Relatório da atividade avaliativa 1 de Cálculo Numérico

Samira Haddad RA: 11201812350
Gabrieli Parra Silva RA: 11201721386
Juliane dos Santos Assis RA: 11201810271
Liandra Cardoso da Silva RA: 11064916
Samara Suellen Miranda de Azevedo RA: 11201820807

April 6, 2021

Exercício 1

(2.0) Calcule $f_{Tn}(\lambda)$ para $\lambda = 1,\ 2,\ 3\ e\ 4$, para $n = 7$. Para $n = 0$, exiba a matriz V e a matriz V' obtida após a aplicação do processo de eliminação à matriz V , com todas as casas decimais disponíveis.

Antes de calcular o valor de $f_{Tn}(\lambda)$ temos que criar nossa matriz T_n, os valores dessa matriz são definidos pela seguinte forma:

$$a_{x,y} = \begin{cases} \lambda, & \text{se x = y} \\ -1, & \text{se x = y + 1 ou x + 1 = y} \\ 0, & \text{caso contrário} \end{cases}$$

De forma que a matriz T_n terá a seguinte forma:

$$\begin{bmatrix} \lambda & -1 & 0 & \cdots & 0 \\ -1 & \lambda & -1 & \cdots & 0 \\ 0 & -1 & \lambda & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda \end{bmatrix}$$

O algoritmo usado para criar tal matriz foi o seguinte:

```
public static Double[][] TGn(int lambda, int n){
    Double A[][] = new Double[n][n];

  //preenchendo a matriz com zeros
    for(int linha = 0; linha < n; linha++){
        for(int coluna = 0; coluna < n; coluna++){
            A[linha][coluna] = 0d;
        }
```

```
    }
  //transformando a matriz em Tn
    for(int i = 0; i < n; i++) {
        if(i < n-1){
            A[i][i+1] = -1d;
            A[i+1][i] = -1d;
        }

        A[i][i] = Double.valueOf(lambda);

    }
    return A;
}
```

Listing 1: Método TGn

A função $f_{Tn}(\lambda)$ tem o intuito de fazer o escalonamento da matriz para posteriormente calcular a determinante da mesma. O algoritmo usado para calcular $f_{TGn}(\lambda)$ foi o seguinte:

```
public static double fTGn(double lambda, int n, boolean debug){

        boolean troca = true;
        double m = 0;
        double temp;
        double determinante = 1;
        Double [][] A = TGn(lambda, n);
        Double [][] A_original = TGn(lambda, n);
        int sgn = 1;
        int l = 0;
        int[] t = new int[2];
        boolean erro = false;

        ArrayList<int[]> changeLine = new ArrayList<int[]>();

        for (int j = 0; j < n; j++){
            troca = true;
            l = j;
            while(l < n){
                if(A[l][j] == 0 && l != n - 1){
                    l = l + 1;
                    sgn = -sgn;

                }else if(A[l][j] == 0 && l == n - 1){
                    l = l + 1;
                    sgn = -sgn;
                    troca = false;

                }else if(A[l][j] != 0){
                    break;
                }
            }
            if(troca == false && A[j][j] == 0){
                // System.err.println("Erro: sistema singular");
                erro = true;
                break;
            }
            else if(troca == true && A[j][j] == 0){
                for(int k = j; k < n; k++){
                    temp = A[j][k];
```

```
                    A[j][k] = A[l][k];
                    A[l][k] = temp;
                }
                t[0] = j + 1;
                t[1] = l + 1;
                changeLine.add(t.clone());
            }
            for (int i = j+1; i < n; i++){
                m = - A[i][j]/A[j][j];
                for(int k = j; k < n; k++){
                    A[i][k] = A[i][k] + m*A[j][k];
                }
            }
        }
        for(int i = 0; i < n; i++){
            determinante = determinante*A[i][i];
        }

        if(erro == false){
            determinante = determinante*sgn;
        }
        else{
            determinante = 0;
        }

        if(lambda == 0 && debug){
            System.out.println("V: ");
            printArray(A_original);
            System.out.println("V': ");
            printArray(A);

            if(changeLine.size() == 0){
                System.out.println("trocaLinhas: Nao houveram trocas");
            }else{
                System.out.println("trocaLinhas:");
                changeLine.forEach((d) -> printArray(d));
            }
        }
        return determinante;
    }
```

Listing 2: Método f_{TGn}

Vale ressaltar que os métodos printArray(double A[][]) e printArray(int A[]) usados dentro de $f_{Tn}(\lambda)$, são os seguintes:

```
public static void printArray(Double A[][])
{
    int n = A.length;
    for(int linha = 0; linha < n; linha = linha + 1)
    {
        for(int coluna = 0; coluna < n; coluna = coluna + 1)
        {
            System.out.print("[" + A[linha][coluna] + "]");
        }
        System.out.print("\n");
    }

}
```

```java
public static void printArray(int A[])
{
    int n = A.length;

    for(int i = 0; i < n; i = i + 1)
    {
        System.out.print("[" + A[i] + "]");
    }
    System.out.print("\n");

}
```

Listing 3: Método printArray

Por fim, para calcular $f_{Tn}(\lambda)$ com $\lambda = 0,\ 1,\ 2,\ 3$ *e* 4 e $n = 7$ temos que chamar o seguinte método dentro do método main, nesse método aproveitamos também para exibir o resultado de $f_{Tn}(\lambda)$

```java
public static void exerc1(){

    int x = 4; // valor maximo que lambda pode valer
    int n = 7; //tamanho da matriz
    double det = 0;
    boolean debug  = true; //variavel que permite a impressao da saida

    for(int lambda = 0; lambda <= x; lambda++){

        det = fTGn(lambda, n, debug);
        System.out.println("fT"+n+"("+ lambda +") vale " + det);

    }

}
```

Listing 4: Método exerc1()

Tendo a seguinte resposta:

```
V:
[0.0][-1.0][0.0][0.0][0.0][0.0][0.0]
[-1.0][0.0][-1.0][0.0][0.0][0.0][0.0]
[0.0][-1.0][0.0][-1.0][0.0][0.0][0.0]
[0.0][0.0][-1.0][0.0][-1.0][0.0][0.0]
[0.0][0.0][0.0][-1.0][0.0][-1.0][0.0]
[0.0][0.0][0.0][0.0][-1.0][0.0][-1.0]
[0.0][0.0][0.0][0.0][0.0][-1.0][0.0]
V':
[-1.0][0.0][-1.0][0.0][0.0][0.0][0.0]
[0.0][-1.0][0.0][0.0][0.0][0.0][0.0]
[0.0][0.0][-1.0][0.0][-1.0][0.0][0.0]
[0.0][0.0][0.0][-1.0][0.0][0.0][0.0]
[0.0][0.0][0.0][0.0][-1.0][0.0][-1.0]
[0.0][0.0][0.0][0.0][0.0][-1.0][0.0]
[0.0][0.0][0.0][0.0][0.0][0.0][0.0]
trocaLinhas:
[1][2]
[3][4]
[5][6]
fT7(0) vale 0.0
fT7(1) vale 1.0
```

```
fT7(2) vale 7.999999999999998
fT7(3) vale 987.0000000000001
fT7(4) vale 10864.0
```

Listing 5: Saída do exercício 1

Exercício 2

```
public class Ex2 {
public static void main(String[] args) {

        java.util.Locale.setDefault(new java.util.Locale("en","US"));
  float x;
        float n=0;
        float e,i;
        float vet_x[] = new float[100];
        double cont=0.0,matriz[][] = new double[100][100],y=0.0,vet_det[] = new
    double[100],cont_det=1.0000000;
  int j=0,l=0,m=0,k,g,h,a,b=0,c,d;
  String ver1,ver2;
  n = 11;

        //cria a matriz
    for (i=0;i<3*n+1;i++){
      l = 0;
      cont_det  =  1.0;
      while (l<n){
        matriz[l][l+1]  =  -1.0;
        matriz[l+1][l]  =  -1.0;
        l = l+1;
      }
                        //calcula o valor de x e o insere na diagonal principal
                        e= 4/(3*n);
      x=(-2+(e*i));
      vet_x[m]  =  x;
      m = m+1;
      for (h=0;h<n;h++){
        matriz[h][h]  =  x;
      }

                        //calcula o determinante
      for (g=0;g<n;g++){
        for (j=g+1;j<n;j++){
          y =(matriz[j][g]/matriz[g][g]);
          if (matriz[j][g]!=0){
            for (k=0;k<n;k++){
              matriz[j][k] =( matriz[j][k]-y*matriz[g][k]);
            }
          }
        }
      }

                        //armazena o determinante
      for (a=0;a<n;a++){
        cont_det =( cont_det*matriz[a][a]);
      }
      vet_det[b]  =  cont_det;
```

```
      b = b+1;
      for (c=0;c<n;c++){
        for (d=0;d<n;d++){
          matriz[c][d] = 0;
        }
      }
    }
                  // verifica onde ha troca de sinal e imprime
    for (j=0;j<3*n+1;j++){
      if ((vet_det[j]>0 && vet_det[j+1]<0) || (vet_det[j]<0 && vet_det[j+1]>0)){
        System.out.print( "["+vet_x[j]+", "+vet_x[j+1]+"]"+"\n");
      }
    }
  }
}
```

Os intervalos obtidos foram:

```
[-2.0, -1.8787879]
[-1.7575758, -1.6363636]
[-1.5151515, -1.3939394]
[-1.030303, -0.9090909]
[-0.5454545, -0.42424238]
[-0.060606003, 0.060606003]
[0.4242425, 0.5454545]
[0.909091, 1.030303]
[1.3939395, 1.5151515]
[1.6363637, 1.757576]
[1.878788, 2.0]
```

Exercício 3

Para resolver esse exercício utilizamos os métodos usados no exercício 1: FtGn, TGn e print array para conseguirmos encontrar as raizes aproximadas através do método da bissecção abaixo:

```
public static double bisseccao(double m, double M, int n_max, double prec, int n
    ){

        double alpha = 0.5*(m+M); // x que sera testado
        int count = 0; // numero de interacoes
        boolean debug = false;

        while((fTGn(alpha-prec, n, debug)*fTGn(alpha+prec, n, debug)) > 0
        && count < n_max){

            count = count + 1;
            alpha = 0.5*(m+M);

            // verifica se a raiz esta no intervalo [m, alpha]
            if(fTGn(alpha, n, debug)*fTGn(m, n, debug) < 0
            || fTGn(alpha, n, debug)*fTGn(m, n, debug) == 0 ){
                M = alpha;
            }
            // verifica se a raiz esta no intervalo [alpha, M]
            if(fTGn(alpha, n, debug)*fTGn(m, n, debug) > 0){
                m = alpha;
```

```
            }
        }
        return alpha;
    }
```

Listing 6: Método Bissecção

Para executar esse exercício iremos ter que chamar o seguinte método dentro do main:

```
public static void exerc3(){
    Double intervalos[][] = {
        {-2.0, -1.8787879},
        {-1.7575758, -1.6363636},
        {-1.5151515, -1.3939394},
        {-1.030303, -0.9090909},
        {-0.5454545, -0.42424238},
        {-0.060606003, 0.060606003},
        {0.4242425, 0.5454545},
        {0.909091, 1.030303},
        {1.3939395, 1.5151515},
        {1.6363637, 1.757576},
        {1.878788, 2.0}
    };
    int tamanho_intervalo = intervalos.length;
    double precisao = Math.pow(10,-12);
    double raiz_aprox_bissec = 0;
    double menor, maior = 0;
    int n_max = 50;
    int n = 11;

    for(int j = 0; j < tamanho_intervalo; j++){

        if(j <= tamanho_intervalo - 1){

            menor = intervalos[j][0];
            maior = intervalos[j][1];

            raiz_aprox_bissec = bisseccao(menor,maior,n_max,precisao, n);

            System.out.println("~> intervalo: [" + menor +","+ maior+"]");
            System.out.println("~~~> raiz aprox (bissec): " + raiz_aprox_bissec)
    ;
            System.out.println("");

        }
    }
}
```

Listing 7: Método exerc3()

Os resultados obtidos foram:

```
~> intervalo: [-2.0,-1.8787879]
~~~> raiz aprox (bissec): -1.931851652578306

~> intervalo: [-1.7575758,-1.6363636]
~~~> raiz aprox (bissec): -1.732050807569201

~> intervalo: [-1.5151515,-1.3939394]
```

```
~~~> raiz aprox (bissec): -1.4142135623727792

~> intervalo: [-1.030303,-0.9090909]
~~~> raiz aprox (bissec): -0.9999999999993041

~> intervalo: [-0.5454545,-0.42424238]
~~~> raiz aprox (bissec): -0.5176380902057762

~> intervalo: [-0.060606003,0.060606003]
~~~> raiz aprox (bissec): 0.0

~> intervalo: [0.4242425,0.5454545]
~~~> raiz aprox (bissec): 0.5176380902047707

~> intervalo: [0.909091,1.030303]
~~~> raiz aprox (bissec): 1.0

~> intervalo: [1.3939395,1.5151515]
~~~> raiz aprox (bissec): 1.4142135623726788

~> intervalo: [1.6363637,1.757576]
~~~> raiz aprox (bissec): 1.7320508075681347

~> intervalo: [1.878788,2.0]
~~~> raiz aprox (bissec): 1.93185165257745
```

Listing 8: Saída do exercício 3

Exercício 4

Para resolver esse exercício, primeiramente criamos um vetor de tamanho 7 para armazenar os sete pontos dados no enunciado para calcular a derivada da função. Após isso, criamos um looping para que a derivada fosse calculada em todos esses pontos obedecendo às condições dadas.

```
import java.util.Scanner;
public class derivada {

    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);

        double lambda;
        int n=11;
        int contador=7;
        double derivada;
        int i;

       double vetor[] = new double [7];

      vetor[0]=-1.9;
      vetor[1]=-1.7;
      vetor[2]=-1.4;
      vetor[3]=0;
      vetor[4]=1;
      vetor[5]=1.5;
```

```
      vetor[6]=2;


      for (i=0; i<7;i++) {
        lambda=vetor[i];

        double z=lambda/2;


        //variaveis para cada item do calculo da segunda condicao
        double c = Math.acos(z);
        double x = 12*Math.cos(11*c);
        double y = Math.sin(11*c);
        double j=Math.sin(c);
        double conta = (y/j)*-z;
        double fin = (x+conta)/(z*z-1);

        if (lambda==-2) {
            derivada= Math.pow(-1, n+1)*((Math.pow((n + 1),5) - Math.pow((n+1)
   ,3))/3);
            System.out.println("A derivada eh "+derivada);
        }
        else if (lambda==2) {
            derivada=(Math.pow((n + 1),5) - Math.pow((n+1),3))/3;
            System.out.println("A derivada eh "+derivada);
        }

        else if(Math.abs(lambda)<2){
                derivada = ((n + 1)*Math.cos((n+1)*Math.acos(z)) - z*(Math.sin((
   n+1)*Math.acos(z)))/(Math.sin(Math.acos(z))))/(2*(Math.pow(z,2) - 1));

                System.out.println("A derivada eh "+derivada);
             }
        else {
            System.out.println("Lambda fora do intervalo");
        }

        z=0;
        derivada=0;

     }
    }

}
```

Sabemos que a expressão calculada nesse algoritmo só é válida quando lambda está dentro do intervalo [-2,2], entretanto, achamos pertinente inserir uma condição que avisa o usuário caso ele digite um valor fora desse intervalo.

```
A derivada eh 38.59015868110006
A derivada eh -19.059018406099963
A derivada eh 11.565042073600003
A derivada eh -6.0
A derivada eh -8.0
A derivada eh 10.901367187500009
A derivada eh 82368.0
```

Exercício 5

(2.0) Utilizando a expressão para a derivada $f'_{Tn}(\lambda)$ acima, calcule novamente as raízes de $f_{T11}(\lambda)$ pelo método de Newton, com precisão e = 10^{12}. Em cada intervalo $[x_j, x_{j+1}]$, utilize o ponto x_j como aproximação inicial para o método de Newton.

Exiba os valores retornados pelo método. Comparando com os valores obtidos pelo método da bissecção, quais desses valores representam, de fato, raízes da equação dada?

Em primeiro lugar devemos declara o método de bissecção, que será responsável por retornar as raízes aproximadas da função $f_{Tn}(\lambda)$.

```
public static double bisseccao(double m, double M, int n_max, double prec, int n
    ){

        double alpha = 0.5*(m+M); // x que sera testado
        int count = 0; // numero de interacoes
        boolean debug = false;

        while((fTGn(alpha-prec, n, debug)*fTGn(alpha+prec, n, debug)) > 0
        && count < n_max){

            count = count + 1;
            alpha = 0.5*(m+M);

            // verifica se a raiz esta no intervalo [m, alpha]
            if(fTGn(alpha, n, debug)*fTGn(m, n, debug) < 0
            || fTGn(alpha, n, debug)*fTGn(m, n, debug) == 0 ){
                M = alpha;
            }
            // verifica se a raiz esta no intervalo [alpha, M]
            if(fTGn(alpha, n, debug)*fTGn(m, n, debug) > 0){
                m = alpha;
            }
        }
        return alpha;
    }
```

Listing 9: Método de Bissecção

Logo depois iremos gerar o método que calculará a derivada da função, iremos usar esse valor futuramente dentro do método de Newton

```
public static double fLinha(double lambda, int n){

        double derivada = 1;
        double z;

        if (lambda == 2){

            derivada = (Math.pow((n + 1),5) - Math.pow((n+1),3))/3;

        }else if(lambda == -2){

            derivada = Math.pow(-1, n+1)*((Math.pow((n + 1),5) - Math.pow((n+1)
    ,3))/3);

        }else{
```

```
            z = lambda/2;
            derivada = ((n + 1)*Math.cos((n+1)*Math.acos(z)) - z*(Math.sin((n+1)
    *Math.acos(z)))/(Math.sin(Math.acos(z))))/(2*(Math.pow(z,2) - 1));
        }

        return derivada;
    }
```

Listing 10: Método fLinha

Agora iremos criar o método newton() que será responsável por calcular as raízes aproximadas pelo método de Newton. Isso permitirá com que tenhamos duas aproximações distintas para as raízes da função $f_{Tn}(\lambda)$. Ele será implementado da seguinte forma:

```
public static double newton(double a, double b, double prec, int n_max, double
    x0, int n){

    double alpha = x0;
    int i = 0;

    while(
            fTGn(alpha - prec, n, false)*fTGn(alpha + prec, n, false) > 0 &&
            i <= n_max
        ){

        i = i + 1;
        if(a <= alpha && b >= alpha){
            alpha = alpha - (fTGn(alpha, n, false)/fLinha(alpha, n));
        }

    }

    return alpha;
}
```

Listing 11: Método de Newton

Por fim geraremos o método exerc5(), seu intuito é calcular as 11 raízes da função $f_{Tn}(\lambda)$ pelo método de newton() e pelo método da bisseccao() sabendo que essas raízes estão nos seguintes intervalos: [-2.0, -1.8787879], [-1.7575758, -1.6363636], [-1.5151515, -1.3939394], [-1.030303, -0.9090909], [-0.5454545, -0.42424238], [-0.060606003, 0.060606003], [0.4242425, 0.5454545], [0.909091, 1.030303], [1.3939395, 1.5151515], [1.6363637, 1.757576] e [1.878788, 2.0]. Vale ressaltar que sabemos previamente quais intervalos devemos procurar as raízes pois eles foram informados no exercício 2, da mesma forma que vale ressaltar que futuramente teremos que chamar esse método dentro do main() para que ele possa ser executado.

```
public static void exerc5(){

        // intevalos onde podemos encontrar as n raizes. Esses dados foram
    retirados do exerc. 2 -> para n = 11
        Double[][] intervalos = {
            {-2.0, -1.8787879},
            {-1.7575758, -1.6363636},
            {-1.5151515, -1.3939394},
            {-1.030303, -0.9090909},
            {-0.5454545, -0.42424238},
            {-0.060606003, 0.060606003},
```

```
            {0.4242425, 0.5454545},
            {0.909091, 1.030303},
            {1.3939395, 1.5151515},
            {1.6363637, 1.757576},
            {1.878788, 2.0}
        };

        int tamanho_intervalo = intervalos.length;
        double precisao = Math.pow(10,-12); // precisao dos metodos
        double menor = 0, maior = 0, x0 = 0;
        double raiz_aprox_newton = 0;
        double raiz_aprox_bissec = 0;
        int n_max = 50; // numero maximo de interacoes para os dois metodos
        int n = 11; // tamanho da matriz

        for(int j = 0; j < tamanho_intervalo; j++){

            if(j <= tamanho_intervalo - 1){

                menor = intervalos[j][0]; // comeco do intervalo
                maior = intervalos[j][1]; // fim do intervalo
                x0 = (maior+menor)/2; // aproxima  o para x0, usado no m todo
    de Newton

                // calculo do metodo de Newton para o nosso intervalo atual
                raiz_aprox_newton = newton(menor,maior, precisao, n_max, x0, n);
                // calculo do metodo de bisseccao para o nosso intervalo atual
                raiz_aprox_bissec = bisseccao(menor,maior,n_max,precisao, n);

                // imprimindo os resultados
                System.out.println(">>>> intervalo: [" + menor +", "+ maior+"]")
    ;
                System.out.println("---> raiz aprox. (newton): "
                + raiz_aprox_newton);
                System.out.println("---> raiz aprox. (bissec): "
                + raiz_aprox_bissec);
            }
        }
    }
```

Listing 12: Método exerc5()

Ao chamar esse método no main() teremos a seguinte saída:

```
>>>> intervalo: [-2.0, -1.8787879]
---> raiz aprox. (newton): -1.9318516525781366
---> raiz aprox. (bissec): -1.931851652578306
>>>> intervalo: [-1.7575758, -1.6363636]
---> raiz aprox. (newton): -1.7320508075688776
---> raiz aprox. (bissec): -1.732050807569201
>>>> intervalo: [-1.5151515, -1.3939394]
---> raiz aprox. (newton): -1.4142135623730971
---> raiz aprox. (bissec): -1.4142135623727792
>>>> intervalo: [-1.030303, -0.9090909]
---> raiz aprox. (newton): -1.0000000000000793
---> raiz aprox. (bissec): -0.9999999999993041
>>>> intervalo: [-0.5454545, -0.42424238]
---> raiz aprox. (newton): -0.5176380902050433
---> raiz aprox. (bissec): -0.5176380902057762
```

```
>>>> intervalo: [-0.060606003, 0.060606003]
---> raiz aprox. (newton): 0.0
---> raiz aprox. (bissec): 0.0
>>>> intervalo: [0.4242425, 0.5454545]
---> raiz aprox. (newton): 0.5176380902050433
---> raiz aprox. (bissec): 0.5176380902047707
>>>> intervalo: [0.909091, 1.030303]
---> raiz aprox. (newton): 1.0000000000000793
---> raiz aprox. (bissec): 1.0
>>>> intervalo: [1.3939395, 1.5151515]
---> raiz aprox. (newton): 1.4142135623730971
---> raiz aprox. (bissec): 1.4142135623726788
>>>> intervalo: [1.6363637, 1.757576]
---> raiz aprox. (newton): 1.7320508075688779
---> raiz aprox. (bissec): 1.7320508075681347
>>>> intervalo: [1.878788, 2.0]
---> raiz aprox. (newton): 1.9318516525781366
---> raiz aprox. (bissec): 1.93185165257745
```

Listing 13: Saída do exerc5()

Ao comparar o método da bissecção com o método de Newton chegamos a conclusão de que o método de Newton obtém resultados mais precisos, já que em muita das saídas suas respostas tem mais casas decimais, além disso podemos interpretar que o método de Newton é mais preciso já que o método de bissecção precisa de um maior número de interações para convergir para uma resolução mais próxima da real se comparado com o método de Newton e no exemplo ambos foram executados no máximo n_{max} vezes.

Exercício 6

Repita os cálculos acima, porém, dessa vez, em cada intervalo $[x_j, x_{j+1}]$, utilize o ponto médio $\frac{x_j + x_{j+1}}{2}$ como aproximação inicial para o método de Newton Pode-se provar que as raízes exatas de f_{T11} são

$$\lambda_k = 2 \cdot cos\left(\frac{k\pi}{12}\right), \qquad k = 1, 2, ..., 11$$

Calcule os erros

$$|\lambda_k - \tilde{\lambda_k}|, \qquad k = 1, 2, ..., 11$$

entre as raízes exatas λ_k e as raízes aproximadas $\tilde{\lambda_k}$ calculadas pelo método de Newton e verifique se a precisão e = 10^{12} foi atingida ou não

Para resolver esse exercício, precisamos primeiro definir quais serão nossos intervalos, cada intervalo irá conter uma raiz, logo serão 11 intervalos para $n = 11$. Para definir os intervalos que seriam usados entre [-2, 2] usamos como base o exercício 2 e chegamos no seguinte: [-2.0, -1.8787879], [-1.7575758, -1.6363636], [-1.5151515, -1.3939394], [-1.030303, -0.9090909], [-0.5454545, -0.42424238], [-0.060606003, 0.060606003], [0.4242425, 0.5454545], [0.909091, 1.030303], [1.3939395, 1.5151515], [1.6363637, 1.757576] e [1.878788, 2.0]

Agora que já sabemos nossos intervalos de interesse podemos começar a fazer o exercício. O primeiro passo é criar o método que calculará a derivada da nossa função, que será posteriormente utilizada no método de Newton. Esse método já foi apresentado acima, mas iremos repeti-lo aqui por questões didáticas, bem como o método de newton().

```
public static double fLinha(double lambda, int n){

        double derivada = 1;
        double z;

        if (lambda == 2){

            derivada = (Math.pow((n + 1),5) - Math.pow((n+1),3))/3;

        }else if(lambda == -2){

            derivada = Math.pow(-1, n+1)*((Math.pow((n + 1),5) - Math.pow((n+1)
    ,3))/3);

        }else{
            z = lambda/2;
            derivada = ((n + 1)*Math.cos((n+1)*Math.acos(z)) - z*(Math.sin((n+1)
    *Math.acos(z)))/(Math.sin(Math.acos(z))))/(2*(Math.pow(z,2) - 1));
        }

        return derivada;
    }
```

Listing 14: Método fLinha

O próximo passo é gerar o método de Newton, que será responsável por calcular a aproximação das raízes de $f_{Tn}(\lambda)$ dentro de cada intervalo. Seu código é o seguinte:

```
public static double newton(double a, double b, double prec, int n_max, double
    x0, int n){

    double alpha = x0;
    int i = 0;

    while(
            fTGn(alpha - prec, n, false)*fTGn(alpha + prec, n, false) > 0 &&
            i <= n_max
        ){

        i = i + 1;
        if(a <= alpha && b >= alpha){
            alpha = alpha - (fTGn(alpha, n, false)/fLinha(alpha, n));
        }

    }

    return alpha;
}
```

Listing 15: Método de Newton

Por fim, devemos chamar o seguinte método dentro do main() para executar nosso exercício. Nosso intuito com o método exerc6() é de encontrar a aproximação para as 11 raízes da função pelos métodos de Newton da mesma forma que temos o objetivo de comparar essas raízes com o valor real das suas respectivas raízes.

```
public static void exerc6(){

```

```
        Double[][] intervalos = {
            {-2.0, -1.8787879},
            {-1.7575758, -1.6363636},
            {-1.5151515, -1.3939394},
            {-1.030303, -0.9090909},
            {-0.5454545, -0.42424238},
            {-0.060606003, 0.060606003},
            {0.4242425, 0.5454545},
            {0.909091, 1.030303},
            {1.3939395, 1.5151515},
            {1.6363637, 1.757576},
            {1.878788, 2.0}
        };
        int tamanho_intervalo = intervalos.length;
        double precisao = Math.pow(10,-12);
        double menor = 0, maior = 0, x0 = 0;
        double raiz_exata = 0;
        double raiz_aprox_newton = 0;
        int n_max = 50;
        int n = 11;
        int k = 11;

        for(int j = 0; j < tamanho_intervalo; j++){

            if(j <= tamanho_intervalo - 1){

                menor = intervalos[j][0];
                maior = intervalos[j][1];
                x0 = (maior+menor)/2;

                raiz_exata = 2*Math.cos(k*Math.PI/12);
                raiz_aprox_newton = newton(menor,maior, precisao, n_max, x0, n);

                System.out.println(">>>> intervalo: [" + menor
                                +", "+ maior+"]");
                System.out.println("---> raiz exata: "
                                + raiz_exata);
                System.out.println("---> raiz aprox. (newton): "
                                + raiz_aprox_newton);
                System.out.println("---> erro (newton): "
                                + Math.abs(raiz_exata - raiz_aprox_newton));
                System.out.println("");

            }
            k = k - 1;

        }

    }
```

Listing 16: Método exerc6()

Ao executar o método exerc6() teremos a seguinte resposta:

```
>>>> intervalo: [-2.0, -1.8787879]
---> raiz exata: -1.9318516525781364
---> raiz aprox. (newton): -1.9318516525781366
---> erro (newton): 2.220446049250313E-16

```

```
>>>> intervalo: [-1.7575758, -1.6363636]
---> raiz exata: -1.7320508075688774
---> raiz aprox. (newton): -1.7320508075688776
---> erro (newton): 2.220446049250313E-16

>>>> intervalo: [-1.5151515, -1.3939394]
---> raiz exata: -1.414213562373095
---> raiz aprox. (newton): -1.4142135623730971
---> erro (newton): 2.220446049250313E-15

>>>> intervalo: [-1.030303, -0.9090909]
---> raiz exata: -0.9999999999999996
---> raiz aprox. (newton): -1.0000000000000793
---> erro (newton): 7.971401316808624E-14

>>>> intervalo: [-0.5454545, -0.42424238]
---> raiz exata: -0.5176380902050413
---> raiz aprox. (newton): -0.5176380902050433
---> erro (newton): 1.9984014443252818E-15

>>>> intervalo: [-0.060606003, 0.060606003]
---> raiz exata: 1.2246467991473532E-16
---> raiz aprox. (newton): 0.0
---> erro (newton): 1.2246467991473532E-16

>>>> intervalo: [0.4242425, 0.5454545]
---> raiz exata: 0.5176380902050415
---> raiz aprox. (newton): 0.5176380902050433
---> erro (newton): 1.7763568394002505E-15

>>>> intervalo: [0.909091, 1.030303]
---> raiz exata: 1.0000000000000002
---> raiz aprox. (newton): 1.0000000000000793
---> erro (newton): 7.904787935331115E-14

>>>> intervalo: [1.3939395, 1.5151515]
---> raiz exata: 1.4142135623730951
---> raiz aprox. (newton): 1.4142135623730971
---> erro (newton): 1.9984014443252818E-15

>>>> intervalo: [1.6363637, 1.757576]
>>>> intervalo: [1.6363637, 1.757576]
---> raiz exata: 1.7320508075688774
---> raiz aprox. (newton): 1.7320508075688779
---> erro (newton): 4.440892098500626E-16

>>>> intervalo: [1.878788, 2.0]
---> raiz exata: 1.9318516525781366
---> raiz aprox. (newton): 1.9318516525781366
---> erro (newton): 0.0
```

Listing 17: Saída do exercício 6()

Ao analisar nossa saída podemos perceber que o resultado decorrente do método de Newton obteve ótimas aproximações para as raízes da função $f_{Tn}(\lambda)$ com precisões que chegam a casa de 10^{-16}