Sistemas de ponto flutuante

Nestas anotações, iremos nos concentrar em sistemas de numeração finitos usados para representar números reais, visando a apresentação dos sistemas de precisão simples e dupla definidos no padrão IEEE754-2008, os quais são fundamentais para a implementação de qualquer método numérico na prática.

1 Sistema de numeração posicional

Um *sistema de numeração posicional* é um esquema de representação de números que leva em consideração a posição dos algarismos que os compõem. O *sistema decimal* é o sistema de numeração posicional que melhor conhecemos. Por exemplo, representamos a quantidade "cento e trinta e dois" pela sequência 132. Decompondo em parcelas, temos que

$$\begin{aligned} 132 &= 100 + 30 + 2 \\ &= 1 \times 10^2 + 3 \times 10^1 + 2 \times 10^0 \end{aligned}$$

No final das contas, os algarismos que compõem o 132 são exatamente os pesos da combinação linear das potências de 10 acima. Dizemos então que as potências de 10 formam uma possível base para a representação de números inteiros, denominada *base decimal*.

Podemos extender esta ideia para os números reais, bastando para isso considerar potências negativas. Por exemplo, a decomposição em potências de 10 do real 3,729 é:

$$\begin{aligned} 3{,}729 &= 3 + 0{,}7 + 0{,}02 + 0{,}009 \\ &= 3 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2} + 9 \times 10^{-3} \end{aligned}$$

Uma pergunta que nos fazemos neste momento é se seria possível utilizar outras potências que não a de 10. A resposta para esta pergunta é sim, podemos utilizar potências das mais variadas. Contudo, é preciso termos cuidado!

Exemplo. (Conversão força bruta) Suponha que desejamos escrever o número 132 como combinação de potências de 8. A maior potência de 8 que pode ser incluída na decomposição do 132 é $8^2 = 64$, pois $8^3 = 512$ já seria maior do que 132. Mas quantas unidades desta potência devemos considerar? Realizando a divisão inteira de 132 por 64, obteremos quantas unidades do 64 cabem em 132. Usando o operador de divisão inteira do Python `//`, temos:

[1]:
```
132//64
```

[1]: 2

Logo, podemos escrever:

$$132 = 2 \times 8^2 + 4$$

Sabendo que $4 = 4 \times 8^0$, obtemos a decomposição:

$$132 = 2 \times 8^2 + 4 \times 8^0$$

Na representação decimal do 132, observe que, da direita para a esquerda, seu primeiro algarismo (2) está associado à potência 10^0, o segundo à potência 10^1 e o terceiro a 10^2. Por isso, dizemos que este sistema de numeração é um sistema *posicional*. Então, a decomposição "completa" do 132 na base 8 precisa incluir o 8^1:

$$132 = 2 \times 8^2 + 0 \times 8^1 + 4 \times 8^0$$

e, agora, podemos estabelecer a seguinte notação:

$$\begin{aligned} 132 &= (204)_8 \\ &:= 2 \times 8^2 + 0 \times 8^1 + 4 \times 8^0 \end{aligned}$$

Observe que omitimos a base subjacente ao número 132, mas poderíamos também ter escrito $(132)_{10}$.

Exemplo. (Base binária) Uma base bastante importante para o cálculo numérico é a base 2 ou base *binária*, a qual consiste de potências de 2. Usando o método força bruta ilustrado no exemplo anterior, vejamos como ficará o 132 na base binária. Primeiramente, determinamos a maior potência de 2 que pode ser usada para decompor o 132. Ora, $2^7 = 128$ e $2^8 = 256$. Portanto, 2^7 é a maior potência de 2 que cabe em 132. Mas quantas unidades do 2^7?

[2]:
```
132//128
```

[2]: 1

Prosseguindo dessa maneira com o resto da divisão inteira, obtemos a decomposição:

$$(132)_{10} = 1 \times 2^7 + 1 \times 2^2$$

Completando a decomposição do 132 na base binária com as potências intermediárias faltantes, teremos:

$$\begin{aligned}(132)_{10} &= 1\times 2^7 + 0\times 2^6 + 0\times 2^5 + 0\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 0\times 2^0 \\ &= (10000100)_2\end{aligned}$$

Observação. (Conjunto dos dígitos) Note que os algarismos que surgem na base decimal são: $0,1,2,3,4,5,6,7,8,9$. Já na base octal, os algarismos variam de 0 a 7. Na base binária, eles só podem valer 0 ou 1 e são conhecidos como *bits* (do inglês, *BInary digiTS*). Os dígitos de um número escrito em uma base $1 < \beta \leq 10$ pertencem ao conjunto $\{0,1,2,\ldots,\beta-1\}$. Quando a base é superior a 10, usamos letras para representar os dígitos maiores ou iguais a 10.

Exemplo. (Algarismos hexadecimais) O sistema de base 16 ou sistema *hexadecimal* é bastante usado em computação para representar números grandes, tais como endereços na memória. Neste caso, os algarismos disponíveis são 0, 1, 2, ..., 8, 9, $A := 10$, $B := 11$, $C := 12$, $D := 13$, $E := 14$ e $F := 15$.

Escrevendo o $(189)_{10}$ na base hexadecimal, teremos:

$$\begin{aligned}(189)_{10} &= 11\times 16^1 + 13\times 16^0 \\ &= (BD)_{16}\end{aligned}$$

Verifique usando o método de força bruta apresentado anteriormente!

1.0.1 Conversão para a base decimal

Dado um número $(x)_\beta$, com base $\beta > 1$, podemos determinar sua representação decimal de modo bem simples. Basta calcular o valor resultante da combinação de potências de β correspondente.

Exemplo. (Octal para decimal) A base 8 ou *octal* é gerada por potências de 8. Podemos converter o número $(35{,}701)_8$ para a base decimal apenas calculando o resultado de sua expansão em potências de 8. Com efeito,

$$\begin{aligned}(35{,}701)_8 &= 3\times 8^1 + 5\times 8^0 + 7\times 8^{-1} + 0\times 8^{-2} + 1\times 8^{-3} \\ &= 24 + 5 + 0{,}875 + 0{,}001953125 \\ &= (29{,}876953125)_{10}\end{aligned}$$

Procedemos do mesmo modo com todas as outras bases.

1.0.2 Conversão de decimal para outras bases

Dado um número decimal com parte inteira $(x)_{10}$ e parte fracionária $(0{,}y)_{10}$, podemos determinar sua representação em uma base $\beta > 1$, convertendo cada uma de suas partes separadamente.

Conversão da parte inteira Para converter o inteiro $(x)_{10}$ para uma base β, podemos utilizar o procedimento a seguir, que é uma espécie de formalização do método força bruta visto anteriormente.

Seja $\div$ o operador de divisão inteira. Considere uma fração com dividendo D e divisor d. Se q é o quociente da divisão inteira de D por d e r o resto correspondente, representaremos isto por:

$$D \div d = q \,\mathrm{R}\, r$$

onde $q \,\mathrm{R}\, r$ pode ser lido como: quociente q e resto r.

Dado $(x)_{10}$, realizamos as divisões inteiras:

$$\begin{aligned}
(x)_{10} \div \beta &= q_1 \,\mathrm{R}\, r_1 \\
q_1 \div \beta &= q_2 \,\mathrm{R}\, r_2 \\
q_2 \div \beta &= q_3 \,\mathrm{R}\, r_3 \\
&\vdots \\
q_{n-2} \div \beta &= q_{n-1} \,\mathrm{R}\, r_{n-1} \\
q_{n-1} \div \beta &= 0 \,\mathrm{R}\, r_n
\end{aligned}$$

até obter $q_n = 0$. Ao final, o número convertido será

$$(r_n r_{n-1} \ldots r_2 r_1)_\beta$$

Exemplo. (Decimal para binário) Considere o problema de convertermos o número $(219)_{10}$ para a base binária. Por conveniência, iremos omitir o subscrito 10. Assim, temos

$$\begin{aligned}
219 \div 2 &= 109 \,\mathrm{R}\, 1 && (8^{\text{o}}\text{ dígito}) \\
109 \div 2 &= 54 \,\mathrm{R}\, 1 && (7^{\text{o}}\text{ dígito}) \\
54 \div 2 &= 27 \,\mathrm{R}\, 0 && (6^{\text{o}}\text{ dígito}) \\
27 \div 2 &= 13 \,\mathrm{R}\, 1 && (5^{\text{o}}\text{ dígito}) \\
13 \div 2 &= 6 \,\mathrm{R}\, 1 && (4^{\text{o}}\text{ dígito}) \\
6 \div 2 &= 3 \,\mathrm{R}\, 0 && (3^{\text{o}}\text{ dígito}) \\
3 \div 2 &= 1 \,\mathrm{R}\, 1 && (2^{\text{o}}\text{ dígito}) \\
1 \div 2 &= 0 \,\mathrm{R}\, 1 && (1^{\text{o}}\text{ dígito})
\end{aligned}$$

Logo, $(219)_{10} = (11011011)_2$.

Exemplo. (Decimal para hexadecimal) Agora, iremos converter o $(219)_{10}$ para a base hexadecimal. Temos que:

$$219 \div 16 = 13\,\mathrm{R}\,11$$
$$13 \div 16 = 0\,\mathrm{R}\,13$$

Como $B := 11$ e $D := 13$, concluímos que $(219)_{10} = (DB)_{16}$.

Conversão da parte fracionária O procedimento de conversão da parte fracionária $(0{,}y)_{10}$ para a base β é o seguinte.

$$\begin{aligned}
(0{,}y)_{10} \times \beta &= (x_1{,}y_1)_{10} \\
(0{,}y_1)_{10} \times \beta &= (x_2{,}y_2)_{10} \\
(0{,}y_2)_{10} \times \beta &= (x_3{,}y_3)_{10} \\
&\vdots \\
(0{,}y_{n-2})_{10} \times \beta &= (x_{n-1}{,}y_{n-1})_{10} \\
(0{,}y_{n-1})_{10} \times \beta &= (x_n{,}y_n)_{10} \\
&\vdots
\end{aligned}$$

Com isso, o valor fracionário convertido será

$$(0{,}x_1x_2\ldots x_{n-1}x_n\ldots)_\beta$$

Exemplo. (Decimal para binário) Considere o problema de convertermos o número $(0{,}73)_{10}$ para a base binária. Por conveniência, iremos omitir o subscrito 10. Assim, temos

$$\begin{aligned}
0{,}73 \times 2 &= 1{,}46 \quad (1^{\text{o}}\ \text{dígito}) \\
0{,}46 \times 2 &= 0{,}92 \quad (2^{\text{o}}\ \text{dígito}) \\
0{,}92 \times 2 &= 1{,}84 \quad (3^{\text{o}}\ \text{dígito}) \\
0{,}84 \times 2 &= 1{,}68 \quad (4^{\text{o}}\ \text{dígito}) \\
0{,}68 \times 2 &= 1{,}36 \quad (5^{\text{o}}\ \text{dígito}) \\
0{,}36 \times 2 &= 0{,}72 \quad (6^{\text{o}}\ \text{dígito}) \\
0{,}72 \times 2 &= 1{,}44 \quad (7^{\text{o}}\ \text{dígito}) \\
0{,}44 \times 2 &= 0{,}88 \quad (8^{\text{o}}\ \text{dígito}) \\
0{,}88 \times 2 &= 1{,}76 \quad (9^{\text{o}}\ \text{dígito}) \\
&\vdots
\end{aligned}$$

Logo, $(0{,}73)_{10} = (0{,}101110101\ldots)_2$.

2 Representação em ponto flutuante

Vimos na seção anterior que muitos números que possuem representação decimal usando um número finito de algarismos podem possuir infinitos dígitos quando escritos em outras bases. Isto pode ser ilustrado pela conversão do $(0{,}1)_{10}$ para a base binária, cujo resultado é $(0{,}000\overline{1100})_2$. Por razões óbvias, conclui-se que é impossível armazenar todos os dígitos de sua representação binária no computador. Uma alternativa para se obter uma representação finita seria, por exemplo, aproximar o valor exato de $(0{,}000\overline{1100})_2$ por $(0{,}0001100)_2$. Esta aproximação introduzirá os *erros de arredondamento*, bastante comuns em sistemas de numeração finitos, mais conhecidos como *sistemas de ponto flutuante*.

Um *sistema de ponto flutuante* é definido por quatro parâmetros inteiros:

(a) sua *base* $\beta > 1$

(b) sua *precisão* $t \geq 2$

(c) seus valores extremos de expoente, $e_{\min} < e_{\max}$.

Dado um sistema de ponto flutuante, um número real x será escrito sob a forma:

$$x = \pm m \times \beta^e$$

onde: * e é o *expoente* de x e satisfaz $e_{\min} \leq e \leq e_{\max}$; * m é a *mantissa* de x e possui a forma $m = (d_0{,}d_1 d_2 d_3 \ldots d_{t-1})_\beta$, com $d_i \in \{0, 1, \ldots, \beta - 1\}$ e $0 \leq m < \beta$.

Para que haja unicidade na representação de cada número, e também para que possamos ter mais números em torno do zero, utiliza-se a *notação científica normalizada*, a qual dá origem a três classes distintas de números:

1. Os números *normais*, quando $1 \leq m < \beta$;
2. Os números *subnormais*, quando $0 < m < 1$ e $e = e_{\min}$;
3. O *zero*, quando $m = (0{,}000\ldots0)_\beta$ e $e = e_{\min}$.

Exemplo. (Sistema decimal finito) Considere um sistema numérico com $\beta = 10$, $t = 3$, $e_{\min} = -2$ e $e_{\max} = 3$. Suponha que desejamos representar $x = (3)_{10}$ neste formato. Primeiramente, tentamos escrevê-lo como um número normal:

$$x = +(3{,}00)_{10} \times 10^0$$

Observe que esta representação atende aos requisitos dos números *normais*. Logo, a representação obtida está correta.

Agora, vejamos como ficaria o número $y = (0{,}002)_{10}$. Caso fosse normal, deveria ser escrito como:

$$y = +(2{,}00)_{10} \times 10^{-3}$$

Mas o expoente -3 extrapolou o $e_{\min}$. Resta-nos então tentar escrevê-lo como um número subnormal. Com efeito,

$$y = +(0{,}20)_{10} \times 10^{-2}$$

Portanto, esta será a representação para y, um número subnormal.

Exemplo. (Sistema binário finito) Considere um sistema binário com $t = 3$, $e_{\min} = 0$ e $e_{\max} = 1$. Dado $x = (0{,}75)_{10}$, tentamos escrevê-lo como um número normal:

$$\begin{aligned} x &= +(0{,}11)_2 \times 2^0, \\ &= +(1{,}10)_2 \times 2^{-1} \end{aligned}$$

Embora a mantissa obtida acima seja permitida para números normais, o expoente -1 é menor do que $e_{\min} = 0$. Logo, x não será um número normal. Vejamos se ele pode atender às condições de número subnormal. Ora,

$$\begin{aligned} x &= +(1{,}10)_2 \times 2^{-1} \\ &= +(0{,}11)_2 \times 2^0 \end{aligned}$$

Como a última representação obtida possui mantissa entre 0 e 1 e o expoente é exatamente o $e_{\min}$, concluímos que x será representado por um número subnormal.

Exemplo. (Enumerando um sistema de ponto flutuante) Neste momento, talvez já tenhamos percebido que um sistema de ponto flutuante consiste de um número finito de números, os quais corresponderão a representações exatas ou aproximadas de números reais. Para enfatizar ainda mais esta finitude, considere um sistema decimal com precisão $t = 2$ e expoentes extremais $e_{\min} = 0$ e $e_{\max} = 1$. Um número x neste sistema terá a forma:

$$x = \pm(d_0{,}d_1)_{10} \times 10^e$$

Se x for um número normal, o dígito d_0 somente pode assumir os valores 1, 2, ... ou 9. Já o dígito d_1, além desses, poderá ser igual a zero. Com isso, a quantidade de permutações determinadas pelos dois dígitos juntos será $9 \times 10 = 90$. Levando em consideração que temos números positivos e negativos, e que os normais podem apresentar qualquer valor de expoente no intervalo de $e_{\min} = 0$ a $e_{\max} = 1$, concluímos que este sistema possui $2 \times 2 \times 90 = 360$ números normais.

Vejamos agora quantos subnormais existirão neste sistema. Se x for subnormal, então:

$$x = \pm(0{,}d_1)_{10} \times 10^{e_{\min}}$$

com $d_1 = 1, 2, \ldots$ ou 9. Considerando as duas possibilidades de sinal ($\pm$), teremos $2 \times 9 = 18$ subnormais.

Falta apenas contabilizar o zero, ou melhor, os dois zeros, o "positivo" e o "negativo". De fato, ambas as situações abaixo representarão o valor zero:

$$x = +(0{,}0)_{10} \times 10^{e_{\min}}, \qquad x = -(0{,}0)_{10} \times 10^{e_{\min}}$$

Embora este sistema seja decimal e o sistema usado em nossos computadores seja binário, podemos utilizar a `Matplotlib` para ao menos ilustrar a distribuição de todos estes valores na reta real.

Primeiramente, carregamos as bibliotecas que usaremos:

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Por simplicidade, iremos armazenar apenas os números normais positivos na lista abaixo:

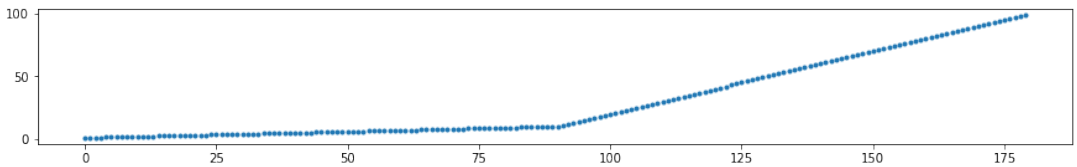
[4]:
```
normal = []
```

Geramos agora os números normais percorrendo todos os valores possíveis para os dígitos d_0 e d_1, e para o expoente e. Com efeito,

[5]:
```
for d0 in range(1,10):
    for d1 in range(10):
        for e in range(2):
            valor = (np.float64(d0) + 0.1*np.float64(d1)) * (10**e) #␣
 ↪convertendo os dígitos para real com a numpy
            normal.append(valor)
```

Finalmente, plotamos a reta real com os pontos calculados com o auxílio da `Matplotlib`

[6]:
```
if (len(normal) == 180): # verificamos se geramos todos os normais␣
 ↪positivos
    # criamos a figura e o eixo coordenado
    fig, ax = plt.subplots(figsize=(15,2))
    plt.rc('xtick', labelsize=16) # configuramos o tamanho da fonte␣
 ↪usada nos eixo x
    plt.rc('ytick', labelsize=16) # configuramos o tamanho da fonte␣
 ↪usada nos eixo y
    normal.sort()
    ax.plot(normal, '.') # plota um segmento de reta para cada número␣
 ↪com altura igual a sua magnitude
    plt.show()
else:
    print("Está faltando algum normal positivo.")
```



Observe que a distância entre os números diminui à medida que a magnitude deles diminui. O oposto acontece quando aumentamos a magnitude.

Valores extremos Por serem finitos, os sistemas de ponto flutuante possuem uma capacidade limitada para a representação de números reais. Dois valores críticos que precisamos conhecer são o maior valor positivo Ω que pode ser representado e o menor valor positivo α diferente de zero. O primeiro será um número normal, enquanto que o segundo será naturalmente um subnormal. Dado um sistema de ponto flutuante arbitrário, teremos:

(a) O maior normal positivo:

$$\Omega = +(d,\underbrace{ddddd\ldots d}_{t-1 \text{ dígitos}})_\beta \times \beta^{e_{\max}},$$

com $d = \beta - 1$. Escrevendo a mantissa de Ω sob a forma de uma progressão geométrica com razão β^{-1}, obtemos:

$$\begin{aligned}(d,ddddd\ldots d)_\beta &= (\beta-1)\left(\beta^0+\beta^{-1}+\beta^{-2}+\cdots+\beta^{1-t}\right)\\ &= (\beta-\beta^{1-t}),\end{aligned}$$

e, com isso, $\Omega = (\beta - \beta^{1-t}) \times \beta^{e_{\max}}$.

(b) O menor subnormal positivo:

$$\alpha = +(0,\underbrace{000000\ldots 1}_{t-1 \text{ dígitos}})_\beta \times \beta^{e_{\min}} = \beta^{-(t-1)}\beta^{e_{\min}} = \beta^{e_{\min}-t+1}$$

Igualmente interessantes são também:

(c) O menor normal positivo:

$$+(1,\underbrace{000000\ldots 0}_{t-1 \text{ dígitos}})_\beta \times \beta^{e_{\min}} = \beta^{e_{\min}}.$$

(d) O maior subnormal:

$$+(0,\underbrace{ddddd\ldots d}_{t-1 \text{ dígitos}})_\beta \times \beta^{e_{\min}} = (1-\beta^{1-t}) \times \beta^{e_{\min}}$$

onde $d = \beta - 1$.

Exemplo. (Sistema binário finito) Para um sistema binário com precisão $t = 3$, $e_{\min} = 0$ e $e_{\max} = 3$, temos:

- O maior normal positivo é $\Omega = (2 - 2^{1-3}) \times 2^1 = (3{,}5)_{10}$
- O menor subnormal positivo é $\alpha = 2^{0-3+1} = (0{,}25)_{10}$
- O menor normal positivo é $2^0 = (1)_{10}$
- O maior subnormal positivo é $(1 - 2^{1-3}) \times 2^0 = (0{,}75)_{10}$

Por conveniência, iremos definir uma função que calcula os limites positivos de um sistema de ponto flutuante arbitrário, dados os valores de β, t, $e_{\min}$ e $e_{\max}$.

[7]:
```
def spf_limites(beta,t,emin,emax):
    normal_min = beta**emin
    normal_max = (beta - beta**(1 - t)) * (beta**emax)
    subnormal_min = beta**(emin - t + 1)
    subnormal_max = (1.0 - beta**(1 - t)) * (beta**emin)
    print(f'Menor normal positivo: {normal_min}')
    print(f'Maior normal positivo: {normal_max}')
    print(f'Menor subnormal positivo: {subnormal_min}')
    print(f'Maior subnormal positivo: {subnormal_max}')
```

Exemplo. (Um sistema binário um pouco mais real) Usando $\beta = 2$, $t = 24$, $e_{\min} = -126$ e $e_{\max} = +127$, temos os limites:

[8]:
```
spf_limites(2,24,-126,127)
```

```
Menor normal positivo: 1.1754943508222875e-38
Maior normal positivo: 3.4028234663852886e+38
Menor subnormal positivo: 1.401298464324817e-45
Maior subnormal positivo: 1.1754942106924411e-38
```

Exemplo. (Precisão dupla na numpy) O sistema padrão para números reais do Python possui precisão variável. Por um lado isto é bom, pois de certa forma ele tentar representar os números do modo mais preciso possível. No entanto, isto pode aumentar o custo dos algoritmos que realizam uma grande quantidade de operações aritméticas em ponto flutuante. Na prática, o sistema mais usado em métodos numéricos é conhecido como sistema de *precisão dupla*. Neste sistema, são usados 64 bits no total, sendo um dedicado ao sinal, 11 para o expoente e 53 bits para a mantissa. Observe que a soma $1 + 11 + 53 = 65$ excede o limite de 64. Isto acontece porque o bit extra que aparece na mantissa não é armazenado de fato, ele surge apenas de modo implícito. Este sistema possui $t = 53$, $e_{\min} = -1022$ e $e_{\max} = +1023$. Com isso, temos

[9]:
```
spf_limites(2,53,-1022,1023)
```

```
Menor normal positivo: 2.2250738585072014e-308
Maior normal positivo: 1.7976931348623157e+308
Menor subnormal positivo: 5e-324
Maior subnormal positivo: 2.225073858507201e-308
```

O tipo `np.float64` é exatamente o sistema de precisão dupla citado acima. A função `np.finfo` permite acessar, dentre outros parâmetros, os limites para os números normais positivos do sistema `np.float64`:

[10]:
```
np.finfo(np.float64).max
```

[10]:
```
1.7976931348623157e+308
```

[11]:
```
np.finfo(np.float64).tiny
```

[11]:
```
2.2250738585072014e-308
```

3 Dígitos significativos

Constumamos definir os *dígitos significativos* de um número x como sendo todos os dígitos de x compreendidos entre os dígitos não nulos mais à esqueda e mais à direita de x, incluindo estes dois dígitos extremos. Assim, o número 3,141592 possui 7 dígitos significativos, enquanto que 0,049900 possui apenas 3, a saber, os dígitos 4, 9 e 9.

Uma pergunta bastante simples de ser respondida por um humano letrado em matemática é: quantos dígitos significativos tem um número? Ora, por simples inspeção, verificando a regra descrita anteriormente, conseguimos rapidamente obter a reposta correta. No entanto, como poderíamos fazer para que, por exemplo, um computador também fosse capaz de responder a esta pergunta?

Suponha que um número real x possui $t > 0$ dígitos significativos. Sem perda de generalidade, suponha que $x = +(d_{t-1}d_{t-2}\cdots d_1d_0)_{10}$, $d_{t-1} \neq 0$. Então,

$$\begin{aligned} x &= +(d_{t-1}d_{t-2}\cdots d_1d_0)_{10} && (1)\\ &\leq +(\underbrace{99\cdots99}_{t \text{ dígitos}})_{10} && (2)\\ &< 10^t && (3) \end{aligned}$$

Por outro lado,

$$\begin{aligned} x &= +(d_{t-1}d_{t-2}\cdots d_1d_0)_{10} && (4)\\ &\geq +(\underbrace{10\cdots00}_{t \text{ dígitos}})_{10} && (5)\\ &= 10^{t-1} && (6) \end{aligned}$$

ou seja,

$$10^{t-1} \leq x \leq 10^t$$

Aplicando o logaritmo na base 10 nesta desigualdade, obtemos:

$$t - 1 \leq \log_{10} x \leq t,$$

logo, $t = \lfloor \log_{10} x \rfloor + 1$.

No caso geral, em uma base β, teremos $\lfloor \log_\beta x \rfloor + 1$ dígitos significativos.

Para números com parte fracionária, basta desconsiderar a vírgula. Por exemplo, para o número $(3{,}75)_{10}$, temos um total de

[12]:
```
import numpy as np

np.floor(np.log10(375)) + 1
```

[12]: 3.0

dígitos decimais.

O mesmo número na base binária vale $(11{,}11)_2$. Aqui, o total de bits será:

[13]:
```
np.floor(np.log2(15)) + 1
```

[13]: 4.0

visto que aplicamos o logaritmo ao número $(11{,}11)_2$ sem a vírgula, ou seja, a $(1111)_2 = (15)_{10}$.

4 Erro absoluto e erro relativo

Suponha que x é o *valor exato* (ou *verdadeiro*) de uma grandeza escalar real e que $\tilde{x}$ é uma aproximação para x. Podemos relacioná-los do seguinte modo:

$$\tilde{x} = x + \mathrm{E}(\tilde{x}), \tag{7}$$

onde $\mathrm{E}(\tilde{x})$ é o *erro exato* associado à aproximação $\tilde{x}$. Isolando $\mathrm{E}(\tilde{x})$ na equação acima, obtemos a definição do erro exato:

$$\mathrm{E}(\tilde{x}) = \tilde{x} - x. \tag{8}$$

Exemplo. (Cálculo do erro exato) Sejam $x = 1{,}27$ e $y = 1341$. Por definição, o erro exato da aproximação $\tilde{x} = 1{,}2$ é $\mathrm{E}(\tilde{x}) = 1{,}2 - 1{,}27 = -0{,}07$. Para $\tilde{y} = 1340$, temos $\mathrm{E}(\tilde{y}) = 1340 - 1341 = -1$.

Observação. É importante perceber que o cálculo do erro exato $\mathrm{E}(\tilde{x})$ requer o conhecimento do valor exato x que está sendo aproximado. Ao longo de todo o curso, veremos que na prática raramente temos acesso a este valor. Isto fará com que trabalhemos com estimativas de erro, ao invés do erro exato.

O sinal no erro exato, embora nos traga a informação de em qual direção estamos errando, para cima ou para baixo, muitas vezes pode ser omitido. Para estes casos, definimos o **erro absoluto exato**, denotado por $\mathrm{EA}(\tilde{x}) = |\mathrm{E}(\tilde{x})|$.

Exemplo. (Cálculo do erro absoluto exato)
Considere novamente $x = 1{,}27$, $y = 1341$ e as aproximações $\tilde{x} = 1{,}2$ e $\tilde{y} = 1340$. Temos que $\mathrm{EA}(\tilde{x}) = |-0{,}07| = 0{,}07$ e $\mathrm{EA}(\tilde{y}) = |-1| = 1$.

Comparando os erros absolutos cometidos no exemplo acima, podemos ser levados a concluir que o erro em y é bem maior do que em x, pois 1 é mais de 10 vezes maior do que 0,07. No entanto, o erro em x representa cerca de 5% do valor exato, enquanto que o erro em y vale cerca de 0,08% do valor de y. Portanto, a melhor aproximação na verdade é a de y e não a de x. Isto nos leva a definir uma nova fórmula para o cálculo do erro, agora considerando a magnitude dos números envolvidos. O **erro relativo exato** $\mathrm{ER}(\tilde{x})$ é dado por:

$$\mathrm{ER}(\tilde{x}) = \frac{\mathrm{EA}(\tilde{x})}{|x|} = \frac{|\tilde{x} - x|}{|x|}, \qquad x \neq 0. \tag{9}$$

Exemplo. (Cálculo do erro relativo exato)
Retomando os valores anteriores: $x = 1{,}27$, $y = 1341$, $\tilde{x} = 1{,}2$ e $\tilde{y} = 1340$; temos:

$$\mathrm{ER}(\tilde{x}) = \frac{|-0{,}07|}{|1{,}27|} \approx 0{,}055 \quad \text{e} \quad \mathrm{ER}(\tilde{y}) = \frac{|-1|}{|1341|} \approx 0{,}000746.$$

Logo, $\tilde{y}$ representa uma melhor aproximação para y do que $\tilde{x}$ para x, como suspeitamos.

5 Exatidão e precisão

Ao lidarmos com aproximações é importante conhecer a diferença entre medidas de *exatidão* (ou *acurácia*) e *precisão*. A *exatidão* é usada para representar a diferença entre uma aproximação $\tilde{x}$ e o valor exato x correspondente. Já a *precisão* está relacionada com a distância entre duas ou mais aproximações. É justamente esta diferença que nos faz usar apenas medidas de precisão nos algoritmos numéricos, visto que dificilmente teremos acesso ao valor exato em questão.

Exemplo. Considere um número $x = 3{,}1415927$ com 8 dígitos significativos. Os vizinhos mais próximos a x, com 7 dígitos significativos, são $x_- = 3{,}141592$ e $x_+ = 3{,}141593$. Analisando os erros absolutos:

$$|x_- - x| = 7 \times 10^{-7} \tag{10}$$
$$|x_+ - x| = 3 \times 10^{-7} \tag{11}$$

podemos concluir que x está mais próximo de x_+. Neste caso, dizemos que $\tilde{x} = x_+$ é uma aproximação para x com exatidão de 7 dígitos significativos.

Exemplo. Considere $x = 5$ e $\tilde{x} = 4{,}994$. Faremos aqui uma pergunta levemente diferente da anterior. Com qual exatidão $\tilde{x}$ aproxima x? Ora, os vizinhos mais próximos de x com 4 dígitos significativos são $x_- = 4{,}999$ e $x_+ = 5{,}001$. Como $\tilde{x}$ não está compreendido entre 4,999 e 5,001, é impossível atingir uma exatidão de 4 dígitos. Por outro lado, ao considerarmos 3 dígitos significativos, os vizinhos de x são $x_- = 4{,}99$ e $x_+ = 5{,}01$ e, agora, $\tilde{x}$ pertence à vizinhança definida por x_- e x_+, para os quais temos os erros:

$$|\tilde{x} - x_-| = 4 \times 10^{-3} \tag{12}$$
$$|\tilde{x} - x| = 6 \times 10^{-3} \tag{13}$$
$$|\tilde{x} - x_+| = 16 \times 10^{-3} \tag{14}$$

Observe que $\tilde{x}$ está mais próximo de x_- do que de x. Portanto, se ele aproxima algum número com acurácia de 3 dígitos, esse número não deveria ser x. Novamente, como $\tilde{x}$ não atingiu a exatidão de 3 dígitos para x, iremos verificar com 2 dígitos. Com efeito, temos $X = 5{,}0$ a melhor aproximação com 2 dígitos significativos para x, acompanhada de seus vizinhos $X_- = 4{,}9$ e $x_+ = 5{,}1$. Analisando os erros:

$$|\tilde{x} - X_-| = 9{,}4 \times 10^{-2} \tag{15}$$
$$|\tilde{x} - X| = 0{,}6 \times 10^{-2} \tag{16}$$
$$|\tilde{x} - X_+| = 10{,}6 \times 10^{-2} \tag{17}$$

concluímos que $\tilde{x} = 4{,}994$ é uma aproximação com exatidão de 2 dígitos significativos para $x = 5$.

Exemplo. Vejamos mais um exemplo interessante, que nos ajudará a estabelecer um critério para decidir a exatidão de uma aproximação. Considere $x = 0{,}5$ e $\tilde{x} = 0{,}51$. Pelo que vimos no exemplo anterior, no máximo teremos exatidão de 2 dígitos. Neste caso, os vizinhos de x são $x_- = 0{,}49$ e $x_+ = 0{,}51$. Logo, $\tilde{x}$ melhor aproxima x_+ com 2 dígitos.

Para analisar a exatidão de 1 dígito, tomamos $X = 0{,}5$, $X_- = 0{,}4$, $X_+ = 0{,}6$ e calculamos os erros:

$$|\tilde{x} - X_-| = 11 \times 10^{-2} \tag{18}$$
$$|\tilde{x} - X| = 1 \times 10^{-2} \tag{19}$$
$$|\tilde{x} - X_+| = 9 \times 10^{-2} \tag{20}$$

Logo, $\tilde{x} = 0{,}51$ aproxima $x = 0{,}5$ com exatidão de 1 dígito significativo.

Observe o padrão que ocorreu na comparação dos erros absolutos. Usando notação científica, para decidir se a exatidão de $\tilde{x} = (d_0{,}d_1d_2\cdots)_{10} \times 10^n$ era de t dígitos, $d_0 \neq 0$, sempre escrevemos os erros sob a forma $E \times 10^{n-t}$. A melhor aproximação foi sempre aquela que obteve o erro absoluto menor ou igual a $5 \times 10^{n-t}$. Com base nisso, estabelecemos a seguinte definição:

Definição. Dizemos que $\tilde{x} = (d_0{,}d_1d_2\cdots)_{10} \times 10^n$, $d_0 \neq 0$, aproxima x com exatidão de t dígitos significativos quando $|\tilde{x} - x| \leq 5 \times 10^{n-t}$.

Como na maioria das vezes, estamos interessados na **exatidão maximal** de uma aproximação, ou seja, no maior valor de t, é comum nos referirmos à exatidão de uma aproximação como sinônimo de exatidão maximal.

Exemplo. (Grinshpan, 2012) Usando esta definição, temos que:

x	$\tilde{x}$	$\lvert\tilde{x} - x\rvert$	exatidão
5	5,1	0,1	1
5	4,995	0,005	3
5	4,994	0,006	2
2	1,4	0,6	0
0,5	0,51	0,01	1

Exemplo. Qual a exatidão da aproximação $\tilde{x} = 1{,}4142$ para o valor da raiz de dois, considerando seu valor exato como aquele arredondado para 12 dígitos significativos? Obviamente, a precisão não excederá a quantidade de dígitos significativos de $\tilde{x}$, que é 5.

Precisamos determinar o maior valor de t que atenda à definição de exatidão estabelecida. Com efeito,

$$|\tilde{x} - x| \leq 5 \times 10^{-t} \tag{21}$$
$$|1{,}4142 - 1{,}41421356237| \leq 5 \times 10^{-t} \tag{22}$$
$$1{,}356237 \times 10^{-5} \leq 5 \times 10^{-t} \tag{23}$$

cuja solução aproximada é $t \leq 5{,}57$. Logo, a exatidão de $\tilde{x}$ é 5.

6 Exatidão de números normais

Felizmente, como veremos a seguir, o conceito de exatidão para números normais permanece válido. Ilustraremos isso com o exemplo a seguir.

Considere um número $x = +(d_0{,}d_1 d_2 \cdots d_{t-1} \cdots)_{10} \times 10^n$, $d_0 \neq 0$. Como vimos anteriormente, os vizinhos de x com t dígitos significativos são:

$$x_- = (d_0{,}d_1 d_2 \cdots d_{t-1})_{10} \times 10^n \tag{24}$$
$$x_+ = (d_0{,}d_1 d_2 \cdots d_{t-1})_{10} \times 10^n + \underbrace{(0{,}00\cdots 1)}_{t \text{ dígitos}}{}_{10} \times 10^n = x_- + 10^{n+1-t} \tag{25}$$

Se $\tilde{x}$ aproxima x com exatidão de t dígitos, então x pertence ao intervalo $[x_-, x_+]$. Isso quer dizer então que:

$$|\tilde{x} - x| \leq \frac{x_+ - x_-}{2} = \frac{1}{2} \times 10^{n+1-t} = 5 \times 10^{n-t}$$

conforme estabelecido anteriormente.

Exemplo. A constante de Ramanujam é $R = e^{\pi\sqrt{163}} = 2{,}6253741264076874399999999999992500\cdots \times 10^{17}$. Qual é a exatidão da aproximação $2{,}6253741264076 \times 10^{17}$? Vejamos:

$$|\tilde{x} - x| \leq 5 \times 10^{17-t} \tag{26}$$
$$0{,}0000000000000874399999999999992500\cdots \times 10^{17} \leq 5 \times 10^{17-t} \tag{27}$$
$$8{,}7439999999999992500\cdots \times 10^{-14} \times 10^{17} \leq 5 \times 10^{17-t} \tag{28}$$
$$8{,}7439999999999992500\cdots \times 10^{3} \leq 5 \times 10^{17-t} \tag{29}$$

cuja solução aproximada é $t \leq 13{,}76$. Portanto, $\tilde{x}$ possui exatidão de 13 dígitos.

6.0.1 Relação com o erro relativo

A definição da quantidade de dígitos significativos foi dada em termos do erro absoluto. Vejamos como relacioná-la com a definição de erro relativo. Seja $x = +(d_0,d_1d_2\cdots d_{t-1}\cdots)_{10} \times 10^n$. Então,

$$(d_0)_{10} \times 10^n \leq |x| \leq (d_0+1)_{10} \times 10^n$$

Se $\tilde{x}$ é uma aproximação de x com exatidão de t dígitos, então:

$$\frac{|\tilde{x}-x|}{|x|} \leq \frac{5\times 10^{n-t}}{|x|} \leq \frac{5\times 10^{n-t}}{d_0\times 10^n} \leq 5\times 10^{-t}$$

O resultado acima nos diz que se $\tilde{x}$ tem exatidão de t dígitos, então o erro relativo correspondente será da ordem de 10^{-t}. No entanto, como veremos agora, não vale a recíproca.

Com efeito, suponha que $\tilde{x}$ é tal que:

$$\frac{|\tilde{x}-x|}{|x|} \leq 5\times 10^{-t}$$

Então,

$$|\tilde{x}-x| \leq 5\,|x| \times 10^{-t} \leq 5(d_0+1)\times 10^{n-t}.$$

Na pior das hipóteses, $d_0 = 9$ e com isso:

$$|\tilde{x}-x| \leq 5\times 10\cdot 10^{n-t} = 5\times 10^{n-(t-1)},$$

isto é, $\tilde{x}$ possuirá no mínimo uma exatidão de $t-1$ dígitos.

Em suma, limitar o erro relativo a 5×10^{-t} não garante que teremos t dígitos significativos de exatidão, mas isso nos dá a certeza de ter ao menos uma exatidão de $t-1$ dígitos.

Exemplo. Seja $\tilde{x} = 3{,}1415$ uma aproximação para $x = \pi$. Os erros absoluto e relativo satisfazem:

$$|\tilde{x}-x| \leq 9{,}27\times 10^{-5} \leq 5\times 10^{-4} \quad \text{e} \quad \frac{|\tilde{x}-x|}{|x|} \leq 2{,}95\times 10^{-5} \leq 5\times 10^{-5}.$$

Portanto, a exatidão de $\tilde{x}$ é somente de 4 dígitos, contrariando o que o erro relativo indica.

Observação. Embora haja essa diferença, na prática, é mais fácil usar o erro relativo como critério de parada de um método numérico do que a quantidade de dígitos significativos. Por isso, alguns livros, como é o caso do nosso livro-texto, optam por afirmar que se o erro relativo de $\tilde{x}$ é limitado por 5×10^{-t}, então ele terá exatidão de t dígitos. Como vimos há pouco, até pode ter, mas eventualmente a precisão poderá ser $t-1$.

7 Regras de arredondamento

Chegando aqui, percebemos que um sistema numérico de ponto flutuante $\mathbb{F}(\beta,t,e_{\min},e_{\max})$ nada mais é do que um subconjunto **finito** dos reais. Portanto, é de se esperar que nem todas as propriedades dos números reais permaneçam válidas neste sistema. De fato, se desejamos representar um número real x por um elemento de $\mathbb{F}$, haverá duas situações. É possível que x seja exatamente um dos elementos de $\mathbb{F}$ ou que x não pertença a $\mathbb{F}$. No primeiro caso, não teremos problema. Já no segundo, precisaremos escolher um elemento de $\mathbb{F}$ que possa representar x da melhor forma possível. A "melhor forma" dependerá de nossos objetivos, como descreveremos em breve. Denominamos de *arredondamento* o processo de escolha de um elemento de $\mathbb{F}$ para representar um real x e denotamos o resultado por $\mathrm{fl}(x)$.

Considere um sistema de ponto flutuante $\mathbb{F}(\beta,t,e_{\min},e_{\max})$. Dado um número **real** $x = (d_0,d_1d_2\ldots d_{t-1}d_t\ldots)_\beta \times \beta^e$, o maior número de $\mathbb{F}$ menor do que x é dado por:

$$x_- = (d_0,d_1d_2\ldots d_{t-1})_\beta \times \beta^e$$

Somando uma unidade ao dígito d_{t-1} deste número obtemos o sucessor de x_- neste sistema, denotado por x_+. Observe que $x_- \leq x \leq x_+$ e que o tamanho do "vazio" entre x_- e x_+ é $\beta^{1-t}\times\beta^e$.

Com base nisso, definimos quatro modos de arredondamento.

- **para baixo**:

$$\mathrm{fl}(x) = \mathrm{RD}(x) := \begin{cases} x_- & \text{se } x > 0 \\ x_+ & \text{caso contrário} \end{cases}$$

- **para cima**:

$$\mathrm{fl}(x) = \mathrm{RU}(x) := \begin{cases} x_+ & \text{se } x > 0 \\ x_- & \text{caso contrário} \end{cases}$$

- **em direção ao zero**:

$$\mathrm{fl}(x) = \mathrm{RZ}(x) := x_-$$

- **para o mais próximo**:

$$\mathrm{fl}(x) = \mathrm{RN}(x),$$

onde $\mathrm{RN}(x)$ é x_- ou x_+, aquele que estiver mais próximo de x. Em caso de empate, aconselha-se quebrar os empates com a regra do dígito par mais à direita (regra do desempate par).

Exemplo. (Arredondamento em sistemas decimais) O algoritmo para arredondamento em sistemas decimais usando a regra de arredondamento para o mais próximo é o seguinte. Dado $x = (d_0,d_1d_2\ldots d_{t-1}d_t\ldots)_\beta \times \beta^e$,

1. Se $d_t < 5$, então $\mathrm{fl}(x) = x_-$;

2. Senão se $d_t > 5$, então $\text{fl}(x) = x_+$;
3. Caso contrário, d_t vale 5 e teremos uma das situações:
4. Se ao menos um dos dígitos subsequentes ao d_t for diferente de zero, então $\text{fl}(x) = x_+$
5. Caso contrário, existe um empate. Para resolvê-lo, analisamos os dígitos $j = (t-1)$ até 1 de ambos, x_- e x_+. Arredondamos x para aquele cujo j-ésimo dígito é par.

Exemplo. Considere um sistema decimal com $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$. Dados $x = (1{,}625)_{10}$ e $y = (1{,}7)_{10}$, quanto valem $\text{RN}(x)$ e $\text{RN}(y)$?

Sabemos que $(1{,}625)_{10}$ está exatamente no meio do intervalo definido por:

$$x_- = (1{,}62)_{10} \times 10^0 \quad \text{e} \quad x_+ = (1{,}63)_{10} \times 10^0.$$

Como x_- possui $d_2 = 2$ par, concluímos que

$$\text{fl}(x) = (1{,}62)_{10} \times 10^0.$$

Observe que o erro absoluto desta aproximação é $\text{EA}(x) = |1{,}62 - 1{,}625| = 0{,}005$, enquanto que o erro relativo é:

$$\text{ER}(x) = \frac{0{,}005}{1{,}625} \approx 0{,}003077$$

É claro ver que no caso de $y = (1{,}7)_{10}$, teremos $\text{fl}(y) = y$ e, com isso, $\text{EA}(y) = \text{ER}(y) = 0$.

Exemplo. (Arredondamento para o mais próximo em sistemas binários) Dado um sistema binário com precisão t e $x = (b_0{,}b_1b_2\ldots b_{t-1}b_t\ldots)_2 \times 2^e$, procedemos com o arredondamento para o mais próximo da seguinte maneira:

1. Se b_t for igual a zero, então $\text{fl}(x) = x_-$;
2. Caso contrário,
   2.1 Se ao menos um dos bits subsequentes for igual a 1, então $\text{fl}(x) = x_+$;
   2.2 Caso contrário, existe um empate. Por isso, é preciso analisar os bits nas posições $j = (t-1)$ até 1 de ambos x_- e x_+. Arredondamos x para aquele cujo j-ésimo bit vale zero (regra do desempate par).

Exemplo. Considere um sistema com $\beta = 2$, $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$. Dados $x = (1{,}625)_{10}$ e $y = (1{,}7)_{10}$, quanto valem $\text{RN}(x)$ e $\text{RN}(y)$? Calcule os erros absoluto e relativo correspondentes.

Sabemos que:

$$(1{,}625)_{10} = (1{,}101000\ldots)_2 \times 2^0$$

o qual está exatamente no meio do intervalo definido por:

$$x_- = (1{,}10)_2 \times 2^0 = (1{,}5)_{10} \quad \text{e} \quad x_+ = (1{,}11)_2 \times 2^0 = (1{,}75)_{10}.$$

Como o bit d_2 de x_- vale 0, concluímos que

$$\mathrm{RN}(x) = x_- = (1{,}10)_2 \times 2^0 = (1{,}5)_{10}.$$

O erro absoluto é $\mathrm{EA}(x) = |1{,}5 - 1{,}625| = 0{,}125$. Já o erro relativo vale:

$$\mathrm{ER}(x) = \frac{0{,}125}{1{,}625} \approx 0{,}076923$$

Agora, para $y = (1{,}7)_{10} = (1{,}10\overline{1100})_2 \times 2^0$, temos:

$$y_- = (1{,}10)_2 \times 2^0 = (1{,}5)_{10} \quad \text{e} \quad y_+ = (1{,}11)_2 \times 2^0 = (1{,}75)_{10}.$$

Como o bit d_3 de y vale 1 e existem bits subsequentes também iguais a 1, concluímos que:

$$\mathrm{RN}(y) = y_+ = (1{,}11)_2 \times 2^0 = (1{,}75)_{10}.$$

Nesse caso, o erro absoluto é $\mathrm{EA}(y) = |1{,}75 - 1{,}7| = 0{,}05$. Já o erro relativo vale:

$$\mathrm{ER}(y) = \frac{0{,}05}{1{,}7} \approx 0{,}029412$$

8 Erros de arredondamento

Uma das vantagens de trabalharmos com sistemas numéricos finitos com precisão fixa é que podemos conhecer antecipadamente qual é o erro máximo que podemos cometer ao arredondar um número. Este limitante é geralmente dado em função de um parâmetro do sistema denominado *precisão de máquina*.

Com efeito, seja y o menor número real representável de modo **exato** em um formato numérico que seja maior do que 1. Definimos a **precisão de máquina** ε deste sistema como sendo igual a $y - 1$ ou, de modo equivalente, $\varepsilon = \beta^{1-t}$.

Exemplo. Tomando um sistema decimal com $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$, temos que:

$$(1)_{10} = +(1{,}00)_{10} \times 10^0.$$

Adicionando uma unidade no dígito menos significativo do 1, obtemos o y da definição que acabamos de dar:

$$y = +(1{,}01)_{10} \times 10^0.$$

Portanto, a precisão de máquina deste formato é $y - 1 = 0{,}01$.

Exemplo. (Precisão dupla) Para o caso do formato numérico de 64 bits da `numpy`, onde $t = 53$, a precisão de máquina é:

[14]:
```
eps = 2.0**(1 - 53)
eps
```

[14]:
```
2.220446049250313e-16
```

A função `np.finfo` também pode nos informar este parâmetro:

[15]:
```
np.finfo(np.float64).eps
```

[15]:
```
2.220446049250313e-16
```

Usando arredondamento para o mais próximo, observe que por definição teremos:

$$1 + \varepsilon/2 == 1$$

De fato,

[16]:
```
(1 + eps/2) == 1
```

[16]:
```
True
```

ou seja,

O fato mais importante que mencionaremos nesta seção, ainda que sem demonstração, é que o erro relativo do arredondamento para o mais próximo de números **normais** em um sistema de ponto flutuante arbitrário obedece sempre à desigualdade:

$$\frac{|\mathrm{RN}(x) - x|}{|x|} \leq \frac{1}{2}\varepsilon.$$

8.0.1 Overflow e underflow

Muitas vezes, quando arredondamos um número **normal** x, ele pode resultar em um valor que em módulo estará mais próximo dos números subnormais. Nestes casos, diremos que ocorre um **undeflow (gradual)**.

Outras vezes, o valor arredondado pode ser tão pequeno que estará mais próximo do zero do que de qualquer subnormal, e este efeito é denominado de **underflow abrupto** ou simplesmente **underflow**. Se $\alpha = \beta^{e_{\min}-t+1}$ é o menor subnormal positivo do sistema, o resultado de $\mathrm{RN}(x)$ será $+0$ sempre que tivermos $0 < x \leq \alpha/2$. Teremos -0 quando x estiver se aproximando por valores menores do que zero.

O efeito oposto ao *underflow* é conhecido como **overflow**. Por exemplo, teremos $\mathrm{RN}(x) = +\infty$ quando:

$$\begin{aligned} x &\geq \Omega + \frac{1}{2}\beta^{1-t} \times \beta^{e_{\max}} \\ &= \left(\beta - \frac{1}{2}\beta^{1-t}\right) \times \beta^{e_{\max}}, \end{aligned}$$

onde Ω é o maior normal positivo e $\beta^{1-t} \times \beta^{e_{\max}}$ é o tamanho do "vazio" entre Ω e seu sucessor, ainda que não existente no sistema de ponto flutuante.

Exemplo. Considere um sistema decimal com $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$. Tomemos $x = (1{,}10)_{10} \times 10^{-1}$ e $y = (7{,}00)_{10} \times 10^{2}$, quanto valem $x \div 300$ e $2 \times y$?

Com o auxílio do Python, obtemos o seguinte:

[17]:
```
x = 1.10 * 10**(-1)
y = 7.00 * 10**(2)

div  = x/300.0
prod = 2.0*y

print(f"x/2 = {div}")
print(f"2*y = {prod}")
```

```
x/2 = 0.0003666666666666667
2*y = 1400.0
```

Mas os limites desse sistema são

[18]:
```
beta = 10
t = 3
emin = -1
emax = 2
spf_limites(beta,t,emin,emax)
```

```
Menor normal positivo: 0.1
Maior normal positivo: 999.0
Menor subnormal positivo: 0.001
Maior subnormal positivo: 0.099
```

Com isso, o valor arredondado da divisão de x por 2 será:

[19]:
```
if div <= 0.001/2:
    div = 0.0
print(f"x/2 = {div:1.2e}")
```

```
x/2 = 0.00e+00
```

e, assim, presenciamos um *underflow* abrupto.

No caso do produto de 2 por y, o vazio entre o maior normal e seu sucessor fictício é:

[20]:
```
vazio = beta**(1-t) * beta**(emax)
vazio
```

[20]:
```
1.0
```

Usando isso, arredondamos o resultado da multiplicação:

[21]:
```
if prod >= 999.0 + vazio/2.0:
    prod = np.inf
print(f"2*y = {prod}")
```

```
2*y = inf
```

o que indica um *overflow*.

Observe que se $2 \times y$ resultar em 999,5 ou mais, teremos como resultado $+\infty$.

9 O sistema de precisão dupla

Um grupo de trabalho intitulado IEEE754 foi formado na década de 70 para definir padrões para a implementação de sistemas de ponto flutuante, a serem adotados pela indústria de computadores. Na ocasião, participavam representantes do IEEE, da Apple, Zilog, DEC, Intel, Hewlett-Packard, Motorola e National Semiconductor. Ao final dos trabalhos, foram estabelecidos:

- padrões de sistemas binários e decimais para números em ponto flutuante
- operações aritméticas de ponto flutuante: adição, subtração, multiplicação, divisão, raiz quadrada, comparação e outras
- critérios para conversão:
  - de inteiros para números em ponto flutuante
  - entre diferentes formatos de ponto flutuante
  - de números em ponto flutuante para outros tipos (por exemplo, caracteres e cadeias de caracteres)
- tratamento de exceções: *Not a Number (NaN)*, *overflow*, *underflow*, etc

Ficaram definidos cinco padrões para a implementação de sistemas de numeração em ponto flutuante:

- Três binários ocupando 32, 64 e 128 bits cada
- Dois decimais ocupando 64 e 128 bits cada

Dentre estes, o padrão mais usado em métodos numéricos é denominado de **sistema de precisão dupla**. É uma representação de um sistema binário que possui ao todo 64 *bits* distribuídos do seguinte modo:

- 1 bit para o sinal (0 := positivo, 1 := negativo)
- 11 bits para o expoente
- 53 bits para a mantissa (levando em consideração um bit implícito, que explicaremos mais à frente)

A figura a seguir ilustra esta configuração:

Observe que o bit implícito não aparece no esquema apresentado acima. Com este bit extra, o formato de precisão dupla consiste na verdade de 65 bits, mas apenas 64 deles realmente ocuparão espaço na memória.

Este padrão está implementado no pacote `numpy` na forma do tipo numérico `float64`. Vejamos algumas características deste tipo usando a função `finfo`. A quantidade de bits ocupados por um número `float64` é:

[22]:
```
import numpy as np # precisamos carregar a numpy ao menos uma vez

np.finfo(np.float64).bits
```

[22]: 64

Já a quantidade de bits destinados ao expoente:

[23]:
```
np.finfo(np.float64).iexp
```

[23]: 11

e o número de bits da mantissa, desconsiderando o bit implícito:

[24]:
```
np.finfo(np.float64).nmant
```

[24]: 52

conforme esperado.

9.0.1 Armazenamento do expoente

Nos padrões binários do IEEE o expoente é armazenado utilizando a mesma base da mantissa, ou seja, a base binária. Embora tenhamos especificado os valores de $e_{\min}$ e $e_{\max}$ nos exemplos fictícios que lidamos anteriormente, na prática, o conjunto de valores possíveis para o expoente dependerá da quantidade de bits reservada para armazená-los, aqui denotada por E.

Por exemplo, em um sistema binário com $E = 2$, teremos apenas $2^2 = 4$ valores distintos de expoente. Observe que, pelo fato da base ser binária, a quantidade de expoentes será sempre par. Com isso, não conseguiremos ter um intervalo perfeitamente simétrico de expoentes, pois o zero deve ser incluído. É por isso que nos sistemas binários do IEEE sempre temos um expoente negativo a menos do que os expoentes positivos.

Por questões de desempenho, o expoente é armazenado como um inteiro sem sinal, usando a técnica de *polarização*. Com efeito, suponha um sistema binário com E bits reservados para o expoente. Neste caso, há 2^E expoentes possíveis, os quais podem ser:

$$\mathscr{E} = \{-(2^{E-1}-1), -(2^{E-1}-2), \ldots, -2, -1, 0, +1, +2, \ldots, +(2^{E-1}-1), +2^{E-1}\}.$$

Dado um expoente $e \in \mathscr{E}$, definimos a função de polarização $P\colon \{0, 1, \ldots, 2^E - 1\} \to \mathscr{E}$ por:

$$P^{-1}(e) = e + b,$$

onde $b = 2^{E-1} - 1$ é denominado *fator de polarização* ou *viés*. Temos então o *valor real* e do expoente e seu *valor armazenado* $P^{-1}(e)$.

Exemplo. Sejam $E = 3$ e $\beta = 2$. Neste caso, temos $2^3 = 8$ expoentes distintos:

$$\mathscr{E} = \{-3, -2, -1, 0, +1, +2, +3, +4\},$$

e o fator de polarização é $b = 2^{3-1} - 1 = 3$. Assim, o expoente $e = -2$ será armazenado como $P^{-1}(-2) = -2 + 3 = 1$, que pode ser escrito em binário usando 3 bits como $(001)_2$.

9.0.2 Bit implícito

Observe que, no caso de sistemas binários, a mantissa de um número normal sempre possui a forma $(1,d_1d_2\ldots d_{t-1})_2$ e de um subnormal é $(0,d_1d_2\ldots d_{t-1})_2$. Por este motivo, o bit d_0 é armazenado apenas de modo *implícito*. Com isso, ganhamos um bit extra de precisão. Este bit é denominado *bit implícito*.

O valor do bit implícito (se 0 ou 1) de um número $x = \pm m \times 2^e$ será indicado pelo valor armazenado no expoente. Quando $P^{-1}(e)$ for igual a *zero*, m será igual a $(0,d_1d_2\ldots d_{t-1})_2$, caso contrário, x possuirá a mantissa de número normal.

9.0.3 Exceções e expoentes disponíveis

É importante lembrar que alguns resultados de operações aritméticas com números reais não podem ser representadas como números normais ou subnormais. Isto acontece quando nos deparamos com $\pm\infty$ ou indeterminações, geralmente indicadas pela expressão *Not a Number (NaN)*, tais como $0/0$, $\pm\infty/\pm\infty$, $0\times\pm\infty$, etc. A ocorrência destas situações será indicada armazenando-se no expoente o valor:

$$2^E - 1 = (\underbrace{11\ldots1}_{E \text{ bits}})_2.$$

Teremos $\pm\infty$ quando, além do expoente acima, m for igual a $\pm(d_0,00\cdots0)_2$, para qualquer valor de d_0. Caso contrário, significará um NaN.

Com isso, os *valores disponíveis* de expoente ficam restritos ao intervalo:

$$e_{\min} = -(2^{E-1}-2) \leq e \leq e_{\max} = +(2^{E-1}-1).$$

Exemplo. No sistema de precisão dupla, onde $E = 11$, temos $e_{\min} = -(2^{11-1}-2) = -1022$ e $e_{\max} = +(2^{11-1}-1) = +1023$. De fato,

[25]:
```
np.finfo(np.float64).minexp
```

[25]:
```
-1022
```

[26]:
```
np.finfo(np.float64).maxexp
```

[26]:
```
1024
```

O que é exibido acima na verdade é uma unidade a mais do que o verdadeiro $e_{\max}$.

A representação do $+\infty$ é:

e na `numpy`

[27]:
```
np.inf
```

[27]:
```
inf
```

Já as indeterminações tipo NaN possuem a forma:

ou

e em Python valem:

[28]:
```
np.nan
```

[28]:
```
nan
```

Exemplo. Vejamos algumas operações aritméticas em precisão dupla que retornam $+\infty$ ou NaN.

[29]:
```
x = np.float64(1.0)
y = np.float64(10**(-309))
x/y
```

```
/tmp/ipykernel_11850/240944173.py:3: RuntimeWarning: overflow␣
↪encountered in
double_scalars
  x/y
```

[29]:
```
inf
```

[30]:
```
x = np.float64(0.0)
y = np.inf
x*y
```

```
/tmp/ipykernel_11850/1791320970.py:3: RuntimeWarning: invalid value␣
↪encountered
in double_scalars
  x*y
```

[30]:
```
nan
```

10 Aritmética de ponto flutuante

Agora, consideraremos as operações aritméticas dos números reais no contexto dos sistemas de ponto flutuante. Dados $x, y \in \mathbb{R}$, definimos

- $x \oplus y = \text{fl}(\text{fl}(x) + \text{fl}(y))$
- $x \ominus y = \text{fl}(\text{fl}(x) - \text{fl}(y))$
- $x \otimes y = \text{fl}(\text{fl}(x) \times \text{fl}(y))$
- $x \oslash y = \text{fl}(\text{fl}(x) / \text{fl}(y))$

11 Aritmética de ponto flutuante

De modo geral, o processo para a realização de uma operação aritmética de ponto flutuante obedece à sequência:

1. arredonde os operandos x e y, obtendo os valores $\text{fl}(x)$ e $\text{fl}(y)$
2. realize a operação aritmética desejada de modo exato usando os valores de $\text{fl}(x)$ e $\text{fl}(y)$

3. arredonde o resultado do passo 2

Para realizar a adição ou a subtração, inicialmente, é necessário igualar os expoentes de x e y, sempre mantendo inalterado o maior dos expoentes. Isto não é necessário para as operações de multiplicação e divisão,.

12 Exemplo

Suponha um sistema decimal com $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$.
Calcule $(350)_{10} \oplus (1{,}5)_{10}$.

13 Exemplo

Suponha um sistema decimal com $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$.
Calcule $x \oslash y$ com $x = (1{,}625)_{10}$ e $y = (1{,}7)_{10}$.

14 float vs numpy.float

O comportamento do `float` da Python é muitas vezes diferente do estabelecido no padrão IEEE.

[31]:
```
import numpy as np
a = np.float64(1.0) # formato IEEE754
b = np.float64(0.0) # formato IEEE754
a/b
```

```
/tmp/ipykernel_11850/4119172958.py:4: RuntimeWarning: divide by zero␣
 ↪encountered
in double_scalars
  a/b
```

[31]:
```
inf
```

15 float vs numpy.float

O comportamento do `float` da Python é muitas vezes diferente do estabelecido no padrão IEEE.

[32]:
```
a = 1.0
b = 0.0
a/b
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call␣
 ↪last)
Input In [32], in <cell line: 3>()
      1 a = 1.0
```

```
      2 b = 0.0
----> 3 a/b

ZeroDivisionError: float division by zero
```

16 Perda de precisão

Em diversas ocasiões, podemos ter perda de precisão em operações de ponto flutuante. É importante conhecermos duas situações clássicas:

- soma de número grande a número pequeno
- subtração de números muito próximos

17 Adição de número grande a pequeno

Considere um sistema decimal com $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$.

Calculemos $400 \oplus 0{,}1$

18 Subtração de números próximos

Considere um sistema decimal com $t = 3$, $e_{\min} = -1$ e $e_{\max} = 2$.

Calculemos $9{,}34 \ominus 9{,}33$

19 Outra subtração de números próximos

Considere o problema de calcular o valor da função abaixo para $x = 10^{-k}$, $k = 0, 1, \ldots, 12$.

$$f(x) = \frac{1 - \cos x}{\operatorname{sen}^2 x}$$

20 Outra subtração de números próximos

Observe o que acontecerá na prática.

[ ]:
```
def f(x):
    return (1.0 - np.cos(x))/(np.sin(x)*np.sin(x))

for i in range(13):
    x = 10.0**(-i)
    print("%.15f %.15f" % (x, f(x)))
```

21 Outra subtração de números próximos

Resolvemos este problema multiplicando ambos o numerador e o denominador de f por $1+\cos x$, obtendo

$$h(x) = \frac{1}{1+\cos x}$$

22 Outra subtração de números próximos

Agora

[ ]:
```
def h(x):
    return 1.0/(1.0 + np.cos(x))

for i in range(13):
    x = 10.0**(-i)
    print("%.15f %.15f %.15f" % (x, f(x), h(x)))
```

23 Análise de erros de arredondamento

Considerando o arredondamento para o mais próximo, dado $x \in \mathbb{R}$ tal que $\text{fl}(x)$ é normal, já sabemos neste momento que

$$\frac{|\text{fl}(x) - x|}{|x|} \leq \frac{1}{2}\varepsilon$$

Defina $\delta = [\text{fl}(x) - x]/x$. Então, pela desigualdade anterior, $|\delta| \leq \frac{1}{2}\varepsilon$.

Isolando $\text{fl}(x)$ na definição de δ, temos que

$$\text{fl}(x) = x(1+\delta).$$

24 Análise de erros de arredondamento

Agora, denote por $\odot$ uma das operações $+$, $-$, $\times$ ou $/$ em ponto flutuante.
Podemos escrever

$$\begin{aligned} x \odot y &= \text{fl}\left(\text{fl}(x) \cdot \text{fl}(y)\right) && (30)\\ &= \text{fl}\left(x(1+\delta_1) \cdot y(1+\delta_2)\right) && (31)\\ &= \left(x(1+\delta_1) \cdot y(1+\delta_2)\right)(1+\delta_3) && (32) \end{aligned}$$

Isto pode ser utilizado para obtermos limitantes para os erros surgidos em operações aritméticas de ponto flutuante, supondo números normais e o arredondamento para o mais próximo, como veremos a seguir.

24.0.1 Exemplo

Sejam x e y números reais positivos.
Estime o erro relativo que ocorrerá na operação $x \oplus y$ em um sistema de ponto flutuante arbitrário

Com efeito,

$$\begin{aligned} x \oplus y &= [x(1+\delta_1) + y(1+\delta_2)](1+\delta_3) \\ &= [(x+y) + \delta_1 x + \delta_2 y](1+\delta_3) \\ &\approx (x+y) + x(\delta_1+\delta_3) + y(\delta_2+\delta_3), \end{aligned}$$

supondo $\delta_1\delta_3 = \delta_2\delta_3 \approx 0$.

24.0.2 Exemplo

Assim, o erro relativo será:

$$\begin{aligned} \mathrm{ER}(x \oplus y) &= \frac{|(x+y) - (x \oplus y)|}{|(x+y)|} \\ &\approx \left| \frac{x(\delta_1+\delta_3) + y(\delta_2+\delta_3)}{(x+y)} \right| \end{aligned}$$

24.0.3 Exemplo

Tomemos $\delta = \max\{|\delta_1+\delta_3|, |\delta_2+\delta_3|\}$. Então,

$$\begin{aligned} \mathrm{ER}(x \oplus y) \approx \left| \frac{x(\delta_1+\delta_3) + y(\delta_2+\delta_3)}{(x+y)} \right| &\leq \left| \frac{x\delta + y\delta}{x+y} \right| \\ &= |\delta| \\ &\leq \varepsilon \end{aligned}$$

25 Saiba mais

- Este material foi preparado com o auxílio de nosso livro texto, Análise Numérica, de Burden et al., e de uma excelente referência complementar, o livro do Numerical Mathematics and Computing, dos professores Cheney e Kincaid.
- A descrição sobre precisão foi inteiramente baseada no texto disponibilizado aqui e no livro: ATKINSON, Kendall. An introduction to numerical analysis. John wiley & Sons, 1991.
- Na prática, a análise dos erros de arredondamento e truncamento torna-se necessária apenas quando desenvolvemos um novo método numérico. Ao usarmos métodos já estabelecidos, na maioria das vezes, nos apoiamos sobre resultados anteriores.

Vicente Helano
UFCA | Centro de Ciências e Tecnologia