CAPÍTULO

1

Aritmética de Ponto Flutuante e Tratamento de Erros

1.1 – Introdução

Neste capítulo, abordaremos formas de representar números reais em computadores. Iniciamos com uma discussão sobre a representação de números com quantidade finita de dígitos, mais especificamente, as representações de números inteiros, ponto fixo e ponto flutuante em computadores. A representação de números e a aritmética em computadores levam aos chamados erros de arredondamento e de truncamento, que são vitais para se avaliar a eficiência de algoritmos numéricos em qualquer tipo de problema.

Finalizamos com uma discussão sobre mensuração de erros numéricos, com destaque para o caso de ocorrência do chamado cancelamento catastrófico.

Alguns trechos de código serão colocados para exemplificar certos conceitos que trataremos. Sugiro que você abra um editor de código, de preferência um *Jupyter Notebook*, para ir executando e verificando as saídas de forma interativa. Isso ajudará ainda mais no entendimento dos conceitos trabalhados.

Quase todos esses trechos usam funções da biblioteca `Numpy`, e, portanto, devemos, antes de tudo, carregá-la.

```
In [ ]: import numpy as np
```

1.2 – Representação de números

Usualmente, utilizamos o sistema de numeração decimal para representar números. Esse é um sistema de numeração posicional onde a posição do dígito indica a potência de 10 que o dígito representa.

Exemplo 2.1

O número 293 pode ser decomposto como

$$\begin{aligned} 293 &= 2 \text{ centenas} + 9 \text{ dezenas} + 3 \text{ unidades} \\ &= 2 \cdot 10^2 + 9 \cdot 10^1 + 3 \cdot 10^0. \end{aligned}$$

Além do sistema decimal, outros sistemas que com certeza vocês já viram em seus estudos, como o binário, também são posicionais, somente usando uma base diferente (decimal, base 10, binário, base 2).

Exemplo 2.2

$$\begin{aligned} x &= (1001, 101)_2 \\ &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 8 + 0 + 0 + 1 + 0,5 + 0 + 0,125 = 9,625. \end{aligned}$$

Ou seja, $(1001, 101)_2$ é igual a $9,625$ no sistema decimal.

De uma forma geral, podemos definir um sistema de numeração posicional de base b usando a seguinte definição:

Definição 1.1. *Dado um número natural $b > 1$ e o conjunto de símbolos $\{\pm, 0, 1, 2, \ldots, b-1\}$, um número x qualquer representado pela sequência*

$$x = \pm\,(d_n\ d_{n-1}\ \cdots\ d_1\ d_0\ d_{-1}\ d_{-2}\ \cdots)_b \tag{1.1}$$

pode ser reescrito no formato

$$d_n \cdot b^n + d_{n-1} \cdot b^{n-1} + \cdots + d_0 \cdot b^0 + d_{-1} \cdot b^{-1} + d_{-2} \cdot b^{-2} + \cdots \tag{1.2}$$

Alternativamente, é costumeiro usarmos uma notação que possibilita uma representação resumida de números reais, chamada de *notação científica*, definida como

Definição 1.2. *Um número x na base b é dito estar representado em notação científica quando está escrito na forma*

$$x = (-1)^s (M)_b \times b^E, \tag{1.3}$$

onde $(M)_b = (d_0, d_{-1}d_{-2}d_{-3}\cdots)_b$, com $d_0 \neq 0$, é chamada de mantissa, o sinal s é 0 para positivo e 1 para negativo, e o valor E é chamado de expoente do número.

Nessa definição, no caso de $x = 0$, a mantissa assume o valor $M_b = (0,00\cdots)_b$.

Exemplo 2.3

O número $x = 50000$, considerando uma base decimal, pode ser representado em notação decimal como $x = 5 \cdot 10^4$. Já o número $y = 0,325$ é representado como $y = 3,25 \cdot 10^{-1}$.

Uma maneira simples de verificar a representação em notação científica de um número decimal usando Python é usando uma das opções de formatação do método `print()` da biblioteca padrão da linguagem.

Para o exemplo anterior, basta executar os comandos a seguir.

```
In [ ]: print("%1.5e" % 50000)
        print("%1.5e" % 0.325)
```

```
5.00000e+04
3.25000e-01
```

Uma maneira mais elegante de fazer essa representação, sem usar o método `print()`, é usar o método `format_float_scientific` da biblioteca `numpy`, cuja documentação pode ser vista nesse link.

Repetindo os exemplos acima, podemos fazer

```
In [ ]: np.format_float_scientific(50000)
```

```
Out[ ]: '5.e+04'
```

```
In [ ]: np.format_float_scientific(0.325)
```

```
Out[ ]: '3.25e-01'
```

Dos exemplos apresentados, três coisas merecem destaque. A primeira é com relação a saída, que apresenta a letra `e` para representação de notação científica, que, nesse caso, representa a base 10 da notação para valores decimais. Dessa forma `e+04` equivale a 10^4 e `e-01` equivale a 10^{-1}.

A segunda é com relação a saída do método `format_float_scientific`, que sempre será uma *string*. Isso porque a representação por notação científica, para um computador, basicamente é uma forma de visualizar um valor.

A terceira, por fim, é sobre a utilidade dessa representação para visualização de número com diferentes ordens de grandeza. A notação científica facilita a comparação direta entre ordens de grandeza diferentes, facilitando, inclusive, operações entre elas.

1.3 – Representação de números reais em computadores

A representação numérica usando notação científica permite representar absolutamente qualquer valor, independente de quão grande ou pequeno ele seja, num forma compacto e fácil de ser computado.

Essas duas características, por si só, favorecem a representação de números reais em computadores. No entanto, computadores tem um problema intrínseco a suas construções: *capacidade finita de memória*. Essa limitação faz com que a quantidade de memória alocada para representação também seja finita, o que, por sua vez, limita a quantidade de números que um computador pode representar.

Sim, é impossível para qualquer computador já construído representar completamente qualquer intervalo de números reais. Na prática, podemos, inclusive, afirmar que **não existem números reais em um computador.**

Computadores estão limitados a trabalhar com números racionais, que são frações entre números inteiros para poder representar *aproximações* de números reais.

Podemos ter uma visão disso usando o método `as_integer_ratio()`, da biblioteca padrão, que retorna um par de inteiros cuja proporção é exatamente igual ao `float` original e com um denominador positivo. O método retorna uma tupla com dois valores, o primeiro referente ao numerador e o segundo o denominador.

Vamos implementar uma pequena função e fazer alguns testes.

In [ ]:
```
def racionais(x):
    num,den = x.as_integer_ratio()
    return print(num,'/',den)
```

In [ ]:
```
racionais(10)
```

Out[ ]:
```
10 / 1
```

In [ ]:
```
racionais(3.14)
```

Out[ ]:
```
7070651414971679 / 2251799813685248
```

In [ ]:
```
racionais(3.141592)
```

Out[ ]:
```
3537118140137533 / 1125899906842624
```

Notemos que nos três testes que fizemos, com 10, 3.14 e 3.141592, esses dois últimos números aproximações de π, os resultados foram frações com numeradores e denominadores que, se divididos retornam o valor “real” representado.

Tendo em vista, então, essas limitações, *como computadores representam números reais (ou suas aproximações)?*. Basicamente, temos duas opções para essa resposta: sistemas de ponto fixo e sistemas de ponto flutuante.

Sistema de ponto fixo

No sistema de ponto fixo, a faixa de números que pode representar um determinado valor é fixa, ou seja, a *posição da vírgula é predeterminada*. Assim, todos os valores representados em ponto fixo para uma determinada operação possuem a mesma quantidade de algarismos inteiros e fracionários. A Figura **??** apresenta uma ilustração da representação de ponto fixo do número 10.5.

	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
0	1	0	1	0	1	0	0
sinal	*parte inteira*				*parte fracionária*		

Figura 1.1: Representação em ponto fixo do número 10.5.

Sistema de ponto flutuante

A representação em ponto fixo tem uma limitação complicada. É preciso garantir que o resultado de operações aritméticas esteja dentro da faixa de valores possíveis, caso contrário, os resultados obtidos serão incorretos. Esse problema é particularmente importante com operações de multiplicação ou divisão que podem gerar números muito maiores ou muito menores que as partes da operação.

Como resolver esse problema? A primeira possibilidade é aumentar a quantidade de espaço de memória para representar valores maiores (ou menores). Mas já sabemos que isso é somente postergar o problema.

A solução mais interessante nesse caso é a adoção de um sistema de ponto flutuante para representar números reais.

Na representação em ponto flutuante, cada número será representado por uma fração, chamada de mantissa ou significando, uma parte inteira, chamada de expoente e um bit no início para representação do sinal (já notou que você já viu esses nomes né?). A Figura **??** ilustra esse conceito.

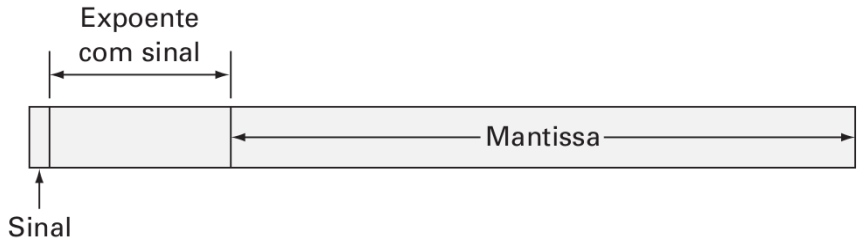

Figura 1.2: Representação gráfica de como um número em ponto flutuante é armazenado.

Mas porque essa representação é mais eficiente do que a anterior? Por que podemos trabalhar com a representação de números com uma faixa significativa de valores, usando uma menor quantidade de *bits* para isso. Vejamos o exemplo a seguir.

Exemplo 3.1

Para representar o número $x = 10000000$ em ponto fixo, precisaremos de, pelo menos 9 *bits*, considerando o do sinal, enquanto na representação em ponto flutuante, conseguimos representar o mesmo número $x = 10000000 = 1 \cdot 10^8$ por 5 bits: um do sinal, um da mantissa e três do expoente.

O ponto flutuante na máquina, será representado, obviamente, com uma base 2 e não 10. Dessa forma, expoente e mantissa mudam.

Uma forma de descobrirmos as partes de um número representado em ponto flutuante usando python é com o método `frexp()` da biblioteca *numpy*, cuja documentação está disponível no link. O método pode receber um escalar ou um `array` e retornará uma tupla com dois valores, sendo o primeiro referente a mantissa e o segundo, ao expoente para a base 2. Assim, para sabermos, por exemplo, como é representado o número 100 em ponto flutuante, podemos escrever o seguinte código

```
In [ ]: np.frexp(10000000)
Out[ ]: (0.5960464477539062, 24)
```

ou seja, $10000000 = 0,5960464477539062 \cdot 2^{24}$! De fato, podemos facilmente conferir isso

```
In [ ]: 0.5960464477539062*2**24
Out[ ]: 10000000.0
```

Então, por quê fazer $1 \cdot 10^8$ também retornará o mesmo resultado?

```
In [ ]: 1. * 10**8
Out[ ]: 10000000.0
```

Isso acontece porquê o interpretador python faz a tradução entre as bases, para que possa adaptar seus cálculos. Lembre-se: máquinas **sempre** operam com base 2!

O Padrão IEEE 754

Uma vez compreendendo que a representação em ponto flutuante é mais interessante, ainda temos o problema de estabelecer a quantidade de *bits* máxima que podemos usar para representar sinal mantissa e expoente. Esse problema tem solução num sistema padronizado pelo IEEE, aceito por praticamente todos os fabricantes de *hardware* e *software* chamado de padrão **IEEE 754/2008** (2008 é a última versão do padrão). Você pode encontrar o padrão na página do IEEE, na aba *IEEE standards* ou acessando o link. Um detalhe: é pago...

Esse padrão estabelece formatos aritméticos, formatos para intercâmbio entre diferentes fabricantes, algoritmos para arredondamento, operações algébricas com pontos flutuantes e manipulação de excessões. Os formatos binários adotados são definidos por uma terna [1](s, e, f), cujos valores estabelecem a quantidade de *bits* para cada campo de representação binária de ponto flutuante:

- s é referente ao sinal;
- e é referente ao expoente e
- f é referente à mantissa ou parte fracionária.

Assim, com relação aos tamanhos desses campos, o padrão estabelece três formatos:

- **single:** no qual se usam 32 *bits*, sendo um *bit* para sinal, 8 para expoente e 23 para a mantissa, ou seja, é representado pela terna (1,8,23);

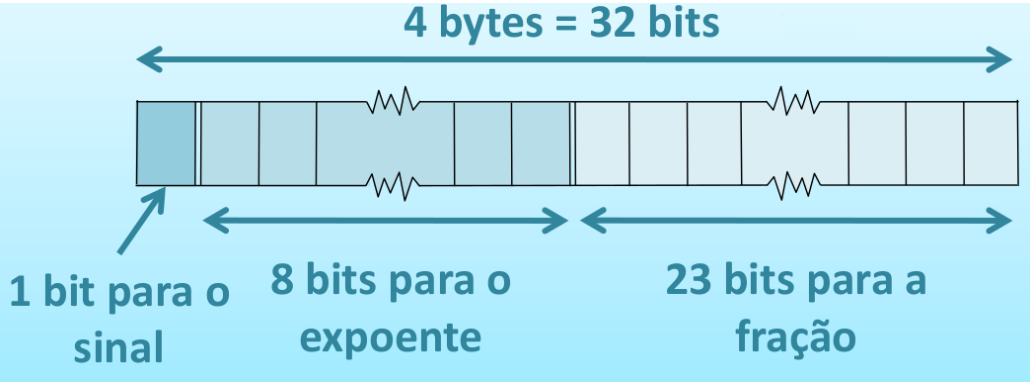

Figura 1.3: Representação gráfica do padrão *single* do IEEE 754.

- **double:** no qual se usam 64 *bits*, sendo um *bit* para sinal, 11 para expoente e 52 para a mantissa, ou seja, é representado pela terna (1,11,52)

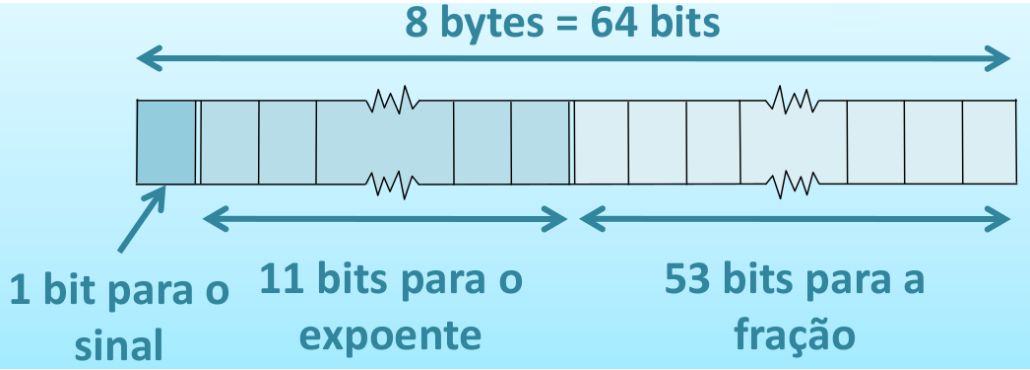

Figura 1.4: Representação gráfica do padrão *double* do IEEE 754.

[1] uma terna é uma tupla com 3 elementos

- **quad:** no qual se usam 128 *bits*, sendo um *bit* para sinal, 15 para expoente e 112 para a mantissa, ou seja, é representado pela terna (1,15,112).

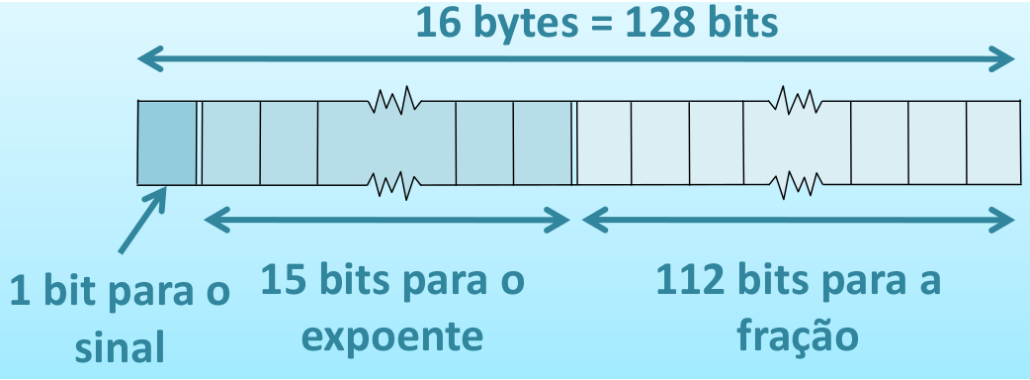

Figura 1.5: Representação gráfica do padrão *quad* do IEEE 754.

Alguns aspectos precisam ser refletidos com relação a essas representações. O primeiro deles é a existência de um intervalo limitado de quantidades que podem ser representadas. Exatamente como no caso da representação em ponto fixo, há números positivos e negativos grandes ou pequenos demais que não podem ser representados. Quando existem tentativas de usar números muito grandes, acima do intervalo aceitável, ocorre um erro chamado de *overflow*, como ilustrado na Figura **??**. Além disso, quando se tenta representar número pequenos demais, a representação não tem "precisão" suficiente para tal, incorrendo no erro chamado de *underflow*. Isso é ilustrado pelo "buraco" causado pelo *underflow* entre zero e o primeiro número positivo na Figura **??**.

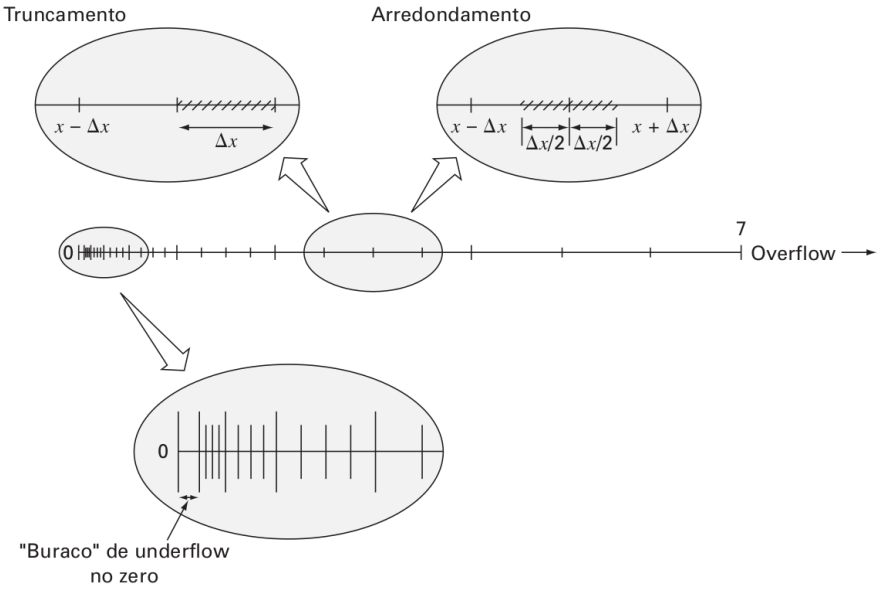

Figura 1.6: Representação da distribuição de números representáveis em um sistema de ponto flutuante. Cada valor é indicado por um traço. Apenas os números positivos são mostrados. Um conjunto idêntico também se estenderia na direção negativa.

Exemplo 3.2

Um exemplo de *overflow* pode ser considerado a partir de operações feitas com números significativamente grandes. Consideremos, por exemplo, o número $a = 10^{308}$.

```
In [ ]: a = 1e+308
```

Podemos ver que, se realizarmos a operação a seguir

```
In [ ]: a == a + (a - a)
Out[ ]: True
```

a saída será verdadeira, como era de se esperar. No entanto, se executarmos primeiro a operação de adição,

```
In [ ]: a == (a + a) - a
Out[ ]: False
```

o resultado não bate. Isso acontece porque ao somar duas vezes o número a, o valor resultante será maior que o limite superior da representação padrão de ponto flutuante que está sendo usada (que é `float32`, equivalente ao *double* do IEEE 754).

De fato, soma $a + a$ resulta em

```
In [ ]: a + a
Out[ ]: inf
```

em que `inf` é uma representação de qualquer valor acima do limite superior de representação numérica (falaremos mais dele em breve).

Exemplo 3.3

Um exemplo de *underflow* pode ser considerado a partir de operações feitas com números significativamente pequenos. Consideremos, por exemplo, o número $b = 2^{-1074}$.

```
In [ ]: b=2**(-1074)
        b
Out[ ]: 5e-324
```

Note que, como vimos antes, $b = 2^{-1074}$ equivale, na base 10, à $b = 10^{-324}$ (lembra do método `frexp`?).

Podemos ver que, se realizarmos as operações a seguir, obteremos resultados que nos mostram que esse valor é menor do que o mínimo valor que pode ser operado pela

máquina.

```
In [ ]: b == b/2 * 2
Out[ ]: False
In [ ]: b/2
Out[ ]: 0
```

O valor 0 resultante da última operação indica que o valor obtido da divisão $b/2$ é menor que o menor valor que a máquina pode considerar e, por isso, ela considera 0. Ou seja, tudo que estiver abaixo dessa precisão, será "arredondado" para 0.

Os dois exemplos acima, além de ilustrarem casos de ocorrência de *overflow* e *underflow* em operações com números limítrofes, nos levantam uma segunda discussão a ser colocada aqui: *existe apenas um número finito de quantidades que podem ser representadas dentro do intervalo*.

Talvez o caso de maior destaque para exemplificar essa afirmativa é o da representação de números irracionais. Obviamente, os números irracionais não podem ser representados exatamente. Além disso, os números racionais que não coincidem exatamente com muitos valores no conjunto também não podem ser representados precisamente. Os erros introduzidos pela aproximação em ambos os casos são chamados de erros de quantização. A aproximação propriamente dita é feita de uma das duas maneiras: truncando ou arredondando.

Por exemplo, suponha que o valor de $\pi = 3,14159265358\ldots$ deva ser armazenado em um sistema numérico na base 10 com sete algarismos significativos. Um método de aproximação seria simplesmente omitir o oitavo termo e os termos mais altos, como em $\pi = 3,141592$. Essa técnica de manter apenas os termos considerados significativos é chamada de *truncamento*. Observe que, para o sistema numérico ilustrado na Figura **??**, truncar significa que qualquer quantidade que caia dentro de um intervalo de comprimento Δx será armazenada como a quantidade na extremidade inicial do intervalo. Assim, o limitante superior do erro para o truncamento é Δx, isto é, o que passar desse intervalo é "cortado".

Uma outra forma de fazer essa aproximação é, após considerar qual o algarismo mais significativo a ser considerado, o que depende da quantidade de memória disponível, realizar um *arredondamento* do último dígito considerado. Por exemplo, na representação de $\pi = 3,141592$, o primeiro valor descartado é 6, de forma que o último dígito mantido deveria ser arredondado para cima para fornecer $3,141593$. Essa abordagem reduz consideravelmente o erro na aproximação.

De uma forma mais precisa, usando a notação da Definição 2, podemos dizer que, para alocar um número $x = \pm(d_0\ d_1\ d_2\ \ldots\ d_{n-1}\ d_n\ d_{n+1}\ \ldots) \cdot b^E$ usando somente n dígitos, podemos usar uma das estratégias:

- **truncamento:** ignoram-se os dígitos $d_n\ d_{n+1}\ \ d_{n+2}\ d_{n+3}\ \ldots$ e o valor aproximado de x se

torna

$$\tilde{x} = \pm(d_0\ d_1\ d_2\ \ldots\ d_{n-1}\ \ldots) \cdot b^E; \tag{1.4}$$

- **arredondamento:** segue-se a regra abaixo

$$\tilde{x} = \begin{cases} \pm(d_0\ d_1\ d_2\ \ldots\ d_{n-1}) \cdot b^E, & \text{quando } d_n < b/2 \\ \pm(d_0\ d_1\ d_2\ \ldots\ (d_{n-1} + b^{-1})) \cdot b^E, & \text{quando } d_n > b/2 \end{cases} \tag{1.5}$$

Existem diversos algoritmos de arredondamento, mas essa regra é a descrita no padrão IEEE 754 e, portanto, nos deteremos em descrever apenas ela. Para ver mais algumas regras, veja o link.

Também pode-se perceber que o intervalo entre os números, Δx, aumenta quando o módulo dos números cresce. É essa característica, claro, que permite que a representação em ponto flutuante preserve os algarismos significativos. Mas ela também significa que o erro de quantização será proporcional ao módulo do número que está sendo representado. Para números em ponto flutuante, essa proporcionalidade pode ser expressa, para os casos em que é empregado o truncamento, como

$$\frac{|\Delta x|}{|x|} \leq \varepsilon \tag{1.6}$$

e, para os casos nos quais o arredondamento é usado, como

$$\frac{|\Delta x|}{|x|} \leq \frac{\varepsilon}{2}. \tag{1.7}$$

O número ε é chamado de *épsilon de máquina*, e é definido como é definido de forma que $1+\varepsilon$ seja o menor número representável maior que 1, isto é, $1+\varepsilon$ é representável, mas não existem números representáveis no intervalo $(1, 1+\varepsilon)$. Pode-se calcular o épsilon da máquina usando a expressão

$$\varepsilon = b^{(1-t)}, \tag{1.8}$$

sendo, de acordo com a notação que adotamos, b a base numérica e t o número de algarismos significativos da mantissa.

Exemplo 3.4

Para o padrão *double* do IEEE 754, cuja mantissa tem 53 *bits* de capacidade de representação, temos que o épsilon será dado por

$$\varepsilon = 2^{(1-53)} = 2^{-52} \approx 2,22 \cdot 10^{-16}. \tag{1.9}$$

O Python dispõe de algumas ferramentas para conseguirmos informações sobre o épsilon da máquina.

A primeira forma é usando o método `float_info` da classe `sys` da biblioteca padrão da linguagem, cuja documentação pode ser vista nesse link. Esse método possui uma lista de possíveis atributos para retornar informações sobre precisão e representação interna, mostrados na Figura **??**.

attribute	float.h macro	explanation
epsilon	DBL_EPSILON	difference between 1.0 and the least value greater than 1.0 that is representable as a float
dig	DBL_DIG	maximum number of decimal digits that can be faithfully represented in a float; see below
mant_dig	DBL_MANT_DIG	float precision: the number of base-`radix` digits in the significand of a float
max	DBL_MAX	maximum representable positive finite float
max_exp	DBL_MAX_EXP	maximum integer *e* such that `radix**(e-1)` is a representable finite float
max_10_exp	DBL_MAX_10_EXP	maximum integer *e* such that `10**e` is in the range of representable finite floats
min	DBL_MIN	minimum representable positive *normalized* float
min_exp	DBL_MIN_EXP	minimum integer *e* such that `radix**(e-1)` is a normalized float
min_10_exp	DBL_MIN_10_EXP	minimum integer *e* such that `10**e` is a normalized float
radix	FLT_RADIX	radix of exponent representation
rounds	FLT_ROUNDS	integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.

Figura 1.7: Atributos do método `float_info` da classe `sys`.

Para visualizarmos o épsilon da máquina para a representação `float` padrão do Python (*double*), podemos fazer

```
In [ ]: sys.float_info.epsilon
Out[ ]: 2.220446049250313e-16
```

que é o valor que nós calculamos no exemplo **??** acima.

Uma outra forma de procurar essa informação é usando a classe `finfo`, da biblioteca `numpy`, cuja documentação pode ser encontrada no link e cuja lista de atributos é mostrada na Figura **??**.

Como a biblioteca `numpy` implementa um conjunto considerável de diferentes tipos de dados (veja a lista nesse link), podemos experimentar diferentes representações para visualizar o épsilon da máquina. Por exemplo, se fizermos

Attributes: **bits** : *int*
The number of bits occupied by the type.

eps : *float*
The difference between 1.0 and the next smallest representable float larger than 1.0. For example, for 64-bit binary floats in the IEEE-754 standard, `eps = 2**-52`, approximately 2.22e-16.

epsneg : *float*
The difference between 1.0 and the next smallest representable float less than 1.0. For example, for 64-bit binary floats in the IEEE-754 standard, `epsneg = 2**-53`, approximately 1.11e-16.

iexp : *int*
The number of bits in the exponent portion of the floating point representation.

machar : *MachAr*
The object which calculated these parameters and holds more detailed information.

machep : *int*
The exponent that yields *eps*.

max : *floating point number of the appropriate type*
The largest representable number.

maxexp : *int*
The smallest positive power of the base (2) that causes overflow.

min : *floating point number of the appropriate type*
The smallest representable number, typically `-max`.

minexp : *int*
The most negative power of the base (2) consistent with there being no leading 0's in the mantissa.

negep : *int*
The exponent that yields *epsneg*.

nexp : *int*
The number of bits in the exponent including its sign and bias.

nmant : *int*
The number of bits in the mantissa.

precision : *int*
The approximate number of decimal digits to which this kind of float is precise.

resolution : *floating point number of the appropriate type*
The approximate decimal resolution of this type, i.e., `10**-precision`.

tiny : *float*
The smallest positive usable number. Type of *tiny* is an appropriate floating point type.

Figura 1.8: Atributos do método `finfo` da classe `numpy`.

```
In [ ]: np.finfo(dtype=np.float16)
Out[ ]: finfo(resolution=0.001, min=-6.55040e+04, max=6.55040e+04,
        dtype=float16)
```

obtemos como saída uma tupla com a resolução e valores mínimo e máximo representáveis por esse tipo de dado.

Mas usando os atributos mostrados na Figura **??**, podemos visualizar muito mais informações. Vejamos o exemplo a seguir.

Exemplo 3.5

Vamos construir uma função básica para visualizarmos quatro informações: o número de *bits* usado na representação do tipo de dado, o épsilon da máquina o valor máximo que pode ser representado (acima do qual teremos *overflow* e o valor mínimo.

```
In [ ]: def Info_Mach(tipo):
            print('Nº de bits usados: ', np.finfo(dtype=tipo).bits)
            print('Epsilon da máquina: ', np.finfo(dtype=tipo).eps)
            print('Máximo valor: ', np.finfo(dtype=tipo).max)
            print('Mínimo valor: ', np.finfo(dtype=tipo).min)
```

Aqui, o atributo `tipo` se refere a um dos tipos da biblioteca `numpy`. Vamos considerar três tipos em três exemplos:

```
In [ ]: Info_Mach(np.float16)
Out[ ]: Nº de bits usados: 16
        Epsilon da máquina: 0.000977
        Máximo valor: 65500.0
        Mínimo valor: -65500.0

In [ ]: Info_Mach(np.float32)
Out[ ]: Nº de bits usados: 32
        Epsilon da máquina: 1.1920929e-07
        Máximo valor: 3.4028235e+38
        Mínimo valor: -3.4028235e+38

In [ ]: Info_Mach(np.float64)
Out[ ]: Nº de bits usados: 64
        Epsilon da máquina: 2.220446049250313e-16
        Máximo valor: 1.7976931348623157e+308
        Mínimo valor: -1.7976931348623157e+308
```

É interessante percebermos que o épsilon varia com a representação numérica adotada. Quanto mais *bits* temos para representação de mantissa e expoente, menor será o épsilon, aumentando, assim, a precisão das aproximações numéricas feitas e diminuindo os erros relativos a truncamento e arredondamento, conforme discutimos anteriormente.

O padrão IEEE 754 também apresenta alguns valores especiais, definidos para lidar com valores que estão acima da representação máxima ou abaixo da mínima e com valores que sejam numericamente indefinidos.

O primeiro desses valores especiais é a representação de valores “infinitos” na

representação em ponto flutuante. Isso é feito definindo todos os bits de expoente como 1 e todos os bits da mantissa como 0. O bit de sinal pode assumir 0 ou 1, e isso permite representar o infinito positivo e negativo. IEEE 754 especifica que `1.0 / 0.0` deve retornar infinito positivo e `-1.0 / 0.0` deve retornar infinito negativo. Algumas linguagens seguem isso, enquanto outras linguagens (por exemplo, Python) fazem algo diferente, por isso é importante saber com antecedência como esses casos esquivos serão tratados. Independentemente disso, o infinito de ponto flutuante segue todas as convenções usuais. Vejamos alguns exemplos.

```
In [ ]: a = float('inf')
        a
Out[ ]: inf

In [ ]: a + 1
Out[ ]: inf

In [ ]: 2 * a
Out[ ]: inf

In [ ]: -a
Out[ ]: -inf

In [ ]: 1/a
Out[ ]: 0.0

In [ ]: a == float('inf')
Out[ ]: True
```

A última linha mostra que é fácil verificar o infinito positivo usando uma declaração condicional simples. Mesmo que o comportamento seja bastante previsível, vale a pena mencionar que nenhuma dessas declarações lançará uma exceção. Geralmente, é melhor que o código falhe de uma maneira mais perceptível, então, a menos que você queira lidar explicitamente com o infinito, vale a pena verificar se há funções que podem retorná-los (por exemplo, funções logarítmicas).

A segunda constante especial é a `NaN`, acrônimo para *Not a Number*, e é usado para representar um valor numericamente indefinido. Isso é feito definindo todos os bits do expoente como 1 (como com o infinito) e tendo pelo menos um *bit* no significando definido como 1. O IEEE 754 especifica que `0.0 / 0.0` deve retornar `NaN`. Além disso, abusar do infinito de ponto flutuante pode fazer com que `NaN` apareça em seus valores numéricos.

In [ ]:
```
a = float('inf')
a
```

Out[ ]:
```
inf
```

In [ ]:
```
a/a
```

Out[ ]:
```
nan
```

In [ ]:
```
a - a
```

Out[ ]:
```
nan
```

`NaN` pode ser particularmente ruim porque é o resultado de uma falha silenciosa do código numérico e pode ter consequências desastrosas para quaisquer cálculos restantes.

In [ ]:
```
b = float('nan')
b
```

Out[ ]:
```
nan
```

In [ ]:
```
b + 1
```

Out[ ]:
```
nan
```

In [ ]:
```
b - b
```

Out[ ]:
```
nan
```

In [ ]:
```
4 * b
```

Out[ ]:
```
nan
```

In [ ]:
```
b / b
```

Out[ ]:
```
nan
```

As próximas três linhas de código mostram outro problema: detectibilidade. O IEEE 754 afirma que `NaN == NaN` deve sempre retornar `False`, e esta é uma maneira (desajeitada) de verificar se há `NaN`.

In [ ]:
```
b == b
```

Out[ ]:
```
False
```

In [ ]:
```
b == float('nan')
```

Out[ ]:
```
False
```

```
In [ ]: b != b
Out[ ]: True
```

Normalmente, há uma função específica do idioma que verifica o `NaN`. No caso do Python, pode-se usar o método `isnan`, da biblioteca `numpy`, cuja documentação pode ser encontrada nesse link, e que testa se um valor é `NaN`, retornando um booleano como resultado.

```
In [ ]: np.isnan(b)
Out[ ]: True
```

É importante destacar que a biblioteca `numpy` também implementa esses valores especiais. Para saber mais, veja a documentação nesse link.

1.4 – Erros em operações numéricas

Na sessão anterior, discutimos a natureza de erros provenientes de aproximações na representação numérica em sistemas computacionais, ocasionados pelo truncamento ou arredondamento dos números representados.

Quando resolvemos problemas com técnicas numéricas, estamos sujeitos a este e outros tipos de erros. Nesta seção, veremos quais são estes erros e como controlá-los, quando possível.

Quando fazemos aproximações numéricas, os erros são gerados de várias formas, sendo as principais delas as seguintes:

1. *Incerteza dos dados* são devidos aos erros nos dados de entrada. Quando o modelo matemático é oriundo de um problema físico, existe incerteza nas medidas feitas pelos instrumentos de medição, que possuem acurácia finita.

2. *Erros de Arredondamento* são aqueles relacionados com as limitações existentes na forma de representar números em máquina.

3. *Erros de Truncamento* surgem quando aproximamos um conceito matemático formado por uma sequência infinita de passos por de um procedimento finito. Por exemplo, a definição de integral é dada por um processo de limite de somas. Numericamente, aproximamos por um soma finita. O erro de truncamento deve ser estudado analiticamente para cada método empregado e normalmente envolve matemática mais avançada que a estudado em um curso de graduação.

Uma questão fundamental é a quantificação dos erros imbricados na computação da solução de um dado problema. Para tanto, precisamos definir medidas de erros (ou de exatidão). As medidas de erro mais utilizadas são o *erro absoluto* e o *erro relativo*.

Definição 1.3. *Seja x um número real e $\overline{x}$, sua aproximação. O* erro absoluto *da aproximação $\overline{x}$ é definido como*

$$|x - \overline{x}|. \tag{1.10}$$

O erro relativo *da aproximação $\overline{x}$ é definido como*

$$\frac{|x - \overline{x}|}{|x|}, \quad x \neq 0. \tag{1.11}$$

Observe que o erro relativo é adimensional e, muitas vezes, é expresso em porcentagens. Mais precisamente, o erro relativo em porcentagem da aproximação $\overline{x}$ é dado por

$$\frac{|x - \bar{x}|}{|x|} \cdot 100\%. \tag{1.12}$$

Exemplo 4.1

Sejam $x = 123456,789$ e sua aproximação $\bar{x} = 123000$. O erro absoluto é

$$|x - \bar{x}| = |123456,789 - 123000| = 456,789 \tag{1.13}$$

e o erro relativo é

$$\frac{|x - \bar{x}|}{|x|} = \frac{456,789}{123456,789} \approx 0,00369999 \text{ ou } 0,36\% \tag{1.14}$$

Exemplo 4.2

Observe os erros absolutos e relativos em cada caso a seguir:

x	$\bar{x}$	Erro absoluto	Erro relativo
$0,3333\ldots \times 10^{-2}$	$0,3 \times 10^{-2}$	$0,\overline{3} \times 10^{-3}$	10%
$0,3333\ldots$	$0,3$	$0,\overline{3} \times 10^{-2}$	10%
$0,3333\ldots \times 10^{2}$	$0,3 \times 10^{2}$	$0,\overline{3} \times 10^{1}$	10%

Podemos ver que, apesar das diferentes ordens de grandeza, as três medidas tem o mesmo erro relativo. Medidas relativas permitem que medidas com diferentes ordens de grandeza sejam comparáveis.

Cancelamento catastrófico

Quando fazemos subtrações com números muito próximos entre si, ocorre o que chamamos de "cancelamento catastrófico", onde podemos perder vários dígitos de precisão em uma única subtração. Vejamos alguns exemplos.

Exemplo 4.3

Efetue a operação

$$0,987624687925 - 0,987624 = 0,687925 \times 10^{-6} \tag{1.15}$$

usando arredondamento com seis dígitos significativos e observe a diferença se comparado com resultado sem arredondamento.

Os números arredondados com seis dígitos para a mantissa resultam na seguinte diferença

$$0,987625 - 0,987624 = 0,100000 \times 10^{-5} \tag{1.16}$$

Observe que os erros relativos entre os números exatos e aproximados no lado esquerdo são bem pequenos,

$$\frac{|0,987624687925 - 0,987625|}{|0,987624687925|} = 0,003159\% \tag{1.17}$$

e

$$\frac{|0,987624 - 0,987624|}{|0,987624|} = 0\%, \tag{1.18}$$

enquanto no lado direito o erro relativo é enorme:

$$\frac{|0,100000 \times 10^{-5} - 0,687925 \times 10^{-6}|}{0,687925 \times 10^{-6}} = 45,36\%. \tag{1.19}$$

Exemplo 4.4

Considere o problema de encontrar as raízes da equação de segundo grau

$$x^2 + 300x - 0,014 = 0, \tag{1.20}$$

usando seis dígitos significativos.

Aplicando a fórmula de Bhaskara com $a = 0,100000 \times 10^{1}$, $b = 0,300000 \times 10^{3}$ e

$c = 0,140000 \times 10^{-1}$, temos o discriminante:

$$\begin{aligned}
\Delta &= b^2 - 4 \cdot a \cdot c && (1.21)\\
&= 0,300000 \times 10^3 \times 0,300000 \times 10^3 && (1.22)\\
&+ 0,400000 \times 10^1 \times 0,100000 \times 10^1 \times 0,140000 \times 10^{-1} && (1.23)\\
&= 0,900000 \times 10^5 + 0,560000 \times 10^{-1} && (1.24)\\
&= 0,900001 \times 10^5 && (1.25)
\end{aligned}$$

e as raízes:

$$\begin{aligned}
x_1, x_2 &= \frac{-0,300000 \times 10^3 \pm \sqrt{\Delta}}{0,200000 \times 10^1} && (1.26)\\
&= \frac{-0,300000 \times 10^3 \pm \sqrt{0,900001 \times 10^5}}{0,200000 \times 10^1} && (1.27)\\
&= \frac{-0,300000 \times 10^3 \pm 0,300000 \times 10^3}{0,200000 \times 10^1} && (1.28)\\
& && (1.29)
\end{aligned}$$

Então, as duas raízes obtidas com erros de arredondamento, são:

$$\begin{aligned}
\tilde{x}_1 &= \frac{-0,300000 \times 10^3 - 0,300000 \times 10^3}{0,200000 \times 10^1}\\
&= -\frac{0,600000 \times 10^3}{0,200000 \times 10^1} = -0,300000 \times 10^3
\end{aligned} \qquad (1.30)$$

e

$$\tilde{x}_2 = \frac{-0,300000 \times 10^3 + 0,300000 \times 10^3}{0,200000 \times 10^1} = 0,000000 \times 10^0 \qquad (1.31)$$

No entanto, os valores das raízes com seis dígitos significativos livres de erros de arredondamento, são:

$$x_1 = -0,300000 \times 10^3 \quad \text{e} \quad x_2 = 0,466667 \times 10^{-4}. \qquad (1.32)$$

Observe que a primeira raiz apresenta seis dígitos significativos corretos, mas a segunda não possui nenhum dígito significativo correto.

Observe que isto acontece porque b^2 é muito maior que $4ac$, ou seja, $b \approx \sqrt{b^2 - 4ac}$, logo a diferença

$$-b + \sqrt{b^2 - 4ac} \qquad (1.33)$$

estará próxima de zero. Uma maneira de evitar o cancelamento catastrófico é aplicar procedimentos analíticos na expressão para eliminar essa diferença. Um técnica padrão

consiste usar uma expansão em série de Taylor em torno da origem, tal como:

$$\sqrt{1-x} = 1 - \frac{1}{2}x + O(x^2). \tag{1.34}$$

Substituindo esta aproximação na fórmula de Bhaskara, temos:

$$\begin{aligned} x &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} && (1.35)\\ &= \frac{-b \pm b\sqrt{1 - \frac{4ac}{b^2}}}{2a} && (1.36)\\ &\approx \frac{-b \pm b\left(1 - \frac{4ac}{2b^2}\right)}{2a} && (1.37)\\ & && (1.38) \end{aligned}$$

Observe que $\frac{4ac}{b^2}$ é um número pequeno e por isso a expansão faz sentido. Voltamos no exemplo anterior e calculamos as duas raízes com o nova expressão

$$\begin{aligned} \tilde{x}_1 &= \frac{-b - b + \frac{4ac}{2b}}{2a} = -\frac{b}{a} + \frac{c}{b} && (1.39)\\ &= -\frac{0,300000 \times 10^3}{0,100000 \times 10^1} - \frac{0,140000 \times 10^{-1}}{0,300000 \times 10^3} && (1.40)\\ &= -0,300000 \times 10^3 - 0,466667 \times 10^{-4} && (1.41)\\ &= -0,300000 \times 10^3 && (1.42) \end{aligned}$$

$$\begin{aligned} \tilde{x}_2 &= \frac{-b + b - \frac{4ac}{2b}}{2a} && (1.43)\\ &= -\frac{4ac}{4ab} && (1.44)\\ &= -\frac{c}{b} = -\frac{-0,140000 \times 10^{-1}}{0,300000 \times 10^3} = 0,466667 \times 10^{-4} && (1.45)\\ & && (1.46) \end{aligned}$$

Observe que o efeito catastrófico foi eliminado.

O cancelamento catastrófico também poderia ter sido evitado através do seguinte truque analítico

$$\begin{aligned} x_2 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} && (1.47)\\ &= \frac{b^2 - (b^2 - 4ac)}{2a\left(-b - \sqrt{b^2 - 4ac}\right)} = \frac{4ac}{2a\left(-b - \sqrt{b^2 - 4ac}\right)} && (1.48)\\ &= -\frac{2c}{\left(b + \sqrt{b^2 - 4ac}\right)} && (1.49) \end{aligned}$$

Exercícios

■ **Exercício 1.1** *O método "divisão e média", um método antigo para estimação de raiz quadrada de um número positivo a, pode ser formulado como*

$$x_{i+1} = \frac{x_i + a/x_i}{2}. \tag{1.50}$$

Calcule o erro relativo da aproximação para as 10 primeiras iterações.

■ **Exercício 1.2** *Para computadores, o épsilon da máquina, ε, pode ser definido como o menor número que, adicionado a um, retorna um número maior que um, como definimos anteriormente. Usando o algoritmo abaixo, implemente um programa que calcula o épsilon da sua máquina. Compare com os resultados obtidos via `numpy`.*

```
Passo 1: Defina e = 1
Passo 2: Se 1 + e for menor ou igual a 1, vá para o Passo 5;
         caso contrário, vá ao Passo 3
Passo 3: e = e/2
Passo 4: Retorne ao Passo 2
Passo 5: e = 2 x e
```

■ **Exercício 1.3** *Considere o seguinte processo iterativo:*

$$\begin{aligned} x^{(1)} &= \frac{1}{3} \\ x^{(n+1)} &= 4x^{(1)} - 1, n = 1, 2, \ldots \end{aligned} \tag{1.51}$$

Observe que $x^{(1)} = \frac{1}{3}$, $x^{(2)} = 4\frac{1}{3} - 1 = \frac{1}{3}$, $x^{(3)} = 4\frac{1}{3} - 1 = \frac{1}{3}$, e por aí vai, ou seja, temos uma sequência constante igual a $\frac{1}{3}$.

Implemente essa série iterativa, verificando se a convergência de fato ocorre e justifique o resultado obtido.

■ **Exercício 1.4** *Considere as expressões:*

$$\frac{exp(1/\mu)}{1 + exp(1/\mu)} \tag{1.52}$$

e

$$\frac{1}{exp(-1/\mu) + 1} \tag{1.53}$$

com $\mu > 0$. Verifique que elas são idênticas como funções reais. Teste no computador cada uma delas para $\mu = 0,1$, $\mu = 0,01$, $\mu = 0,001$ ou menor e responda: Qual dessas expressões é mais adequada quando μ é um número pequeno? Por quê?

■ **Exercício 1.5** *Observe a seguinte identidade*

$$f(x) = \frac{(1+x)-1}{x}. \tag{1.54}$$

Não é muito difícil verificar, analiticamente, que, para qualquer valor de x, sempre teremos $f(x) = 1$. Faça um programa que calcule o valor da expressão para $x = 10^{-12}$, $x = 10^{-15}$ e $x = 10^{-17}$. Compare os resultados com o resultado analítico e explique-os.

■ **Exercício 1.6** *Existem diferentes métodos para calcular aproximadamente o valor de π computacionalmente. Vários desses métodos usam a aproximação por séries de Taylor para o arcotangente, definida por:*

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{2n+1}$$

São alguns desses métodos:

a) ***Fórmula de Machin***

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

Nota: essa fórmula foi usada em 1949 para calcular o valor de π com 2035 casas decimais no ENIAC.

b) ***Fórmula de Hutton***

$$\frac{\pi}{4} = \arctan\frac{1}{2} + \arctan\frac{1}{3}$$

c) ***Fórmula de Clausen***

$$\frac{\pi}{4} = 2\arctan\frac{1}{3} + \arctan\frac{1}{7}$$

d) ***Fórmula de Dase***

$$\frac{\pi}{4} = \arctan\frac{1}{2} + \arctan\frac{1}{5} + \arctan\frac{1}{8}$$

Gere gráficos que comparem a evolução do erro em função do número de termos considerados, para os vinte primeiros termos de cada método descrito acima e comente os resultados obtidos.