UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO
INE5202 - CÁLCULO NÚMERICO EM COMPUTADORES

LEONARDO DE SOUSA MARQUES

RELATÓRIO I: DISTRIBUIÇÃO DE TEMPERATURA EM UMA PLACA

Docente:

Drª. Priscila Cardoso Calegari

Florianópolis
2024

Sumário

Lista de tabelas

1 INTRODUÇÃO

O presente relatório tem como objetivo apresentar soluções aproximadas para o problema de distribuição de temperatura em uma placa quadrada, utilizando o **Método SOR** (*Successive Over Relaxation*), ou Relaxações Sucessivas. Este método é uma técnica iterativa que aprimora o método de Gauss-Seidel ao introduzir um parâmetro de relaxação, ω, que controla a taxa de convergência do processo iterativo. Ajustando adequadamente ω, é possível acelerar significativamente a convergência, tornando o método uma escolha eficiente para resolver sistemas lineares derivados de problemas físicos, como a distribuição de temperatura.

No método SOR, a atualização de cada elemento x_i na solução é realizada de acordo com a seguinte fórmula de relaxação:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \tag{1.1}$$

onde $x_i^{(k)}$ representa o valor de x_i na iteração anterior k, e $x_i^{(k+1)}$ é o valor atualizado. O termo de relaxação ω, tal que $1 < \omega < 2$, permite que o método ajuste o valor de x_i de maneira mais rápida em direção à solução, acelerando a convergência do processo iterativo.

Como objeto de estudo, consideramos uma placa quadrada de lado N sujeita a condições de contorno fixas. Representamos essa placa em uma malha discreta com pontos de temperatura organizados em uma matriz T_{ij}, onde $0 \leq i, j \leq N$. Essa matriz possui as seguintes características de contorno: a extremidade direita da placa, correspondente à última coluna da matriz (T_{iN}), é mantida a uma temperatura constante de 50°C. Já as extremidades esquerda, superior e inferior da placa são mantidas a uma temperatura de 0°C, representadas pelas linhas e colunas T_{0j}, T_{Nj}, e T_{i0}.

Essas condições de contorno estabelecem um ambiente em que o calor flui do lado direito para o restante da placa, enquanto as outras bordas permanecem isoladas termicamente a 0°C.

Para calcular a temperatura nos pontos internos da placa, utilizamos o modelo discreto derivado da Equação de Poisson, onde a temperatura T_{ij} em um ponto é dada pela média aritmética das temperaturas de seus quatro vizinhos:

$$T_{ij} = \frac{T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}}{4}, \quad 1 \leq i, j \leq N-1 \tag{1.2}$$

Assim, para cada ponto interno T_{ij}, calculamos a temperatura em função dos pontos vizinhos, assumindo que as bordas da matriz T já foram fixadas pelas condições de contorno.

A matriz T, representada para um exemplo de $N = 4$, fica com a seguinte estrutura:

$$T = \begin{bmatrix} 0 & 0 & 0 & 0 & 50 \\ 0 & T_{11} & T_{12} & T_{13} & 50 \\ 0 & T_{21} & T_{22} & T_{23} & 50 \\ 0 & T_{31} & T_{32} & T_{33} & 50 \\ 0 & 0 & 0 & 0 & 50 \end{bmatrix}$$

Essa estrutura mostra que a matriz é esparsa e possui valores apenas nas posições de contorno e nas células internas calculadas pelo método iterativo.

2 IMPLEMENTAÇÃO DO MODELO

Para implementar o código, vamos utilizar a linguagem de programação `Python`, com auxílio das bibliotecas `numpy`, para realizar as operações de matrizes e vetores, `matplotlib`, para gerar os gráficos, e `pandas`, para organizar as informações obtidas em tabelas.

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
```

O código fonte será formado por três funções principais: `SOR(N, ω)`, `graph(N_values, df)` e a função `main()`. Vamos explicitar cada uma delas.

2.1 Função SOR

Na implementação do método SOR utilizada neste relatório, fazemos uma adaptação em relação à fórmula original. Na formulação clássica, o método SOR é aplicado para resolver sistemas lineares genéricos, onde as atualizações das variáveis são armazenadas em um vetor separado. No entanto, neste caso específico, estamos trabalhando com uma distribuição de temperatura em uma placa e, portanto, representamos a matriz de coeficientes e o vetor solução diretamente em uma matriz.

Essa adaptação permite que o método seja aplicado de forma mais direta e intuitiva, atualizando os valores diretamente em T, sem a necessidade de criar vetores separados. Assim, cada elemento é atualizado iterativamente, utilizando os valores calculados em tempo real dos pontos vizinhos. Esse armazenamento direto na matriz economiza memória e simplifica a estrutura do código.

Com isso, podemos criar o seguinte bloco de código.

```python
def SOR(N, w):
    # Inicializa a matriz T com N+1 zeros
    T = np.zeros((N+1, N+1))
    # Aplica a condicao de contorno
    for i in range(N+1):
        T[i][N] = 50  # Uma extremidade esta a 50 graus

    # Definicoes de controle para iteracao
    num_iteracoes = 0
    itmax = 1.0e4
    residuo = 1
    tolerancia = 1.0e-4

    # Iteracao do metodo de SOR
    while (residuo > tolerancia and num_iteracoes < itmax):
        # Copia a matriz antiga para calculo do erro
        T_old = T.copy()

        for i in range(1, N):
            for j in range(1, N):
                # Aplicacao do metodo SOR modificado
                soma_t = T[i-1][j] + T[i][j-1]
                         + T[i+1][j] + T[i][j+1]
                T[i][j] = (1 - w) * T[i][j] + (w / 4) * soma_t

        # Calcula a diferenca maxima entre a matriz nova e antiga
        residuo = np.max(np.abs(T - T_old))
        num_iteracoes += 1

    return T, num_iteracoes, residuo
```

2.2 Função Graph

Para podermos observar a mudança de temperatura de forma dinâmica, vamos criar gráficos com os valores de T e cores que variam de acordo com a temperatura. Como vamos rodar o código para diversos valores de N e ω, como veremos na função `main`, vamos gerar os gráficos com o valor ótimo para o parâmetro de relaxação encontrado no intervalo testado. Portanto, optou-se por passar como parâmetro para a função os valores de N e o dataframe com os dados, a fim de obter esse valor.

```python
def graph(N_values, df):
    plt.style.use('_mpl-gallery')

    for N in N_values:
        # Filtra o DataFrame para o valor atual de N
        df_N = df[df['N'] == N]

        # Encontra o w com o menor numero de iteracoes
        w_opt = df_N.loc[df_N['Numero de Iteracoes'].idxmin(), 'w']

        # Executa o metodo SOR com o w otimo
        T_opt, it, residuo = SOR(N, w_opt)

        # Plota e salva o grafico com um tamanho fixo de 6x6
           polegadas
        fig, ax = plt.subplots(figsize=(6, 6))
        im = ax.imshow(T_opt, cmap='jet', extent=[0, 1, 0, 1])
        ax.set_title(f'Distribuicao de Temperatura (N={N},
                                    w={w_opt:.1f})')
        fig.colorbar(im, ax=ax, label='Temperatura (C)')
        plt.savefig(f'distribuicao_temperatura_N{N}_w{w_opt}.png')
        plt.close(fig)

    return
```

2.3 Função Main

Na função `main()`, vamos definir os parâmetros necessários para a execução do método SOR e a análise dos resultados. Primeiramente, configuramos os valores de N e ω que serão testados: $N \in \{10, 20, 40, 80\}$ e $\omega \in \left\{ \frac{k}{10} \mid 1 \leq k < 20,\ k \in Z \right\}$. Para cada combinação desses valores, a função `SOR` será chamada, e os dados de convergência (número de iterações e resíduo) serão armazenados em um vetor `data`.

O objetivo principal da `main()` é identificar o valor ótimo de ω para cada N, ou seja, o valor de ω que minimiza o número de iterações necessárias para atingir a convergência. Ao final das execuções, os resultados são organizados em um `DataFrame`, facilitando a análise e o acesso aos dados.

Por último, a função `main()` chama a função `graph()`, que utiliza o valor ótimo de ω encontrado para cada N e gera gráficos de distribuição de temperatura na placa. Esses gráficos permitem uma visualização clara dos resultados e da eficiência do método para diferentes tamanhos de malha e parâmetros de relaxação.

```
def main():
    w_values = [k / 10 for k in range(1, 20)]  # Intervalo de w no
       intervalo [0.1, 1.9]
    n_values = [10, 20, 40, 80]  # Valores de N
    data = []  # Lista para armazenar os dados

    for N in n_values:
        for w in w_values:
            # Executa o metodo SOR e captura o numero de iteracoes
            T, num_iteracoes, residuo = SOR(N, w)
            data.append([N, w, num_iteracoes, residuo])  # Adiciona
                os dados na lista

    # Cria um DataFrame com os resultados
    df = pd.DataFrame(data,columns=['N','w','Iteracoes','Residuo'])
    # Salva o DataFrame em um arquivo CSV
    df.to_csv('comparacao_SORX.csv', index=False)
    # Gera os graficos para cada valor de N com o w otimo
    graph(n_values, df)
```

3 EXECUÇÃO DO CÓDIGO: RESULTADOS

Tendo ciência de todos os métodos que compõem o modelo, vamos executar o código partindo da função main. Com isso, para os valores de N e ω previstos, temos os seguintes resultados, apresentados separadamente para melhor compreensão dos dados.

Tabela 1 – Número de Iterações e Resíduo para $\omega = 0.1$

N	Número de Iterações	Resíduo
10	1349	9.9687×10^{-5}
20	4301	9.9906×10^{-5}
40	10000	2.5683×10^{-4}
80	10000	1.0146×10^{-3}

Tabela 2 – Número de Iterações e Resíduo para $\omega = 0.2$

N	Número de Iterações	Resíduo
10	709	9.9574×10^{-5}
20	2313	9.9897×10^{-5}
40	7210	9.9981×10^{-5}
80	10000	6.5276×10^{-4}

Tabela 3 – Número de Iterações e Resíduo para $\omega = 0.3$

N	Número de Iterações	Resíduo
10	474	9.9490×10^{-5}
20	1565	9.9690×10^{-5}
40	4969	9.9921×10^{-5}
80	10000	3.6543×10^{-4}

Tabela 4 – Número de Iterações e Resíduo para $\omega = 0.4$

N	Número de Iterações	Resíduo
10	350	9.7921×10^{-5}
20	1163	9.9441×10^{-5}
40	3736	9.9972×10^{-5}
80	10000	1.6666×10^{-4}

Tabela 5 – Número de Iterações e Resíduo para $\omega = 0.5$

N	Número de Iterações	Resíduo
10	272	9.6936×10^{-5}
20	908	9.9726×10^{-5}
40	2945	9.9834×10^{-5}
80	9059	9.9968×10^{-5}

Tabela 6 – Número de Iterações e Resíduo para $\omega = 0.6$

N	Número de Iterações	Resíduo
10	218	9.6116×10^{-5}
20	731	9.9623×10^{-5}
40	2388	9.9756×10^{-5}
80	7431	9.9935×10^{-5}

Tabela 7 – Número de Iterações e Resíduo para $\omega = 0.7$

N	Número de Iterações	Resíduo
10	178	9.6033×10^{-5}
20	600	9.9297×10^{-5}
40	1971	9.9829×10^{-5}
80	6193	9.9961×10^{-5}

Tabela 8 – Número de Iterações e Resíduo para $\omega = 0.8$

N	Número de Iterações	Resíduo
10	147	9.6516×10^{-5}
20	498	9.9663×10^{-5}
40	1646	9.9650×10^{-5}
80	5214	9.9904×10^{-5}

Tabela 9 – Número de Iterações e Resíduo para $\omega = 0.9$

N	Número de Iterações	Resíduo
10	122	9.8491×10^{-5}
20	417	9.8504×10^{-5}
40	1383	9.9793×10^{-5}
80	4414	9.9932×10^{-5}

Tabela 10 – Número de Iterações e Resíduo para $\omega = 1.0$

N	Número de Iterações	Resíduo
10	102	9.4977×10^{-5}
20	350	9.7734×10^{-5}
40	1166	9.9469×10^{-5}
80	3745	9.9919×10^{-5}

Tabela 11 – Número de Iterações e Resíduo para $\omega = 1.1$

N	Número de Iterações	Resíduo
10	85	9.3792×10^{-5}
20	293	9.8985×10^{-5}
40	982	9.9523×10^{-5}
80	3174	9.9911×10^{-5}

Tabela 12 – Número de Iterações e Resíduo para $\omega = 1.2$

N	Número de Iterações	Resíduo
10	70	9.6020×10^{-5}
20	245	9.7104×10^{-5}
40	824	9.9161×10^{-5}
80	2678	9.9938×10^{-5}

Tabela 13 – Número de Iterações e Resíduo para $\omega = 1.3$

N	Número de Iterações	Resíduo
10	57	9.2042×10^{-5}
20	202	9.9366×10^{-5}
40	685	9.9555×10^{-5}
80	2241	9.9906×10^{-5}

Tabela 14 – Número de Iterações e Resíduo para $\omega = 1.4$

N	Número de Iterações	Resíduo
10	45	8.3419×10^{-5}
20	165	9.5215×10^{-5}
40	562	9.9537×10^{-5}
80	1851	9.9677×10^{-5}

Tabela 15 – Número de Iterações e Resíduo para $\omega = 1.5$

N	Número de Iterações	Resíduo
10	32	8.5637×10^{-5}
20	130	9.9647×10^{-5}
40	451	9.9952×10^{-5}
80	1497	9.9759×10^{-5}

Tabela 16 – Número de Iterações e Resíduo para $\omega = 1.6$

N	Número de Iterações	Resíduo
10	30	6.3578×10^{-5}
20	99	8.9608×10^{-5}
40	350	9.9710×10^{-5}
80	1173	9.9493×10^{-5}

Tabela 17 – Número de Iterações e Resíduo para $\omega = 1.7$

N	Número de Iterações	Resíduo
10	41	8.0995×10^{-5}
20	65	9.8827×10^{-5}
40	256	9.7431×10^{-5}
80	870	9.9859×10^{-5}

Tabela 18 – Número de Iterações e Resíduo para $\omega = 1.8$

N	Número de Iterações	Resíduo
10	62	7.1689×10^{-5}
20	70	9.8384×10^{-5}
40	163	9.3851×10^{-5}
80	582	9.8982×10^{-5}

Tabela 19 – Número de Iterações e Resíduo para $\omega = 1.9$

N	Número de Iterações	Resíduo
10	124	9.5377×10^{-5}
20	126	9.8774×10^{-5}
40	155	9.5004×10^{-5}
80	286	9.8064×10^{-5}

Percebemos, no geral, que quanto maior N, maior é o número de iterações. Isso é de se esperar, visto que forma-se uma matriz de ordem $(N+1)^2$ e, como vimos no código da função `SOR`, temos um loop `for` aninhado, que aumenta a complexidade para $O(n^2)$, no mínimo, dependendo de quantas vezes o loop `while` é repetido para alcançar a convergência.

Observa-se que o ajuste adequado de ω reduz significativamente o número de iterações necessárias. Como exemplo, para $N = 80$ com $\omega = 0.1$, foram atingidas as 10.000 iterações máximas toleráveis, sugerindo que seriam necessárias muitas outras para alcançar a convergência. Em contrapartida, com $\omega = 1.9$, o número de iterações foi reduzido para apenas 286 – um número ainda considerável, mas que representa uma eficiência computacional notável para uma matriz de 6.400 células (80×80). Essa diferença resulta numa redução computacional de aproximadamente $\frac{10.000}{286} \approx 34.97$ vezes, indicando que o ajuste de ω não só acelera a convergência, mas é essencial para tornar o método SOR aplicável e eficiente em escalas maiores.

4 GRÁFICOS

Agora que já possuímos todos os dados necessários, vamos utilizar a função `graph`, em sequência como visto na função `main` para criarmos os gráficos que correspondem a distribuição de calor nas placas analisadas. No tópico de conclusão, faremos a análise dessas imagens.

4.1 Gráfico para $N = 10$ e $\omega = 1.6$

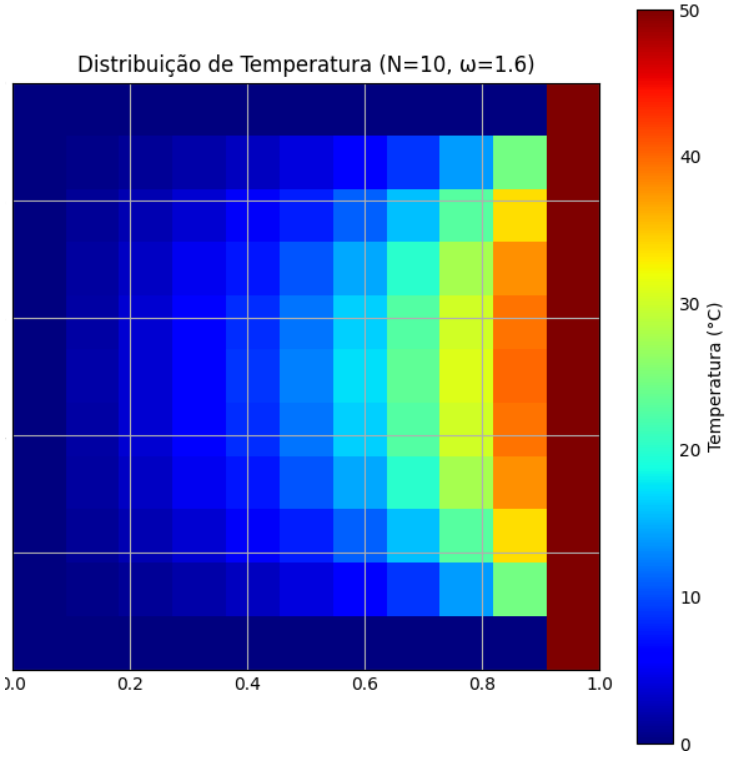

Figura 1

4.2 Gráfico para $N = 20$ e $\omega = 1.7$

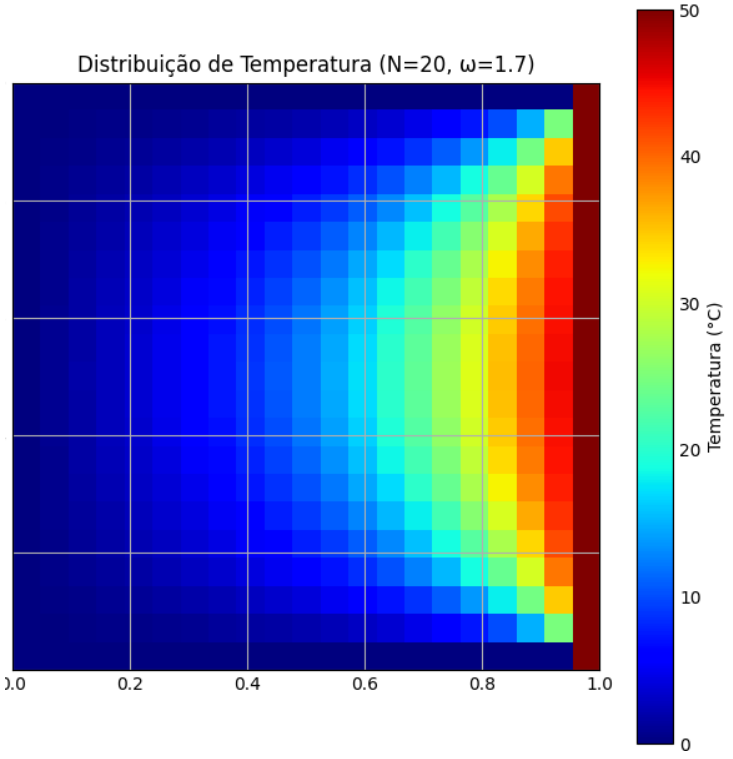

Figura 2

4.3 Gráfico para $N = 40$ e $\omega = 1.9$

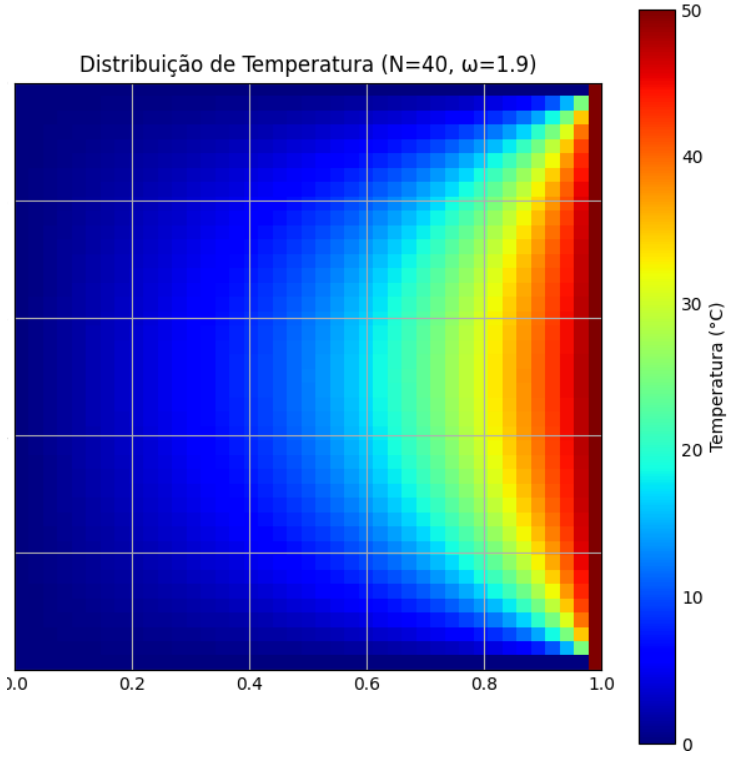

Figura 3

4.4 Gráfico para $N = 80$ e $\omega = 1.9$

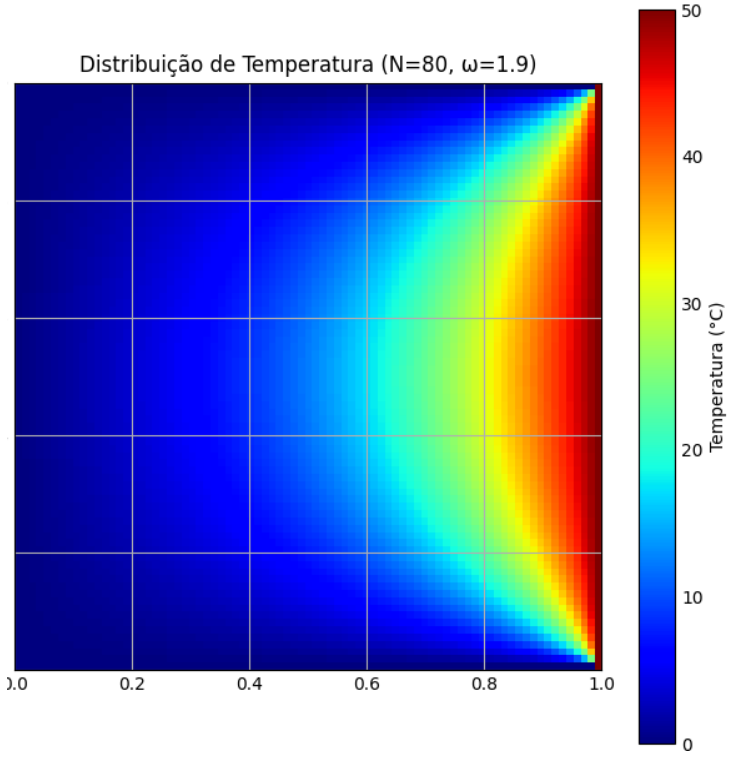

Figura 4

5 CONCLUSÃO

Podemos observar que, conforme N aumenta, temos uma resolução maior na visualização da transferência de calor de uma extremidade para a outra. Cada *pixel* na figura representa uma solução aproximada para a temperatura naquele ponto da placa, obtida através do Método SOR.

Essa distribuição de temperatura gera um gradiente visível, em que as regiões mais próximas da extremidade aquecida mostram valores mais altos de temperatura. À medida que N cresce, conseguimos observar esse gradiente com maior precisão, evidenciando a suavidade na transição das temperaturas, o que reflete a propagação térmica esperada para um meio com essas condições de contorno.

Essas visualizações mostram como o Método SOR consegue, de forma eficiente, aproximar a solução para cada ponto da malha, até que o sistema atinja a convergência dentro da tolerância especificada. Isso destaca a importância de escolhermos o valor adequado de ω. Neste experimento, buscamos valores para o parâmetro de relaxação dentro do intervalo [0.1, 1.9], mas é importante ressaltar que existe uma fórmula para realizar essa aproximação:

$$\omega_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \rho(T)^2}}$$

tal que $\rho(T)$ é o **raio espectral** da matriz de iteração T utilizada no método de Gauss-Seidel. O raio espectral é definido como o maior valor absoluto dos autovalores da matriz T.

6 REFERÊNCIAS

PETERS, S. & SZEREMETA, J.F. **Cálculo Numérico Computacional**. Editora da UFSC, 2018.