Curso de Cálculo Numérico

Professor Raymundo de Oliveira

Capítulo 2 - Representação binária de números inteiros e reais

2.1 Representação de um número na base dois

Escrever um número inteiro em binário, isto é, na base dois, não apresenta problema. Cada posição digital representará uma potência de dois, da mesma forma que nos números decimais, cada posição representa uma potência de dez. Assim, 23.457 significa:

$2x10^4 + 3x10^3 + 4x10^2 + 5x10^1 + 7x10^0$.

Na base dois, a base usada nos computadores binários, o número 110101 representa:

$1x2^5 + 1x2^4 + 1x2^2 + 1x2^0 = (53)_{decimal}$

Os números com parte fracionária, da mesma forma, podem ser representados, usando-se potências negativas de dez, na base dez e de dois, na base dois.

Assim, 456,78 significa: $4x10^2 + 5x10^1 + 6x10^0 + 7x10^{-1} + 8x10^{-2}$.

O número binário 101,101 significa, na base dois:

$1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} = 5,625$

Sabe-se que, na base dez, para se multiplicar um número pela base, isto é, por dez, basta deslocar a vírgula uma casa para a direita.

O mesmo ocorre com qualquer base, em particular com a base dois. Para mutiplicar um número por dois, basta deslocar a vírgula uma casa para a direita.

7 = 111 , 14 = 1110 , 28 = 11100 , 3,5 = 11,1

Mostra-se que:

0,8 = 0,1100110011001100...

0,4 = 0,01100110011001100...

1,6 = 1,1001100110011...

1,2 = 1,001100110011...

2.2 Conversão Decimal >> Binário

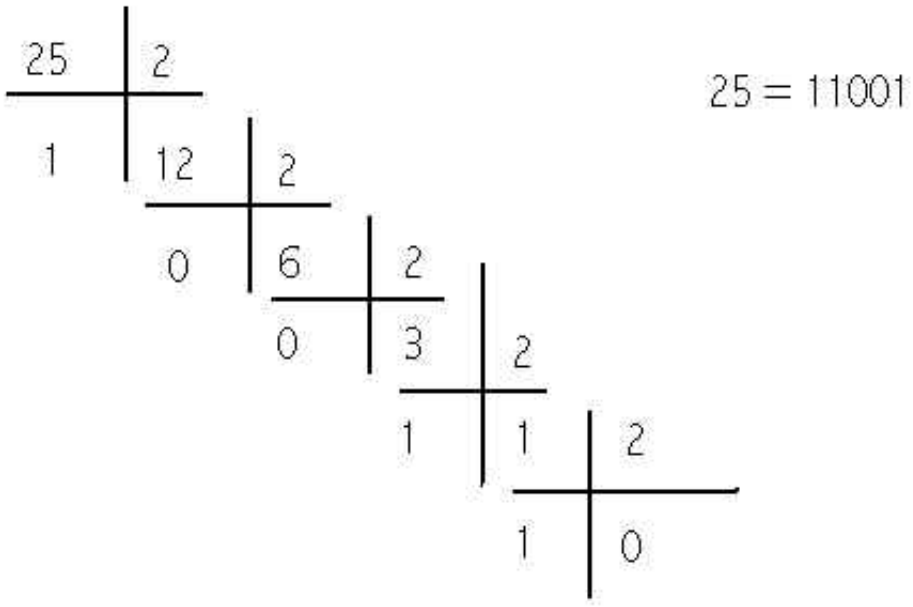
Números Inteiros

A conversão do número inteiro, de decimal para binário, será feita da direita para a esquerda, isto é, determina-se primeiro o algarismos das unidades (o que vai ser multiplicado por 2^0) , em seguida o segundo algarismo da direita (o que vai ser multiplicado por 2^1) etc...

A questão chave, por incrível que pareça, é observar se o número é par ou ímpar. Em binário, o número par termina em 0 e o ímpar em 1. Assim determina-se o algarismo da direita, pela simples divisão do número por dois; se o resto for 0 (número par) o algarismo da direita é 0; se o resto for 1 (número ímpar) o algarismo da direita é 1.

Por outro lado, é bom lembrar que, na base dez, ao se dividir um número por dez, basta levar a vírgula para a esquerda. Na base dois, ao se dividir um número por dois, basta levar a vírgula para a esquerda. Assim, para se determinar o segundo algarismo, do número em binário, basta lembrar que ele é a parte inteira do número original dividido por dois, abandonado o resto.

Vamos converter 25 de decimal para binário.



Parte Fracionária do Número

A conversão da parte fracionária do número será feita, algarismo a algarismo, da esquerda para a direita, baseada no fato de que se o número é maior ou igual a 0,5 , em binário aparece 0,1, isto é, o correspondente a 0,5 decimal.

Assim, 0,6 será 0,1_ _ ..., ao passo que 0,4 será 0,0_ _ ...

Tendo isso como base, basta multiplicar o número por dois e verificar se o resultado é maior ou igual a 1. Se for, coloca-se 1 na correspondente casa fracionária, se 0 coloca-se 0 na posição. Em qualquer dos dois casos, o processo continua, lembrando-se, ao se multiplicar o número por dois, a

vírgula move-se para a direita e, a partir desse ponto, estamos representando, na casa à direita, a parte decimal do número multiplicado por dois.

Vamos ao exemplo, representando, em binário, o número 0,625.

0,625 x 2 = 1,25 , logo a primeira casa fracionária é 1.

Resta representar o 0,25 que restou ao se retirar o 1 já representado.

0,25 x 2 = 0,5 , logo a segunda casa é 0.

Falta representar o 0,5 .

0,5 x 2 = 1 , logo a terceira casa é 1.

$0{,}625_{10} = 0{,}101_2$

Quando o número tiver parte inteira e parte fracionária, podemos calcular, cada uma, separadamente.

Tentando representar 0,8, verifica-se que é uma dízima.

0,8 = 0,110011001100....

Da mesma forma, vê-se que 5,8 = 101,11001100... , também uma dízima.

11,6 = 1011,10011001100... o que era óbvio, bastaria deslocar a vírgula uma casa para a direita, pois 11,6 = 2 x 5,8 .

2.3 Ponto fixo e ponto flutuante

Em todos esses exemplos, a posição da vírgula está fixa, separando a casa das unidades da primeira casa fracionária.

Entretanto, pode-se variar a posição da vírgula, corrigindo-se o valor com a potência da base, seja dez ou dois, dependendo do sistema que se use.

Façamos, mais uma vez, analogia com o sistema decimal. 45,31 corresponde a $4x10^1 + 5x10^0 + 3x10^{-1} + 1x10^{-2}$. Esse mesmo número poderia ser escrita como sendo 4,531 x 10^1 ou $0{,}4531x10^2$ ou $453{,}1x10^{-1}$ etc…Chama-se a isso ponto flutuante (floating point), pois no lugar de se deixar sempre a posição da vírgula entre a casa das unidades e a primeira casa decimal, flutua-se a posição da vírgula e corrige-se com a potência de dez. Forma normalizada é a que tem um único dígito, diferente de zero, antes da vírgula; no caso seria: 4,531 x 10^1.

Com a base dois pode-se fazer exatamente a mesma coisa, escrevendo-se o mesmo número 110101 como sendo $110{,}101x2^3$ ou $1{,}10101x2^5$ ou $0{,}0110101x2^7$ ou outras formas. Claro que esses expoentes também deverão ser escritos na base dois, onde $(3)_{10} = (11)_2$ e $(7)=(111)_2$, e assim por diante, ficando: 110,101 x $(10)^{11}$ ou $1{,}10101x(10)^{101}$ ou $0{,}0110101x(10)^{111}$.

2.4 Forma normalizada

Como se vê, há diferentes maneiras de escrever o mesmo número. Como já afirmamos, chama-se forma normalizada aquela que apresenta um único dígito, diferente de zero, antes da vírgula.

$110101 = 1{,}10101 \times 2^5$ ou, escrevendo-se o próprio 5 também na base dois, $1{,}10101 \times 2^{101}$. A base 2 está sendo mantido na forma decimal, 2 , e não na binária 10 , porque ela não precisará ser representada, por ser implícita.

Chama-se mantissa ao número 1,10101 e expoente ao número 101, deste exemplo.

Seguem-se outros exemplos:

1110,01 ou $1{,}11001 \times 2^3$ ou $1{,}11001 \times 2^{11}$, que corresponde a 14,25, em decimal.

0,0011 ou $1{,}1 \times 2^{-3}$ ou $1{,}1 \times 2^{-11}$, que corresponde a 0,1875 em decimal.

0,1001 ou $1{,}001 \times 2^{-1}$, que corresponde a 0,5625 em decimal.

Pode-se observar que, para se representar um número real, é necessário armazenar a mantissa e o expoente, sendo dispensável representar o "1,", por estar sempre presente, sendo, também, desnecessário, armazenar o dois, base do sistema.

Para se definir a maneira como o computador armazenará o número real em ponto flutuante, é preciso definir o número de bits que ele usará para representar a mantissa e o número de bits para o expoente.

Suponha-se que um determinado computador reserve 1 byte, isto é, 8 bits, para representar os números reais. Admita-se que usa o primeiro bit para sinal do número, três bits seguintes para o expoente e os últimos quatro bits para o restante da mantissa.

bit 0	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7

O bit 0 indica o sinal do número: 0 positivo, 1 negativo.

Os bits 1, 2 e 3 constituem o expoente e precisam representar tanto expoentes positivos quanto expoentes negativos. Com esses três bits, há 8 possibilidades: 000, 001, 010, 011, 100, 101, 110, 111, que representariam números de 0 até 7. Isso não serviria pois precisamos de expoentes negativos, além dos positivos.

Deram-se aos expoentes 000 e 111 significados especiais, a serem tratados daqui a pouco.

Admita-se que o número 3, isto é, 011 represente o expoente zero. Os

anteriores representarão os expoentes negativos e os posteriores os expoentes positivos. Dessa maneira, o número que representa o expoente será o número em binário menos três, conforme tabela a seguir.

Bits	001	010	011	100	101	110
valor	1	2	3	4	5	6
expoente	-2	-1	0	1	2	3

É importante observar que num número diferente de zero, normalizado na base 2, a mantissa sempre começará por 1. Assim sendo, não há necessidade de se representar o (1,) pois isso ficaria implícito, bastando representar os dígitos que aparecem depois da vírgula. Sendo m o número de bits representados da mantissa, o número representado terá, sempre, m+1 dígitos.

Observem-se os exemplos a seguir:

3,5 … 11,1 … 1,11 x 2^1 … 1,1100 x 2^1 … 01001100

2,75 … 10,11 … 1,0110 x 2^1 … 01000110

7,5 … 111,1 … 1,1110 x 2^2 … 1,1110 x 2^{10} … 01011110

-7,25 … -111,01 … -1,1101 x 2^2 … -1,1101 x 2^{10} … 11011101

-0,375… -0, 011 … -1,1000 x 2^{-2} … -1,1000 x 2^{-10} … 10011000

Como se pode ver, o maior número positivo que pode nele ser representado é: 01101111 , que corresponde a 1,1111 x 2^3 , isto é: 1111,1, ou 15,5.

O menor número positivo seria: 00010000 , que corresponde a 1,0000 x 2^{-2} , isto é: 0,25.

Aqui há uma observação a ser feita. Lembremo-nos que os expoentes 000 e 111, não foram considerados, até agora; eles teriam tratamento especial.

Todos os números estão na forma normalizada, isto é: (± 1,_ _ _ …. x 2^{exp}).

Usa-se o expoente 000 , quando se quer indicar que o número não está normalizado, ficando o 000 como o menor expoente, isto é, -2 , no nosso exemplo.

A mantissa passa a ser 0, _ _ _ …. Onde, depois da vírgula, estão os dígitos da mantissa, só que, agora, sem a precedência do (1,) , como era antes, e sim (0,). Neste caso, não teremos, mais, m+1 dígitos significativos, como tínhamos quando os números eram normalizados.

Exemplo:

00001111 …0,1111 x 2^{-2} … 0,001111 … 0,25 – 2^{-6}… portanto menor que 0,25

00001110 …0,1110 x 2^{-2} … 0,001110 … 0,25 – 2^{-5} … portanto menor que o anterior

00000100 …0,0100 x 2^{-2} … 0,0001 … 0,0625

e assim por diante.

O menor número positivo é portanto: 00000001 … 0,0001 x 2^{-2} … 0,000001 …2^{-6} … 0,015625 .

O número 00000000 representa + 0 e o número 10000000 representa - 0, devendo ambos serem reconhecidos como iguais nas comparações.

O expoente 111 é reservado para representar mais infinito, com zero na frente (01110000) e menos infinito com 1 na frente (11110000), em ambos os casos a mantissa sendo 0000.

O mesmo expoente 111 é ainda utilizado para caracterizar indeterminação, 11111000.

As demais combinações com o expoente 111 não são válidas, sendo consideradas (not a number).

Nesse exemplo que estamos explorando, o número real é representado em 8 bits, sendo 1 para o sinal, 3 para o expoente e 4 para a mantissa, não sendo representado o 1, que antecede os 4 bits.

E quando a mantissa não cabe nos 4 bits ? Somos obrigados a arredondar a mantissa para que ela caiba nos 4 bits. Vamos, assim, perder precisão no número e ele não mais representará, exatamente, o número desejado.

Vamos tentar representar, nesse nosso computador hipotético, o número 9,375.

9,375 = 1001,011 = 1,001011 . 2^3 .

Mas só existe bit para 4 dígitos, sobrando 0,000011 . 2^3 . É feito o arredondamento. Assim, se a parte abandonada é inferior à metade da última unidade representada, ela é abandonada, se for superior à metade, soma-se 1 à última unidade. Nesse caso, só havendo 4 bits, armazenam-se os 4 primeiros bits após o 1, , isto é: 0010. Seria abandonado 11 após o último zero. Isso representa 0,11 da última unidade, que é maior que a metade da última unidade. Logo, soma-se 1 à última unidade, que fica sendo 1 e não zero.

1,001011 . 2^3 = 1,0011 . 2^3 . Este número não é mais 9,375 e sim 9,5. Perdemos precisão. Na verdade, esse computador não sabe representar 9,375 e, ao tentar fazê-lo, representa 9,5. Ele não tem bits suficientes para fazer uma

representação melhor.

Se quiséssemos representar 9,125 = 1001,001 = 1,001001 . 2^3 , a quantidade a ser abandonada, após os 4 bits, seria 01, o que representa 0,01 da última unidade representada. Isso é menor que metade dessa unidade e, portanto, será simplesmente abandonada, sem alterar o último bit representado.

9,125 = 1001,001 = 1,001001 . 2^3 = 1,0010 . 2^3 = 1001. Esse número vale 9 e não 9,125.

Resumindo, é feito o arredondamento na última casa (bit) representado. Quando o que restar for maior que metade da última unidade, soma-se 1 a essa unidade; quando for menor que metade da última unidade, essa parte restante é abandonada.

E quando for exatamente igual à metade ?

Por exemplo, vamos tentar representar 9,25, nesse computador hipotético.

9,25 = 1001,01 = 1,00101 . 2^3 .

Sobra 1 após os quatro bits, isto é, sobra exatamente a metade da última unidade.

Neste caso, observa-se o bit anterior. Se for zero, mantem-se o zero, se for 1, soma-se 1 a esse bit.

Neste caso o último bit é zero, logo assim permanece.

9,25 = 1001,01 = 1,00101 . 2^3 = 1,0010 . 2^3 = 1001 = 9.

Se fôssemos representar 10,75 = 1010,11 = 1,01011 . 2^3 .

Sobra 1 após os quatro bits, isto é, sobra exatamente a metade da última unidade.

O bit que resta segue um bit 1. Neste caso, soma-se 1 a este bit, ficando com:

1,0110 . 2^3 = 1011 = 11 (onze).

Como se vê, ora se aumenta o número, ora se diminui. Isso evita que, no caso de grande número de aproximações, haja tendência num certo sentido, buscando-se equilíbrar os aumentos com as diminuições.

Usamos, até agora, a hipótese de representar o número real em 8 bits. Na realidade, usam-se mais bits, nessa representação.

A norma IEEE 754, publicada em 1985, procurou uniformizar a maneira como as diferentes máquinas representam os números em ponto flutuante, bem como devem operá-los.

Essa norma define dois formatos básicos para os números em ponto flutuante: o formato simples, com 32 bits e o duplo com 64 bits. O primeiro bit é para o sinal: 0 representa número positivo e 1 representa número negativo. No formato simples o expoente tem 8 bits e a mantissa tem 23 bits; no formato duplo, o expoente tem 11 bits e a mantissa 52 bits.

No formato simples, o menor expoente é representado por 00000001, valendo -126, e o maior expoente é representado por 11111110, valendo +127. Em ambos os casos, o expoente vale o número representado em binário menos 127.

No formato duplo, o menor expoente é representado por 00000000001, valendo -1022, e o maior expoente é representado por 11111111110, valendo +1023. Em ambos os casos, o expoente vale o número representado em binário menos 1023.

Em ambos os formatos, a norma IEEE 754 prevê o chamado underflow gradual, permitindo obter números bem mais próximos de zero. Para isso, como mostrado no exemplo hipotético, o expoente representado por 000…000 representa o menor expoente, "-126" no formato simples e "–1022" no formato duplo, e a mantissa deixa de ser normalizada. Dessa maneira podemos representar, como menor número positivo:

no formato simples: 0 00000000 00….01 isto é: $2^{-126} \times 2^{-23} = 2^{-149}$

no formato duplo: 0 00000000000 00….01 isto é: $2^{-1022} \times 2^{-52} = 2^{-1074}$

No formato simples, o zero possui, ainda, duas representações 0 00000000 000...00 , correspondendo a mais zero e 1 00000000 000...00 , correspondendo a menos zero, ambas iguais em qualquer operação de comparação.

Mais infinito é representado por 0 11111111 000...00 e menos infinito por 1 11111111 000...00 .

Indeterminado é representado por 1 11111111 100...00 .

As demais combinações não são válidas, sendo consideradas "not a number".

Dessa maneira, o intervalo de números representáveis será:

no formato simples:

de $-(2-2^{-23}) \times 2^{127}$ a -2^{-149}

zero

de 2^{-149} a $(2-2^{-23}) \times 2^{127}$.

no formato duplo:

De $-(2-2^{-52}) \times 2^{1023}$ a -2^{-1074}

zero

de 2^{-1074} a $(2-2^{-52}) \times 2^{1023}$.

2.5 Erro Relativo máximo de um número em ponto flutuante

Ao se tentar representar um número real em ponto flutuante, normalmente, ele é arredondado, de modo a poder ser escrito no número finito de bits onde precisa ser armazenado. Dessa forma, é, automaticamente, introduzido um erro e ele não mais representa o número desejado. Ainda assim, é possível determinar o maior erro relativo possível dessa representação.

Vamos admitir que, no nosso sistema, a mantissa seja representada em m bits.

Seja o número **a** , que representado na forma normalizada seria: 1,**b** x $2^{\mathbf{c}}$. Vamos admitir que **c** esteja dentro dos limites dessa representação, caso contrário teríamos overflow.

Entretanto, a mantissa **b** precisa caber no número previsto de bits, reservados para a mantissa. Isso pode levar à necessidade de arredondar a mantissa **b**, introduzindo erro.

Vamos admitir, como exemplo, que o número **a** seja :

a = 1,101010110100101… x $2^{\mathbf{c}}$ e que m, número de bits da mantissa, seja 5.

Teríamos que escrever **a** como sendo: 1,10101 x $2^{\mathbf{c}}$, abandonando 0,000000110100101.. x $2^{\mathbf{c}}$, que seria o erro introduzido. O Erro de **a** será, portanto: $Erro_{\mathbf{a}}$ = 0,0110100101… x $2^{\mathbf{c}-5}$, onde, insisto, 5 é o valor de m, número de bits da mantissa.

O erro relativo será:

$E_{\mathbf{a}} = Erro_{\mathbf{a}} / \mathbf{a}$, isto é: 0,0110100101… x $2^{\mathbf{c}-5}$ / 1,101010110100101… x $2^{\mathbf{c}}$

$E_{\mathbf{a}}$ = 0,0110100101… x 2^{-5} / 1,101011110100101…

O erro relativo será máximo, quando tivermos o maior numerador e o menor denominador, isto é:

$E_{\mathbf{a}}$ (max) = 0,01111111….. x 2^{-5} / 1,0000000…

Lembrando que 0,011111…. < 0,5 , tem-se: $E_{\mathbf{a}}$ (max) < 0,5 x 2^{-5} , onde 5 está representando m, número de bits da mantissa.

Portanto, $E_{\mathbf{a}}$ (max) < $2^{-(m+1)}$.

Assim, o número de bits da mantissa condiciona a precisão com que se pode trabalhar.

A expressão acima, do erro máximo, só é válida na região normalizada, pois na região não normalizada, bem próxima a zero, pelo underflow gradual, fica reduzida a precisão dos números que passam a não ter mais m+1 dígitos significativos, preço necessário para se poder obter valores bem mais próximos do zero.

O número de bits do expoente indica a faixa (range) de variação dos números possíveis, indicando os limites, fora dos quais haverá overflow ou underflow.

2.6 Valor verdadeiro do número armazenado

Já vimos que ao se tentar armazenar o número 0,8 em ponto flutuante, o número será arredondado, tendo seu valor alterado.

Assim, sabendo-se que 0,8 = 0,1100110011001100.... = $1,100110011001100...2^{-1}$

e que, em **precisão simples**, terem os **23** bits após a vírgula, qual será o verdadeiro valor armazenado ?

Vamos calcular.

0,8 » $1,10011001100110011001100110011001100...\ 2^{-1}$

onde em vermelho estão os bits depois do vigésimo terceiro.

Como os bits que serão abandonados representam mais do que a metade da última unidade, soma-se 1 ao vigésimo terceiro bit, obtendo-se:

0,8 » $1,10011001100110011001101.2^{-1} > 0,8$

Será armazenado um número maior que 0,8, neste caso.

Chamando esse valor armazenado de A, tem-se:

A - 0,8 = $(1,0 - 0,110011001100...).2^{-1}.2^{-23} = (1,0 - 0,8).2^{-24} = 0,2.2^{-24} = 1,19209..\ 10^{-8}$

A = 0,8000000119209... Tem-se um número ligeiramente maior que 0,8, neste caso.

E se quiséssemos armazenar 1,8, o que seria, de fato, armazenado?

1,8 = 1,1100110011001100... » $1,11001100110011001100110.2^{0} = B$

B será ligeiramente menor que 1,8, pois foi abandonado 0,011001100... $2^{-23} = 0,4.2^{-23}$

1,8 - B = $0,4.2^{-23} = 4,76837..10^{-8}$

B = 1,79999995232... Tem-se um número ligeiramente menor que 1,8, neste caso.

O importante disso tudo é ver que, em binário, o número é formado pela soma de potências de dois; potências positiva e negativas. Quando um número não pode ser formado por um número finito dessas potências, como é o caso de 0,8, ele se torna no que em decimal se chama dízima, necessitando de infinitos bits para representá-lo. Como, em ponto flutuante, temos um número finito de bits, o número será arredondado, para caber nesse número de bits. Daí o erro em sua representação.

O arredondamento pode ser para mais ou para menos, como acabamos de ver. Tudo depende do valor a ser abandonado. Se esse valor for maior que metade da unidade representada pelo último bit, arredonda-se para cima, somando-se 1 ao último bit representado. Se esse valor for menor que metade da unidade representada pelo último bit, arredonda-se para baixo, eliminando-se os bits a partir do vigésimo quarto.

No caso de precisão dupla, onde se tem 52 bits após a vírgula, o erro no arredondamento será menor.

$0,8 = 1,10011001100110011001100110011001100110011001100110011001100110011 .. 2^{-1}$

A parte a ser abandonada é maior que metade da unidade, pois é 0,1001100...

Logo, arredonda-se para cima, somando-se 1 ao última bit.

$0,8 \approx 1,10011001100110011001100110011001100110011001100110011010 .2^{-1} = A$

$A - 0,8 = (10 - 1,100110011...). 2^{-1}2^{-52}$

$A - 0,8 = (2 - 1,6).2^{-53} = 4,44089...10^{-17}$

A = 0,8000000000000000444089...

RESUMO

Representação de números reais em um computador

1. há um número finito de números reais representáveis num dado computador, função da maneira como ele armazena o número em Ponto flutuante.
2. é possível determinar o maior e o menor número real, positivo ou negativo, representável nesse computador.
3. dado um número real, é possível determinar os número reais vizinhos dele, imediatamente acima e imediatamente abaixo, representáveis num dado computador, função da maneira como esse computador representa os números reais em ponto flutuante.
4. ao se tentar representar um número real em ponto flutuante, pode-se

determinar o maior erro relativo possível dessa representação.

5. conhecidos os erros relativos de dois números, pode-se determinar o maior erro relativo possível de sua soma, subtração, multiplicação e divisão.
6. a norma IEEE 754, publicada em 1985, procurou uniformizar a maneira como as diferentes máquinas representam os números em ponto flutuante, bem como devem operá-los.
7. essa norma define dois formatos básicos para os números em ponto flutuante: o formato simples, com 32 bits e o duplo com 64 bits. O primeiro bit é para o sinal: 0 representa número positivo e 1 representa número negativo. No formato simples o expoente tem 8 bits e a mantissa tem 23 bits; no formato duplo, o expoente tem 11 bits e a mantissa 52 bits.
8. no formato simples, o menor expoente é representado por 00000001, valendo -126, e o maior expoente é representado por 11111110, valendo +127. Em ambos os casos, o expoente vale o número representado em binário menos 127.
9. no formato duplo, o menor expoente é representado por 00000000001, valendo -1022, e o maior expoente é representado por 11111111110, valendo +1023. Em ambos os casos, o expoente vale o número representado em binário menos 1023.
10. em ambos os formatos, a norma IEEE 754 prevê o chamado underflow gradual, sendo o expoente representado por 000…000, e a mantissa deixa de ser normalizada, permitindo obter números bem mais próximos de zero.
11. O menor número positivo possível será, portanto:
    0 00000000 00000000000000000000001 , que corresponde a:
    $0{,}00000000000000000000001 \times 2^{-126}$, isto é: $2^{-126-23} = 2^{-149}$, para precisão simples e $2^{-1022-52} = 2^{-1074}$, para precisão dupla.
12. ao se tentar representar um número real em ponto flutuante, é possível determinar o maior erro relativo possível dessa representação.
13. conhecidos os erros relativos de dois números, é possível determinar o maior erro relativo possível de uma operação entre eles, como por exemplo: sua soma, subtração, multiplicação e divisão.

Se você tiver dúvidas sobre a matéria, meu e-mail é:

raymundo.oliveira@terra.com.br

Home | Programa | Exercícios | Provas | Professor | Links |