:

0 : *exponente* = 0 y *fracción* = 0

número no normalizado : *exponente* = 0 y *fracción* ≠ 0

infinito : *exponente* = 255 (o 2047) y *fracción* = 0

NaN (not a number): *exponente* = 255 (o 2047) y *fracción* ≠ 0

Ejemplo de suma de números de punto flotante

$$9.999 \times 10^1 + 1.610 \times 10^{-1} = \dots$$

suponemos significantes de 4 dígitos, y exponentes de dos

Primero, hay que alinear el punto decimal del número con el menor exponente:

$$1.610 \times 10^{-1} = 0.1610 \times 10^0 = 0.01610 \times 10^1 \rightarrow \mathbf{0.016 \times 10^1}$$

es decir, hicimos *shift* a la derecha del significante, aumentando el exponente en 1 cada vez, hasta quedar con el exponente correcto

... y recordamos que solo podemos representar 4 dígitos

...

...

Luego, sumamos los significantes, y normalizamos y redondeamos el resultado:

$$9.999 + 0.016 = 10.015 \rightarrow 10.015 \times 10^1 = 1.0015 \times 10^2 \rightarrow 1.002 \\ \times 10^2$$

al normalizar, aumentando o disminuyendo el exponente, hay que revisar si se produce overflow o underflow

... y al redondear, hay que revisar si el resultado se mantiene normalizado, o si es necesario normalizarlo de nuevo

Algoritmo de suma de punto flotante

1. comparar los exponentes; “shift” el número más pequeño a la derecha hasta que su exponente sea igual al exponente más grande
2. sumar los significantes
3. normalizar la suma, ya sea “shifting” a la derecha e incrementando el exponente, o “shifting” a la izquierda y decrementando el exponente

¿“overflow” o “underflow”? → excepción
4. redondear el significante al número apropiado de bits

¿está normalizado? → terminar

volver al paso 3

Algoritmo de multiplicación de punto flotante

1. sumar los exponentes desfasados y restar el desfase a la suma para obtener el nuevo exponente desfasado
2. multiplicar los significantes
3. normalizar el producto si es necesario, “shifting” a la derecha e incrementando el exponente
¿“overflow” o “underflow”? → excepción
4. redondear el significante al número apropiado de bits
¿está normalizado? → ir al paso 5
volver al paso 3
5. poner el signo del producto en positivo si los signos de los operandos son iguales; en negativo, en caso contrario
terminar

Ejemplo de multiplicación de números de punto flotante

$$1.000 \times 2^{-1} \times -1.110 \times 2^{-2} = \dots$$

Primero, sumamos los exponentes:

$$\text{sin desfasarlos} \rightarrow -1 + (-2) = -3$$

$$\text{o desfasándolos} \rightarrow (-1+127) + (-2+127) - 127 = -3 + 127 = 124$$

Luego, multiplicamos los significantes y dejamos el producto en 4 bits:

$$1.000 \times 1.110 = 1.110000 \rightarrow 1.110000 \times 2^{-3} \rightarrow 1.110 \times 2^{-3}$$

...

...

Revisamos que el producto esté normalizado → lo está

... y que el exponente no haya producido overflow o underflow:

$127 \geq -3 \geq -126 \rightarrow$ no se produjo ninguno (con desfase: $254 \geq 124 \geq 1$)

Redondeamos el producto, sin efecto en este caso

... y finalmente hacemos que el signo del producto sea negativo: –

1.110 × 2⁻³