Capítulo 1

Objetivos Específicos de Aprendizagem

Ao finalizar este Capítulo, você será capaz de:

- Compreender os fundamentos dos sistemas de numeração utilizados atualmente;
- Efetuar conversões entre diferentes sistemas de numeração;
- Realizar representação digital de quantidades reais e inteiras; e
- Identificar os tipos e as consequências de erros existentes em representações digitais.

1 Sistemas de Numeração e Erros Numéricos

Neste Capítulo, vamos apresentar alguns tópicos que fundamentam o cálculo numérico computacional, como as representações de quantidades em computadores, suas respectivas propriedades e os possíveis erros gerados.

Você já observou como um árbitro de futebol indica os tempos de acréscimo em uma partida? Ele normalmente levanta a mão com três dedos abertos associando esse gesto à indicação de três minutos de acréscimo.

1.1 Representação de Quantidades

As formas de representação de quantidades ou números, chamadas de Sistemas de Numeração, evoluíram e continuam evoluindo no curso da história humana, passando dos sistemas meramente "**associativos**" para os sistemas "simbólicos", em uso há milênios e nos quais determinado símbolo pode representar vários valores, e, finalmente, para os sistemas "lógicos magnéticos", atualmente em uso nos computadores.

Podemos agrupar os sistemas de numeração simbólicos em duas grandes categorias. Na primeira, denominada "aditiva", cada símbolo tem um valor fixo e o valor total é representado pela junção dos valores específicos de cada símbolo do número, por exemplo, os antigos sistemas romano, grego, egípcio, hebraico. A segunda categoria, que praticamente prevaleceu à primeira e foi desenvolvida pelos astrônomos hindus por volta do século VI, é denominada "posicional". Nesta categoria, o valor de cada símbolo é relativo e depende da sua posição no número.

A representação simbólica de um número depende da quantidade de símbolos utilizados. Essa quantidade é denominada de **base**, que denotaremos por um número natural β.

A seguir, vamos apresentar exemplos de um sistema aditivo e de sistemas posicionais com suas respectivas bases.

Exemplo de sistema aditivo romano:

a) **Sistema Romano**: base $\beta = 7$ e símbolos $\{I, V, X, L, C, D, M\}$. Note a inexistência do "zero" e a limitação dos valores apenas à ordem dos milhares.

Exemplos de três sistemas posicionais:

a) **Sistema Decimal**: base $\beta = 10$ e símbolos $\{0,1,2,3,4,5,6,7,8,9\}$, denominados de **dígitos**, ou **algarismos**. Note a presença do "zero".

b) **Sistema Binário**: base $\beta = 2$ e símbolos $\{0,1\}$, denominados de *bits*, ou *binary digits*. A aritmética binária foi desenvolvida no início do século XVII pelo matemático Leibnitz; em 1814, Boole criou a álgebra booleana; e, em 1935, Shannon a implementou em circuitos.

c) **Sistema Ternário-Quântico**: base $\beta = 3$ e símbolos $\{0,1,0\hat{}1\}$, em que $0\hat{}1$ é uma superposição dos dois *bits*, denominados de *qubits*, ou *bit* quântico. Este sistema é usado na moderna computação quântica, que faz uso direto de propriedades da mecânica quântica, ampliando a representação e o processamento.

Dígito – do latim *digitus*, mesmo que dedo; sinal convencional que representa graficamente os números; e cada um dos símbolos que representam os números inteiros positivos menores que a base de um sistema numérico. Fonte: Adaptado de Instituto Antônio Houaiss (2009).

Algarismo – derivação da palavra *algorismi*, versão latina do nome al-Khwarizmi, matemático árabe que escreveu tratados sobre aritmética e álgebra, entre outros. Fonte: Só Matemática (1998–2016).

Denominamos "**notação**" a forma ou o tipo de palavra com que representamos um número X na base, β, X_β, usando um sistema posicional. Hoje, usamos quatro tipos de notação:

Neste livro, utilizaremos formato de notação científica com "ponto" em vez de "vírgula", para facilitar o processo de copiar, colar e testar exemplos e algoritmos, uma vez que as linguagens de programação usam notação com o ponto "." para separar a parte inteira da parte fracionária dos números, conforme padrão em países de cultura inglesa.

a) **Notação convencional,** ou de **ponto fixo**: nesse caso, representamos um X_β pela "palavra": $X_\beta = (a_1 a_2 \ldots a_k \,.\, a_{k+1} \ldots a_{k+n})_\beta$, em que β é a base, $a_i \in \{0,1,2,\ldots,\beta-1\}$, k é o comprimento da parte inteira, e n da parte fracionária do número, com $k,\ n \in \mathbb{N}$.

b) **Notação fracionária**: nesse caso, representamos tipicamente um X_β por $X_\beta = (a_1 / a_2)_\beta$, em que $a_2 \neq 0$ e $a_i \in \mathbb{Z}$. Essa representação é limitada a números racionais.

c) **Notação fatorada**: nesse caso, representamos um X_β por $X_\beta = \sum_{i=1}^{k} a_i \beta^{k-i} + \sum_{i=k+1}^{n+k} a_i \beta^{k-i}$, em que $a_i \in \{0,1,2,\ldots,\beta-1\}$, k é o comprimento da parte inteira, e n da parte fracionária do número, com $k,\ n \in \mathbb{N}$, quando representada na notação convencional.

d) **Notação em ponto flutuante**: nesse caso, representamos um X_β por $X_\beta = (0 \,.\, a_1 a_2 \ldots a_t)_\beta \ \beta^E = (\frac{a_1}{\beta^1} + \frac{a_2}{\beta^2} + \ldots + \frac{a_t}{\beta^t})_\beta \ \beta^E$, em que $a_i \in \{0,1,2,\ldots,\beta-1\}$. O grupo de símbolos $(a_1 a_2 \ldots a_t)_\beta$ é chamado de mantissa, contém os t **símbolos significativos** da representação e tem comprimento t fixo. Os expoentes E da base β estão situados em um intervalo de números inteiros, $I \leq E \leq S$, e dependem da forma de normalização da mantissa, ou seja, dependem da posição padronizada do ponto (ou vírgula), se antes ou depois do primeiro dígito significativo não nulo. Dessa forma, um sistema de ponto flutuante qualquer pode ser definido pela quádrupla $F(\beta, t, I, S)$.

Símbolos significativos

Símbolos significativos são os não nulos, contados a partir da esquerda até o último não nulo à direita, ou até o último (nulo ou não) indicado (zeros à direita são significativos e zeros à esquerda não são).

Atualmente, para fins de eficiência computacional, representamos um número na notação em ponto flutuante por: $X_\beta = (a_1 \, . \, a_2 \ldots a_t)_\beta \; \beta^E$, tanto em calculadoras científicas quanto em computadores.

O sistema de representação de quantidades que mais utilizamos é o sistema de numeração decimal. Entretanto, para facilitar a representação física dos números e otimizar as operações aritméticas em computadores, fazemos uso de outros sistemas de representação, como o sistema de numeração binário.

Ao longo do texto, apresentaremos exemplos com o objetivo de reforçar o conteúdo exposto.

Exemplo 1.1: represente o decimal $X = 309.57$ nas suas notações de ponto fixo, fatorada e ponto flutuante.

Solução:

Considerando a sua notação em ponto fixo:

$$X = (309.57)_{10}$$

Considerando a sua notação fatorada:

$$X = (309.57)_{10} = 3*10^2 + \; 0*10^1 + \; 9*10^0 + \; 5*10^{-1} + \; 7*10^{-2}$$

Considerando a sua notação em ponto flutuante:

$$X = (3.0957)_{10} * 10^{+2}$$

Observe que a construção das operações aritméticas deve considerar o valor relativo de cada símbolo numérico como apresentado na sua **notação fatorada**. Assim, quando fazemos qualquer operação aritmética, como a adição, operamos os termos envolvidos considerando

a posição do ponto (vírgula) como referência, operando unidade com unidade, dezena com dezena e assim por diante.

Na próxima seção, vamos detalhar os sistemas de numeração e as notações de números mais comuns em computadores.

1.2 Sistema de numeração Decimal ($\beta = 10$)

A aceitação do sistema de numeração decimal como padrão se deve a algumas de suas características, que detalharemos a seguir.

1.2.1 Utilizar Apenas Dez Símbolos

Tais símbolos são representados por 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, conhecidos também por algarismos arábicos ou hindu-arábicos, que são derivados da versão ainda hoje usada no mundo árabe (escrita da direita para a esquerda):

٩, ٨, ٧, ٦, ٥, ٤, ٣, ٢, ١,

Historicamente, a simbologia dos dígitos tem diversas hipóteses sobre sua origem e evolução no mundo ocidental, como:

a) número de ângulos existentes no desenho de cada algarismo;
b) número de traços contidos no desenho de cada algarismo;
c) número de pontos de cada algarismo;
d) número de diâmetros e arcos de circunferência contidos no desenho de cada algarismo; e
e) figuras desenhadas a partir dos traços de um quadrado e suas diagonais, conforme a Figura 1.1.

Figura 1.1 – Origem e evolução dos números hindu-arábicos

Fonte: Portal Escolar (2011)

1.2.2 Fazer Uso do Zero

O zero foi aceito com certa relutância na origem do sistema decimal, pois era difícil imaginar a quantificação e a representação do nada, do inexistente. Hoje é indispensável, pois é o indicador da ausência de certas potências da base.

Exemplo 1.2: no número decimal

$$(702.4)_{10} = 7*10^2 + 0*10^1 + 2*10^0 + 4*10^{-1}$$

a posição correspondente à dezena, ou à potencia 10^1, está ausente.

1.2.3 Adotar o Princípio da Posicionalidade

No sistema posicional, o valor de cada símbolo é relativo, isto é, depende da sua posição no número.

Exemplo 1.3: nos números decimais

$$(34\mathbf{8})_{10} = 3*10^2 + \mathbf{4}*10^1 + 8*10^0$$
$$(57\mathbf{4})_{10} = 5*10^2 + 7*10^1 + \mathbf{4}*10^0$$
$$(702.\mathbf{4})_{10} = 7*10^2 + 0*10^1 + 2*10^0 + \mathbf{4}*10^{-1}$$

o dígito, ou símbolo, ou algarismo **4** representa **4** dezenas, **4** unidades e **4** décimos, respectivamente. Note que nenhum dígito interfere na posição do outro, eles são inteiramente independentes entre si.

Agora, vamos utilizar as duas últimas características do sistema decimal para estabelecer outros sistemas de numeração.

1.3 Sistema Binário ($\beta = 2$)

O sistema binário, como vimos, utiliza apenas dois **símbolos**, 0 e 1, também chamados de *bits* , o **zero** e o princípio da **posicionalidade**. Nesse novo sistema de numeração, a correspondência com o decimal será:

Decimal	0	1	2	3	4	5	6	7	8	9	10	...	19	...
Binário	0	1	10	11	100	101	110	111	1000	1001	1010	...	10011	...

Exemplo 1.4: $(10011)_2 = (1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0)_{10} = (19)_{10}$

Observe no **Exemplo 1.4** que a forma fatorada do número binário já está representada na base decimal.

O sistema binário tem sido amplamente utilizado em computadores, por permitir a otimização de recursos físicos, e apresenta vantagens sobre o sistema decimal como:

a) **Simplicidade de representação física**: bastam dois estados físicos distintos para representar os dígitos da base:

   1 pode ser representado por $-$, on, ... e

   0 pode ser representado por $+$, off, ...

   Assim, ficou mais fácil construir uma máquina que reconheça dois estados físicos do que dez estados distintos.

b) **Facilidade na definição de operadores lógicos para operações aritméticas fundamentais**.

Acompanhe no Exemplo 1.5 a demonstração da vantagem do sistema binário sobre o decimal em relação à forma de implementação digital das operações aritméticas.

Exemplo 1.5: vamos calcular o número de combinações dos dígitos inteiros x e y necessário para obter a operação de adição. Em base decimal, temos 100 combinações dos possíveis 10 valores de x com 10 valores de y para definir a função adição entre dois dígitos ou algarismos. Podemos reduzir essas 100 combinações a 19 resultados diferentes, observe:

$$0+0=00$$
$$\vdots$$
$$0+9=09$$
$$1+0=01$$
$$\vdots$$
$$1+9=10$$
$$\vdots$$
$$\vdots$$

$$9+0=09$$
$$\vdots$$
$$9+9=18$$

Na base binária, necessitamos de apenas quatro **combinações de dígitos binários** x e y, com basicamente dois resultados diferentes:

$$0+0=0$$
$$0+1=1$$
$$1+0=1$$
$$1+1=10$$

Observe que, na última operação, temos um ***bit*** 1 extra transportado para o registro imediatamente à esquerda, com uma operação chamada popularmente de "vai-um" ("*carry out*").

Porém, uma **desvantagem** do sistema binário em relação ao sistema decimal é a necessidade de **registros longos** para armazenar números representando quantidades relativamente pequenas.

Exemplo 1.6: $(597)_{10}=(1001010101)_2$

Observe que, no **Exemplo 1.6**, foi necessário um registro com capacidade de armazenamento de 10 *bits* para representar a grandeza decimal $(597)_{10}$ de apenas 3 dígitos.

Combinações de dígitos binários

Podemos obter os resultados dessas adições binárias usando expressões booleanas entre x e y:

- *XOR* – também conhecida como **disjunção binária exclusiva**, devolve um *bit* 1 sempre que o número de operandos iguais a 1 é **ímpar**, consequentemente devolve um *bit* 0 sempre que o número de *bits* 1 for zero ou dois, e serve para definir o primeiro dígito (à direita); e
- *AND* – também conhecida como **conjunção binária**, devolve um *bit* 1 somente quando **ambos** os operandos sejam 1 e serve para definir o dígito extra, "vai-um" (à esquerda). Fonte: Patterson e Hennessy (1998).

1.4 Sistema Hexadecimal ($\beta = 16$)

No sistema hexadecimal, que também é sistema posicional, são símbolos representativos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, em que A, B, C, D, E, F representam as quantidades decimais 10, 11, 12, 13, 14, 15, respectivamente.

Exemplo 1.7: $(\text{FA9C})_{16} = 15*16^3 + 10*16^2 + 9*16^1 + 12*16^0 = (64156)_{10}$

$$(\text{FA9C})_{16} = (1111\ 1010\ 1001\ 1100)_2$$

O sistema hexadecimal utiliza um número reduzido de símbolos para representar grandes quantidades, por isso é um sistema de numeração interessante para compactação, visualização e armazenamento de dados, ou seja, usa **registros curtos**. Os registros binários internos de uma variável podem ser convertidos de forma direta para hexadecimal, como veremos na próxima seção.

Exemplo 1.8: $(11010110)_2 = (D6)_{16} = (214)_{10}$

8 *bits*	2 símbolos hexadecimais	3 símbolos decimais

1.5 Conversões entre Sistemas de Numeração

Conversões entre bases são necessárias para possibilitar a comunicação homem–computador e vice-versa, no caso dos sistemas decimal e binário, bem como para a compactação e descompactação das representações, nos casos das conversões binário para hexadecimal e vice-versa.

1.5.1 Conversão de Base β para Base 10

Obtemos diretamente as conversões de uma base β para base 10 escrevendo o número na sua forma fatorada, com fatores representados na base decimal. Essas conversões são utilizadas pelos

computadores para **apresentar** os **resultados** calculados, para a nossa visualização e o nosso uso.

Confira o desenvolvimento dos Exemplos 1.9 e 1.10 para entender as conversões entre binário e decimal utilizadas nas diversas formas de interação entre humano-computador, como os comandos de entrada e saída de dados utilizados nas linguagens de programação.

Exemplo 1.9: converta o binário $X = (101.1)_2$ para decimal:
Solução:

$$X = (101.1)_2 = 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} = (5.5)_{10}$$

1.5.2 Conversão de Base 10 para Base β

As conversões de base 10 para base $\beta = 2$ são utilizadas por computadores para **armazenar os dados**.

Exemplo 1.10: simule a entrada do número $X = (19.03125)_{10}$ em computador convertendo-o para binário.
Solução:
Como X não é um número inteiro, temos que converter separadamente as suas partes, inteira $(19)_{10}$ e fracionária $(0.03125)_{10}$:

a) Na parte inteira do número, efetuamos a sua divisão inteira pela base β, sucessivamente, e construímos o número convertido tomando o último quociente e os restos da última divisão até o da primeira.

$(19)_{10} = (\ ?\ )_2$

19	|2			
1	9	|2		
	1	4	|2	
		0	2	|2
			0	1

$(19)_{10} = (\underline{1}0011)_2$

Verificação: $(\underline{1}0011)_2 = \underline{1} * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = (19)_{10}$

b) Na parte fracionária do número: efetuamos a sua multiplicação inteira pela base β, sucessivamente; construímos o número convertido tomando os inteiros resultantes de cada produto; e multiplicamos novamente a parte fracionária do produto anterior até que o produto final seja um inteiro ou atinja a quantidade limite de símbolos na representação.

$(0.03125)_{10} = (?)_2$

0.03125	0.06250	0.12500	0.25000	0.50000
$*2$	$*2$	$*2$	$*2$	$*2$
0.06250	0.12500	0.25000	0.50000	1.00000
0	0	0	0	1

$(0.03125)_{10} = (0.00001)_2$

Verificação: $(0.00001)_2 = 0 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 0 * 2^{-4} + 1 * 2^{-5} = (0.03125)_{10}$

Logo, juntando as duas partes $(19.03125)_{10} = (10011.00001)_2$

A seguir, apresentaremos outro exemplo de conversão decimal binária, cujo resultado tem número de *bits* ilimitado.

Exemplo 1.11: converta $X = (0.1)_{10}$ para binário e armazene-o com 10 *bits* significativos.

Solução:

$$\begin{array}{cccccccccc} 0.1 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 \\ \hline *2 & *2 & *2 & *2 & *2 & *2 & *2 & *2 & *2 & *2 \\ \hline 0.2 & 0.4 & 0.8 & 1.6 & 1.2 & 0.4 & 0.8 & 1.6 & 1.2 & 0.4 \\ 0 & \underbrace{0 \quad 0 \quad 1 \quad 1} & & & & \underbrace{0 \quad 0 \quad 1 \quad 1} & & & & 0 \end{array}$$

Assim, $(0.1)_{10} = (0.0\underbrace{0011}\underbrace{0011}\underbrace{0011}\underbrace{0011}...)_2$ é exato na base binária somente na sua forma de dízima periódica com infinitos *bits*.

Um número com infinitos ***bits*** não pode ser armazenado, por isso precisamos desprezar parte dos seus ***bits*** **menos significativos** por um processo chamado **arredondamento**, veja o tópico Arredondamento Decimal na seção **Complementando...** ao final deste Capítulo.

Considerando que o registro binário disponível tenha apenas 10 *bits* para armazenar os dígitos significativos do **Exemplo 1.11**, faremos uma extensão direta do arredondamento decimal usual para o sistema binário da seguinte forma:

$$(0.1)_{10} = (0.000\underbrace{1100110011}_{t=10}\ \underbrace{0011..}_{\text{arredondamento}}\ )_2$$

Observe que a parcela a ser arredondada $(0.\underbrace{0011..}_{;})_2$ é menor do que $(0.5)_{10} = (0.1000...)_2$ e deve ser arredondada para "zero" de modo a gerar o menor erro. Logo,

$$(0.1)_{10} = (0.000\underbrace{1100110011}_{t=10}\underbrace{0011..}_{;})_2 \cong (0.0001100110011)_2$$

$$(0.1)_{10} \cong (0.0001100110011)_2 \cong (0.0999755859375)_{10}$$

Observe agora que $(0.1)_{10}$ era uma fração decimal exata e a convertemos para uma representação binária ilimitada, também exata enquanto for representada por um número de *bits* infinito. Ou seja, nesse exemplo, note que, na conversão de base, descartamos parte dos *bits* da representação binária, por limitação do número de *bits* representáveis, o que gerou um erro de arredondamento.

Perceba nas conversões dos **Exemplos 1.10** e **1.11** que:

a) $\left(0.03125\right)_{10} = \left(0.00001\right)_2$

$\left(0.03125\right)_{10} = 3125/10000 = 1/32 = 1/2^5$ (o denominador é uma potência da base 2)

b) $\left(0.1\right)_{10} = (0.000\underbrace{1100110011}_{t=10}\underbrace{0011...}) _2 \cong \left(0.0001100110011\right)_2$

$\left(0.1\right)_{10} = \left(1/10\right)_{10}$ (a fração não pode ser expressa por outra equivalente cujo denominador seja uma potência de 2)

Isso se deve ao fato de que um racional $X_\beta = p/q$ será limitado nessa base β se, e somente se, puder ser expresso na forma de $X_\beta = v/\beta^k$.

A seguir, vamos apresentar conversões importantes para entender o mecanismo de visualização de representações de base binária por meio de representações em base hexadecimal.

1.5.3 Conversões Diretas entre Binário e Hexadecimal

Sabemos que cada símbolo hexadecimal corresponde a quatro dígitos binários, pois $16^1 = 2^4$. Por exemplo, $\left(\mathrm{F}\right)_{16} = \left(1111\right)_2$ é armazenado em 4 *bits*. Então, fazemos a conversão direta associando a cada hexadecimal quatro dígitos binários. Ou seja, agrupamos os dígitos binários em grupos de quatro a partir da posição do ponto (vírgula). Caso seja necessário, completamos o grupo de quatro *bits* com zeros não significativos.

Exemplo 1.12: converta $(\mathrm{A1.B})_{16} = (?)_2$

Solução:

Como

$$(\mathrm{A})_{16} = (1010)_2$$

$$(1)_{16} = (0001)_2$$

$$(\mathrm{B})_{16} = (1011)_2$$

teremos:

$$(\mathrm{A1.B})_{16} = (1010\ 0001\ .\ 1011)_2$$

Exemplo 1.13: converta diretamente o registro de 16 *bits* $(1000101001101110)_2$ para hexadecimal.

Solução:

Agrupando os *bits* em grupos de quatro, temos:

$$\left(\underbrace{1000}_{8}\underbrace{1010}_{\mathrm{A}}\underbrace{0110}_{6}\underbrace{1110}_{\mathrm{E}}\right)_2 = (8\mathrm{A6E})_{16}$$

Exemplo 1.14: demonstre a mesma conversão do **Exemplo 1.13** através da conversão por parcelas decimais.

Solução:

Observe que

$$(1000101001101110)_2 =$$

$$\left(1*2^{15}+0*2^{14}+0*2^{13}+0*2^{12}+1*2^{11}+0*2^{10}+1*2^{9}+0*2^{8}+0*2^{7}+1*2^{6}+1*2^{5}+0*2^{4}+1*2^{3}+1*2^{2}+1*2^{1}+0*2^{0}\right)_{10}$$

fatorando cada grupo de quatro *bits*, em potências de $16 = 2^4$, conforme segue:

$$= \underbrace{1*2^{3}+0*2^{2}+0*2^{1}+0*2^{0}}_{8}\left(2^4\right)^3 + \underbrace{1*2^{3}+0*2^{2}+1*2^{1}+0*2^{0}}_{10}\left(2^4\right)^2 + \underbrace{0*2^{3}+1*2^{2}+1*2^{1}+0*2^{0}}_{6}\left(2^4\right)^1 + \underbrace{1*2^{3}+1*2^{2}+1*2^{1}+0*2^{0}}_{14}\left(2^4\right)^0$$

$$= 8*(16)^3 + \mathrm{A}*(16)^2 + 6*(16)^1 + \mathrm{E}*(16)^0 = (8\mathrm{A6E})_{16}$$

Nas conversões diretas entre as bases binária e hexadecimal não há perda de dígitos significativos (não há arredondamentos).

1.6 Representação Digital de Números

Conforme vimos na seção 1.1, nos sistemas computacionais, representamos um número na forma de notação em **ponto flutuante**, de maneira a racionalizar o armazenamento digital, pois, se utilizássemos um armazenamento em **ponto fixo** (ou vírgula fixa), seria necessário um número de posições (dígitos) no mínimo igual à variação dos limites dos expoentes.

Por exemplo, para obter a representação de uma calculadora científica comum, com menor positivo $mp = 1.0 * 10^{-99}$ e maior positivo $MP = 9.999999999 * 10^{+99}$, seriam necessárias:

a) 99 posições para a parte fracionária, entre $1.0 * 10^{-99}$ e 1, pois $1.0 * 10^{-99} = 0.\underbrace{000.....00001}_{99}$ tem 99 dígitos depois do ponto (vírgula).

b) 100 posições para a parte inteira, entre 1 e $9.99999999 * 10^{+99}$, pois $9.99999999 * 10^{+99} = \underbrace{9999999999000.....0000}_{100}$ tem 100 dígitos.

c) Mais uma posição para o sinal (definido por *s*), totalizando 200 posições no *display* de uma hipotética calculadora científica "de ponto fixo", conforme segue:

s		
	100 posições para a parte inteira	99 posições para a parte fracionária

Por isso, essas representações de ponto fixo ficaram limitadas às calculadoras para operações básicas.

Por outro lado, uma representação em ponto flutuante (ou vírgula flutuante), como a das calculadoras científicas comuns, funciona com pouco mais de **10 dígitos** e com as posições reservadas ao expoente e aos sinais.

Normalmente apresentam 10 dígitos decimais na tela e mais alguns dígitos internos extras, também chamados dígitos de "guarda", usados para ampliar a faixa de abrangência das operações aritméticas.

Então, uma representação em ponto flutuante é aquela em que o ponto se desloca entre os dígitos significativos da mantissa,

segundo certo padrão de normalização para a sua localização, se antes ou depois do primeiro dígito significativo, conforme vimos no tópico 1.1.

Exemplo 1.15: represente os números em notação científica ou ponto flutuante e sua forma fatorada:
Solução:

a) $\left(+0.0003501\right)_{10} = +3.501*10^{-4} = +\left(3*10^{0} + 5*10^{-1} + 0*10^{-2} + 1*10^{-3}\right)*10^{-4}$

b) $\left(+101.011\right)_{2} = \left(+1.01011\right)_{2} * 2^{2} = +(1*2^{0} + 0*2^{-1} + 1*2^{-2} + 0*2^{-3} + 1*2^{-4} + 1*2^{-5}) * 2^{2}$

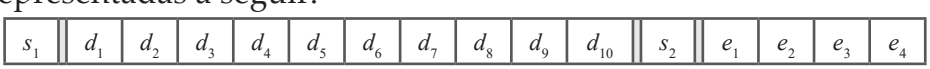
Na próxima seção, vamos exemplificar a representação digital de números em computadores, mostrando os padrões clássicos como registrado na literatura pertinente, a exemplo de Patterson e Hennessy (1998).

1.6.1 Padrão 16 *Bits* Original

A representação clássica da variável de 16 *bits*, utilizada em computadores com arquitetura de 16 *bits*, não é mais utilizada, mas a detalhamos aqui por motivos didáticos e históricos.

Em um sistema de representação em ponto flutuante e base binária ($\beta = 2$), com $t = 10$ dígitos binários (*bits*) na mantissa e expoentes limitados entre $I = -15$ e $S = +15$, ($15_{10} = 1111_{2}$), simbolicamente temos: $F(\beta, t, I, S) = F\left(2, 10, -15, +15\right)_{10}$.

Já para a representação esquemática de dígitos significativos binários, cada *bit* é alocado em um registro, conforme células representadas a seguir:



t dígitos significativos da mantissa f exp=expoente

s_1= sinal do número (da mantissa f)

t = número de dígitos significativos da mantissa f

s_2 = sinal do expoente

exp = expoente

Por convenção, temos:

Se $s_1 = 0 \Rightarrow$ número positivo

Se $s_1 = 1 \Rightarrow$ número negativo

$s_2 \Rightarrow$ idem

E, no registro total, temos 16 *bits*:

1 *bit* para o sinal dos dígitos significativos (mantissa), s_1

10 *bits* para armazenar os dígitos significativos da mantissa, f

1 *bit* para o sinal do expoente, s_2

4 *bits* para o módulo do expoente, exp

Nessa representação, os dígitos significativos são armazenados no padrão de normalização com $d_1 \neq 0$ posicionado à direita do ponto. Um número v armazenado nesse registro poderia ser interpretado por:

$$v = (-1)^{s_1} \ (0.f)_\beta \ 2^{\pm \exp}$$

ou

$$v = (-1)^{s_1} \ (0 \ . \ d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10})_\beta \ 2^{\pm \exp}$$

Exemplo 1.16: vamos representar $-(101.011)_2$ na variável de 16 *bits* estabelecida anteriormente.

Primeiramente, precisamos normalizar a posição do ponto (vírgula), conforme padrão adotado para 16 *bits*:

$-(101.011)_2 = -(0.1010110000)_2 * 2^{+3}$ (com 1º significativo d_1 à direita do ponto). Logo, sinal $s_1 = 1$, mantissa $f = (1010110000)_2$, convertendo o expoente para binário: $\exp = +(3)_{10} = +(0011)_2$, com sinal $s_2 = 0$, temos então o registro de 16 *bits*:

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

A seguir, apresentaremos os limites da representação de números Reais em ponto flutuante no padrão 16 *bits* apresentado.

1.6.1.1 Menor Positivo Representável (mp)

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

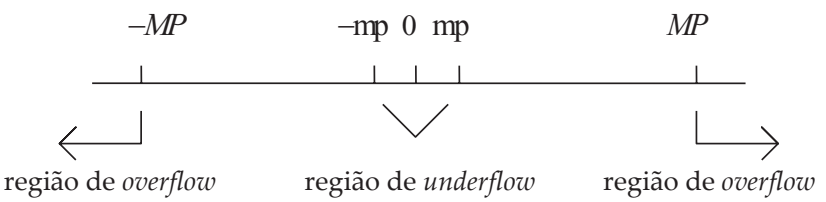
$$mp = +(0.1)_2 * 2^{-15} = \left(2^{-1} * 2^{-15}\right)_{10} = \left(2^{-16}\right)_{10} = (0.0000152587890625)_{10}$$

1.6.1.2 Maior Positivo Representável (MP)

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

$$MP = +(0.1111111111)_2 * 2^{15} = \left(2^{-1} + 2^{-2} + 2^{-3} + \ldots + 2^{-10}\right) * 2^{15} = (32736)_{10} \cong \left(1 * 2^{15}\right)$$

Na reta Real, temos a seguinte **representação** para $F(2,10,-15,+15)$:



Os limites de representação dos números negativos são simétricos aos limites positivos apresentados.

a) Região de **underflow**: $\{x \in \mathbb{R} / -mp < x < mp\}$, que compreende os números, em módulo, abaixo do mínimo representável e, caso surjam, são arredondados para o mais próximo entre $-\text{mp},\ 0,\ +\text{mp}$.

b) Região de **overflow**: $\{x \in \mathbb{R} / x < -MP \text{ e } x > MP\}$, que compreende os números, em módulo, acima do máximo armazenável e não são armazenados (normalmente travam o computador com alguma mensagem de erro).

Exemplo 1.17: se uma operação aritmética gerasse o número $(0.00001527)_{10}$, como poderíamos representá-lo em $F(2,10,-15,+15)_{10}$?

Solução:

Como esse valor não tem representação binária exata, nós o representaríamos pelo valor discreto mais próximo, no caso $(0.0000152587890625)_{10}$, que é o próprio mp.

1.6.1.3 Representação do Zero

A representação do zero é obtida sempre com mantissa nula e o menor expoente representável (*I*).

Exemplo 1.18: vamos representar o zero em $F(2,10,-15,+15)$.

Solução:

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

exp mínimo *I*

Devemos lembrar que esse número é o único escrito na forma "não normalizada", pois sua mantissa é toda zero.

Nos próximos exemplos, veremos o que poderia acontecer em uma operação de adição se o expoente do zero fosse diferente do limite inferior *I*.

Exemplo 1.19: simule a operação de adição: $0.000135 + 0.0$ em $F(10,4,-10,+10)$, considerando a representação do zero com expoente nulo $(0.0000 * 10^{0})$.

Solução:

$$\begin{array}{r} 0.1350 * 10^{-3} \\ \underline{+0.0000 * 10^{0}} \end{array}$$

Toda operação de adição/subtração de dois termos é efetuada com os seus dois expoentes iguais ao do maior termo (alinhamento de expoentes):

$$\begin{array}{r} 0.000135 * 10^{0} \\ \underline{+0.0000 \quad * 10^{0}} \\ \cong 0.0001 \quad * 10^{0} = 0.1000 * 10^{-3} \end{array}$$

Nesse exemplo, ocorre arredondamento com perda dos 2 dígitos significativos excedentes, 3 e 5, que não cabem dentro dos $t = 4$ registros da mantissa. Em calculadoras científicas, esses dígitos excedentes normalmente ficam guardados como dígitos internos de "guarda".

Exemplo 1.20: simule esta operação de adição: $0.000135 + 0.0$ em $F(10,4,-10,+10)$, considerando a representação do zero com expoente mínimo $I\left(0.0000 * 10^{-10}\right)$.
Solução:

$$\begin{array}{r} 0.1350 * 10^{-3} \\ \underline{+0.0000 * 10^{-10}} \end{array}$$

e alinhamento de expoentes pelo maior termo:

$$\begin{array}{r} 0.1350 * 10^{-3} \\ \underline{+0.0000 * 10^{-3}} \\ = 0.1350 * 10^{-3} \end{array}$$

No **Exemplo 1.20**, temos um resultado exato, sem arredondamentos, ou seja, o zero representado pelo menor expoente possível retrata corretamente o elemento neutro da operação de adição em meio digital, ou seja, não altera outro número.

A seguir, apresentaremos a quantificação dos elementos representáveis em notação de ponto flutuante.

1.6.1.4 Quantidade Máxima de Elementos Representáveis

Podemos notar que a distribuição de números representáveis em ponto flutuante é padronizada por meio de **tipos de variáveis** e é sempre **discreta** (somente alguns valores são representáveis), enquanto a distribuição de valores da reta Real é **contínua** (qualquer valor é representável).

Exemplo 1.21: represente os dois primeiros números positivos do sistema $F(2, 10, -15, +15)$ de 16 *bits*.

Solução:

1º positivo -

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

$$+\left(0.1000000000\right)_2 * \ 2^{-15} = \ \left(0.0000152587890625\right)_{10}$$

2º positivo -

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

$$+\left(0.1000000001\right)_2 * \ 2^{-15} = \left(0.000015288591384887695 3125\right)_{10}$$

Observe que existe uma distância entre esses dois números consecutivos discretos, enquanto na reta Real podem existir infinitos números entre dois números quaisquer. Além disso, a distribuição de números representáveis de $F(\beta, t, I, S)$ não é uniforme em $\mathbb{R}$, e o número máximo de elementos representáveis é uma combinação das possibilidades de preenchimento de cada registro.

Para cada expoente, existe uma quantidade fixa de números representáveis na mantissa *NM*, conforme a combinação de possibilidades de seus dígitos, por exemplo, em um sistema genérico de base β, com normalização $d_1 \neq 0$ depois do ponto (vírgula):

a) no 1º registro, não podemos ter o número 0 (normalização com $d_1 \neq 0$), logo d_1 pode assumir valores a partir de 1, totalizando $(\beta - 1)$ possibilidades; e

b) do 2º registro até o valor da posição t, podemos ter representados qualquer um dos β valores da base, para cada um dos $t-1$ registros restantes, ou seja, β^{t-1} possibilidades; logo, $NM = (\beta - 1)\beta^{t-1}$.

Para cada mantissa, existe uma quantidade fixa de expoentes representáveis *NE*, incluindo *S*, *I* e o expoente nulo:

$$NE = S - I + 1$$

Dessa forma, o número total de elementos positivos representáveis *NP* em uma variável genérica $F(\beta, t, I, S)$ é dado por:

$$NP(\beta, t, I, S) = NM * NE = (S - I + 1)(\beta - 1)\beta^{t-1}$$

Ao dobrar esse número para incluir os negativos e mais um representável para o zero, temos o número total de elementos representáveis *NR*:

$$NR(\beta, t, I, S) = 2(S - I + 1)(\beta - 1)\beta^{t-1} + 1$$

Exemplo 1.22: em $F(2, 3, -1, +2)$, temos as seguintes representações possíveis:

a)mantissas possíveis:

0.100
0.101
0.110
0.111

b) expoentes possíveis:

2^{-1}
2^{0}
2^{+1}
2^{+2}

A combinação de **quatro possibilidades de mantissas** para cada expoente da base ($NM = (\beta - 1)\beta^{t-1} = 4$ para $\beta = 2$ e $t = 3$) com **quatro possibilidades de expoentes** ($NE = S - I + 1 = 4$ para $S = 2$ e $I = -1$) definem o número total de positivos representáveis ($NP(\beta, t, I, S) = NM * NE = 16$).

De mesmo modo, incluimos as representações dos negativos e do zero, dobrando esse número e somando 1; logo, $NR = 33$.

Exemplo 1.23: em $F(2, 10, -15, +15)$ (binário de 16 *bits* totais), temos: $NP = 2 * (2-1) * (15-(-15)+1) * 2^{10-1} = 31745$ elementos representáveis, incluindo os positivos, os negativos e o zero.

Exemplo 1.24: em $F(10, 10, -99, +99)$ (calculadora científica comum de 10 decimais, com normalização $d_1 \neq 0$ antes do ponto (vírgula)), temos:

s_1	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	s_2	e_1	e_2

Como d_1 deve ser sempre diferente de zero, essa 1ª posição da mantissa tem nove possibilidades em vez de dez (d_1 está entre 1 e 9), enquanto as demais nove posições têm dez possibilidades de dígitos diferentes (d_i está entre 0 e 9), logo o número de mantissas diferentes é:

$$\text{NM} = (9) * 10 * 10 * 10 * 10 * 10 * 10 * 10 * 10 * 10 = 9 * 10^9$$
$$\text{NM} = (\beta - 1)\beta^{t-1} = (10 - 1) * 10^{10-1}$$

Já o número de expoentes depende somente dos seus limites:

$$NE = S + I + 1 = +99 - (-99) + 1$$

Logo, $\text{NR} = 2(\beta - 1)\beta^{t-1}(\text{S} - \text{I} + 1) + 1 = 3.582 * 10^{12}$ elementos.

A representação binária usada no padrão de 16 *bits* evoluiu juntamente com os computadores e atingiu uma forma mais otimizada de representação, incluindo a **polarização** dos expoentes, mais flexibilidade na normalização da mantissa para permitir maiores faixas de representação, mais **precisão** e **exatidão**, entre outras.

Confira os conceitos de precisão e exatidão na seção **Complementando...** ao final deste Capítulo.

Dessa evolução, em 1985 surgiu o padrão **IEEE 754**, que é amplamente utilizado para padronização de variáveis (PATTERSON; HENNESSY, 1998). Em 2008, o padrão **IEEE 754** referenciou oficialmente também o padrão otimizado de 16 *bits*, denominando-o de *binary16*.

1.6.2 Otimizações da Variável de 16 *Bits*, Segundo o Padrão IEEE 754

Otimizações para a variável de 16 *bits* foram adotadas, como ***Binary16***, com o objetivo de ampliar a sua representação, reduzindo as regiões de *underflow* e *overflow*.

A seguir, apresentaremos o detalhamento dessas otimizações.

1.6.2.1 Polarização

O padrão IEEE 754 adotou a **Polarização**, ou Excesso, para armazenamento dos expoentes, que é um valor acrescentado (em Excesso) a todos os expoentes de um sistema de representação em ponto flutuante, com o objetivo de tornar todos os expoentes positivos, suprimindo o sinal + e ampliando o valor do expoente superior (*S*). Naturalmente, todas as operações aritméticas devem considerar essa polarização introduzida e, no final, subtraí-la para imprimir os resultados.

Exemplo 1.25: na variável de 16 *bits* $F(2,\ 10,\ -15,\ 15)$, podemos usar uma polarização $p = +(15)_{10} = +(1111)_2$

Observe que os expoentes da variável de 16 *bits* ocupam 5 *bits* (destacados no registro em cinza), uma posição para o sinal do expoente e quatro para o seu módulo. Logo, os expoentes polarizados devem continuar cabendo nos cinco registros binários disponíveis:

$$I + p = -(15)_{10} + p = -(1111)_2 + p = -(1111)_2 + (1111)_2 = (00000)_2$$

$$S + p = +(15)_{10} + p = +(1111)_2 + p = +(1111)_2 + (1111)_2 = (\underline{11110})_2$$

5 bits

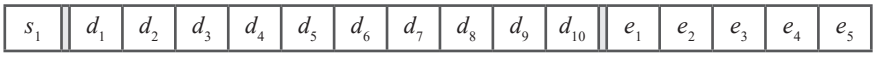
Como *I* e *S* polarizados têm agora o mesmo sinal (+), podemos suprimir s_2 e usar todos os cinco registros binários reservados ao expoente. Assim, os limites dos expoentes polarizados assumem os novos valores:

$$I = (00000)_2 = (0)_{10}$$

$$S = (11110)_2 = (30)_{10}$$

Podemos aproveitar melhor os 5 *bits* reservados ao expoente **ampliando** o maior valor possível do **expoente superior** e adotando um novo *S* como $(11111)_2 = (31)_{10}$. Note que pudemos incluir esse expoente extra porque o expoente zero original (não polarizado) usava duas representações, –0 e +0, e uma delas foi eliminada. Agora, o expoente zero polarizado é representado apenas pelo seu valor positivo, +0, que é representado sem a posição do sinal +.

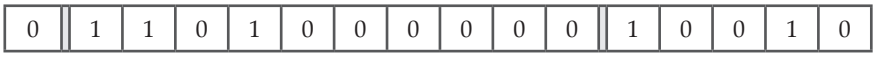
Na forma polarizada dessa variável de 16 *bits*, qualquer número *v* representado poderá ter esta forma:



t dígitos significativos da mantissa *f* — *e* polarizado

Agora *s* é único, e $v = (-1)^s (0.f)_2 \, 2^{e-15}$.

Exemplo 1.26: no registro binário, a seguir, com exponte polarizado com $p = +(15)_{10}$, qual é o decimal representado?



s *t* = 10 dígitos significativos da mantissa *e*

Solução:

Temos que $v = (-1)^s (0.f)_2 \, 2^{e-15}$,

$s = 0$, $f = 110100000$ e $e = (10010)_2 = (18)_{10}$

$v = (-1)^0 (0.110100000)_2 \, 2^{18-15} = +(110.1)_2 = (6.5)_{10}$

1.6.2.2 Não Armazenamento do Primeiro *Bit* Não Nulo da Mantissa

Outra otimização adotada pelo padrão IEEE 754 foi o **não armazenamento do primeiro** *bit* **da mantissa** $d_1 \neq 0$, que é sempre **unitário**. Nessa otimização, usamos uma representação implícita do primeiro *bit* unitário e abrimos uma posição binária para armazenamento de um novo *bit* significativo. Logo, o número total de dígitos binários significativos é ampliado para $1+t$, a mantissa *f* representada passa a conter os *bits* de d_2 à d_{t+1} e a posição normalizada do ponto (vírgula) é à direta de $d_1 = 1$, ou seja:

$$v = (-1)^s (1.f)_2 \, 2^{e-15}$$

s_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	e_1	e_2	e_3	e_4	e_5
s	*t* = 10 dígitos significativos de *f*, depois do ponto										*e* polarizado				

Observe que $d_1 = 1$ está implícito antes do ponto e não aparece nesse registro.

Exemplo 1.28: no registro binário, a seguir, com polarização e com primeiro *bit* implícito, qual é o decimal representado?

0	1	1	0	1	0	0	0	0	0	0	1	0	0	1	0
s	*t* = 10 dígitos significativos de *f*, depois do ponto										*e* polarizado				

Solução:

Temos que $v = (-1)^s (1.f)_2 \, 2^{e-15}$

$s = 0$, $f = 110100000$ e $e = (10010)_2 = (18)_{10}$

$v = (-1)^0 (1.110100000)_2 \, 2^{18-15} = +(1110.1)_2 = (14.5)_{10}$

Observe que, com os mesmos *bits* representados, podemos agora armazenar números com mais dígitos significativos totais. Logo, o valor do *MP* foi ampliado para:

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

$$MP = +\left(1.1111111111\right)_2 * 2^{30-15} = (2^0 + 2^{-1} + \ 2^{-2} + \ 2^{-3} + \ ... \ + \ 2^{-10}) * 2^{15} = \left(65504\right)_{10} \cong \left(2^{16}\right)$$

(*MP* era $(32736)_{10}$ (conforme vimos na seção 1.6.1.2)

1.6.2.3 Flexibilidade na Normalização da Mantissa

O padrão IEEE 754 também adotou a **flexibilidade na normalização da mantissa** para ampliar a faixa de abrangência de números pequenos, como para o primeiro número positivo *mp*, reduzindo a região de *underflow*.

Como vimos anteriormente, o *mp* era

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

$$mp = +(0.1000000000)_2 * 2^{-15} = (2^{-16}) = (0.0000152587890625)_{10}$$

Com a otimização, que reposicionou a normalização do $d_1 \neq 0$ colocando o ponto à sua direita, deixando $d_1 = 1$ implícito, o novo valor de *mp* seria:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$mp = +(1.0000000000)_2 * 2^{0-15} = (2^{-15})_{10}$, que seria maior ainda do que o anterior.

Essa normalização da mantissa foi flexibilizada especificamente para o caso de representação de números com o menor expoente polarizado possível, ou seja, para $e = (00000)_2$, a **normalização** com $d_1 = 1$ implícito foi **eliminada**. Nesse caso, a mantissa *f* pode assumir qualquer valor, mas o **valor mínimo do expoente**, não polarizado, deve ser $e = -14_{10}$, e o novo *mp* se torna:

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

sem normalização (mantissa sem restrições) $\Rightarrow$ para $e = 00000_2$

Isso ocorre para manter uma continuidade na representação entre:
- menor representável com "$e = 1$", na faixa $0 < e < 31$, dado por $v = (-1)^s 2^{(1-15)}(1.0000000000)_2$ usando a normalização; e
- maior representável com "$e = 0$", dado por $v = (-1)^s 2^{(-14)}(0.1111111111)_2$ sem normalização.

Logo,

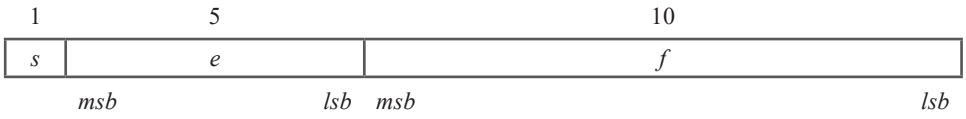
$$mp = +(0.0000000001)_2 * 2^{-14} = (2^{-24})_{10} = 5.960464478 * 10^{-8}$$

(*mp* era $(2^{-16})_{10} = 1.52587890625 * 10^{-5}$, conforme vimos em 1.6.1.1)

1.6.2.4 Identificação da Região de *Overflow*

Por último, uma convenção adicional foi adotada no padrão IEEE 754 para **identificar a região de *overflow***: a todo número gerado dentro dessa região, atribuímos expoente igual ao seu limite superior: $S = (31)_{10}$.

O novo padrão IEEE 754 otimizado para 16 *bits* (2 *bytes*) ficou nesta forma final:



Em que:

$s = 0 \Rightarrow v$ positivo e $s = 1 \Rightarrow v$ negativo

e = expoente polarizado

f = mantissa fracionária, a partir do segundo *bit* significativo (o primeiro *bit* significativo será sempre unitário e não armazenado neste registro)

polarização = $(15)_{10} = (01111)_2$

msb = *bit* mais significativo

lsb = *bit* menos significativo, de cada *e* e *f*

Um número *v* armazenado nesse registro é interpretado conforme o valor de seu expoente polarizado *e* da seguinte forma:

Se $0 < e < 31$, então $v = (-1)^s 2^{e-15} (1.f)_2$

Se $e = 0$ e $f \neq 0$, então $v = (-1)^s 2^{-14} (0.f)_2$

Se $e = 0$ e $f = 0$, então $v = (-1)^s 2^{-14} (0.0)_2 = zero$

Se $e = 31 = 2^5 - 1$, então *v* pertence à região de *overflow*

Esse padrão ***binary16*** com 16 *bits* praticamente não é mais utilizado, pois atualmente temos custos de memória muito reduzidos e a arquitetura comum nos computadores evoluiu para 64 *bits*, o que nos permite usar variáveis de 64 *bits* com a mesma velocidade de acesso de variáveis de 32 ou 16 *bits*.

1.7 Representações Numéricas, Segundo o Padrão IEEE 754

O padrão IEEE 754 de 1985 é amplamente utilizado em linguagens de programação comerciais e o apresentaremos na sua forma esquemática para representação na forma de variáveis reais.

1.7.1 Variável de 32 *Bits* – *Binary32*, Tipo *Float* do C, Precisão Simples

No padrão de **4 *bytes*** (1 *byte* = 8 *bits*) ou 32 *bits* (precisão de 7 a 8 dígitos decimais significativos equivalentes), podemos representar um número Real *v* por um registro de 32 *bits*:

que é análogo ao padrão ***Binary16***, apenas com faixa de valores mais ampliada, como polarização $(127)_{10} = (01111111)_2$, em que um número *v* armazenado em ***binary32*** é interpretado conforme o valor de seu expoente polarizado *e*.

Dessa forma:

Se $0 < e < 255$, então $v = (-1)^s \, 2^{e-127} \, (1.f)_2$;

Se $e = 0$ e $f \neq 0$, então $v = (-1)^s \, 2^{-126} \, (0.f)_2$;

Se $e = 0$ e $f = 0$, então $v = (-1)^s \, 2^{-126} \, (0.0)_2$ = zero; e

Se $e = 255 = 2^8 - 1$, então *v* pertence à região de *overflow*.

A apresentação desses registros binários em computadores digitais normalmente é feita em *bytes*. Nesse padrão, temos 4 *bytes*, e cada *byte* é composto de 2 registros hexadecimais (8 *bits* para cada *byte*).

Exemplo 1.28: defina o decimal armazenado em

0	0 0 0 0 0 0 1 1	1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
s	*e* polarizado	*t* = 10 dígitos significativos de *f*, depois do ponto

Solução:

Observe que

$$s = 0 \Rightarrow v \text{ positivo}$$

$$e = (00000011)_2 = 1*2^1 + 1*2^0 = 3_{10}$$

$$f = (10010000000000000000000)_2$$

como $e = 3_{10}$, logo $0 < e < 255$, então $v = (-1)^s 2^{e-127}(1.f)_2$

Assim, o número *v* armazenado é:

$$v = (-1)^0 2^{3-127}(1.10010000000000000000000)_2$$

$$v = (+)2^{-124}(1*2^0 + 1*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4} + 0*2^{-5} + ...)_{10}$$

$$v = (+)4.70197740328915*10^{-38}(1 + 0.5 + 0 + 0 + 0.0625 + 0...)_{10}$$

$$v = +7.346839693*10^{-38}$$

Exemplo 1.29: represente $(0.1)_{10}$ na base binária no padrão IEEE de 32 *bits*.

Solução:

Para tal, temos duas formas de conversão possíveis.

Primeira forma de conversão: podemos converter $(0.1)_{10}$ diretamente para binário, conforme efetuamos no **Exemplo 1.11**:

$$(0.1)_{10} = (0.0001100110011001100110011001100110011...)_2$$

e adequar o resultado diretamente ao padrão 32 *bits*, obtendo *s*, *e*, *f*.

Primeiramente, normalizamos a mantissa, colocando o 1º dígito significativo (mais significativo), unitário, antes do ponto (vírgula), na forma $(1.f)_2$, para obter o expoente não polarizado original ($^{e-127}$), supondo que e esteja na faixa: $0 < e < 255$ em que $v = (-1)^s 2^{e-127} (1.f)_2$:
Assim,

$$(0.10)_{10} = 2^{-4} (1.10011001100110011001100110011001100110011...)_2$$

O expoente original, -4, não polarizado, deve ser equivalente ao expoente $(e-127)$ da fórmula $v = (-1)^s 2^{e-127} (1.f)_2$, também não polarizado, logo: $(e-127) = -4$
Então, o expoente e polarizado é o expoente original -4 somado à polarização +127:

$$e = -4 + 127 = 123$$

Como $e = 123$ está no **intervalo** $0 < e < 255$, então confirmamos a representação do número $(0.1)_{10}$ pela formula prevista $v = (-1)^s 2^{e-127} (1.f)_2$
Precisamos ainda definir s e f:

Caso e estivesse fora da faixa $0 < e < 255$, teríamos que adequar a representação a outro formato do mesmo padrão IEEE, com $e = 0$ ou com $e = 255$.

a) o sinal de $+(0.1)_{10}$ é positivo, logo $s = 0$, então:

$$+(0.1)_{10} = (-1)^0 2^{(123-127)} (1.10011001100110011001100\ 11001100...)_2$$

b) a mantissa f, composta dos *bits* depois do 1º *bit* unitário, tem infinitos dígitos significativos (é uma dízima periódica). Por isso, precisamos limitá-la aos 23 *bits* do registro padrão IEEE de 32 *bits* totais. Essa limitação de registros segue os mesmos princípios do arredondamento decimal. Logo, precisamos arredondar a parcela do número binário a partir do 24º *bit*, depois do ponto (vírgula), e gerar o menor erro possível:

$$+(0.1)_{10} = (-1)^0 2^{(123-127)} (\underline{1.10011001100110011001100}\ \underbrace{11001100...}_{\text{arredondamento}})_2$$

Então, a parcela que temos de eliminar, $(0.11001100...)_2 * 2^{-23}$, deve ser avaliada segundo os critérios de arredondamento, ou seja, podemos aproximá-la para $(0.)_2 * 2^{-23}$ ou para $(1.)_2 * 2^{-23}$. Como devemos escolher o valor aproximado com menor erro e verificamos que $\left(0.11001100...\right)_2 > (0.5)_{10}$, então fazemos o arredondamento $\left(0.11001100...\right)_2 \cong (1.)_2$.

Observe que:

i) toda fração binária de mais de um *bit* significativo que comece por 1 é maior $(0.5)_{10}$, e que comece por 0 é menor $(0.5)_{10}$; e

ii) se a fração binária for de um "único" *bit* significativo e "unitário", então a fração binária será equivalente a $(0.5)_{10}$ e deve seguir o critério do *bit* anterior "par ou ímpar", conforme previsto no processo de arredondamento.

Logo, pelos mesmos critérios de minimização de erros, o nosso número fica arredondado para cima, somando 1 no seu último *bit*:

$$+(0.1)_{10} \cong (-1)^0 2^{(123-127)} (1.10011001100110011001100)_2$$
$$\underline{+1}$$
$$+(0.1)_{10} \cong (-1)^0 2^{(123-127)} (1.10011001100110011001101)_2$$

Também devemos converter o valor do expoente *e* para binário: $(e-127) = -4 \;\Rightarrow\; e = (123)_{10} = (01111011)_2$ (*e* deve ser completado com "zero(s)" à esquerda para ficar com 8 *bits*).

Em resumo:

$s = 0$

$e = (123)_{10} = (01111011)_2$

$f = (10011001100110011001101)_2$

0	0 1 1 1 1 0 1 1	1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1
s	e	f

Segunda forma de conversão: alternativamente, podemos converter $+(0.1)_{10}$ diretamente para a fórmula do padrão 32 *bits*, supondo inicialmente que *e* esteja na faixa $0 < e < 255$:

$$+(0.1)_{10} = (-1)^s \, 2^{e-127} \left(1.f\right)_2$$

Assim, temos uma equação com três incógnitas, *s*, *e*, *f*, sendo $s = 0$, para números positivos; e *e*, *f*, incógnitas que precisamos definir, mas *e* é um inteiro, então considerando o valor mínimo de *f*, ou seja, $f = 0$, para poder estimar o valor de *e*, temos:

$$+(0.1)_{10} = (-1)^0 \, 2^{e-127} \left(1.0\right)_2$$

f = 0 representa uma mantissa menor do que a verdadeira, pois $1 \leq (1.f)_2 < 2$.

Nesse caso, **determinamos um valor de *e* maior do que o verdadeiro**, mas *e* terá a parte inteira correta:

$$(0.1)_{10} = \; 2^{e-127}$$

$$e = \; \left(123.6781...\right)_{10}$$

Logo, tomamos o menor inteiro $e = (123)_{10}$ ("eliminamos" a sua parte fracionária). Verificamos que a suposição inicial para o expoente *e* na faixa $0 < e < 255$ é válida, ou teríamos que tentar armazenar o nosso número em outra fórmula, com $e = (0)_{10}$ ou com $e = (255)_{10}$.

Agora, determinamos o valor de *f*, com os valores de *s* e *e* obtidos anteriormente:

$$+(0.1)_{10} = (-1)^0 \, 2^{123-127} \left(1.f\right)_2$$

$$(1.f)_2 = (0.1)_{10} / 2^{123-127}$$

$(1.f)_2 = (1.6)_{10}$, em que $(1.f)_2$ deve ser sempre um número entre 1 e 2

Por fim, convertendo $(1.6)_{10}$ para binário, temos:

$$+(1.6)_{10} = (\underline{1.10011001100110011001100} \; \underbrace{11001100...}_{\text{Arredondamento}})_2$$

Com a parcela a ser arredondada $(0.11001100...)_2 > (0.5)_{10}$, aproximamos para $(1.)_2$ e a somamos no último *bit*:

$$\begin{array}{r} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad +1 \\ +(1.6)_{10} \cong (1.10011001100110011001100)_2 \\ \hline +(1.6)_{10} \cong (1.10011001100110011001101)_2 \end{array}$$

Gerando os mesmos resultados do item (a) anterior:

0	0 1 1 1 1 0 1 1	1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1
s	*e*	*f*

Observe que esse número binário armazenado foi arredondado para cima, ou seja, o número binário exato (dízima periódica) foi aproximado para um valor superior ao exato, conforme arredondamento da fração $(0.11001100...)_2 \cong (1.)_2$, que alterou o último *bit* para cima (destacado com sublinhado).

Exemplo 1.30: converta os 32 *bits* do **Exemplo 1.29** do padrão IEEE para o número decimal equivalente.

Solução:

Interpretando os 32 *bits*, podemos converter o registro para decimal, conforme segue:

$$s = 0$$

$$e = (123)_{10} = (01111011)_2$$

$$f = (10011001100110011001101)_2$$

Logo, se $0 < e < 255$, então:

$$v = (-1)^0 2^{(123-127)} (1.10011001100110011001101)_2$$

$$v = (+)2^{-4}(1*2^0 + 1*2^{-1} + 0 + 0 + 1*2^{-4} + 1*2^{-5} + 0 + 0 + 1*2^{-8} + 1*2^{-9} + 1*2^{-12} + 1*2^{-13} + 1*2^{-16} + 1*2^{-17} + 1*2^{-20} + 1*2^{-21} + 1*2^{-23})_2$$

$v = 0.100000001490116$ (com 16 decimais significativos).

Os dígitos sublinhados representam os arredondamentos gerados na representação do número decimal original 0.1 exato para sua representação em binária, no padrão IEEE de 32 *bits*.

Exemplo 1.31: calcule o erro de arredondamento cometido na conversão do decimal 0.1 para a representação binária padrão IEEE de 32 *bits* do **Exemplo 1.29**.

Solução:

O *Erro* é sempre calculado por meio de uma comparação entre o valor aproximado obtido *VA* e o seu valor exato *VE* (quando disponível). Nesse caso, queremos o erro de:

$VA = 0.100000001\underline{490116}$ (valor armazenado em binário, com erro de arredondamento) e temos $VE = 0.100000000000000$ (valor exato, disponível na representação decimal).

Logo,

$$Erro = |VA - VE| \text{ (Erro absoluto)}$$
$$Erro = |0.100000001\underline{490116} - 0.100000000000000|$$
$$Erro = 0.000000001\underline{490116}$$

Podemos apresentar esse erro de forma relativa ou percentual:

$$Erro\,\text{Re}lativo = \left|\frac{VA - VE}{VE}\right| = \left|\frac{0.100000001490116 - 0.1}{0.1}\right| = 1.490116 * 10^{-8}$$

$$Erro\,\text{Re}lativo\ \% = \left|\frac{VA - VE}{VE}\right| 100\% = \left|\frac{0.100000001490116 - 0.1}{0.1}\right| 100\% = 1.490116 * 10^{-6}\%$$

Note que esse erro pode ser calculado de forma exata em uma calculadora científica decimal.

Exemplo 1.32: represente o zero e os limites da variável padrão IEEE de 32 *bits*.

Solução:

a) Representação do Zero: $\begin{cases} s = 0 \\ e = 00000000 \\ f = 00000000000000000000000 \end{cases}$

0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$$zero = (-1)^0 2^{-126}(0.)_2$$

b) Representação do mp: $\begin{cases} s = 0 \\ e = 00000000 \\ f = 00000000000000000000001 \end{cases}$

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

$$mp = (-1)^0 2^{-126}(0.00000000000000000000001)_2$$

$$mp = (+)2^{-126}(1 * 2^{-23}) = 1.4012985 * 10^{-45}$$

(menor *e* e menor *f*)

c) Representação do MP: $\begin{cases} s = 0 \\ e = 11111110 = (254)_{10} \\ f = 11111111111111111111111 \end{cases}$

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

$$MP = (-1)^0 2^{254-127}(1.11111111111111111111111)_2$$

$$MP = 2^{127}\left(1.99999988\right)_{10} = \left(3.4028235 * 10^{38}\right)_{10}$$

(maior *e* e maior *f*)

Exemplo 1.33: defina as regiões de *underflow* e *overflow* da variável IEEE de 32 *bits*.

Solução:

região de ***underflow***: $\left\{x \in \mathbb{R} \,/\, -1.4012985 * 10^{-45} < x < 1.4012985 * 10^{-45}\right\}$

região de ***overflow***: $\left\{x \in \mathbb{R} \,/\, x < -3.4028235 * 10^{38} \text{ e } x > +3.4028235 * 10^{38}\right\}$

Exemplo 1.34: apresente na forma hexadecimal a representação binária obtida no **Exemplo 1.29**:

0	0 1 1 1 1 0 1 1	1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 <u>1</u>

Solução:

Representando os 32 *bits* em grupos de 4 *bits*, temos:

0011 1101 1100 1100 1100 1100 1100 1101

Desse modo, efetuamos a conversão direta de cada grupo de 4 *bits* para um hexadecimal:

0011 1101 1100 1100 1100 1100 1100 1101
3 D C C C C C D

8 *bytes* (16 hexadecimais) equivalentes: **3D CC CC CD**

Esta representação hexadecimal pode ser apresentada na forma de 4 *bytes* invertidos como CD CC CC 3D (no caso de compiladores Pascal).

1.7.2 Variável de 64 *Bits* – *Binary64*, Tipo *Double* do C, Precisão Dupla

Nesse padrão 8 *bytes* ou 64 *bits* (precisão de 16 a 17 dígitos decimais significativos equivalentes), podemos representar um número Real *v* por um registro de 64 *bits*:

Em que a polarização $p = (1023)_{10} = (011111111111)_2$.

Um número *v* armazenado no registro 64 *bits* é interpretado da seguinte forma:

Se $0 < e < 2047$, então $v = (-1)^s \, 2^{e-1023} \, (1.f)_2$

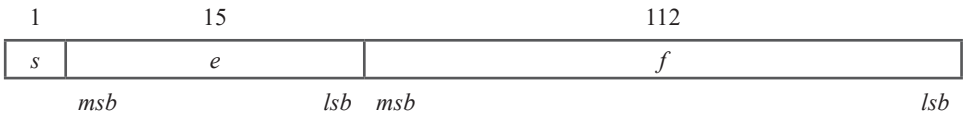
Se $e = 0$ e $f \neq 0$, então $v = (-1)^s \, 2^{-1022} \, (0.f)_2$

Se $e = 0$ e $f = 0$, então $v = (-1)^s \, 2^{-1022} \, (0.0)_2 = zero$

Se $e = 2047 = 2^{11} - 1$, então v pertence à região de *overflow*

1.7.3 Variável de 128 *Bits* – *Binary128,* Precisão Quádrupla

No padrão 16 *bytes,* ou 128 *bits* (precisão de 34 a 35 dígitos decimais significativos equivalentes), podemos representar um número Real *v* por um registro de 128 *bits:*



Em que a polarização $p = (16383)_{10} = (01111111111111)_2$.

Um número *v* armazenado no registro 128 *bits* é interpretado da seguinte forma:

Se $0 < e < 32767$, então $v = (-1)^s 2^{e-16383} (1.f)_2$

Se $e = 0$ e $f \neq 0$, então $v = (-1)\ 2^{16382} (0.f)$

Se $e = 0$ e $f = 0$, então $v = (-1)^s 2^{-16382} (0.0)_2 = zero$

Se $e = 32767 = 2^{15} - 1$, então *v* pertence à região de *overflow*

1.8 Representação de Variáveis Inteiras: padrão ANSI C

Os formatos de representação de variáveis do tipo inteiras podem seguir diversos padrões e tamanhos. Vamos apresentar um dos tipos para mostrar como a representação de inteiros também pode gerar erros numéricos.

1.8.1 *Short Int*

Short int são tipos inteiros limitados à faixa entre –32768 e +32767, correspondendo ao armazenamento como 2 *bytes* (16 *bits*), na qual os inteiros "negativos" são armazenados em forma de complemento de dois.

Exemplo 1.35: defina os limites da variável *short int*.

Solução:

Zero $0_{10} = (0000\ 0000\ 0000\ 0000)_2 = (0000)_{16}$

Maior Positivo $+32767_{10} = (0111\ 1111\ 1111\ 1111)_2 = (7FFF)_{16}$

Menor Negativo $-32768_{10} = -(1000\ 0000\ 0000\ 0000)_2$

Observe que O 1º *bit* "zero", indica um número positivo.

Observe que esse menor negativo não será armazenado com o sinal "–" e sim por complemento de dois. Todos os inteiros negativos s*hort int* são representados por complemento de dois, que é o seu complemento de um (diferença de cada dígito do número negativo em relação a um) somado à unidade. Então, o menor negativo será armazenado como:

$$\begin{aligned}\text{Menor Negativo } -32768_{10} = & -(1000\ 0000\ 0000\ 0000)_2 \\ & (0111\ 1111\ 1111\ 1111)_2 \rightarrow \text{complemento de um} \\ & +1 \rightarrow \text{soma 1} \\ \hline & (1000\ 0000\ 0000\ 0000)_2 \rightarrow \text{complemento de dois}\end{aligned}$$

Observe que o 1º *bit* é "um", o que indica um número negativo.

Logo, o menor negativo é representado por:

$$-32768_{10} = (1000\ 0000\ 0000\ 0000)_2 = (8000)_{16} \text{ (sem sinal "–")}$$

Agora observe o que acontece quando executamos a soma de uma "unidade" com o "maior positivo" da variável *short int*, 32767_{10}, ou seja, quando calculamos um número na sua região de ***overflow***:

$$\begin{aligned}\text{Unidade} \quad & 1_{10} = (0000\ 0000\ 0000\ 0001)_2 = (0001)_{16} \\ +\text{Maior Positivo} \quad & +32767_{10} = (0111\ 1111\ 1111\ 1111)_2 = (7FFF)_{16} \\ \hline & 1_{10} + 32767_{10} = (1000\ 0000\ 0000\ 0000)_2 = (8000)_{16}\end{aligned}$$

Complemento de dois

Um número iniciado pela unidade é um negativo armazenado como complemento de dois, que precisa ser convertido de volta ao seu valor negativo original, fazendo um complemento de dois. Então, vamos interpretar qual é o verdadeiro número armazenado por $1_{10} + 32767_{10} = (1000\ 0000\ 0000\ 0000)_2 = (8000)_{16}$:

$(1000\ 0000\ 0000\ 0000)_2$

$(0111\ 1111\ 1111\ 1111)_2 \rightarrow$ complemento de um

$+\ 1 \rightarrow$ soma 1

$-(1000\ 0000\ 0000\ 0000)_2 = -(8000)_{16} = -(32768)_{10}$

Logo, $1_{10} + 32767_{10} = -(32768)_{10}$

Esse resultado armazenado em s*hort int* é interpretado como um **complemento de dois** (por iniciar com 1), portanto é um número negativo.

Atenção! Cuidado com as operações com inteiros, pois podemos achar que $1_{10} + 32767_{10} = (32768)_{10}$, mas estamos armazenando, na verdade, o primeiro número negativo -32768_{10}. Logo, **não podemos ultrapassar os limites dos tipos inteiros**, porque os resultados obtidos trocam de sinal e normalmente não existem avisos aos usuários, ou seja, podemos estar executando algoritmos com números inconsistentes e sem saber disso.

Existem outras formas derivadas da variável *short int* de 16 *bits*, como *long int,* com 32 *bits*, e *long long int,* com 64 *bits*.

1.9 Tipos de Erros Existentes em Representações Digitais

É muito importante conhecer as possibilidades de erros das representações numéricas digitais executadas em calculadoras ou computadores e entender as suas causas, para poder estabelecer a confiabilidade de um algoritmo e de seus resultados.

A seguir, confira os principais tipos de erros que podem existir em representações digitais.

1.9.1 Erros Inerentes

Erros inerentes são aqueles existentes nos **dados de entrada** de um algoritmo numérico. Decorrem, por exemplo, de medições experimentais, de outras simulações numéricas *etc*.

1.9.2 Erros de Arredondamento

Os erros de arredondamento ocorrem quando são desprezados os dígitos menos significativos, que não são fisicamente confiáveis na representação numérica, ou estão além da capacidade de armazenamento.

1.9.2.1 Arredondamento Manual

Como já vimos, arredondamento é o processo de eliminação de dígitos menos significativos de um número. O objetivo é sempre representar o número com menor quantidade de dígitos significativos totais e ainda minimizar os erros decorrentes dessa redução de significativos, conforme tópico sobre Arredondamento Decimal na seção **Complementando...** ao final deste Capítulo.

Exemplo 1.36: represente os números listados, a seguir, com quatro dígitos significativos:

69.348 ≅69.35 → parcela descartada 0.8 é maior que 0.5 → +1 no dígito anterior.
69.34433 ≅69.34 → parcela descartada 0.433 é menor que 0.500 → dígito anterior inalterado.
69.335 ≅69.34 → parcela descartada é 0.5 e dígito anterior impar → +1 no dígito anterior.
69.345 ≅69.34 → parcela descartada é 0.5 e dígito anterior par → dígito anterior inalterado.

No **Exemplo 1.36**, realizamos o arredondamento de forma ponderada, baseados em critérios de minimização de erro gerado. Mas também podemos realizar o arredondamento por cancelamento puro: quando a parte indesejada do número é simplesmente cancelada, independentemente do seu valor, assumindo um erro de arredondamento global maior.

1.9.2.2 Arredondamento Digital

O arredondamento de representações digitais, em calculadoras ou computadores, acontece por limitação no número de dígitos significativos (tamanho de registro limitado) disponíveis para armazenamentos de: Racionais ilimitados, Irracionais, com mudança de base e operações aritméticas.

A tradicional aritmética de ponto flutuante é aplicada a problemas de natureza contínua. Devido às limitações de representação dos números no computador, precisamos utilizar o arredondamento para representá-los de forma discreta no mundo digital.

A seguir, vamos apresentar alguns exemplos de armazenamento digital.

1.9.2.2.1 Armazenamento de Racionais Ilimitados

Exemplo 1.37: represente a fração $(1/3)_{10}$ em $F(10,10,-99,+99)$ (normalização usual: ponto (vírgula) depois do 1º dígito significativo).

Solução:

$$(1/3)_{10} = (\underbrace{3.333333333}_{t=10}333...*10^{-1})_{10}$$

$$(1/3)_{10} \neq (3.333333333*10^{-1})_{10} \rightarrow \text{Representação arredondada de } (1/3)_{10}$$

Exemplo 1.38: represente a fração decimal $(1/10)_{10}$ na variável binária de 16 *bits* $F(2,10,-15,+15)$ IEEE 754.

Solução:

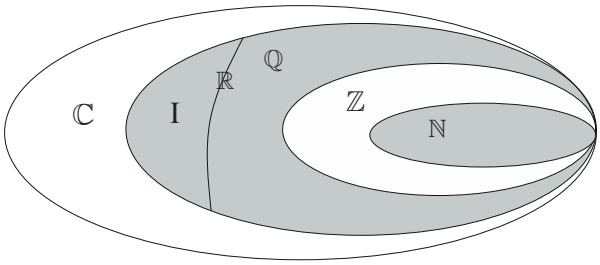
$$(1/10)_{10} = (1/1010)_2 = (1.\underbrace{1001100110}_{t=10}0110011...)_2 * 2^{-4} \rightarrow \text{dízima periódica}$$

Logo, $(1/10)_{10} \cong (1.1001100110)_2 * 2^{-4}$

1.9.2.2.2 Armazenamento de Irracionais

O conjunto dos números Irracionais I compreende todas as representações por meio de dízimas não periódicas e infinitas, o qual, unido aos números Racionais $\mathbb{Q}$ gera o conjunto dos Reais $\mathbb{R}$ ($\mathbb{Q} \cup$ I) , conforme Figura 1.2.

Figura 1.2: Diagrama dos Conjuntos Numéricos



Fonte: Elaboração própria

Existem representações que generalizam os números como os Complexos $\mathbb{C}$. Um Complexo com parte imaginária nula se torna um Real $\mathbb{R}$, logo $\mathbb{C} \supset \mathbb{R}$.

Exemplo 1.39: represente em $F(10,10,-99,+99)$ os seguintes Irracionais.

Solução:

a) $\pi = 3.141592653589...$

$\pi = (3.141592653\underline{589...})_{10} * 10^0$

$\pi \cong (3.141592654)_{10} * 10^0$

b) $\sqrt{2} = 1.414213562373...$

$\sqrt{2} = (1.414213562\underline{373...})_{10} * 10^0$

$\sqrt{2} \cong (1.414213562)_{10} * 10^0$

1.9.2.2.3 Armazenamento com Mudança de Base

Conforme vimos, a representação de números na base binária é amplamente utilizada em computadores, devido às suas

vantagens na representação de dados e na implementação de operações aritméticas.

Assim, toda grandeza física, expressa inicialmente em base decimal, é armazenada nos computadores e operada em base binária, por isso são necessárias conversões entre as bases decimal-binária e vice-versa, podendo a primeira ser fonte de erro de arredondamento.

No **Exemplo 1.40**, apresentaremos um caso de erro, na sua forma percentual, do armazenamento com mudança de base do **Exemplo 1.38**.

Exemplo 1.40: calcule o erro de arredondamento percentual ocorrido no armazenamento da fração decimal exata $(0.1)_{10}$ na variável binária de 16 *bits* $(2,10,-15,+15)$ IEEE 754.

Solução:

$(0.1)_{10} = (0.0001100110011\underline{0011}....)_2$ → gerou dízima periódica binária.

$(0.1)_{10} = 2^{-4} * (1.\underbrace{1001100110}_{t=10}\underbrace{0110011}_{arredondar}...,)_2$ → antes do arredondamento.

$(0.1)_{10} \cong 2^{-4} * (1.1001100110)_2 = (0.099975585)_{10}$ → depois do arredondamento

Nesse caso, podemos calcular o **erro de arredondamento** em uma calculadora decimal, pois temos o Valor Exato *VE* original e o Valor Aproximado *VA* pós-arredondamento, ambos na base decimal:

$VE = (0.1)_{10}$

$VA = 2^{-4} * (1.1001100110)_2 = (0.099975585)_{10}$

$$Erro\text{Re}lativo\ \% = \left|\frac{VA - VE}{VE}\right| * 100\% = 0.02441\%$$

1.9.2.2.4 Armazenamento de Resultados de Operações Aritméticas

Os operadores lógicos e aritméticos operam números armazenados em variáveis, representadas em ponto flutuante e/ou inteiras,

e consequentemente ficam com abrangência limitada conforme a notação adotada pela variável utilizada para o armazenamento dos resultados. Vejamos os exemplos a seguir.

Exemplo 1.41: efetue a adição de $a = 0.0165$ e $b = 10.51$ em $F(10,4,-10,+10)$.

Solução:

Representação em ponto flutuante:

$$a = 1.650 * 10^{-2}$$
$$b = 1.051 * 10^{1}$$

Agora, vamos implementar a adição de forma simplificada, sempre usando alinhamento pelo maior expoente:

$$\begin{array}{rl} a = 0.001\underline{650} & *10^{1} \\ +b = 1.051 \quad\; & *10^{1} \\ \hline a+b = 1.052\underline{650} & *10^{1} \\ a+b \cong 1.053 \quad\; & *10^{1} \end{array}$$

Observe que o menor número a, quando alinhado pelo maior expoente (de b), gera três dígitos fora da faixa de abrangência de armazenamento, $\underline{650}$, que normalmente são armazenados como dígitos de guarda durante a operação aritmética, mas são arredondados na representação final.

Exemplo 1.42: efetue a adição de $a = (10.01)_2$ e $b = (0.0101)_2$ em $F(2,4,-15,+15)$.

Solução:

Representação em ponto flutuante:

$$a = (1.001)_2 * 2^{1}$$
$$b = (1.010)_2 * 2^{-2}$$

Vamos novamente implementar a adição de forma simplificada:

$$\begin{array}{r} a = (1.001)_2 \quad * 2^1 \\ +b = (0.001010)_2 * 2^1 \\ \hline a + b = (1.010\underline{010})_2 * 2^1 \\ a + b \cong (1.010)_2 \quad * 2^1 \end{array}$$

Observe que o menor número *b*, quando alinhado pelo maior expoente (de *a*), gera 3 dígitos, <u>010</u>, fora da faixa de abrangência de armazenamento que acabam sendo arredondados.

Esses fatos ocorrem, geralmente, na soma de números com potências muito diferentes. Nesses casos, o número de menor expoente pode perder dígitos significativos, total ou parcialmente, frente ao número de maior expoente. Ou seja, devido à faixa limitada de abrangência dos registradores em ponto flutuante, o número menor perde dígitos significativos quando comparado com o número maior.

Exemplo 1.43: efetue a subtração $a - b$ com $a = 1.351$ e $b = 1.369$ em F (10, 4, –10, +10).

Solução:

Efetuando a subtração de forma simplificada, temos:

$$\begin{array}{r} a = +1.351 * 10^0 \\ -b = -1.369 * 10^0 \\ \hline a - b = -0.018 * 10^0 \\ a - b \cong -1.800 * 10^{-1} \end{array}$$

Note que o resultado final não sofreu arredondamentos, mas também perdeu dígitos significativos, pois as parcelas *a* e *b* têm quatro dígitos significativos, e a subtração $a - b$ tem apenas dois. Isso ocorre quando as parcelas subtraídas têm valores próximos.

Exemplo 1.44: reescreva as expressões das soluções de equações de segundo grau de forma a minimizar perdas de dígitos significativos.

Solução:

Para $a*x^2+b*x+c=0$, temos que as suas raízes na forma convencional são:

$$x_{1,2}=\frac{-b\pm\sqrt{b^2-4*a*c}}{2*a}$$

Ou alternativamente, por racionalização do numerador, obtemos:

$$x_{1,2}=\frac{-2*c}{b\mp\sqrt{b^2-4*a*c}}$$

Observe que as duas formas podem apresentar perdas de significação diferentes quando as parcelas b e $\sqrt{b^2-4*a*c}$ forem de magnitudes próximas e estiverem sujeitas a uma operação de subtração.

Assim, recomendamos que sejam utilizadas, em conjunto, as duas expressões propostas no **Exemplo 1.44**, escolhendo o sinal do radicando de acordo com o sinal de b ($sign(b)$), de modo que as parcelas b e $\sqrt{b^2-4*a*c}$ fiquem sujeitas sempre à operação de adição, nas duas fórmulas, usando uma para obter x_1, e outra para obter x_2, ou seja:

$$x_1=\frac{-b-sign(b)\sqrt{b^2-4*a*c}}{2*a}$$ (Fórmula tradicional)

$$x_2=\frac{-2*c}{b+sign(b)\sqrt{b^2-4*a*c}}$$ (Fórmula racionalizada)

Exemplo 1.45: considere os seguintes sistemas lineares:

a) $\begin{cases}x_1+3x_2=4\\x_1+3.00001x_2=4.00001\end{cases}$ solução exata $Sa=\{1,\ 1\}$

b) $\begin{cases}x_1+3x_2=4\\x_1+2.99999x_2=4.00002\end{cases}$ solução exata $Sb=\{10,-2\}$

Observe que o sistema de equações (b) pode ser derivado do sistema (a), com uma pequena perturbação em dois de seus coeficientes, devido hipoteticamente a arredondamentos, e que provoque uma variação ao coeficiente a_{22} de 3.00001 para 2.99999 e ao b_2 de 4.00001 para 4.00002, da ordem de 0.025%. Note, ainda, que essa pequena variação acarreta uma variação enorme na solução do sistema (b) da ordem de até 900%.

Denominamos esses sistemas altamente sensíveis a variações nos seus coeficientes de **sistemas mal condicionados**, cuja abordagem trataremos no Capítulo 2.

Exemplo 1.46: efetue a divisão a / b, com $a = 1332$ e $b = 0.9876$, em F (10, 10, –99, +99).
Solução:

$$a / b = 1348.724179829890643...$$

Esse resultado está sujeito ao armazenamento em apenas 10 dígitos significativos e precisará ser arredondado:

$$a / b = 1348.724180$$

Dessa forma, o armazenamento limitado causará uma perda de dígitos significativos e gerará um resultado aproximado.

Nas operações de divisão entre duas parcelas quaisquer, normalmente são gerados resultados com um número de dígitos maior do que o limite da representação em ponto flutuante.

1.9.2.3 Consequências dos Erros de Arredondamento

Vamos conhecer agora as principais consequências dos erros de arredondamento.

1.9.2.3.1 Propagação de Erros Devido à Perda de Significação

A perda de dígitos significativos nas operações aritméticas é a principal consequência dos erros de arredondamento, conforme exemplos já vistos anteriormente.

A seguir, apresentaremos um exemplo de propagação de erros devido a perdas de significação sequenciais.

Exemplo 1.47: para a função $f(x) = 37500/(25 - x^2)$, obtenha os valores de $f(x)$ em $x = 4.999$ e em $x = 4.9990005$ em uma calculadora científica com representação de 10 dígitos.

Solução:

Calculando parcialmente os termos de $f(x)$, temos para:

$$x = 4.999 \quad \rightarrow x^2 = 24.99000100 \rightarrow VE = 24.99000100$$

$$x = 4.9990005 \rightarrow x^2 = 24.99000600 \rightarrow VE = 24.99000599900025$$

em que VE é o valor exato.

Observe que, para $x = 4.9990005$, o valor parcial de x^2 sofre perda de dígitos significativos, ficando com apenas 8 dígitos significativos, enquanto o valor exato tem 16 dígitos exatos.

Para:

$$x = 4.999 \quad \rightarrow (25 - x^2) = 9.999 \quad *10^{-3} \rightarrow VE = 9.999 * 10^{-3}$$

$$x = 4.9990005 \rightarrow (25 - x^2) = 9.994001 * 10^{-3} \rightarrow VE = 9.99400099975 * 10^{-03}$$

Observe agora que, para $x = 4.9990005$, o valor parcial de $(25 - x^2)$ também sofre perda de dígitos significativos, ficando com apenas 7 dígitos significativos devido ao arredondamento, enquanto o seu valor exato tem 16 dígitos.

No cálculo de $f(x)$ em uma única instrução, utilizando os dígitos internos de guarda para ampliar a representação (para isso use diretamente os resultados gerados na tela, não redigite-os), temos:

$x = 4.999 \rightarrow f(4.999) = 3750375.038 \rightarrow VE = 3750375.0375$

$x = 4.9990005 \rightarrow f(4.9990005) = 3752250.975 \rightarrow VE = 3752250.975455187$

Por fim, observe que, para x = 4.9990005, o valor total de $f(x)$ também sofre perda de mais dígitos significativos, ficando com apenas 10 dígitos (limite da calculadora), enquanto o valor exato tem mais de 16 dígitos exatos.

Uma variação de 0.00001% na variável independente x, que poderia vir de um erro inerente do dado de entrada, propaga uma variação na ordem de 0.05% no resultado final de $f(x)$, ou seja, gera uma alteração (erro) no resultado quase 5.000 vezes maior do que a alteração do dado de entrada. Isso caracteriza uma instabilidade intrínseca a esse modelo matemático que deve ser evitada, se o modelo tiver outra forma possível, para minimizar a propagação de erros que sempre vão existir nos dados de entrada.

1.9.2.3.2 Instabilidade Numérica nos Algoritmos

A acumulação **sucessiva** de erros de arredondamento, em um algoritmo de repetição, pode conduzir a resultados absurdos, por exemplo, a função $f(x) = (n+1)*x - 1$ é uma constante para $x = 1/n$ (na ausência de arredondamentos), pois $f(1/n) = (n+1)*1/n - 1 = 1 + 1/n - 1 = 1/n$, mas gera valores absurdos a partir de pequenos arredondamentos iniciais. Verifique essa afirmação sobre a desestabilização de resultados implementando o exercício 1.11 em computador.

Na próxima seção, apresentaremos o principal erro dos métodos de aproximação numérica, que são decorrentes de truncamentos de cada tipo de aproximação.

1.9.3 Erros de Truncamento

Os erros de truncamento são erros intrínsecos da **aproximação** dos métodos numéricos. Ocorrem sempre que sustamos uma sequência de cálculo infinita, tornando-a finita, por conveniência ou por incapacidade de execução. Assim, ocorrem erros de truncamento, além dos erros de arredondamento, em praticamente todos os métodos e algoritmos do cálculo numérico.

Exemplo 1.48: seja a seguinte expansão em série infinita de Taylor/MacLaurin para a função $\exp(x) = e^x$:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$$

Sabemos que não é possível usar os infinitos termos dessa série para obter um valor de e^x, então precisamos estabelecer um limite para o número de termos utilizados. Essa limitação no número de termos gera um erro de truncamento na série aproximada que corresponde ao somatório dos termos abandonados.
Podemos representar de forma exata uma função $f(x_0 + x)$ através de expansão em série de Taylor convergente, dada genericamente por:

$$f(x_0 + x) = f(x_0) + f'(x_0)\frac{x}{1!} + f''(x_0)\frac{x^2}{2!} + f'''(x_0)\frac{x^3}{3!} + \ldots + f^{(n)}(x_0)\frac{x^n}{n!} + \ldots$$

Expandindo a função e^x em torno de $x_0 = 0$, temos:

$$f(x_0 = 0) = e^0 = 1$$
$$f'(x_0 = 0) = e^0 = 1$$
$$f''(x_0 = 0) = e^0 = 1$$

$$\vdots$$

$$f^{(n)}(x_0 = 0) = e^0 = 1$$

Logo, $f(0+x) = e^{0+x} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + O(x^{(n+1)})$

Se aproximamos e^x com apenas n termos, estamos desprezando uma série infinita de termos a partir do termo n+1,

$$O(x^{(n+1)}) = \frac{x^{n+1}}{(n+1)!} + \frac{x^{n+2}}{(n+2)!} + \ldots$$

Logo, $e^x \cong 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$

Assim, o termo $O(x^{(n+1)})$ caracteriza o **erro de truncamento** da aproximação nesse caso, ou seja, o erro da aproximação em relação ao valor exato.

Lembre-se de que x representa o incremento de x entre o ponto inicial $x_0 = 0$ e o ponto final $0 + x$.

Observe na Tabela 1.1, a convergência de e^1 pelo valor aproximado $VA^{(n)} \cong 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$, em função de n, comparado com o valor exato e = 2.718281828459045...

Tabela 1.1 – Convergência de e[1]

n	$VA^{(n)}$	$VA^{(n)}-VA^{(n-1)}$	$VE = e$	Erro exato = $\lvert VA^{(n)}-VE\rvert$
0	1.000000000000000	-	2.718281828459040	1.71828182845904
1	2.000000000000000	1.00000000000000	2.718281828459040	0.71828182845904
2	2.500000000000000	0.50000000000000	2.718281828459040	0.21828182845904
3	2.666666666666670	0.16666666666667	2.718281828459040	0.05161516179237
4	2.708333333333330	0.04166666666667	2.718281828459040	0.00994849512571
5	2.716666666666670	0.00833333333333	2.718281828459040	0.00161516179237
6	2.718055555555560	0.00138888888889	2.718281828459040	0.00022627290348
7	2.718253968253970	0.00019841269841	2.718281828459040	**0.00002786020507**
8	2.718278769841270	**0.00002480158730**	2.718281828459040	0.00000305861777
9	2.718281525573190	0.00000275573192	2.718281828459040	0.00000030288585
10	2.718281801146380	0.00000027557319	2.718281828459040	0.00000002731266
11	2.718281826198490	0.00000002505211	2.718281828459040	0.00000000226055
12	2.718281828286170	0.00000000208768	2.718281828459040	0.00000000017287
13	2.718281828446760	0.00000000016059	2.718281828459040	0.00000000001228
14	2.718281828458230	0.00000000001147	2.718281828459040	0.00000000000081
15	2.718281828458990	0.00000000000076	2.718281828459040	0.00000000000005
16	2.718281828459040	0.00000000000005	2.718281828459040	0.00000000000000
17	2.718281828459050	0.00000000000000	2.718281828459040	0.00000000000001
18	2.718281828459050	0.00000000000000	2.718281828459040	0.00000000000001

Nesse exemplo, podemos obter o resultado aproximado com 5 dígitos significativos totais da seguinte forma:

a) Preliminarmente, podemos estimar o número de dígitos confiáveis como aqueles que permanecem constantes entre uma aproximação com n–1 e outra com n parcelas, que pode ser quantificado pelas diferenças em módulo, chamado "critério de parada", dado por $\left|VA^{(n)} - VA^{(n-1)}\right|$. Assim, se quisermos um resultado com 5 dígitos (4 dígitos depois do ponto (vírgula)), poderemos interromper a sequência com n=8 termos, conforme a planilha desse exemplo, cujo critério de parada fica inferior a $1*10^{-4}$, no caso $0.2480158730*10^{-4}$ (destacado em negrito).

b) Idealmente, deveríamos avaliar o número de dígitos exatos calculando o erro exato de cada aproximação, conforme

$Erro\ exato = \left| VA^{(n)} - VE \right|$ apresentado na última coluna da planilha desse exemplo, com *VE* disponível. Assim, obtemos o mesmo resultado, com 4 dígitos exatos depois do ponto, interrompendo a sequência com $n = 7$ termos, cujo Erro Exato é inferior a $1*10^{-4}$, no caso $0.2786020507*10^{-4}$ (destacado em negrito), mas raramente o valor exato *VE* está disponível.

c) O resultado que obtivemos com o critério de parada em (a) é mais conservativo do que o resultado obtido com o erro exato em (b), ambos com limite $1*10^{-4}$, pois em (a) temos resultado mais exato do que em (b). Normalmente, podemos usar os critérios de parada em um processo repetitivo para obtenção de resultados por aproximações numéricas como referência para o erro, mas esse tema será abordado posteriormente para cada método numérico apresentado ao longo deste livro.

Exemplo 1.49: aproximações numéricas de derivadas de funções.
Por definição, $f'(x)$ é dada por, $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$

Se não pudermos obter esse limite exato, podemos efetuar uma aproximação numérica tomando o incremento *h* como finito e promovendo refinamentos de resultados em razão do *h* adotado. Assim, podemos obter uma sequência de aproximações sucessivas de $f'(x)$, com exatidão crescente, usando incrementos *h* cada vez menores, mas não podemos chegar a um incremento *h* numericamente nulo ($h \cong O(10^{-16})$ para variáveis do tipo *double*), pois nesse caso, o numerador do limite se torna nulo por perda de significação. Então, também precisamos quebrar o processo matemático de refinamentos sucessivos, assumindo um erro de truncamento de limite tolerável.

Exemplo 1.50: aproximações numéricas para derivada de 2ª ordem, a partir de três pontos vizinhos de $f(x)$, com espaçamento *h*:

$f(x_0 - h)$	$f(x_0)$	$f(x_0 + h)$
$x_0 - h$	x_0	$x_0 + h$

Podemos somar a série de Taylor para $f(x_0+h)$ com a de $f(x_0-h)$:

$$f(x_0+h)=f(x_0)+f'(x_0)\frac{h}{1!}+f''(x_0)\frac{h^2}{2!}+f'''(x_0)\frac{h^3}{3!}+\ldots+f^{(n)}(x_0)\frac{h^n}{n!}+\ldots$$

+

$$f(x_0-h)=f(x_0)-f'(x_0)\frac{h}{1!}+f''(x_0)\frac{h^2}{2!}-f'''(x_0)\frac{h^3}{3!}-\ldots+f^{(n)}(x_0)\frac{h^n}{n!}+\ldots$$

Resultando em

$$f(x_0+h)+f(x_0-h)=2f(x_0)+2f''(x_0)\frac{h^2}{2!}+2f^{(4)}(x_0)\frac{h^4}{4!}+\ldots$$

Isolando a segunda derivada, temos:

$$f''(x_0)=\left(f(x_0+h)+f(x_0-h)-2f(x_0)\right)/h^2+f^{(4)}(x_0)\frac{h^2}{4!}+\ldots$$

$$f''(x_0)=\left(f(x_0+h)+f(x_0-h)-2f(x_0)\right)/h^2+O(h^2)$$

O termo de segunda ordem $O(h^2)$ representa o somatório dos infinitos termos truncados decorrentes da aproximação em série e podemos defini-lo como erro de truncamento da aproximação,

$$O(h^2)=f^{(4)}(x_0)\frac{h^2}{4!}+f^{(6)}(x_0)\frac{h^4}{6!}+\ldots$$

O erro de truncamento da aproximação $O(h^2)$ pode ser reduzido com h. Então, desprezando esse termo $O(h^2)$, assumimos um erro de truncamento de 2ª ordem e temos uma aproximação para $f''(x_0)$ dada por

$$f''(x_0)\cong\left(f(x_0+h)+f(x_0-h)-2f(x_0)\right)/h^2$$

1.9.4 Medida de Erros

Podemos fazer a avaliação de erros numéricos se tivermos a disponibilidade do **valor exato** *VE* para o resultado desejado. Dessa forma, podemos calcular qualquer erro numérico por meio das formas

apresentadas anteriormente usando o valor aproximado obtido *VA* e o valor exato *VE*. A representação dos erros numéricos em forma de erro relativo (ou percentual) é mais representativa e usual.

Também podemos obter uma **estimativa** dos erros numéricos, em resultados de algoritmos numéricos que não têm solução exata conhecida, utilizando o próprio algoritmo numérico a fim de obter uma **estimativa de valores exatos**, ou mais próximos do exato, mas precisaremos usar mais recursos computacionais, como precisão dupla, para minizar os arredondamentos; e mais termos nas séries, mais repetições de cálculos, entre outros, para minimizar os erros de truncamento.

No **Exemplo 1.51**, apresentaremos a estimativa de erro de arredondamento de um valor *VA* obtido por algoritmo executado com certa precisão limitada. Nesse exemplo, podemos **estimar** um valor mais exato, que denotaremos por VE^e como sendo aquele obtido com o próprio algoritmo numérico operando com **mais precisão**, como **precisão duplicada**, ou seja, reavaliamos o valor obtido numericamente, no mesmo sistema de numeração (normalmente binário), mas agora com precisão superior, e esses novos resultados serão mais exatos.

Exemplo 1.51: estime o erro de arredondamento gerado no armazenamento de $x = (0.1)_{10}$ em uma variável binária de 32 *bits* usando recursos da própria base binária.

Solução:

Se simularmos esse cálculo em linguagem C/C++, temos:

$x = (0.1)_{10}$ armazenado em 32 *bits* (*float*) gera o decimal:

$$x = 0.100000001490116$$

e

$xx = (0.1)_{10}$ armazenado em 64 *bits* (*double*) gera o decimal:

$$xx = 0.10000000000000000055511151$$

que também é um valor aproximado, mas tem o dobro de dígitos exatos em relação a *x*.

Logo, podemos obter o erro de arredondamento estimado de *x* pela diferença entre $VA = x$ e $VE^e = xx$ (estimativa do valor exato):

$$Erro\ exato\ estimado = |VA - VE^e| =$$
$$= |0.100000001490116 - 0.10000000000000001| = 0.1490116 * 10^{-8}.$$

Observe que esse erro é calculado na própria base binária, portanto usa valores decimais aproximados e mesmo assim gera um erro estimado compatível com o próprio erro exato.

No **Exemplo 1.52**, apresentaremos a estimativa de erro de truncamento de um valor VA obtido numericamente. Podemos estimar um valor mais exato VE^e como sendo aquele obtido com o próprio algoritmo numérico operando com mais exatidão nas aproximações e mais termos na série aproximadora. Por segurança, devemos estimar o valor mais exato VE^e, em precisão duplicada, visando minimizar os efeitos de erros de arredondamentos sobre os erros de truncamento.

Exemplo 1.52: calcule o erro de truncamento da aproximação $VA^{(n-7)}$ que obtivemos no **Exemplo 1.48** para $n = 7$ parcelas.

Solução:

O valor aproximado $VA^{(n-7)}$ obtido com $n = 7$ é 2.718253968253970, e sabemos que tem 5 dígitos exatos totais (4 decimais depois do ponto).

Podemos simular um valor exato VE^e de referência operando com n maior. Idealmente obtemos o valor exato com $n \to \infty$, o que normalmente é inviável, então precisamos estimar um número finito mínimo de parcelas, como $2n = 14$ parcelas, gerando o seguinte resultado: 2.718281828458230.

Observe que esse valor VE^e também é aproximado, no caso com $2n = 14$ parcelas, mas contém bem mais dígitos exatos que o $VA^{(n-7)}$ inicial. Esse VE^e tem 12 dígitos exatos totais.

Então,

$VA^{(n=7)} = 2.718253968253970$ (com $n = 7$)

$VE^e = 2.718281828458230$ (VE^e estimado com $2n = 14$)
$VE = 2.718281828459040$ (valor exato com infinitas parcelas, limitado a 16 dígitos significativos)
Logo,

$$Erro\ Exato = \left|VA^{(n)} - VE\right| = 0.2786020507 * 10^{-04}$$
$$Erro\ Exato\ estimado = \left|VA^{(n)} - VE^e\right| = 0.2786020426 * 10^{-04}$$

Assim, podemos estimar o erro exato usando simulações do valor exato VE^e, mas precisamos otimizar os recursos computacionais necessários para essa simulação. Observe que, se o valor exato estimado VE^e for calculado por VA com menos parcelas (mas $n > 7$), também teremos resultados razoáveis:

$$Erro\ exato\ estimado = \left|VA^{(n=7)} - VA^{(n=14)}\right| = 0.2786020426 * 10^{-04}$$
$$Erro\ exato\ estimado = \left|VA^{(n=7)} - VA^{(n=13)}\right| = 0.2786019279 * 10^{-04}$$
$$Erro\ exato\ estimado = \left|VA^{(n=7)} - VA^{(n=12)}\right| = 0.2786003220 * 10^{-04}$$
$$Erro\ exato\ estimado = \left|VA^{(n=7)} - VA^{(n=11)}\right| = 0.2785794452 * 10^{-04}$$
$$Erro\ exato\ estimado = \left|VA^{(n=7)} - VA^{(n=10)}\right| = 0.2783289242 * 10^{-04}$$
$$Erro\ exato\ estimado = \left|VA^{(n=7)} - VA^{(n=9)}\right| = 0.2755731922 * 10^{-04}$$
$$Erro\ exato\ estimado = \left|VA^{(n=7)} - VA^{(n=8)}\right| = 0.2480158730 * 10^{-04}$$

Note que todas essas simulações do erro de truncamento de $VA^{(n=7)}$ geraram resultados com a mesma ordem de grandeza, portanto, para essa aproximação em série de MacLaurin, foi suficiente simular o valor exato, VE^e, com apenas $n + 1$ parcelas. Assim, o erro estimado $\left|VA^{(n=7)} - VA^{(n=8)}\right| = 0.2480158730 * 10^{-04}$ foi da mesma ordem do erro exato.

Os erros estimados anteriormente foram "todos" inferiores ao critério de parada que utilizamos no **Exemplo 1.48**, para $n = 7$ parcelas, $\left|VA^{(n)} - VA^{(n-1)}\right| = 1.9841269841 * 10^{-4}$. Logo, podemos usar o critério de parada como limite superior do erro de truncamento estimado, nesse tipo de aproximação, considerando que, se o critério de parada já é aceito como suficientemente pequeno, o erro exato será ainda menor.

Um cálculo com mais exatidão, para servir de estimativa do valor exato, naturalmente exige mais recursos computacionais, mas deve ser aplicado para validação dos resultados obtidos por algum critério de parada, conforme veremos detalhadamente para cada tipo de aproximação numérica aplicada ao longo deste livro.

Por fim, existem formas alternativas de estimar valores exatos do resultado desejado que minimizam os erros numéricos de arredondamento e/ou truncamento além das presentes no escopo deste livro. São elas:

a) **simulações numéricas utilizando matemática intervalar**: possibilitam limitar o erro existente a um intervalo aritmético;

b) **variável de comportamento estatístico**: possibilita prever os limites do erro segundo um tratamento estatístico das variáveis envolvidas;

c) **simulação do algoritmo em Sistemas de Computação Algébrica (SCA)**: possibilita recorrer a simulações com precisão ilimitada e, em alguns casos especiais, proceder à simulação exata do algoritmo;

d) **aritmética discreta de corpos finitos**: possibilita que todos os elementos de um corpo tenham uma representação na memória do computador, portanto não há erros de arredondamento, sendo possível utilizá-la especificamente para problemas discretos, como aqueles encontrados em criptografia, códigos de correção de erros, filtros digitais etc.; e

e) **representação de números fracionários**: baseada no padrão IEEE chamado de ***unum (universal number)***, possibilita uma série de vantagens na sua representação para evitar os arredondamentos, adicionando aos campos *bit* de sinal, expoente e mantissa, já conhecidos, mais três campos (*utag*): um *bit* que marca se o número é exato ou se está entre valores exatos (*ubit*), um campo para o tamanho do expoente e outro para o tamanho da fração, entretanto ainda não está difundida nas linguagens de programação comerciais (GUSTAFSON, 2015).

Saiba mais sobre essa inovadora proposta no livro "The End of Error: unum computing", de John Gustafson, conhecido pela formulação da Lei de Gustafson.

1.10 Conclusões

O perfeito entendimento das causas dos erros numéricos, gerados em calculadoras e computadores, é um fator decisivo para que você, como analista numérico, possa fazer a escolha do método computacionalmente mais eficiente e a validação dos resultados. Assim, para resolver um determinado modelo matemático, o analista numérico deve se preocupar com:

a) a escolha de métodos de resolução com menor número de operações aritméticas envolvidas; e

b) a escolha de uma linguagem de programação para implementar o algoritmo que represente as variáveis envolvidas com precisão suficientemente grande.

No final de todo o processo, o analista numérico deve ser capaz de:

a) obter uma solução de custo mínimo, ou seja, com menor demanda de memória utilizada e menor tempo de processamento; e

b) dimensionar o grau de confiabilidade dos resultados obtidos como solução do modelo matemático, ou seja, apresentar o resultado obtido e o seu erro máximo.

Lembre-se que obter resultados de algoritmos numéricos é relativamente fácil, pois qualquer algoritmo compilado gera resultados numéricos. Para assegurar que esses resultados são as respostas esperadas, é necessária a **estimativa dos erros numéricos** associados à solução aproximada do modelo matemático.

A estimativa dos erros numéricos de truncamento associada à solução aproximada do modelo matemático será discutida ao longo de cada capítulo deste livro.

Complementando...

Nesta seção, detalharemos os conceitos de alguns termos utilizados no Capítulo 1.

Arredondamento Decimal

O objetivo de um arredondamento é representar o número com menor número de dígitos significativos totais e ainda minimizar os erros decorrentes dessa redução de significativos, seja qual for a base. Vamos detalhar esse conceito através do exemplo a seguir.

Exemplo 1.53: represente os números decimais, a seguir, com 4 dígitos significativos totais:

a) $32.4374 = 32.43 + 0.74 * 10^{-2}$

Solução:

Aqui precisamos definir o que fazer com a parcela $0.74 * 10^{-2}$, que corresponde aos 2 dígitos menos significativos dessa representação. Podemos aproximá-la para $1.00 * 10^{-2}$ ou para $0.00 * 10^{-2}$, e o critério de escolha deve ser tal que o resultado vai gerar o menor erro no número final arredondado, ou seja,

i) se $0.74 * 10^{-2} \cong 1. * 10^{-2}$, o erro é $0.26 * 10^{-2}$; e

ii) se $0.74 * 10^{-2} \cong 0. * 10^{-2}$, o erro é maior, $0.74 * 10^{-2}$.

Logo, $32.4374 = 32.43 + 0.74 * 10^{-2} \cong 32.43 + 1. * 10^{-2} \cong 32.44$

b) $32.4324 = 32.43 + 0.24 * 10^{-2}$

Solução:

Aqui também precisamos definir o que fazer com a parcela $0.24 * 10^{-2}$, e aproximar $0.24 * 10^{-2} \cong 0. * 10^{-2}$ gerará menor erro.

Logo, $32.4324 = 32.43 + 0.24 * 10^{-2} \cong 32.43 + 0. * 10^{-2} \cong 32.43$

c) $32.4350 = 32.43 + 0.50 * 10^{-2}$

Solução:

Podemos aproximar a parcela $0.50 * 10^{-2}$ para $1.00 * 10^{-2}$ ou para $0.00 * 10^{-2}$, mas nesse caso o erro gerado é o mesmo. Então, recorremos a um critério de distribuição dos erros para esse grupo de números que tenham fração 0.50, de modo que alguns números ganhem um pouco (aproximando para $1.00 * 10^{-2}$) e outros percam um pouco (aproximando para $0.00 * 10^{-2}$). Para isso, fazemos uma distribuição estatística, de modo que 50% dos números ganhem um pouco e outros 50% percam um pouco. Isso foi padronizado com critério baseado no dígito anterior à parcela a ser arredondada, que pode ser um número par ou um ímpar, com 50% de chances em cada caso. Assim, para os números com dígito anterior PAR, convencionou-se arredondar para $0.00 * 10^{-2}$, e para números com dígito anterior ÍMPAR, convencionou-se arredondar para $1.00 * 10^{-2}$. Nesse exemplo, arredondamos a parcela $0.50 * 10^{-2}$ para $1.00 * 10^{-2}$ (pois o dígito anterior 3 é IMPAR).

Logo, $32.4350 = 32.43 + 0.50 * 10^{-2} \cong 32.43 + 1. * 10^{-2} \cong 32.44$

d) $32.4450 = 32.44 + 0.50 * 10^{-2}$

Solução:

Nesse exemplo, arredondamos a parcela $0.50 * 10^{-2}$ para $0.00 * 10^{-2}$ (pois o dígito anterior 4 é PAR).

Logo, $32.4450 = 32.44 + 0.50 * 10^{-2} \cong 32.44 + 0. * 10^{-2} \cong 32.44$

Precisão

Precisão é um conceito objetivo que estabelece a quantidade de algarismos significativos que representam um número. A precisão de uma representação digital, de uma variável, por exemplo, é definida como o número de dígitos significativos totais da mantissa na base β de armazenamento; e precisão decimal equivalente d baseada nos d significativos decimais equivalentes. Assim, para variáveis de 32 *bits* padrão IEEE 754, com $v = (-1)^s (1.f)_2 \, 2^{e-127}$, temos $t = 23$ *bits* depois do ponto (vírgula), uma precisão binária total de 24 *bits* $(1 + t)$ e precisão decimal equivalente de 7 a 8 significativos.

A precisão decimal d equivalente pode ser definida pela equivalência direta da faixa de abrangência entre os dígitos mais e menos significativos em cada base; pela equivalência direta entre os expoentes dos dígitos menos significativos da mantissa em cada base; e pela verificação direta do número de dígitos decimais exatos que podem ser armazenados.

Faixa de abrangência entre os dígitos mais e menos significativos em cada base

Para variáveis de 32 *bits* padrão IEEE 754, temos um total de 24 *bits* de precisão binária, pois existe um dígito implícito antes do ponto (vírgula), $d_1 = 1$.

A mantissa total de dígitos significativos é dada por: $v = (d_1 . d_2 d_3 \ldots d_{22} d_{23} d_{24})_2$, logo:

a) o dígito binário mais significativo está antes do ponto (vírgula), $d_1 = 1$, que não é armazenado nos 32 *bits*, e equivale a: $Dmsb = (1.00000000000000000000000)_2 = 1 * 2^0 = 1.0$;

b) o dígito binário menos significativo está no **23º registro armazenado**, d_{24}, pois temos $t = 23$ *bits* depois do ponto (vírgula) e equivale a:

$$Dlsb = (0.00000000000000000000001)_2 = 1 * 2^{-23} = 1.1920928955 * 10^{-7}$$

23º registro armazenado

O expoente do dígito binário menos significativo é considerado a representação da precisão decimal equivalente da variável no padrão IEEE. No Octave é armazenado na função *eps*.

Se somarmos esses dois valores, teremos a faixa de abrangência representada pelos dígitos em binário ou em decimal:

$$
\begin{array}{rll}
Dmsb & = 1 * 2^{0} & = 1.0 * \qquad\qquad 10^{0} \qquad\qquad \text{– 1º dígito decimal, antes do ponto.} \\
+\ Dlsb & = 1 * 2^{-23} & = 1.1920928955 * 10^{-7} \qquad \text{– 7º dígito decimal, depois do ponto.} \\
 & & = (\underline{1.00000000000000000000001})_{2} \quad \text{– 24 dígitos binários totais (sublinhados).} \\
 & & = (\underline{1.0000001}\ 190928955)_{10} \quad \text{– 8 dígitos decimais totais (sublinhados).}
\end{array}
$$

Podemos concluir que a precisão decimal é de até 8 dígitos decimais equivalentes (limite máximo).

Para representações fora do padrão IEEE 754, como a antiga variável de 16 *bits*, temos de considerar que o primeiro dígito binário também é fixo, $d_1 \neq 0 = 1$, mas posicionado à direita do ponto (vírgula) e ocupa um registro binário, que tem $t = 10$ *bits* significativos totais.

Nesse caso, o binário mais significativo seria:

$Dmsb = (0.1000000000)^2 = 1 * 2^{-1} = 0.5,$

e o binário menos significativo seria:

$Dmsb = (0.0000000001)_2 = 1 * 2^{-10} = 0.009765625 = 9.765625e-4,$

cuja faixa de abrangência dos valores representados pelos dígitos, em binário ou em decimal, é

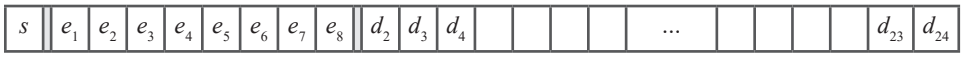
$= (0.\underline{1000000001})_2$ - 10 dígitos binários totais (sublinhados); e

$= (0.\underline{5009}765625)_{10}$ - 04 dígitos decimais totais (sublinhados).

Assim, podemos concluir que a precisão decimal é de até 4 dígitos decimais equivalentes.

Equivalência direta entre os expoentes dos dígitos menos significativos

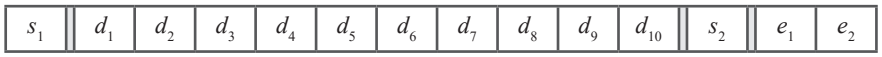
Na representação de 32 *bits* padrão IEEE 754:



$t = 23$ dígitos binários significativos, da mantissa f

a sua menor representação binária possível é 2^{-t} ($t = 23$ é o número de *bits* depois do ponto (vírgula) e representa o *bit* menos significativo).

Na representação de uma calculadora científica decimal comum (ex. para $d = 10$ significativos):



$t = 9$ dígitos significativos da mantissa f, após o ponto

O decimal equivalente a d_{10} é $10^{-(d-1)}$ ($d - 1$ é o número de dígitos decimais depois do ponto (vírgula) e representa o dígito menos significativo da calculadora), então existe uma equivalência entre a representação binária de $t + 1$ *bits* totais e a decimal de d decimais:

$$2^{-((t+1)-1)} = 10^{-(d-1)}$$
$$2^{-t} = 10^{1-d}$$
$$d = 1 + t * \log(2)$$
$$d = 7.92368990 \text{ (para } t = 23 \textit{ bits}\text{, depois do ponto (vírgula))}$$

Portanto, temos uma precisão decimal equivalente entre $d = 7$ e 8 decimais significativos, ou seja, uma calculadora científica de 8 decimais seria suficiente para armazenar os $1 + t = 24$ *bits* de forma exata.

Verificação direta do número de dígitos decimais exatos que podem ser armazenados

Trata-se de um teste direto (experimentação numérica), no qual podemos armazenar um decimal exato **conhecido** na variável que se deseja testar a precisão decimal. Assim, se tomarmos os decimais conhecidos, a seguir, e armazenarmos em uma variavel de 32 *bits* padrão IEEE 754, usando um tipo *float* no C, Java, entre outros, teremos:

a) $(1/3) \cong 0.\underline{333333}43267441$ – temos 7 decimais exatos

$0.333333333333333...$ – valor exato.

b) $(12345678901234567890123 45) \cong \underline{1.23456}794679859e{+}24$ – temos 7 decimais exatos.

$1.2345679012345678901234 5e{+}24$ – valor exato.

c) $(0.100000000)_{10} \cong (0.\underline{10000000}1490116)_{10}$ – temos 8 decimais exatos

$0.10000000000000000000000000$ – valor exato

Assim, essa variável padrão IEEE 754 de 32 *bits* totais tem entre 7 e 8 dígitos decimais equivalentes. Isso significa que todos os números armazenados neste padrão têm pelo menos 7 dígitos decimais significativos "exatos", e alguns têm até 8 dígitos "exatos".

Também usamos o termo precisão para qualificar o número de dígitos significativos convergidos em resultados aproximados por métodos iterativos, quando a solução exata está indisponível.

Exatidão

Exatidão é um conceito subjetivo e diz respeito à forma que melhor representa uma grandeza numérica, ou seja, uma representação

é mais exata quando tem o menor erro em relação ao seu valor exato. Então, usamos o termo **exatidão** para qualificar resultados aproximados quando a solução exata está disponível.

Por exemplo, se o valor de π exato $(3.1415926535...)$ é representado por:

a) 3.14 → precisão de 3 dígitos decimais (*erro* = 0.0015926535), π exato arredondado com 3 dígitos é o mesmo 3.14, então a exatidão também é de 3 dígitos decimais;

b) 3.<u>151</u> → precisão de 4 dígitos decimais (*erro* = 0.0094073465), π exato arredondado com 4 dígitos é 3.142, então 3.1<u>51</u> tem apenas 2 dígitos decimais exatos 3.1;

c) 3.14<u>15</u> → precisão de 5 dígitos decimais (erro = 0.0000073465), π exato arredondado com 5 dígitos é 3.1416, então 3.1415 tem apenas 4 dígitos decimais exatos 3.141.

Note que, se classificarmos esses números quanto à sua exatidão, teríamos que: (a) está mais próximo do π exato do que (b); (b) é menos exato do que (c); e (c) é o mais exato de todos.

Normalmente, os resultados mais **exatos** (com menores erros) são obtidos com as representações mais **precisas** (com mais dígitos significativos representados).

Você encontrará exercícios atualizados e revisados de todos os capítulos deste livro no Caderno de Exercícios e Respostas disponível no *link* <http://sergiopeters.prof.ufsc.br/exercicios-e-respostas/>. Faça o *download* do Caderno e reforce o seu aprendizado realizando os exercícios.