

Representações Numéricas no Computador

Fundamentos Computacionais para Cálculo Numérico

Prof. Dr. Raphael Teixeira

2026-10-29

Table of contents

1	Introdução: O abismo entre matemática e computação	2
1.1	Por que isso importa para Cálculo Numérico	2
2	Sistemas de Numeração Posicional	2
2.1	Base decimal (base 10)	2
2.2	Base binária (base 2)	3
2.3	Conversão decimal para binário	3
3	Inteiros em Máquina: Aritmética de Ponto Fixo	4
3.1	Inteiros sem sinal (unsigned integers)	4
3.2	Inteiros com sinal: Complemento de dois	4
3.3	Overflow inteiro	4
4	Números Reais em Máquina: Ponto Flutuante	5
4.1	Notação científica	5
4.2	Representação de ponto flutuante normalizada	5
5	Padrão IEEE 754	5
5.1	Formatos principais	5
5.2	Estrutura de um número double (64 bits)	6
5.3	Valores especiais em IEEE 754	6
5.4	Números desnormalizados (subnormal)	7
6	Precisão e Erros de Máquina	7
6.1	Machine epsilon (ϵ)	7
6.2	Maior e menor número representável	8
6.3	Dígitos significativos decimais	9
7	Arredondamento	9
7.1	Estratégias de arredondamento	9
7.2	Erro de arredondamento	10
8	Operações Aritméticas em Ponto Flutuante	10
8.1	Perda de significância (catastrophic cancellation)	10
8.2	Associatividade quebrada	11
8.3	Absorção numérica	11

9	Implementação em Computador	12
9.1	Verificação de machine epsilon em Python	12
9.2	Manipulação de representação binária	13
9.3	Verificação de overflow/underflow	14
10	Aplicações e Conexões com Cálculo Numérico	14
10.1	Por que isso importa	14
10.2	Próximos capítulos	15
11	Exercícios Propostos	15
12	Referências	16
13	Anexo: Código Python Completo para Experimentos	16

1 Introdução: O abismo entre matemática e computação

Na matemática, operamos com números reais em **precisão infinita**. O conjunto $\mathbb{R}$ é denso: entre dois reais quaisquer existe sempre outro real. Podemos trabalhar com π , e , ou $\sqrt{2}$ sem restrições.

No computador, tudo muda.

Um computador tem memória **finita**. Se cada número real precisasse de infinitos dígitos para representação exata, seria impossível armazená-lo. Logo, **todo número é uma aproximação**.

1.1 Por que isso importa para Cálculo Numérico

1. **Erros de arredondamento** — cada operação introduz pequenos erros
2. **Propagação de erros** — erros crescem ao longo de algoritmos
3. **Instabilidade numérica** — alguns algoritmos amplificam erros, outros os controlam
4. **Escolha de método** — um algoritmo “bom” matematicamente pode ser ruim em máquina

Exemplo clássico

Resolver $x^2 - 1000x + 1 = 0$ com a fórmula de Bhaskara pode resultar em **catastrófica perda de dígitos significativos** devido a cancelamento numérico.

2 Sistemas de Numeração Posicional

2.1 Base decimal (base 10)

Um número em base 10 é representado como:

$$N = d_n \cdot 10^n + d_{n-1} \cdot 10^{n-1} + \cdots + d_1 \cdot 10 + d_0 + d_{-1} \cdot 10^{-1} + \cdots$$

onde cada $d_i \in \{0, 1, \dots, 9\}$.

Exemplo: $1234.56 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10 + 4 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2}$

2.2 Base binária (base 2)

Um número em base 2 usa apenas dígitos $\{0, 1\}$ (bits):

$$N = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \dots + b_0 + b_{-1} \cdot 2^{-1} + b_{-2} \cdot 2^{-2} + \dots$$

onde cada $b_i \in \{0, 1\}$.

💡 Exemplo

$$\begin{aligned}(1101.101)_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 = 13.625 \text{ em decimal}\end{aligned}$$

2.3 Conversão decimal para binário

Parte inteira: divisões sucessivas por 2, tomando os restos.

Parte fracionária: multiplicações sucessivas por 2, tomando as partes inteiras.

💡 Exemplo: Converter 0.375

$$\begin{aligned}0.375 \times 2 &= 0.75 && (\text{dígito: } 0) \\ 0.75 \times 2 &= 1.5 && (\text{dígito: } 1) \\ 0.5 \times 2 &= 1.0 && (\text{dígito: } 1)\end{aligned}$$

$$\text{Logo, } (0.375)_{10} = (0.011)_2$$

⚠️ Atenção

Nem todo número decimal tem representação binária **finita**!

Por exemplo, $(0.1)_{10} = (0.0001100110011 \dots)_2$ é dízima periódica infinita.

Implicação: 0.1 não pode ser representado exatamente em computador!

```
# Demonstração: 0.1 não é exato em binário
x = 0.1
print(f"0.1 em Python (16 casas): {x:.16f}")
print(f"0.1 em Python (20 casas): {x:.20f}")

# Isso causa problemas
print(f"\n0.1 + 0.2 = {0.1 + 0.2:.20f}")
print(f"Esperado: 0.3")
```

0.1 em Python (16 casas): 0.10000000000000000

0.1 em Python (20 casas): 0.10000000000000000555

$0.1 + 0.2 = 0.30000000000000004441$

Esperado: 0.3

3 Inteiros em Máquina: Aritmética de Ponto Fixo

3.1 Inteiros sem sinal (unsigned integers)

Um inteiro sem sinal com n bits pode representar inteiros de 0 a $2^n - 1$.

Bits	Intervalo	Uso
8	$[0, 255]$	1 byte
16	$[0, 65535]$	2 bytes
32	$[0, 4.3 \times 10^9]$	4 bytes
64	$[0, 1.8 \times 10^{11}]$	8 bytes

3.2 Inteiros com sinal: Complemento de dois

Padrão moderno para representar negativos (sem ambiguidade).

Um inteiro com sinal em n bits é representado por:

$$N = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

onde o bit mais significativo (MSB) é o **bit de sinal**.

Intervalo: -2^{n-1} a $2^{n-1} - 1$

💡 Exemplo com 8 bits

- $(00000101)_2 = 5$
- $(11111011)_2 = -5$
- $(10000000)_2 = -128$ (menor valor)
- $(01111111)_2 = 127$ (maior valor)

Intervalo: $[-128, 127]$

3.3 Overflow inteiro

Se o resultado de uma operação inteira exceder o intervalo representável, ocorre **overflow**.

⚠ Comportamentos diferentes por linguagem

- **C/Fortran:** comportamento indefinido ou truncamento silencioso
- **Python:** inteiros crescem dinamicamente (sem overflow)
- **Java:** wraparound (volta ao intervalo válido)

Cuidado: Em linguagens compiladas, overflow silencioso é perigoso!

4 Números Reais em Máquina: Ponto Flutuante

4.1 Notação científica

Em matemática, qualquer número real pode ser escrito em notação científica:

$$x = \pm d_0.d_1d_2d_3 \dots \times 10^e$$

onde $d_0 \in \{1, 2, \dots, 9\}$ (um dígito antes do ponto) e e é o expoente.

Exemplo: $1234.56 = 1.23456 \times 10^3$

4.2 Representação de ponto flutuante normalizada

Em base 2, um número de ponto flutuante tem a forma:

$$x = \pm(1.d_1d_2d_3 \dots) \times 2^e$$

onde: - O bit implícito na posição 2^0 é sempre 1 (em representação normalizada) - $d_i \in \{0, 1\}$ são os dígitos da mantissa (ou fração) - e é o expoente

Componentes: 1. **Sinal:** 1 bit (0 = positivo, 1 = negativo) 2. **Expoente:** e bits (codificado com bias) 3. **Mantissa/Fração:** p bits (precisão)

Precisão total: $p + 1$ dígitos binários (o “+1” é o bit implícito)

5 Padrão IEEE 754

IEEE 754 (IEEE Standard for Floating-Point Arithmetic) é o padrão internacional para aritmética de ponto flutuante. Garante consistência entre plataformas.

5.1 Formatos principais

💡 Exemplos de operações especiais

```
print(1/0)          # inf
print(-1/0)         # -inf
print(0/0)          # nan
print(float('inf') - float('inf')) # nan
```

5.4 Números desnormalizados (subnormal)

Quando o expoente atinge seu mínimo ($E = 0$), números menores que o “mínimo normal” podem ser representados com precisão reduzida:

$$x = \pm(0.d_1d_2\dots) \times 2^{e_{\min}}$$

Isto fornece **underflow gradual** ao invés de queda abrupta para zero.

6 Precisão e Erros de Máquina

6.1 Machine epsilon (ϵ)

Definição: Menor número ϵ tal que $1 + \epsilon \neq 1$ em aritmética de ponto flutuante.

Fórmula:

$$\epsilon = 2^{-p}$$

onde p é o número de dígitos binários de precisão (após o bit implícito).

Formato	p	Machine epsilon	Valor aproximado
float32	24	2^{-23}	1.19×10^{-7}
float64	53	2^{-52}	2.22×10^{-16}

⚠️ Interpretação

Machine epsilon é uma medida da **precisão relativa**: em máquina, qualquer número é representado com erro relativo no máximo da ordem de $\epsilon/2$.

```
# Verificar machine epsilon em Python
import numpy as np

eps32 = np.finfo(np.float32).eps
eps64 = np.finfo(np.float64).eps

print(f"Machine epsilon (float32): {eps32}")
```

```

print(f"Machine epsilon (float64): {eps64}")

print(f"\nTeórico (2-23): {2**(-23)}")
print(f"Teórico (2-52): {2**(-52)}")

# Verificar experimentalmente
def find_epsilon(dtype):
    eps = dtype(1.0)
    while dtype(1.0) + eps != dtype(1.0):
        eps = eps / dtype(2.0)
    return eps * 2 # Volta um passo

print(f"\nExperimental float64: {find_epsilon(np.float64)}")

```

Machine epsilon (float32): 1.1920928955078125e-07
Machine epsilon (float64): 2.220446049250313e-16

Teórico (2⁻²³): 1.1920928955078125e-07
Teórico (2⁻⁵²): 2.220446049250313e-16

Experimental float64: 2.220446049250313e-16

6.2 Maior e menor número representável

Formato	Menor positivo normal	Maior
float32	$\approx 1.18 \times 10^{-38}$	$\approx 3.40 \times 10^{38}$
float64	$\approx 2.23 \times 10^{-308}$	$\approx 1.80 \times 10^{308}$

```

# Verificar limites em Python
info32 = np.finfo(np.float32)
info64 = np.finfo(np.float64)

print("Float32:")
print(f"  Menor: {info32.tiny}")
print(f"  Maior: {info32.max}")

print("\nFloat64:")
print(f"  Menor: {info64.tiny}")
print(f"  Maior: {info64.max}")

```

Float32:
Menor: 1.1754943508222875e-38
Maior: 3.4028234663852886e+38

Float64:
Menor: 2.2250738585072014e-308
Maior: 1.7976931348623157e+308

6.3 Dígitos significativos decimais

Número de dígitos decimais significativos:

$$n.^{\circ} \text{ de dígitos decimais} \approx p \times \log_{10}(2)$$

- **float32** ($p = 24$): $\approx 24 \times 0.301 \approx 7$ dígitos
- **float64** ($p = 53$): $\approx 53 \times 0.301 \approx 16$ dígitos

```
# Demonstração
x_float32 = np.float32(1.23456789)
x_float64 = np.float64(1.23456789)

print(f"Entrada: 1.23456789")
print(f"float32: {x_float32:.8f}")
print(f"float64: {x_float64:.16f}")

# Valores maiores
x_large32 = np.float32(123456789.0)
x_large64 = np.float64(123456789.0)

print(f"\nEntrada: 123456789.0")
print(f"float32: {x_large32:.1f}")
print(f"float64: {x_large64:.10f}")
```

```
Entrada: 1.23456789
float32: 1.23456788
float64: 1.2345678899999999
```

```
Entrada: 123456789.0
float32: 123456792.0
float64: 123456789.0000000000
```

7 Arredondamento

7.1 Estratégias de arredondamento

Quando um número não cabe exatamente em p bits, deve-se arredondar. IEEE 754 suporta 4 modos:

1. **Round to nearest (padrão)**: arredonda ao número flutuante mais próximo; em caso de empate, arredonda ao par
2. **Round toward zero (truncation)**: descarta os dígitos fora de p bits
3. **Round toward +infinity**: sempre arredonda para cima
4. **Round toward -infinity**: sempre arredonda para baixo

7.2 Erro de arredondamento

Se x é um número real e $\text{fl}(x)$ é sua representação em máquina, o erro relativo é limitado por:

$$\frac{|x - \text{fl}(x)|}{|x|} \leq \frac{\epsilon}{2}$$

onde ϵ é a machine epsilon.

8 Operações Aritméticas em Ponto Flutuante

8.1 Perda de significância (catastrophic cancellation)

Um problema grave ocorre quando **subtraímos dois números muito próximos**.

⚠ Exemplo clássico

Calcular $\sqrt{x^2 + 1} - x$ para $x = 10^8$.

Matematicamente, para x grande:

$$\sqrt{x^2 + 1} - x \approx \frac{1}{2x}$$

Em máquina, $\sqrt{x^2 + 1} \approx x$ (pois x é muito grande), então:

$$\sqrt{x^2 + 1} - x \approx 0 \quad \text{resultado errado!}$$

Solução: Racionalizar

$$\sqrt{x^2 + 1} - x = \frac{(x^2 + 1) - x^2}{\sqrt{x^2 + 1} + x} = \frac{1}{\sqrt{x^2 + 1} + x}$$

que é numericamente estável.

```
import math

x = 1e8

# Método instável
result1 = math.sqrt(x**2 + 1) - x
print(f"Instável: sqrt(x^2 + 1) - x = {result1}")

# Método estável
result2 = 1 / (math.sqrt(x**2 + 1) + x)
print(f"Estável: 1/(sqrt(x^2 + 1) + x) = {result2}")

# Comparação
```

```
print(f"\nValor teórico: 1/(2x) = {1/(2*x)}")
print(f"Erro instável: {abs(result1 - 1/(2*x))}")
print(f"Erro estável: {abs(result2 - 1/(2*x))}")
```

Instável: $\sqrt{x^2 + 1} - x = 0.0$
 Estável: $1/(\sqrt{x^2 + 1} + x) = 5e-09$

Valor teórico: $1/(2x) = 5e-09$
 Erro instável: $5e-09$
 Erro estável: 0.0

8.2 Associatividade quebrada

Operações não são associativas em ponto flutuante:

$$(a + b) + c \neq a + (b + c)$$

```
# Demonstração
a = 1.0
b = 1e-16
c = -1e-16

result1 = (a + b) + c
result2 = a + (b + c)

print(f"(a + b) + c = {result1}")
print(f"a + (b + c) = {result2}")
print(f"Iguais? {result1 == result2}")

# Outro exemplo
x = [1e10, 1, -1e10]
print(f"\nSoma esquerda: ((1e10 + 1) - 1e10) = {(x[0] + x[1]) + x[2]}")
print(f"Soma direita: (1e10 + (1 - 1e10)) = {x[0] + (x[1] + x[2])}")
```

$(a + b) + c = 0.9999999999999999$
 $a + (b + c) = 1.0$
 Iguais? False

Soma esquerda: $((1e10 + 1) - 1e10) = 1.0$
 Soma direita: $(1e10 + (1 - 1e10)) = 1.0$

8.3 Absorção numérica

Se $|a| \gg |b|$, então $a + b \approx a$ em máquina (o b é “absorvido”).

```
# Demonstração
a = 1e16
b = 1

result = a + b - a
print(f"(1e16 + 1) - 1e16 = {result}")
print(f"Esperado: 1")
print(f"Obtido: {result}")

print(f"\nComparação:")
print(f"1e16 + 1 = {a + b}")
print(f"1e16      = {a}")
print(f"São iguais? {a + b == a}")
```

(1e16 + 1) - 1e16 = 0.0

Esperado: 1

Obtido: 0.0

Comparação:

1e16 + 1 = 1e+16

1e16 = 1e+16

São iguais? True

9 Implementação em Computador

9.1 Verificação de machine epsilon em Python

```
import numpy as np

# Método 1: usando numpy
eps64 = np.finfo(np.float64).eps
print(f"NumPy epsilon (float64): {eps64}")

# Método 2: encontrar experimentalmente
def find_machine_epsilon():
    epsilon = 1.0
    while 1.0 + epsilon != 1.0:
        epsilon = epsilon / 2.0
    return epsilon * 2

eps_experimental = find_machine_epsilon()
print(f"Experimental: {eps_experimental}")

# Método 3: usando sys
```

```
import sys
print(f"sys.float_info.epsilon: {sys.float_info.epsilon}")
```

```
NumPy epsilon (float64): 2.220446049250313e-16
Experimental:           2.220446049250313e-16
sys.float_info.epsilon: 2.220446049250313e-16
```

9.2 Manipulação de representação binária

```
def float_to_binary(f):
    """Converte float64 para representação binária"""
    bits = struct.pack('>d', f)
    return ''.join(f'{b:08b}' for b in bits)

def analyze_float(x):
    """Analisa componentes de um float64"""
    bits = float_to_binary(x)
    sign = bits[0]
    exponent = bits[1:12]
    mantissa = bits[12:]

    exp_value = int(exponent, 2)
    exp_real = exp_value - 1023

    print(f"Número: {x}")
    print(f"Binário: {bits}")
    print(f"  Sinal:      {sign} ({'+' if sign == '0' else '-'})")
    print(f"  Expoente: {exponent} ({exp_value} → real: {exp_real})")
    print(f"  Mantissa: {mantissa[:16]}... ({mantissa})")
    print()

# Exemplos
analyze_float(1.0)
analyze_float(0.5)
analyze_float(0.1)
```

Número: 1.0
Binário: 0011111111110000000000000000000000000000000000000000000000000000
Sinal: 0 (+)
Expoente: 0111111111 (1023 → real: 0)
Mantissa: 000000000000000... (0000000000000000000000000000000000000000000000000)

Número: 0.5
Binário: 0011111111110000000000000000000000000000000000000000000000000000
Sinal: 0 (+)
Expoente: 01111111110 (1022 → real: -1)
Mantissa: 000000000000000... (0000000000000000000000000000000000000000000000000)

Número: 0.1
Binário: 0011111110111001100110011001100110011001100110011010
Sinal: 0 (+)
Expoente: 0111111011 (1019 → real: -4)
Mantissa: 1001100110011001... (100110011001100110011001100110011001100110011010)

9.3 Verificação de overflow/underflow

```
import numpy as np

# Limites
info = np.finfo(np.float64)
print("Float64 limits:")
print(f"  Epsilon:           {info.eps}")
print(f"  Tiny (min normal):    {info.tiny}")
print(f"  Max:                   {info.max}")

# Gerar overflow
x = np.float64(1e308)
print(f"\n1e308 * 10 = {x * 10}") # inf

# Underflow gradual
y = np.float64(1e-320)
print(f"1e-320 = {y}")
print(f"1e-330 = {np.float64(1e-330)}") # Desnormalizado
```

Float64 limits:

Epsilon:	2.220446049250313e-16
Tiny (min normal):	2.2250738585072014e-308
Max:	1.7976931348623157e+308

1e308 * 10 = inf
1e-320 = 1e-320
1e-330 = 0.0

```
/tmp/ipykernel_10094/740743619.py:12: RuntimeWarning: overflow encountered in scalar multiply
print(f"\n1e308 * 10 = {x * 10}") # inf
```

10 Aplicações e Conexões com Cálculo Numérico

10.1 Por que isso importa

1. **Propagação de erros:** Pequenos erros de arredondamento crescem ao longo de algoritmos iterativos

2. **Escolha de algoritmo:** Métodos numericamente estáveis são essenciais
3. **Análise de erro:** Devemos estimar como erros se propagam em nossas contas
4. **Teste de convergência:** Saber ϵ ajuda a definir tolerâncias apropriadas

10.2 Próximos capítulos

- **Cap 1: Propagação de Erros** — Como estimar e controlar erros
 - **Cap 2: Sistemas Lineares** — Resolução estável via métodos diretos e iterativos
 - **Cap 3: Interpolação** — Aproximação de funções e análise de erro
 - **Cap 4: Integração Numérica** — Quadraturas e fórmulas compostas
-

11 Exercícios Propostos

1. Conversão de bases

- Converter $(101.11)_2$ para decimal
- Converter $(0.1)_{10}$ para binário (até 16 dígitos)

2. Machine epsilon

- Qual é o machine epsilon para float32 e float64?
- Implementar um algoritmo em Python que encontre experimentalmente

3. Limites numéricos

- Em double precision IEEE 754, qual é o maior expoente?
- Qual é o valor máximo representável?

4. Não-associatividade

- Verificar numericamente que $(a + b) + c \neq a + (b + c)$ com valores apropriados

5. Cancelamento numérico

- Implementar ambas as formulações de $\sqrt{x^2 + 1} - x$ em Python
- Comparar resultados para $x = 10^{10}$

6. Absorção numérica

- Verificar que $(10^{16} + 1) - 10^{16} = 0$ em float64
 - Explorar diferentes valores de 10^k para encontrar threshold
-

12 Referências

- Burden, R. L., & Faires, J. D. (2016). *Numerical analysis* (10th ed.). Cengage Learning.
 - Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 5–48.
 - Higham, N. J. (2002). *Accuracy and stability of numerical algorithms* (2nd ed.). SIAM.
 - IEEE. (2019). IEEE 754-2019: IEEE standard for floating-point arithmetic. Institute of Electrical and Electronics Engineers.
 - Knuth, D. E. (1997). *The art of computer programming: Seminumerical algorithms* (Vol. 2, 3rd ed.). Addison-Wesley.
-

13 Anexo: Código Python Completo para Experimentos

```
# Código para experimentos com representação numérica

import numpy as np
import struct
import sys

class FloatAnalyzer:
    """Classe para analisar representações de ponto flutuante"""

    @staticmethod
    def to_binary(x):
        """Converte float64 para binário"""
        bits = struct.pack('>d', x)
        return ''.join(f'{b:08b}' for b in bits)

    @staticmethod
    def from_binary(bits_str):
        """Converte binário para float64"""
        bytes_val = bytes(int(bits_str[i:i+8], 2)
                           for i in range(0, 64, 8))
        return struct.unpack('>d', bytes_val)[0]

    @staticmethod
    def analyze(x):
        """Analisa componentes de float64"""
        bits = FloatAnalyzer.to_binary(x)
        sign_bit = bits[0]
        exp_bits = bits[1:12]
        mant_bits = bits[12:]
```

```

exp_val = int(exp_bits, 2)
exp_real = exp_val - 1023

return {
    'valor': x,
    'binario': bits,
    'sinal': sign_bit,
    'expoente_bits': exp_bits,
    'expoente_valor': exp_val,
    'expoente_real': exp_real,
    'mantissa': mant_bits
}

# Exemplos de uso
print("=== Análise de Números em IEEE 754 ===\n")

for num in [1.0, 0.5, 2.0, 0.1]:
    result = FloatAnalyzer.analyze(num)
    print(f"Número: {num}")
    print(f"  Expoente real: {result['expoente_real']}")
    print(f"  Binário: {result['binario'][:16]}...")
    print()

```

=== Análise de Números em IEEE 754 ===

Número: 1.0
 Expoente real: 0
 Binário: 001111111110000...

Número: 0.5
 Expoente real: -1
 Binário: 0011111111100000...

Número: 2.0
 Expoente real: 1
 Binário: 0100000000000000...

Número: 0.1
 Expoente real: -4
 Binário: 0011111110111001...

Última atualização: 2024