UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
CENTRO TECNOLÓGICO - CTC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA - INE

Calculo Numérico em Computadores:

Capítulo 1

Sistemas de Numeração e Erros numéricos

Autores: Prof. Sérgio Peters
Acad. Andréa Vergara da Silva

e-mail: sergio.peters@ufsc.br

Florianópolis, 2013.

1. SISTEMAS DE NUMERAÇÃO

1.1 - INTRODUÇÃO

Atualmente o sistema padronizado de representação de quantidades para o uso e a comunicação entre as pessoas é o sistema decimal. Entretanto, para facilitar a representação física, a definição das operações aritméticas e a comunicação entre as máquinas digitais, é necessário fazer uso de outros sistemas de representação.

Como premissa básica, conceitua-se **número** como a representação simbólica de determinada quantidade matemática e **base** de um sistema de numeração a quantidade de símbolos distintos utilizados nesta representação. Desta forma, um número real qualquer X na base β pode ser algebricamente representado através de:

$$X_\beta = (a_1\, a_2 \ldots a_k\, ,\, a_{k+1}\; a_{k+2} \ldots a_{k+n})_\beta \qquad (1)$$

onde β é a base, $a_i \in \{0,1,2,....,\beta\text{-}1\}$, i = 1,2,...,k+n , k é o comprimento da parte inteira e n da parte fracionária do número, com $k,n \in \aleph$.

Ex. 1: $X=(309,57)_{10}$

Para fins de uso algébrico X pode também ser representado na **forma fatorada** equivalente:

$$X_\beta = \sum_{i=1}^{k} a_i \beta^{k-i} + \sum_{j=1}^{n} a_{k+j} \beta^{-j} \qquad (2)$$

Ex. 2: $X=(309,57)_{10} = 3.10^2 + 0.10^1 + 9.10^0 + 5.10^{-1} + 7.10^{-2}$

A seguir, serão abordados alguns sistemas de numeração e as formas de representação de números de amplo uso nas máquinas digitais.

1.2 - SISTEMA DECIMAL ($\beta = 10$)

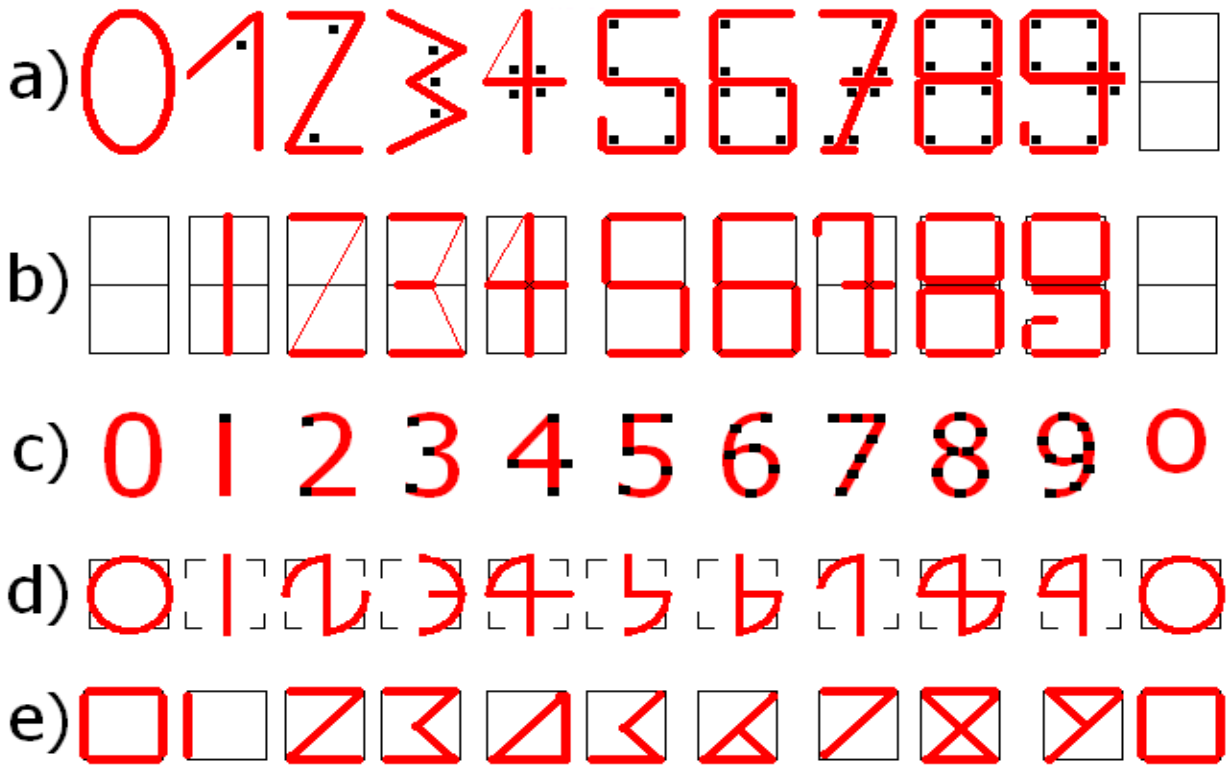
O sistema decimal de numeração, adotado pela maioria dos países, foi desenvolvido pelos astronômos hindus por volta do século V e divulgado ao mundo islâmico em 825 no livro do matemático Alkhawarismi e definitivamente adotado no ocidente no século XVI. Sua aceitação como padrão deve-se a algumas de suas características tais como:

i). Utiliza dez símbolos, dígitos (digitus = dedo em latim) ou algarismos (curruptela lógica de Alkhawarismi). Tais símbolos atualmente são representados por: 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9, conhecidos por algarismos arábicos, que são derivados da versão ainda hoje usada no mundo muçulmano:

٩, ٨, ٧, ٦, ٥, ٤, ٣, ٢, ١, ٠

Historicamente estes números tem diversas hipóteses sobre sua origem::

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) Figuras desenhadas a partir dos traços de um quadrado e suas diagonais.



ii). Faz uso do zero. O zero, aceito com muita relutância, é o indicador da ausência de certas potências da base na representação de um número na forma fatorada;

iii). Adota o princípio da posicionalidade. No sistema posicional o valor de cada símbolo é relativo, isto é, depende da sua posição no número.

Ex. 3: Nos números

a) $(574)_{10}$ = 5x102 + 7x10^1 + 4x10^0

b) $(348)_{10}$ = 3x10^2 + 4x10^1 + 8x10^0

c) $(432,5)_{10}$ = 4x10^2 + 3x10^1 + 2x10^0 + 5x10^{-1}

O símbolo 4 representa, respectivamente, quatro unidades, quatro dezenas e quatro centenas.

Note que nenhum dígito interfere na posição do outro, eles são inteiramente independentes entre si.

Utilizando-se das duas últimas características do sistema decimal, a seguir serão estabelecidos outros sistemas de numeração, para facilitar a comunicação homem-máquina.

1.3 - SISTEMA BINÁRIO (β = 2)

Fazendo uso apenas dos símbolos 0 e 1, também chamados de *bits* (abreviatura de *binary digits*); do zero e da posicionalidade, gera-se um novo sistema de numeração cuja 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	.

Utilizando-se da notação fatorada, tem-se por exemplo,

Ex. 4:

$(10011)_2 = (1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 1.2^0)_{10} = (19)_{10}$

Obs.: A forma fatorada do número binário (base 2) está representada na base 10.

- **Vantagens do Sistema Binário em Relação ao Sistema Decimal**

(i). Simplicidade de representação física, bastam 2 estados distintos de uma máquina digital para representar os dígitos da base: 0 = - , *off*
1 = + , *on*

Obs.: No futuro poderá se chegar a distinção de bits através dos atomos, que é a concepção do computador quântico (*quantum-bits*, ou *qubits*).

(ii). Simplicidade na definição de operações aritméticas fundamentais:

Ex. 5: Adição:
$+ : \Re x \Re \rightarrow \Re$
$(x,y) \rightarrow x + y$ definida por:

Em $\beta = 10$ necessita-se de 100 combinações dos possíveis valores de x e y para se definir a função adição.

Em $\beta = 2$ tem-se apenas 4 combinações:

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

- **Desvantagens do Sistema Binário**

(i). Necessidade de registros longos para armazenamento de números.

Ex. 6: $(597)_{10} = (1001010101)_2$

Observa-se que foi necessário um registro com capacidade de armazenamento de dez símbolos binários para representar a grandeza decimal $(597)_{10}$ de apenas três dígitos decimais.

1.4 - SISTEMA HEXADECIMAL ($\beta = 16$)

Símbolos representativos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F. Onde A, B, C, D, E e F representam as quantidades decimais 10, 11, 12, 13, 14 e 15, respectivamente.

Este também é um sistema posicional.

Ex. 7: $(1A0, C)_{16} = 1x16^2 + Ax16^1 + 0x16^0 + Cx16^{-1}$
$= (1x16^2 + 10x16^1 + 0x16^0 + 12x16^{-1})_{10}$
$= (256 + 160 + 0 + 12/16 = 416{,}75)_{10}$

- **Vantagem do Sistema Hexadecimal**

(i). Número reduzido de símbolos para representar grandes quantidades, por isso é um sistema de numeração interessante para visualização e armazenamento de dados. Os registros binários internos de uma máquina digital são convertidos de forma direta para Hexadecimal quando são necessárias visualizações externas, requisitadas pelo usuário.

Ex. 8: $(\underline{1101}\ \underline{0110})_2 = (D6)_{16} = (214)_{10}$
oito bits 2 símbolos hexadecimais
Obs.: Até a década de 70 as máquinas digitais se utilizavam do sistema de numeração octal, de base

8, para visualizar os registros binários internos.

1.5 - CONVERSÕES ENTRE SISTEMAS DE NUMERAÇÃO

As conversões entre bases são necessárias para que se possa melhor entender algumas das causas dos erros existentes nas representações digitais de quantidades, pois o homem utiliza o sistema decimal e os computadores as convertem para bases binária, hexadecimal, ou outra.

1.5.1 - CONVERSÃO DE BASE β PARA BASE 10

Nestes casos as conversões são obtidas escrevendo o respectivo número na sua forma fatorada, representada na base decimal.

$$(a_1a_2a_3,a_4a_5)_\beta = (a_1 . \beta^2 + a_2 . \beta^1 + a_3 . \beta^0 + a_4 . \beta^{-1} + a_5 . \beta^{-2})_{10}$$

Ex. 9: $(101,1)_2 = 1 . 2^2 + 0 . 2^1 + 1 . 2^0 + 1 . 2^{-1} = (5,5)_{10}$

$(1A,B)_{16} = 1 . 16^1 + A . 16^0 + B . 16^{-1} = (16,6875)_{10}$

1.5.2 - CONVERSÃO DE BASE 10 PARA BASE β

$(\underline{17},\overline{5})_{10} = ((i),(ii))_\beta$

Procedimento de conversão:

i) na parte inteira do número: divide-se sucessivamente a parte inteira do número decimal pela base β, e constrói-se o novo número escrevendo o último quociente e os restos obtidos nas divisões, para separar as potências sucessivas de β componentes da parte inteira. Desta forma agrupam-se as diferentes potências da base β.

Ex. 10: $(19)_{10} = (\ )_2 \Rightarrow$

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

$\rightarrow \quad (10011)_2 = 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 1.2^0$

$(527)_{10} = (\ )_{16} \Rightarrow$

```
527 | 16
 15  32 | 16
     0   2
```

$\rightarrow \quad (20F)_{16} = 2.16^2 + 0.16^1 + F.16^0$

ii) na parte fracionária do número: multiplica-se sucessivamente a parte fracionária do número decimal pela base β, e constrói-se o número escrevendo os inteiros resultantes de cada produto. A parte fracionária restante é novamente multiplicada por β até que o produto final seja um inteiro ou, a quantidade limite de dígitos na representação seja atingida. Desta forma constróem-se as frações sucessivas de β.

Ex. 11: $(0,03125)_{10} = (\ )_2$

0,03125	0,06250	0,125	0,25	0,5
x 2	x 2	x 2	x 2	x 2
0,06250	0,12500	0,250	0,50	1,0

$= (0,00001)_2$

Ex. 12: $(0,1)_{10} = (\ )_2$

0,1	0,2	0,4	0,8	0,6	0,2	0,4	0,8	0,6	0,2
x 2	x 2	x 2	x 2	x 2	x 2	x 2	x 2	x 2	x 2
0,2	0,4	0,8	1,6	1,2	0,4	0,8	1,6	1,2	0,4
0	0	0	1	1	0	0	1	1	0 ...

$(0,0\underline{0011}\ \underline{0011}...)_2$

dízima periódica ↵

Então: $(0,1)_{10} = (0,0001100110011\underline{0011...})_2 \cong (0,0001100110011)_2 \cong 2^{-4}.(1,100110011)_2 \cong$

$(0,1)_{10} \cong (0,099975585)_{10}$ ↳ dígitos desprezados (arredondamento) por limitação de armazenamento no registro binário (10 bits significativos)

Devemos notar neste exemplo que, na conversão das bases, parte dos dígitos da representação binária são descartados, por limitação do número de dígitos representáveis, o que gera um erro de arredondamento.

Exercícios:

Exercício - Efetuar as seguintes conversões de base:

a) $(10,57)_{10} = (\ )_2$
b) $(10,1011)_2 = (\ )_{10}$
c) $(513,98)_{10} = (\ )_{16}$
d) $(A,0F)_{16} = (\ )_{10}$

Exercício - Implemente o algoritmo abaixo em um compilador com processamento numérico (por exemplo: Pascal, C, Fortran, ...).

```
Início
        leia n
        x = 1/n
        Para i = 1 até 100
                x = (n + 1) x - 1
                imprimir i , x
        Fim para
Fim
```

Testar para n = 2, 3, 10 e 16 e avaliar a evolução de x com o número de iterações i. Note que se $x = 1/n \Rightarrow x \Rightarrow (n + 1).1/n - 1 = 1 + 1/n - 1 = 1/n = x$. Logo, o valor de x não deveria se alterar com cálculos sucessivos.

1.3 - Existe uma base onde todo racional tem representação finita, de acordo com George Cantor (1869) “todo racional tem representação finita na base fatorial”.

Conceitualmente, a base fatorial é semelhante à decimal com a diferença de que em um número $X_{F!} = (a_n\ a_{n-1}\ ...\ a_1\ ,\ a_{-1}\ a_{-2}\ ...\ a_{-m})$ cada a_i só pode assumir um valor do intervalo $0 \le a_i \le |\ i\ |$ e

$(X_i)_{F!} = (a_n\ a_{n-1}\ ...\ a_1)_{F!} = \left(\sum_{i=n}^{m} a_i\ i!\right)_{10}$ é a parte inteira e

$(X_{-i})_{F!} = (0,\ a_{-1}\ a_{-2}\ ...\ a_{-m})_{F!} = \left(\sum_{i=1}^{m} a_{-i} \,/\, (i+1)!\right)$ é a parte fracionária

Lembre-se que:

$(4321)_{F!}$ terá como seu sucessor $(10000)_{F!}$
↓ ↓

$(119)_{10}$ $(120)_{10}$

Represente os números na forma fatorada e converta para a base decimal:

i) $(3021)_{F!}$ = 3x4! + 0x3! + 2x2! + 1x1! = $(77)_{10}$
ii) $(321,123)_{F!}$ =
iii) $(0,02)_{F!}$ =
iv) $(0,113)_{F!}$ =

Note que nos exercícios (ii), (iii) e (iv) tem-se representações exatas de números racionais, que na base decimal são dízimas periódicas.

1.5.3 - CONVERSÕES DIRETAS ENTRE BINÁRIO E HEXADECIMAL:

Estas conversões são importantes para se entender os mecanismos de operacionalização de máquinas digitais que implementam as operações aritméticas em base binária e visualizar as representações em base hexadecimal.

Sabemos que um dígito hexadecimal corresponde a quatro dígitos binários, pois $16^1 = 2^4$ e note que $(15)_{10} = (F)_{16} = (1111)_2$

Então, fazemos a conversão direta associando a cada um dígito hexadecimal quatro dígitos binários. Para tal, agrupamos os dígitos binários em grupos de quatro a partir da posição da vírgula, para a direita e para a esquerda. Caso seja necessário, completa-se o grupo de quatro *bits* com zeros não significativos.

Ex. 13: $(A1,B)_{16} = (\ )_2$

Como:

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

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

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

temos:

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

Ex. 14: $(10001\ ,\ 01001\ 1001\ 1001\ ...)_2 = (\ )_{16}$

Agrupando-se os dígitos em grupos de quatro e completando com zeros:

$\Rightarrow (\underline{0001}\ \underline{0001}\ ,\ \underline{0100}\ \underline{1100}\ \underline{1100}\ \underline{1100}\ ...)_2$

1 1 4 C C C $\Rightarrow$ $(11,4CCC...)_{16}$

Ex. 15: $(110001,01) = (\ )_{16}$

$(\underline{0011}\ \underline{0001}\ ,\ \underline{0100})_2 = (31,4)_{16}$

3 1 4

Pois, $(00110001,0100)_2 =$

$$\underline{0x2^7 + 0x2^6 + 1x2^5 + 1x2^4} + \underline{0x2^3 + 0x2^2 + 0x2^1 + 1x2^0} + \underline{0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 0x2^{-4}}$$

$$(0x2^3 + 0x2^2 + 1x2^1 + 1x2^0)\,(2^4)^1 + (0x2^3 + 0x2^2 + 0x2^1 + 1x2^0)\,(2^4)^0 + (0x2^3 + 1x2^2 + 0x2^1 + 0x2^0)\,(2^4)^{-1}$$

$$\underline{3 \times 16^1} \qquad \underline{1 \times 16^0} \qquad \underline{4 \times 16^{-1}}$$

logo

$3x16^1 + 1x16 + 4x16 = (31,4)_{16}$

Obs.: Nas conversões diretas entre as bases binária e hexadecimal (ou entre as bases binária e octal), não há perda de dígitos (arredondamento). Mas nas conversões de base decimal para base

binária, ou para base hexadecimal, ou para base octal, podemos perder dígitos significativos. Por exemplo:

$$(17,3)_{10} = (11,4CCC\ ...)_{16} = (\ )_{10}$$
$$= 1 \cdot 16^{1} + 1 \cdot 16^{0} + 4 \cdot 16^{-1} + 12 \cdot 16^{-2} + 12 \cdot 16^{-3} + \ ...$$

Assim,

$$(17,3)_{10} = (11,4CCCC...)_{16} \cong (11,4CC)_{16} \qquad \text{(5 significativos)}$$
$$\cong (17,296875)_{10}$$

Exercícios:

1.4 - Converter os números para as bases na ordem indicada:

a) $(10111,1101)_2 = (\ )_{16} = (\ )_{10}$
b) $(BD,0E)_{16} = (\ )_{10} = (\ )_2$
c) $(41,1)_{10} = (\ )_2 = (\ )_{16}$

Obs.: Verifique se houve perda de dígitos significativos em alguma das conversões, considerando um número limitado de dígitos representáveis.

2. REPRESENTAÇÃO DIGITAL DE NÚMEROS:

2.1. - INTRODUÇÃO

De uma maneira geral, nos sistemas computacionais, um número $X \in \Re$ é representado na forma de notação em ponto flutuante, de maneira a racionalizar o armazenamento digital.

Se utilizassemos um armazenamento em **ponto fixo** (vírgula fixa) seria necessário um número de posições (dígitos) no mínimo igual a variação dos limites dos expoentes. Por exemplo, para se obter a representação de uma calculadora científica comum com limites positivos entre $1{,}0.10^{-99}$ e $9{,}999999999.10^{+99}$ seria necessário:

(i). Entre $1{,}0.10^{-99}$ e 1 seriam necessárias 99 posições:

$1{,}0.10^{-99}$ = 0,000.....00001 $\rightarrow$ 99 dígitos após a vírgula

(ii). Entre 1 e $9{,}99999999.10^{+99}$ seriam necessárias 100 posições:

$9{,}99999999.10^{+99}$ = 9999999999000.....0000, $\rightarrow$ 100 dígitos inteiros

(iii). Seria necessário mais uma posição para o sinal (s), para as representações de negativos, totalizando 200 posições em cada registro:

s		

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

Por outro lado, em uma representação em **Ponto Flutuante**, esta calculadora científica funciona com pouco mais de dez dígitos, incluindo posições reservadas ao expoente.

Então, em uma representação em Ponto Flutuante, onde a vírgula flutua segundo um certo padrão, temos a seguinte representação genérica na base β:

$$X = \pm [\, d_1/\beta + d_2/\beta^2 + d_3/\beta^3 + ... + d_t/\beta^t \,] \,.\, \beta^{exp}$$

ou

$$X = \pm (0{,}d_1\, d_2\, d_3 \,...\, d_t)_\beta \,.\, \beta^{exp}$$

onde

d_i = números inteiros contidos em $0 \le d_i \le (\beta - 1)$ (i = 1, 2, ... , t) que constituem a mantissa.

Obs.: É necessário algum tipo de normalização para padronização da mantissa, no caso adota-se $d_1 \neq 0$.

exp = expoente de β, assume valores limites I (Inferior) e S (Superior) onde $I \le exp \le S$.

t = número de dígitos significativos do sistema de representação, é chamado de precisão da máquina.

Ex. 16: Representar em ponto flutuante:

a) $(3{,}501)_{10} = (3/10 + 5/10^2 + 0/10^3 + 1/10^4) \,.\, 10^1 = 0{,}3501 \times 10^1$

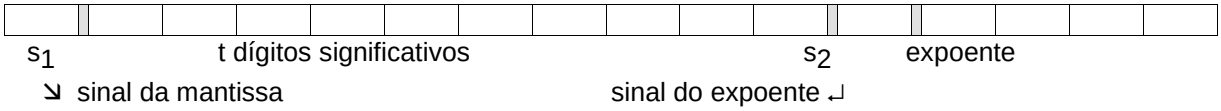
b) $(101{,}011)_2 = (1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + 1x2^{-6}) \,.\, 2^3 = (0{,}101011)_2 \,.\, 2^3$

Vamos agora exemplificar a representação digital de números em computador mostrando três exemplos práticos:

2.2. - PADRÃO 16 BITS

Vamos mostrar o sistema de Representação em Ponto Flutuante em uma máquina binária ($\beta = 2$), com t = 10 dígitos na mantissa e expoentes limitados entre I = -15 e S = + 15 ($15_{10} = 1111_2$), de modo que simbolicamente temos: $F(\beta, t, I, S) = F(2, 10, -15, 15)_{10}$. Esta é a representação clássica da *máquina de 16 bits*, que é detalhada aqui por motivos históricos.

Representação esquemática de dígitos significativos binários, onde cada *bit* é alocado em um registro (célula quadrada):



Convenciona-se que:

- Se $s_1 = 0 \rightarrow$ número positivo.
- Se $s_1 = 1 \rightarrow$ número negativo.
- s_2 idem.

No registro total tem-se:

- 1 *bit* para sinal da mantissa .
- 10 *bits* para armazenar os dígitos significativos da mantissa (t=10).
- 1 *bit* para sinal do expoente.
- 4 *bits* para o módulo do expoente

Totalizando 16 *bits* neste registro.

Devemos notar também que os dígitos significativos são armazenados no padrão de normalização com $d_1 \neq 0$, conforme estabelecido anteriormente.

Ex. 17: Representar + 0,101011 . 2^3 na máquina de 16 *bits* estabelecida anteriormente.

Convertendo-se o expoente: $(3)_{10} = (0011)_2$, tem-se

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

Limites da Representação em ponto Flutuante

Estes limites de representação serão exemplificados através da máquina de 16 bits:

a). Menor positivo representável (m.p.):

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

⌊ Lembre-se de que toda representação na máquina de 16 bits usa normalização com padrão $d_1 \neq 0$.

m.p. = $+(0,1)_2 \cdot 2^{-15} = (2^{-1} \cdot 2^{-15})_{10} = (2^{-16})_{10} = (0,0000152587)_{10}$

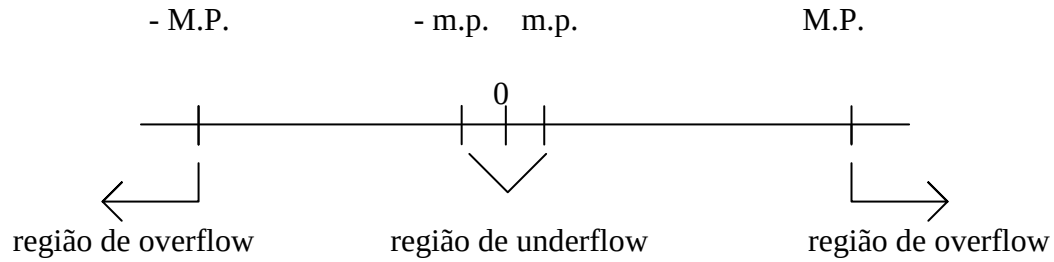
b) Maior positivo representável (M.P.):

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

M.P. = $+(0,1111111111)_2 \cdot 2^{15} = (2^{-1} + 2^{-2} + 2^{-3} + ... + 2^{-10}) \cdot 2^{15} = (32736)_{10} \cong (1 \cdot 2^{15})$

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

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



Obs.*:*

- região de underflow: $\{x \in \Re \mid -mp < x < mp\}$
- região de overflow: $\{x \in \Re \mid x < -MP \text{ e } x > MP\}$

c) Representação do zero:

É obtida com mantissa nula e o menor expoente representável (I).

Ex. 18: Representar o zero em F(2,10,-15,+15).

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

expoente mínimo

Devemos lembrar que este é o único número escrito não normalizado, pois sua mantissa é zero.

No exemplo 20, a seguir, pode-se visualizar o que poderia acontecer se o expoente do zero fosse diferente do limite inferior I.

Ex. 19: Simular a operação de adição: 0,000135 + 0 na máquina F(10,4,-10,+10)

↓ ↓

a b

1º) Considerando a representação do zero com expoente nulo (b = 0,0000 . 10^0):

a = 0,1350 . 10^{-3}
b = 0,0000 . 10^0

a = 0,000135 . 10^0

b = 0,0000 . 10^0

a + b = 0,0001 . 10^0 = 0,1000 . 10^{-3}

2º) Considerando a representação do zero com expoente mínimo I (b = 0,0000 . 10^{-10}):

a = 0,1350 . 10^{-3}
b = 0,0000 . 10^{-10}

a = 0,1350 . 10^{-3}
b = 0,0000 . 10^{-3}

a + b = 0,1350 . 10^{-3} = a

Neste segundo caso o zero representado pela máquina digital representa corretamente o elemento neutro da operação de adição.

d). Número máximo de elementos representáveis:

Podemos notar que a distribuição de números representáveis em ponto flutuante é discreta (somente alguns valores são representáveis), enquanto a distribuição de valores na parte Real é contínua (qualquer valor é representável).

Ex. 20: Representar os dois primeiros números positivos do sistema F(2, 10, -15, +15)

1º positivo -

0	1 0 . . .	0	1	1 1 1 1

$= (0,0000152587)_{10}$

2º positivo -

0	1 0 . . .	0 1	1	1 1 1 1

$= (0,0000152885)_{10}$

Ex. 21: Caso uma operação aritmética gere o número $(0,00001527)_{10}$, como ele será representado?

Como o valor acima não tem representação binária exata, ele será representado pelo valor discreto mais próximo, no caso $(0,0000152587)_{10}$ que é o menor positivo representável (mp).

Pode-se notar que a distribuição de números representáveis de F(β, t, I, S) não é uniforme em $\Re$, e que para cada potência da base β existe uma quantidade fixa de números representáveis dada por:

$$NC = (\beta - 1) . \beta^{t - 1}$$

Ex. 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 em cada potência da base ($(\beta - 1).\beta^{t-1} = 4$ para $\beta = 2$ e $t = 3$), com as quatro possibilidades de expoentes ($S - I + 1 = 4$ para $S = 2$ e $I = -1$) define o número total de positivos representáveis (NP = 16).

Desta forma o número total de elementos representáveis em uma máquina genérica F(β, t, I, S) é dado por:

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

incluindo os positivos, negativos e o zero.

Ex. 23: Em F(2, 10, -15, +15) (máquina de 16 *bits*) temos:

$NF = 2 . (2 - 1) . (15 - (- 15) + 1) . 2^{10-1}$

$= 31745$ elementos incluindo os positivos, negativos e o zero.

Ex. 24: Em F(10, 10, -99, +99) (calculadora científica comum) temos:

$NF = 2 . (10 - 1) . (99 - (- 99) + 1) . 10^{10-1} + 1$

$= 3,582 . 10^{12}$ elementos

Esta representação da máquina 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, dentre outras. Mais tarde surgiu o padrão **IEEE 754** (1985), que é amplamente utilizado no armazenamento de variáveis (vide seção 4.1).

e). Polarização na Representação em Ponto Flutuante

Polarização (ou Excesso) é um valor acrescentado (em Excesso) a todos expoentes de um sistema de representação em ponto flutuante com o objetivo de tornar todos os expoentes positivos e ampliar a representação do expoente superior (S). Naturalmente todas as operações aritméticas devem considerar esta polarização introduzida.

Ex. 25: Na máquina de 16 *bits* F(2, 10, -15, 15) podemos usar uma polarização p = + 15.

$p = +15 = + (1111)_2$

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

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

5 *bits*

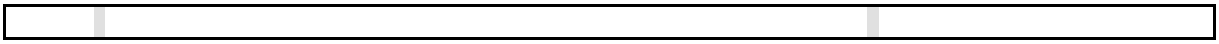
Como I e S têm agora o mesmo sinal, (+), podemos usar todos os registros binários reservados ao expoente, inclusive a posição do sinal, para representar o expoente polarizado (sem o sinal). Assim, os limites polarizados do expoente são:

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

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

Podemos aqui aproveitar melhor os 5 *bits* reservados ao expoente tomando o maior valor possível, adotando $S = (11111)_2 = (31)_{10}$.

Na forma polarizada qualquer número v representado nesta máquina deverá seguir a forma abaixo:



s_1 t dígitos significativos da mantissa expoente

$$v = (-1)^s . (0, \text{mantissa})_2 . 2^{exp - 15}$$

Ex. 26: Na representação

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

tem-se: s = 0 , m = 110100000 e $exp = (10010)_2 = (18)_{10}$

$$v = (-1)^0 . (0, 1101000000)_2 . 2^{18 - 15} = +(110,1)_2$$

Outra otimização adotada no padrão IEEE 754 foi o não armazenamento do primeiro bit da mantissa, que é sempre unitário, ou seja, usa-se a representação implícita do primeiro bit e abre-se uma posição binária para armazenamento de um novo bit:

$$v = (-1)^s . (1, \text{mantissa})_2 . 2^{exp - 15}$$

que veremos em detalhes na seção 4.1.

Exercícios:

2.1 - Na máquina F(2, 3, -3, +3) com $d_1 \neq 0$ (não polarizada) calcule:

a) O número de elementos representáveis;
b) Esquematize a representação de todos os elementos positivos na base 2;
c) Defina as regiões de underflow e overflow;
d) Estime a precisão decimal equivalente;

e) Proponha uma transformação da máquina F apresentada em uma máquina com polarização que utilize os limites dos 3 *bits* totais reservados ao sinal.

2.2 - Na máquina F(2,3,0,7) com $d_1 \neq 0$ e polarização p = +3 calcule:

a) O número de elementos representáveis;
b) Esquematize a representação de todos os elementos positivos na base 2;
c) Defina as regiões de underflow e overflow;
d) Estime a precisão decimal equivalente;

2.3). Precisão versus Exatidão

- PRECISÃO: é um conceito objetivo que estabelece a quantidade de algarismos significativos que representam um número. A precisão de uma máquina digital é definida como o número de dígitos t da mantissa na base β, e a precisão decimal "d " equivalente pode ser definida baseada na equivalência entre as variações dos dígitos menos significativos em cada base, da seguinte forma:

$$10^{1-d} = \beta^{1-t}$$
$$\log(10^{1-d}) = \log(\beta^{1-t})$$
$$1 - d = (1 - t)\log\beta$$
$$d = 1 + (t - 1)\log\beta$$

Ex. 27: Calcule a precisão decimal equivalente da máquina de 16 *bits* F(2, 10, -15, 15)

$\beta = 2 \quad t = 10$
$d = 1 + (10 - 1)\log 2$
$d = 1 + 9.(\log 2)$
$d = 3{,}71$
$d = 3 - 4$ dígitos (ou seja, pode representar entre 3 e 4 dígitos)

Ex. 28: Considere uma máquina F(2, 27, -20, 20)

$2^{-26} = 10^{-d+1} \Rightarrow \log 2^{-26} = \log 10^{-d+1}$
$-d + 1 = -26\log 2 \Rightarrow d = 1 + 26\log 2 \approx 8{,}8$

Assim, esta última máquina tem entre 8 e 9 dígitos significativos equivalentes. Isto não significa que todas as frações decimais de 8 dígitos possam ser representadas precisamente em 27 *bits*, visto que a representação é discreta (com espaços vazios entre dois números consecutivos) e nem todas as frações decimais têm representação binária finita. Isto significa que todos as representações binárias da máquina estão "corretas" para 8 dígitos significativos na base 10, ou seja, apresentam decimais equivalentes com pelo menos 8 dígitos corretos.

- EXATIDÃO: conceito relacionado com a forma que melhor representa uma grandeza numérica, ou seja, uma representação é mais exata quando tem o menor desvio (erro) em relação ao valor exato.

Ex. 29: Representar o π (3,1415926535) por:

(a) 3,14 $\rightarrow$ precisão de três dígitos
(b) 3,151 $\rightarrow$ precisão de quatro dígitos
(c) 3,1416 $\rightarrow$ precisão de cinco dígitos

Obs.: Note que se fossemos classificar o números acima quanto a exatidão teríamos o seguinte:

a) é mais exato que (b) (ou seja, (a) está mais próximo de π do que (b));
b) é menos exato que (c);
c) é mais exato que (a).

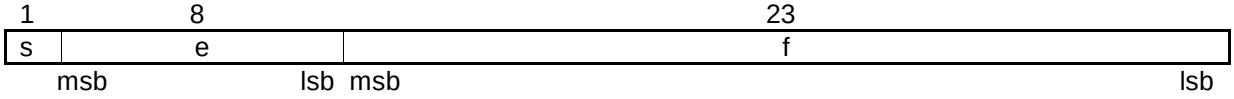
3 – REPRESENTAÇÃO NUMÉRICA SEGUNDO O PADRÃO IEEE 754 (1985)

Este padrão é utilizado em linguagens comerciais como o Pascal e C, e será apresentado a seguir na sua forma esquemática para representação em ponto flutuante e na forma de variáveis inteiras.

4.1). VARIÁVEL SINGLE DO PASCAL (OU FLOAT DO C):

Padrão: 4 bytes ou 32 bits (precisão de 7 a 8 dígitos significativos equivalentes).

Neste padrão um número real v pode ser representado por:



onde

$s = 0 \Rightarrow$ v positivo e $s = 1 \Rightarrow$ v negativo
e = expoente
f = mantissa
polarização $= (127)_{10} = 2^7 - 1 = (01111111)_2$
msb = bit mais significativo e lsb = bit menos significativo

Um número v armazenado no registro acima é interpretado da seguinte forma:

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

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

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

- Se $e = 255$, então v pertence a região de overflow.

Obs.:

(i). A representação destes registros binários em computadores digitais é feita em grupos de bytes (8 bits) escritos de forma invertida (de traz para frente) em relação ao esquema binário apresentado acima. Neste exemplo tem-se quatro bytes, onde cada byte é composto por dois registros hexadecimais (8 bits).

A seguir apresenta-se um exemplo desta representação para a fração 1/10, que representada na variável SINGLE do Pascal gera:

x = 0.10000000149 - (representação decimal, note o erro de arredondamento)

.
(ii). Se na janela 'watch' do Pascal, onde se pode visualizar as variáveis na base hexadecimal escrevendo 'x,m', tem-se a seguinte representação hexadecimal de uma variável SINGLE:

x,m = CD CC CC 3D - (representação hexadecimal no computador)

Esta deve ser interpretada na forma de bytes em ordem invertida, para compor o registro binário correspondente. Para obter este registro procede-se da seguinte forma:

a). Invertemos os bytes (grupos de dois hexadecimais):

3D CC CC CD

b). Efetua-se a conversão direta para base binária:

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

c). Distribui-se os bits no registro SINGLE:

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

(CD CC CC 3D - representação hexadecimal no computador)

d). Interpretando os 32 bits acima pode-se converter o registro para decimal, conforme segue:

s = 0
$e = (01111011)_2 = (123)_{10}$ (0<e<255)
$f = (10011001100110011001101)_2$

$x = (-1)^0 . 2^{123-127} . (1, 10011001100110011001101)_2 .$
x = 0.10000000149 (Representação do Pascal)

Os registros sublinhados representam os arredondamentos gerados na representação SINGLE de 1/10.

e). Para representar $(0,10)_{10}$ na base binária no padrão IEEE de 32 bits temos duas possibilidades:

e.1). Converter $(0,10)_{10}$ para binário e adequar o resultado ao padrão 32 bits:

$(0,10)_{10} = (0,0001100110011001100110011001100110011001100110011...)_2$
$(0,10)_{10} = 2^{-4} . (1,10011001100110011001100110011001100110011001100110011...)_2$

Numero positivo s=0.

$(0,10)_{10} = (-1)^0 . 2^{-4} . (1,10011001100110011001100 \mid 110011001100110011...)_2$
→ ARREDONDAMEMTO na mantissa f
$(0,110011001100110011...)_2 > (0,5)_{10}$
+1
$(0,10)_{10} \cong (-1)^0 . 2^{-4} . (1,10011001100110011001100)_2$

$(0,10)_{10} \cong (-1)^0 . 2^{-4} . (1,10011001100110011001101)_2$

$(e-127)_{10} = (-4)_{10} \Rightarrow e = (123)_{10} = (01111011)_2$

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

e.2). Converter $(0,10)_{10}$ diretamente para o padrão 32 bits, considerando 0 < e < 255:

$(0,10)_{10} = (-1)^s . 2^{(e-127)} . (1,f)_2$

(e.2.1). s = 0 => para números positivos $+(0,10)_{10}$

(e.2.2). Considerando que ainda temos 2 incógnitas, 'e' e 'f', vamos considerar o valor Mínimo de 'f', f = 0, e calcular 'e':

$(0,10)_{10} = (-1)^0 . 2^{(e-127)} . (1,0)_2$

No caso, determina-se um valor de 'e' maior que o verdadeiro, no caso, $'e' = (123,6781...)_{10}$ e toma-se o seu menor inteiro, $'e' = (123)_{10}$ (DESCONSIDERAR A PARTE FRACIONARIA)

(e.2.3). Agora determinamos o valor de 'f' com os valores de 's' e 'e' obtidos acima:

$(0,10)_{10} = (-1)^0 . 2^{(123-127)} . (1,f)_2$

$(1,f)_2 = (0,1)_{10} / 2^{(123-127)}$

$(1,f)_2 = (1,6)_{10}$ => (1,f deve ser um número entre 1 e 2, se 'e' foi escolhido corretamente)

Convertendo $(1,6)_{10}$ para binário, temos:

$(1,6)_{10} = (1,10011001100110011001100 \mid 110011001100110011...)_2$

→ ARREDONDAMEMTO na mantissa f

$(0,110011001100110011...)_2 > (0,5)_{10}$

+1

$(1,6)_{10} \cong (1,10011001100110011001100)_2$

$(1,6)_{10} \cong (1,10011001100110011001101)_2$

Resumindo:
s = 0
$e = (123)_{10} = (01111011)_2$
$(1,f)_2 \cong (1,10011001100110011001101)_2$

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

(ii). Note que a normalização da mantissa é uma otimização decorrente da normalização aplicada na máquina de 16 bits. No caso de máquinas binárias o primeiro dígito significativo não nulo é sempre o bit unitário, então não é necessário reservar um registro para armazenar sempre o mesmo valor. Assim, adotou-se a normalização padrão IEEE com um bit unitário antes da vírgula, que não é armazenado na memória dos computadores. Armazena-se a partir do 2º bit significativo.

(iii). Na representação específica dos primeiros valores positivos (e negativos) adotou-se uma flexibilização da normalização, permitindo mantissas com primeiro(s) dígito(s) nulo, o que gerou uma diminuição da região de underflow.

Ex. 40: Limites da variável SINGLE (ou FLOAT):

i). 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

(00 00 00 00 - representação em hexadecimal)

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

ii). Menor positivo (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 1

(01 00 00 00 - representação em hexadecimal)

$mp = (-1)^0 . 2^{-126} . (0,00000000000000000000001)_2$ (menor mantissa possível)

$mp = 2^{-126} . 2^{-23} = 2^{-149}$

mp = $1{,}4012985 \cdot 10^{-45}$

iii). Maior positivo (MP): $\begin{cases} s = 0 \\ e = (11111110)_2 = (254)_{10} \\ f = 11111111111111111111111 \end{cases}$

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

(FF FF 7F 7F - representação em hexadecimal)

MP = $(-1)^0 \cdot 2^{254-127} \cdot (1{,}11111111111111111111111)_2$

MP = $2^{127} \cdot (1{,}99999988)_{10}$

MP = $(3{,}4028235 \cdot 10^{38})_{10}$

4.2) Variável DOUBLE do Pascal (ou DOUBLE do C):

Padrão: 8 bytes ou 64 bits (precisão de 16 a 17 dígitos significativos equivalentes).

1	11	52
s	e	f
msb	lsb	msb lsb

onde

polarização = $(1023)_{10} = 2^{10} - 1 = (01111111111)_2$

Um número v armazenado no registro acima é interpretado da seguinte forma:

- Se $0 < e < 2047$, então $v = (-1)^s \cdot 2^{(e-1023)} \cdot (1,f)_2$
- Se $e = 0$ e $f \neq 0$, então $v = (-1)^s \cdot 2^{-1022} \cdot (0,f)_2$
- Se $e = 0$ e $f = 0$, então $v = (-1)^s \cdot 2^{-1022} \cdot (0,) = (-1)^s \cdot 0$ (zero)
- Se $e = 2047$, então v pertence a região de overflow.

4.3). VARIÁVEL EXTENDED DO PASCAL (OU LONG DOUBLE DO C):

Padrão: 10 bytes ou 80 bits (precisão de 19 a 20 dígitos significativos equivalentes).

1	15	1	63
s	e	i	f
msb	lsb	msb	lsb

onde

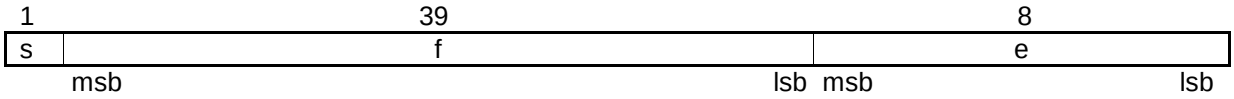
polarização = $(16383)_{10} = 2^{14} - 1 = (011111111111111)_2$

Um número v armazenado no registro acima é interpretado da seguinte forma:

- Se $0 < e < 32767$, então $v = (-1)^s \cdot 2^{(e-16383)} \cdot (i,f)_2$ (onde i pode assumir 0 ou 1)
  (se $e = 0 \Rightarrow i = 1$)
- Se $e = 32767$ e $f = 0$, então v pertence a região de overflow.

4.4) Variável REAL (padrão BORLAND):

Padrão: 6 bytes ou 48 bits (precisão de 12 a 13 dígitos significativos equivalentes).



onde

polarização $= (129)_{10} = 2^7 + 1 = (10000001)_2$

Um número v armazenado no registro acima é interpretado da seguinte forma:

- Se $0 < e <= 255$, então $v = (-1)^s . 2^{(e-129)} . (1,f)_2$
- Se $e = 0$, então $v = 0$, (independe de f)

Ex. 41: Simulação do algoritmo para avaliação da precisão decimal equivalente de uma variável.

```
p1=1
repita
        p1=p1/2
        p2=1+p1
até p2=1
```

Devemos ressaltar que no caso da operação soma os expoentes devem estar alinhados pelo maior expoente para que a soma possa ocorrer, independentemente da normalização da representação em ponto flutuante (vide cap. 3). Assim,

i). Representação da unidade: $1 = \begin{cases} e = (01111111)_2 = (127)_{10} \\ f = 00000000000000000000000 \end{cases}$

0	0 1 1 1 1 1 1 1	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

(00 00 80 3F - representação em hexadecimal)

$1 = (-1)^0 . 2^{(127-127)} . (1,00000000000000000000000)_2$

$1 = 2^0 . (1,00000000000000000000000)_2$

ii). Representação do menor número (P1) que pode ser somado a unidade (note que no momento da soma os expoentes devem estar alinhados para que a soma possa ocorrer no processador aritmético e portanto o expoente de p1 deve ser o mesmo da representação da unidade):

$P1 = \begin{cases} e = (01111111)_2 = (127)_{10} & - \text{ expoente igual a unidade (item i)} \\ f = 00000000000000000000001 & - \text{ menor mantissa possível} \end{cases}$

$P1 = (-1)^0 . 2^{(127-127)} . (0,00000000000000000000001)_2$ - este valor de P1 é conseguido após 23 divisões binárias.

Então,

$1 = 2^0 . (1,00000000000000000000000)_2$

$P1 = 2^0 . (0,00000000000000000000001)_2$

$1+P1 = 2^0 . (1,00000000000000000000001)_2$

Logo, $P1 = 2^0 . 2^{-23} = 2^{-23} = 1,1921.10^{-7}$

Neste caso a precisão decimal equivalente está entre 7 e 8 dígitos significativos.

P1 = $(-1)^0 . 2^{(104-127)} . (1,00000000000000000000000)_2$ - valor normalizado

$$P1 = \begin{cases} e = (01101000)_2 = (104)_{10} \\ f = 00000000000000000000000 \end{cases}$$

0	0 1 1 0 1 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

(00 00 00 34 - representação em hexadecimal)

Após a 23ª divisão de P1=1 por 2, P1 representará um número desprezível na soma frente a unidade, pois 1 + P1 = 1.

Exercícios:

4.1). a). Avalie as regiões de underflow e overflow de cada uma das variáveis apresentadas acima e faça uma verificação em um compilador Pascal ou C;

b). Avalie a precisão decimal equivalente de cada variável, através da fórmula de equivalência;

4.2). Simule o algoritmo do exemplo 41, em um compilador Pascal ou C, usando as variáveis apresentadas e determine o menor valor binário de p1 que pode ser adicionado a unidade sem perda total de significação (com operações aritméticas em binário), avaliando a precisão decimal equivalente. Acompanhe os resultados na janela 'watch' com as variáveis registradas em hexadecimal.

4.5). REPRESENTAÇÃO DE VARIÁVEIS INTEIRAS

Os formatos para representação de variáveis do tipo inteiras podem seguir diversos padrões:

4.5.1). *Shortint*: são tipos inteiros limitados aos valores - 128_{10} e + 127_{10}, sua representação interna é feita em forma de *byte* (8 *bits*) com *bit* de sinal. Valores negativos são armazenados em forma de complemento de dois.

Ex. 42: $+ 0_{10} = 00000000_2 = 00_{16}$

$+ 127_{10} = 01111111_2 = 7F_{16}$

$- 128_{10} = 2^8 - 128 = 128 = 10000000_2 = 80_{16}$

$- 5_{10} = 2^8 - 5 = 123 = 11111011_2 = FC_{16}$

Obs.: Veja como é feito o complemento de dois no exemplo 32.

Ex. 43: Se for efetuada a operação de adição entre + 127_{10} e -127_{10}, em forma de complemento e dois, tem-se:

$$\begin{cases} +127 & \begin{cases} 01111111 \\ \underline{10000001} \end{cases} \\ -127 & \end{cases}$$

$$00000000$$

4.5.2). *byte*: são tipos inteiros sem bit de sinal (não permite armazenar negativos). Estão limitados aos valores 0 e 255.

Ex. 44: $0_{10} = \underline{0000}\ \underline{0000}_2 = 00_{16}$

$255_{10} = \underline{1111}\ \underline{1111}_2 = FF_{16}$

4.5.3). *Integer*: são tipos inteiros limitados à faixa entre - 32768 e + 32767, correspondendo ao armazenamento como 2 *bytes* com *bit* de sinal ("negativos" são armazenados em forma de complemento de dois). ***Integer*** equivale ao '***Integer'*** da linguagem Pascal.

Ex. 45:

Zero $0_{10} = 0000\ 0000\ 0000\ 0000 = 0000_{16}$

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

$-32767_{10} = -0111\ 1111\ 1111\ 1111_2$

$1000\ 0000\ 0000\ 0000_2 \rightarrow$ complemento de um de -32767_{10}

$\underline{+\ 1}$

$1000\ 0000\ 0000\ 0001_2 \rightarrow$ complemento de dois de -32767_{10}

$= 8001_{16}$

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

$0111\ 1111\ 1111\ 1111_2 \rightarrow$ complemento de um de -32768_{10}

$\underline{+\ 1} \rightarrow$ soma 1

$1000\ 0000\ 0000\ 0000_2 \rightarrow$ complemento de dois de -32768_{10}

$= 8000_{16}$

Agora observe o que acontece se for executada a soma da 'unidade' com o 'maior positivo' da variável ***Integer*** 32767_{10}, ou seja, estamos calculando um número na região de **overflow**:

Unidade $1_{10} = 0000\ 0000\ 0000\ 0001_2 = 0001_{16}$

$+$

Maior Positivo $+32767_{10} = \underline{0111\ 1111\ 1111\ 1111}_2 = 7FFF_{16}$

$1_{10} + 32767_{10} = 1000\ 0000\ 0000\ 0000_2 = 8000_{16} \rightarrow$ Complemento de 2 (inicia com 1)

Precisamos obter o complemento de 2 novamente para obter o número 'negativo' armazenado:

$0111\ 1111\ 1111\ 1111_2$ → complemento de 1 de $1000\ 0000\ 0000\ 0000_2$

$+\ 1$ → soma 1

$-1000\ 0000\ 0000\ 0000_2$ → complemento de 2 de $1000\ 0000\ 0000\ 0000_2$

$= -8000_{16}$

$= -32768_{10}$

Então, cuidado podemos achar que 1_{10} $+32767_{10}$ = $+32768_{10}$, mas estamos armazenando -32768_{10}.

4.5.4). *Word:*** armazenamento de 2 bytes sem *bit* de sinal limitado a faixa entre 0 e 65535.

Ex. 46: $0_{10} = 0000_{16}$

$65535_{10} = 1111\ 1111\ 1111\ 1111_2 = FFFF_{16}$

$65536_{10} = 0000_{16} = 0_{10}$

Obs.: Note que a representação de números inteiros acima do limite superior acarreta uma grande perda de significação, cuja representação volta ao zero (vide ex. 45).

4.5.5). *Longint:*** corresponde ao "double word" (4 *bytes*) com bit de sinal. Limita-se entre -2147483648 (-2^{31}) e +2147483647 ($2^{31} - 1$). ***Longint*** equivale ao '**Int**' da linguagem C/C++.

Ex. 47: $0_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 00\ 00\ 00\ 00_{16}$

$+2147473647_{10} = 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 7F\ FF\ FF\ FF_{16}$

$-2147483647_{10} = -\ 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$ → complemento de 1

$+\ 1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 = 80000001_{16}$ → complemento de 2

$-2147483648_{10} = -1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$ → complemento de 1

$+\ 1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 80000000_{16}$ → complemento de 2

Observe novamente que se for executada a soma da 'unidade' com o 'maior positivo' da variável L***ongint*** 2147473647_{10}, estaremos calculando um número na região de overflow da variável ***Longint***:

Unidade $1_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 00\ 00\ 00\ 01_{16}$

+

Maior Positivo $+2147473647_{10} = 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 7F\ FF\ FF\ FF_{16}$

$1_{10} + 2147473647_{10} = 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 80\ 00\ 00\ 00_{16}$ →

É um complemento de 2 (inicia com 1)

Precisamos obter o complemento de 2 novamente para obter o número 'negativo' armazenado:

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$ → complemento de 1

$+\ 1$ → soma 1

$-1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$ → complemento de 2

$= -80\ 00\ 00\ 00\ 00\ 00\ 00_{16}$

$= -2147483648_{10}$

Então, cuidado podemos achar que 1_{10} $+2147483647_{10}$ = $+2147483648_{10}$, mas estamos armazenando -2147483648_{10}.

Obs.: Note que o armazenamento de inteiros negativos é sempre feito em forma de complemento de dois. Isto é uma vantagem do ponto de vista de operações aritméticas, pois o carregamento na unidade aritmética já está na sua forma final com expoentes iguais (alinhados) para efetuar as operações de adição (ou subtração). Note que isto só é possível, pois os expoentes equivalentes na notação de Tipos Inteiros são todos iguais a zero.

Por exemplo, tem-se a seguinte nomenclatura em C:

Type	Bytes	Bits	Range	
short int	2	16	-32,768 -> +32,767	(16kb)
unsigned short int	2	16	0 -> +65,535	(32Kb)
unsigned int	4	16	0 -> +4,294,967,295	(4Gb)
int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(2Gb)
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
float	4	32		
double	8	64		
long double	12	96		

4). TIPOS DE ERROS EXISTENTES EM MÁQUINAS DIGITAIS

É muito importante conhecer as possibilidades de erros na representação numérica em máquinas digitais e entender as suas causas para se poder estabelecer a confiabilidade de um software.

Todo estudo apresentado neste capítulo é necessário para que se possa entender as causas de cada tipo de erro existente em máquinas digitais.

Pode-se classificar os erros nos seguintes tipos principais:

5.1) Erros Inerentes:

São aqueles existentes nos dados de entrada de um software numérico. Decorre, por exemplo, de medições experimentais, de outras simulações numéricas, ...

5.2) Erros de truncamento:

Ocorrem quando quebramos um processo matematicamente infinito, tornando-o finito, por incapacidade de execução ou armazenamento.

A seguir serão apresentados exemplos de fontes de erro de truncamento:

Ex. 48: Veja a seguinte expansão em série infinita para e^x:

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

Sabe-se que não é possível usar infinitos termos para avaliar uma função, então é necessário estabelecer um limite para o número de parcelas utilizadas. Esta limitação nas parcelas gera um erro de truncamento na série, que corresponde ao somatório dos termos abandonados.

Obs.: A representação de f(x) em série será mostrada de forma ilustrativa a seguir,

É possível representar, de forma exata, uma função f(x) em um ponto qualquer x_0+x a partir de sua representação em x_0, através de expansão em Séries de Taylor, dada genericamente por:

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

Expandindo a função, por exemplo, em torno de $x_0 = 0$ (quando x_0=0, temos o caso particular da série Maclaurin), tem-se:

$$f(x) = e^x$$
$$f(x_0 = 0) = e^0 = 1$$
$$f'(x_0 = 0) = e^0 = 1$$
$$f''(x_0 = 0) = e^0 = 1$$
$$\vdots$$
$$f^n(x_0 = 0) = e^0 = 1$$

Gerando então,

$$f(0+x) = e^{x+0} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!} + O(x^{(n+1)}) = \sum_{i=0}^{n} \frac{x^i}{i!} + O(x^{(n+1)})$$

Se aproximamos e^x usando até o termo de ordem 'n', estamos desprezando os termos $O(x^{(n+1)}) = \frac{x^{n+1}}{(n+1)!} + \frac{x^{n+2}}{(n+2)!} + ...$

então $e^x \cong 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!}$

Assim, o termo $O(x^{(n+1)})$ caracteriza o Erro de Truncamento da aproximação, lembrando que x, nesse caso, representa o incremento de x entre o ponto inicial x_0 e o novo ponto x_0+x.

Trabalho:

Dado e^1 = 2.71828182845904523536028747135266249775724709369995959574966967...

a). Calcule o Erro exato cometido quando se aproxima e^x, via série de Taylor, usando até o termo de ordem 'n=7', para x=1, em uma variável IEEE de 32 bits (tipo float);
b). Calcule os erros Estimados de Arredondamento, Truncamento e Total, quando não dispomos do valor exato.

Ps.:

- O valor exato "Estimado", com pouco Arredondamento, pode ser calculado aproximando e^x, via série de Taylor, usando até o termo de ordem 'n' em uma variável IEEE de 64 bits (tipo double), cujos arredondamentos são muito inferiores ($Erro_{Arred}$=|$VA_{32bits,n}$ -$VE_{64bits,n}$|);
- O valor exato "Estimado", com pouco Truncamento, pode ser calculado aproximando e^x via série de Taylor, usando até o termo de ordem 'n_2=2*n' com variáveis IEEE de 64 bits (tipo double), para não ter influência dos arredondamentos ($Erro_{Truncam}$=|$VA_{64bits,n}$ -$VE_{64bits,2*n}$);
- O erro Total, quando não dispomos do valor exato, pode ser obtido por:
$Erro_{Total}$=|$VA_{32bits,n}$ -$VE_{64bits,2*n}$|, que considera variações no arredondamento das variáveis e no truncamento do processo de aproximação em série.

Ex. 49: Aproximações numéricas de limites de funções,

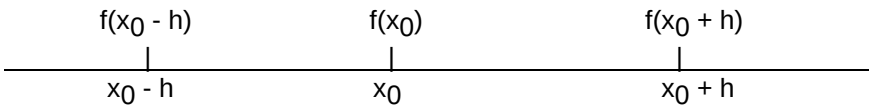
Por definição f'(x) é dada por,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Porém, se este limite exato não puder ser obtido, pode-se promover uma aproximação numérica deste, tomando o incremento h como finito e promovendo sucessivos refinamentos. Assim, pode-se obter uma seqüência de aproximações sucessivas de f'(x), com incremento cada vez menor, mas não se pode chegar ao incremento nulo (h $\rightarrow$ 0). Então, também se quebra o processo matemático, de refinamentos sucessivos, gerando um erro de truncamento do processo, que era matematicamente infinito, tornando-o finito.

Ex. 50: Aproximações de derivadas

Para avaliar numericamente $f'(x_0)$ a partir de três pontos vizinhos de f(x):



Pode-se subtrair $f(x_0 - h)$ de $f(x_0 + h)$:

$$f(x_0 + h) = f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2!} + f'''(x_0)\frac{h^3}{3!} + f^{iv}(x_0)\frac{h^4}{4!} + ... + f^n(x_0)\frac{h^n}{n!}$$

-

$$f(x_0 - h) = f(x_0) - f'(x_0)h + f''(x_0)\frac{h^2}{2!} - f'''(x_0)\frac{h^3}{3!} + f^{iv}(x_0)\frac{h^4}{4!} - ... + f^n(x_0)\frac{h^n}{n!}$$

$$f(x_0 + h) - f(x_0 - h) = f'(x_0)(2h) + 2.f'''(x_0)\frac{h^3}{3!} + 2.f^{IV}(x_0)\frac{h^4}{4!} + ...$$

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{1}{h}.f'''(x_0)\frac{h^3}{3!} - \frac{1}{h}.f^{V}(x_0)\frac{h^5}{5!} + ...$$

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

onde o termo de segunda ordem $O(h^2)$ representa o somatório de todos os termos decorrentes da aproximação em série, e é definido como Erro de Truncamento da aproximação.

$$O(h^2) = f'''(x_0)\frac{h^2}{3!} + f^{V}(x_0)\frac{h^4}{5!} + ...$$

Então, desprezando o termo $O(h^2)$, assumindo assim um erro de truncamento de segunda ordem, tem-se uma aproximação para $f'(x_0)$ dada por,

e $$f'(x_0) \cong \frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

5.3). Erros de Arredondamento

Ocorrem quando são desprezados os últimos dígitos que, ou não são fisicamente significativos na representação numérica, ou estão além da capacidade de armazenamento na máquina digital.

5.3.1). Arredondamento manual:

Ex. 51: Representar os seguintes números com quatro dígitos significativos:

69,348 = 69,35 → parcela descartada é maior que 5 ⇒ +1 no dígito anterior.
69,34433 = 69,34 → parcela descartada é menor que 500 ⇒ dígito anterior inalterado.
69,335 = 69,34 → parcela descartada é igual a 5 e dígito anterior impar ⇒
+1 no dígito anterior.
69,345 = 69,34 → parcela descartada é igual a 5 e dígito anterior par ⇒
dígito anterior inalterado.

No exemplo anterior o arredondamento foi feito de forma ponderada, baseado em critérios estatísticos para descartar parte dos dígitos do número. Pode-se seguir o seguinte raciocínio:

Analisando estatisticamente um conjunto de números que terão parcelas descartadas. Podemos admitir que, em uma distribuição normal de erros, 50% dos valores a descartar são maiores que 5 e que 50% são menores, então

- Se todos os dígitos a descartar forem simplesmente cancelados, sem nenhum critério de compensação, o conjunto inicial de números perde parte de seu significado, gerando um erro de arredondamento global.

- Se, por outro lado, procurar-se distribuir o erro de arredondamento entre os números do conjunto escolhido, pode-se minimizar os efeitos globais dos erros de arredondamento sobre este conjunto. Assim, promove-se uma atualização do dígito anterior ao descartado nos 50% dos casos cuja parcela descartada é maior que 5, adicionando-se uma unidade. Nos demais casos simplesmente descarta-se a parcela indesejada, sem nenhuma atualização no dígito anterior. Desta forma distribui-se, estatisticamente, a parcela perdida entre os elementos do conjunto.

No caso específico de parcelas descartadas iguais a 5, tem-se um impasse que deve também ser decidido estatisticamente. Nestes casos também dividem-se as possibilidades em dois grupos, um com parcela anterior par e outro com parcela anterior impar. No primeiro grupo simplesmente descartam-se os dígitos indesejados, sem nenhuma atualização no dígito anterior, e segundo grupo, com parcela anterior impar, atualiza-se este dígito adicionando-lhe uma unidade. Também nesta situação procurou-se distribuir estatisticamente o erro devido as parcelas descartadas.

O arredondamento também pode ser feito por cancelamento puro, onde a parte indesejada do número é simplesmente cancelada, independente do seu valor, assumindo um erro de arredondamento global para valores menores em todos os elementos de um conjunto de números.

5.3.2). Arredondamento em máquinas digitais:

Neste caso o arredondamento pode ocorrer nas seguintes situações básicas:

(i). Armazenamento de racionais ilimitados:

Ex. 52: Representar a fração $(1/3)_{10}$ em F(10,6,-99,+99) (normalização: vírgula antes do 1º dígito).

$$(1/3)_{10} = (0,\underbrace{333333}_{t=6}333...10^{0})_{10}$$

$(1/3)_{10} \cong (0,333333.10^{0})_{10} \Rightarrow$ Representação arredondada de $(1/3)_{10}$.

Ex. 53: Representar a fração decimal $(1/10)_{10}$ na máquina binária F(2,10,-15,+15).

$$\left(\frac{1}{10}\right)_{10} = \left(\frac{1}{1010}\right)_{2} = (0,00011001100110011...)_{2}$$

$\Rightarrow$ dízima periódica em base binária

$$(1/10)_{10} = (0,\underbrace{1100110011}_{t=10}0011...)_{2}.2^{-3}$$

$$\left(\frac{1}{10}\right)_{10} \cong (0,1100110011)_{2}.2^{-3}$$

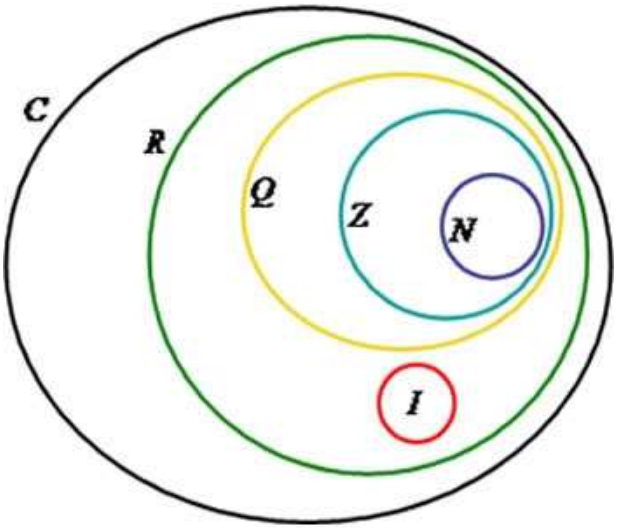
Obs.: Note que uma fração decimal exata (1/10 = 0,1) quando armazenada em uma máquina binária se transforma em uma fração binária periódica, que deve ser aproximada devido à limitação do registro em ponto flutuante utilizado no armazenamento.

(ii). Armazenamento de Irracionais:

O conjunto dos números irracionais compreende todas as representações através de dízimas

não periódicas e infinitas.

Conjuntos:



N - Naturais

Z - Inteiros

Q - Racionais

I - Irracionais

R - Reais

C - Complexos

Obs.: Existem representações que generalizam todos os números como complexos, de modo que um complexo com parte imaginária nula se torna um real.

Ex. 54: Representar em F(10,6,-99,+99),

(a). $\pi \Rightarrow 3.141592653589...$

$\pi \Rightarrow (0,314159\underline{2653589...})_{10} .10^{+1}$

$\pi \cong (0,314159)_{10} .10^{+1}$

(b) $\sqrt{2} \Rightarrow 1.414213562373...$

$\sqrt{2} \Rightarrow (0,141421\underline{3562373...})_{10} . 10^{+1}$

$\sqrt{2} \cong (0,141421)_{10} . 10^{+1}$

(iii). Abrangência limitada da notação em ponto flutuante:

Ex. 55: Efetue a soma de a = 0,0135 e b = 10,51 em F(10,4,-10,+10) e g = 0.

Representação em ponto flutuante:

$a = 0,1350 . 10^{-1}$

$b = 0,1051 . 10^{2}$

Vamos implementar a soma, de forma simplificada, usando alinhamento pelo maior expoente

$a = 0,1350 . 10^{-1}$

$b = 0,1051 . 10^{2}$

$a = 0,000\underline{135} . 10^{2}$

\+

$b = 0,1051 . 10^{2}$

$a + b = 0,1052 . 10^{2}$

Ex. 56: Efetue a soma de $a = (10,01)_2$ e $b = (0,0101)_2$ em F(2,4,-15,+15) e g = 0.

Representação em ponto flutuante:

$a = (0,1001)_2 \cdot 2^2$

$b = (0,1010)_2 \cdot 2^{-1}$

Vamos novamente implementar a soma, de forma simplificada,

$a = (0,1001)_2 \cdot 2^2$

$b = (0,0001\underline{010})_2 \cdot 2^2$

$a + b = (0,1010)_2 \cdot 2^2$

Obs.: Estes fatos ocorrem, geralmente, na soma de números com potências muito diferentes. Neste caso o número de menor potência pode perder significação, total ou parcial, frente ao número de maior potência. Ou seja, devido a faixa limitada de abrangência dos registradores em ponto flutuante, o número menor perde dígitos significativos quando comparado com o número maior.

(iv). Mudança de base para armazenamento e operações aritméticas

Sabe-se que a representação de números em base binária é amplamente utilizada em máquinas digitais (computadores), devido as suas vantagens no armazenamento e implementação de operações aritméticas.

O que ocorre na prática é que a interface entre o usuário e os computadores deve ser feita em base decimal, para que a representação de grandezas físicas seja naturalmente entendida pelos usuários dos computadores.

Então toda grandeza física é expressa inicialmente em base decimal, e o seu efetivo armazenamento nos computadores é feito em base binária, por isso é necessária uma conversão entre as bases decimal e binária e vice-versa.

Ex. 57: Representar $(0,1)_{10}$ em F(2,10,-15,+15).

$(0,1)_{10} = (0,00011001100110011...)_2 \Rightarrow$ fração decimal exata gerou dízima periódica binária

$(0,1)_{10} = 2^{-4}.(1,100110011.\underline{0011}....)_2 \rightarrow$ representação exata, anterior ao arredondamento

$(0,1)_{10} \cong 2^{-4}.(1,100110011.)_2 \cong (0,099975585)_{10} \rightarrow$ representação aproximada, pós arredondamento

Nesse caso, o Erro de Arredondamento pode ser calculado, pois temos o Valor Exato (VE) original e o Valor Aproximado (VA) , pós arredondamento:

VE= $(0,1)_{10}$

VA= $2^{-4}.(1,100110011.)_2 = (0,099975585)_{10}$

$$\text{Erro Relativo } \% = \left|\frac{\text{VA - VE}}{\text{VE}}\right|.100\% = -0,02441\%$$

Conseqüências:

Os erros de arredondamento podem causar:

(a). Perda de significação:

Esta é uma conseqüência de erros de arredondamento, que gera perda, total ou parcial, de dígitos significativos.

Esta perda de dígitos significativos pode ocorrer nos seguintes casos:

(a1). Soma de parcelas de grandezas muito diferentes:

Vide exemplos 55 e 56 apresentados anteriormente.

(a2). Subtração de parcelas de grandezas muito próximas:
Ex. 58: Efetuar a - b com a = 0,1351 e b = 0,1369 em F(10,4,-10,+10) e g = 0.

Efetuando a subtração de forma simplificada tem-se:

a = 0,1351
- b = - 0,1369
a - b = - 0,0018 = - $0,1800.10^{-2}$

Note que o resultado final não sofreu arredondamentos, mas perdeu dígitos significativos, pois as parcelas a e b tem quatro dígitos significativos e a subtração a - b tem apenas dois dígitos significativos.

Obs.: A expressão de Baskara, para a solução exata da equação de segundo grau, muitas vezes, aparece expressa de forma alternativa para minimizar perdas de significação:

Para $a x^2 + b x + c = 0$, tem-se as seguintes raízes:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$ (Formula de Baskara)

$$x_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$ (Fórmula obtida da racionalização do numerador da expressão anterior)

Pode-se observar que as duas formas formas apresentadas para a solução podem apresentar perdas de significação, quando a parcelas b e $\sqrt{b^2 - 4ac}$ forem de magnitudes próximas e estiverem sujeitas a uma operação de subtração.

Assim, recomenda-se utilizar as expressões propostas acima, escolhendo o sinal do radicando de modo que as parcelas b e $\sqrt{b^2 - 4ac}$ fiquem sujeitas a operação de adição nas duas parcelas, em uma expressão obtendo x_1 e em outra obtendo x_2.

Exercícios:

5.1). Achar as duas raízes de $x^2 + 62,10 x + 1 = 0$, utilizando o operador aritmético F(10,4,-99,+99) e g = 0 (quatro dígitos significativos nas operações).

a). Use a fórmula de Baskara normal;
b). Use a fórmula de Baskara racionalizada;
c). Avalie os erros relativos nas duas formas de avaliação das raízes, sabendo que os seus valores exatos são x_1 = - 0,01610 e x_2 = - 62,08.

(a3). Nas operações de divisão:

Em geral, nas operações de divisão entre duas parcelas quaisquer, normalmente, são gerados resultados com um número de dígitos maior que o permitido na representação em ponto flutuante.

Ex. 59: Efetuar a / b, com a = 1332 e b = 0,9876, no operador F(10, 4,-99,+99) e g = 0.

Efetuando esta operação em uma calculadora de 10 dígitos, tem-se,
a / b = 1348,72418

Porém, se esta operação está sujeita a apenas 4 dígitos significativos (F10,4,-99,+99), o resultado será,
a / b = 1348

Causando desta forma uma perda de dígitos significativos.

(b). Instabilidade Numérica:

A acumulação sucessiva de erros de arredondamento pode conduzir um algoritmo de repetição a resultados absurdos, por exemplo,

A avaliação sucessiva de x = f(x) com f(x) = (N+1) x - 1 $\Rightarrow$ é uma constante $\forall$ x=1/N (na ausência de arredondamentos)

Verifique esta afirmação, implementando o exercício 4.2 em computador.

Existem também outras formas de instabilidade, como aquelas associadas ao modelo matemático, por exemplo,

Ex. 60: Dada a função $f(x) = \frac{27985}{9{,}1 - x^2}$

Avalie f(x) em x = 3 e em x = 3,00001 utilizando uma calculadora científica com representação de 10 dígitos.

f(3) = 279850
f(3.00001) = 280018,0108

Note que uma variação de 0,0003333% na variável independente x gera uma variação de 0,06% no resultado final de f(x), ou seja, uma variação no resultado de cerca de 180 vezes superior. Isto carateriza uma instabilidade intrínseca do modelo matemático em relação aos seus dados de entrada.

Ex. 61: Avaliar f(x) = 1 - cos(x) para x = 10^{-4} (radianos), na mesma calculadora científica (10 dígitos).

f(10^{-4}) = 4,9 . 10^{-9}

Se for utilizada uma forma alternativa para o modelo da função:

g(x) = (1 - cos(x)) . $\left(\frac{1+\cos(x)}{1+\cos(x)}\right) = \left(\frac{sen^2(x)}{1+\cos(x)}\right)$

tem-se então,

g(10^{-4}) = 5 . 10^{-5}

Note que devido a perda de dígitos significativos por arredondamento, duas representações idênticas da mesma função geram respostas diferentes. Neste caso, também tem-se um exemplo de instabilidade do modelo, porém neste exemplo específico, se tem a possibilidade de reformular a sua representação, gerando uma forma com menor perda de significação (g(x)).

Considerações finais:

(i). A avaliação de erros numéricos em máquinas digitais pode ser feita caso se tenha disponibilidade de uma estimativa do valor exato para o resultado desejado.

Desta forma pode-se avaliar o Erro Numérico através das seguintes formas:

- Erro absoluto = | Valor obtido - Valor exato |

- Erro relativo = $\frac{\text{Erro absoluto}}{\text{Valor exato}}$

- Erro percentual = Erro relativo . 100%

A representação dos erros numéricos em forma de Erro relativo (ou percentual) é mais realística (vide exercício 4.3)

(ii). A estimativa de valores exatos, ou mais próximos do exato, para os resultados de algoritmos numéricos, podem ser obtidas das seguinte formas:

- **Estimar o valor mais exato como sendo aquele obtido com o algoritmo numérico operando em dupla precisão, ou seja, reavalia-se o valor obtido numericamente, mas agora com precisão superior, na expectativa de que os resultados sejam mais exatos.**

- Tentar estimar o valor mais exato através de simulação do algoritmo numérico em Sistemas de Computação Algébrica (SCA). Nestes sistemas pode-se recorrer a simulações com precisão ilimitada e alguns casos especiais é possível se proceder a simulação exata do algoritmo.

Obs.: Existem outras alternativas, como por exemplo, proceder simulações numéricas utilizando matemática intervalar, de forma que se possa limitar o erro existente a um intervalo aritmético. Outra possibilidade é o tratamento do erro como uma variável de comportamento estatístico, deste modo pode-se prever os limites do erro, segundo um tratamento estatístico de variáveis.

Exercícios:

5.2). Implemente o algoritmo abaixo em um compilador com aritmética numérica para P1 e P2 tipo real (Pascal, C, Fortran, ...)

```
p1:= 1;
i:= FALSE;
j:= 1;
Repita
   p1:= p1/2;
   p2:= p1 + 1;
   j=j+1;
   SE p2 <= 1 então
        Escreva ('Para minha precisao unitária', p1, ' eh  0 ', 'em ', j, ' repeticoes');
        i:= TRUE;
   fim SE
Até (i=TRUE);
```

Avalie a posição do 1º dígito significativo de P1, que indica a sua precisão relativa ao número 1 (unidade). Verifique que P1 vai diminuindo até que fique tão pequeno que não altera mais o valor de P2. Avalie a influência dos erros de arredondamento no resultados.

5.3) Considerando o compilador Turbo Pascal e suas cinco variáveis do tipo real:

TIPO	FAIXA	DÍGITOS	BYTES
real	2.9e-39...1.7e38	11-12	6
single	1.5e-45..3.4e38	7-8	4
double	5.0e-324...1.7e308	15-16	8
extended	3.4e-4932..1.1e4932	19-20	10
comp	-9.2e18..9.2e18	19-20	8

Execute o seguinte algoritmo e avalie seus resultados em um processador numérico:

```
Variáveis reais e, f, g, h, x, y;
   h:= 1/2;
   x:= 2/3 - h;
   y:= 3/5 - h;
   e:= (x + x + x) - h;
   f:= (y + y + y + y + y) - h;
```

```
g:= e/f;
Imprima ('h = ' ,h, '   ',h:5:15); {imprima em notação científica e também na notação decimal}
Imprima ('x = ' ,x, '   ',x:5:15);
Imprima ('y = ' ,y, '   ',y:5:15);
Imprima ('e = ' ,e, '   ',e:5:15);
Imprima ('f  = ' ,f, '    ',f:5:15);
Imprima ('g = ' ,g, '   ',g:5:15);
```

Experimente trocar o tipo das variáveis, no inicio do programa, execute o programa novamente e compare os resultados, com os tipos REAL, SINGLE, DOUBLE, EXTENDED E COMP.

5.4). Dadas algumas estimativas do valor exato e de valores aproximados numericamente em um algoritmo, avalie o erro absoluto, relativo e percentual, existente nas seguintes situações:

a). Valor aproximado = 1102,345 e Valor exato = 1100,9.
b). Valor aproximado = 0,01245 e Valor exato = 0,0119.
c). Verifique que o erro absoluto obtido, segundo as várias formas de avaliação, pode não refletir a realidade.

Conclusões:

O perfeito entendimento das causas dos erros numéricos gerados em máquinas digitais é um fator decisivo para que o analista numérico possa fazer a escolha computacionalmente mais eficiente, no momento de programar a resolução de um modelo matemático. Para resolver um determinado modelo o analista numérico deve se preocupar com:

- a escolha de métodos de resolução com menor número de operações aritméticas envolvidas;
- a escolha de um algoritmo com menor número de recursividade;
- a escolha de um compilador, 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:

- obter uma solução de custo mínimo, ou seja, com menor demanda de memória e menor tempo de CPU;
- dimensionar o grau de confiabilidade dos resultados obtidos como solução do modelo matemático.

Pois obter resultados em um programa de computador é muito fácil, qualquer programa compilado gera resultados numéricos, mas daí a se dizer que estes resultados são as respostas para a solução de um modelo matemático, tem-se um longo caminho. É necessário que se faça um minucioso processo de análise numérica dos resultados obtidos, para depois atribui-los à solução de um modelo matemático.