Universidade Federal de Santa Catarina - UFSC Centro Tecnológico - CTC
Departamento de Informática e Estatística - INE

Lista de Exercícios 1 - INE5202/INE5232/INE5409
Prof. Sérgio Peters (sergio.peters@ufsc.br)

Resoluções [1]

1) (a) $(10.1011)_2 = 1 \cdot 2^1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = \mathbf{(2.6875)_{10}}$

(b) $(10.57)_{10} = (?)_2$

$$(10)_{10} = (?)_2 \longrightarrow \begin{array}{ccccc} 10 & \multicolumn{1}{|c}{2} & & & \\ \cline{2-2} 0 & 5 & \multicolumn{1}{|c}{2} & & \\ \cline{3-3} & 1 & 2 & \multicolumn{1}{|c}{2} & \\ \cline{4-4} & & 0 & 1 & \multicolumn{1}{|c}{2} \\ \cline{5-5} & & & 1 & 0 \end{array}$$

$$\longleftarrow$$

$$(10)_{10} = (1010)_2 = 1 \cdot 2^3 + 1 \cdot 2^1$$

$$(0.57)_{10} = (?)_2 \longrightarrow$$

0.57	0.14	0.28	0.56	0.12	0.24	0.48	0.96	0.92	0.84	0.68	0.36	0.72	0.44	
×2	×2	×2	×2	×2	×2	×2	×2	×2	×2	×2	×2	×2	×2	...
1.14	0.28	0.56	1.12	0.24	0.48	0.96	1.92	1.84	1.68	1.36	0.72	1.44	0.88	

$$(0.57)_{10} = (0.10010001111010\ldots)_2$$

Logo, $(10.57)_{10} = \mathbf{(1010.10010001111010\ldots)_2}$.

2)
```python
from functools import reduce

for n in [2, 3, 10, 16]:
    values = reduce(lambda x, _: x + [(n + 1) * x[-1] - 1], range(10), [1/n])
    print("\n".join("{:.30f}".format(i) for i in values))
```

- A função *reduce* é utilizada para criar o padrão requerido pelo enunciado: uma lista de números onde todos os valores estão diretamente relacionados a seus antecessores.
- Aplica-se a fórmula do enunciado no último termo da lista a cada iteração, por 10 vezes, iniciando pela inversa multiplicativa do número fornecido.
- Adiciona-se então mais casas após à vírgula para melhor visualização de como o arredondamento funciona.
- O objetivo do algoritmo é verificar como os valores são influenciados gradativamente por computações anteriores: estes perdem precisão exponencialmente pois um valor decimal não pode ser representado perfeitamente com um número finito de bits, exceto por potências de 2. A saída do algoritmo mostra duas possibilidades para cada caso, e como a quantidade de bits divergentes à divisão inicial, que já não se mostra exata, aumenta ao longo das iterações.
- Exemplo de saída (parcial) do programa:

```
0.333333333333333314829616256247
0.333333333333333259318465024990
0.333333333333333037273860099958
0.333333333333332149095440399833
0.333333333333328596381761599332
0.333333333333314385527046397328
0.333333333333257542108185589314
0.333333333333030168432742357254
0.333333333332120673730969429016
0.333333333328482694923877716064
0.333333333313930779695510864258
```

3) (a) $(3021)_{F!} = 3 \cdot 4! + 0 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! = \mathbf{(77)_{10}}$

(b) $(4321)_{F!} = 4 \cdot 4! + 3 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! = \mathbf{(119)_{10}}$

(c) $(10000)_{F!} = 1 \cdot 5! = \mathbf{(120)_{10}}$

(d) $(0.02)_{F!} = \frac{0}{2!} + \frac{2}{3!} = (\frac{2}{6})_{10} = (\frac{1}{3})_{10} = \mathbf{(0.333333\ldots)_{10}}$

[1] por Gustavo Zambonin (gustavo.zambonin@grad.ufsc.br)

(e) $(0.113)_{F!} = \frac{1}{2!} + \frac{1}{3!} + \frac{3}{4!} = (\frac{19}{24})_{10} = \mathbf{(0.791\overline{6}\ldots)_{10}}$

(f) $(321.123)_{F!} = 3 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + \frac{1}{2!} + \frac{2}{3!} + \frac{3}{4!} = (\frac{575}{24})_{10} = \mathbf{(23.958\overline{3})_{10}}$

4) (a) $(10111.1101)_2 = (0001\ 0111.1101)_2 = (17.D)_{16} = 1 \cdot 16^1 + 7 \cdot 16^0 + 13 \cdot 16^{-1} = (\frac{381}{16})_{10} = \mathbf{(23.8125)_{10}}$

(b) $(BD.0E)_{16} = (1011\ 1101.0000\ 1110)_2 = 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} = (\frac{24199}{128})_{10} = \mathbf{(189.0546875)_{10}}$

(c) $(41.1)_{10} = (?)_2 = (?)_{16}$

$$(41)_{10} = (?)_2 \longrightarrow \begin{array}{rrrrrrr} 41 & \underline{|\ 2} & & & & & \\ 1 & 20 & \underline{|\ 2} & & & & \\ & 0 & 10 & \underline{|\ 2} & & & \\ & & 0 & 5 & \underline{|\ 2} & & \\ & & & 1 & 2 & \underline{|\ 2} & \\ & & & & 0 & 1 & \underline{|\ 2} \\ & & & & & 1 & 0 \end{array}$$

$$\longleftarrow$$

$$(41)_{10} = (101001)_2 = 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^0$$

$$(0.1)_{10} = (?)_2 \longrightarrow$$

$$\begin{array}{cccccccccccc} 0.1 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 & \\ \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \ldots \\ 0.2 & 0.4 & 0.8 & 1.6 & 1.2 & 0.4 & 0.8 & 1.6 & 1.2 & 0.4 & \end{array}$$

$$(0.1)_{10} = (0.00011001100110011\ldots)_2$$

Logo, $(41.1)_{10} = (101001.0\overline{0011})_2 = (0010\ 1001.0001\ \overline{1001})_2 = \mathbf{(29.1\overline{9})_{16}}$. Haverá perda de dígitos significativos.

5) $F(\beta, t, I, S) \longrightarrow F(2, 3, -3, +3) \longrightarrow$ $\begin{array}{|c|c|c|c|c|c|c|c|}\hline s_1 & d_1 & d_2 & d_3 & s_2 & e_1 & e_2 & e_3 \\ \hline\end{array}$

(a) $NM = (\beta - 1) \cdot \beta^{t-1} = (2-1) \cdot 2^{3-1} = \mathbf{4}$

(b) $NE = S - I + 1 = 3 - (-3) + 1 = \mathbf{7}$

(c) $NR = 2 \cdot NM \cdot NE + 1 = \mathbf{57}$

(d) $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ \hline\end{array} \longrightarrow m.p. = (0.1)_2 \cdot (2^{-3})_{10} = (2^{-1})_{10} \cdot (2^{-3})_{10} = (0.0625)_{10}$

Logo, a região de *underflow* é $\boldsymbol{\{x \in \mathbb{R} \mid -(0.0625)_{10} < x < (0.0625)_{10}\}}$.

$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ \hline\end{array} \longrightarrow M.P. = (0.111)_2 \cdot (2^3)_{10} = (0.875)_{10} \cdot (2^3)_{10} = (7)_{10}$

Logo, a região de *overflow* é $\boldsymbol{\{x \in \mathbb{R} \mid x < -(7)_{10} \cup x > (7)_{10}\}}$.

(e) A precisão decimal equivalente é aproximadamente $log_{10}(2^3) = 3 \cdot log_{10}(2) \approx \mathbf{0.90308998699}$.

(f) $F(2, 3, 0, 6)$ (polarização $p = +3$)

6) $F(\beta, t, I, S) \longrightarrow F(2, 3, 0, +7) \longrightarrow$ $\begin{array}{|c|c|c|c|c|c|c|c|}\hline s_1 & d_1 & d_2 & d_3 & 0 & e_1 & e_2 & e_3 \\ \hline\end{array}$

(a) $NM = (\beta - 1) \cdot \beta^{t-1} = (2-1) \cdot 2^{3-1} = \mathbf{4}$

(b) $NE = S - I + 1 = 7 - 0 + 1 = \mathbf{8}$

(c) $NR = 2 \cdot NM \cdot NE + 1 = \mathbf{65}$

(d) $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline\end{array} \longrightarrow m.p. = (0.1)_2 \cdot (2^0)_{10} = (2^{-1})_{10} \cdot (1)_{10} = \mathbf{(0.5)_{10}}$

Logo, a região de *underflow* é $\boldsymbol{\{x \in \mathbb{R} \mid -(0.5)_{10} < x < (0.5)_{10}\}}$.

$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ \hline\end{array} \longrightarrow m.p. = (0.111)_2 \cdot (2^7)_{10} = (0.875)_{10} \cdot (2^7)_{10} = \mathbf{(112)_{10}}$

Logo, a região de *overflow* é $\boldsymbol{\{x \in \mathbb{R} \mid x < -(112)_{10} \cup x > (112)_{10}\}}$.

(e) A precisão decimal equivalente é aproximadamente $log_{10}(2^3) = 3 \cdot log_{10}(2) \approx \mathbf{0.90308998699}$.

7) O formato de ponto flutuante de precisão dupla tem 1 bit de sinal, 11 bits para expoente e 52 bits explícitos para decimais (o 1º bit é implícito por conta do armazenamento do número de acordo com o padrão IEEE 754).

Também foi decidido neste padrão facilitar o armazenamento do expoente. Assim, um conceito chamado *exponent bias*, calculado por $2^{k-1} - 1$ onde k é o número de bits do expoente, foi criado com esta função. Este *bias* é essencial para a codificação do expoente como um valor `unsigned`. Para obter o expoente verdadeiro, subtrai-se o *bias* do expoente representado.

Assim sendo, $bias = 2^{11-1} - 1 = 1023$. $e_{min} = 1 - 1023 = -1022$ e $e_{max} = 2046 - 1023 = 1023$. (Os expoentes 0 e 2047 são reservados.)

Portanto, o menor número passível de ser representado com precisão dupla é $2^{-1022} \approx \mathbf{2.225 \cdot 10^{-308}}$, e o maior número é 2^{1023} com todos os bits da mantissa ativados, ou seja, $2^{1023} \cdot (2 - 2^{-52}) \approx \mathbf{1.797 \cdot 10^{308}}$. Às custas da perda de precisão, existem números que vão abaixo da fronteira de *underflow*, chamados de subnormais.

```
>>> 2**(-1022 - 52)
5e-324
>>> 2**(-1022)
2.2250738585072014e-308
>>> (2 - 2**(-52)) * (2**1023)
1.7976931348623157e+308
>>> float(2**1024)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

8) $log_{10}(2^{53}) = 53 \cdot log_{10}(2) \approx \mathbf{15.9546}$

9)
```
from numpy import float32 as single

h = 1/2
x = 2/3 - h
y = 3/5 - h
e = 3*x - h
f = 5*y - h
g = e/f

for i in [h, x, y, e, f, g]:
    print("{:.55f}\n{:.55f}\n".format(single(i), i))
```

- O processo descrito em código pode ser reescrito aritmeticamente da seguinte maneira:

$$g = \frac{e}{f} = \frac{3 \cdot x - h}{5 \cdot y - h} = \frac{3 \cdot (\frac{2}{3} - h) - h}{5 \cdot (\frac{3}{5} - h) - h} = \frac{3 \cdot (\frac{2}{3} - \frac{1}{2}) - \frac{1}{2}}{5 \cdot (\frac{3}{5} - \frac{1}{2}) - \frac{1}{2}} = \frac{3 \cdot \frac{1}{6} - \frac{1}{2}}{5 \cdot \frac{1}{10} - \frac{1}{2}} = \frac{\frac{1}{2} - \frac{1}{2}}{\frac{1}{2} - \frac{1}{2}} = \frac{0}{0}$$

  Porém, nota-se que na saída do programa, as variáveis `x` e `y` são responsáveis pelo resultado válido ($\texttt{g} = 1$), em virtude da imprecisão de seu armazenamento quando $\beta = 2$ (as multiplicações realizadas nunca serão iguais a um valor exato binariamente como $\frac{1}{2}$). Assim, como $\texttt{e} = \texttt{f}$ e $e, f \neq 0$, então $\frac{e}{f} = g = 1$.

- Saída do programa:

```
0.5000000000000000000000000000000000000000000000000000000
0.5000000000000000000000000000000000000000000000000000000

0.1666666716337203979492187500000000000000000000000000000
0.1666666666666666296592325124947819858789443969726562500

0.1000000014901161193847656250000000000000000000000000000
0.0999999999999999777955395074968691915273666381835937500

-0.0000000000000001110223024625156540423631668090820312500
-0.0000000000000001110223024625156540423631668090820312500

-0.0000000000000001110223024625156540423631668090820312500
-0.0000000000000001110223024625156540423631668090820312500

1.0000000000000000000000000000000000000000000000000000000
1.0000000000000000000000000000000000000000000000000000000
```

10)
```
def quad_eq(a, b, c):
    d = (b**2 - 4*a*c)**(1/2)
    return (- b / (2*a)) if d == 0 else ((- b + d) / (2*a), (- b - d) / (2*a))


def rat_quad_eq(a, b, c):
    d = (b**2 - 4*a*c)**(1/2)
    return (- b / (2*a)) if d == 0 else ((- 2*c) / (b + d), (- 2*c) / (b - d))


print("x1 = {:.25f} x2 = {:.25f}".format(*quad_eq(1, 62.1, 1)))
print("x1 = {:.25f} x2 = {:.25f}".format(*rat_quad_eq(1, 62.1, 1)))
print("x1 = {} {: <20} x2 = {}".format(round(-0.0161072, 4), "",
                                       round(-62.0839, 2)))
print("x1 = {} {: <17} x2 = {}".format(-0.0161072, "", -62.0839))
```

- O código acima simula os cálculos com precisão dupla nos dois tipos de equação quadrática, além de imprimir o resultado arredondado para o operador aritmético $F(10, 4, -99, +99)$ e os resultados reais.
- Saída do programa:

```
x1 = -0.0161072374089670233843208 x2 = -62.0838927625910343977011507
x1 = -0.0161072374089685811660022 x2 = -62.0838927625970313783909660
x1 = -0.0161                      x2 = -62.08
x1 = -0.0161072                   x2 = -62.0839
```

(a) $x_1 = \mathbf{-0.0161}$ e $x_2 = \mathbf{-62.08}$.

(b) A fórmula racionalizada só mostra diferença de resultados com maior precisão. Também foi testada em uma calculadora científica convencional e os resultados são iguais à fórmula usual, embora todos diferentes dos reais.

(c) $abs(\frac{VA-VE}{VE}) = abs(\frac{-0.0161-(-0.0161072)}{-0.016172}) = abs(\frac{7.2\cdot 10^{-6}}{-0.016172}) \approx \mathbf{0.000447}$

$abs(\frac{VA-VE}{VE}) = abs(\frac{-62.08-(-62.0839)}{-62.0839}) = abs(\frac{3.9\cdot 10^{-3}}{-62.0839}) \approx \mathbf{0.00006282}$

11) (a)
- Erro absoluto: $abs(VA - VE) = abs(1102.345 - 1100.9) = \mathbf{1.445}$
- Erro relativo: $abs(\frac{VA-VE}{VE}) = abs(\frac{1.445}{1100.9}) \approx \mathbf{0.00131}$
- Percentual: $0.00131 \cdot 100 = \mathbf{0.131\%}$

(b)
- Erro absoluto: $abs(VA - VE) = abs(0.01245 - 0.0119) = \mathbf{0.00055}$
- Erro relativo: $abs(\frac{VA-VE}{VE}) = abs(\frac{0.00055}{0.0119}) \approx \mathbf{0.04622}$
- Percentual: $0.04622 \cdot 100 = \mathbf{4.622\%}$

12) (a) Não é necessário fazer nenhum cálculo com a parte inteira. Então, $-(0.05)_{10} = (?)_2$

$$\begin{array}{cccccccccc} 0.05 & 0.1 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 & 0.4 & 0.8 & 0.6 \\ \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \ldots \\ 0.1 & 0.2 & 0.4 & 0.8 & 1.6 & 1.2 & 0.4 & 0.8 & 1.6 & 1.2 \end{array}$$

$$-(0.05)_{10} = -(0.000\overline{011})_2 = 1 \cdot 2^{-5} \cdot -(0.\overline{1001})_2$$

$s = 1$ (sinal negativo)

$e = -5 + 127 = 122$ (polarização ou *exponent bias*)

O arredondamento no bit final da mantissa acontece pois a fração binária começa por 1, ou seja, é maior do que $(0.5)_{10}$. A parcela binária arredondada é igual a $2^{-23} \cdot (0.10011001\ldots)_2$

Resultado final: $\boxed{\mathbf{1}}\boxed{\mathbf{01111010}}\boxed{\mathbf{10011001100110011001101}}$

(b) O erro de arredondamento é próximo a $2^{-24} \cdot (1.\overline{1001})_2 \approx (9.53674 \cdot 10^{-8})_{10} \approx \mathbf{9.53674 \cdot 10^{-6}\%}$.

13) $\boxed{1}\boxed{00000000}\boxed{10000000000000000000001}$ (Número subnormal)

$$e = 0, f \neq 0 \longrightarrow v = (-1)^s \cdot \left(\sum_{i=1}^{23} b_{23-i} 2^{-i}\right) \cdot 2^{e-126} = (-1)^1 \cdot (2^{-1} + 2^{-23}) \cdot 2^{-126} \approx \mathbf{-5.877473 \cdot 10^{-39}}$$

14) (a) $(-1.51 \cdot 10^{+37})_{10} = (-1.01101011100001001110111101\ldots)_2 \cdot 2^{123}$

$s = 1$ (número negativo)

$e = 123 + 127 = (250)_{10} = (11111010)_2$ (*bias* + 123 bits decimais)

$f = (01101011100001001110111)_2 + (1)_2 = (01101011100001001111000)_2$ (primeiros 23 bits + arredondamento)

Resultado final: $\boxed{\mathbf{1}}\boxed{\mathbf{11111010}}\boxed{\mathbf{01101011100001001111000}}$

(b) $2^{-24} \cdot (1.010101100111000111010110 11\ldots)_2 \approx (7.97316 \cdot 10^{-8})_{10} \approx \mathbf{7.97316 \cdot 10^{-6}\%}$

15) (a) $(-1.1 \cdot 10^{-41})_{10} = (-1.11101010100111011100110 0000\ldots)_2 \cdot 2^{-137}$

$s = 1$ (número negativo)

$e = -137 + 127 = -10 < 0 \longrightarrow$ número subnormal

É necessário mover o decimal para que o número possa ser representado com $e = 0$. Então,
$(-1.111010101001110111001100000\ldots)_2 \cdot 2^{-137} = (-0.00000000001111010101001\ldots)_2 \cdot 2^{-126}$.

Resultado final: $\boxed{\mathbf{1}}\boxed{\mathbf{00000000}}\boxed{\mathbf{00000000001111010101010}}$

(b) $2^{-23} \cdot (1.101110011000\ldots)_2 \approx (1.3126 \cdot 10^{-7})_{10} \approx \mathbf{1.3126 \cdot 10^{-5}\%}$.

16) Caso seja viável calcular o resultado desejado à mão, ou utilizando ferramentas de maior precisão, é possível verificar o erro relativo sem maiores dificuldades. Porém, na maioria dos casos isso não acontece, e é necessário observar como o computador se comporta nestes casos. Esta aritmética de ponto flutuante é uma grande área de estudo do cálculo numérico.

17, 18)

```python
from functools import reduce
from math import factorial as fact
from numpy import float32 as single

n = 4
x = -.111
desired = 0.894938748929031

for y in [single(x), x]:
    obtained = reduce(lambda w, i: w + (y**i)/fact(i), range(n), 0)
    print("{:.65f}%".format(abs((desired - obtained) / desired) * 100))
```

- O decimal desejado foi retirado do Wolfram Alpha, para comparação com os resultados calculados pelo programa em variáveis de precisão simples e dupla, respectivamente. O erro da variável de precisão simples é maior pela quantidade reduzida de bits disponíveis para armazenar a parte decimal do resultado.
- Novamente, a função *reduce* é utilizada para simular o cálculo da série de Maclaurin com os argumentos fornecidos.
- Saída do programa:

```
0.00069152325516942810771509053680006218201015144586563110351562500%
0.00069138016857893662848316695956896182906348258256912231445312500%
```