Universidade de São Paulo
Instituto de Física de São Carlos
7600017 - Introdução à Física Computacional
Docente: Francisco Castilho Alcaraz

Projeto 03:
Métodos básicos integro-diferenciais

Andressa Colaço
Nº USP: 12610389

São Carlos
2022

Sumário

1 Tarefa A

A primeira tarefa deste projeto tem o objetivo de encontrar o valor das derivadas para $x = 0.5$ da função:

$$f(x) = cosh(3x)sin(x/2) \tag{1}$$

utilizando os métodos do cálculo numérico.

Em primeiro momento, temos que as expressões analíticas para as derivadas primeira, segunda e terceira são:

$$f'(x) = 3sinh(3x)sin(x/2) + \frac{1}{2}cosh(3x)cos(x/2) \tag{2}$$

$$f''(x) = \frac{35}{4}cosh(3x)sin(x/2) + 3sinh(3x)cos(x/2) \tag{3}$$

$$f'''(x) = \frac{99}{4}sinh(3x)sin(x/2) + \frac{107}{8}cosh(3x)cos(x/2) \tag{4}$$

Estas serão utilizadas para encontrar o desvio que obtemos com os diferentes testes realizados pelo programa.

As equações do cálculo numérico que serão avaliadas serão as seguintes (as siglas entre parênteses são os nomes das variáveis que as representam no código):

Derivada simétrica de 3 pontos (DS3):

$$f' = \frac{f_1 - f_{-1}}{2h} + O(h^2) \tag{5}$$

Derivada para a frente de 2 pontos (DF2):

$$f' = \frac{f_1 - f_0}{h} + O(h) \tag{6}$$

Derivada para trás de 2 pontos (DT2):

$$f' = \frac{f_1 - f_{-1}}{h} + O(h^2) \tag{7}$$

Derivada simétrica de 5 pontos (DS5):

$$f' = \frac{-f_{-2} - 8f_{-1} + 8f_1 - f_2}{12h} + O(h^4) \tag{8}$$

Derivada segunda simétrica de 5 pontos (D2S5):

$$f'' = \frac{-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2}{12h^2} + O(h^4) \tag{9}$$

Derivada terceira anti-simétrica de 5 pontos (D3AS5):

$$f''' = \frac{-f_{-2} + 2f_{-1} - 2f_1 + f_2}{2h^3} + O(h^2) \tag{10}$$

Onde a notação f_n se refere à expressão $f_n = f(x + n \cdot h)$. É notável que todas as expressões utilizam-se de um parâmetro h o qual ainda não foi definido: este valor deve ser escolhido de acordo com a função que se deseja derivar de forma a obter o menor erro possível para as derivadas. Para avaliar quais valores de h são melhores para este caso, é escrito um código em fortran que calcula, através das expressões acima, os valores das derivadas da função para $h = 0.5, 0.2, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005, 0.00001, 0.000001, 0.0000001$ e 0.00000001 e seus respectivos erros em relação às expressões analíticas, calculadas com precisão 10^{-11}:

```
1         PROGRAM calculo_derivadas
2
3         implicit real*8 (a-h, o-z)
4         parameter(x = 0.5d0)
5
6         dimension h(1:14)
7         parameter(h = (/0.5d0, 0.2d0, 1d-1, 5d-2, 1d-2, 5d-3, 1d-3, 5d-4,
8        $ 1d-4, 5d-5, 1d-5, 1d-6, 1d-7, 1d-8/))
9
10   11   format(A8, 2A24, A20, A16, 2A22, /)
11   12   format(e16.8, 6e20.11)
12   13   format(/, A16, 6e20.11)
13
14        open(10, FILE='saida-a-12610389.dat')
15
16   c    Expressoes exatas
17        d1 = 3*dsinh(3*x)*dsin(x/2d0)+0.5d0*dcosh(3*x)*dcos(x/2)
18        d2 = (12*dcos(x/2d0)*dsinh(3*x)+35*dsin(x/2)*dcosh(3*x))*0.25d0
19        d3 = (198*dsin(x/2)*dsinh(3*x)+ 107*dcos(x/2)*dcosh(3*x))*0.125d0
20
21        write(10, 11) 'h', 'D. 3 pts', 'D. p/ frente 2 pts',
22       $ 'D. p/ trás 2 pts', 'D. 5 pts', 'D. 2ª 5 pts', 'D. 3ª 5 pts'
23
24        do i = 1, 14
25
26             f0 = f(x, 0, h(i))
27             f1 = f(x, 1, h(i))
28             fm1 = f(x, -1, h(i))
29             f2 = f(x, 2, h(i))
```

```
30              fm2 = f(x, -2, h(i))
31
32              ds3 = (f1 - fm1)/(2d0*h(i))
33              df2 = (f1 - f0)/h(i)
34              dt2 = (f0 - fm1)/h(i)
35              ds5 = (fm2 - 8*fm1 + 8*f1 - f2)/(12d0*h(i))
36              d2s5 = (-fm2 + 16*fm1 - 30*f0 + 16*f1 - f2)/(12d0*(h(i))**2)
37              d3as5 = (-fm2 + 2*fm1 - 2*f1 + f2)/(2d0*(h(i))**3)
38
39              eds3 = abs(d1-ds3)
40              edf2 = abs(d1-df2)
41              edt2 = abs(d1-dt2)
42              eds5 = abs(d1-ds5)
43              ed2s5 = abs(d2-d2s5)
44              ed3as5 = abs(d3-d3as5)
45
46              write(10, 12) h(i), eds3, edf2, edt2, eds5, ed2s5, ed3as5
47          end do
48
49          write(10, 13) 'Exatas:', d1, d1, d1, d1, d2, d3
50
51          end program
52
53          function f(x, n, h)
54             implicit real*8(a-h, o-z)
55
56             x_novo = x + n*h
57             f = dcosh(3.0d0*x_novo)*dsin(x_novo/2.0d0)
58          return
59          end function
```

Neste programa, vemos que nas primeiras linhas são definidos os parâmetros, a formatação e as expressões de comparação para o erro. Começa-se definido a precisão dupla para todas as variáveis reais e colocando-se como parâmetro o x escolhido. Em sequência, os valores de h que serão utilizados são colocados em um vetor, a ser percorrido durante a execução dos cálculos. Como este código gera uma tabela em um arquivo *.dat*, definimos a formatação desta: a linha identificada com 11 define a formatação de texto do cabeçalho; a 12 a formatação das linhas com os dados obtidos ($e16.8$ é a formatação de h, que não precisa de tantos dígitos na representação e $e20.11$ é a formatação dos erros); e a 13 define a formatação dos valores exatos das derivadas. Em sequência, o arquivo de saída é

aberto e se define os valores das expressões analíticas, com as funções em dupla precisão, derivadas manualmente. Antes de entrar no loop de cálculos, o cabeçalho é escrito no arquivo.

Iniciando os cálculos, temos um laço que percorre o vetor com as opções de h que possuímos. Para cada um, começamos calculando os valores de f_0, f_1, f_{-1}, f_2 e f_{-2}, chamando a função definida ao final do programa. Isso é feito para evitar chamar estas funções desnecessariamente, uma vez que, durante o cálculo das derivadas os mesmos valores aparecem múltiplas vezes. Após os cálculos, os erros são calculados através da diferença dos valores obtidos e dos valores exatos calculados antes do loop. Por fim, os erros são escritos no arquivo junto com o valor de h utilizado.

Ao final, os valores exatos também são escritos no arquivo.

O código resulta na seguinte tabela (dividida em duas apenas aqui para melhor visualização dos dados):

Tabela 1: Tabela com valores de h e os respectivos erros gerados para cada derivada.

h	Derivada simétrica 3 pontos	Derivada para frente 2 pontos	Derivada para trás 2 pontos
0.50000000E+00	0.21066783236E+01	0.57693816932E+01	0.15560250460E+01
0.20000000E+00	0.29727941484E+00	0.14753862835E+01	0.88082745381E+00
0.10000000E+00	0.72980975766E-01	0.64324067537E+00	0.49727872384E+00
0.50000000E-01	0.18162341135E-01	0.30097486969E+00	0.26465018743E+00
0.10000000E-01	0.72543510181E-03	0.57140167455E-01	0.55689297251E-01
0.50000000E-02	0.18135051068E-03	0.28386409815E-01	0.28023708794E-01
0.10000000E-02	0.72539146787E-05	0.56481181412E-02	0.56336103119E-02
0.50000000E-03	0.18134777444E-05	0.28222432842E-02	0.28186163287E-02
0.10000000E-03	0.72539239948E-07	0.56415835341E-03	0.56401327493E-03
0.50000000E-04	0.18135536184E-07	0.28206104123E-03	0.28202477016E-03
0.10000000E-04	0.71058536832E-09	0.56409288394E-04	0.56407867223E-04
0.10000000E-05	0.33263614085E-10	0.56409544134E-05	0.56410209406E-05
0.10000000E-06	0.97695318502E-09	0.56301634332E-06	0.56497024969E-06
0.10000000E-07	0.56843849627E-08	0.72297766440E-07	0.60928996515E-07
Exatas:	0.27200159512E+01	0.27200159512E+01	0.27200159512E+01

Tabela 2: Tabela com valores de h e os respectivos erros gerados para cada derivada.

h	Derivada simétrica 5 pontos	Derivada 2ª simétrica 5 pontos	Derivada 3ª anti-simétrica 5 pontos
0.50000000E+00	0.14953176672E+01	0.13930924069E+01	0.42924442382E+02
0.20000000E+00	0.29645512679E-01	0.29781890786E-01	0.55152777314E+01
0.10000000E+00	0.17851705916E-02	0.18137228867E-02	0.13362264183E+01
0.50000000E-01	0.11053707552E-03	0.11262443369E-03	0.33144630861E+00
0.10000000E-01	0.17633168392E-06	0.17982254086E-06	0.13224613322E-01
0.50000000E-02	0.11019688273E-07	0.11230733676E-07	0.33058936435E-02
0.10000000E-02	0.17618795312E-10	0.24231283646E-09	0.13222165163E-03
0.50000000E-03	0.11781686737E-11	0.58463101027E-09	0.34077936256E-04
0.10000000E-03	0.42632564146E-13	0.25823698735E-07	0.10980696774E-03
0.50000000E-04	0.15232259898E-11	0.12204302813E-06	0.49838502635E-03
0.10000000E-04	0.16610712805E-10	0.70676048836E-06	0.33034807631E+00
0.10000000E-05	0.61019189701E-10	0.39798156613E-03	0.67498841066E+02
0.10000000E-06	0.12545089412E-08	0.10177196041E-01	0.11097877900E+06
0.10000000E-07	0.93851282301E-08	0.35212575114E+01	0.11102234599E+09
Exatas:	0.27200159512E+01	0.11281716150E+02	0.43523461396E+02

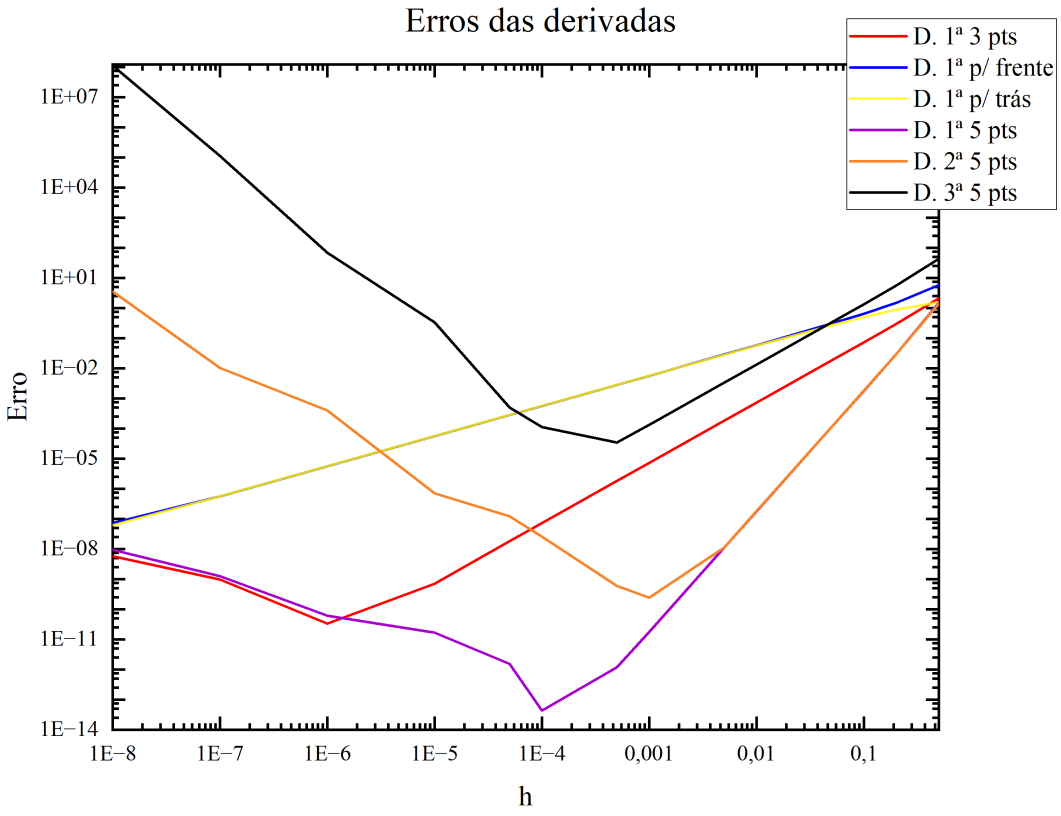
Podemos ver que, em alguns casos, conforme h diminui, a precisão do cálculo aumenta. Porém, na maioria deles, a precisão aumenta até certo ponto com a diminuição de h e depois volta a diminuir. Analisando por um contexto geral, vemos que para as derivadas que utilizam 5 pontos, o ideal é utilizar $h = 5 \cdot 10^{-4}$ ou $h = 1 \cdot 10^{-4}$. Para as derivadas primeiras de 2 e 3 pontos é interessante utilizar valores menores de h, que resultam em erros menores. Para cada uma delas, temos:

Tabela 3: Melhor escolha de h para cada derivada.

Derivada simétrica 3 pontos	Derivada para frente 2 pontos	Derivada para trás 2 pontos	Derivada simétrica 5 pontos	Derivada 2ª simétrica 5 pontos	Derivada 3ª anti-simétrica 5 pontos
0.1-05	0.1E-07	0.1E-07	0.1E-03	0.1E-02	0.5E-03

Podemos ter uma visualização melhor deste comportamento através do seguinte gráfico:

Figura 1: Erros das derivadas em relação a h.



Através dele, podemos ver que a maioria delas tem valor alto para valores bem pequenos de h e diminuem conforme este aumenta. Em certo ponto, atingem um mínimo e continuam a aumentar conforme h aumenta. Só não verificamos esse comportamento para as curvas das primeiras derivadas de 2 pontos (que no gráfico parecem uma linha verde/amarela, pois devido ao comportamento similar elas se sobrepoem), que continuam diminuindo juntamente com h. Isso parece indicar que ainda não encontramos o h ideal para estas duas derivadas, já que testamos valores apenas até o limite de 10^{-8}, devendo este ser ainda menor.

Isto ocorre por conta do limite físico que possuímos nas computações: quando h é pequeno, os erros de arredondamento que precisam ser performados para representar o valor são mais significativos, em especial na execução de operações aritméticas. Ao aumentar h, reduzimos esse efeito, mas aproximamos a função com uma precisão menor. A questão levantada nesta tarefa foi justamente observar onde e como o balanço dos dois fatores ocorres, minimizando o erro dos resultados.

2 Tarefa B

Na segunda tarefa, aproximaremos o valor da seguinte integral:

$$\int_0^1 \exp(x/4)\sin(\pi x)dx \tag{11}$$

através dos métodos de integração numéricos da regra do trapézio, regra de Simpson e regra de Boole. Estes pertencem às chamadas fórmulas de Newton-Cotes, que utilizam polinômios de Lagrange para aproximar o valor da integral em um dado intervalo.

A regra do trapézio, que utiliza um polinômio de Lagrange de ordem 1, consiste em calcular aproximações da área sob uma função em um determinado intervalo através do cálculo da área de um trapézio cujos vértices são os pontos do intervalo no eixo x e os pontos que marcam suas alturas. Para melhorar um pouco a escrita do código, podemos considerar a cada iteração a soma das áreas de dois intervalos, que possui como expressão numérica:

$$\int_{-h}^{h} f(x)dx = \frac{h}{2}(f_{-1} + 2f_0 + f_1) + O(h^3) \tag{12}$$

A regra de Simpson (1/3) resulta da integração da função f(x) no intervalo $[a, b]$ a partir da aproximação polinomial de segundo grau de Lagrange, com pontos igualmente espaçados que formam pares de subintervalos. dividindo nosso intervalo grande por N, podemos percorrer a cada iteração dois subintervalos, obtendo aproximações através da expressão:

$$\int_{-h}^{h} f(x)dx = \frac{h}{3}(f_{-1} + 4f_0 + f_1) + O(h^5) \tag{13}$$

Este dá o resultado exato da integral quando aplicado a qualquer polinômio com grau três ou menos.

Por fim, a regra de Boole utiliza um polinômio de Lagrange de grau 4, utilizando a expressão abaixo para calcular a área em um subintervalo utilizando 5 pontos:

$$\int_{-h}^{h} f(x)dx = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4) + O(h^7) \tag{14}$$

Onde, da mesma forma que na diferenciação, $f_n = f(x + n \cdot h)$.

Para comparar os métodos, vamos calcular os valores das aproximações da integral acima produzidos por cada um deles para diferentes números de partições N. O tamanho de cada partição, ou melhor, de cada intervalo utilizado, é dado por $h = (b - a)/N$ (b e a sendo os limites de integração), ou melhor, $h = 1/N$, já que $b - a = 1 - 0 = 1$.

Além disso, analiticamente podemos obter o valor exato desta integral através da avaliação da antiderivada nos limites de integração:

$$\int_0^1 \exp(x/4)\sin(\pi x)dx = \left.\frac{4\exp(x/4)(\sin(\pi x) - 4\pi\cos(\pi x)}{16\pi^2 + 1}\right|_0^1 = 0.72245288409 \tag{15}$$

O qual será utilizado para avaliar o erro dos métodos testados.

É então produzido o seguinte código:

```
1           PROGRAM integracao_numerica
2
3           implicit real*8 (a-h, o-z)
4           parameter(a = 0d0)
5           parameter(b = 1d0)
6
7     11    format(A6, A10, 3A20, /)
8     12    format(I8, f12.8, 3e20.11)
9     13    format(/, A30, e22.11)
10
11          open(10, FILE='saida-b-12610389.dat')
12
13          N = 12
14          val_exato = antideriv(b)-antideriv(a)
15
16          write(10, 11) 'N', 'h', 'Trapézio', 'Simpson', 'Boole'
17
18          do i = 1, 10
19               h = 1d0/N
20               area_trap = 0d0
21               area_simp = 0d0
22               area_boole = 0d0
23
24               do j = 1, N-1, 2
25                    x0 = a+j*h
26
27                    f0 = f(x0, 0, h)
28                    f1 = f(x0, 1, h)
29                    fm1 = f(x0, -1, h)
30
31                    area_trap = area_trap + (h/2d0)*(fm1+2*f0+f1)
32                    area_simp = area_simp + (h/3d0)*(fm1+4*f0+f1)
33
34               end do
35
36               do k = 0, N-4, 4
37                    x0 = a+k*h
38
```

```
39                f0 = f(x0, 0, h)
40                f1 = f(x0, 1, h)
41                f2 = f(x0, 2, h)
42                f3 = f(x0, 3, h)
43                f4 = f(x0, 4, h)
44
45                area_boole = area_boole +((2*h)/45d0)*(7d0*f0+32d0*f1+
46     $          12d0*f2+32d0*f3+7d0*f4)
47
48           end do
49
50           write(10, 12) N, h, abs(val_exato-area_trap),
51     $     abs(val_exato-area_simp), abs(val_exato-area_boole)
52
53           N = N*2
54      end do
55
56      write(10, 13) 'Valor exato: ', val_exato
57
58      close(10)
59
60      end program
61
62      function f(x, n, h)
63          implicit real*8 (a-h, o-z)
64          parameter(pi = dacos(-1d0))
65
66          x_n = x + n * h
67          f = dexp(x_n/4d0)*dsin(pi*x_n)
68
69      return
70      end function
71
72      function antideriv(x)
73          implicit real*8 (a-h, o-z)
74          parameter(pi = dacos(-1d0))
75
76          antideriv = (4*dexp(x/4d0)*(dsin(pi*x)-4*pi*dcos(pi*x)))/
77     $                ((16*pi**2)+1)
78
```

```
79          return
80          end function
```

Este começa declarando a precisão dupla para variáveis reais que será utilizada no programa inteiro, bem como os valores dos limites de integração. Em sequência, descreve as formatações utilizadas para a impressão da tabela final no arquivo de saída, aberto na linha seguinte. É declarado o N inicial (será incrementado durante o loop principal, uma vez que os valores de N testados aumentam sempre por um fator de 2) e é calculado o valor exato da integral com a expressão derivada anteriormente, definida em uma função separada no final do código.

Entra-se, então, no loop principal, ao qual iteração corresponde a um número de partições diferentes testadas para as aproximações. As áreas são inicializadas em 0 e então calcula-se seguindo o seguinte procedimento: cria-se um loop que percorre cada partição e calcula a aproximação para aquela região através do valor de $x_0 = a + n * h$, onde n indica quantas partições foram percorridas.

Para a regra do trapézio e a regra de Simpson o cálculo sempre é feito tomando um ponto antes e um depois do central. Por conta disso, a iteração começa no ponto $x_1 = x + h$ e vai até $x_{N-1} = x + (N-1)h$ em incrementos de 2, que correspondem ao passo necessário para não contar duas vezes a área da mesma região no somatório. Assim como na diferenciação, as funções são calculadas antes da expressão para não chamar a função mais vezes que o necessário.

Como a regra de Boole segue um passo diferente, optou-se por fazer um loop separado para calculá-lo. Já que ele só utiliza pontos à frente do ponto inicial, nossa iteração começa em $x_0 = a$ e vai até $x_{N-4} = x + (N-4)h$ e segue em passos de tamanho 4. Neste caso, não há muito ganho em calcular as funções de antemão, mas manteve-se esse formato para continuar o padrão dos outros códigos.

Ao fim de todas as iterações, os erros são calculados com o valor exato calculado com as antiderivadas da função e impressos no arquivo de saída. Também é impresso o valor exato da integral em precisão 10^{-11} (cabe notar que, apesar dos cálculos das integrais serem em dupla precisão, os erros são dados com a mesma precisão do valor exato, uma vez que não faria sentido estes serem mais precisos que nosso padrão de comparação).

A saída do código se encontra na seguinte tabela:

Tabela 4: Erros obtidos nas integrações numéricas.

N	h	Trapézio	Simpson	Boole
12	0.08333333	0.41571359729E-02	0.18759302373E-04	0.48003127096E-06
24	0.04166667	0.10384097929E-02	0.11656005068E-05	0.73129511335E-08
48	0.02083333	0.25954789053E-03	0.72743578161E-07	0.11355028029E-09
96	0.01041667	0.64883564023E-04	0.45448130814E-08	0.17715828804E-11
192	0.00520833	0.16220677987E-04	0.28402491470E-09	0.27866597918E-13
384	0.00260417	0.40551561835E-05	0.17751800030E-10	0.44408920985E-15
768	0.00130208	0.10137882135E-05	0.11098899577E-11	0.22204460493E-15
1536	0.00065104	0.25344700150E-06	0.69166894434E-13	0.00000000000E+00
3072	0.00032552	0.63361746738E-07	0.49960036108E-14	0.44408920985E-15
6144	0.00016276	0.15840436296E-07	0.44408920985E-15	0.22204460493E-15

Valor exato: 0.72245288409

Podemos observar que a regra do Trapézio é a que apresenta os maiores erros, seguida pela regra de Simpson. A regra de Boole é a mais precisa das três, assim como esperado de acordo com a ordem dos erros nas expressões numéricas dos métodos. Porém, a regra de Boole demanda o cálculo de expressões maiores, o que pode reduzir a performance do código para o caso de funções muito complicadas.

Também é notável que, em todos os casos há a tendência do erro diminuir conforme se aumenta o número de partições. Isto é compreensível e esperado, uma vez que estamos nos aproximando cada vez mais das curvas da função e computando mais pontos, tal qual é idealizado o procedimento das somas de Riemann. Nos casos como o método de Boole, que utiliza polinômios de grau mais alto para as computações, podemos ter oscilações como as que vemos nos últimos valores deste, por conta da utilização destas funções.

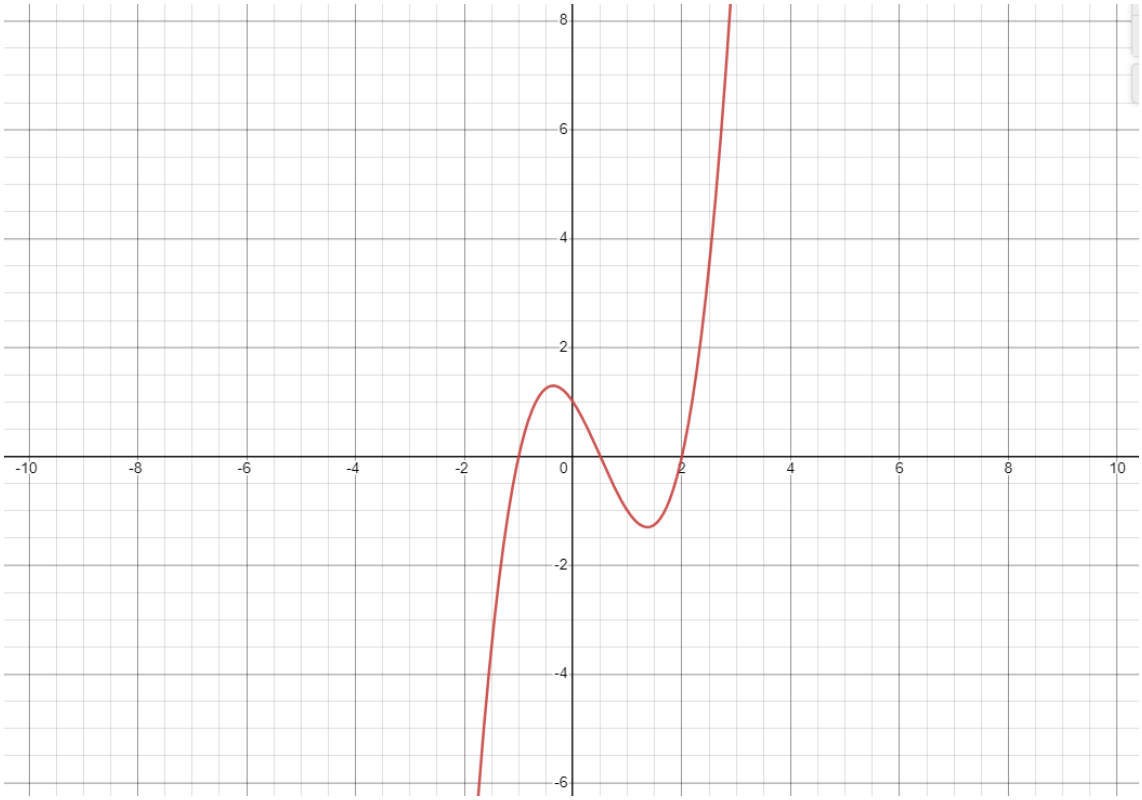
3 Tarefa C

Na terceira tarefa, vamos utilizar e comparar métodos diferentes para a procura de raízes de um polinômio. O polinômio a ser analisado é:

$$x^3 - 3/2x^2 - 3/2x + 1 \tag{16}$$

Pode-se fazer uma análise preliminar testando alguns valores ou criando o gráfico deste para ter uma noção do intervalo no qual as raízes se encontram:

Figura 2: Plot de $x^3 - 3/2x^2 - 3/2x + 1$.



Aqui, podemos ver que todas as raízes são valores reais e relativamente próximos de 0. Para determinar as raízes sem o auxílio desta ferramenta, ou para casos mais complicados de polinômios, serão investigados os seguintes métodos: busca direta, método de Newton-Raphson e método da Secante.

A busca direta consiste em percorrer intervalos, verificando se há mudança de sinal da função calculada nos extremos de cada um. Caso haja, podemos dividir esse intervalo e verificar em qual das partes há a mudança de sinal e assim repetidamente, até que a diferença entre os valores de $f(x)$ calculada nos extremos seja menor do que um certo valor, a tolerância da busca.

O método de Newton-Raphson consiste em eleger um valor inicial de procura e calcular o valor da função neste ponto juntamente com a expressão para sua derivada. A partir disso, encontra-se o ponto onde a derivada corta o eixo x e este torna-se o novo x, para o qual também é calculado o valor da função e sua derivada e onde esta corta o eixo x é eleito um novo x. Este processo se repete até que o valor da função esteja suficientemente próximo a 0, o que indica que encontramos uma raiz com uma determinada precisão, fornecida por nós. Caso aconteça de, após muitas iterações, nos afastarmos da raiz, podemos combinar este método com a busca direta, que se torna muito mais eficiente estando próxima da raiz. A expressão a ser calculada a cada iteração é:

$$x_{i+1} = x_n - \frac{f(x_i)}{f'(x_i)} \tag{17}$$

Este método, no entanto, depende do conhecimento da expressão para a derivada em cada ponto, algo que pode não ser conhecido para todas as funções que se deseje achar a raiz. Portanto, o método da secante utiliza uma lógica similar ao do método de Newton, porém aproxima a derivada da função em um ponto por:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \tag{18}$$

Portanto, a expressão a ser calculada a cada iteração é:

$$x_{i+1} = x_n - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} \tag{19}$$

Como é possível notar na expressão, o método da secante depende da utilização de um ponto adicional. Este ponto influencia na quantidade de iterações que são necessárias para encontrar a raiz, assim como o primeiro ponto. Em determinados casos, certas escolhas pode fazer o programa cair em loop infinito, então deve-se escolher um valor adequado e fazer a verificação durante o código, já que nem todas as funções terão o mesmo comportamento, ainda mais se for o caso de não termos conhecimento do gráfico dela. Isto será um ponto melhor especificado na explicação do código.

A implementação feita foi a seguinte:

```
1         PROGRAM raizes_polinomio
2         implicit real*8(a-h, o-z)
3
4         parameter(eprec = 1e-6)
5         parameter (a = -10)
6         parameter(b = 10)
7         parameter(h = 0.1d0)
8         parameter(h_n = 1d0)
9
10        open(10, FILE='saida-c-12610389.dat')
11  11    format(/, A24, /)
12  12    format(I4, f20.11)
13  13    format(A20, f20.11, A20, I2)
14
15  c     Método de busca direta
16
17        write(10, 11) 'Busca direta'
18        write(*, *) 'Busca direta'
19
20        N = (b-a)/h
21
```

```
22          do n = 0, N-1
23
24               xe = a + n*h
25               xd = xe + h
26
27               fe = f(xe)
28               fd = f(xd)
29
30               if (fe * fd .lt. 0) then
31                    i = 0
32
33    20             if (abs(f(xe)-f(xd)) .gt. eprec) then
34                         xm = (xe+xd)/2.d0
35
36                         if (f(xd)*f(xm) .gt. 0) then
37                              xd = xm
38                         else
39                              xe = xm
40                         end if
41
42                         i = i+1
43                         write(10, 12) i, xm
44
45                         goto 20
46                    end if
47
48               write(*, 13) 'Raiz encontrada: ', xm, 'Iterações: ', i
49               write(10, 13) 'Raiz encontrada: ', xm
50
51               end if
52          end do
53
54    c     Método de Newton Raphsen
55          write(10, 11) 'Método de Newton-Raphson'
56          write(*, *) 'Método de Newton-Raphson'
57
58          x = a
59          ult_raiz = 0d0
60          n_raizes = 0
61
```

```
30      if (n_raizes < 3) then
             i = 0
             x_ant = x

40          if (abs(f(x)) > eprec) then
                if (df(x) .ne. 0) then
                     x = x - (f(x)/df(x))
                else
                     x = x+h
                end if

                i = i+1
                write(10, 12) i, x
                goto 40
            end if

            if (abs(x-ult_raiz) > eprec .or. n_raizes .eq. 0) then
                write(*, 13) 'Raiz encontrada: ', x, 'Iterações: ', i
                write(10, 13) 'Raiz encontrada: ', x

                n_raizes = n_raizes+1
                ult_raiz = x
                x_ant = ult_raiz
            end if

            x = x_ant + h_n
            goto 30
         end if

c       Método da Secante
        write(10, 11) 'Método da Secante'
        write(*, *) 'Método da Secante'

        x = a
        xnm1 = a+h_n
        ult_raiz = 0d0
        n_raizes = 0

50      if (n_raizes < 3) then
            i = 0
```

```
102             x_ant = x
103
104 60          if (abs(f(x)) > eprec) then
105                  if ((f(x)-f(xnm1)) .ne. 0) then
106                      xnp1 = x - (f(x)*(x-xnm1)/(f(x)-f(xnm1)))
107
108                      xnm1 = x
109                      x = xnp1
110                  else
111                      xnm1 = x
112                      x = x+h
113                  end if
114
115                  i = i +1
116
117                  write(10, 12) i, x
118                  goto 60
119             end if
120
121             if (abs(x-ult_raiz) > eprec .or. n_raizes .eq. 0) then
122                 write(*, 13) 'Raiz encontrada: ', x, 'Iterações: ', i
123                 write(10, 13) 'Raiz encontrada: ', x
124
125                 n_raizes = n_raizes+1
126                 ult_raiz = x
127                 x_ant = ult_raiz
128             end if
129
130             x = x_ant + 2*h_n
131             xnm1 = x - h_n
132             goto 50
133          end if
134
135         end program
136
137         function f(x)
138             implicit real*8(a-h, o-z)
139             f = x**3-(1.5d0)*x**2-(1.5d0)*x+1
140         return
141         end function
```

```
142
143         function df(x)
144             implicit real*8(a-h,o-z)
145             df = 3*x**2-3*x-(1.5d0)
146         return
147         end function
```

O código foi construído da seguinte maneira: os três métodos estão implementados em sequência e define-se no cabeçalho parâmetros que serão utilizados por todos: a tolerância para a parada das iterações nos métodos (*eprec*), o valor inicial da procura das raízes a, um parâmetro auxiliar b utilizado na busca direta para estabelecer um limite de procura e o tamanho do espaçamento de procura da busca direta. Em sequência, é aberto um arquivo de saída e as formatações de impressão são definidas. A expressão do polinômio e sua derivada são definidas ao final do código.

Iniciando pela busca direta, temos o cálculo do número de partições que são feitas no intervalo $[a, b]$ e entramos em um laço que caminha por todas as partições realizando o seguinte procedimento: se $f(x_n) \cdot f(x_{n+1})$ for menor que 0, isto indica que a função mudou de sinal dentro do intervalo, portanto, há uma raiz contida ali e podemos realizar sua computação. Caso contrário, incrementamos x_n e x_{n+1}, passando para analisar o próximo intervalo.

O processo de procura da raiz quando se encontra um intervalo elegível se dá da seguinte forma: fazemos um loop que para quando atingimos a tolerância desejada para a raiz, 10^{-6}. A cada iteração, calculamos o ponto médio do intervalo e verificamos em qual das duas partições criadas está a raiz, diminuindo o intervalo de busca para o qual verificamos a mudança de sinal. Ao final, imprimimos a quantidade de iterações necessária para encontrar a raiz e seu valor. Neste método recuperamos todas as raízes por padrão, já que vamos de intervalo em intervalo em um intervalo maior que contém todas as raízes. Para determinar esse intervalo, pode-se tentar esboçar o gráfico ou analisá-lo já plotado. Porém, esta última opção pode ser difícil em casos onde a função é muito complicada. Neste caso, temos uma função simples, com raízes que podem ser encontradas facilmente, todas relativamente próximas de $x = 0$, então houve a certeza que todas as raízes se encontravam no intervalo e pertenciam aos reais.

Prosseguindo para o método de Newton-Raphson, temos uma lógica e variáveis um pouco diferentes que no método da raiz. *A priori*, o método de Newton toma um ponto inicial e encontra uma raiz (às vezes pode ocorrer de não retornar uma raiz, ficando "preso" entre dois pontos), mas pára neste momento. Ele não encontra todas as raízes como a busca direta e devemos escolher outro ponto inicial que "caia"na zona adequada para convergir para outra raiz.

Em uma tentativa de generalizar este processo, criou-se este código: ele possui um contador para o número de raízes encontradas e deve parar ao encontrar todas elas (rei-

terando o conhecimento de que esta função tem somente raízes reais, que são os números que estamos trabalhando). Enquanto não são todas encontradas, definimos uma variável x_{ant} que guarda o valor de x antes dos cálculos das raízes, já que este é alterado. É calculado o valor da raiz seguindo a fórmula do método de Newton, tomando o ponto de cruzamento da derivada da função no ponto x como o próximo x até que se encontre uma raiz dentro da tolerância desejada. Em sequência, o valor encontrado da raiz é comparado com o último valor encontrado para a raiz, caso já tenha sido encontrada uma raiz ($n\_raizes \neq 0$). Caso este seja maior do que a tolerância (isto é, é uma raiz diferente), incrementamos o número de raízes e definimos uma variável ult_raiz para guardar x. Também fazemos $x_{ant} = x$, já que: caso seja encontrada uma raiz, o valor inicial de x para buscar a próxima raiz deve ser o valor da raiz mais um incremento. Caso não seja encontrada, isto é, os valores dos incrementos fazem o processo encontrar a mesma raiz de antes (não saiu da "bacia" desta raiz), apenas incrementa-se o valor original de x antes de encontrar esta raiz repetida.

O código foi desenvolvido desta maneira para poder servir para vários incrementos, porém, na prática, isto significa que, para incrementos muito pequenos pode ser necessário computar várias vezes a mesma raiz até encontrar uma diferente. Uma opção seria analisar o gráfico da função e escolher um valor adequado. Outra foi rodar o código algumas vezes com incrementos diferentes, percebendo que, para $h_n = 1$, encontramos todas as raízes em sequência, pois adicionar $x = 1$ a cada raiz nos faz cair na região que converge para a próxima. Portanto, poderia-se apagar a parte de comparação com a raiz anterior, mas optou-se por deixar o código desta maneira para ficar mais geral, ainda que, se utilizar valores de h_n muito pequenos ou grandes podemos cair em indeterminação ou precisar de muitas iterações para atingir o resultado.

Por fim, o método da secante foi criado da mesma maneira que o método de Newton, porém teve-se que adicionar um termo $xnm1$ para ser o ponto que auxilia na determinação da secante. A implementação é bem similar no ponto de vista da lógica de como se faz para encontrar todas as raízes, mas, ao final, incrementa-se as duas variáveis que servirão como pontos iniciais para as iterações. O método da secante é um pouco mais sensível com a escolha de pontos: observou-se que, para este ponto inicial, uma opção de segundo ponto inicial era algum que imaginamos estar antes da primeira raiz. Manteve-se o mesmo h_n do método anterior, pois observou-se que este se adequa bem à função de nosso interesse.

Optou-se por gerar duas saídas: uma resumida, no terminal, apenas com as raízes e número de iterações obtidos e uma detalhada, com todos os valores das iterações, num arquivo de saída (por isso há a repetição de alguns títulos e resultados no código).

Os resultados obtidos estão descritos abaixo:

Figura 3: Saída da tarefa C.

```
andressacolaco@ametista10:/public/fiscomp2022-2-alcaraz/proj3/proj3_12610389/taref
a-c$ .
/tarefa-c-12610389.exe
 Busca direta
   Raiz encontrada:       -1.00000019073       Iterações: 19
   Raiz encontrada:        0.49999961853       Iterações: 18
   Raiz encontrada:        1.99999980927       Iterações: 19
 Método de Newton-Raphson
   Raiz encontrada:       -1.00000000027       Iterações:  9
   Raiz encontrada:        0.50000006938       Iterações:  3
   Raiz encontrada:        2.00000005047       Iterações:  6
 Método da Secante
   Raiz encontrada:       -1.00000002960       Iterações: 12
   Raiz encontrada:        0.49999999260       Iterações:  1
   Raiz encontrada:        1.99999999997       Iterações:  7
```

Para a busca direta foram necessárias 18 ou 19 iterações, sem contar os intervalos percorridos que não continham uma raiz:

Tabela 5: Valor aproximado a cada iteração para a busca direta.

	r1	r2	r3
1	-1.05000000000	0.45000000000	1.95000000000
2	-1.02500000000	0.47500000000	1.97500000000
3	-1.01250000000	0.48750000000	1.98750000000
4	-1.00625000000	0.49375000000	1.99375000000
5	-1.00312500000	0.49687500000	1.99687500000
6	-1.00156250000	0.49843750000	1.99843750000
7	-1.00078125000	0.49921875000	1.99921875000
8	-1.00039062500	0.49960937500	1.99960937500
9	-1.00019531250	0.49980468750	1.99980468750
10	-1.00009765625	0.49990234375	1.99990234375
11	-1.00004882812	0.49995117188	1.99995117188
12	-1.00002441406	0.49997558594	1.99997558594
13	-1.00001220703	0.49998779297	1.99998779297
14	-1.00000610352	0.49999389648	1.99999389648
15	-1.00000305176	0.49999694824	1.99999694824
16	-1.00000152588	0.49999847412	1.99999847412
17	-1.00000076294	0.49999923706	1.99999923706
18	-1.00000038147	0.49999961853	1.99999961853
19	-1.00000019073	-	1.99999980927
Exatas	-1.00000000000	0.500000000000	2.00000000000

Para o método de Newton, encontramos as raízes em bem menos iterações:

Tabela 6: Valor aproximado a cada iteração para ao método de Newton-Raphson.

	r1	r2	r3
1	-6.54794520548	0.66666666703	3.16666574165
2	-4.27066035406	0.49572649570	2.48738423785
3	-2.78881901191	0.50000006938	2.13548110487
4	-1.85590340089	-	2.01516550660
5	-1.31599451432	-	2.00022469317
6	-1.06704278833	-	2.00000005047
7	-1.00407065093	-	-
8	-1.00001646594	-	-
9	-1.00000000027	-	-
Exatas	-1.00000000000	0.500000000000	2.00000000000

Neste caso, temos que h_n já foi ajustado para a função, logo não foram feitas mais iterações. Também é notável que a primeira raiz leva bem mais operações que as outras para ser encontrada e isso se deve à distância dos pontos iniciais de procura: os das duas últimas são logo após uma raiz, em um incremento relativamente grande, logo devem levar menos passos para serem encontradas.

Vemos que o método de Newton encontra bem mais rapidamente que a busca direta uma raiz, mas esta última é mais simples e geral e não depende do conhecimento da derivada da função, além de não encontrar problemas como o método de Newton de poder cair em um loop infinito de operações. Na prática, os dois métodos são bem usados em conjunto: reduz-se o intervalo de procura com Newton e então faz-se uma busca direta em uma região bem restrita. Desta forma, contornamos os problemas citados para ambos.

Para o método da secante, temos:

Tabela 7: Valor aproximado a cada iteração para o método da secante.

	r1	r2	r3
1	-6.19463087248	0.49999999260	1.76315789219
2	-4.91734424484	-	1.90393914353
3	-3.56040857902	-	2.03068167960
4	-2.67894053787	-	1.99689766262
5	-2.00853968327	-	1.99990681930
6	-1.55012612455	-	2.00000028980
7	-1.25107110155	-	1.99999999997
8	-1.08504934682	-	-
9	-1.01691759082	-	-
10	-1.00133287078	-	-
11	-1.00002223311	-	-
12	-1.00000002960	-	-
Exatas	-1.00000000000	0.500000000000	2.00000000000

É notável que ele realiza alguns passos a mais que Newton, porém, considerando que este método toma um x a mais para fazer aproximações da secante, vemos que ele é bem sensível às mudanças do segundo ponto tomado. Aqui, usamos o ponto inicial mais o incremento h_n por generalidade, mas trocar este pode fazer a raiz ser encontrada em uma ou duas iterações a menos, conforme testado para alguns valores em $[-10, -2]$.

Os desvios em relação ao valor exato estão contidos na seguinte tabela:

Tabela 8: Erros obtidos para as raízes em cada método.

	Busca direta	Newton-Raphson	Secante
r1	1.9073e-07	2.7e-09	2.960e-08
r2	3.8147e-07	6.938e-08	7.40e-09
r3	1.9073e-07	5.047e-08	3.0e-11

É notável através desta tabela que o método da secante possui a melhor precisão entre os métodos para duas raízes, enquanto a busca direta possui o maior erro para todas as raízes. Isto pode ser explicado por conta do próprio método, que apenas procura por um um intervalo cuja altura das funções esteja em uma tolerância desejada. Os métodos de Newton e da secante são um pouco mais refinados, utilizando a derivada ou secante para cortar intervalo. Apesar disso, por conta de suas limitações, usar uma combinação de busca direta com Newton ou Secante pode ser interessante em diversos casos.