Relatório de Cálculo Numérico

Alison de Oliveira Tristão
Vitor João de Andrade

9 de dezembro de 2024

1 Introdução

Este relatório aborda a aplicação de métodos numéricos para resolver problemas matemáticos. São explorados cinco questões principais, desde a aproximação de integrais, resolução de equações diferenciais parciais (EDPs), representação de sistemas lineares, interpolação polinomial até a otimização de funções. A utilização de métodos como os de Retângulos, Trapézios e Simpson para integração, Euler e Runge-Kutta para EDPs, bem como técnicas como minimização por LDLT, fatoração de Cholesky e gradiente residual para ajuste de sistemas lineares, com implementações em Python.

2 Problema 1: Integração

O cálculo de integrais é um problema comum em cálculo, simulação e programação, especialmente quando a função não tem uma expressão analítica ou é difícil de ser definida. Esse tipo de problema aparece em diversas áreas, como processamento de sinais, controle de sistemas embarcados, e simulação de processos físicos. Muitas vezes, a solução numérica é suficiente, uma vez que buscamos apenas um valor aproximado da integral, permitindo uma flexibilidade na precisão computacional com controle de custo e tempo de execução.

Para calcular a integral, utilizamos diferentes métodos de aproximação utilizando amostragem da função e interpolação entre as amostras, sendo as mais comuns:

- **Método dos Retângulos:** Aproxima o valor entras as amostras com uma reta horizontal, mantendo o valor da amostra constante até a próxima, que pode ser da esquerda para direita, vice-versa ou usando o ponto médio.
- **Método dos Trapézios:** Conecta duas amostras consecutivas por uma linha reta, proporcionando uma aproximação mais precisa da área sob a curva.
- **Método de Simpson:** Usa uma parábola para aproximar a função entre duas amostras consecutivas, oferecendo uma precisão ainda maior em comparação com os métodos anteriores.

Para resolver numericamente a integral da função $\sin(t^2)$ entre 0 e $x = 20$, utilizamos esses três métodos, variando o tamanho dos passos $h = \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$.

2.1 Método dos Retângulos

O método dos retângulos aproxima a integral de uma função somando áreas de retângulos, com base na largura h e na altura dada pelos valores da função. A fórmula do método é:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(x_i) \cdot h$$

Onde $h = \frac{b-a}{n}$ e $x_i = a + i \cdot h$.

A função em Python para este método é a seguinte:

```python
def retangle_method(func, a, b, h):
    # number of points in interval
    N = int((b - a)/h)

    # sum of all points with a h step
    result = 0
    for i in range(N -1):
        result += func(a + i * h)

    # return the sum multiplied by space between points
    return result * h
```

Listing 1: Python - Método dos Retângulos

Os resultados para diferentes valores de h são apresentados na Tabela 1.

h	Valor	Tempo (s)
1	-4.02525	0.000004
10^{-1}	0.59766	0.000035
10^{-2}	0.64968	0.000483
10^{-3}	0.64106	0.004036
10^{-4}	0.63994	0.036321
10^{-5}	0.63983	0.375208

Tabela 1: Resultados do Método dos Retângulos

2.2 Método dos Trapézios

O método dos trapézios usa uma linha reta para conectar amostras consecutivas. A fórmula para esse método é:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right)$$

Onde $h = \frac{b-a}{n}$ e $x_i = a + i \cdot h$.

A função Python correspondente é:

```python
def trapeze_method(fun, a, b, h):
    # number of points in interval
    N = int((b - a)/h)

    # sum all points with a space between them
    result = 0
    for i in range(1, N -1):
        result += fun(a + i * h) * 2

    # sum with the first and last point
    result += (fun(a) + fun(b))

    # multiply by the space between points
    return result * h/2
```

Listing 2: Python - Método dos Trapézios

Os resultados estão na Tabela 2.

h	Valor	Tempo (s)
1	-3.09559	0.000004
10^{-1}	0.59767	0.000054
10^{-2}	0.64543	0.000855
10^{-3}	0.64064	0.006119
10^{-4}	0.63990	0.039687
10^{-5}	0.63983	0.378847

Tabela 2: Resultados do Método dos Trapézios

2.3 Método de Simpson

O método de Simpson utiliza parábolas para aproximar a função. Sua fórmula é:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left(f(a) + 4\sum_{i=1,\,\text{ímpar}}^{n-1} f(x_i) + 2\sum_{i=2,\,\text{par}}^{n-2} f(x_i) + f(b)\right)$$

A função Python correspondente é:

```python
def simpson_method(func, a, b, h):
    # number of points in interval
    N = int((b - a)/h)

    # sum all odd points
    result = 0
    for i in range(1, N - 1, 2):
        result += 4 * func(a + i * h)

    # sum all even points
    for i in range(2, N - 2, 2):
        result += 2 * func(a + i * h)

    # sum the first and last points
    result += func(a) + func(b)

    # multiply by the space between points
    return result * h/3
```

Listing 3: Python - Método de Simpson

Os resultados para os espaços h estão na Tabela 3.

h	Valor	Tempo (s)
1	-3.59979	0.000003
10^{-1}	0.99788	0.000037
10^{-2}	0.64898	0.000644
10^{-3}	0.64146	0.004454
10^{-4}	0.63999	0.039835
10^{-5}	0.63983	0.375584

Tabela 3: Resultados do Método de Simpson

3 Questão 2: Equação de Calor no Espaço-Tempo

Uma EDP bastante estudada é a equação do calor:

$$\frac{\partial T}{\partial t} = k\frac{\partial^2 T(x,t)}{\partial x^2} - q(x,t) \tag{1}$$

Esta equação descreve os valores de temperatura $\mathbf{T}$ ao longo de um eixo unidimensional e ao longo do tempo, onde $x \in (0,1)$. Para resolver a equação, consideramos o fluxo de calor exógeno $q(x,t) = 0\ \forall x,t$, por motivos de simplificação, e então começamos definindo as condições de contorno $\forall t$:

$$\begin{cases} T(0,t) = T_{\text{begin}} \\ T(1,t) = T_{\text{end}} \end{cases} \tag{2}$$

Também definimos as condições iniciais no tempo, $\forall x \in (0,1)$:

$$T(x,0) = T0 \tag{3}$$

Aplicando o método das Diferenças Finitas, a EDP pode ser representada como uma EDO, aproximada pela seguinte equação discreta:

$$\dot{\mathbf{T}} = \left(\frac{k}{h^2}\right)(\mathbf{D_2 T} - \mathbf{BC}) \tag{4}$$

Onde $\mathbf{D_2}$, $\mathbf{T}$, $\mathbf{Q}$ e $\mathbf{BC}$ 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:

$$x_{i+1} = x_i + h \tag{5}$$

Assim, podemos definir nossa matriz de temperaturas como:

$$\mathbf{T} = \begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_N \end{pmatrix} = \begin{pmatrix} T(x_1) \\ T(x_2) \\ \vdots \\ T(x_N) \end{pmatrix} \tag{6}$$

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

$$N = \frac{1}{h} - 1 \tag{7}$$

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

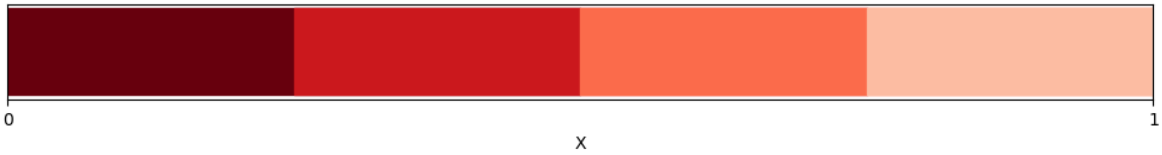

Figura 1: Discretização da Temperatura no Eixo x

A matriz $\mathbf{D_2} \in \mathbb{R}^{N\times N}$ é construída através da discretização da derivada parcial no espaço. A matriz $\mathbf{BC} \in \mathbb{R}^{N}$ resulta das condições de contorno. As representações dessas matrizes são dadas a seguir:

$$\mathbf{D_2} = \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \tag{8}$$

$$\mathbf{BC} = \begin{pmatrix} T_{\text{begin}} \\ 0 \\ \vdots \\ T_{\text{end}} \end{pmatrix} \tag{9}$$

Generalizamos a matriz D_2 da seguinte forma: para a linha i e a coluna j, a estrutura é dada por:

$$\mathbf{D_2}(i,j) = \begin{cases} -2 & \text{se } i = j \\ 1 & \text{se } i = j+1 \text{ ou } i = j-1 \\ 0 & \text{caso contrário} \end{cases} \tag{10}$$

Sabendo disso, iremos resolver três situações de maneira numérica utilizando os métodos Euler e Runge–Kutta.

- $k = 2.5$
- $T_{\text{begin}} = 60$
- $T_{\text{end}} = 40$
- $T0 = 50$

Considerando:

$$h = \begin{cases} 0.2 & \text{caso (a)} \\ 0.1 & \text{caso (b)} \\ 0.05 & \text{caso (c)} \end{cases} \tag{11}$$

Sabendo disso, para aplicar a resolução, usamos o método de Euler como aproximação de primeira ordem, definido como:

$$T_{n+1} = T_n + \Delta t \cdot f(T_n) \tag{12}$$

Também o método de Runge-Kutta como uma aproximação de quarta ordem:

$$
\begin{aligned}
k_1 &= \Delta t \cdot f(T_n) \\
k_2 &= \Delta t \cdot f\left(T_n + \frac{k_1}{2}\right) \\
k_3 &= \Delta t \cdot f\left(T_n + \frac{k_2}{2}\right) \\
k_4 &= \Delta t \cdot f(T_n + k_3) \\
T_{n+1} &= T_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}
\tag{13}
$$

Utilizando Python, o algoritmo para solução desenvolvido foi:

```python
def function(T):
    return const * (D2 @ T + BC)

def euler_step(T, step):
    return T + step * function(T)

def runge_kutta_step(T, step):
    k1 = step * function(T)
    k2 = step * function(T + 0.5 * k1)
    k3 = step * function(T + 0.5 * k2)
    k4 = step * function(T + k3)
    return T + (k1 + 2*k2 + 2*k3 + k4) / 6

def solver(T, t_end, dt, step_method):
    time_steps = int(t_end/dt)
    T_history = np.zeros((time_steps, N))

    for i in range(time_steps):
        T_history[i] = T
        T = step_method(T, dt)

    return T_history.T

min_step = 1/max(np.linalg.eig(abs(const*D2))[0])
time_simulation = 0.05

step = min_step
if time_simulation/min_step > 1000:
    print("Quantidade maxima de steps excedidas!")
    step = time_simulation/1000

result_euler = solver(np.full(N, t0, dtype=float),
   time_simulation, step, euler_step)
result_runge = solver(np.full(N, t0, dtype=float),
   time_simulation, step, runge_kutta_step)
```

Listing 4: Python - Euler e Runge-Kutta

Para calcular a constante de tempo, calculamos o inverso do maior autovalor de $\left(\frac{k}{h^2}\right)\mathbf{D_2}$.

3.1 Situação 1: $h = 0.2$

O comportamento no tempo resolvendo com o método de Euler:

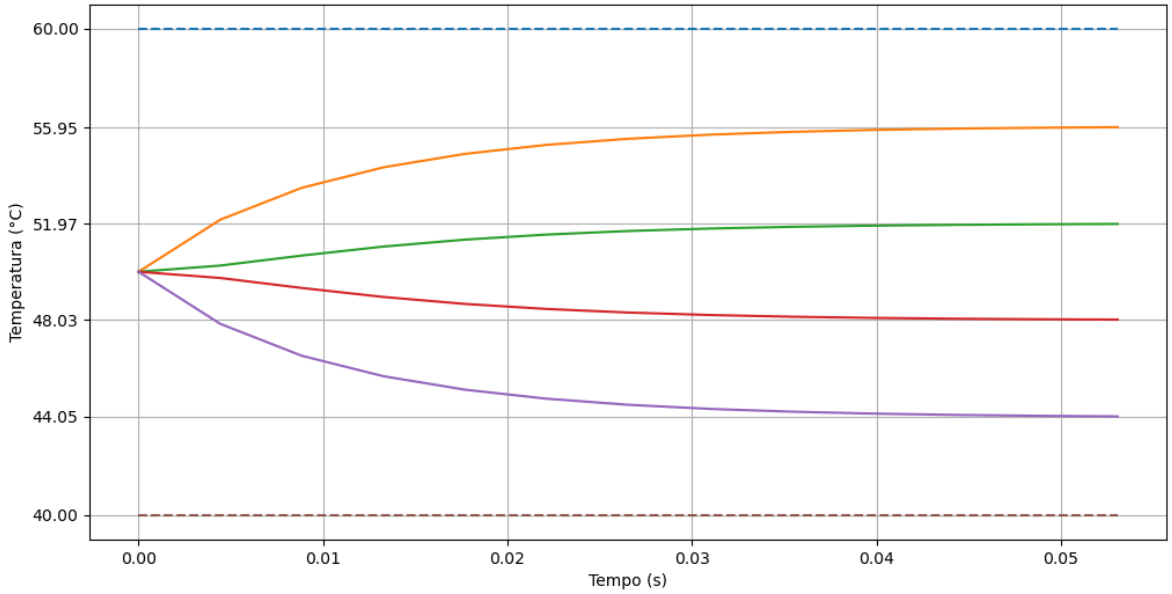

Figura 2: Temperatura por tempo: $h = 0.2$, Método de Euler

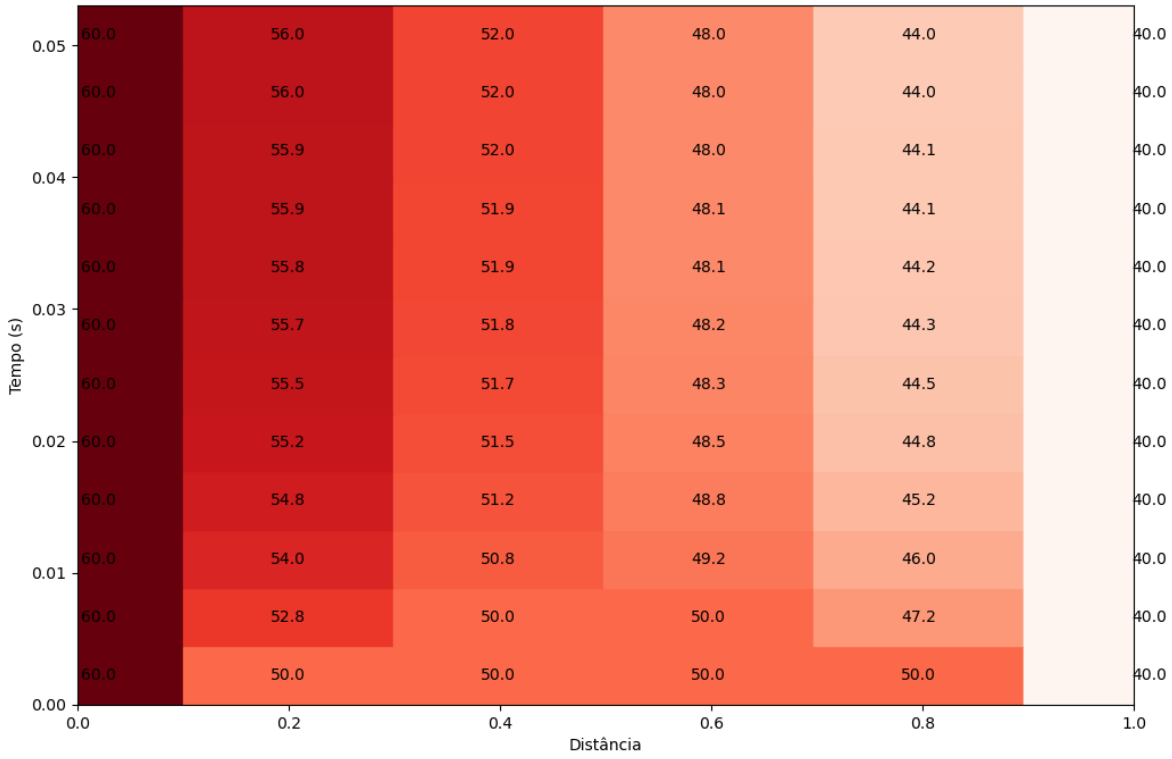

Figura 3: Temperatura por tempo e espaço: $h = 0.2$, Método de Euler

Resolvendo com o método de Runge-Kutta:

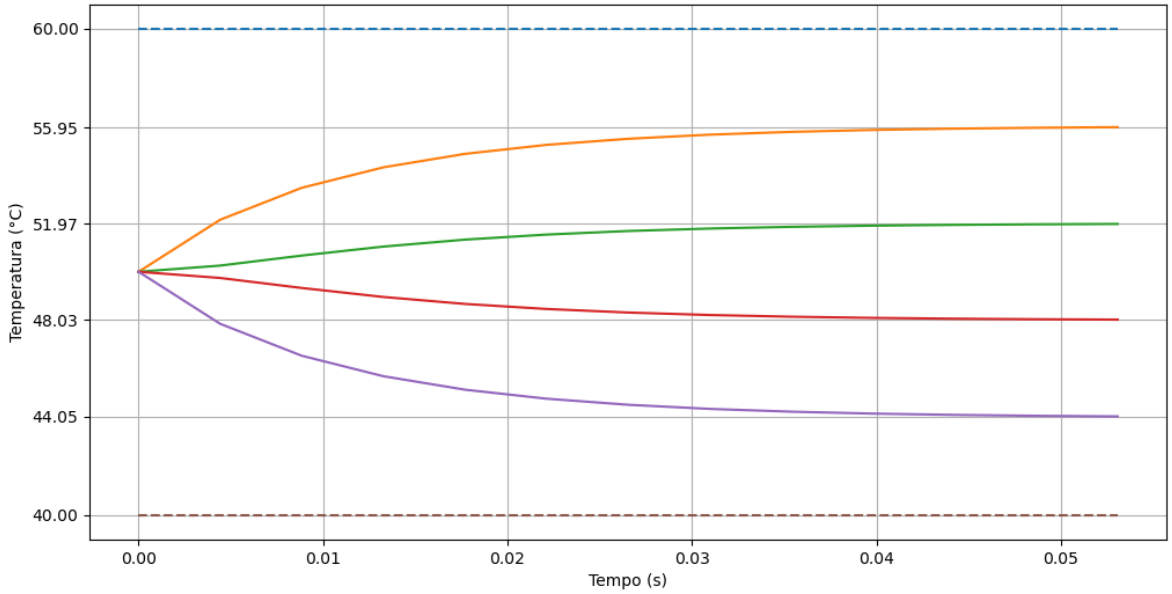

Figura 4: Temperatura por tempo: $h = 0.2$, Método de Runge-Kutta

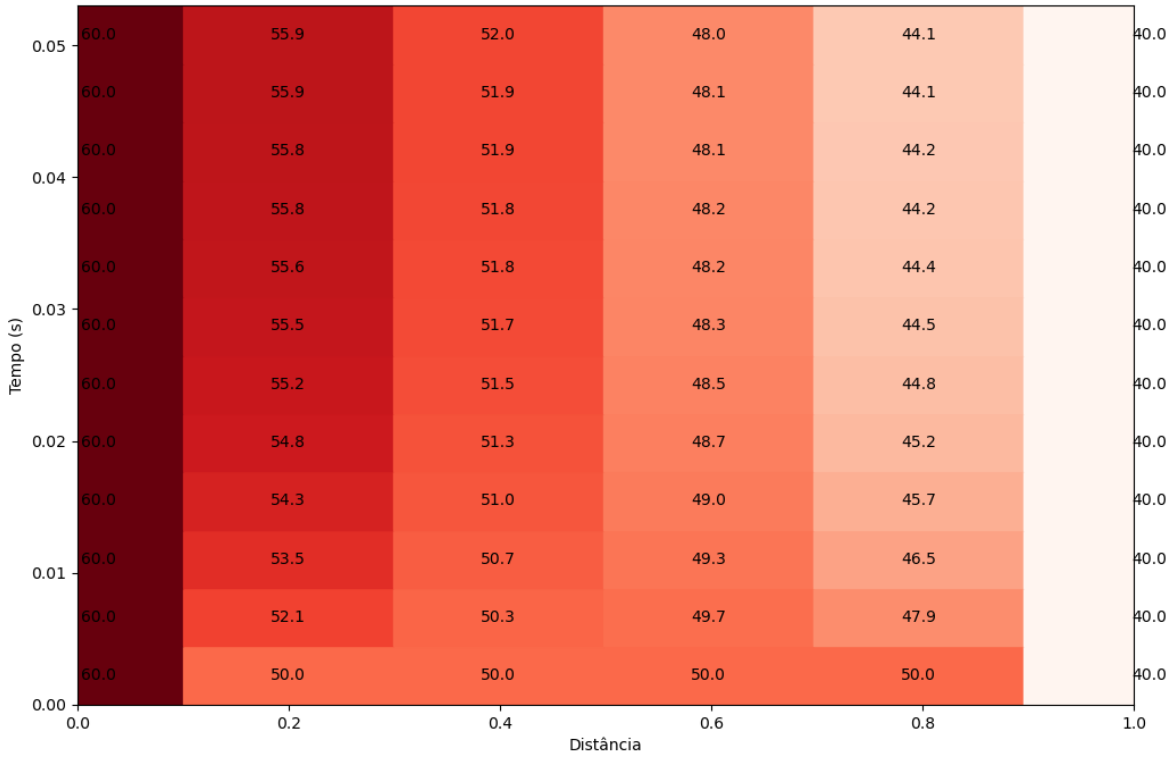

Figura 5: Temperatura por tempo e espaço: $h = 0.2$, Método de Runge-Kutta

3.2 Situação 2: $h = 0.1$

O comportamento no tempo resolvendo com o método de Euler:

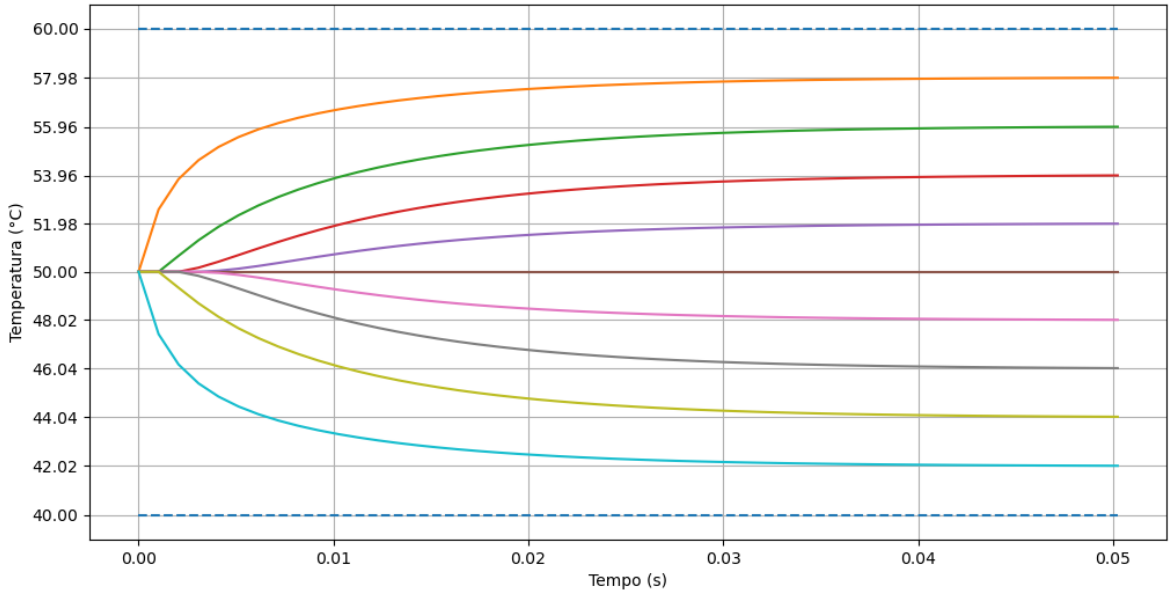

Figura 6: Temperatura por tempo: $h = 0.1$, Método de Euler

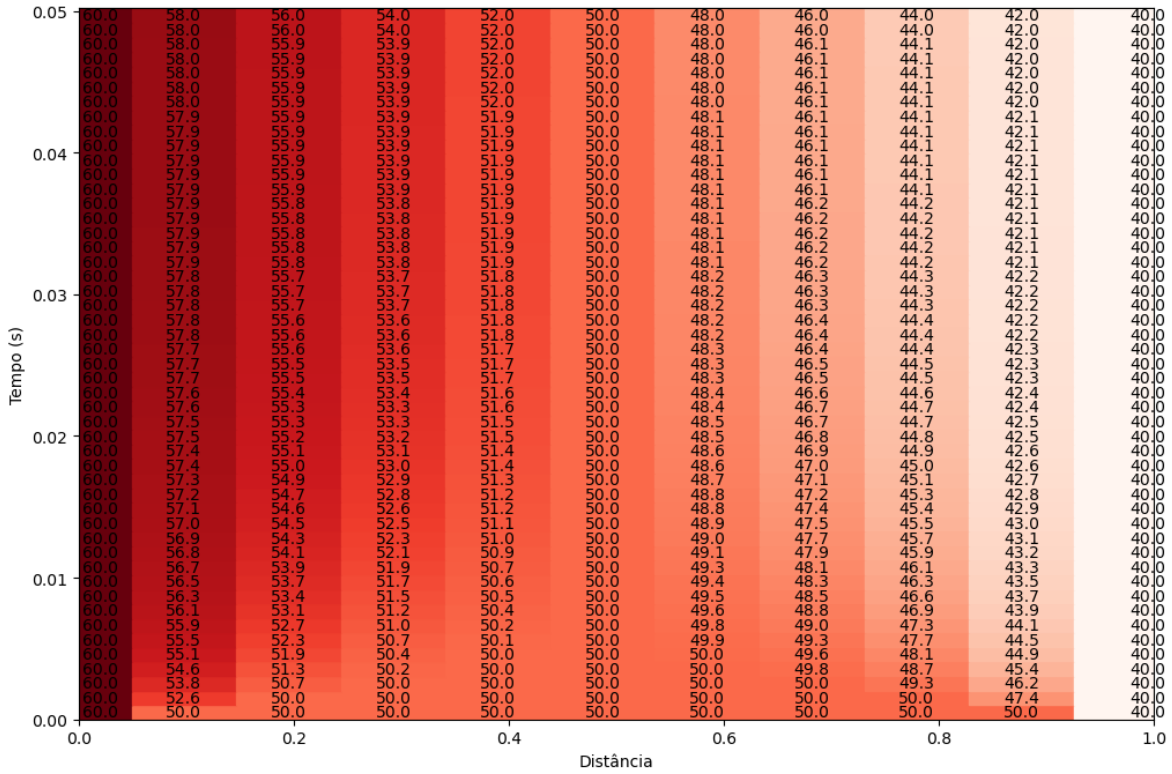

Figura 7: Temperatura por tempo e espaço: $h = 0.1$, Método de Euler

Resolvendo com o método de Runge-Kutta:

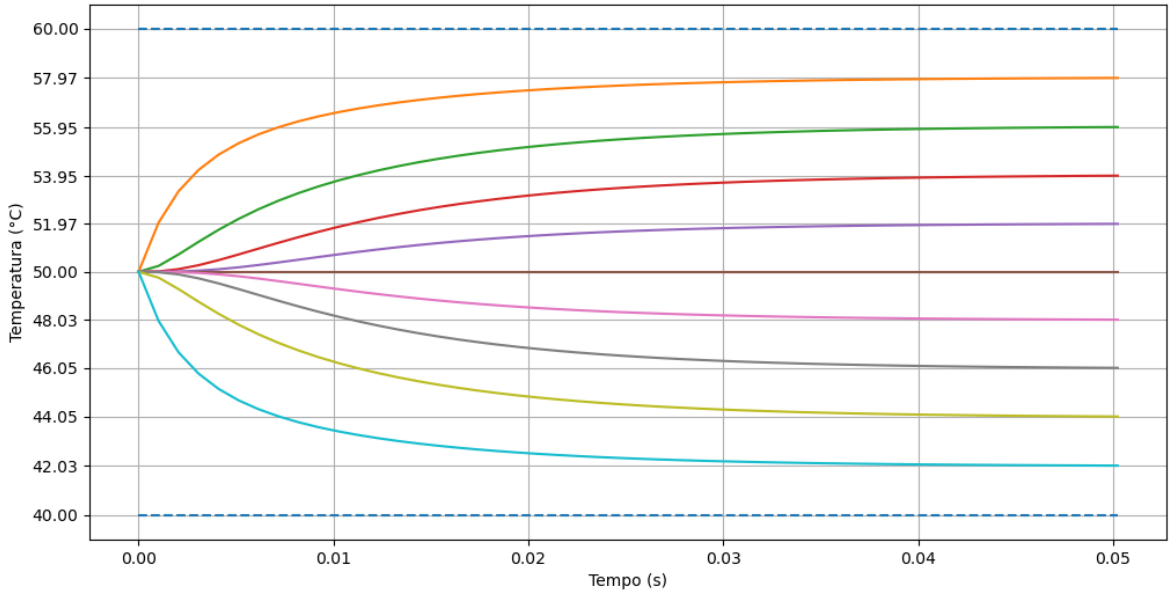

Figura 8: Temperatura por tempo: $h = 0.1$, Método de Runge-Kutta

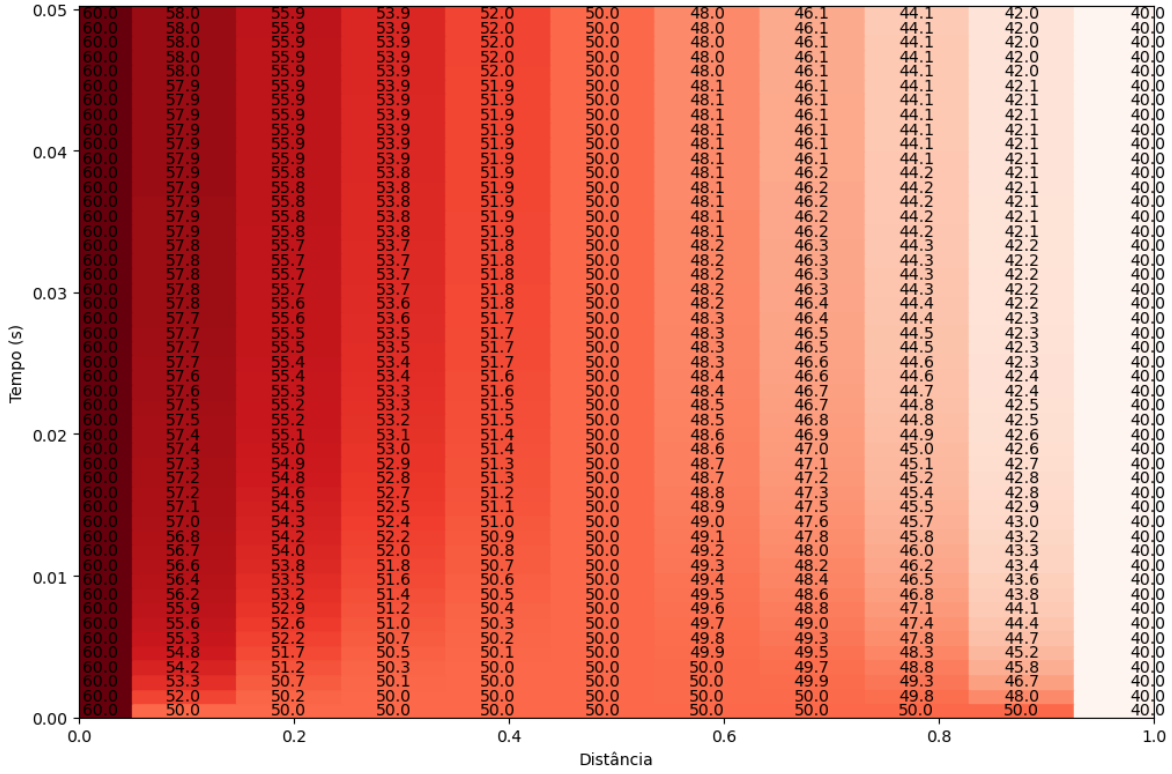

Figura 9: Temperatura por tempo e espaço: $h = 0.1$, Método de Runge-Kutta

3.3 Situação 3: $h = 0.05$

O comportamento no tempo resolvendo com o método de Euler:

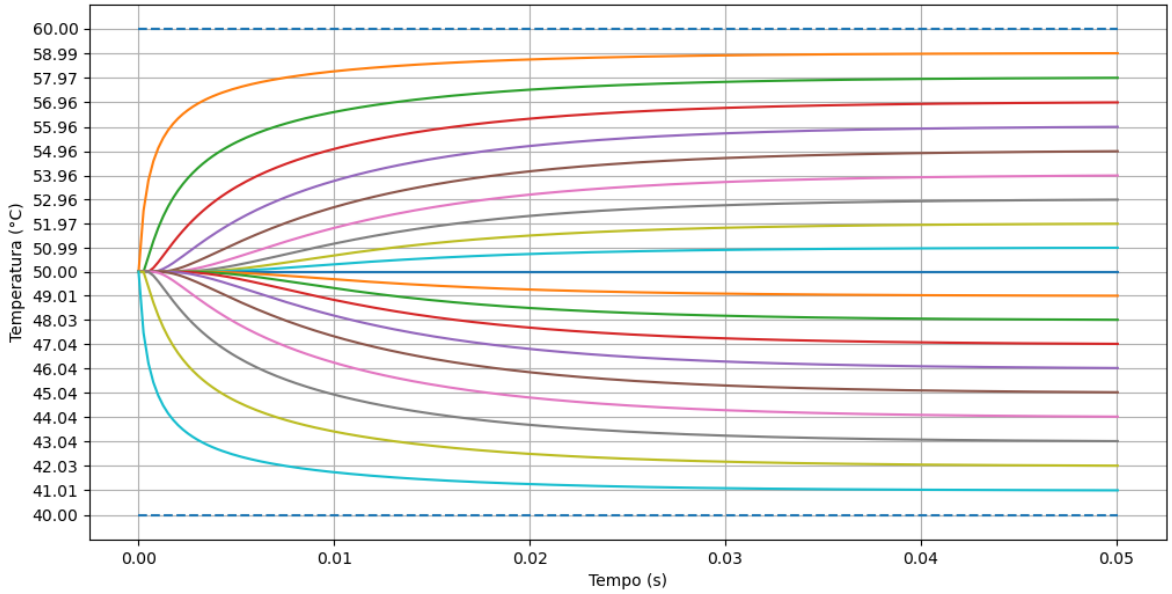

Figura 10: Temperatura por tempo: $h = 0.05$, Método de Euler

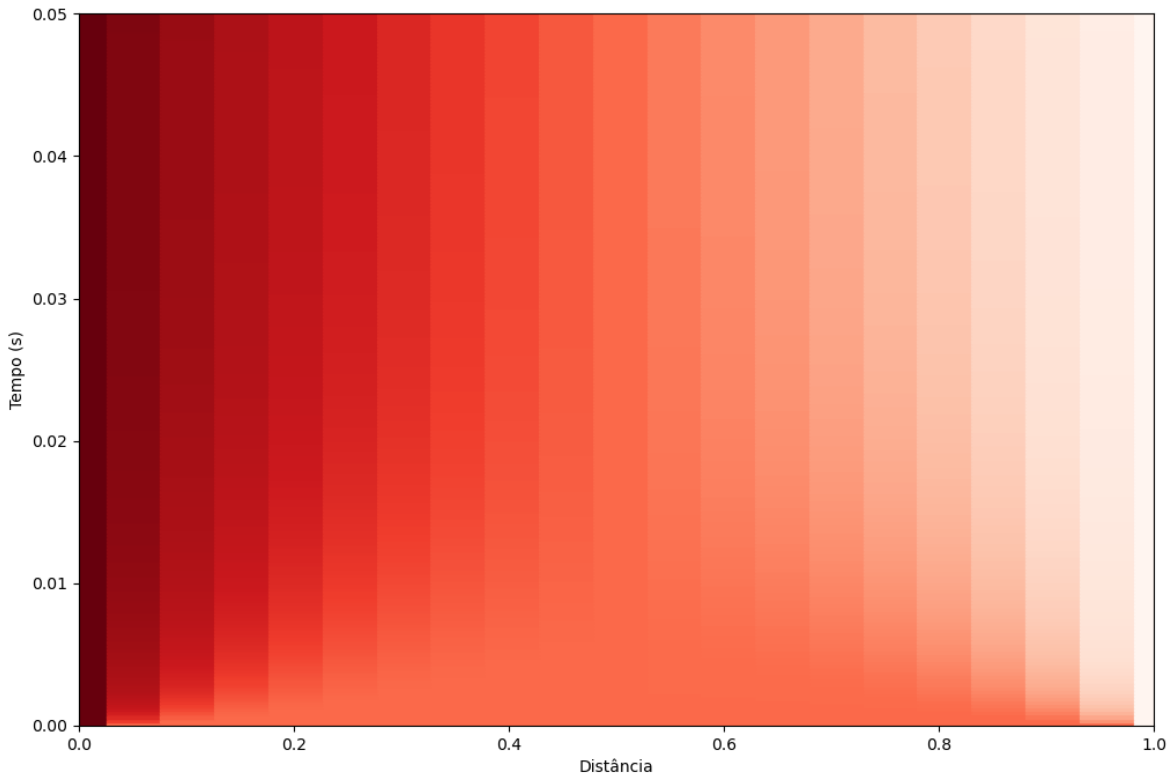

Figura 11: Temperatura por tempo e espaço: $h = 0.05$, Método de Euler

Resolvendo com o método de Runge-Kutta:

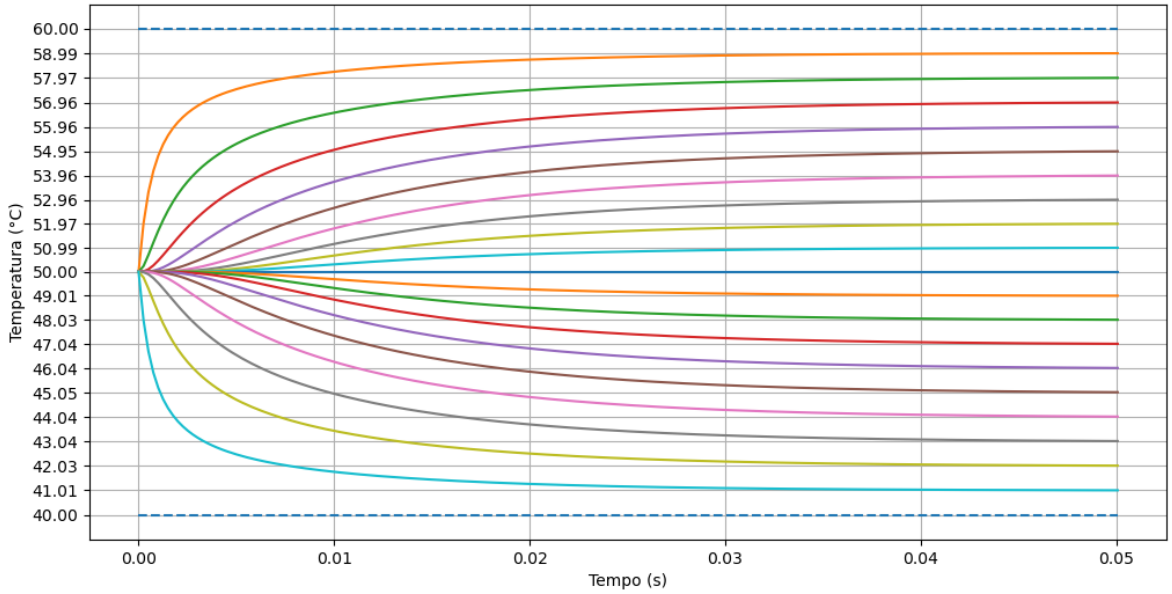

Figura 12: Temperatura por tempo: $h = 0.05$, Método de Runge-Kutta

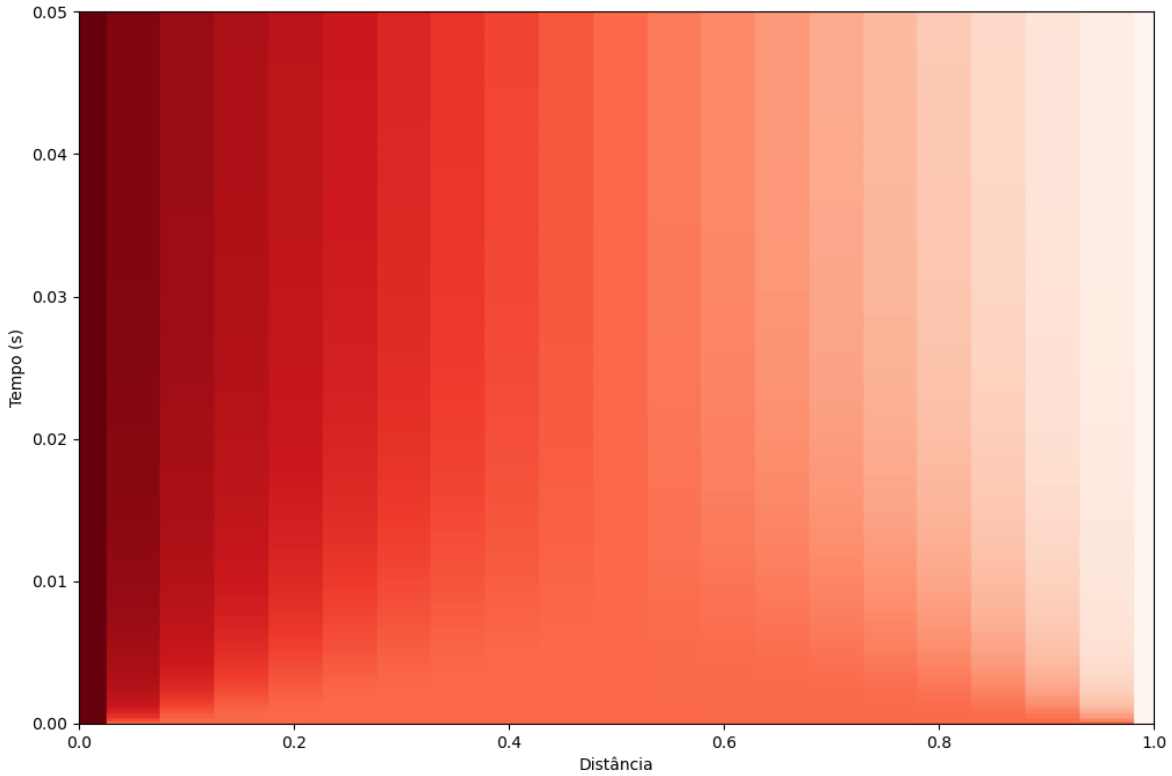

Figura 13: Temperatura por tempo e espaço: $h = 0.05$, Método de Runge-Kutta

3.4 Alterando a Situação de Contorno T_{begin}

Incrementando T_{begin} em 5 a cada 0.2 e simulando ao longo de 2 segundos temos:

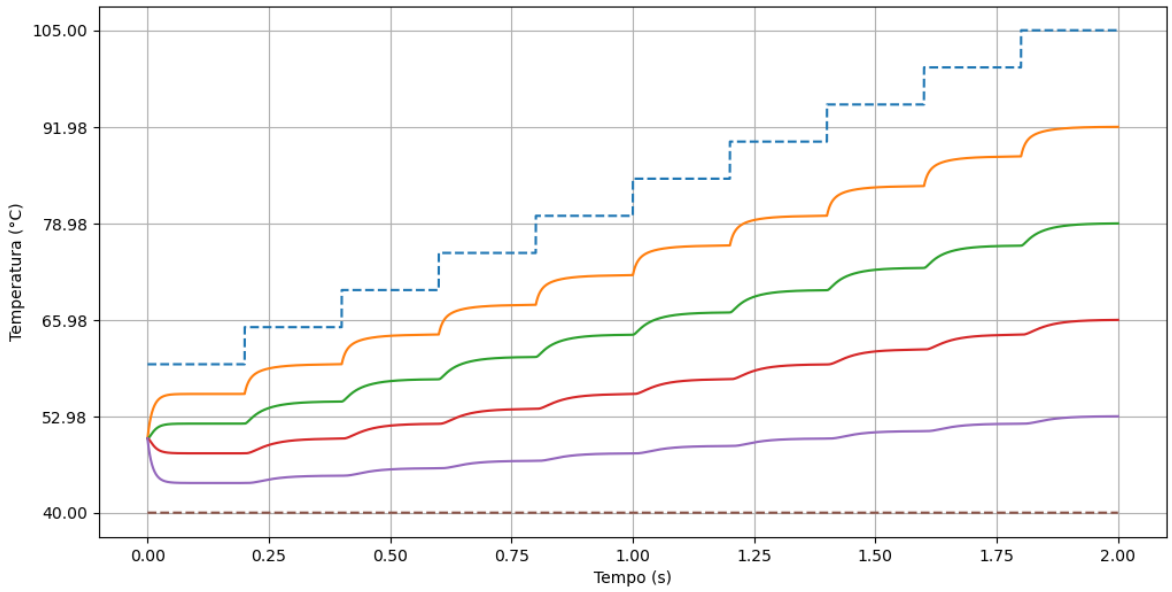

Figura 14: Temperatura por tempo: $h = 0.2$, Método de Euler (Variação de T_{begin})

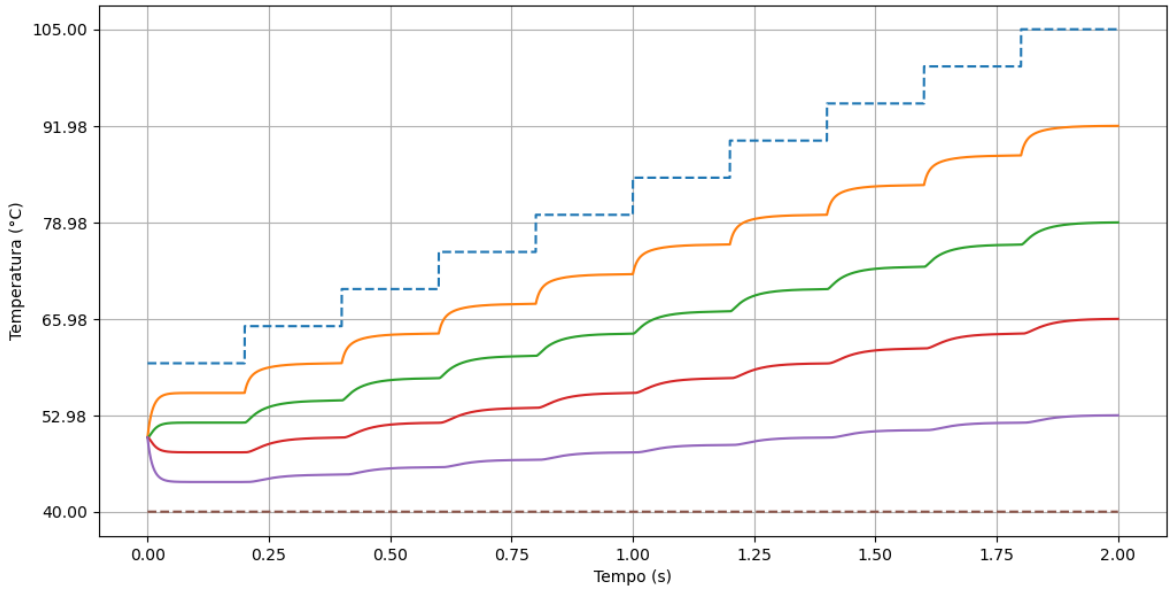

Figura 15: Temperatura por tempo: $h = 0.2$, Método de Runge-Kutta (Variação de T_{begin})

4 Questão 3: Sistema Linear

Observamos que, na questão anterior, é possível representar o comportamento temporal de cada valor de temperatura discreta em x como:

$$\mathbf{T}[k+1] = \mathbf{A} \cdot \mathbf{T}[k] + \mathbf{B} \cdot T_{\text{begin}}[k], \tag{14}$$

onde $\mathbf{A}$ é uma matriz $N \times N$ e $\mathbf{B}$ é uma matriz coluna de tamanho N, com $N(h) = 1/h - 1$.

Usando os dados da simulação anterior com o método de Runge-Kutta, com $h = 0.2$ e incrementando o valor de T_{begin}, iremos determinar as matrizes $\mathbf{A}$ e $\mathbf{B}$ que aproximam o valor da temperatura mais próxima de T_{end}, ou seja, T_N. Para isso, aplicaremos o método dos mínimos quadrados, minimizando os valores de $\mathbf{A}$ e $\mathbf{B}$ pela expressão:

$$\min_{\mathbf{A},\mathbf{B}} J(\mathbf{A}, \mathbf{B}) = \frac{1}{2} \sum_{k=0}^{K} \|\mathbf{T}[k+1] - (\mathbf{A} \cdot \mathbf{T}[k] + \mathbf{B} \cdot T_{\text{begin}}[k])\|^2. \tag{15}$$

Para encontrar o mínimo dessa função, derivamos e igualamos a zero, localizando o ponto de mínimo global da função quadrática. Podemos reescrever a equação na forma matricial $\mathbf{Mx} = \mathbf{f}$, onde:

$$\mathbf{M} = \begin{bmatrix} \mathbf{T}^T[0] & T_{\text{begin}}[0] \\ \mathbf{T}^T[1] & T_{\text{begin}}[1] \\ \vdots & \vdots \\ \mathbf{T}^T[K] & T_{\text{begin}}[K] \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{A} \\ \mathbf{B}^T \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{T}_N^T[1] \\ \mathbf{T}_N^T[2] \\ \vdots \\ \mathbf{T}_N^T[K+1] \end{bmatrix}. \tag{16}$$

A função objetivo torna-se então:

$$\min_{\mathbf{x}} J(\mathbf{x}) = \frac{1}{2} \|\mathbf{M} \cdot \mathbf{x} - \mathbf{f}\|^2. \tag{17}$$

Derivando a equação, obtemos:

$$\frac{\partial J(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{M}^T(\mathbf{M} \cdot \mathbf{x} - \mathbf{f}). \tag{18}$$

Logo, igualando a zero e resolvendo o sistema, encontramos os valores de $\mathbf{A}$ e $\mathbf{B}$ que minimizam a função J. A solução é dada por:

$$\mathbf{M}^T\mathbf{M}\mathbf{x} = \mathbf{M}^T\mathbf{f}, \tag{19}$$

onde $\mathbf{M}^T\mathbf{M}$ é uma matriz simétrica e definida positiva, podendo ser resolvida com a fatoração de Cholesky.

Porém, não conseguimos solucionar o sistema de maneira satisfatória usando apenas os valores T_{begin} como entrada para o sistema. Isso ocorre porque, com apenas uma entrada, não conseguimos representar dois comportamentos diferentes: a descida no início e a resposta aos degraus após o primeiro incremento em T_{begin}. Como podemos ver no gráfico a seguir, mesmo acrescentando uma regularização em $\mathbf{x}$, o resultado ainda é insuficiente.

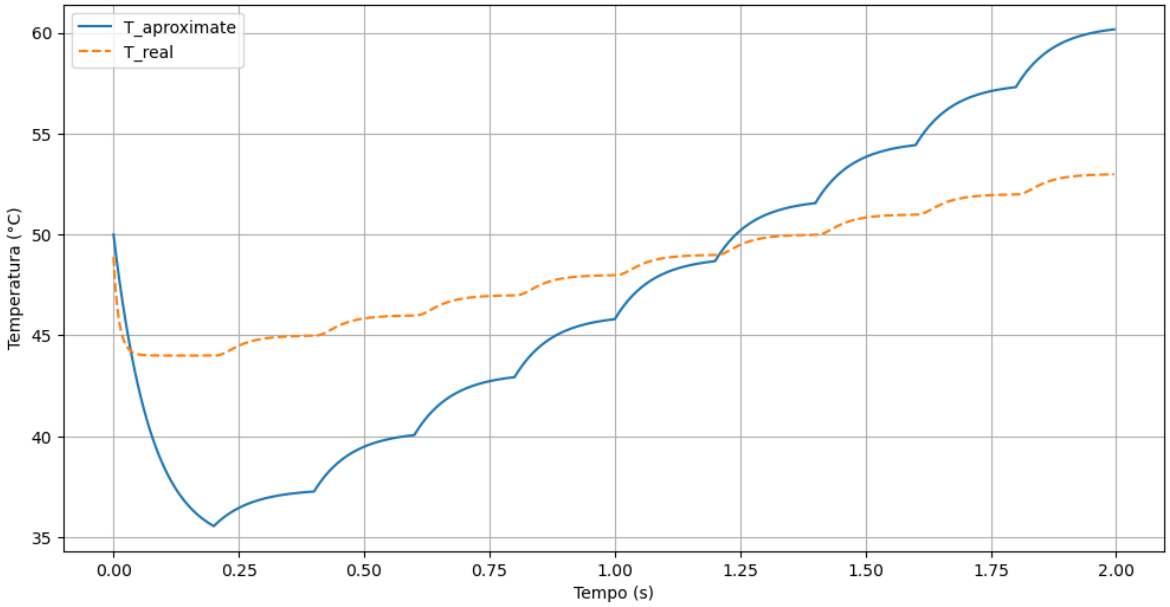

Figura 16: Representação do sistema de calor (apenas com T_{begin})

Então decidimos considerar também T_{end}, modificando a equação para:

$$\mathbf{T}[k+1] = \mathbf{A} \cdot \mathbf{T}[k] + \mathbf{B} \cdot T_{\text{begin}}[k] + \mathbf{C} \cdot T_{\text{end}}[k]. \tag{20}$$

O codigo desenvolvido em python foi:

```python
import numpy as np
from scipy.linalg import ldl, cholesky

def solver_linear(A, b):
    A_ = A.T @ A
    b_ = A.T @ b
    L = cholesky(A_)
    y = np.linalg.solve(L, b_)
    x = np.linalg.solve(L.T, y)

    return x

T = result_runge_with_incr.T
f = T[1:]

M = np.zeros((T.shape[0] -1, T.shape[1] +2))

for i in range(4):
    M[:,i] = result_runge_with_incr[i][:-1]
M[:, 4] = T_begin_history[:-1]
M[:, 5] = np.ones(T.shape[0] -1) * Tend

W = solver_linear(M, f)

```

```python
a = W[:4]
b = W[4]
c = W[5]

print(a, b, c)
```

Listing 5: Python - Gauss com Pivoteamento

Com isso, $\mathbf{M}$, $\mathbf{x}$ e $\mathbf{f}$ passam a ser:

$$\mathbf{M} = \begin{bmatrix} \mathbf{T}^T[0] & T_{\text{begin}}[0] & T_{\text{end}}[0] \\ \mathbf{T}^T[1] & T_{\text{begin}}[1] & T_{\text{end}}[1] \\ \vdots & \vdots & \\ \mathbf{T}^T[K] & T_{\text{begin}}[K] & T_{\text{end}}[K] \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{A} \\ \mathbf{B}^T \\ \mathbf{C}^T \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{T}^T[1] \\ \mathbf{T}^T[2] \\ \vdots \\ \mathbf{T}^T[K+1] \end{bmatrix}.$$

Resolvendo o sistema, conseguimos um resultado satisfatório:

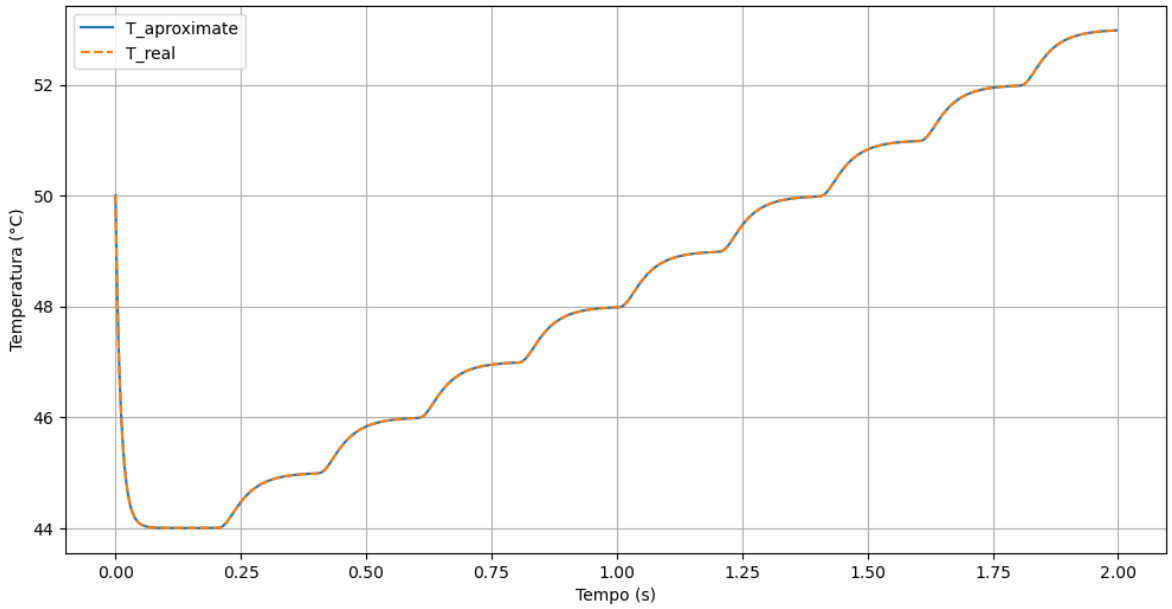

Figura 17: Representação do sistema de calor

Com as matrizes sendo:

$$\mathbf{A} = \begin{bmatrix} 0.78493 & 0.09781 & 0.00613 & 0.00024 \\ 0.09781 & 0.79106 & 0.09806 & 0.00613 \\ 0.00613 & 0.09806 & 0.79106 & 0.09781 \\ 0.00024 & 0.00613 & 0.09781 & 0.78493 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 0.110860 \\ 0.006652 \\ 0.000264 \\ 0.000010 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 0.000010 \\ 0.000264 \\ 0.006652 \\ 0.110860 \end{bmatrix}.$$

Onde, para T_N, o vetor $\mathbf{a}$ e as constantes b e c, responsáveis por seu comportamento, são dados por: $\mathbf{a} = \begin{bmatrix} 0.00024 & 0.00613 & 0.09781 & 0.78493 \end{bmatrix}^T$, $b = 0.000264$ e $c = 0.006652$.

5 Questão 4

Temos três pontos dados:

$$P_1(3,2), \quad P_2(7,5), \quad P_3(-5,-1)$$

A partir desses pontos, queremos encontrar um polinômio de oitava ordem $p(x)$ que os interpole. O polinômio tem a seguinte forma:

$$p(x) = \sum_{i=0}^{8} w_i x^i \tag{21}$$

Onde $w_0, w_1, \ldots, w_8$ são os coeficientes que devem ser determinados de modo que o polinômio passe pelos pontos fornecidos, minimizando a norma quadrática $\|w\|^2$, dada por:

$$\|\mathbf{w}\|^2 = w_0^2 + w_1^2 + \cdots + w_8^2 \tag{22}$$

De acordo com os pontos dados e a estrutura do polinômio, obtemos o sistema de equações:

$$\mathbf{A} = \begin{bmatrix} 1.0 & 3.0 & 9.0 & 27.0 & 81.0 & 243.0 & 729.0 & 2187.0 & 6561.0 \\ 1.0 & 7.0 & 49.0 & 343.0 & 2401.0 & 16807.0 & 117649.0 & 823543.0 & 5764801.0 \\ 1.0 & -5.0 & 25.0 & -125.0 & 625.0 & -3125.0 & 15625.0 & -78125.0 & 390625.0 \end{bmatrix}$$

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} 2 \\ 5 \\ -1 \end{bmatrix}$$

Então, montamos o sistema linear: $\mathbf{Aw} = \mathbf{b}$.

Para a minimização da norma, temos a seguinte expressão:

$$J(\mathbf{x}) = \|\mathbf{Aw} - \mathbf{b}\|^2 + \lambda\|\mathbf{w}\|^2 \tag{23}$$

Onde, encontramos seu mínimo derivando e igualando a zero:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = (\mathbf{A^T A} + \lambda \mathbf{I})\mathbf{w} - \mathbf{A^T b} \tag{24}$$

Para isso, precisamos apenas solucionar:

$$(\mathbf{A^T A} + \lambda \mathbf{I})\mathbf{w} = \mathbf{A^T b} \tag{25}$$

5.1 Minimização Usando LDL^T

Este sistema pode ser resolvido usando o método LDL^T com AMD, por conta de sua simetria em $\mathbf{A^T A}$. O código Python a seguir implementa a solução:

```python
import numpy as np
from scipy.linalg import ldl

POINTS = np.array([[3, 2], [7, 5], [-5, -1]])
ORDER = 8

def solver(A, b, lambda_value, ORDER):
    A_ = A.T @ A + lambda_value * np.eye(ORDER + 1)
    b_ = A.T @ b

    L, D, perm = ldl(A_)
    z = np.linalg.solve(L, b_)
    y = z / np.diag(D)
    x_regularizado = np.linalg.solve(L.T, y)

    return x_regularizado

A = np.zeros((len(POINTS), ORDER + 1))
for i in range(len(POINTS)):
    for j in range(ORDER + 1):
        A[i][j] = POINTS[i][0]**j

b = np.array([point[1] for point in POINTS])
x1_regularizado = solver(A, b, 1, ORDER)
x2_regularizado = solver(A, b, 1e6, ORDER)
print(x_regularizado)
```

Listing 6: Python - Solução LDL^T

Após a resolução do sistema linear com regularização, obtivemos os coeficientes $\mathbf{w}$ para dois valores de λ: $\lambda = 10^6$ e $\lambda = 1$. A tabela a seguir mostra os coeficientes w_i obtidos para cada valor de λ.

Índice i	$w_i(\lambda = 10^6)$	$w_i(\lambda = 1)$
0	$1.34505281 \times 10^{-6}$	$4.12287279 \times 10^{-6}$
1	$3.92951125 \times 10^{-6}$	$1.20435014 \times 10^{-5}$
2	$1.22044597 \times 10^{-5}$	$3.74113504 \times 10^{-5}$
3	$3.37475797 \times 10^{-5}$	$1.03416627 \times 10^{-4}$
4	$1.10068549 \times 10^{-4}$	$3.37444617 \times 10^{-4}$
5	$2.47554288 \times 10^{-4}$	$7.58113782 \times 10^{-4}$
6	$8.86263492 \times 10^{-4}$	$2.71771853 \times 10^{-3}$
7	$5.31941626 \times 10^{-5}$	$1.42214668 \times 10^{-4}$
8	$-2.55881297 \times 10^{-5}$	$-7.72699662 \times 10^{-5}$

Tabela 4: Valores dos coeficientes $\mathbf{w}$ para $\lambda = 10^6$ e $\lambda = 1$

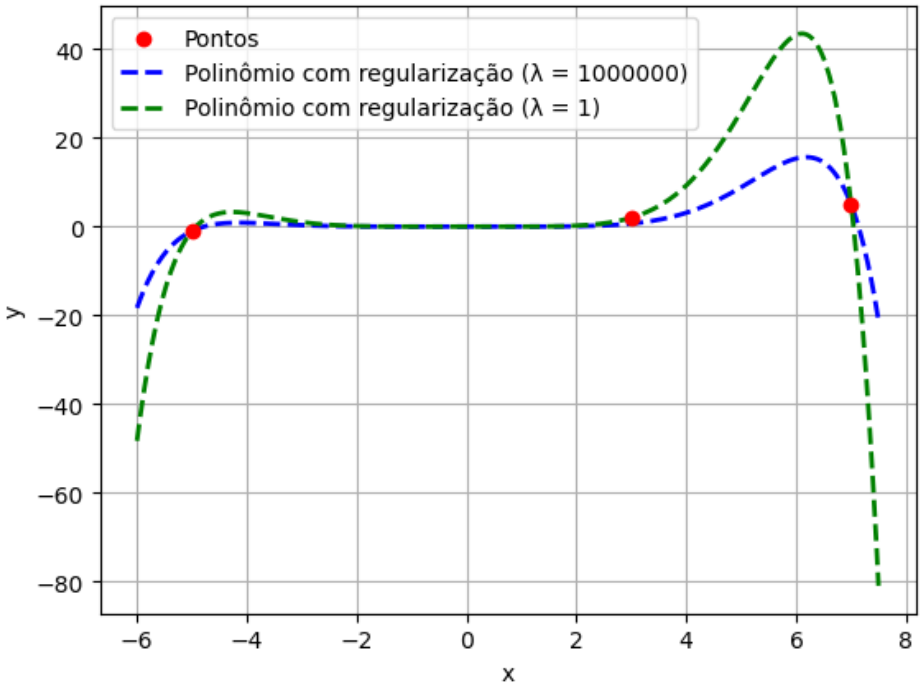

Figura 18: Interpolação - Polinômio de 8ª Ordem

5.2 Minimização Usando Gradiente Residual

Também podemos resolver essa questão usando o método iterativo do gradiente residual, que consiste em, a cada iteração, de acordo com o resíduo do sistema, incrementar as variáveis de solução até que o resíduo seja pequeno o suficiente. Este método é eficiente quando a solução direta do sistema linear é computacionalmente cara ou impraticável.

Definimos o resíduo $\mathbf{r}$ como a diferença entre o lado esquerdo e o lado direito da equação $\mathbf{Ax} = \mathbf{b}$, ou seja:

$$\mathbf{r} = \mathbf{br} - \mathbf{Arx}$$

Onde $Ar = A^TA + I\lambda$ e $br = A^TAb$, apresentados anteriormente no sitema com regularização.

A cada iteração, o vetor de solução $\mathbf{x}$ é atualizado pela fórmula:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

onde $\mathbf{p}_k$ é o vetor direcional de busca, calculado com base no gradiente do resíduo e α_k é o passo de atualização, dado pela fórmula:

$$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}$$

Esse processo continua até que o resíduo seja suficientemente pequeno, ou seja, $\|\mathbf{r}_k\|$ atinja um valor desejado, indicando que a solução $\mathbf{x}$ foi encontrada com a precisão desejada.

```
A = np.zeros((len(POINTS), ORDER + 1))
for i in range(len(POINTS)):
    for j in range(ORDER + 1):
        A[i][j] = POINTS[i][0]**j

b = np.array([point[1] for point in POINTS])

def residual_gradient(A_, b_, x0, lambda_value, tol=1e-6,
   max_iter=1000):
    A = A_.T @ A_ + lambda_value * np.eye(ORDER + 1)
    b = A_.T @ b_

    x = x0
    r = b - np.dot(A, x)
    p = r
    k = 0

    for _ in range(max_iter):
        alpha_k = np.dot(r.T, r) / np.dot(np.dot(p.T, A), p)
        x = x + alpha_k * p
        r_new = r - alpha_k * np.dot(A, p)
        beta_k = np.dot(r_new.T, r_new) / np.dot(r.T, r)
        p = r_new + beta_k * p
        r = r_new
```

```
        k += 1

        if np.linalg.norm(r) < tol:
            print(f'Convergiu em {k} itera  es')
            break

    return x

x1_regularizado = residual_gradient(A, b, [0, 1, 2, 3, 4, 5, 6,
   7, 8], 1)
x2_regularizado = residual_gradient(A, b, [0, 1, 2, 3, 4, 5, 6,
   7, 8], 1000000)
```

Listing 7: Python - Solução LDL^T

No nosso caso, o método convergiu após 10 iterações para $\lambda = 10^5$ e após 16 iterações para $\lambda = 1$, para os valores:

Índice i	$w_i(\lambda = 10^5)$	$w_i(\lambda = 1)$
0	$1.34505282 \times 10^{-6}$	$1.58324477 \times 10^{-6}$
1	$3.92951113 \times 10^{-6}$	$3.95525090 \times 10^{-6}$
2	$1.22044614 \times 10^{-5}$	$1.63759881 \times 10^{-5}$
3	$3.37475746 \times 10^{-5}$	$2.77898875 \times 10^{-5}$
4	$1.10068564 \times 10^{-4}$	$1.80761394 \times 10^{-4}$
5	$2.47554161 \times 10^{-4}$	$8.55803541 \times 10^{-5}$
6	$8.86263564 \times 10^{-4}$	$2.92607339 \times 10^{-3}$
7	$5.31941711 \times 10^{-5}$	$1.75071692 \times 10^{-4}$
8	$-2.55881320 \times 10^{-5}$	$-8.41852815 \times 10^{-5}$

Tabela 5: Valores dos coeficientes $\mathbf{w}$ para $\lambda = 10^5$ e $\lambda = 1$

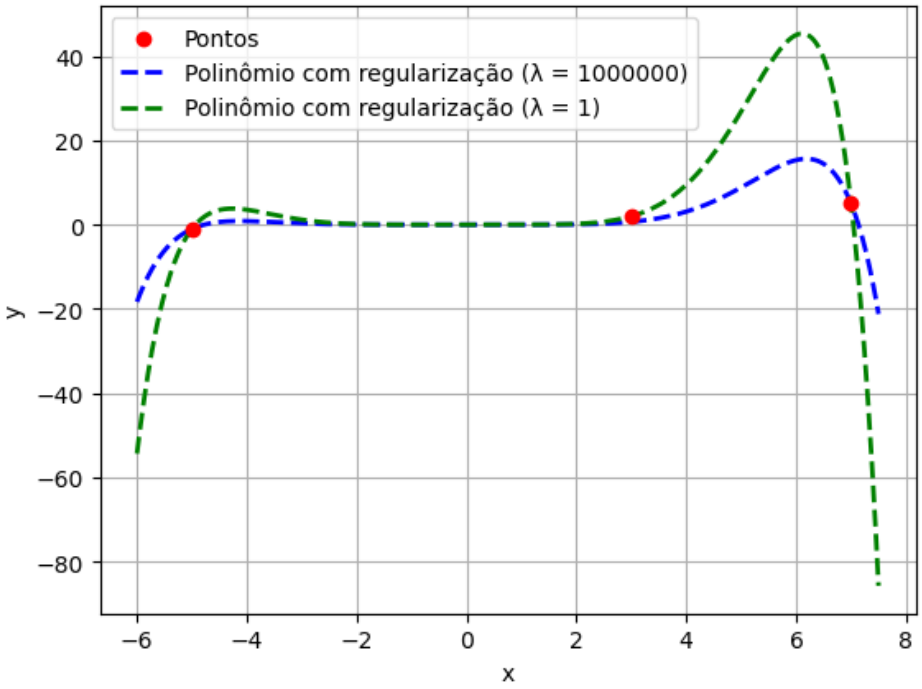

Figura 19: Interpolação com Resíduo Gradiente - Polinômio de 8ª Ordem

6 Problema 5: Mínimo de Função

Uma empresa tem o custo de fabricação de um produto dado por:

$$\bar{C}(q) = 0.0001q^2 - 0.08q + \frac{5000}{q} + 65$$

Nosso objetivo é encontrar o custo mínimo de produção, ou seja, o ponto ótimo, onde se tem o menor custo com a maior taxa de fabricação. Para isso, precisamos analisar a função e verificar se ela é convexa para a parte positiva de q, que corresponde à quantidade possível de fabricação, já que não se produzem quantidades negativas.

Para saber se a função é convexa na parte positiva, precisamos encontrar a segunda derivada. Caso ela seja positiva para todo o intervalo, a função é convexa.

Assim, as derivadas estão definidas como:

Primeira derivada:

$$\frac{d\bar{C}}{dq} = 0.0002q - 0.08 - \frac{5000}{q^2}$$

Segunda derivada:

$$\frac{d^2\bar{C}}{dq^2} = 0.0002 + \frac{10000}{q^3}$$

Logo, podemos ver que a segunda derivada é positiva para todo valor $q > 0$. Então sim, podemos achar o ponto ótimo nessa função para uma quantidade positiva do produto.

Para encontrar o ponto mínimo, implementaremos o método de Newton, utilizando a segunda derivada, que vai nos garantir que, a cada iteração, iremos em direção a um mínimo local, verificando a concavidade da função.

Então, a função para iterar será:

$$q_{k+1} = q_k - \frac{f'(q_k)}{f''(q_k)}$$

onde iniciaremos com $q_k = q_0 = 50$.

Implementamos em Python como:

```
ALPHA = 0.0001
BETA = -0.08
LAMBDA = 5000
GAMA = 65

def f(q):
    return ALPHA*q**2 + BETA*q + LAMBDA/q + GAMA

def f_d1(q):
    return 2*ALPHA*q + BETA - LAMBDA/(q**2)

def f_d2(q):
    return 2*ALPHA + 2*LAMBDA/(q**3)

def newton_method(q0):
    # init the values
```

```
    last_q  = q0
    q       = q0
    counter = 1

    for _ in range(1000):
        # calculate the new value
        q = last_q - f_d1(last_q)/f_d2(last_q)

        # verify if the solution is close enough
        if abs(q - last_q) < 1e-6:
            break

        # update the last value
        last_q = q
        counter += 1

    return q, counter
```

Listing 8: Python - Método de Newton

Ao executar o teste, convergimos em 10 iterações para $q = 500$, onde os primeiros 4 valores foram $[75.81, 116.18, 181.14, 285.35]$. Podemos ver no gráfico abaixo que realmente esse valor se situa no mínimo da função:

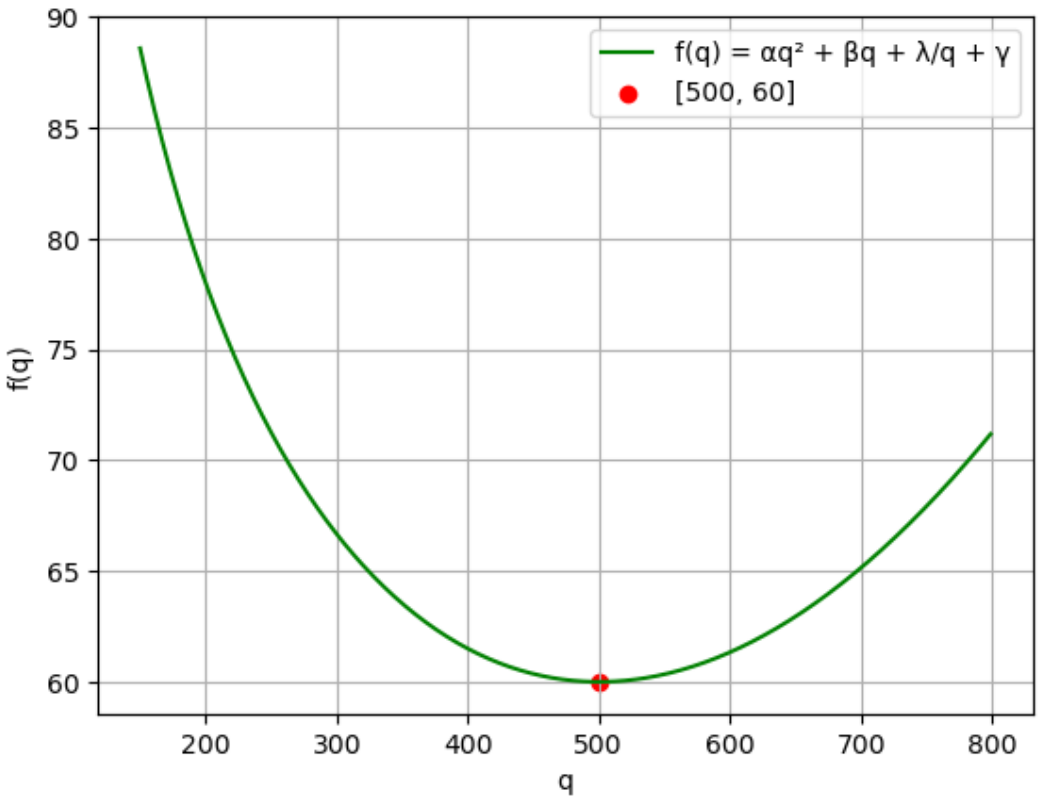

Figura 20: Custo Produto

7 Conclusão

Os métodos numéricos aplicados neste trabalho mostraram-se ferramentas poderosas e versáteis para resolver problemas complexos. Os métodos de integração evidenciaram como diferentes abordagens podem equilibrar precisão e eficiência computacional, enquanto a resolução de EDPs destacou o impacto da discretização e das condições de contorno nos resultados obtidos. Para sistemas lineares e interpolação, as técnicas de regularização, como a fatoração de Cholesky, e os métodos iterativos demonstraram ser soluções robustas para problemas mal condicionados ou com restrições específicas. Por fim, a otimização via método de Newton reforçou a importância de métodos baseados em derivadas para encontrar soluções rápidas e precisas em funções convexas.