Cálculo Numérico
Projeto 1 - Turma D

Lucas Baganha Galante 182364
Tiago Loureiro Chaves 187690

2 de Outubro de 2018

1 Questão 1

Fazer o projeto 1: “Precisão da Máquina”, da página 24 do livro *Cálculo Numérico*, Ruggiero-Lopes, 2a. edição.

1.A

O algoritmo descrito no exercício foi implementado na linguagem C, por apresentar variáveis com precisão simples e dupla. A saída contém a precisão de máquina de cada tipo de variável expressa em 20 casas decimais, como visto na Listagem 1.

Listagem 1: **Precisão de máquina para variáveis simples e duplas.**

```
The single precision is: 0.00000011920928955078
The double precision is: 0.00000000000000022204
```

A variável com precisão dupla apresenta maior precisão que a variável com precisão simples; com 16 casas decimais, que compara com 7 casas decimais da precisão simples. Portanto o ganho é maior que o dobro, como esperado.

1.B

Quando se sai do laço while se perde uma precisão a mais que a precisão de máquina, que é o motivo para se sair do laço, pois a soma $1+A$ não é maior que 1, devido a aritmética de ponto flutuante. Deste modo é necessário multiplicar A por 2, para que se retorne ao menor valor em que a máquina possui precisão.

1.C

O algoritmo executado na `Parte A` foi modificado para se utilizar um valor **VAL** para inicialização ao invés da referência, o número 1. A saída é vista abaixo na Listagem 2

Listagem 2: **Precisão de máquina para diferentes valores VAL.**

```
The single precision for VAL:    10.0 is: 0.00000095367431640625
The single precision for VAL:    17.0 is: 0.00000190734863281250
The single precision for VAL:   100.0 is: 0.00000762939453125000
The single precision for VAL:   184.0 is: 0.00001525878906250000
The single precision for VAL:  1000.0 is: 0.00006103515625000000
The single precision for VAL:  1575.0 is: 0.00012207031250000000
The single precision for VAL: 10000.0 is: 0.00097656250000000000
The single precision for VAL: 17893.0 is: 0.00195312500000000000
```

Se percebe da saída que quanto maior o valor de **VAL** menor se torna a precisão da máquina. Isso ocorre pois serão necessários mais dígitos da mantissa para representar na variável de iteração (no algoritmo atribuída como variável `s`), o que diminui o número de dígitos que podem ser usados para determinar a precisão.

Código

Listagem 3: **Base de código para a questão 1.**

```c
#include<stdio.h>
#include<stdlib.h>

void singlePrecision();
void doublePrecision();
void singlePrecisionVal(float VAL);

int main(){
  /*PARTE 1.A*/
  //singlePrecision();
  //doublePrecision();

  /*PARTE 1.B*/
  singlePrecisionVal(10.0);
  singlePrecisionVal(17.0);
  singlePrecisionVal(100.0);
  singlePrecisionVal(184.0);
  singlePrecisionVal(1000.0);
  singlePrecisionVal(1575.0);
  singlePrecisionVal(10000.0);
  singlePrecisionVal(17893.0);
  return 0;
}

```

```c
void singlePrecision(){
  float A = 1.0;
  float s = 2.0;

  while(s > 1.0){
    A = A/2.0;
    s = 1.0 + A;
  }

  printf("The single precision is: %.20f\n", 2*A);
  return;
}

void singlePrecisionVal(float VAL){
  float A = 1.0;
  float s = VAL + A;

  while(s > VAL){
    A = A/2.0;
    s = VAL + A;
  }

  printf("The single precision for VAL: %7.1f is: %.20f\n",VAL, 2*A);
  return;
}


void doublePrecision(){
  double A = 1.0;
  double s = 2.0;

  while(s > 1.0){
    A = A/2.0;
    s = 1.0 + A;
  }

  printf("The double precision is: %.20lf\n", 2*A);
  return;
}
```

2 Questão 2

Uma raiz α de $f(x) = 0$ e dita múltipla de multiplicidade p se $0 \neq |g(\alpha)| < \infty, f(x) = (x-\alpha)^p g(x)$. Assim, se α é de multiplicidade p então

$$f(\alpha) = f'(\alpha) = ... = f^{(p-1)}(\alpha) = 0 \quad e \quad f^{(p)}(\alpha) \neq 0$$

Vimos que e α é uma raiz múltipla então o método de Newton não converge quadraticamente, mas sim linearmente com constante assintótica do erro (taxa de convergência) igual a $(1-1/p)$.

a) Mostre, com o devido desenvolvimento, uma forma de recuperar a convergência quadrática do método de Newton.

b) Defina um problema com raiz múltipla, implemente o Método da Bisseção, o Método da Secante, o método de Newton original e a sua modificação discutida na letra a) e os utilize para calcular a raiz, mostrando numericamente a taxa e a ordem de convergência obtidas para cada método.

2.A

Uma sequência de iterações $x_n/n \geq 0$ é dita convergente para um ponto α, com ordem de convergência $p \geq 1$ se para algum $c > 0$ (taxa de convergência):

$$|\alpha - x_{n+1}| \leq c \cdot |\alpha - x_n|^p, \quad n \geq 0$$

, ou ainda:

$$\lim_{n\to\infty} \frac{|\alpha - x_{n+1}|}{|\alpha - x_n|^p} = c$$

Analisando o método de Newton modificado [1, p. 101] para uma função f com raiz α de $f(x)=0$ de multiplicidade m:

$$x_{n+1} = x_n - m \cdot \frac{f(x_n)}{f'(x_n)}$$

$$f(\alpha) = f'(\alpha) = \ldots = f^{(m-1)}(\alpha) = 0 \quad e \quad f^{(m)}(\alpha) \neq 0$$

Expandindo $f(x)$ e $f'(x)$ em torno de α:

$$f(x) = \cancelto{0}{f(\alpha)} + \cancelto{0}{\ldots} + \frac{(x-\alpha)^m}{m!} \cdot f^{(m)}(\alpha) + \frac{(x-\alpha)^{(m+1)}}{(m+1)!} \cdot f^{(m+1)}(\alpha) + \underbrace{O((x-\alpha)^{m+2})}_{\approx 0}$$

$$f'(x) = \cancelto{0}{f'(\alpha)} + (x-\alpha) \cdot \cancelto{0}{f''(\alpha)} + \cancelto{0}{\ldots} + \frac{(x-\alpha)^{(m-1)}}{(m-1)!} \cdot f^{(m)}(\alpha) + \underbrace{O((x-\alpha)^m)}_{\approx 0}$$

Avaliando-as então em x_n:

$$f(x_n) = \frac{(x_n-\alpha)^m}{m!} \cdot f^{(m)}(\alpha) + \frac{(x_n-\alpha)^{(m+1)}}{(m+1)!} \cdot f^{(m+1)}(\alpha)$$

$$f'(x_n) = \frac{(x_n-\alpha)^{(m-1)}}{(m-1)!} \cdot f^{(m)}(\alpha)$$

Assim, podemos reescrever $x_{n+1} - \alpha$ como:

$$x_{n+1} - \alpha \stackrel{def.}{=} (x_n - m \cdot \frac{f(x_n)}{f'(x_n)}) - \alpha$$

$$= x_n - m \cdot \frac{\frac{(x_n-\alpha)^m}{m!} \cdot f^{(m)}(\alpha) + \frac{(x_n-\alpha)^{(m+1)}}{(m+1)!} \cdot f^{(m+1)}(\alpha)}{\frac{(x_n-\alpha)^{(m-1)}}{(m-1)!} \cdot f^{(m)}(\alpha)} - \alpha$$

$$= x_n - \cancel{m} \cdot \frac{(x_n-\alpha)^{\cancel{m}} \cdot [(m+1) \cdot f^{(m)}(\alpha) + (x_n-\alpha) \cdot f^{(m+1)}(\alpha))]}{(m+1)\cancel{!}} \cdot \frac{\cancel{(m-1)!}}{\cancel{(x-\alpha)^{m-1}} \cdot f^{(m)}(\alpha)} - \alpha$$

$$= x_n - \frac{(x_n-\alpha) \cdot [(m+1) \cdot f^{(m)}(\alpha) + (x_n-\alpha) \cdot f^{(m+1)}(\alpha))]}{(m+1) \cdot f^{(m)}(\alpha)} - \alpha$$

$$= \cancel{x_n} - (\cancel{x_n} - \cancel{\alpha}) - \frac{(x_n-\alpha)^2 \cdot f^{(m+1)}(\alpha)}{(m+1) \cdot f^{(m)}(\alpha)} - \cancel{\alpha}$$

$$= -(x_n-\alpha)^2 \cdot \frac{f^{(m+1)}(\alpha)}{(m+1) \cdot f^{(m)}(\alpha)}$$

Logo:

$$|x_{n+1}-\alpha| = \underbrace{|\frac{f^{(m+1)}(\alpha)}{(m+1) \cdot f^{(m)}(\alpha)}|}_{=c} \cdot |x_n-\alpha|^{\overbrace{2}^{=p}}$$

$\Rightarrow$ ordem de convergência 2 (convergência quadrática)

Outra forma de modificar o método de Newton [2, p. 353-356], implementada na parte B, é:

$$x_{n+1} = x_n - \frac{U(x_n)}{U'(x_n)} \quad , \quad \text{onde} \quad U(x) = \frac{f(x)}{f'(x)}$$

Usando as expansões de f e f' calculadas anteriormente com $A = \frac{f^{(m)}(\alpha)}{m!} \neq 0$, $B = \frac{f^{(m+1)}(\alpha)}{(m+1)!}$ e $C = \frac{B}{A}$, constantes:

$$f(x) = A \cdot (x-\alpha)^m + B \cdot (x-\alpha)^{m+1}$$

$$f'(x) = m \cdot A \cdot (x-\alpha)^{m-1}$$

Substituindo em $\frac{U(x)}{U'(x)}$:

$$U(x) = \frac{1}{m} \cdot (x-\alpha) + \frac{B}{m \cdot A} \cdot (x-\alpha)^2, \quad U'(x) = \frac{1}{m} + \frac{2B}{m \cdot A} \cdot (x-\alpha)$$

$$\frac{U(x)}{U'(x)} = \frac{\cancel{\frac{1}{m}} \cdot (x-\alpha) \cdot [1 + C \cdot (x-\alpha)]}{\cancel{\frac{1}{m}} \cdot [1 + 2C \cdot (x-\alpha)]}$$

Portanto o erro na $(n+1)$-ésima iteração é $e_{n+1} = x_{n+1} - \alpha \overset{def.}{=} x_n - \frac{U(x)}{U'(x)} - \alpha = e_n - \frac{U(x)}{U'(x)}$:

$$e_{n+1} = e_n - \frac{e_n \cdot (1 + C \cdot e_n)}{(1 + 2C \cdot e_n)}$$

$$= e_n \cdot (1 - \frac{(1 + C \cdot e_n)}{(1 + 2C \cdot e_n)}) = e_n \cdot (\frac{C \cdot e_n}{1 + 2C \cdot e_n})$$

$$= e_n^2 \cdot (\frac{C}{1 + 2C \cdot e_n})$$

Assim, temos que:

$$\lim_{n\to\infty}\frac{|e_{n+1}|}{|e_n|^2} = \lim_{n\to\infty}\left|\frac{C}{1+2C\cdot e_n}\right| = |C|\cdot\lim_{n\to\infty}\frac{1}{1+2|C|\cdot|\cancel{\underbrace{x_n}_{\to\alpha}-\alpha}|} = |C|\cdot\lim_{n\to\infty}\frac{1}{1} = |C|$$

$$\therefore \lim_{n\to\infty}\frac{|e_{n+1}|}{|e_n|^2} = |C| \Rightarrow \text{ordem de convergência 2 (convergência quadrática)}$$

Concluindo, caso estejamos tratando uma função f com raiz α de multiplicidade m, as seguintes modificações no método de Newton recuperam sua convergência quadrática:

$$x_{n+1} = x_n - m\cdot\frac{f(x_n)}{f'(x_n)}$$

$$x_{n+1} = x_n - \frac{U(x_n)}{U'(x_n)}, \quad U(x) = \frac{f(x)}{f'(x)}$$

Além disso, como veremos na parte B abaixo, para um polinômio com raízes múltiplas foram necessárias 9 iterações do método de Newton original, enquanto que com a segunda modificação apresentada foram necessárias somente 2 iterações para o mesmo ponto inicial.

2.B

Foram implementados os 4 algoritmos pedidos no enunciado, Método da Bisseção, o Método da Secante, o Método de Newton original e sua modificação (na `Parte A` dessa questão foram provados duas maneiras distintas de se obter a convergência quadrática do método de Newton, foi implementada a segunda maneira pois não depende saber m a priori).

Para exemplificar este exercício foi utilizado um polinômio de raizes múltiplas:

$$2x^5 + -20x^4 + 68x^3 - 104x^2 + 74x - 20 = 2(x-1)^3(x-2)(x-5)$$

O cálculo da raiz de cada um desses algoritmos está exibido na Listagem 4, assim como o número de iterações necessárias para convergir. Os métodos da bisseção e secante convergem a taxas menores, e portanto realizam maior número de iterações. (Adendo: todos os métodos são dependentes dos pontos inicias, e podem convergir mais rapidamente dependendo destes pontos.) Os métodos de Newton comparados apresentaram uma diferença significativa em fator de convergência, sendo o método modificado mais que quadraticamente (no caso até cubicamente mais rápido) mais rápido para o mesmo ponto inicial.

Listagem 4: **Passo a passo de diferentes algoritmos para determinar a raiz de função com raizes múltiplas.**

```
Media: 4.5 F_m: -107.1875
Media: 6.75 F_m: 3160.56835938
Media: 5.625 F_m: 448.283996582
Media: 5.0625 F_m: 25.666475296
Media: 4.78125 F_m: -65.7846035361
```

```
Media: 4.921875 F_m: -27.5399343763
Media: 4.9921875 F_m: -2.97468937194
Media: 5.02734375 F_m: 10.8144866157
Media: 5.009765625 F_m: 3.78982958678
Media: 5.0009765625 F_m: 0.375396885005
Media: 4.99658203125 F_m: -1.30764677991
Media: 4.99877929688 F_m: -0.468130417219
Media: 4.99987792969 F_m: -0.0468688014225
Media: 5.00042724609 F_m: 0.164138449422
Media: 5.00015258789 F_m: 0.0586034363523
Media: 5.00001525879 F_m: 0.00585947185755
Media: 4.99994659424 F_m: -0.0205066260205
Media: 4.99998092651 F_m: -0.0073240674119
Media: 4.99999809265 F_m: -0.000732420361601
Media: 5.00000667572 F_m: 0.00256349510164
Media: 5.00000238419 F_m: 0.000915529709346
Media: 5.00000023842 F_m: 9.15527575671e-05
METODO DA BISSECAO
Achou a raiz: 5.00000023842
Numero de iteracoes: 22
------------------------------------
Media: 0.00627352572146 F_m: -19.5398354788
Media: 0.272664468156 F_m: -6.28387508792
Media: 0.398944824875 F_m: -3.19916587126
Media: 0.529910753606 F_m: -1.36530864458
Media: 0.627415026044 F_m: -0.620845449741
Media: 0.708728762294 F_m: -0.273858511139
Media: 0.772905410655 F_m: -0.121498345517
Media: 0.824082544553 F_m: -0.0534669661609
Media: 0.864303485423 F_m: -0.0234718342023
Media: 0.895777234458 F_m: -0.0102613711505
Media: 0.92022481204 F_m: -0.00447304307117
Media: 0.939117152973 F_m: -0.00194447669548
Media: 0.953645431535 F_m: -0.00084343269782
Media: 0.96477452829 F_m: -0.000365176821177
Media: 0.973272255917 F_m: -0.000157879038625
Media: 0.979744168215 F_m: -6.81778053888e-05
METODO DA SECANTE
Achou a raiz: 0.979744168215
Numero de iteracoes: 16
------------------------------------
Media: 0.645161290323 F_m: -0.527208787994
Media: 0.751292823997 F_m: -0.163234649909
Media: 0.827637023342 F_m: -0.050096355305
Media: 0.881697563505 F_m: -0.0152506576222
Media: 0.919439444395 F_m: -0.00461069390615
Media: 0.945474617282 F_m: -0.00138619147897
Media: 0.963263407207 F_m: -0.000414978367587
Media: 0.97532981075 F_m: -0.000123840254886
```

```
 Media:  0.983471233981  F_m:  -3.68741353824e-05
METODO DE NEWTON
 Achou a raiz: 0.983471233981
 Numero de iteracoes: 9
----------------------------------------
 Media:  1.05138339921  F_m:  0.00101632866931
 Media:  1.00120254996  F_m:  1.38914231229e-08
METODO DE NEWTON PARA RAIZES MULTIPLAS
 Achou a raiz: 1.00120254996
 Numero de iteracoes: 2
----------------------------------------
```

Código

Listagem 5: **Base de código para a questão 2.**

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math
EPSILON = 0.0001

#Iteration counters
i_bissecao = 0
i_fpos = 0
i_newton = 0
i_newton_mr = 0
i_secante = 0


#Function and respective derivative and second derivative
def func(x):
    return 2*x**5 - 20*x**4 + 68*x**3 - 104*x**2 + 74*x - 20

def dfunc(x):
    return 10*x**4 - 80*x**3 + 204*x**2 - 208*x + 74

def ddfunc(x):
    return 40*x**3 -  240*x**2 + 408*x - 208

def section_divider():
    print("----------------------------------------")

#Methods for finding roots
def bissecao(a, b):
    global i_bissecao
    i_bissecao += 1
    f_a = func(a)
```

```
    f_b = func(b)
    if(f_a*f_b > 0):
        print("Escolha a e b diferentes")
        return
    m = (a+b)/2
    f_m = func(m)
    print('Media: '+str(m)+' F_m: '+str(f_m))
    if(math.fabs(f_m) < EPSILON):
        print("METODO DA BISSECAO")
        print("Achou a raiz: "+str(m))
        print("Numero de iteracoes: "+str(i_bissecao))
        section_divider()
        return
    if(f_a*f_m < 0):
        bissecao(a,m)
    else:
        bissecao(m,b)

def falsaposicao(a,b):
    global i_fpos
    i_fpos += 1
    f_a = func(a)
    f_b = func(b)
    if(f_a*f_b > 0):
            print("Escolha a e b diferentes")
            return
    m = (a*f_b - b*f_a)/(f_b - f_a)
    f_m = func(m)
    print('Media: '+str(m)+' F_m: '+str(f_m))
    if(math.fabs(f_m) < EPSILON):
        print("METODO DA FALSA POSICAO")
        print("Achou a raiz: "+str(m))
        print("Numero de iteracoes: "+str(i_fpos))
        section_divider()
        return
    if(f_a*f_m < 0):
        falsaposicao(a,m)
    else:
        falsaposicao(m,b)

def newton(x):
    global i_newton
    i_newton += 1
    f_x = func(x)
    df_x = dfunc(x)
    if(df_x == 0.0):
        print('DIVISION BY ZERO EXCEPTION: df_x is zero, try another value.')
        exit(1)
    m = x - (f_x/df_x)
```

```
    f_m = func(m)
    print('Media: '+str(m)+' F_m: '+str(f_m))
    if(math.fabs(f_m) < EPSILON):
        print("METODO DE NEWTON")
        print("Achou a raiz: "+str(m))
        print("Numero de iteracoes: "+str(i_newton))
        section_divider()
        return
    else:
        newton(m)

def secante(x1,x0):
    global i_secante
    i_secante += 1
    f_x1 = func(x1)
    f_x0 = func(x0)
    m = x1 - f_x1*((x1 - x0)/(f_x1 - f_x0))
    f_m = func(m)
    print('Media: '+str(m)+' F_m: '+str(f_m))
    if(math.fabs(f_m) < EPSILON):
        print("METODO DA SECANTE")
        print("Achou a raiz: "+str(m))
        print("Numero de iteracoes: "+str(i_secante))
        section_divider()
        return
    else:
        secante(m,x1)

#Newton method for square roots
def newton_sqrt(a):
    i = 1 #counter
    x = a #init x
    f_x = 1 #init f_x
    while(math.fabs(f_x) > EPSILON):
        x = (x**2 + a)/(2*x)
        f_x = x**2 - a
        #print('X: '+str(x)+' F_x: '+str(f_x))
        i += 1
    print('THE SQUARE ROOT OF', a ,'IS', x, 'WITH',i,'ITERATIONS')
    section_divider()

#Newton method for 1/a calculations
def divbyA(a):
    i = 1 #counter
    x = 0.1 #init x
    f_x = 1 #init f_x
    while(math.fabs(f_x) > EPSILON):
        x = 2*x - a*x**2
        f_x = (1/x) - a
```

```
        print('X: '+str(x)+' F_x: '+str(f_x))
        i += 1
    print('1/A FOR A:', a ,'IS', x, 'WITH',i,'ITERATIONS')
    section_divider()

#Newton method for roots with multiplicity
def newton_mult_root(x):
    global i_newton_mr
    i_newton_mr += 1
    f_x = func(x)
    df_x = dfunc(x)
    ddf_x = ddfunc(x)
    if(df_x == 0.0):
        print('DIVISION BY ZERO EXCEPTION: df_x is zero, try another value.')
        exit(1)
    m = x - (f_x*df_x)/((df_x**2)-f_x*ddf_x)
    f_m = func(m)
    print('Media: '+str(m)+' F_m: '+str(f_m))
    if(math.fabs(f_m) < EPSILON):
        print("METODO DE NEWTON PARA RA[U+FFFD]ZES M[U+FFFD]LTIPLAS")
        print("Achou a raiz: "+str(m))
        print("Numero de iteracoes: "+str(i_newton_mr))
        section_divider()
        return
    else:
        newton_mult_root(m)


##bissecao(1.2,2.4)
##falsaposicao(1.2,2.4)
##newton(2.0)
##secante(1.2,2.4)

##newton(10.0)
##newton_sqrt(10.0)

##newton(3.0)

#divbyA(4.0)

bissecao(0.0,9.0)
secante(0.0,9.0)
newton(0.5)
newton_mult_root(0.5)
```

3 Questão 3

Sejam $A \in \mathbb{R}^{n\times n}$ uma matriz inversível, $\mathbf{b} \in \mathbb{R}^n$ um vetor dado e o problema: encontrar $\mathbf{x} \in \mathbb{R}^n$ tal que

$$A\mathbf{x} = \mathbf{b}. \tag{1}$$

a) Utilizando como base algum dos diversos algoritmos encontrados nas referencias bibliográficas, escreva um programa para realizar a fatoração $A = LU$ e, utilizando esta fatoração, resolver o problema (1). Discuta sua complexidade computacional.

b) Com base no algoritmo da letra a), desenvolva um programa especializado para o caso em que a matriz A é tridiagonal, ou seja $A = (a_{ij})$ com $a_{ij} = 0$ para $|i-j| > 1$, discutindo sua complexidade computacional.

c) Utilizando os programas das letras a) e b), resolva o sistema dado por:

$$-x_{k-1} + 2x_k - x_{k+1} = \frac{8}{(n+1)^2}, \quad k = 1,2,...,n$$

com $x_0 = x_{n+1} = 0$, para vários valores de n e compare sua solução com a solução analítica:

$$x_k = 4\left[\frac{k}{n+1} - \left(\frac{k}{n+1}\right)^2\right].$$

Discuta o comportamento dos programas para o caso em que n é grande.

3.A

Avaliando a ordem de operações realizadas pelo algoritmo de fatoração $A = LU$ (vide Listagem 7 linha 4), temos:

$$n^2 + n^2 + \sum_{j=0}^{n-1}\left[\sum_{i=0}^{j}\sum_{k=0}^{i} 1 + \sum_{i=j}^{n-1}\sum_{k=0}^{j} 1\right] \in O(n^3)$$

Enquanto que para as funções (análogas) de substituição que serolvem os sistemas triangulares $L\mathbf{y} = \mathbf{b}$ e $U\mathbf{x} = \mathbf{y}$ (linhas 15 e 22 da Listagem 7) temos:

$$n + \sum_{i=0}^{n-1} i \in O(n^2)$$

Portanto o algoritmo tem complexidade computacional $O(n^3)$.

3.B

Uma vez que só temos valores não nulos nos elementos a_{ij} com $|i-j| \leq 1$, há $(3n-2)$ elementos a serem considerados em uma matriz $n \times n$.

Assim, utilizando um algoritmo especializadopara uma matriz A tridiagonal [3] , a fatoração $A = LU$ leva $O(n)$ operações (pois podemos realizá-la em apenas um laço, vide Listagem 8 linha 28).

Deste modo, a complexidade computacional do algoritmo passa a ser dominada pelas mesmas funções de substituição para frente e para trás utilizadas na parte A, que executam $O(n^2)$ operações para resolver os sistemas triangulares resultantes.

3.C

Como esperado, para todos os valores testados de n o algoritmo da parte B foi mais rápido que o da parte A (vide Listagem 6).

Contudo, notou-se um resultado interessante ao executar-se o código da Listagem 9, os erros entre a solução analítica do sistema e os algoritmos das partes A e B foram iguais, sendo o maior erro de um sistema 256×256 ($n = 256$) da ordem de $e^{-14} \approx 10^{-6}$.

Listagem 6: **Razão entre o tempo de execução do algoritmo da parte A (t_1) pelo da parte B (t_2) para solução do sistema da parte C de tamanho $n \times n$. Média para 10 repetições, conforme código da Listagem 10.**

```
n: t1/t2
4: 1.8256878668084204
16: 3.2834055804664617
64: 18.917487992035966
128: 30.56859656804358
256: 45.38773459051425
```

Código

Listagem 7: **Base de código para a questão 3.a**

```python
from pprint import pprint
from numpy import dot

def lu(A):
    n = len(A)
    L = [[(1.0 if i == j else 0.0) for i in range(n)] for j in range(n)]
    U = [[0.0] * n for i in range(n)]
    for j in range(n):
        for i in range(j+1):
            U[i][j] = A[i][j] - sum(U[k][j] * L[i][k] for k in range(i))
        for i in range(j, n):
            L[i][j] = (A[i][j] - sum(U[k][j] * L[i][k] for k in range(j))) / U[j][j]
    return (L, U)

def forward_substitute(L, b):
    n = len(L)
    y = [0.0 for i in range(n)]
    for i in range(n):
        y[i] = (b[i] - dot(L[i], y)) / L[i][i]
```

```
    return y

def back_substitute(U, y):
    n = len(U)
    x = [0.0 for i in range(n)]
    for i in reversed(range(n)):
        x[i] = (y[i] - dot(U[i], x)) / U[i][i]
    return x

def decompose_and_solve(A, b):
    L, U = lu(A)
    y = forward_substitute(L, b)
    x = back_substitute(U, y)
    return (L, U, y, x)
```

Listagem 8: **Base de código para a questão 3.b**

```
from pprint import pprint
from numpy import dot

def diagonals(A):
  n = len(A)
  a = [0.0 for i in range(n-1)] # lower diagonal (i-j==1)
  b = [0.0 for i in range(n)]   # main  diagonal  (i==j)
  c = [0.0 for i in range(n-1)] # upper diagonal (j-i==1)
  c[0] = A[0][1]
  b[0] = A[0][0]
  for i in range(1, n-1):
    a[i-1] = A[i][i-1] # a starts with a_2, not a_1
    b[i] = A[i][i]
    c[i] = A[i][i+1]
  a[n-2] = A[n-1][n-2]
  b[n-1] = A[n-1][n-1]
  return (a, b, c)

def tridiagonal_lu(A):
    a, b, c = diagonals(A)
    n = len(b) # main diagonal
    L = [[(1.0 if i == j else 0.0) for i in range(n)] for j in range(n)]
    U = [[(1.0 if i == j else 0.0) for i in range(n)] for j in range(n)]
    for i in range(n-1):
      U[i][i+1] = c[i]

    U[0][0] = b[0]
    for k in range(1, n):
      L[k][k-1] = a[k-1]/U[k-1][k-1] # we use a[k-1] instead of a[k] since a starts at a_2
      U[k][k] = b[k] - L[k][k-1]*c[k-1]

```

```python
    return (L, U)

def forward_substitute(L, b):
    n = len(L)
    y = [0.0 for i in range(n)]
    for i in range(n):
        y[i] = (b[i] - dot(L[i], y)) / L[i][i]
    return y

def back_substitute(U, y):
    n = len(U)
    x = [0.0 for i in range(n)]
    for i in reversed(range(n)):
        x[i] = (y[i] - dot(U[i], x)) / U[i][i]
    return x

def tridiagonal_decompose_and_solve(A, b):
    L, U = tridiagonal_lu(A)
    y = forward_substitute(L, b)
    x = back_substitute(U, y)
    return (L, U, y, x)
```

Listagem 9: **Base de código para a questão 3.c**

```python
from pprint import pprint
from numpy import dot, subtract
import time

def create_system(n):
  A = [[(2.0 if i == j else -1.0 if abs(i-j) == 1 else 0.0) for i in range(n)]
    for j in range(n)]
  x = [4*( (k/(n+1)) - (k/(n+1))**2 ) for k in range(1, n+1)] # analytic
    solution
  b_val = 8.0 / (n+1)**2
  b = [b_val for i in range(n)]
  return (A, x, b)

def solve_system(n):
  A, x, b = create_system(n)

  print("A: ")
  pprint(A, width=30)
  print("b: ")
  pprint(b, width=30)
  print("x: ")
  pprint(x, width=30)

  start = time.time()
  _, _, _, x1 = decompose_and_solve(A, b)
```

```
  end = time.time()
  t1 = end - start
  print("\nx1 = decompose_and_solve(A, b): ")
  pprint(x1, width=30)

  start = time.time()
  _, _, _, x2 = tridiagonal_decompose_and_solve(A, b)
  end = time.time()
  t2 = end - start
  print("\nx2 = tridiagonal_decompose_and_solve(A, b): ")
  pprint(x2, width=30)

  e1 = subtract(x, x1)
  print("\nnormal factorization error (x - x1): ")
  pprint(e1, width=30)

  e2 = subtract(x, x2)
  print("\ntridiagonal specific factorization error (x - x2): ")
  pprint(e2, width=30)

  print("\ne1 == e2: " + str((e1 == e2).all()))

  print("\nt1: " + str(t1) + " | t2: " + str(t2))
  print("t1 > t2" if t1 > t2 else "t1 < t2")
  print("t1/t2 == " + str(t1/t2))
```

Listagem 10: **Base de código para análise de tempo da questão 3.c**

```
ns = [4, 16, 64, 128, 256]
t = [0.0]*len(ns)
m = 10
for j in range(m):
  i = 0
  for n in ns:
    t[i] += solve_system(n)
    i += 1
print("n: t1/t2")
for i in range(len(ns)):
  t[i] = t[i]/m
  print(str(ns[i]) + ": " + str(t[i]))
```

Referências

[1] V. L. Lopes and M. Ruggiero, “Cálculo numérico, aspectos teóricos e computacionais.”

[2] A. Ralston and P. Rabinowitz, “A first course in numerical analysis,” 1978.

[3] L. Barannyk, “Lu factorization for a tridiagonal matrix,” https://www.webpages.uidaho.edu/~barannyk/Teaching/LU_factorization_tridiagonal.pdf, acessado em: Setembro/2018.