Erros em Processos Numéricos

J. A. Salvador

Dizia um grande professor meu que " Um professor nunca erra, raramente se engana, e quando se engana a culpa é do aluno"

Introdução

Erros em processos numéricos ocorrem em todos os Tópicos:
Zeros de Funções
Sistemas Lineares e não lineares
Interpolação
Ajuste de curvas, mmc.
Integração numérica
Soluções Numéricas de EDO

Conteúdo do capítulo

Erros em processos numéricos
Problema real - Erro na fase de elaboração do Modelo Matemático
Solução - Erro na fase de resolução - Método Numérico
Implementação computacional: mudança de base, precisão dos dados de entrada/ forma de representação/operações efetuadas, arredondamento e truncamento.
Propagação dos erros.
Erro absoluto, Erro relativo, limitantes.

Introdução

A base para o desenvolvimento dos métodos numéricos de aproximação que serão tratados está no estudo dos erros. Muitos tipos de problemas são estudados numericamente, portanto apresentamos alguns conceitos básicos da teoria dos erros e certos elementos matemáticos sobre os quais se fundamenta grande parte dos Métodos de Análise Numérica.

No estudo do Cálculo Diferencial e Integral, Geometia Analítica e na Matemática Universitária Básica são introduzidos conceitos que auxiliam na compreensão e resolução dos mais diversos problemas científicos. No entanto, muitas vezes não podemos aplicá-los imediatamente em um problema que exige uma solução numérica. A Análise Numérica procura prover os meios

necessários para que métodos numéricos possam ser aplicados a problemas que não possuem solução analítica ou que a mesma é de difícil obtenção e, que se tornem facilmente executáveis numericamente.
Podemos dizer assim, que a Análise Numérica é o ramo da Matemática que desenvolve técnicas para a resolução de problemas do Cálculo Numérico.

Se no mundo de hoje a prática industrial necessita da engenharia avançada, que por sua vez, necessita da engenharia básica que necessita dos conhecimentos das ciências e da matemática avançada, esta última e que por sua vez necessita da matemática básica.
Os erros que ocorrem nos processos numéricos, podem ser oriúndos desde a modelagem do problema real, na fase de elaboração do Modelo Matemático, na fase da escolha do Método Numérico, na fase de resolução. Na implementação computacional: mudança de base, precisão dos dados de entrada/ forma de representação/operações efetuadas e erros de arredondamento e/ou de truncamento
Além disso temos a propagação dos erros.
Nem sempre o Erro Absoluto (EA) é uma boa medida do erro, é necessário calcularmos o Erro Relativo (ER), e mesmo assim, nem sempre os conseguimos, logo é necessário calcularmos principalmente os limitantes para o erro.

Exemplos

Expor um problema de crescimento exponencial cuja solução $x(t) = x(0)\,\mathbf{e}^{(k\,t)}$
Propor aos estudantes para escreverem o valor do número de Euler e no quadro.

Abordemos um caso simples do processo de cálculo de limite visto numa disciplina de Cálculo Diferencial e Integral 1 .

$$\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x$$

Do cálculo elementar sabemos que existe o limite e também que seu valor é um número irracional **e**.
De fato, para calculá-lo podemos usar a função logarítmica, que destroi o expoente x, assim,

$$\ln\left(\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x\right) = \lim_{x \to \infty} \ln\left(\left(1 + \frac{1}{x}\right)^x\right) = \lim_{x \to \infty} x \ln\left(1 + \frac{1}{x}\right) = \lim_{x \to \infty} x\left(\ln\left(1 + \frac{1}{x}\right) - \ln(1)\right)$$

$$= \lim_{x \to \infty} \frac{\ln\left(1 + \frac{1}{x}\right) - \ln(1)}{\frac{1}{x}} = \mathrm{D}(\ln)(1) = 1.$$

Assim, se o

$$\ln\left(\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x\right) = 1$$

o

$$\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x = e.$$

Evidentemente, foi definido esse número para que se possa usá-lo, e para isso é preciso conhecer seu valor. Com definição acima, dada para o número *e*, encontra-se dificuldade de atingir o valor exato, tanto pela complexidade das operações a efetuar como pela impossibilidade de atingir o limite. Apela-se, então, para um processo de cálculo mais simples, que fornece um valor aproximado desse número dentro de um certo grau de exatidão considerado satisfatório. Entramos então no domínio da Análise Numérica.

Como vamos lidar com uma aproximação, é preciso estabelecer critérios de avaliar o seu grau de exatidão. Um critério intuitivo é o de obter um valor aproximado cujos algarismos coincidem com os do valor exato, a partir da esquerda até o de certa casa decimal.

Calculando o limite acima para *x* de 100 em 100 a partir de 1 até 1001 temos

```
> for x from 1 to 1001 by 100 do print( [x,  (1. + 1/x)^x]) od;
```

[1, 2.]

[101, 2.704945951]

[201, 2.711550420]

[301, 2.713780007]

[401, 2.714900629]

[501, 2.715573948]

[601, 2.716024604]

[701, 2.716346406]

[801, 2.716585983]

[901, 2.716775093]

[1001, 2.716925290]

```
> restart;
```

Compare com os valores escritos no quadro.

****

```
>
```

Evidentemente foi definido esse número para que se possa usá-lo, e para isso é preciso conhecer o seu valor. Com a definição acima é difícil encontrar o s seu valor exato, tanto pela complexidade das operações como pela impossibilidade de atingir o limite, o que nos leva a apelar para um processo numérico que nos fronece um valor aproximado dentro de um certo grau de precisão.

```
> e := evalf(exp(1), 40);
```

$e := 2.7182818284590452353602874713526624977 57$

```
>
```

Outra forma de calcular o valor de *e* é usando o que estudamos em Cálculo Diferencial e Séries. De fato, desenvolvendo a série de potências

```
> exp(x) = Sum( x^n/n!, n=0..infinity);
```

$$\mathbf{e}^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

e fazendo $x = 1$ temos;

```
> exp(1) = Sum( 1^n/n!, n=0..infinity);
```

$$\mathbf{e} = \sum_{n=0}^{\infty} \frac{1}{n!}$$

É evidente que podemos calcular o valor aproximado de *e* truncando a série numérica obtida num determinado termo. Por exemplo, se considerarmos

```
> exp(1)[aprox] = Sum( 1^n/n!, n=0..10);
```

$$(\mathbf{e})_{aprox} = \sum_{n=0}^{10} \frac{1}{n!}$$

teremos

```
> exp(1)[aprox] = sum( 1.^n/n!, n=0..10);
```

$$(\mathbf{e})_{aprox} = 2.718281801$$

e cometeremos o erro

```
> E[10] = Sum( 1^n/n!, n=11..infinity);
```

$$E_{10} = \sum_{n=11}^{\infty} \frac{1}{n!}$$

A princípio também não temos idéia da ordem do erro cometido. Entretanto, observando que

$$\frac{1}{11!} = \frac{1}{10\,(11)}, \qquad \frac{1}{12!} = \frac{1}{10!\,11\,(12)}, \qquad \frac{1}{13!} = \frac{1}{10!\,11\,(12)\,(13)}, \ldots$$

temos que

$$\frac{1}{11!} + \frac{1}{12!} + \frac{1}{13!} + \ldots\ldots = \frac{1}{10!}\left(\frac{1}{11} + \left[\frac{1}{11}\right]\left[\frac{1}{12}\right] + \left[\frac{1}{11}\right]\left[\frac{1}{12}\right]\left[\frac{1}{13}\right] + \ldots.\right)$$

$$\frac{1}{11!} + \frac{1}{12!} + \frac{1}{13!} + \ldots\ldots < \frac{1}{10!}\left(\frac{1}{11} + \frac{1}{11^2} + \frac{1}{11^3} + \ldots.\right)$$

Como a série geométrica $\frac{1}{11} + \frac{1}{11^2} + \frac{1}{11^3} + \ldots.$ converge para $\frac{1\,(1)}{11\left(1 - \frac{1}{11}\right)} = \frac{1}{10}$

Um limitante para o erro cometido ao truncarmos a série para $n = 10$ é

```
> Erro < 1/10! * (1/10); Erro < 1/10! * (1./10);
```

$$Erro < \frac{1}{36288000}$$

$$Erro < 0.2755731922\ 10^{-7}$$

```
>
```

O erro cometido é da ordem de $10^{(-7)}$. Com uma aproximação pré-fixada conseguimos o valor aproximado de *e* com um certo número de casas decimais corretas. Neste caso, garantimos que o valor de e_{aprox} está correto até a sexta casa decimal!

De fato, o computador pode nos dar uma aproximação para o valor de e com 10 dígitos;

```
> 'e' = evalf(exp(1));
```

$$e = 2.718281828$$

e até a sexta casa decimal, podemos garantir que $e = 2.718281$.

```
>
```

Ex. 1) Explore agora o problema inverso. Estabelecendo o número de casas decimais corretas (m), mostre que para determinar uma aproximação para a série $\sum_{n=0}^{\infty} \frac{1}{n!}$, de modo que o erro seja menor ou igual a $10^{(-m)}$ devemos ter que devemos expandir a série até N satisfazendo a inequação $\left[\frac{N+2}{(N+1)!\,(N+1)}\right] < 10^{(-m)}$

Ex. 2) Qual deve ser o valor de N de modo que o valor aproximado de $e = \sum_{n=0}^{N} \frac{1}{n!}$ tenha 4 casas decimais corretas?

Devemos ter $\frac{N+2}{(N+1)!\,(N+1)} < 10^{(-4)}$, logo tentamos

```
> solve( (N+2)/((N+1)!*(N+1)) < 10^(-6), N);
```

ou então, testamos valores de N até encontrar um que satisfaça a inequação acima

Também podemos definir uma função de N e tentar encontrar N tal que $f(N) < 10^{(-4)}$

```
> f := N -> (N+2)/((N+1)!*(N+1)) ;
```

$$f := N \to \frac{N+2}{(N+1)!\,(N+1)}$$

```
> 'f(1)' = evalf(f(1)); 'f(2)' = evalf( f(2));
```

$$f(1) = 0.7500000000$$

$$f(2) = 0.2222222222$$

```
> 'f(6)' = evalf(f(6)); 'f(7)' = evalf( f(7));
```

$$f(6) = 0.0002267573696$$

$$f(7) = 0.00002790178571$$

```
>
```

Concluímos que para $N = 7$ tempos que $\sum_{n=0}^{N} \frac{1}{n!}$ aproxima o valor de e com 4 casas decimais precisas.

```
>
```

Ex. Área de um círculo de raio 10 cm

```
> A3 := evalf( 10^2 *Pi, 3);
```

Considerando Pi com 3 casas

$$A3 := 314.$$

Considerando π com 5 casas

```
> A := evalf(10^2*Pi, 5);
```

$$A := 314.16$$

Considerando π com 10 casas, default do Maple V

```
> A := evalf( 10^2 *Pi);
```

$$A := 314.1592654$$

Considerando π com 25 casas

```
> A := evalf( 10^2 * Pi, 25);
```

$$A := 314.15926535897932384626433$$

O que você pode concluir? Como os valores da área foram obtidos diferentemente um do outro?

Vejamos agora alguns cálculos com o Maple, e tente refazê-los usando simples calculadoras, em seguida cheque-os com valores obtidos usando o computador.

```
> S1 := Sum( 0.5, i=1..30000); S1 := sum( 0.5, i=1..30000);
```

$$S1 := \sum_{i=1}^{30000} 0.5$$

$$S1 := 15000.$$

```
> S2 :=  Sum(0.11, i=1..30000); S2 := evalf( sum(0.11,
  i=1..30000), 15);
```

$$S2 := \sum_{i=1}^{30000} 0.11$$

$$S2 := 3300.$$

Sistemas Binário e Decimal

"No meu curso de cálculo Numérico, há 10 tipos de pessoas: as que entendem o sistema binário; e as que não entendem!"

O número 130153 na base decimal é escrito na base 10 como uma combinação das potências de 10. De fato;

$$130153 = 3 * 10^0 + 5 * 10 + 1 * 10^2 + 0 * 10^3 + 3 * 10^4 + 1 * 10^5$$

Na base 2 teríamos

```
> convert(130153, binary);
```

$$11111110001101001$$

Enquanto que o número 117 na base 2 é escrito como;

$$(1110101) = 1*2^0 + 0 * 2 + 1 * 2^2 + 0 * 2^3 + 1*2^4 + 1 * 2^5 + 1 * 2^6$$

Conferindo no Maple, temos que

```
> convert( 1110101, decimal, binary);
```

$$117$$

Para converter um número da representação binária para decimal, podemos colocar em evidência o número 2 acima e

$$1 + 2\,(0 + 2\,(1 + 2\,(0 + 2\,(1 + 2\,(1 + 2)))))$$

Esta decomposição equivale a dividirmos o número da base decimal por 2, o seu quociente por 2, e assim sucessivamente até que o resto seja menor do que 2.

```
> 1 + 2* ( 0 + 2*(1 + 2*(0 + 2*(1 + 2*(1 + 2)))));
```

117

```
> ( 0 + 2*(1 + 2*(0 + 2*(1 + 2*(1 + 2)))));
```

58

```
> (1 + 2*(0 + 2*(1 + 2*(1 + 2))));
```

29

```
> (0 + 2*(1 + 2*(1 + 2)));
```

14

```
> (1 + 2*(1 + 2));
```

7

```
> (1 + 2);
```

3

```
>
```

Em seguida basta tomarmos o resultado final seguido dos restos obtidos na sequência inversa. De fato;

$\frac{117}{2} = 58$ com resto 1, $\frac{58}{2} = 29$ com resto 0, $\frac{29}{2} = 14$ com resto 1, $\frac{14}{2} = 7$ com resto 0, $\frac{7}{2} = 3$ com resto 1, $\frac{3}{2} = 1$ com resto 1.

Assim, o quociente final 1 é seguido dos restos obtidos nas ordem inversa fica 1 1 1 0 1 0 1. Realmente, verificamos que

```
> convert( 117, binary);
```

1110101

Um número *n* escrito numa base β geralmente é representado por

$$n = d_j\, d_{j-1}\, d_{j-2} \ldots d_2\, d_1\, d_0\, d_{-1}\, d_{-2} \ldots d_{-j}$$

Exemplos de conversão da base 10 para binária:

```
> O[10] = convert(0, decimal, binary);
```

$O_{10} = 0$

```
> '1' = convert( 1, decimal, binary);
```

$1 = 1$

```
> '2' = convert( 2, binary);
```

Obs: $0\,2^0 + 1\,(2)$

$2 = 10$

```
> '23'[2] = convert(23, binary);
```

Obs: $1\,2^0 + 1\,(2) + 1\,2^2 + 0\,2^3 + 1\,2^4$

$23_2 = 10111$

```
> '231'[2] = convert(231, binary);
```

Obs:

$1\,2^0 + 1\,(2) + 1\,2^2 + 0\,2^3 + 0\,2^4 + 1\,2^5 + 1\,2^6 + 1\,2^7$

$231_2 = 11100111$

```
> '0.1875'[2] = convert(0.1875, binary);
```

Obs:

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

$$0.1875_2 = 0.001100000000$$

```
> '0.46875'[2] = convert(0.46875,binary);
```

$$0.46875_2 = 0.01111000000$$

```
> '13.25'[2] = convert(13.25, binary);
```

$$13.25_2 = 1101.010000$$

```
> '(0.2)'[2] = convert(0.2, binary);
```

$$0.2_2 = 0.001100110011$$

```
> '(7)[2]' = convert(7, binary);
```

$$7_2 = 111$$

```
> '(25)[2]' = convert( 25, binary);
```

$$25_2 = 11001$$

```
> '(11)[2]' = convert(11, binary);
```

$$11_2 = 1011$$

```
> '(101)[2]' = convert(101,  binary);
```

$$101_2 = 1100101$$

```
> '(101)[10]' = convert(101, decimal, binary);
```

Obs: $1\,2^0 + 0\,(2) + 1\,2^2$

$$101_{10} = 5$$

```
> '23'[8] = convert(23, octal); # 7*8^'0' + 2*8^(1)
```

$$23_8 = 27$$

```
> '347' = convert(347,  binary);
```

$$347 = 101011011$$

```
> '125' = convert(125,  binary);
```

$$0.125 = 1111101$$

No Maple também podemos converter do sistema hexagesimal para a base 10.

```
> 'convert(1*A, decimal, hex)' = convert(`1A`, decimal, hex);
```

$$\text{convert}(A, \textit{decimal}, \textit{hex}) = 26$$

Números decimais com representação finita podem gerar números com representação binária infinita.

```
> '(0.1)[2]' = convert(0.1,  binary);
```

$$0.1_2 = 0.0001100110011$$

No caso de números decimais o procedimento é multiplicar a parte decimal por 2. A seguir continuar multiplicando a parte decimeal do resultado obtido por 2. O número na base 2 será então obtido tomando-se a parte inteira do resultado de cada multiplicação.
e transformando os números da base 10 para a base 2, obtemos

```
> '(0.75)[10]' = convert( 0.75, binary);
```

$$0.75_{10} = 0.1100000000$$

Aumentando o número de dígitos para 30 no Maple, temos

```
> Digits := 40:
> '0.1' = convert(0.1,  binary);
```

$$0.1 = 0.0001100110011001100110011001100110011001100110011001100$$

```
> '0.11' = convert( 0.11, binary);
```

$$0.11 = 0.0001110000101000111101011100001010001111010$$

```
> '0.111' = convert( 0.111, binary);
```

$$0.111 = 0.0001110001101010011111101111100111011011001$$

```
> '17.6' = convert(17.6, binary);
```

$$17.6 = 10001.10011001100110011001100110011001100110011001100$$

```
> '14.2' = convert(14.02, binary, 50);
```

$$14.2 = 1110.00000101000111101011100001010001111010111000 01$$

A volta pode ser acometida de erro!

```
> convert(1110.00000101000111101011100001010001111010111000 01,
  decimal, binary);
```

$$14.019999999999996020960679743438959121 70$$

```
> convert( 1110, decimal, binary);
```

$$14$$

Voltemos ao default do Maple que é utilizando 10 dígitos

```
> Digits := 10:
```

Para representar um número da base β_1 para uma base β_2, primeiro mudamos o número da base β_1 para a base 10, em seguida da base 10 para a base β_2.

```
>
```

Ponto Flutuante

```
> 'exp(1) ' = evalf( exp(1));
```

$$\mathbf{e} = 2.718281828$$

```
> x := .937*10^(4); y := 0.1272*10^(2);
```

$$x := 9370.000$$

$$y := 12.7200$$

```
> (.937 + 0.001272)*10^4;
```

$$9382.720000$$

É o resultado exato! Mas arredondado

```
> Digits := 4:
> (.937 + 0.001272)*10^4;
```

$$9383.$$

ou truncado teríamos

```
> 9382*10^4;
```

$$93820000$$

```
> x*y;
```

$$119200.$$

```
> Digits := 20: # Nos dá o valor exato
> x*y;
```

$$119186.4000000$$

Arredondamento

```
> restart;
> 'exp(1)' = round(evalf(exp(1), 3));
```

$$\mathbf{e} = 3$$

```
> 'exp(1)' = round(evalf( exp(1)*10^(-3), 14)*10^3);
```

$$\mathbf{e} = 3$$

```
> 'exp(1)' = evalf(exp(1), 5);
```

$$\mathbf{e} = 2.7183$$

Truncamento

```
> Digits := 10:
> 'exp(1)' = evalf(exp(1));
```

$$\mathbf{e} = 2.718281828$$

Truncando **e** no terceiro algarismo

```
> trunc(2.718281828*10^(4))*10^(-4.);
```

$$2.718200000$$

Arredondamento

```
> floor(2.718281828*10^(4))*10^(-4.);
```

$$2.718200000$$

ou

```
> round( 2.718281828*10^(3))*10^(-3.);
```

$$2.718000000$$

```
> round(2.718281828*10^(4))*10^(-4.);
```

$$2.718300000$$

Erro absoluto

Quando se substitui um valor x por um outro valor aproximado x_{aprox}, diz-se que o erro absoluto cometido é:

$$\Delta = (x - x_{aprox})$$

Nota-se que o erro absoluto simplesmente não traduz nada se não soubermos a ordem de grandeza do valor calculado. Normalmente, admite uma tolerância (ε) para esse erro, isto é: $\Delta = x - x_{aprox} < \varepsilon$.

```
> Eax := abs( x - x[aprox]);
```

$$Eax := \left| x - x_{aprox} \right|$$

```
> EaPi := abs( pi - evalf(Pi,4));
```

$$EaPi := \left| \pi - 3.142 \right|$$

```
> 3.14 < x[aprox] ;  x[aprox] < 3.15;
>
```

$$3.14 < x_{aprox}$$

$$x_{aprox} < 3.15$$

```
> abs( Eax) < 3.15 - 3.14;
```

$$|x - x_{aprox}| < 0.01$$

>

Erro relativo

O erro relativo tem a finalidade de dar uma idéia do grau de influência do erro no valor desejado, ou seja, uma análise mais ampla se o erro obtido influi muito ou pouco no valor.
Exemplo: O erro obtido da distância da Terra à Lua (380000 km) foi de aproximadamente 1 km. Nota-se que o erro é desprezível não influi muito na distância real de que é muito grande.
Mas, se esse mesmo erro fosse obtido na distância de São Carlos a Ibaté o erro teria uma certa influência que teria que ser considerado.

O cálculo do erro relativo cometido sobre um valor x, quando este é aproximado por x[aprox] é dado por:

```
> EaTL := abs( d[TL]- d[aprox]) = 1;
```

$$EaTL := |d_{TL} - d_{aprox}| = 1$$

```
> Erx := Eax/x[aprox];
```

$$Erx := \frac{x - x_{aprox}}{x_{aprox}}$$

```
> ErTL := 1/380000.;
```

$$ErTL := 0.2631578947\ 10^{-5}$$

```
> dSCarI := 12;
```

$$dSCarI := 12$$

```
> EaSCarI := 1;
```

$$EaSCarI := 1$$

```
> ErSCarI := 1/12.;
```

$$ErSCarI := 0.08333333333$$

>

Exercícios

```
> restart;
> convert( 37, binary);
```

100101

```
> convert( 2345, binary);
```

100100101001

```
> convert( 0.1217, binary, 30);
```

0.00011111001001111011011001011100

```
> convert( 101101, binary,  decimal);
```

11000101011101101

```
> convert( 110101011, binary,  decimal);
```

11010010000000001000010011

```
> convert([2,2,1],base,3,10);
```

[7, 1]

```
> convert( [0,1,1,0,1], base, 10, 2);
```

[0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1]

$5\ 10^{0} + 2\ 10$

```
> convert( [5, 2], base, 10, 2);
```

[1, 0, 0, 1, 1]

```
> convert( [1,0,0, 1,1], base, 2, 10);
```

[5, 2]

```
> convert([0,1,1,1,1,1,1,1,0,1],base,2,10);
```

[6, 6, 7]

```
> a := convert( 0.1, binary,  8);
```

$a := 0.00011001100$

```
> b := convert( 0.11, binary, 8);
```

$b := 0.00011100000$

```
> c := convert( 0.6, binary, 20);
```

$c := 0.10011001100110011001$

```
> [a, b,c];
```

[0.00011001100, 0.00011100000, 0.10011001100110011001]

```
> restart;
> Digits := 8;
```

$Digits := 8$

```
> x := 0.7237*10^4;
  y := 0.2145*10^(-3);
  z := 0.2585*10;
```

$x := 7237.0000$

$y := 0.00021450000$

$z := 2.5850$

Escrevemos

```
> x := 0.00000007237*10^(-3);
  y := 0.2145*10^(-3);
  z := 258.5*10^(-3);
```

$x := 0.72370000\ 10^{-10}$

$y := 0.00021450000$

$z := 0.25850000$

```
> 'x+y+z' = evalf(x + y +z) ;
```

$x + y + z = 0.25871450$

```
> evalf(x+y+z);
```

0.25871450

```
> evalf(x-y-z);
```

-0.25871450

```
> evalf(x/y);
```

$0.33738928\ 10^{-6}$

```
> evalf((x*y)/z);
```

$$0.60051702\ 10^{-13}$$

```
> evalf(x*(y/z));
```

$$0.60051702\ 10^{-13}$$

```
> restart; Digits:=4;
```

$$Digits := 4$$

```
> eq1 := 0.0030 * x + 30.0000*y = 5.0010;
  eq2 := 1.0008 * x +  4.0000*y = 1.0000;
```

$$eq1 := 0.0030\,x + 30.0000\,y = 5.0010$$

$$eq2 := 1.0008\,x + 4.0000\,y = 1.0000$$

```
> solve( {eq1, eq2}, {x,y});
```

$$\{y = 0.1667, x = 0.3331\}$$

```
> convert(%, fraction);
```

$$\{y = \frac{1}{6}, x = \frac{1}{3}\}$$

```
> restart; Digits:= 4;
```

$$Digits := 4$$

```
> eq1 := 0.0030 * x + 30.0000*y = 5.0010;
  eq2 := 1.0000 * x +  4.0000*y = 1.0000;
```

$$eq1 := 0.0030\,x + 30.0000\,y = 5.0010$$

$$eq2 := 1.0000\,x + 4.0000\,y = 1.0000$$

```
> eq3 := eq1 * (-1/0.0030);
```

$$eq3 := -0.9999\,x - 9999.\,y = -1667.$$

```
>  lhs(eq3) + lhs(eq2) = rhs(eq3) + rhs(eq2);
```

$$0.0001\,x - 9995.\,y = -1666.$$

```
> solve( - 9995.99999999 * y = -1666.00, y);
```

$$0.1667$$

```
> solve( subs( y=.166666667, eq1), x);
```

$$0$$

```
>
> x := 1.25:  y := 1.75:
> round(x);
```

$$1$$

```
> trunc(x);
```

$$1$$

```
> floor(x);
```

$$1$$

```
> ceil(x);
```

$$2$$

```
> frac(x);
```

$$0.25$$

```
> round(y);
```

$$2$$

```
> trunc(y);
```

$$1$$

```
> floor(y);
```

$$1$$

```
> ceil(y);
```

$$2$$

```
> frac(y);
```

$$0.75$$

```
>
> restart;
>
```

Um sistema que opera em aritmética de ponto flutuante com t dígitos na base 10.

Seja x escrito na forma $x = f_x\, 10^e + g_x\, 10^{(e-t)}$, onde $.1 < f_x$ e $f_x < 1$, $0 \le g_x$ e $g_x < 1$.

Por exemplo se $t = 4$

```
> x := 234.57;
```

$$x := 234.57$$

$x = .2345\ 10^3 + .7\ 10^{(-1)}$

```
>
>  .2345*10^3 + 0.7 * 10^(-1); trunc(x*10)*10^(-1.);
```

$$234.5700000$$

$$234.5000000$$

$$235$$

onde $f_x = .2345$ e $g_x = .7$

No truncamento, $g_x\, 10^{(e-t)}$ é desprezado e $x_{aprox} = .2345\ 10^e$.

```
>
> restart;
```

No arredondamento, f_x é modificado para levar em consideração g_x.

```
>  round(x*10)*10^(-1.);
```

$$234.6000000$$

```
>  x := piecewise( abs(g[x]) > 1/2, f[x]*10^(e), abs(g[x]) >= 1/2,
  f[x]*10^(e) + 10^(e-t));
Warning, recursive definition of name
```

$$x := \begin{cases} f_x\, 10^e & \frac{1}{2} < |g_x| \\ f_x\, 10^e + 10^{(e-t)} & \frac{1}{2} \le |g_x| \end{cases}$$

```
>
```

De fato;

```
>  x := piecewise( abs(g[x]) > 1/2, f[x]*10^(e), abs(g[x]) >= 1/2,
  f[x]*10^(e) + 10^(e-t));
```

$$x := \begin{cases} f_{234.57}\, 10^{e} & \frac{1}{2} < \left| g_{234.57} \right| \\ f_{234.57}\, 10^{e} + 10^{(e-t)} & \frac{1}{2} \le \left| g_{234.57} \right| \end{cases}$$

>

As unidades de informação

Em Informática é muito importante considerar a capacidade de armazenamento da máquina que estamos usando. Ou seja, quando se faz algo no computador, podemos armazená-lo para poder continuar a usá-lo posteriormente. Evidentemente, quando se armazena algo, isto ocupa um certo espaço de armazenamento.

Assim como a água é medida em litro (1 litro = 1 dm^3), ou o açúcar é medido em quilos (kg), os dados de um computador são medidos em bits e bytes.

Um bit é uma informação tão pequena, que é preciso unir os pequenos bits em grupos de 8, para formar um byte. Os bits, são entendidos pelo computador em código binário, que é formado unicamente por zero e um.
Por exemplo, a letra "A", ocupa um byte para o computador e é codificada como um grupo de 8 bits, que são: "11000001".

Assim, um byte é um grupo de 8 bits e gera um caractere (letra ou símbolo do teclado).

A quantidade de espaço disponível é medida em bytes sendo os seus múltiplos:

quilobyte (Kb, equivale a 2^{10} = 1024 bytes),

```
> 1024*1024;
```

1048576

>

megabyte (Mb = 1024 Kb = 1 048 576),
gigabyte (Gb = 1024 Mb =) e
terabyte (Tb = 1024 Gb).

Por que 1 Kb equivale a 1024 bytes?

No caso do quilo e de outras medidas de nosso dia-a-dia, a estrutura numérica é construída sobre a base 10. O termo quilo representa a milhar constituída de alguma coisa. Nossa base de trabalho numérica, sendo 10, faz com que, quando a base é elevada à terceira potência, atinja a milhar exatamente com 1000 unidades.

Quando falamos em bytes, grupos de bits, não estamos falando em base 10 (dez), mas sim em uma estrutura calcada na base 10 (dois no sisitema binário), isto é, no sistema binário.

Assim, quando queremos um quilo de bytes, temos que elevar essa base a algum número inteiro, de sorte que consigamos atingir a milhar. Mas não há número inteiro possível para atingir exatamente 1000. Então, ao elevarmos a base 2 à décima potência, teremos 1024.

Com esse raciocínio agora podemos entender a seguinte tabela:

Unidade de medida	Número de caracteres	Espaço
1 byte	1	8 bits
1 quilobyte (Kb)	$2^{10} = 1024 \sim 10^3$ 1024 bytes	
1 megabyte (Mb)	$1048576 \sim 10^6$	1024 Kb
1 gigabyte (Gb)	$1073741824 \sim 10^9$	1024 Mb
1 terabyte (Tb)	$1099511627776 \sim 10^{12}$	1024 Gb

Obs.

```
> 1024*1024*1024*1024;
                                  1099511627776
> 2^10;
                                        1024
> 2^20;
                                      1048576
> 2^30;
                                    1073741824
> 2^40;
                                  1099511627776
>
```