Aritmética de Ponto Flutuante

texto redigido por Márcia A Gomes–Ruggiero

Entre 1970 e 1980 um grupo formado por cientistas e engenheiros de diferentes empresas de computação realizou um trabalho intenso na tentativa de encontrar um padrão de representação dos números, que deveria ser adotado por todas as indústrias na construção de seus computadores. A necessidade desta padronização tinha como principal objetivo uniformizar os resultados obtidos por um mesmo programa computacional executado em diferentes máquinas, [1].

Esta discussão teve início em 1976, e este grupo de trabalho ficou conhecido como `IEEEp754`, pois foi organizado pelo Institute for Electrical and Electronics Engineers `IEEE`. Entre os fabricantes estavam Apple, Zilog, DEC, Intel, Hewlett-Packard, Motorola e National Semiconductor. O prof. William Kahan liderava o grupo de cientistas e pelo trabalho desenvolvido neste projeto, recebeu o prêmio Turing Prize em 1989, [1].

Este projeto tinha como metas principais:

- especificar como representar os números em precisão simples e dupla;
- padronizar o arredondamento nas operações neste sistema;
- estabelecer critérios para padronizar situações como divisão por zero, operações envolvendo infinito.

Em 1985 o resultado deste trabalho foi publicado e ficou conhecido oficialmente como `ANSI/IEEE Std 754-1985`, [2].

Representação em precisão simples e dupla

A base numérica no padrão `IEEE754` é a binária. Neste padrão são adotados dois formatos para representação de números: precisão simples e precisão dupla. (Na base binária, um dígito binário é denominado `bit` e um `byte` é um conjunto de 8 bits).

Ficou estabelecido que no padrão `IEEE754`, em precisão simples, um número real seria representado por 32 bits, (4 bytes), sendo que:

1 bit é reservado para o sinal do número (positivo ou negativo);
8 bits são reservados para o expoente da base, que é um número inteiro;
23 bits são reservados para a mantissa:

$\pm$	$e_1e_2 \ldots e_8$	$d_1d_2 \ldots d_{23}$

Sobre a representação do expoente e:
os 8 bits reservados para representar o expoente devem conter também a informação do sinal deste expoente. No padrão `IEEE754`, a sequência de 8 bits armazena o número $s = e + 127$. Desta forma, evita–se o teste sobre o valor do bit para saber se o número é positivo ou negativo e para recuperar o expoente, é realizada a operação $e = s - 127$.

Alguns exemplos: (lembrando que $(127)_{10} = (1111111)_2$):
$e = (1)_{10} = (1)_2$ é armazenado como: $(1111111)_2 + (1)_2 = (10000000)_2$;
$e = (3)_{10} = (11)_2$ é armazenado como: $(1111111)_2 + (11)_2 = (10000010)_2$;
$e = (-3)_{10} = -(11)_2$ é armazenado como: $(1111111)_2 - (11)_2 = (01111100)_2$;
$e = (52)_{10} = (110100)_2$ é armazenado como: $(1111111)_2 + (110100)_2 = (10110011)_2$.

Para obter a forma como o expoente será armazenado podemos também trabalhar na base 10 e depois converter o resultado final. Por exemplo, se $e = (52)_{10}$ iremos armazenar $(127)_{10} + (52)_{10} = (179)_{10} = (10110011)_2$.

É importante destacar que as sequências de bits para o expoente: (00000000) e (11111111) são reservadas para representar o zero, e infinito ou ocorrência de erro (NaN: not a number) respectivamente.

O maior expoente é representado pela sequência $(11111110)_2$ que, na base 10, representa o número $(256 - 2)_{10} = (254)_{10}$. Então o maior expoente é: $127 + e = 254 \Rightarrow e = 254 - 127 = 127$.

O menor expoente é representado pela sequência $(00000001)_2 = (1)_{10}$. Daí temos que o menor expoente é: $127 + e = 1 \Rightarrow e = 1 - 127 = -126$.

Considerando agora a representação da mantissa. Vimos que no sistema normalizado $d_1 \neq 0$. Dado que a base é dois, teremos que o primeiro dígito no sistema normalizado será sempre igual a 1 e por esta razão não é armazenado. É o denominado *bit* escondido. Esta normalização permite um ganho na precisão, pois podemos considerar que a mantissa é armazenada em 24 *bits*.

Por exemplo, o número $(1/8)_{10} = (0.125)_{10} = (0.001)_2$ será assim armazenado:

0	01111100	0000...0

pois, o expoente -3 é representando pela sequência $(01111100)_2$ conforme vimos anteriormente, e a mantissa tem apenas um dígito significativo, que é armazenado no *bit* escondido, pois o sistema é normalizado. Logo, os 23 bits para a mantissa ficam iguais a zero.

Considerando agora o número:
$(0.1)_{10} = (0.0001100110011\overline{0011}\ldots)_2 = (1.10011001\overline{10011}\ldots)_2 * 2^{-4}$.
Temos que o expoente $e = (-4)_{10} = (100)_2$ será assim representado: $(127)_{10} + (-4)_{10} = (123)_{10} = (01111011)_2$. O número $(0.1)_{10}$ será armazenado como:

0	01111011	10011001100110011001100

A menor mantissa é 0.1 e o menor expoente $(00000001)_2$, então o menor número positivo que pode ser armazenado é $(1.000000\ldots)_2 * 2^{-126} = 2^{-126} \sim 1.2 * 10^{-38}$. No formato `IEEE754` teremos este número assim representado:

0	00000001	00000000000000000000000

O maior número positivo terá como mantissa um número com 24 bits iguais a um e como expoente $(11111110)_2 = (1.1111\ldots11)_2 * 2^{127} = (2 - 2^{-23}) * 2^{127} \approx 2^{128} \approx 3.4 * 10^{38}$. No formato `IEEE754` teremos este número assim representado:

0	11111110	11111111111111111111111

Qualquer resultado acima de $+3.4 * 10^{38}$ ou abaixo de $-3.4 * 10^{38}$ resultará em $\pm\infty$ (*overflow*) e serão representados por:

1	11111111	00000000000000000000000

0	11111111	00000000000000000000000

O zero é representado com as sequências de bits todos nulos tanto para o expoente quanto para a mantissa:

0	00000000	00000000000000000000000

Se a sequência de *bits* para o expoente for nula e a sequência de *bits* para a mantissa for não nula então, temos a ocorrência de números menores que $2^{-126} \approx 1.2 * 10^{-38}$, que não estarão na forma normalizada, isto é, o primeiro dígito da mantissa não será igual a 1. Por exemplo:

0	00000000	10000000000000000000000

representa $(2^{-1} * 2^{-126}) = 2^{-127}$ e

0	00000000	00000000000000000000001

representa $(2^{-23} * 2^{-126}) = 2^{-149}$.

Se a sequência de *bits* para o expoente for composta por todos dígitos iguais a um e a sequência de *bits* para a mantissa for não nula então temos o ocorrência dos denominados *NaN: Not a Number*, que representam resultados de expressões inválidas como:
$0 * \infty$;
$0/0$;
∞/∞;
$\infty - \infty$.

Há um consenso que certas expressões, ainda que envolvam ∞ ou zero, tenham um resultado plausível, tais como:
$z * 0 = 0$;
$z/0 = +\infty$, se $z > 0$;

$z/0 = -\infty$, se $z < 0$;
$z * \infty = \infty$.
$\infty + \infty = \infty$.

Observamos que a justificativa para a convenção em se adotar NaN para $\infty - \infty$ segue do estudo de limite de sequências. Por exemplo, se duas sequências, z_k e w_k divergem para $+\infty$, então, a adição dos termos de sequência, resultará em outra sequência, $t_k = z_k + w_k$, que também diverge para $+\infty$. Porém, se subtrairmos os termos das sequências, $t_k = z_k - w_k$, nada poderemos afirmar sobre a sequência resultante, pois o seu limite dependerá da rapidez de convergência de cada sequência envolvida.

Em alguns casos, pode ser necessário trabalhar com precisões maiores, ou ainda, pode ser necessário trabalhar com números bastante grandes ou muito pequenos. Para estes casos, é conveniente trabalhar com o sistema de precisão dupla, que permite uma precisão maior para a mantissa e também intervalos maiores para o expoente.

No sistema de precisão dupla, são reservados 64 bits (8 bytes) para armazenar um número real, sendo que:

1 bit é reservado para o sinal do número (positivo ou negativo);
11 bits são reservados para o expoente da base, que é um número inteiro;
52 bits são reservados para a mantissa:

$\pm$	$e_1 e_2 \dots e_{11}$	$d_1 d_2 \dots d_{52}$

Analisando os bits reservados para o expoente, e seguindo a convenção descrita anteriormente para a representação deste expoente, teremos que no sistema de precisão dupla o expoente é representado por $s = 1023 + e$, uma vez que a sequência de 10 bits iguais a 1 corresponde a $2^{10} - 1 = 1023$. O menor expoente no sistema de precisão dupla é representado pela sequência de bits: 00000000001 e, portanto, $e = 1 - 1023 = -1022$.

E, o maior expoente representado neste sistema será dado pela sequência de bits, 11111111110, que corresponde ao número $2^{11} - 2 = 2046$. Daí, teremos $2046 = 1023 + e \Rightarrow e = 2046 - 1023 = 1023$.

O menor número positivo será: $1.00000...0 * 2^{-1022} \approx 2.3332 * 10^{-302}$ e o maior número positivo será: $1.1111.....1 * 2^{1023} \approx (2 - 2^{-52}) * 2^{1023} \approx 1.7977 * 10^{308}$.

Operações em Aritmética de Ponto Flutuante

Para simplificar o entendimento sobre a forma como são realizadas as operações em ponto flutuante, vamos supor um sistema de aritmética em ponto flutuante que trabalha com 4 dígitos na mantissa, na base 10, expoente e no intervalo $e \in [-5 \;\; 5]$.

Considerando que $x = 0.9370 * 10^4$ e $y = 0.1272 * 10^2$ estão representados exatamente vamos realizar a adição de x com y e representar o valor resultante neste sistema.

A adição em ponto flutuante requer que as parcelas tenham o mesmo expoente para que as mantissas possam ser adicionadas. Para isto, a mantissa do número de menor expoente deve ser deslocada para a direita. Este deslocamento deve ser de um número de casas decimais iguais à diferença entre os dois expoentes.

Considerando os números acima, temos que alinhar a mantissa de y de modo a escrevê–lo com expoente igual a 4, desta forma, $y = 0.1272 * 10^2 = 0.001272 * 10^4$. Observe que neste caso, a mantissa de y não está normalizada (primeiro dígito é nulo) e o número de digítos na mantissa é maior que 4. Por esta razão, temporariamente, y será armazenado num acumulador de precisão dupla, pois assim, garante–se uma precisão maior no resultado. Teremos então:

$$z = x + y = 0.93700000 * 10^4 + 0.00127200 * 10^4 = 0.93827200 * 10^4.$$

Este é o resultado exato desta operação. Dado que no sistema adotado o número de dígitos na mantissa é igual a 4, o resultado z deverá ser armazenado neste sistema. Se for usado o arredondamento, teremos, $z = 0.9383 * 10^4$ e se for usado o truncamento, teremos, $z = 0.9382 * 10^4$.

Consideremos agora a operação $x + y$, com $x = 0.127 * 10^3$ e $y = 0.97 * 10^{-1}$. A mantissa de y deve ser deslocada 4 casas para a direita:

$$z = x + y = 0.12700000 * 10^3 + 0.00009700 * 10^3 = 0.12709700 * 10^3.$$

Armazenando z no sistema da máquina (4 dígitos), teremos: $z = 0.1271 * 10^3$ no arredondamento. No truncamento o resultado armazenado será $z = 0.1270 * 10^3 = x$ e, neste caso, podemos constatar que a parcela y não contribuiu para o resultado final, isto é, y atuou como elemento neutro nesta adição.

A propriedade do elemento neutro na adição de números reais: (*existe e é único o número real* $\bar{e}$ *tal que* $a + \bar{e} = 0$, $\forall a \in \mathbb{R}$), não se verifica na adição em ponto flutuante, pois cada número representado exatamente no sistema admite um conjunto de elementos neutros.

Considerando novamente, $x = 0.9370 * 10^4$ e $y = 0.1272 * 10^2$ vamos realizar a multiplicação $x*y$ e armazenar o resultado neste sistema . Na operação de multiplicação, realizamos o produto das mantissas e o expoente final da base é obtido, adicionando os expoentes de cada parcela. Então, teremos $z = (0.9370 * 0.1272) * 10^6 = 0.1191864 * 10^6$. E, finalmente, $z = 0.1192 * 10^6$ se for efetuado o arrendodamento e $z = 0.1191 * 10^6$ se for efetuado o truncamento.

Observamos que a propriedade associativa não se verifica em aritmética de ponto flutuante. Para exemplificar, consideremos o cálculo de $w = (a * b)/c$ onde $a = 0.1000 * 10^3$, $b = 0.3500 * 10^4$ e $c = 0.7000 * 10^1$.

Calculando w através da sequência de operações: $t = (a * b)$ e $w = t * (1/c)$ teremos: $a * b = (0.1000 * 0.3500) * 10^3 * 10^4 = 0.03500 * 10^7 = 0.3500 * 10^6$. Porém, o expoente da base é 6, que ultrapassa o limite para o maior expoente, que é 5. Portanto, esta sequência de operações conduz à ocorrência de *overflow*.

No entanto, se efetuarmos primeiramente uma divisão, por exemplo, $r = b/c$, teremos:

$$r = b/c = (0.3500/0.7000) * 10^4 * 10^{-1} = 0.5000 * 10^3$$

e, em seguida a multiplicação:

$$a * r = (0.1000 * 0.5000) * 10^3 * 10^3 = 0.0500 * 10^6 = 0.5000 * 10^5.$$

Neste caso, o *overflow* é evitado.

Referências Bibliográficas

[1] M. L. Overton, Numerical Computing with IEEE Floating Point Arithmetic, SIAM, Philadelphia, 2001.

[2] IEEE standard for binary floating–point arthmetic: ANSI/IEEE std 754–1985, 1985. Reprinted in *SIGPLAN Notices*, 22, pp. 9–25, 1987. `http://grouper.ieee.org.groups/754/`