# Relatório de Cálculo Numérico: Trabalho 1

Alison de Oliveira Tristão
Vitor João de Andrade

14 de outubro de 2024

## 1 Introdução

O cálculo numérico desempenha um papel fundamental na resolução de problemas matemáticos que não podem ser resolvidos analiticamente de maneira exata. Este trabalho visa explorar métodos numéricos para a solução de equações não lineares e sistemas de equações lineares, apresentando uma análise detalhada de cada método. apresentando soluções por meio de diferentes métodos como os métodos de Newton, falsa posição, secante, bisseção e ponto fixo também presentamos soluções com a utilização de jacobianas e para equações lineares com o método de Gauss

## 2 Problema 1: Equação Não Linear

A função sinc(t) em sua versão padrão é representada como:

$$\text{sinc}(t) = \frac{\sin(t)}{t} \tag{1}$$

Assim, as raízes da função estão localizadas em $k\pi$, onde $k \in \mathbb{Z}$ e $k \neq 0$. Sabendo disso, vamos aplicar cinco métodos numéricos para aproximar as duas raízes presentes no intervalo $[0.1, 8.0]$, que são $\pi$ e $2\pi$.

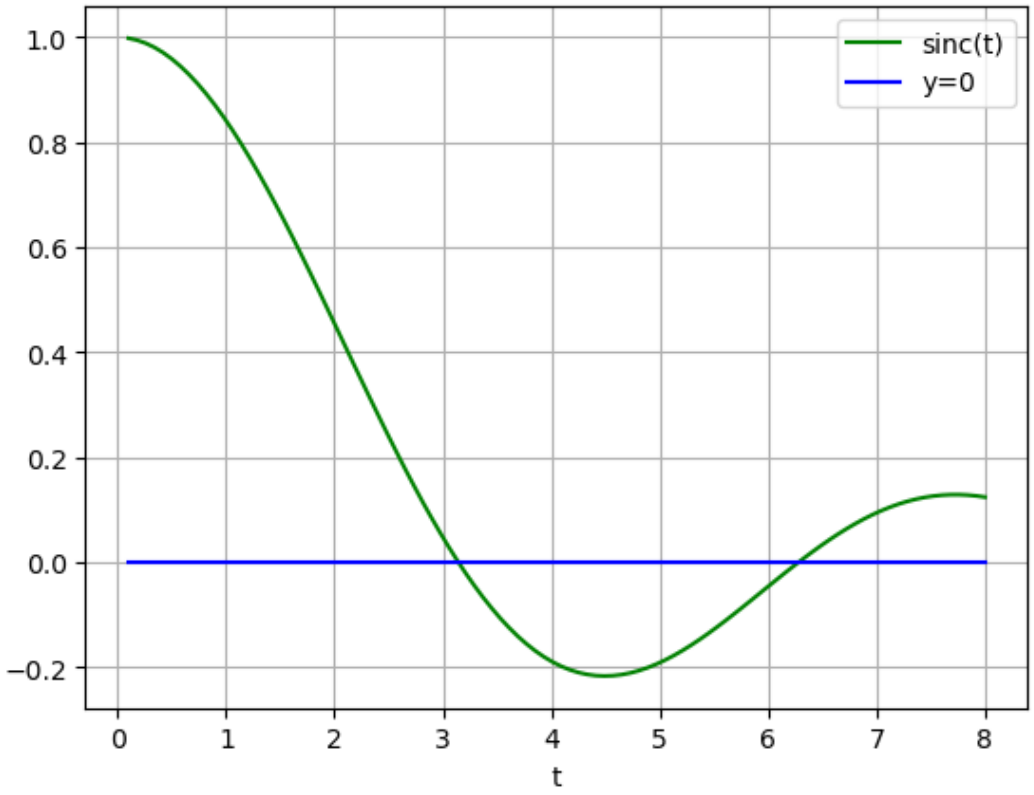


Figura 1: sinc(t)

Os métodos usados são os seguintes: método da bisseção, falsa posição, newton, secante e ponto fixo. Suas aplicaçôes e resultados estão a seguir.

> ***Nota:*** *O método do ponto fixo não foi mencionado no exercício, mas decidimos incluí-lo por conta do estudo realizado.*

## 2.1 Método da Bissecção

Para aproximar $x$, onde $f(x) = 0$, utilizando o método da bissecção, precisamos definir um intervalo $[a, b]$ que contenha uma raiz. Esse intervalo deve satisfazer o Teorema de Bolzano, que estabelec que $f(a)$ e $f(b)$ devem ter sinais opostos, ou seja:

$$f(a) \cdot f(b) < 0$$

Caso $f(a) \cdot f(b) = 0$, então $a$ ou $b$ é uma raiz da função.

Obedecendo a essas condições, o método consiste em verificar se o ponto médio do intervalo se aproxima da raiz e, dependendo do sinal resultante, trocamos o um dos limites do intervalo pelo ponto médio e repetimos até encontrarmos um $f(ponto\_medio) = \alpha$ tal que $\alpha$ seja suficientemente pequeno.

## 2.2 Algoritmo da Bissecção

O algoritmo implementado para realizar a bissecção verifica o Teorema de Bolzano, checa se a raiz da função foi passada como parâmetro e, por fim, realiza iterações

salvando os pontos intermediários.

```
import sys

def bisection(init, end, function):
    a = init
    b = end

    # precisao de maquina
    eps = sys.float_info.epsilon

    # se colocar invertido o algoritmo buga
    if(a > b):
        a, b = b, a

    # verifica se o intervalo passa no teorema de Bolzano
    if(function(a) * function(b) > 0):
        print("O intervalo nao passa no teorema de Bolzano.")
        return None

    # verifica se o intervalo ja e a raiz
    if(abs(function(a)) < eps):
        print("Raiz em x:", a)
        return None
    if(abs(function(b)) < eps):
        print("Raiz em x:", b)
        return None

    # salva o valor final da raiz
    mid_point = 0

    # zera os arrays
    values_biss = []
    error_biss = []

    # iteracoes
    for i in range(max_iter):
        # calculo do ponto medio
        mid_point = (a + b)/2
        values_biss.append(mid_point)

        # verificacao de parada
        error_biss.append(abs(function(mid_point)))
        if error_biss[i] < error:
            break

        # calculamos ponto para a proxima iteracao
        if function(mid_point) < 0:
            b = mid_point
        else:
```

```
            a = mid_point

    return mid_point, values_biss, error_biss
```

Listing 1: Python - Método da Bissecção

## 2.3 Resultados da Bissecção

Utilizando o intervalo [3.0, 3.5] para aproximar $\pi$, obtemos os seguintes resultados:

- Raiz aproximada: 3.141593933105469
- Iterações necessárias: 16
- Erro absoluto final: $1.28 \times 10^{-6}$

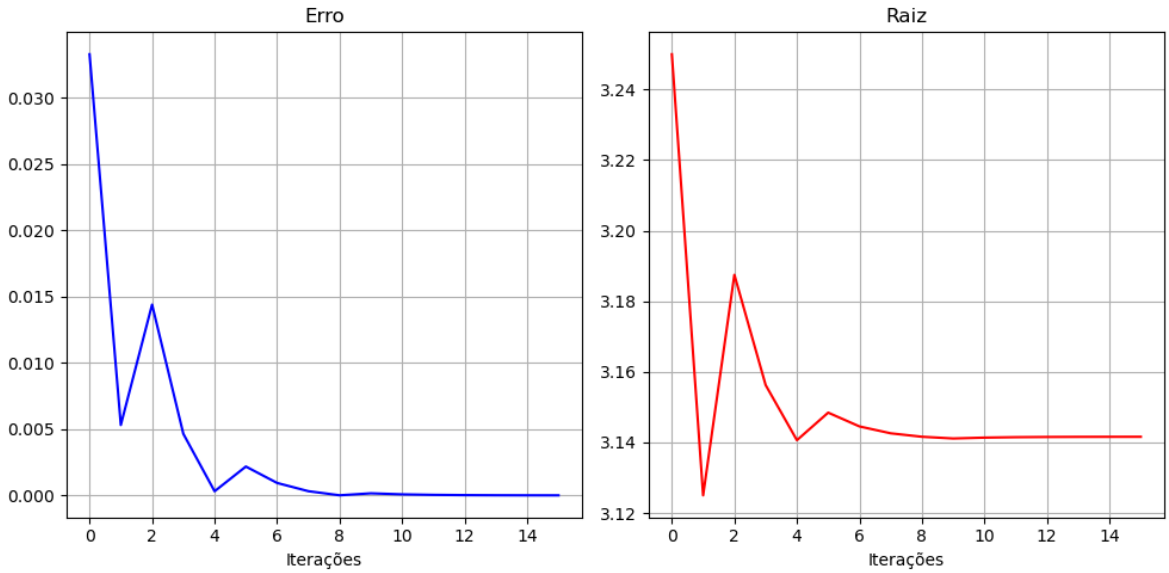


Figura 2: Resultados - Bissecção

## 2.4 Método da Falsa Posição

Semelhante ao método da bisseção, a falsa posição precisa de um intervalo $[a, b]$ que contenha uma raiz e apresente uma troca de sinal. No entanto, aproxima o valor da raiz utilizando a interseção da reta que passa por $(a, f(a))$ e $(b, f(b))$. Desse modo, iteramos $x$ até encontrar a raiz com a precisão desejada.

$$x^{x+k} = x^k - \frac{b-a}{f(a) - f(b)} \cdot f(x) \tag{2}$$

## 2.5 Algoritmo da Falsa Posição

Verficamos as condições necessarias para no intervalo, se a raiz está como parametro e então iteramos até a condição de parada salvando os valores intermediarios.

```
import sys

def false_position(init, end, function):
    a = init
    b = end

    # precisao de maquina
    eps = sys.float_info.epsilon

    # se colocar invertido o algoritmo buga
    if(a > b):
        a, b = b, a

    # verifica se o intervalo passa no teorema de Bolzano
    if(function(a) * function(b) > 0):
        print("O intervalo nao passa no teorema de Bolzano.")
        return None

    # verifica se o intervalo ja e a raiz
    if(abs(function(a)) < eps):
        print("Raiz em x:", a)
        return None
    if(abs(function(b)) < eps):
        print("Raiz em x:", b)
        return None

    x1 = a
    x2 = b
    f1 = function(x1)
    f2 = function(x2)
    xa = a
    xs = b

    # zera os arrays
    values_false = [x1]
    error_false = [f1]

    # itera
    for i in range(max_iter):
        # calculo da solucao
        xa = xs
        xs  = x1 - (f1*(x2 - x1)/(f2 - f1))
        fs = function(xs)
        values_false.append(xs)

        # verificacao de parada
```

```
        error_false.append(abs(fs))
        if abs(error_false[i + 1]) < error or abs(xs - xa) <
            converge:
            break

        # atualiza os valores
        if fs*f1 < 0:
            x2 = xs
            f2 = fs
        else:
            x1 = xs
            f1 = fs

    return xs, values_false, error_false
```

Listing 2: Python - Método do Ponto Fixo

## 2.6 Resultados da Falsa Posição

Aplicando o algoritmo no intervalo [6.0, 6.5], aproximamos $2\pi$ como:

- Raiz aproximada: 6.283185614449988
- Iterações necessárias: 5
- Erro absoluto final: $3.07 \times 10^{-7}$

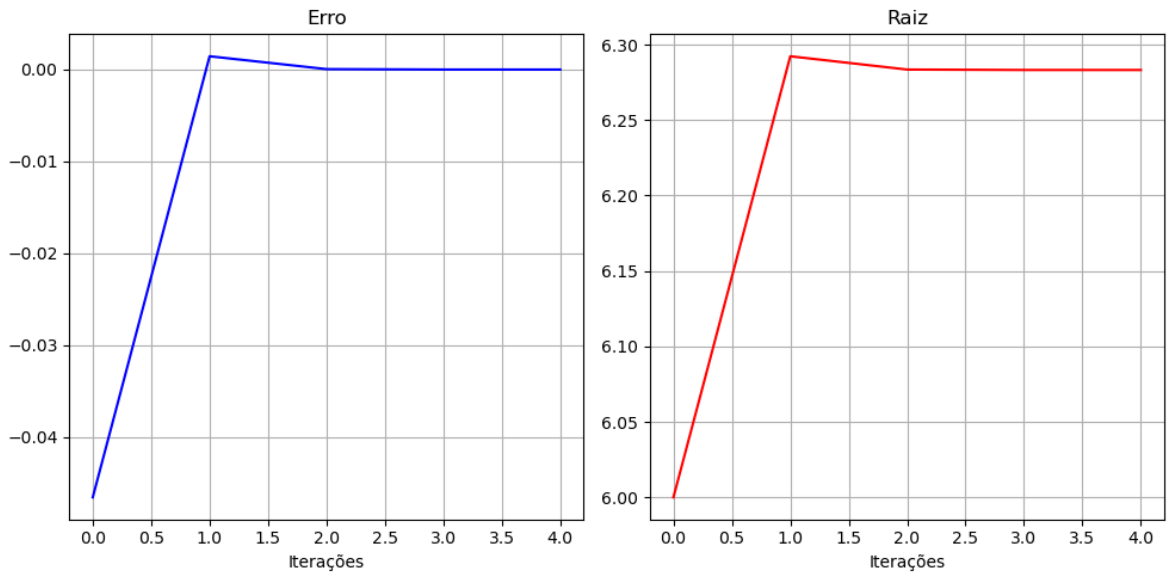


Figura 3: Resultados - Falsa Posição

## 2.7 Método de Newton

No método de newton, utilizamos expensão de taylor de primeira ordem em torno do iterado, aproximando o proximo valor utilizando a reta tangente no ponto.

Sendo:

$$f(x^{k+1}) = f(x^k) + \frac{df(x^k)}{dx}(x^{k+1} - x^k) + O\left(\|x^{k+1} - x^k\|^2\right) \tag{3}$$

Para $x^{k+1} - x^k$ pequeno, fazemos:

$$f(x^{k+1}) \approx f(x^k) + \frac{df(x^k)}{dx}(x^{k+1} - x^k) \tag{4}$$

O que nos leva a relação:

$$x^{k+1} = x^k - \frac{f(x)}{f'(x)} \tag{5}$$

Considerando a função $\text{sinc}(x) = \frac{\sin(x)}{x}$, utilizamos sua derivada dada por:

$$\text{sinc}'(x) = \frac{x\cos(x) - \sin(x)}{x^2} \tag{6}$$

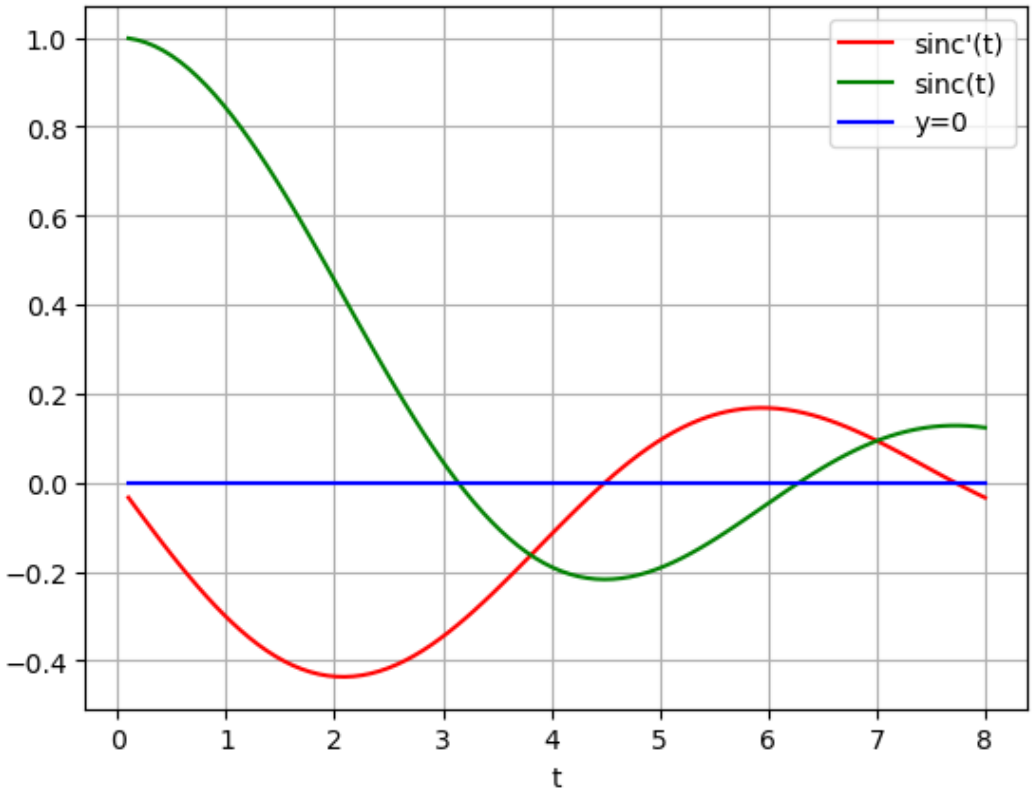


Figura 4: sinc'(t)

E então temos a seguinte formula para iterar:

$$x^{k+1} = x^k - \frac{\frac{sin(x)}{x}}{\frac{xcos(x)-sin(x)}{x^2}} \tag{7}$$

## 2.8 Algoritmo de Newton

```
def newton(solution, function, funtion_linha):
    # zera os arays
    values_new = [solution]
    error_new  = [abs(function(solution))]

    for i in range(max_iter):
        # calculo da solucao
        current_solution = solution - (function(solution)/
           funtion_linha(solution))
        values_new.append(current_solution)

        # atualiza as solucaoes
        solution = current_solution

        # verificacao de parada
        error_new.append(abs(function(current_solution)))
        if(error_new[i+1] < error):
            break
    return solution, values_new, error_new
```

Listing 3: Python - Método de Newton

## 2.9 Resultados de Newton

Com o chute inicial de 3.4, sendo que deveria chegar em $\pi$, o algoritmo atingiu os seguintes resultados:

- Raiz aproximada: 3.141592634392667
- Iterações necessárias: 4
- Erro absoluto final: $1.92 \times 10^{-8}$

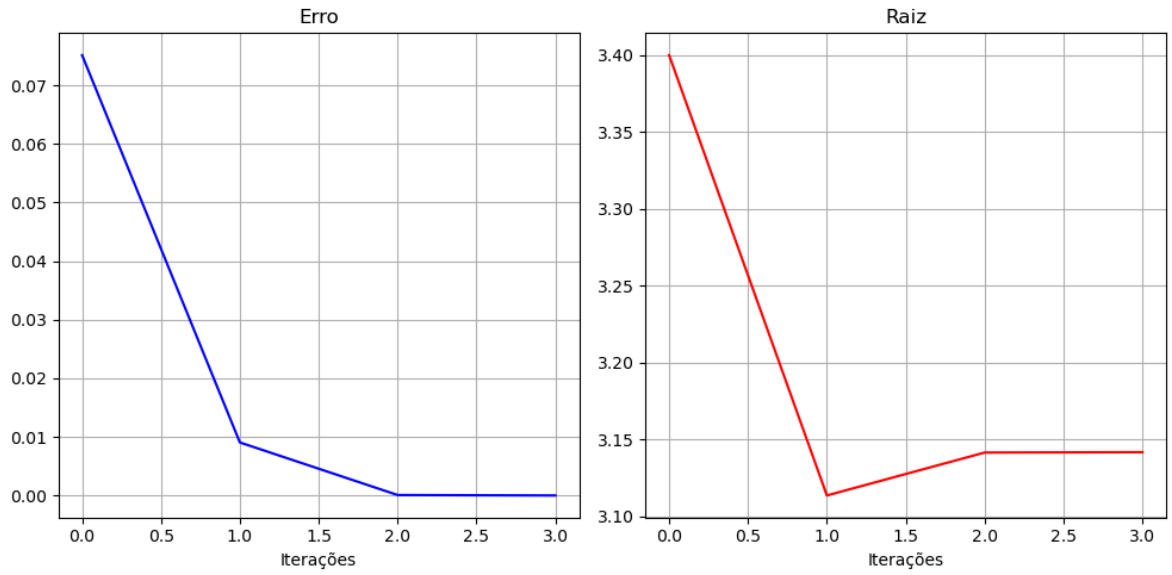


Figura 5: Resultados - Newton

## 2.10 Método da Secante

Em geral, quando encontrar a derivada da função para aplicar o método de Newton é muito complicado, podemos usar uma aproximação linear da derivada utilizando os últimos pontos calculados: $\frac{(x^k - x^{k-1})}{f(x^k) - f(x^{k-1})}$.

Exemplificando:

$$x^{k+1} = x^k - \frac{(x^k - x^{k-1})}{f(x^k) - f(x^{k-1})} \cdot f(x^k) \tag{8}$$

Esta fórmula é semelhante ao método da falsa posição, contudo se diferencia pelo fato de que não precisa ser aplicada em um intervalo com troca de sinal.

Ao aplicar esse método, computacionalmente, podem ocorrer problemas de overflow. Para evitar essas complicações, aplicamos da seguinte forma:

$$x_{k+1} = x_k - \left(\frac{(x_{k-1} - x_k) f(x_k)}{f(x_{k-1})}\right) \Bigg/ \left(1 - \frac{f(x_k)}{f(x_{k-1})}\right) \tag{9}$$

E, por fim, iteramos até um valor suficientemente satisfatório.

## 2.11 Algoritmo da Secante

```
def secant(x0, x1, sinc):
    # valores iniciais
    f0 = sinc(x0)
    f1 = sinc(x1)

    # troca os valores caso a funcao nao seja decrescente
    if(abs(f1) < abs(f0)):
        x0, x1 = x1, x0
```

```
        f0, f1 = f1, f0

    # zera os arrays
    values_sec = [x0]
    error_sec  = [f0]

    for i in range(max_iter):
        # salva o valor anterior
        values_sec.append(x1)

        # calcula o proximo valor utilizando o metodo que tenta
           evitar overflow
        s = f1/f0
        p = (x0-x1)*s
        q = 1 - s
        x2 = x1 - p/q

        # verificacao de parada
        error_sec.append(abs(f1))
        if(abs(x1 - x2) < converge or error_sec[i+1] < error):
            x1 = x2
            break

        # atualiza os valores
        f2 = sinc(x2)

        # inverte os valores para a proxima iteracao
        if(abs(f2) > abs(f1)):
            x0, f0 = x2, f2
        else:
            x0, f0 = x1, f1
            x1, f1 = x2, f2

    return x1, values_sec, error_sec
```

Listing 4: Python - Método da Secante

## 2.12 Resultados da Secante

Inciando o algoritmo com os valores 6.0 e 6.1, os resultados alcaçados ao aproximar $2\pi$ foram:

- Raiz aproximada: 6.283185307177334
- Iterações necessárias: 5
- Erro absoluto final: $2.25 \times 10^{-12}$

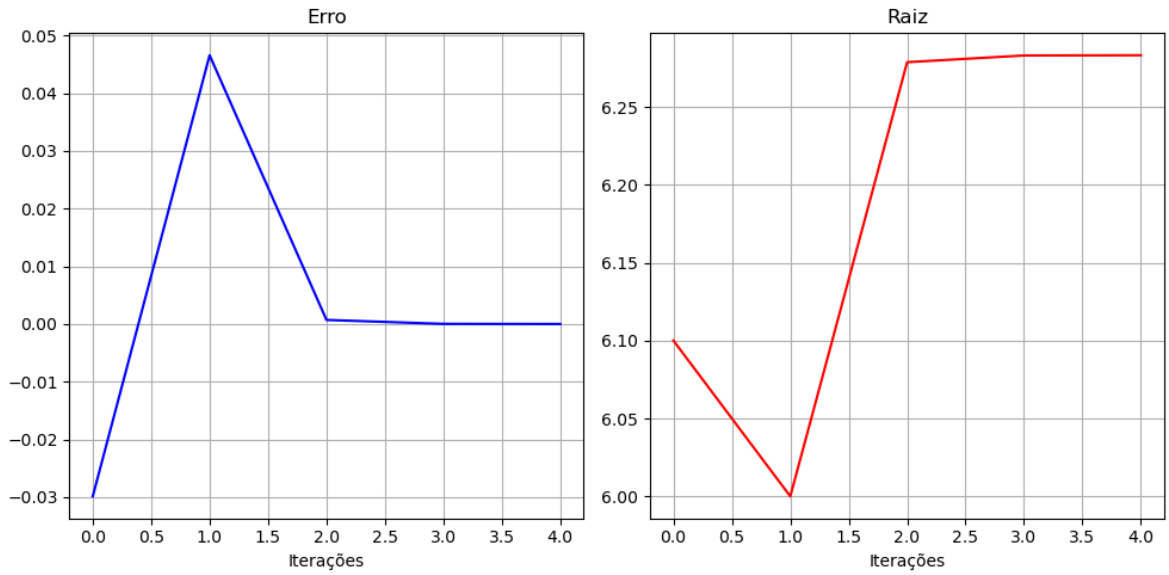


Figura 6: Resultados - Secante

## 2.13 Método do Ponto Fixo

No método do ponto fixo, precisamos fazer uma substituição $g(x) = x + c(x) \cdot f(x)$, onde $c(x)$ deve ser escolhida de forma que $c(x) \neq 0$ para todo o intervalo [a, b]. Então, escrevemos nossa $g(x)$ como:

$$\mathrm{g}(x) = x - \frac{\sin(x)}{x} \tag{10}$$

Onde $c(x) = -1$.

Após definir $g(x)$, procuramos um ponto fixo $x^*$ tal que $x^* = g(x^*)$, definindo $x^{k+1} = g(x^k)$ e iteramos até encontrar um valor pequeno o bastante.

Note que:

$$x^* = g(x^*) \iff f(x^*) = 0$$

Também é necessário mostrar que, para garantir a convergência das iterações pelo teorema do ponto fixo, é preciso que $\forall x \in I$, $|g'(x)| \leq L < 1$.

Considerando isso, nossa função $g'$ é menor que $L$, sendo $L = 0.855$, para todo o intervalor [6.0, 6.5].

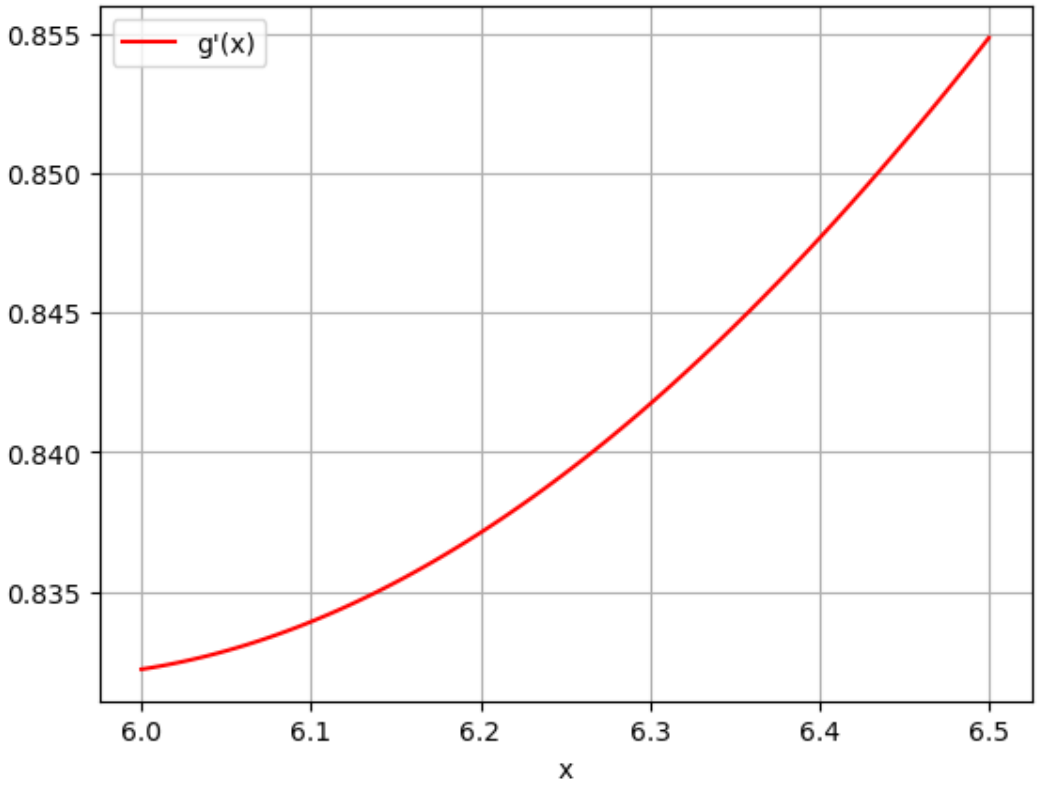


Figura 7: g'(x)

## 2.14 Algoritmo do Ponto Fixo

O algoritmo desenvolvido consiste em realizar as interações explicadas salvando os valores intermediarios para plotagem.

```
def fixed_point(solution, funtion, funtion_g):
    # zera os arrays
    values_fix = []
    error_fix = []

    for i in range(max_iter):
        # calculo da solucao
        current_solution = funtion_g(solution)
        values_fix.append(current_solution)

        # verificacao de parada
        error_fix.append(abs(funtion(current_solution)))
        if(error_fix[i] < error or abs(current_solution -
           solution) < converge):
            break

        # atualiza as solucaoes
        solution = current_solution

    return solution, values_fix, error_fix
```

Listing 5: Python - Método do Ponto Fixo

## 2.15 Resultados do Ponto Fixo

Utilizando o intervalo [6.0, 6.5] para aproximar $2\pi$, obtemos os seguintes resultados:

- Raiz aproximada: 6.283192210886315
- Iterações necessárias: 61
- Erro absoluto final: $6.90 \times 10^{-6}$

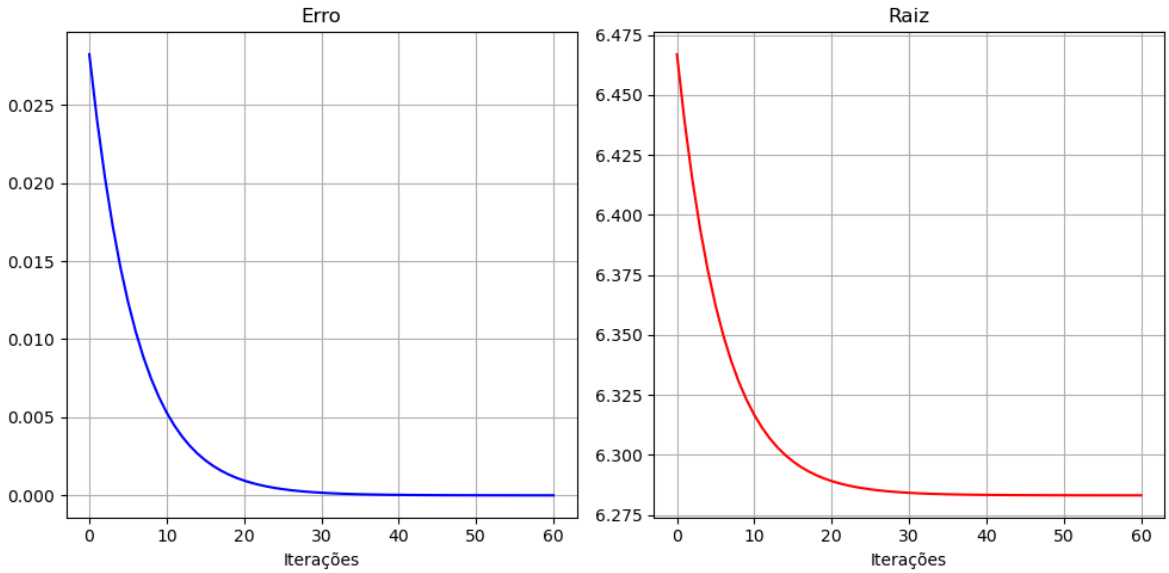


Figura 8: Resultados - Ponto Fixo

# 3 Problema 2: Equação de Calor

Uma EDP bastante estudada é a equação do calor:

$$k\frac{\partial^2 T(x)}{\partial x^2} = q(x) \tag{11}$$

Esta equação descreve os valores de temperatura $T$ ao longo de um eixo unidimensional onde $x \in (0, 1)$, considerando o fluxo de calor exógeno $q(x)$. Para resolver a equação, começando definindo as condições de contorno:

$$\begin{cases} T(0) = T_{\text{begin}} \\ T(1) = T_{\text{end}} \end{cases} \tag{12}$$

Não podemos garantir que uma EDP, linear ou não, tenha uma solução analítica. A equação do calor apresentada pode ter solução, mas resolvemos o problema de forma numérica, reformulando-a para uma maneira computacionalmente viável.

Para isso, aplicando o método das Diferenças Finitas, a EDP pode ser representada por um sistema de equações lineares, aproximado pela seguinte equação discreta:

$$\left(\frac{k}{h^2}\right) D^2T = Q - \left(\frac{k}{h^2}\right) BC \tag{13}$$

Onde $D^2$, $T$, $Q$ e $BC$ são matrizes e vetores de dimensões adequadas. Já o parâmetro $h$ é o espaçamento dos pontos de interesse, sendo os mesmos definidos da seguinte forma:

$$x_{i+1} = x_i + h \tag{14}$$

Assim, podemos definir nossa matriz de temperaturas como:

$$T = \begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_N \end{pmatrix} = \begin{pmatrix} T(x_1) \\ T(x_2) \\ \vdots \\ T(x_N) \end{pmatrix} \tag{15}$$

Onde $N \in \mathbb{N}$:

$$N = \frac{1}{h} - 1 \tag{16}$$

Logo, a distribuição de temperaturas pode ser vista de uma maneira discreta ao longo de X:

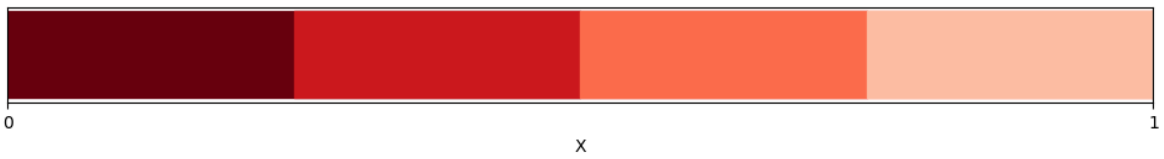


Figura 9: Discretização da Temperatura no Eixo x

A matriz $D^2 \in \mathbb{R}^{N \times N}$ é construída através da discretização da derivada parcial no espaço. A matriz $BC \in \mathbb{R}^N$ resulta das condições de contorno, enquanto $Q \in \mathbb{R}^N$ está associada ao termo $q(x_i)$. As representações dessas matrizes são dadas a seguir:

$$D^2 = \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \tag{17}$$

$$BC = \begin{pmatrix} T_{\text{begin}} \\ 0 \\ \vdots \\ T_{\text{end}} \end{pmatrix} \tag{18}$$

$$Q = \begin{pmatrix} q(x_1) \\ q(x_2) \\ \vdots \\ q(x_N) \end{pmatrix} \tag{19}$$

Generalizamos a matriz $D^2$ da seguinte forma: para a linha $i$ e a coluna $j$, a estrutura é dada por:

$$D^2(i,j) = \begin{cases} -2 & \text{se } i = j \\ 1 & \text{se } i = j+1 \text{ ou } i = j-1 \\ 0 & \text{caso contrário} \end{cases} \tag{20}$$

Sabendo disso, iremos resolver três situações de maneira numérica utilizando o método de Gauss com pivoteamento. As situações são as seguintes:

- $k = 2.5$
- $T_{\text{begin}} = 60$
- $T_{\text{end}} = 40$
- $q(x) = \sin(x) + 1$

Considerando:

$$h = \begin{cases} 0.2 & \text{caso (a)} \\ 0.1 & \text{caso (b)} \\ 0.05 & \text{caso (c)} \end{cases} \tag{21}$$

Para resolver esse sistema de equações, consideramos o sistema na forma $Ax = b$, onde $A = \left(\frac{k}{h^2}\right) D^2$ e $b = Q - \left(\frac{k}{h^2}\right) BC$.

Em seguida, definimos nossa matriz expandida como $Ab$ e a escalonamos utilizando o método de Gauss com pivoteamento, garantindo assim uma melhor estabilidade numérica.

Utilizando Python, o algoritmo para solução desenvolvido foi:

```
def pivoteamento(matriz, b, k):
    # obtem o index do maior valor abaixo da diagonal principal
    max_index = np.argmax(np.abs(matriz[k:, k])) + k

    # inverte as linhas para que o maior valor fique na diagonal
       principal
    if max_index != k:
        matriz[[k, max_index]] = matriz[[max_index, k]]
        b[[k, max_index]] = b[[max_index, k]]

def escalonamento(matriz, b):
    n = len(b)
    matriz = matriz.astype(float)
```

```
    b = b.astype(float)

    for k in range(n):
        # coloca os maiores valores abaixo da diagonal principal
        pivoteamento(matriz, b, k)

        # escola a coluna
        for i in range(k + 1, n):
            fator = matriz[i, k] / matriz[k, k]
            matriz[i, k:] -= fator * matriz[k, k:]
            b[i] -= fator * b[k]

    return matriz, b

def solucao(matriz, b):
    n = len(b)
    x = np.zeros(n)

    # substitui os valores para achar a solucao
    for i in range(n - 1, -1, -1):
        x[i] = (b[i].item() - np.dot(matriz[i, i + 1:], x[i +
            1:])) / matriz[i, i].item()

    return x

def gauss(matriz, b):
    # escalona a matriz extendida
    matriz, b = escalonamento(matriz, b)

    # retorna a solucao
    return solucao(matriz, b)
```

Listing 6: Python - Gauss com Pivoteameento

## 3.1 Situação 1: h = 0.2

- $N = 4$
- $AT = b$ :

$$A = \begin{bmatrix} -125.00 & 62.50 & 0.00 & 0.00 \\ 62.50 & -125.00 & 62.50 & 0.00 \\ 0.00 & 62.50 & -125.00 & 62.50 \\ 0.00 & 0.00 & 62.50 & -125.00 \end{bmatrix}$$

$$b = \begin{bmatrix} -3748.80 \\ 1.39 \\ 1.56 \\ -2498.28 \end{bmatrix}$$

- Os valores de T ao longo de X:

Tabela 1: T (°C)

| | |
|---|---|
| $T(x_1)$ | 60.00 |
| $T(x_2)$ | 55.96 |
| $T(x_3)$ | 51.93 |
| $T(x_4)$ | 47.93 |
| $T(x_5)$ | 43.95 |
| $T(x_6)$ | 40.00 |


Figura 10: Temperatura no Eixo X: h = 0.2

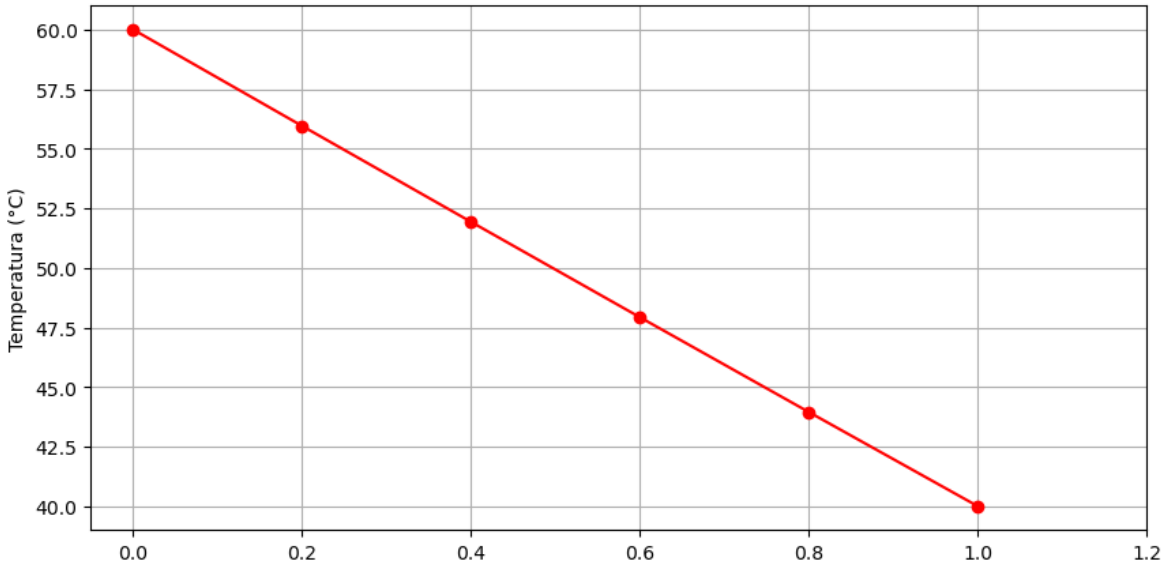


Figura 11: Temperatura no Eixo X: h = 0.2

# 4 Situação 2: h = 0.1

- $N = 9$
- $AT = b$ :

$$A = \begin{bmatrix} -500.00 & 250.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 250.00 & -500.00 & 250.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 250.00 & -500.00 & 250.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 250.00 & -500.00 & 250.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 250.00 & -500.00 & 250.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 250.00 & -500.00 & 250.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 250.00 & -500.00 & 250.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 250.00 & -500.00 & 250.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 250.00 & -500.00 \end{bmatrix}$$

$$b = \begin{bmatrix} -14998.90 \\ 1.20 \\ 1.30 \\ 1.39 \\ 1.48 \\ 1.56 \\ 1.64 \\ 1.72 \\ -9998.22 \end{bmatrix}$$

- Os valores de T ao longo de X:

Tabela 2: T (°C)

| | |
|---|---|
| T($x_1$) | 60.00 |
| T($x_2$) | 57.98 |
| T($x_3$) | 55.96 |
| T($x_4$) | 53.94 |
| T($x_5$) | 51.93 |
| T($x_6$) | 49.93 |
| T($x_7$) | 47.93 |
| T($x_8$) | 45.94 |
| T($x_9$) | 43.95 |
| T($x_10$) | 41.97 |
| T($x_11$) | 40.00 |

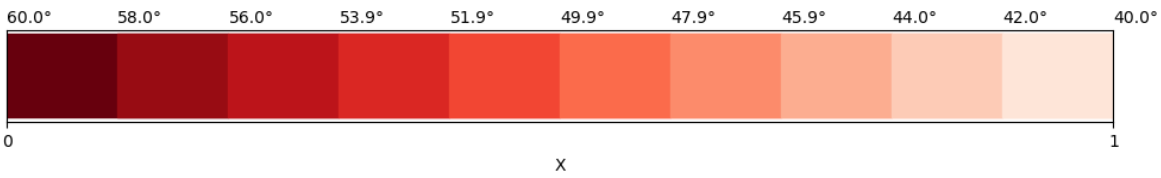


Figura 12: Temperatura no Eixo X: h = 0.1

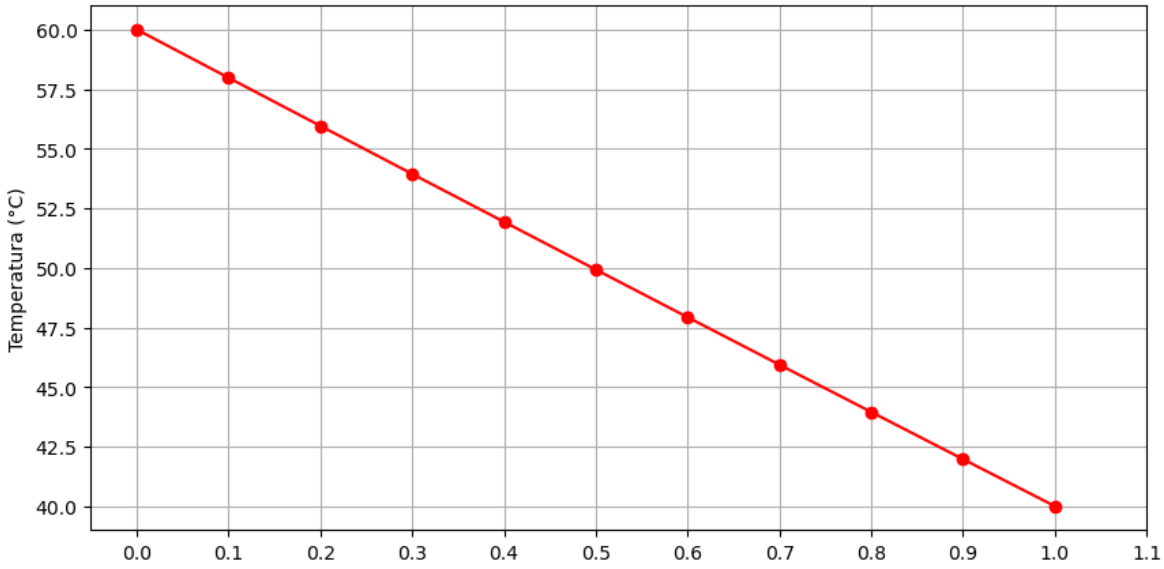


Figura 13: Temperatura no Eixo X: h = 0.1

# 5 Situação 2: h = 0.05

- $N = 19$
- $AT = b$ :

$$A = \begin{bmatrix} -2000 & 1000 & 0 & \cdots & 0 & 0 & 0 \\ 1000 & -2000 & 1000 & \cdots & 0 & 0 & 0 \\ 0 & 1000 & -2000 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -2000 & 1000 & 0 \\ 0 & 0 & 0 & \cdots & 1000 & -2000 & 1000 \\ 0 & 0 & 0 & \cdots & 0 & 1000 & -2000 \end{bmatrix}$$

$$b = \begin{bmatrix} -59998.95 \\ 1.10 \\ 1.15 \\ 1.20 \\ 1.25 \\ 1.30 \\ 1.34 \\ 1.39 \\ 1.43 \\ 1.48 \\ 1.52 \\ 1.56 \\ 1.61 \\ 1.64 \\ 1.68 \\ 1.72 \\ 1.75 \\ 1.78 \\ -39998.19 \end{bmatrix}$$

- Os valores de T ao longo de X:

Tabela 3: T (°C)

| | |
|---|---|
| T($x_1$) | 60.00 |
| T($x_2$) | 58.99 |
| T($x_3$) | 57.98 |
| T($x_4$) | 56.97 |
| T($x_5$) | 55.96 |
| T($x_6$) | 54.95 |
| T($x_7$) | 53.94 |
| T($x_8$) | 52.94 |
| T($x_9$) | 51.93 |
| T($x_10$) | 50.93 |
| T($x_11$) | 49.93 |
| T($x_12$) | 48.93 |
| T($x_13$) | 47.93 |
| T($x_14$) | 46.93 |
| T($x_15$) | 45.94 |
| T($x_16$) | 44.94 |
| T($x_17$) | 43.95 |
| T($x_18$) | 42.96 |
| T($x_19$) | 41.97 |
| T($x_20$) | 40.98 |
| T($x_21$) | 40.00 |

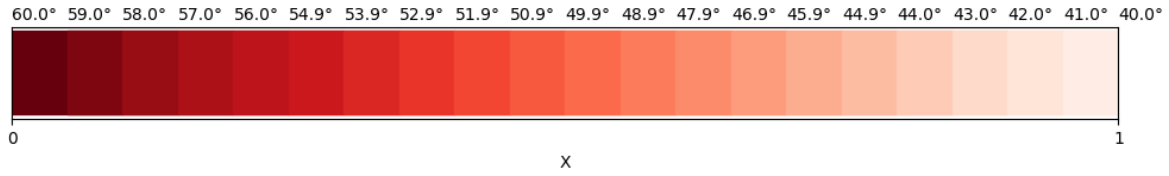


Figura 14: Temperatura no Eixo X: h = 0.05

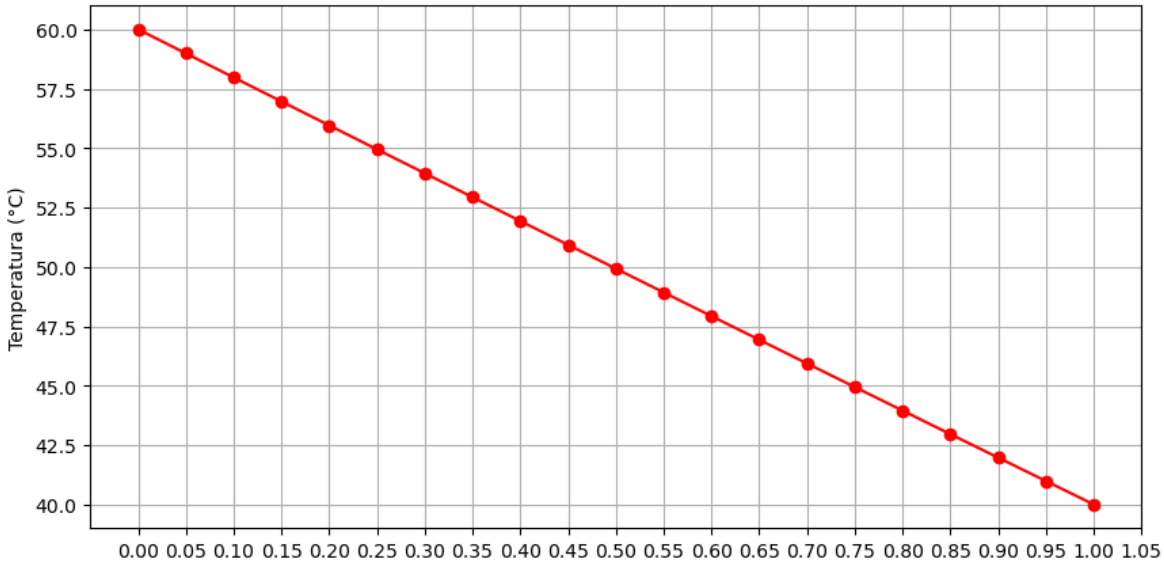


Figura 15: Temperatura no Eixo X: h = 0.05

# 6 Problema 3: Sistemas Não-lineares

As equações de equilíbrio para uma aeronave específica consistem em um sistema de 5 equações e 8 variáveis com a forma:

$$F(x) = Ax + g(x) = 0$$

onde $F : \mathbb{R}^5 \to \mathbb{R}^5$, e a matriz $A$ é dada por:

$$A = \begin{bmatrix} -3.933 & 0.107 & 0.126 & 0 & -9.99 \\ 0 & -0.987 & 0 & -22.95 & 0 \\ 0.002 & 0 & -0.235 & 0 & 5.67 \\ 0 & 1.0 & 0 & -1.0 & 0 \\ 0 & 0 & -1.0 & 0 & -0.196 \end{bmatrix}$$

A parcela não-linear é definida como:

$$g(x) = \begin{bmatrix} -0.727x_2x_3 + 8.39x_3x_4 - 684.4x_4x_5 + 63.5x_4x_2 + 0.2 \\ 0.949x_1x_3 + 0.173x_1x_5 + 0.35 \\ -0.716x_1x_2 - 1.578x_1x_4 + 1.132x_4x_2 + 0.6 \\ -x_1x_5 + 0.7 \\ x_1x_4 + 1 \end{bmatrix}$$

**6.1 Utilizando $x^{(0)} = \begin{bmatrix} 0.2 \\ 0.5 \\ 0.85 \\ 0.93 \\ 0.22 \end{bmatrix}$ como condição inicial, o código em Python para resolver este sistema pelo método de Newton-Raphson é:**

```python
import numpy as np

# Definindo a matriz A
A = np.array([
    [-3.933, 0.107, 0.126, 0, -9.99],
    [0, -0.987, 0, -22.95, 0],
    [0.002, 0, -0.235, 0, 5.67],
    [0, 1.0, 0, -1.0, 0],
    [0, 0, -1.0, 0, -0.196]
])
# Definindo a funcao g(x)
def g(x):
    x1, x2, x3, x4, x5 = x
    return np.array([
        -0.727*x2*x3 + 8.39*x3*x4 - 684.4*x4*x5 + 63.5*x4*x2 +
           0.2,
        0.949*x1*x3 + 0.173*x1*x5 + 0.35,
        -0.716*x1*x2 - 1.578*x1*x4 + 1.132*x4*x2 + 0.6,
        -x1*x5 + 0.7,
        x1*x4 + 1
    ])

# Definindo a funcao F(x)
def F(x):
    return A @ x + g(x)

# Definindo a Jacobiana J(x)
def J(x):
    x1, x2, x3, x4, x5 = x
    return np.array([
        [-3.933, 0.107 - 0.727*x3 + 63.5*x4, 0.126 - 0.727*x2 +
           8.39*x4, 8.39*x3 - 684.4*x5 + 63.5*x2, -684.4*x4 -
           9.99],
        [0.949*x3 + 0.173*x5, -0.987, 0.949*x1, 0, 0.173*x1],
        [-0.716*x2 - 1.578*x4, -0.716*x1 + 1.132*x4, 0, -1.578*x1
            + 1.132*x2, 0],
        [-x5, 1.0, 0, -1.0, -x1],
        [x4, 0, -1.0, x1, -0.196]
    ])

```

```
# Condicao inicial
x = np.array([0.2, 0.5, 0.85, 0.93, 0.22])

# Criterio de parada
tolerance = 1e-6

# Metodo de Newton-Raphson
def newton_raphson(x, tolerance):
    while np.linalg.norm(F(x)) > tolerance:
        delta_x = np.linalg.solve(J(x), -F(x))
        x = x + delta_x
    return x

# Executando o metodo
x_sol = newton_raphson(x, tolerance)
print("Solucao:", x_sol)
```

Listing 7: Resolução do sistema não linear

A solução encontrada é: $\begin{bmatrix} -4.97500107 \\ -10.97244334 \\ 60.54818574 \\ -11.95614875 \\ -0.33843318 \end{bmatrix}$.

## 6.2 Plotando a norma quadrática $\|F(x)\|$ ao longo do runtime do algoritmo, temos o código:

```
import numpy as np
import matplotlib.pyplot as plt

# Definindo a matriz A
A = np.array([
    [-3.933, 0.107, 0.126, 0, -9.99],
    [0, -0.987, 0, -22.95, 0],
    [0.002, 0, -0.235, 0, 5.67],
    [0, 1.0, 0, -1.0, 0],
    [0, 0, -1.0, 0, -0.196]
])

# Definindo a funcao g(x)
def g(x):
    x1, x2, x3, x4, x5 = x
    return np.array([
        -0.727*x2*x3 + 8.39*x3*x4 - 684.4*x4*x5 + 63.5*x4*x2 +
           0.2,
        0.949*x1*x3 + 0.173*x1*x5 + 0.35,
        -0.716*x1*x2 - 1.578*x1*x4 + 1.132*x4*x2 + 0.6,
        -x1*x5 + 0.7,
        x1*x4 + 1
```

```
    ])

# Definindo a funcao F(x)
def F(x):
    return A @ x + g(x)

# Definindo a Jacobiana J(x)
def J(x):
    x1, x2, x3, x4, x5 = x
    return np.array([
        [-3.933, 0.107 - 0.727*x3 + 63.5*x4, 0.126 - 0.727*x2 +
           8.39*x4, 8.39*x3 - 684.4*x5 + 63.5*x2, -684.4*x4 -
           9.99],
        [0.949*x3 + 0.173*x5, -0.987, 0.949*x1, 0, 0.173*x1],
        [-0.716*x2 - 1.578*x4, -0.716*x1 + 1.132*x4, 0, -1.578*x1
            + 1.132*x2, 0],
        [-x5, 1.0, 0, -1.0, -x1],
        [x4, 0, -1.0, x1, -0.196]
    ])

# Condicao inicial
x = np.array([0.2, 0.5, 0.85, 0.93, 0.22])

# Criterio de parada
tolerance = 1e-6

# Metodo de Newton-Raphson com armazenamento da norma de F(x)
def newton_raphson(x, tolerance):
    norms = []
    while True:
        norm = np.linalg.norm(F(x))
        norms.append(norm)
        if norm < tolerance:
            break
        delta_x = np.linalg.solve(J(x), -F(x))
        x = x + delta_x
    return x, norms

# Executando o metodo
x_sol, norms = newton_raphson(x, tolerance)

# Plotando a norma quadratica de F(x)
plt.plot(norms)
plt.yscale('log')  # Escala logaritmica para visualizar melhor a
   convergencia
plt.xlabel('Iteracao')
plt.ylabel('Norma quadratica de F(x)')
plt.title('Convergencia do Metodo de Newton-Raphson')
plt.grid(True)
plt.show()
```

```
68
69 print("Solucao:", x_sol)
```

Listing 8: Plot da norma quadrática

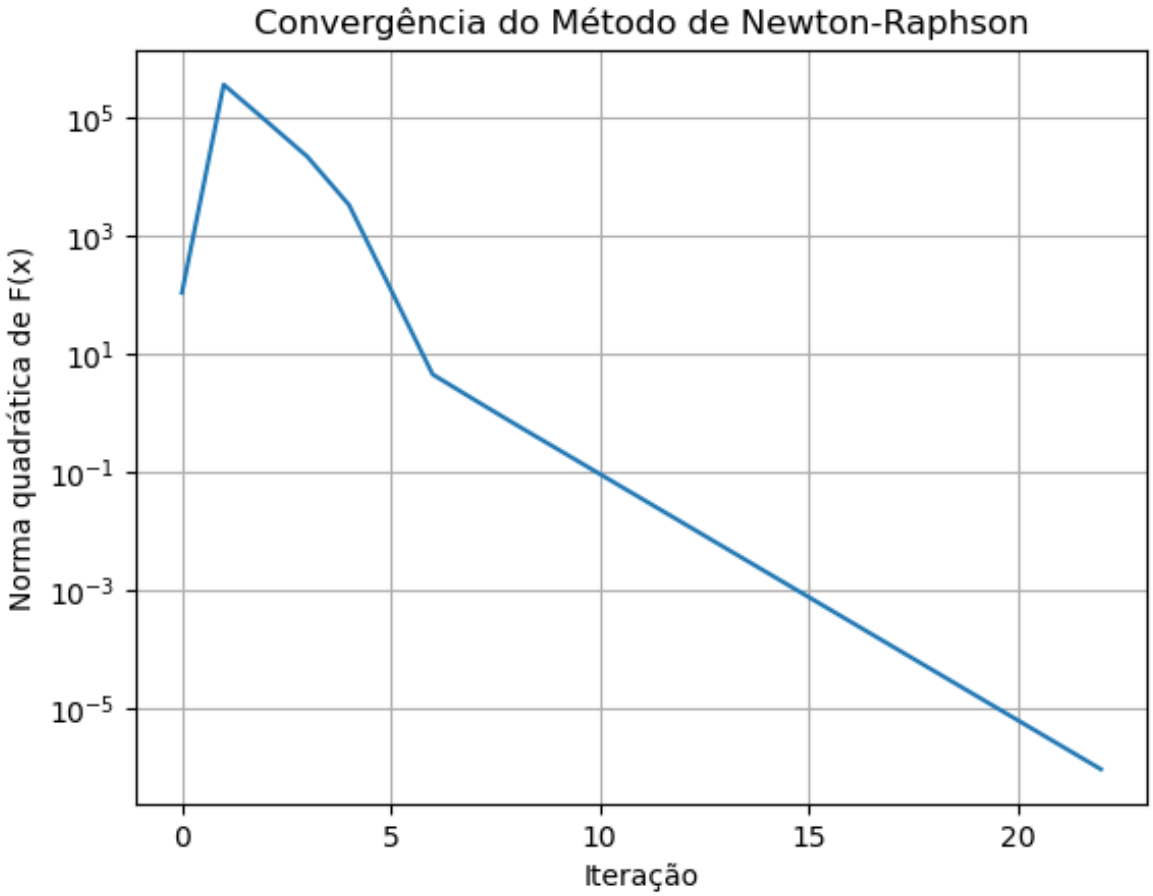


Figura 16: Norma quadrática

## 6.3 Colocando em uma tabela os 5 primeiros valores achados para $x$ pelo método de Newton, temos:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # Definindo a matriz A
5  A = np.array([
6      [-3.933, 0.107, 0.126, 0, -9.99],
7      [0, -0.987, 0, -22.95, 0],
8      [0.002, 0, -0.235, 0, 5.67],
9      [0, 1.0, 0, -1.0, 0],
10     [0, 0, -1.0, 0, -0.196]
11 ])
12
```

```
# Definindo a funcao g(x)
def g(x):
    x1, x2, x3, x4, x5 = x
    return np.array([
        -0.727*x2*x3 + 8.39*x3*x4 - 684.4*x4*x5 + 63.5*x4*x2 + 0.2,
        0.949*x1*x3 + 0.173*x1*x5 + 0.35,
        -0.716*x1*x2 - 1.578*x1*x4 + 1.132*x4*x2 + 0.6,
        -x1*x5 + 0.7,
        x1*x4 + 1
    ])

# Definindo a funcao F(x)
def F(x):
    return A @ x + g(x)

# Definindo a Jacobiana J(x)
def J(x):
    x1, x2, x3, x4, x5 = x
    return np.array([
        [-3.933, 0.107 - 0.727*x3 + 63.5*x4, 0.126 - 0.727*x2 + 8.39*x4, 8.39*x3 - 684.4*x5 + 63.5*x2, -684.4*x4 - 9.99],
        [0.949*x3 + 0.173*x5, -0.987, 0.949*x1, 0, 0.173*x1],
        [-0.716*x2 - 1.578*x4, -0.716*x1 + 1.132*x4, 0, -1.578*x1 + 1.132*x2, 0],
        [-x5, 1.0, 0, -1.0, -x1],
        [x4, 0, -1.0, x1, -0.196]
    ])

# Condicao inicial
x = np.array([0.2, 0.5, 0.85, 0.93, 0.22])

# Criterio de parada
tolerance = 1e-6

# Metodo de Newton-Raphson com armazenamento da norma de F(x) e valores de x
def newton_raphson(x, tolerance):
    norms = []
    x_values = []
    iteration = 0
    while True:
        norm = np.linalg.norm(F(x))
        norms.append(norm)
        x_values.append(x.copy())
        if norm < tolerance or iteration >= 5:
            break
        delta_x = np.linalg.solve(J(x), -F(x))
        x = x + delta_x
```

```
        iteration += 1
    return x, norms, x_values

# Executando o metodo
x_sol, norms, x_values = newton_raphson(x, tolerance)

# Plotando a norma quadratica de F(x)
plt.plot(norms)
plt.yscale('log')  # Escala logaritmica para visualizar melhor a
    convergencia
plt.xlabel('Iteracao')
plt.ylabel('Norma quadratica de F(x)')
plt.title('Convergencia do Metodo de Newton-Raphson')
plt.grid(True)
plt.show()

# Imprimindo os 5 primeiros valores de x
import pandas as pd

columns = ['x1', 'x2', 'x3', 'x4', 'x5']
df = pd.DataFrame(x_values, columns=columns)
print("5 primeiros valores de x:")
print(df.head(5))

print("Solucao final:", x_sol)
```

Listing 9: Tabela dos 5 primeiros valores de $x$

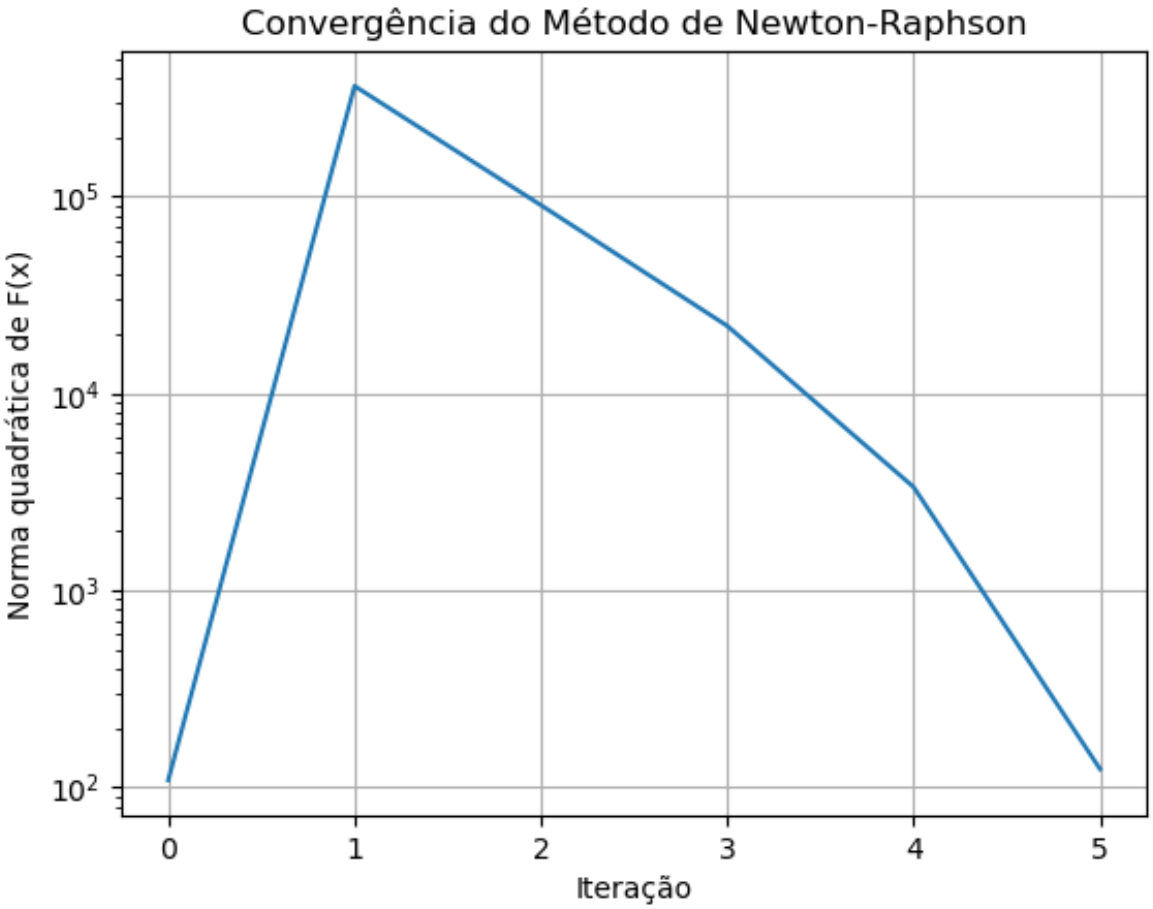


Figura 17: Norma quadrática