Representação Binária de Números Reais

- Representação numérica posicional:
  - $34{,}567_{10} = 3\cdot 10^{1} + 4\cdot 10^{0} + 5\cdot 10^{-1} + 6\cdot 10^{-2} + 7\cdot 10^{-3}$
  - $101{,}1001_{2} = 1\cdot 2^{2} + 0\cdot 2^{1} + 1\cdot 2^{0} + 1\cdot 2^{-1} + 0\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4}$

Representação em Ponto Flutuante

- A representação de ponto flutuante é feita utilizando notação científica, com alguns bits reservados para a *mantissa* e alguns bits para o *expoente*. Isto implica em que apenas números com uma quantidade pré-determinada de dígitos significativos podem ser representados.
- Representação em Ponto Flutuante Simples: 

Uma representação em ponto flutuante $fl(x) = SPF(\beta, t, m, M)$ do número x na base β é dada por

$fl(x) = \pm(.d_1 d_2 \dots d_t) \times \beta^{e}$, onde:

- $(d_1 d_2 \dots d_t)$ é a mantissa com t dígitos;
- $0 \le d_j \le \beta - 1 \quad \forall j$;
- $e \in [m, M]$ é o expoente (geralmente $m = -M$)

Dizemos que o número é *normalizado* quando $d_1 \neq 0$

Exemplo:

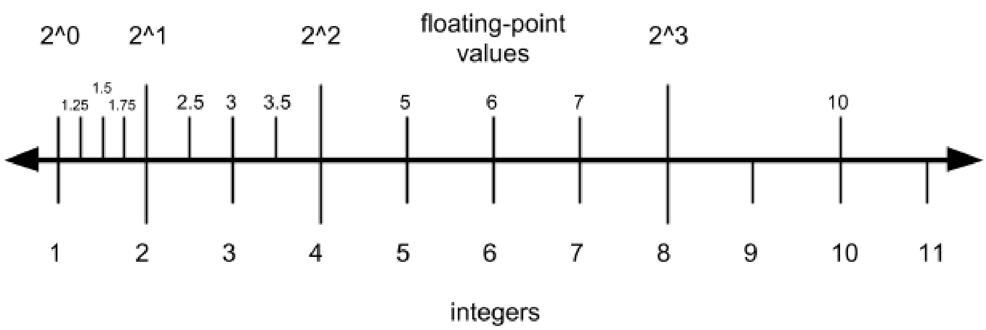
Seja o Sistema de Ponto Flutuante $SPF(\beta, t, m, M) = SPF(10, 3, -9, 9)$

Operações aritméticas: iguala-se o expoente ao maior e opera-se a mantissa. Ex.:
$x = 1.23 \times 10^{2}, y = 1.00 \times 10^{0}$

$x + y = 1.23 \times 10^{2} + 0.01 \times 10^{2} = 1.24 \times 10^{2}$

Precisão e Capacidade de Representação

Considere um $SPF(2,3,1,4)$ que dá origem à figura abaixo:

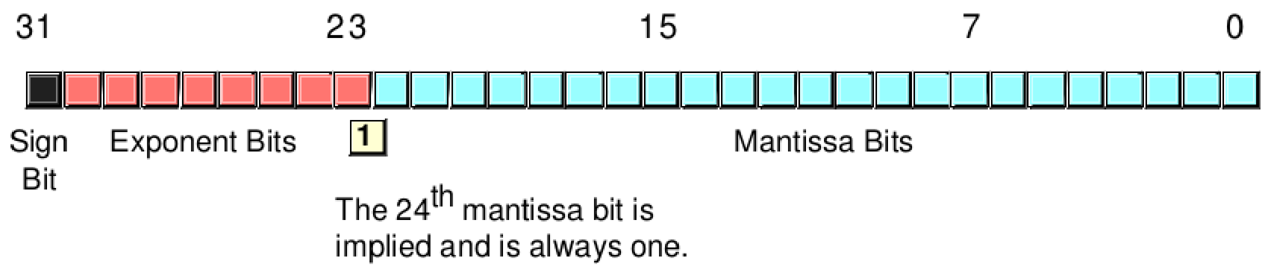


- Os números representáveis são:

$e=1$	$e=2$	$e=3$	$e=4$
$.100\times 2^1=1_{10}$	$.100\times 2^2=2_{10}$	$.100\times 2^3=4_{10}$	$.100\times 2^4=8_{10}$
$.101\times 2^1=1.25_{10}$	$.101\times 2^2=2.5_{10}$	$.101\times 2^3=5_{10}$	$.101\times 2^4=10_{10}$
$.110\times 2^1=1.5_{10}$	$.110\times 2^2=3_{10}$	$.110\times 2^3=6_{10}$	$.110\times 2^4=12_{10}$
$.111\times 2^1=1.75_{10}$	$.111\times 2^2=3.5_{10}$	$.111\times 2^3=7_{10}$	$.111\times 2^4=14_{10}$

- Observe que sempre há 4 números em ponto flutuante entre duas potências de dois
  - Para cada aumento na potência de 2, a quantidade de inteiros representados duplica, mas a quantidade de números em ponto flutuante é constante.
  - A precisão do número em ponto flutuante é proporcional à sua magnitude. Quanto maior o número, menor a precisão.
- À medida em que o valor aumenta, diminui a precisão do número em ponto flutuante, enquanto que a precisão do número inteiro continua a mesma (mais sobre isso quando tratarmos de erro).

Formato IEEE 754



- **Mantissa (f)**
  - contém a fração do número normalizada, sem representar o primeiro bit 1
    - há um dígito 1 implícito, de forma que o valor da mantissa é (1+f). Este primeiro bit é deslocado à esquerda do ponto, equivalendo a uma representação $(d_1.d_2 \ldots d_t) \times 2^e$
    - aumenta em 1 bit a precisão do número, mas como representar o zero?
  - Manter números normalizados utiliza o máximo de bits de precisão para os cálculos, ajustando o expoente.
- **Expoente (e)**
  - Representado como um número com um desvio $\Delta = 127$. Ex:
    - se expoente 4 então $e = 4 + 127 = 131_{10} = 10000011_2$
    - se $e = 01011101_2 = 93_{10}$ então o expoente é $93 - 127 = -34$
    - Permite a comparação dos números como se fossem inteiros.
- **Forma Geral**
  - $(-1)^s \cdot (1+f) \cdot 2^{(e-\Delta)}$, onde $\Delta = 127$ para ***float*** e $\Delta = 1023$ para ***double***
- **Exercícios**
  - Converter: 1 01111100 11000000000000000000000
    - $s = 1$, $e = 01111100 = 124_{10}$, $f = 11000000000000000000000 = 0{,}75_{10}$
    - Colocar na fórmula acima
    - $(-1)^1 \cdot (1+0{,}75) \cdot 2^{(124-127)} = -1{,}75 \cdot 2^{-3} = -0{,}21875$
  - Converter: 639,6875
    - Normalizar o número: $1001111111{,}1011_2 = 1{,}0011111111011 \times 2^9$
    - $s = 0$, $e = 9 + 127 = 136 = 10001000_2$, $f = 00111111110110000000000$

- **Parâmetros do Formato IEEE 754**

	float	double
Bits de precisão (mantissa)	23+1	52+1
Precisão decimal (dígitos decimais)	6,5	14,5
Bits do expoente	8	11
Expoente máximo	127	1023
Expoente mínimo	-126	-1022
Deslocamento do expoente	127	1023
Maior/menor número	$10^{\pm 38}$	$10^{\pm 308}$

- **Valores reservados**
  - Todos os bits do expoente =0
    - Todos os bits da mantissa = 0, então temos o valor ZERO. Pode ter sinal positivo ou negativo.
    - Algum bit da mantissa ≠ 0, então temos um número não normalizado
  - Todos os bits do expoente =1
    - Todos os bits da mantissa = 0, valor reservado para **infinity**. Permite um soft overflow. Também ocorre na divisão por zero.
    - Algum bit da mantissa ≠ 0, valor reservado para **NaN** (Not a Number), resultado de uma operação aritmética em que ao menos um operando é **infinity**.

- **Números denormalizados**
  - Números com expoente menor que o mínimo (denormalizados) possibilitam underflow gradual.
  - O tempo para efetuar operações aritméticas com números denormalizados é significativamente maior do que para números normalizados.

<table>
<tr><th>e</th><th>E
(expoente real)</th><th>f</th><th>Valor</th></tr>
<tr><td rowspan="2">0000 0000</td><td rowspan="2">Reservado</td><td>000...0</td><td>0_{10}</td></tr>
<tr><td>xxx...x</td><td>Denormalizado
$(-1)^s \times 2^{-126} \times (0.f)$</td></tr>
<tr><td>0000 0001</td><td>-126_{10}</td><td rowspan="6">xxx...x</td><td rowspan="6">Normalizado
$(-1)^s \times 2^{(e - \Delta)} \times (1.f)$
Δ = 127 (float)
Δ = 1023 (double)</td></tr>
<tr><td>0000 0010</td><td>-125_{10}</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>0111 1111</td><td>0_{10}</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>1111 1110</td><td>127_{10}</td></tr>
<tr><td rowspan="2">1111 1111</td><td rowspan="2">Reservado</td><td>000...0</td><td>Infinito (∞)</td></tr>
<tr><td>xxx...x</td><td>NaN</td></tr>
</table>

A ordem das operações pode afetar a acurácia do resultado

- Problemas com operações sucessivas que vão acumulando os erros (exemplo para $SPF(\beta, t, m, M) = SPF(10{,}3, -9{,}9)$)
  - Ex: $x = 1{,}23 \times 10^{3}$, $y = 1{,}00 \times 10^{0}$

`z = 0;`
`for (int i=0; i<10; ++i)`
`   z += y`
`z += x`
	`z = x;`
	`for (int i=0; i<10; ++i)`
	`  z += y`

  - Observar que a ordem das somas altera o resultado.

Sempre que subtrair números com mesmo sinal ou adicionar números com sinais distintos, a acurácia do resultado pode ser menor do que a precisão do formato de ponto flutuante.

- $1.23 \times 10^{0} - 1.22 \times 10^{0} = 0.01 \times 10^{0}$. Apesar de ser matematicamente equivalente a 1.00×10^{-2} , este último resultado dá a entender que a precisão é de dois dígitos, o que não é verdade.

Quando efetuar uma cadeia de cálculos envolvendo adições, subtrações, multiplicações e divisões, procure efetuar as multiplicações e divisões primeiro.

- Multiplicações e divisões não sofrem dos mesmos problemas que adições e subtrações, pois as mantissas são multiplicadas/divididas e os expoentes somados/subtraídos. Entretanto, eles ampliam os erros na proporção de seus multiplicandos/dividendos.
- $x * (y + z) = x * y + x * z$

Quando multiplicar e dividir conjuntos de números, procure multiplicar números grandes com números pequenos; e dividir números com magnitudes semelhantes.

- Overflow e Underflow (multiplicação somando expoentes grandes, divisão subtraindo expoentes negativos)

Comparação de números em ponto Flutuante

- Nunca comparar números em ponto flutuante diretamente. Dadas as inacurácias dos cálculos, os bits menos significativos de dois números que deveriam ser iguais dificilmente o serão
  - `= if abs(x-y) <= error`
  - `≠ if abs(x-y) > error`
  - `< if (x-y) < error`
  - `≤ if (x-y) <= error`
  - `> if (x-y) > error`
  - `≥ if (x-y) >= error`
- Cuidado na hora de definir o valor de erro. Ele deve ser um pouco maior do que o maior erro esperado nos cálculos. E isto depende da ordem de grandeza dos números (expoentes).
- Usando erro proporcional aos números
  - `= if abs(x-y) <= abs(x+y)*erro`
- **EPSILON**: a diferença entre 1.0 e o menor valor maior que 1.0 representável.
  - ***float.h***: `FLT_EPSILON, DBL_EPSILON`
  - Pode ser utilizado para comparações entre números entre 1.0 e 2.0
  - Para saber se dois números são iguais, é preciso saber qual a menor diferença representável na ordem de grandeza daqueles números. Se os números tem ordens de grandeza diferentes, toma-se um epsilon proporcional à ordem do maior número.

```
int AlmostEqualRelative(float A, float B)
{
    // Calculate the difference.
    float diff = fabs(A - B);
    A = fabs(A);
    B = fabs(B);
    // Find the largest
    float largest = (B > A) ? B : A;

    if (diff <= largest * FLT_EPSILON)
        return 1;
    return 0;
}
```

- **ULP** (Units in the Last Place): Quantos números podem ser representados entre dois números quaisquer.
    - Se a representação inteira de dois floats de mesmo sinal é subtraída, então o valor absoluto do resultado é igual a um mais o número de floats representáveis entre eles.

Não há bala de prata. Você precisa decidir:

- Se você está comparando com zero, então comparações com Epsilon relativo ou ULP não ajudam. Você precisa de um valor absoluto de epsilon, por exemplo um pequeno múltiplo de FLT_EPSILON. Basicamente você precisa decidir qual o tamanho do seu zero.
- Se você está comparando números não nulos, então epsilon relativo ou ULP podem funcionar. Um pequeno múltiplo de FLT_EPSILON ou um pequeno valor de ULPs.
- Se você precisa comparar números arbitrários que podem ser ou não nulos, então boa sorte!
    - Ex.: para números na ordem de 2^{24} , o valor de $Epsilon=2.0$, ou seja, o número que está a 1 ULP de distância é 2.0 unidade maior que o anterior.

Comentários finais:

1. Apenas 7 dígitos decimais são representáveis em **float** e em torno de 15 em **double**;
2. Toda vez que há conversão de decimal para binário e vice-versa pode haver perda de precisão;
3. Sempre use comparações seguras;
4. Cuidado com adições e subtrações que podem rapidamente erodir a verdadeira significância do resultado. O computador não conhece bits significativos;
5. Conversões entre tipos de dados (double, float, integer) podem ser imprecisas. Conversões para double não aumentam o número de bits significativos (inserem lixo). Conversões para inteiros truncam a mantissa para zero;
6. Arquiteturas diferentes podem apresentar resultados diferentes para operações aritméticas em ponto flutuante.
7. Dado um número em ponto flutuante, o próximo número que pode ser representado depende do expoente. Quanto maior o expoente, maior a diferença numérica entre dois números subsequentes.
8. A precisão de um **float** é menor que a de um **int32** a partir de 2^{23} . 16.777.215 é o maior **float** ímpar;
9. Para imprimir ponto flutuante:

```
printf("%1.8e\n", d); // float, always with an exponent
printf("%.9g\n", d); // float, shortest possible
printf("%1.16e\n", d); // double, always with an exponent
printf("%.17g\n", d); // double, shortest possible
```

Exercícios

1. Considere o programa a seguir e responda as questões.

```
#include <stdio.h>
#include <stdlib.h>

#define VALOR        0.6f
#define NUM_ELEMENTOS 10000

float somaSequencia( float *dados, unsigned int tam )
{
    float soma = 0.0;
    while ( tam-- )
    {
        soma += dados[tam];
    }

    return soma;
}

float somaPar( float *dados, unsigned int tam )
{
    if (tam == 2)
        return dados[0] + dados[1];

    if (tam == 1)
        return dados[0];

    unsigned int div = tam / 2;

    return somaPar(dados, div) + somaPar(dados+div, tam-div);
}

void main()
{
    // Preenche um vetor
    float *dados = (float*) malloc(NUM_ELEMENTOS * sizeof(float));

    for (unsigned int i = 0; i < NUM_ELEMENTOS; ++i)
        dados[i] = VALOR;

    float soma1 = somaSequencia( dados, NUM_ELEMENTOS );
    printf("Soma sequencia: %1.15f\n", soma1);

    float soma2 = somaPar( dados, NUM_ELEMENTOS );
    printf("Soma par: %1.15f\n", soma2);

    free (dados);
}
```

(a) Explique o que faz cada uma das funções acima;

(b) Execute o programa. Qual a razão da diferença nos resultados? Por que ”SomaPar” apresenta um resultado mais preciso do que “SomaSequencia”?

(c) Aumentando o valor de NUM_ELEMENTOS em 10, 100 e 1000 vezes, o que ocorre? Justifique.

(d) Pesquise sobre o “Algoritmo de Soma de Kahan”. Compreenda-o, implemente-o e compare os resultados.

2. Utilize a estrutura de dados apresentada abaixo e escreva um programa que lhe permita imprimir os números em ponto flutuante e seus componentes de forma separada;

```
#include <stdio.h>
#include <stdint.h>
#include <float.h>
#include <math.h>

typedef union
{
    int32_t i;
    float f;
    struct
    {   // Bitfields for exploration. Do not use in production code.
        uint32_t mantissa : 23;
        uint32_t exponent : 8;
        uint32_t sign : 1;
    } parts;
} Float_t;

void printFloat_t( Float_t num )
{
   printf("f:%1.9e, ix:0x%08X, s:%d, e:%d, mx:0x%06X\n",
            num.f, num.i,
            num.parts.sign, num.parts.exponent, num.parts.mantissa);
}

int main()
{
    printf("\nEpsilon: %1.15f\n", FLT_EPSILON);
}
```

Valor float	Valor Inteiro	Sinal	Expoente	Valor Expoente	Mantissa
0.0	0x00000000	0	0	-126	0
1.40129846e-45	0x00000001	0	0	-126	1
1.17549435e-38	0x00800000	0	1	-126	0
0.2	0x3E4CCCCD	0	124	-3	0x4CCCCD
1.0	0x3F800000	0	127	0	0
1.5	0x3FC00000	0	127	0	0x400000
1.75	0x3FE00000	0	127	0	0x600000
1.99999988	0x3FFFFFFF	0	127	0	0x7FFFFF
2.0	0x40000000	0	128	1	0
16777215	0x4B7FFFFF	0	150	23	0x7FFFFF
3.40282347e+38	0x7F7FFFFF	0	254	127	0x7FFFFF
Infinito positivo	0x7f800000	0	255	Infinito!	0

(a) Procure reproduzir a tabela apresentada acima. Verifique o que acontece com números próximos à medida em que seus valores aumentam;

(b) Verifique o que acontece com números muito próximos de zero, especialmente com números não normalizados;

3. Faça um programa que receba um número qualquer e calcule o valor de Epsilon relativo para aquele número;

4. Calcule o ULP entre dois números em ponto flutuante

5. Verifique se é viável e como pode-se utilizar Epsilon ou ULP relativo para:

(a) Comparar números de mesma ordem de grandeza;

(b) Comparar números de ordens de grandeza muito diferentes;

(c) Comparar números muito pequenos com zero (0.0);

6. Por que um inteiro com 32 bits (**int**) tem mais precisão do que um ponto flutuante com 32 bits (**float**) para valores entre 2^{24} e 2^{31} ?

7. Explique por que os valores de `soma1` e `soma2` no programa abaixo são diferentes;

```
#include <stdio.h>

int main()
{
   float soma1=0.0f, soma2=0.0f;

   for (int i=1; i<=200; ++i)
      soma1 += 1.0f / (i*i);

   for (int i=200; i>=1; --i)
      soma2 += 1.0f / (i*i);

   printf("Soma1: %.10g \t Soma2: %.10g\n\n", soma1, soma2);
   return 0;
}
```

8. Considere o trecho de código `C` abaixo para comparar dois números em ponto flutuante (**`float`**)

```
if (fabs(num1 - num2) < FLT_EPSILON)
```

Confirme ou conteste as afirmações abaixo, **justificando-as** e **corrigindo-as**, conforme o caso:

(a) A comparação funciona para valores de **`num1`** e **`num2`** da ordem de 2^0 ;

(b) A comparação será sempre falsa para valores de **`num1`** e **`num2`** da ordem de 2^{23} ;

(c) A comparação funciona para valores de `num1` e `num2` da ordem de 2^{-23} .

9. Considere um equipamento cujo sistema de ponto flutuante **normalizado** de **base 2**, possui **4 dígitos na mantissa**, **menor expoente -1** e **maior expoente 2**. Para este sistema:

(a) Qual o menor número positivo exatamente representável, em base 2?

(b) Qual o próximo positivo, depois do menor positivo representável, em base 2?

(c) Transforme o menor positivo e o próximo para a base decimal.

(d) Verifique se existem números reais entre o menor e o próximo positivo. Comente as implicações de sua verificação.

10. Considere o código a seguir:

```
1  // p, n, x: idem à função anterior
2  // px: valor do polinomio no ponto x
3  // dpx: valor da primeira derivada do polinomio no ponto x
4  void calculaPolinomioEDerivada( double *p, int n,
5                                  double x, double *px, double *dpx )
6  {
7  }
8
9  // p: coeficientes de um polinomio
10 // n: grau do polinomio p
11 // x: valor inicial e resposta
12 // erroMax: maior erro aceitavel
13 int funcaoFazAlgo(double *p, int n, double *x, double erroMax )
14 {
15    double px, dpx, erro, x_new;
16    do {
17      calculaPolinomioEDerivada(p, n, *x, &px, &dpx);
18      if (dpx == 0.0)
19         return -1;
20      x_new = *x - px / dpx;
21      erro = fabs( x_new - *x );
22      *x = x_new;
23   } while (erro > erroMax);
24   return 0;
25 }
```

Considerando o código acima, a aritmética em ponto flutuante e o padrão IEEE 754 responda:

(a) O que faz a função "`funcaoFazAlgo`"?

(b) Qual o comportamento da função "`funcaoFazAlgo`" caso o método não convirja? Proponha uma solução melhor para este caso. Você não precisa reescrever todo o código, basta indicar a linha a partir do qual seu código deve ser inserido.

(c) Qual o problema numérico do código na linha 18? Reescreva esta linha de forma a eliminar o problema.

(d) Por que um inteiro com 32 bits (**int**) tem mais precisão do que um ponto flutuante com 32 bits (**float**) para valores entre 2^{24} e 2^{31} .

(e) Identifique uma linha de código na qual pode surgir o valor ±**inf** (±∞)? Justifique sua resposta.