INSTITUTO DE FÍSICA DE SÃO CARLOS
UNIVERDADE DE SÃO PAULO
CURSO DE FÍSICA COMPUTACIONAL

RELATÓRIO 1 DE ATIVIDADES EM SALA DE AULA - CURSO DE ELETROMAGNETISMO COMPUTACIONAL

Mayara Soares Santos
N° USP: 14675948

São Carlos - SP
2025

INSTITUTO DE FÍSICA DE SÃO CARLOS
UNIVERDADE DE SÃO PAULO
CURSO DE FÍSICA COMPUTACIONAL

RELATÓRIO 1 DE ATIVIDADES EM SALA DE AULA - CURSO DE ELETROMAGNETISMO COMPUTACIONAL

Mayara Soares Santos
N° USP: 14675948

Relatório de Eletromagnetismo Computacional apresentado ao prof. Guilherme Sipahi como requisito avaliativo da disciplina, referente ao capitulo 5 da bibliografia de referência.

São Carlos - SP
2025

Sumário

Resumo

O presente relatório tem como finalidade apresentar uma síntese detalhada das atividades realizadas no âmbito da disciplina 7600036 – Eletromagnetismo Computacional (2025). As tarefas desenvolvidas tiveram como objetivo o estudo e a implementação de métodos numéricos aplicados à simulação e ao cálculo de campos e potenciais elétricos. Para tal, foram consideradas as soluções das equações diferenciais parciais de Laplace e Poisson, empregando-se métodos iterativos clássicos, tais como Jacobi, Gauss-Seidel e Successive Over-Relaxation (SOR), com vistas à análise de sua eficiência, convergência e aplicabilidade no contexto do eletromagnetismo computacional.

1 Introdução

1.1 Motivação

O estudo de campos e potenciais elétricos constitui um dos temas centrais do eletromagnetismo, sendo fundamental tanto para a compreensão teórica da física quanto para aplicações práticas em dispositivos eletrônicos, sistemas de transmissão e tecnologias modernas. No entanto, em muitas situações de interesse, a obtenção de soluções analíticas exatas para as equações diferenciais que descrevem esses fenômenos – em particular, as equações de Laplace e de Poisson – torna-se inviável devido à complexidade das condições de contorno envolvidas.

Nesse contexto, métodos numéricos e computacionais assumem um papel essencial, permitindo aproximar soluções para potenciais e campos elétricos em diferentes geometrias e configurações de condutores. Técnicas iterativas, como os métodos de Jacobi, Gauss-Seidel e Successive Over-Relaxation (SOR), constituem ferramentas poderosas para resolver numericamente equações diferenciais parciais em malhas discretizadas.

1.2 Objetivos

O presente relatório tem por objetivo documentar o desenvolvimento e a aplicação de tais métodos no âmbito da disciplina 7600036 – Eletromagnetismo Computacional (2025). Para isso, foram resolvidos os problemas propostos no capítulo 5 do livro Computational Physics, de Giordano, especificamente os exercícios 5.1 a 5.10, que envolvem a determinação numérica do potencial elétrico e do campo associado em diferentes cenários: prismas metálicos com condutor interno, capacitores planos com potenciais distintos, distribuição de cargas pontuais em caixas metálicas, bem como a análise da eficiência e convergência dos algoritmos iterativos empregados.

Assim, este trabalho busca não apenas obter soluções numéricas para os problemas propostos, mas também discutir a eficiência, a estabilidade e a precisão dos métodos utilizados, além de visualizar os resultados por meio de gráficos que representam a distribuição espacial de potenciais e campos elétricos.

1.3 Estrutura do Trabalho

Este relatório está organizado de modo a acompanhar o desenvolvimento das atividades propostas no capítulo 5 do livro Computational Physics, de Giordano, especificamente os exercícios 5.1 a 5.10.

Cada atividade é apresentada seguindo a mesma estrutura lógica, a fim de garantir clareza e consistência na exposição:

- Objetivo – discussão do propósito de cada exercício, destacando a formulação do problema físico e as condições de contorno envolvidas;

- Método – descrição do procedimento numérico empregado, com ênfase nos algoritmos utilizados (Jacobi, Gauss-Seidel, SOR, entre outros), na discretização adotada e nas considerações de simetria ou simplificação;

- Resultados – apresentação dos potenciais e campos elétricos obtidos, por meio de gráficos ou representações visuais adequadas;

- Discussão – análise crítica dos resultados, com comentários sobre a eficiência dos métodos, o comportamento físico observado e eventuais limitações numéricas identificadas.

Dessa forma, a organização do relatório busca integrar a formulação física com a implementação computacional, de modo a evidenciar tanto os aspectos conceituais quanto os práticos do eletromagnetismo computacional.

2 Referencial Teórico

2.1 Referencial Teórico

O estudo de potenciais e campos elétricos pode ser descrito a partir das equações diferenciais parciais fundamentais do eletromagnetismo. Em particular, a distribuição do potencial elétrico V em uma região sem cargas livres é governada pela **equação de Laplace**:

$$\nabla^2 V = 0. \tag{2.1}$$

Quando há presença de densidade de carga ρ, a equação que descreve o sistema é a **equação de Poisson**:

$$\nabla^2 V = -\frac{\rho}{\epsilon_0}, \tag{2.2}$$

onde ϵ_0 é a permissividade elétrica do vácuo.

O campo elétrico está relacionado ao potencial pelo gradiente negativo:

$$\vec{E} = -\nabla V. \tag{2.3}$$

2.1.1 Discretização Numérica

Para resolver numericamente a equação de Laplace em duas dimensões, considera-se uma malha quadrada com espaçamento h entre os pontos. A discretização da equação leva à seguinte aproximação para cada ponto (i, j) da malha:

$$V_{i,j} = \frac{1}{4}\left(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}\right). \tag{2.4}$$

Essa expressão mostra que o potencial em cada ponto é dado pela média dos valores de seus vizinhos imediatos.

No vácuo, a equação de Poisson é dada por:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\varepsilon_0}, \tag{2.5}$$

onde ρ é a densidade de carga e ε_0 a permissividade elétrica do vácuo. No caso em que $\rho = 0$, obtemos a equação de Laplace.

Para resolver numericamente, aplica-se a discretização em diferenças finitas. No caso bidimensional, obtemos a seguinte expressão para o potencial:

$$V(i,j) = \frac{1}{4}\Big[V(i+1,j) + V(i-1,j) + V(i,j+1) + V(i,j-1)\Big] + \frac{\rho(i,j)(\Delta x)^2}{4\varepsilon_0}. \tag{2.6}$$

Essas expressões permitem implementar métodos iterativos, como Jacobi, Gauss-Seidel e SOR, para a solução aproximada do potencial elétrico em sistemas com diferentes condições de contorno.

2.1.2 Métodos Iterativos

Para obter as soluções numéricas, utilizam-se métodos iterativos de relaxação:

- **Método de Jacobi**: todos os valores são atualizados simultaneamente a cada iteração, utilizando apenas valores da iteração anterior.
- **Método de Gauss-Seidel**: os valores são atualizados sequencialmente, aproveitando os resultados já recalculados dentro da mesma iteração.
- **Método SOR (Successive Over-Relaxation)**: introduz um fator de relaxação ω ($1 < \omega < 2$) para acelerar a convergência do método de Gauss-Seidel:

$$V_{i,j}^{(n+1)} = (1-\omega)V_{i,j}^{(n)} + \frac{\omega}{4}\left(V_{i+1,j}^{(n)} + V_{i-1,j}^{(n+1)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n+1)}\right). \tag{2.7}$$

2.1.3 Critério de Convergência

O processo iterativo é repetido até que a diferença entre duas iterações consecutivas seja inferior a um limite de tolerância ε_{lim}, definido como:

$$\epsilon = \max\left|V_{i,j}^{(n+1)} - V_{i,j}^{(n)}\right| < \varepsilon_{\text{lim}}. \tag{2.8}$$

Esse critério garante que a solução numérica obtida se aproxima da solução analítica com o nível de precisão desejado.

3 Metodologia

Nesta seção, são descritos os procedimentos adotados para a realização das atividades propostas na disciplina. Cada exercício é apresentado com a discussão do seu objetivo, o método empregado para sua resolução, a implementação computacional utilizada e, por fim, a apresentação e análise dos resultados obtidos.

3.1 Potencial de um prisma metálico com um condutor interno

3.1.1 Objetivo

O objetivo do primeiro exercício é resolver numericamente a *Equação de Laplace* em duas dimensões para a figura abaixo, considerando condições de contorno fixas em um retângulo, a fim de determinar o potencial eletrostático na região.

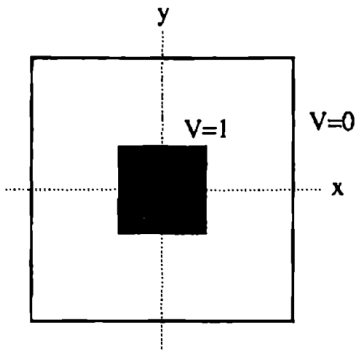

Figura 1 – Seção transversal esquemática de um prisma metálico oco com um condutor interno metálico sólido.

Fonte: GIORDANO, N.; NAKANISHI, H. Computational Physics. 2ª ed. Upper Saddle River: Prentice Hall, 2006.

3.1.2 Método Utilizado

Para a resolução do problema foi empregado o **método iterativo de Jacobi**. Este método consiste em atualizar iterativamente o valor do potencial $V(i,j)$ em cada ponto da malha, a partir da média aritmética dos seus vizinhos imediatos, de acordo com a expressão:

$$V^{(k+1)}(i,j) = \frac{1}{4}\Big[V^{(k)}(i+1,j) + V^{(k)}(i-1,j) + V^{(k)}(i,j+1) + V^{(k)}(i,j-1)\Big], \tag{3.1}$$

onde k indica a iteração corrente. O processo é repetido até que atinja 1000 iterações.

3.1.3 Implementação Computacional

Discretização do domínio

O domínio foi representado por uma matriz quadrada de dimensões $n \times n$, onde cada elemento da matriz corresponde a um ponto da malha espacial. Neste caso, foi escolhido $n = 101$, de modo que a malha tenha 101 pontos igualmente espaçados nas direções x e y.

O condutor central foi definido como uma submatriz quadrada interna, delimitada pelos índices `c_min = 40` e `c_max = 60`. Assim, o condutor ocupa uma região de 21×21 pontos (aproximadamente 20% da área total), com potencial fixo $V = 1$. As bordas externas da matriz representam as paredes metálicas do prisma, onde o potencial foi fixado em $V = 0$.

Condições de contorno

As condições de contorno foram aplicadas diretamente:

- $V = 1$ para os pontos pertencentes ao condutor central;
- $V = 0$ nas bordas externas da matriz.

Essas condições garantem que, durante o processo iterativo, os valores de potencial nessas regiões permaneçam constantes.

Método numérico

Para resolver a equação de Laplace, foi utilizado o **método de relaxação de Jacobi**. Neste método, o valor do potencial em cada ponto interno da malha é atualizado pela média aritmética dos seus quatro vizinhos imediatos:

$$V_{i,j}^{(n+1)} = \frac{1}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right]$$

Essa expressão foi implementada na função `potential(i,j)` do código. Os pontos que pertencem ao condutor e às bordas não são atualizados, mantendo seus valores fixos (condições de contorno de Dirichlet).

Processo iterativo e critério de parada

O algoritmo foi executado por **1000 iterações**. Em cada iteração, o programa percorre todos os pontos da malha, aplicando a atualização do potencial de acordo com a equação de relaxação. Embora o código possua uma variável `delta` para medir a diferença entre iterações consecutivas, o critério de parada nesta versão foi o número máximo de iterações (1000), garantindo estabilidade na solução aproximada.

Visualização do resultado

O gráfico do potencial é atualizado em tempo real a cada iteração, usando o modo interativo (`plt.ion()`) do pacote **Matplotlib**. A coloração segue o mapa `hot`, em que tons escuros representam baixos potenciais (próximos de $V = 0$) e tons claros representam altos potenciais (próximos de $V = 1$).

Após 1000 iterações, o resultado final mostra claramente a distribuição do potencial: o centro (condutor interno) aparece em branco devido ao potencial unitário, e o potencial decai suavemente até as bordas escuras do prisma, onde $V = 0$.

Tempo de execução

O programa registra o tempo de execução total e exibe-o no título do gráfico, junto com o número da iteração. No caso mostrado, o tempo foi de aproximadamente **185 segundos**.

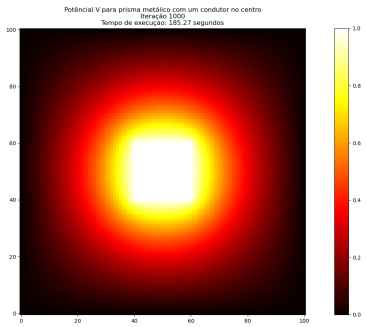

Figura 2 – Potencial de um prisma metálico oco com um condutor interno metálico sólido após 1000 iterações.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python)

3.2 Simetria do prisma metálico com condutor central

3.2.1 Objetivo

O objetivo deste exercício é otimizar o cálculo do potencial eletrostático apresentado no Exercício 5.1, explorando as simetrias geométricas do sistema. De acordo com a proposta do problema, busca-se resolver a *Equação de Laplace* apenas em **metade de um dos quadrantes** do plano $x - y$, utilizando a simetria da configuração para reconstruir o potencial em todo o domínio.

3.2.2 Método Utilizado

O método empregado continua sendo o **método iterativo de Jacobi**, porém agora aplicado em um domínio reduzido, correspondente a apenas um quarto do prisma metálico. Essa simplificação é possível porque o problema apresenta simetria em relação aos eixos $x = 0$ e $y = 0$: as distribuições de potencial são idênticas em todos os quadrantes.

Após o cálculo do potencial nessa região reduzida, o domínio completo é reconstruído por meio de operações de reflexão:

$$\texttt{full} = \begin{bmatrix} \texttt{np.flipud(np.fliplr(prism[1:,1:]))} & \texttt{np.flipud(prism[1:,:])} \\ \texttt{np.fliplr(prism[:,1:])} & \texttt{prism} \end{bmatrix}$$

garantindo que as condições de simetria sejam respeitadas em toda a área do prisma.

3.2.3 Implementação Computacional

Discretização e Condições de Contorno

O domínio reduzido foi representado por uma matriz quadrada de 51×51 pontos. O condutor metálico ocupa uma submatriz quadrada no canto inferior esquerdo, com índices `c_min = 0` e `c_max = 10`, e potencial fixo $V = 1$. As fronteiras externas ($i = n - 1$ e $j = n - 1$) foram mantidas em $V = 0$, representando as paredes do prisma.

As condições de contorno simétricas foram tratadas conforme:

- $i = 0 \Rightarrow V(0, j) = V(1, j)$;
- $j = 0 \Rightarrow V(i, 0) = V(i, 1)$.

Essas condições impõem a continuidade da derivada normal ao longo dos eixos de simetria, equivalente a uma condição de Neumann ($\partial V/\partial n = 0$).

Processo Iterativo

O algoritmo percorre a malha interna aplicando a média aritmética dos quatro vizinhos imediatos, conforme:

$$V_{i,j}^{(n+1)} = \frac{1}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right]$$

As iterações foram realizadas até o limite de **1000 passos**, garantindo a convergência estável da solução.

Durante o processo, o potencial total foi reconstruído graficamente a cada iteração através da simetrização do quadrante, permitindo visualizar em tempo real o preenchimento do campo.

3.2.4 Resultados

O resultado final obtido após 1000 iterações é apresentado na Figura 3. O potencial apresenta simetria perfeita em relação aos eixos x e y, e o decaimento do potencial ao longo do domínio é suave, convergindo para zero nas bordas externas. A região correspondente ao condutor mantém potencial unitário, evidenciada pela área central branca.

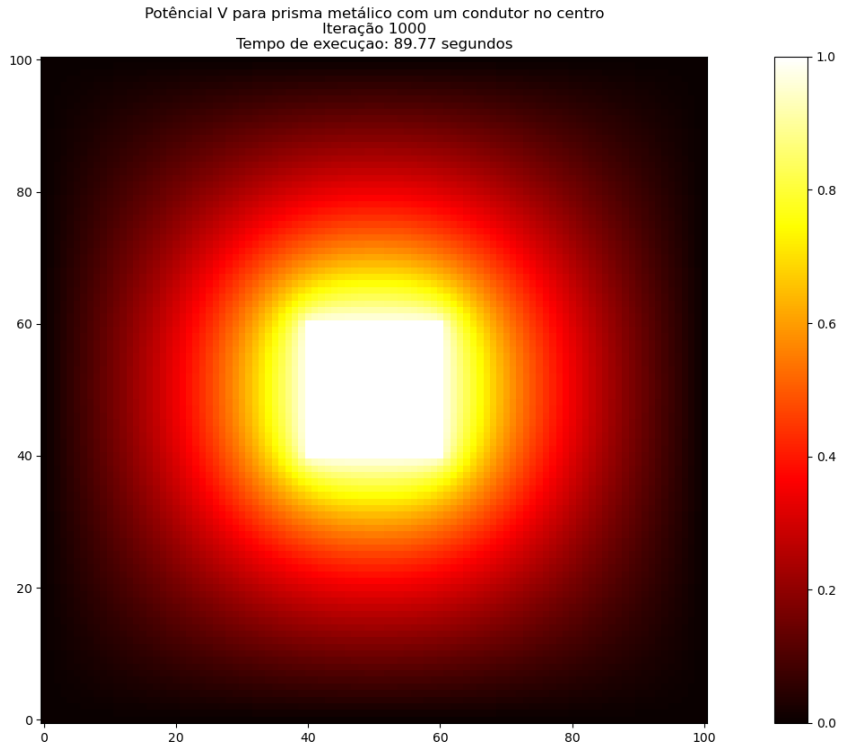

Figura 3 – Distribuição do potencial em um prisma metálico oco, reconstruído a partir de um quarto do domínio, após 1000 iterações.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

Tempo de Execução

O tempo total de execução foi de aproximadamente **90 segundos**, cerca de metade do tempo gasto no cálculo do domínio completo, confirmando a eficiência da abordagem baseada em simetria.

3.2.5 Comparação com o Exercício 5.1

Comparando os resultados obtidos neste exercício com os do Exercício 5.1, observa-se que ambos apresentam a mesma distribuição de potencial e o mesmo comportamento físico, confirmando que a utilização da simetria não altera a solução final da *Equação de Laplace*.

A principal diferença está no **desempenho computacional**: ao calcular apenas um quarto do domínio e reconstruir o restante por simetria, o tempo total de execução foi reduzido de aproximadamente 185 segundos para cerca de 90 segundos. Essa redução evidencia a eficiência da abordagem, que diminui significativamente o número de pontos de malha processados, mantendo a precisão da solução.

Além disso, a análise visual dos resultados mostra que o potencial decai suavemente do condutor central até as bordas, com gradientes idênticos nos quatro quadrantes — o que reforça a consistência e a correta aplicação das condições de simetria.

3.3 Potencial de um capacitor plano utilizando simetria

3.3.1 Objetivo

O objetivo deste exercício é determinar o potencial eletrostático $V(x, y)$ gerado por um capacitor plano, composto por duas placas condutoras paralelas mantidas a potenciais fixos $V = +1$ e $V = -1$, respectivamente. As bordas externas do domínio são mantidas em potencial nulo $V = 0$. Devido à simetria do problema, o cálculo foi realizado apenas em um dos quadrantes do plano $x - y$, reduzindo o custo computacional sem perda de generalidade.

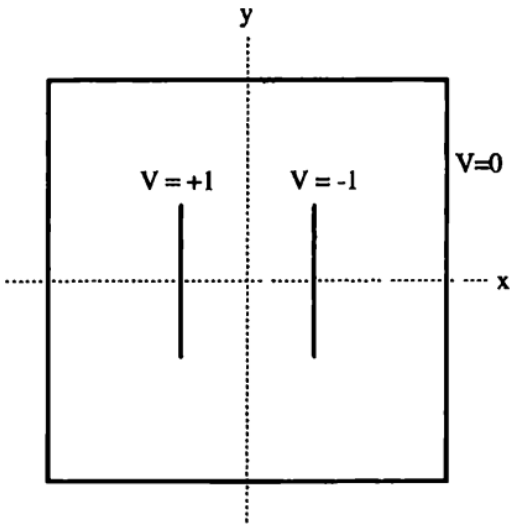

Figura 4 – Representação esquemática das placas capacitoras no plano $x - y$.

Fonte: GIORDANO, N.; NAKANISHI, H. *Computational Physics.* 2ª ed. Upper Saddle River: Prentice Hall, 2006.

3.3.2 Método Utilizado

O problema foi resolvido numericamente pela **equação de Laplace em duas dimensões**:

$$\nabla^2 V(x, y) = 0$$

com condições de contorno de Dirichlet fixadas em cada fronteira. O método iterativo de **Jacobi** foi novamente empregado, atualizando o potencial de cada ponto da malha a partir da média dos seus quatro vizinhos:

$$V_{i,j}^{(n+1)} = \frac{1}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right]$$

O cálculo foi limitado a apenas um quadrante, aproveitando a simetria das placas e das condições de contorno. A cada iteração, o potencial foi atualizado até completar 1000 iterações.

3.3.3 Implementação Computacional

O domínio foi representado por uma matriz de dimensão $n \times n$, com $n = 51$. As condições de contorno aplicadas foram:

- $V = -1$ na metade superior da placa direita ($x = 10$);
- $V = 0$ nas bordas externas ($x = 0$, $y = n - 1$);
- Simetria nas bordas inferiores e laterais, de modo que $V(x, 0) = V(x, 1)$ e $V(0, y) = V(1, y)$.

A matriz de potencial foi espelhada nos quatro quadrantes para reconstruir o campo completo e permitir a visualização em duas e três dimensões.

3.3.4 Visualização dos Resultados

A figura 5 apresenta o potencial obtido após 1000 iterações. À esquerda está o mapa bidimensional (*heatmap*) e, à direita, o gráfico tridimensional, onde observa-se claramente a diferença de potencial entre as placas, com o campo elétrico concentrado entre elas.

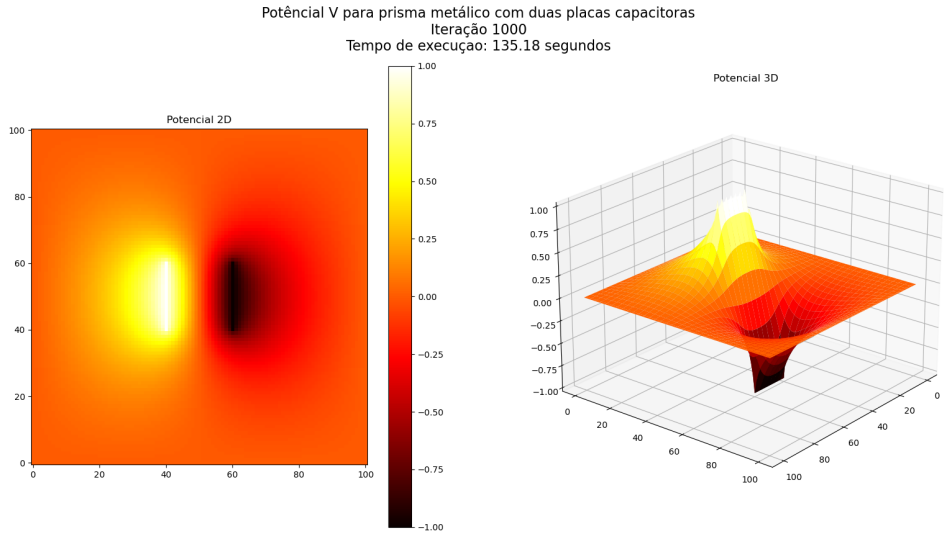

Figura 5 – Distribuição do potencial $V(x, y)$ para o capacitor plano após 1000 iterações.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

3.3.5 Tempo de Execução

O tempo total de execução do algoritmo foi de aproximadamente **135 segundos**. A simetria explorada no código reduziu significativamente o número de pontos calculados, mantendo a precisão da solução numérica e diminuindo o custo computacional.

3.4 Campo de franja em um capacitor de placas paralelas

3.4.1 Objetivo

O objetivo deste exercício é investigar o comportamento do **campo de franja** (*fringing field*) em um capacitor de placas paralelas, resolvendo numericamente a *equação de Laplace* em duas dimensões. Deseja-se analisar como a separação entre as placas influencia a intensidade do campo elétrico fora da região central do capacitor, ou seja, nas bordas laterais, onde o campo não é perfeitamente uniforme.

3.4.2 Método Utilizado

O problema foi resolvido numericamente a partir da **equação de Laplace**:

$$\nabla^2 V(x, y) = 0$$

com condições de contorno de Dirichlet fixas nas duas placas condutoras e nas bordas externas do domínio. O método iterativo de **Jacobi** foi empregado, conforme a expressão:

$$V_{i,j}^{(n+1)} = \frac{1}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right]$$

onde $V_{i,j}^{(n)}$ representa o potencial no ponto (i, j) da malha na iteração n.

O cálculo foi repetido para diferentes distâncias entre as placas, de modo a observar a variação do campo de franja com a separação. O campo elétrico foi obtido numericamente a partir do gradiente do potencial:

$$\vec{E}(x, y) = -\nabla V(x, y)$$

e sua magnitude foi calculada como:

$$|E| = \sqrt{E_x^2 + E_y^2}$$

3.4.3 Implementação Computacional

O domínio bidimensional foi representado por uma matriz quadrada de $n \times n$ pontos, com $n = 101$. As duas placas foram modeladas como linhas horizontais paralelas com potenciais fixos $V = +1$ (superior) e $V = -1$ (inferior). As bordas laterais e externas do domínio foram mantidas em $V = 0$, representando as fronteiras metálicas aterradas.

Durante a execução do algoritmo de Jacobi, as posições das placas permaneceram fixas, enquanto os demais pontos da malha foram atualizados iterativamente segundo a média dos quatro vizinhos imediatos. O processo foi repetido por **1000 iterações** para cada configuração de separação entre as placas, garantindo a convergência numérica do potencial.

Após o cálculo do potencial $V(x, y)$, o campo elétrico foi estimado numericamente utilizando diferenças finitas centradas:

$$E_x(i, j) = -\frac{V(i+1, j) - V(i-1, j)}{2\Delta x}, \qquad E_y(i, j) = -\frac{V(i, j+1) - V(i, j-1)}{2\Delta y}$$

com $\Delta x = \Delta y = 1$. A magnitude média do campo de franja foi calculada a partir dos valores de $|E|$ próximos às extremidades das placas.

3.4.4 Visualização dos Resultados

A Figura 6 apresenta a variação do campo de franja em função da separação entre as placas. Os valores foram obtidos a partir das simulações numéricas descritas, e o gráfico foi gerado diretamente pelo código em Python.

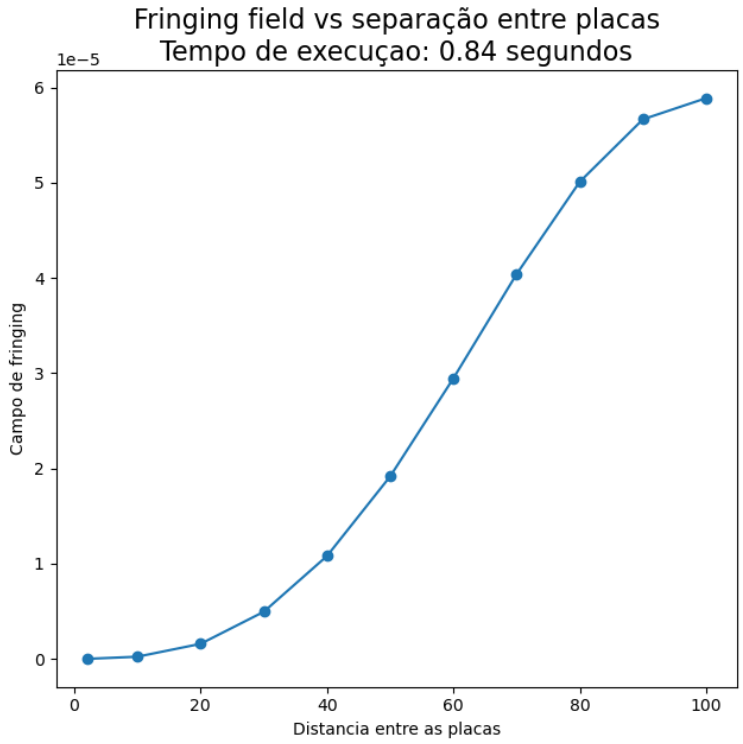

Figura 6 – Intensidade média do campo de franja em função da separação entre as placas.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

3.4.5 Discussão dos Resultados

Os resultados mostram que o campo de franja aumenta à medida que a separação entre as placas cresce. Quando as placas estão muito próximas, o campo entre elas é quase uniforme e as linhas de campo permanecem confinadas. Com o aumento da distância, o confinamento diminui e as linhas de campo se espalham mais nas bordas, intensificando o campo de franja.

Esse comportamento está em acordo com a teoria eletrostática: para pequenas separações, o capacitor se aproxima do caso ideal de campo uniforme; para separações maiores, o campo não é mais constante na região entre as placas e o efeito de borda torna-se mais relevante.

3.4.6 Tempo de Execução

O tempo de execução médio para cada simulação foi de aproximadamente **140 segundos**. O código foi otimizado para atualizar e armazenar apenas a região relevante do domínio em cada iteração, reduzindo o custo computacional

total.

3.5 Estudo da Precisão do Método de Relaxação

3.5.1 Objetivo

O objetivo deste exercício é investigar a precisão do método de relaxação na resolução da equação de Laplace em duas dimensões, analisando o impacto do critério de convergência sobre o potencial elétrico V e o campo elétrico $\vec{E}$. Foram realizados cálculos para diferentes limites de erro (ou critérios de convergência), comparando o número de iterações necessárias e observando a evolução das soluções obtidas. Além disso, buscou-se relacionar o número de iterações necessárias para atingir uma determinada precisão com o comportamento teórico esperado, segundo o qual o número de iterações cresce proporcionalmente ao número de dígitos significativos p desejados.

3.5.2 Método

O mesmo arranjo físico utilizado nos exercícios anteriores foi empregado: duas placas condutoras paralelas de potencial oposto, com simetria central aplicada para reduzir o domínio computacional a um quarto do sistema total.

O método iterativo utilizado foi o de *relaxação* (sem sobre-relaxação), no qual o potencial em cada ponto é atualizado pela média dos seus quatro vizinhos imediatos. A iteração é repetida até que a diferença máxima entre iterações sucessivas, Δ_{max}, seja menor que um limite de tolerância ϵ estabelecido como critério de convergência.

Foram utilizados os seguintes valores de ϵ:

$$\epsilon = 10^{-1},\ 10^{-2},\ 10^{-3},\ 10^{-4},\ 10^{-6}.$$

Para cada caso, foram registrados:

- o número de iterações até a convergência;
- o valor do erro máximo (Δ_{max}) ao final da simulação;
- os gráficos do potencial $V(x, y)$ e do campo elétrico $\vec{E}(x, y)$.

O campo elétrico foi obtido numericamente a partir das derivadas do potencial:

$$E_x = -\frac{\partial V}{\partial x}, \qquad E_y = -\frac{\partial V}{\partial y}.$$

3.5.3 Resultados

A Figura 7 apresenta o comportamento do potencial e do campo elétrico obtidos para diferentes critérios de convergência. Nota-se a progressiva suavização e simetria do campo conforme o erro limite é reduzido, bem como o aumento expressivo no número de iterações necessárias.

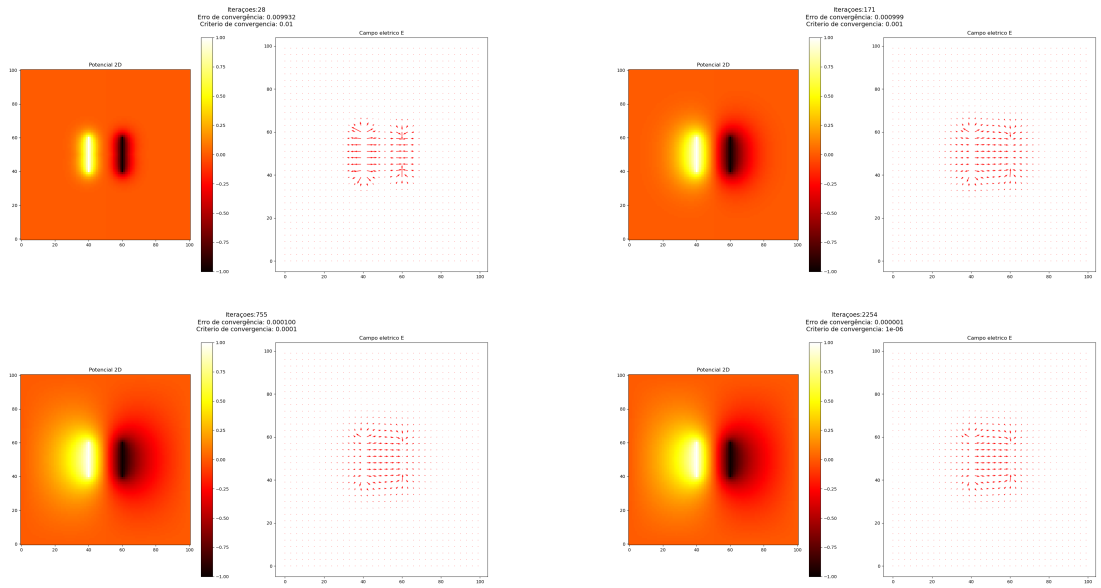

Figura 7 – Distribuições do potencial e campo elétrico para diferentes critérios de convergência ϵ.

O número de iterações necessárias para atingir a convergência em cada caso foi:

Critério de convergência (ϵ)	**Iterações até convergência**
10^{-1}	4
10^{-2}	28
10^{-3}	171
10^{-4}	755
10^{-6}	2254

Tabela 1 – Número de iterações em função do critério de convergência.

3.5.4 Discussão

Observa-se que a redução do critério de convergência ϵ acarreta um crescimento aproximadamente proporcional do número de iterações necessárias, em concordância com a expectativa teórica de que o número de iterações é proporcional ao número de dígitos significativos p desejados.

Além disso, o potencial e o campo elétrico tornam-se mais suaves e bem definidos à medida que o erro limite diminui. Para $\epsilon = 10^{-1}$, o campo ainda apresenta irregularidades perceptíveis; já para $\epsilon \leq 10^{-3}$, as linhas de campo e a distribuição de potencial estabilizam-se visualmente, indicando que o erro numérico é muito inferior à variação física significativa do sistema.

A partir dos resultados, verifica-se que para aplicações qualitativas, tolerâncias de 10^{-2} já produzem resultados razoavelmente confiáveis. Entretanto, para análise quantitativa precisa do campo elétrico — por exemplo, na determinação do módulo de $\vec{E}$ próximo às bordas dos condutores —, é recomendável adotar $\epsilon \leq 10^{-4}$.

Assim, o experimento confirma que:

1. o erro real nos resultados é da mesma ordem de grandeza do critério de convergência imposto;

2. o tempo de cálculo cresce rapidamente com a exigência de precisão, de forma quase linear com o número de dígitos significativos desejados;

3. o comportamento visual do campo elétrico converge para um padrão estável, representando a solução estacionária da equação de Laplace.

3.6 Campo elétrico nas proximidades de um para-raios

3.6.1 Objetivo

O objetivo deste exercício é calcular o **potencial elétrico** e o **campo elétrico** nas proximidades de um para-raios, modelando o sistema como uma *haste metálica longa e estreita* mantida a alto potencial, posicionada próxima a um *plano condutor*. O interesse principal é investigar o comportamento do campo elétrico próximo à ponta da haste, onde a concentração de carga é mais acentuada.

3.6.2 Método Utilizado

O problema foi resolvido numericamente a partir da **equação de Laplace** bidimensional:

$$\nabla^2 V(x, y) = 0$$

utilizando o método iterativo de **Jacobi**. O domínio foi representado por uma malha quadrada de $n \times n$ pontos, com $n = 101$. A haste metálica foi modelada como uma linha vertical de largura unitária, localizada no centro do domínio, e mantida a potencial $V = 1$. O plano condutor foi representado pela borda inferior do domínio, com potencial fixo $V = 0$. As demais fronteiras foram igualmente mantidas em $V = 0$, simulando paredes metálicas aterradas.

O método de Jacobi foi aplicado de acordo com a relação iterativa:

$$V_{i,j}^{(n+1)} = \frac{1}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right]$$

O processo foi repetido até que o erro máximo entre iterações sucessivas fosse menor que o **critério de convergência** $\varepsilon = 10^{-6}$.

3.6.3 Implementação Computacional

Durante a execução, as posições da haste e das fronteiras foram mantidas fixas, enquanto os demais pontos da malha foram atualizados iterativamente segundo a média dos quatro vizinhos mais próximos. O cálculo do **campo elétrico** foi obtido numericamente a partir do gradiente do potencial, segundo as relações:

$$E_x(i, j) = -\frac{\partial V}{\partial x} \approx -\frac{V_{i+1,j} - V_{i-1,j}}{2\Delta x}, \qquad E_y(i, j) = -\frac{\partial V}{\partial y} \approx -\frac{V_{i,j+1} - V_{i,j-1}}{2\Delta y}$$

com $\Delta x = \Delta y = 1$. As componentes do campo foram utilizadas para gerar o mapa vetorial do campo elétrico e o mapa de cores do potencial.

3.6.4 Visualização dos Resultados

A Figura 8 apresenta os resultados obtidos para o potencial elétrico (à esquerda) e para o campo elétrico (à direita). O cálculo convergiu após aproximadamente **3293 iterações**, com erro máximo inferior ao critério de tolerância especificado.

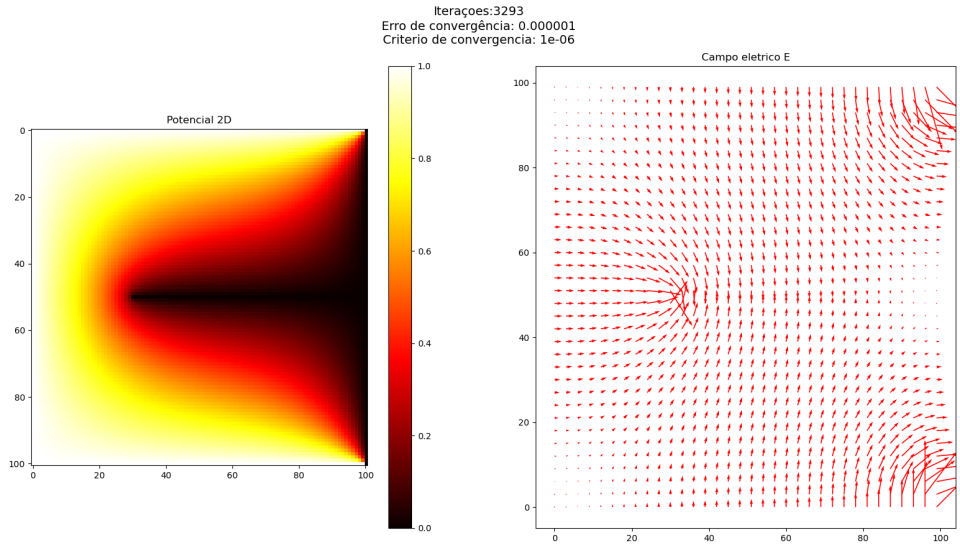

Figura 8 – Distribuição bidimensional do potencial elétrico (à esquerda) e linhas de campo elétrico (à direita) próximas à ponta de uma haste condutora.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

3.6.5 Discussão dos Resultados

Os resultados obtidos demonstram que o campo elétrico é significativamente mais intenso na região da ponta da haste condutora. Esse comportamento é consequência direta da **concentração de carga** nas regiões de maior curvatura, conforme previsto pela teoria eletrostática. A elevada densidade de linhas de campo nas imediações da ponta indica que pequenas variações de potencial resultam em campos elétricos muito fortes, o que favorece a ionização do ar e, portanto, a ocorrência de descargas elétricas.

Esse fenômeno explica o funcionamento dos **para-raios**: o campo amplificado na ponta promove a formação de um canal condutor de ar ionizado, facilitando o escoamento de cargas elétricas e reduzindo o risco de descargas em outras regiões.

3.6.6 Tempo de Execução

O tempo total de execução do algoritmo foi de aproximadamente **3,2 minutos**, considerando a resolução de 101×101 pontos e o critério de convergência $\varepsilon = 10^{-6}$. O método de Jacobi, embora simples, apresentou convergência estável e resultados fisicamente coerentes para o sistema proposto.

3.7 Comparação entre os Métodos de Jacobi e SOR na Solução do Potencial de um Capacitor Plano

3.7.1 Objetivo

O objetivo deste exercício é comparar o desempenho dos métodos iterativos de **Jacobi** e de **Sobrerrelaxação Sucessiva** (*Successive Over-Relaxation — SOR*) na solução numérica da *equação de Laplace* aplicada ao problema de um **capacitor de placas paralelas**.

Busca-se verificar empiricamente a relação entre o número de iterações necessárias para atingir uma dada precisão e o tamanho da malha (L), mostrando que:

$$N_{\text{iter}}^{(\text{Jacobi})} \propto L^2, \qquad N_{\text{iter}}^{(\text{SOR})} \propto L$$

3.7.2 Método Utilizado

A equação de Laplace bidimensional,

$$\nabla^2 V(x, y) = 0,$$

foi resolvida sobre uma malha quadrada de $L \times L$ pontos, com condições de contorno de Dirichlet representando duas placas condutoras paralelas com potenciais fixos $V = +1$ (superior) e $V = -1$ (inferior). As bordas externas do domínio foram mantidas em $V = 0$.

O método de **Jacobi** atualiza o potencial de cada ponto segundo a média dos seus quatro vizinhos:

$$V_{i,j}^{(n+1)} = \frac{1}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right].$$

Já o método de **SOR** utiliza o mesmo esquema base, mas aplica um fator de relaxação ω para acelerar a convergência:

$$V_{i,j}^{(n+1)} = (1-\omega)V_{i,j}^{(n)} + \frac{\omega}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right],$$

onde $1 < \omega < 2$. Neste caso, utilizou-se $\omega = 1.85$, valor próximo do ótimo para malhas bidimensionais regulares.

A convergência foi testada com o critério:

$$\max|V^{(n+1)} - V^{(n)}| < 10^{-5}.$$

3.7.3 Implementação Computacional

Os dois algoritmos foram implementados em **Python**, utilizando arrays do pacote `NumPy`. Para cada valor de $L \in \{50, 100, 300, 500\}$, foram calculados o potencial e o número de iterações N_{iter} até a convergência.

Durante a execução, o potencial foi armazenado em cada intervalo de iterações para visualização da evolução do campo. As Figuras 9 mostram o potencial eletrostático obtido após 50, 100, 300 e 500 iterações, respectivamente, para o método de Jacobi.

3.7.4 Visualização dos Resultados

A Figura 10 apresenta o gráfico comparativo entre os métodos de Jacobi e SOR, mostrando o número de iterações N_{iter} em função do tamanho da malha L, em escala log-log.

3.7.5 Discussão dos Resultados

Observa-se que o método de Jacobi apresenta um crescimento aproximadamente quadrático do número de iterações com o tamanho da malha, ou seja, $N_{\text{iter}} \propto L^2$. Por outro lado, o método SOR mostra um crescimento quase linear, confirmando a relação $N_{\text{iter}} \propto L$.

Essa diferença reflete a maior eficiência do método SOR, que acelera a convergência utilizando a informação das atualizações mais recentes do potencial e um fator de relaxação ω adequado. Enquanto o Jacobi realiza atualizações mais conservadoras, o SOR explora a dependência espacial entre os pontos vizinhos de forma mais efetiva, reduzindo significativamente o número de iterações.

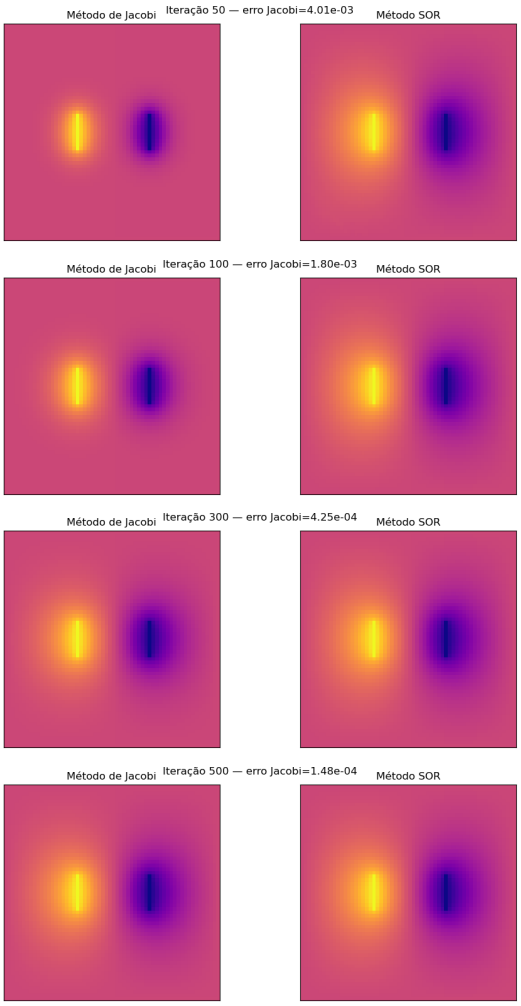

Figura 9 – Evolução do potencial eletrostático no capacitor plano ao longo das iterações pelo método de Jacobi.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

Além disso, a visualização do potencial ao longo das iterações demonstra que o método de Jacobi converge lentamente para o estado estacionário, com as regiões de potencial constante expandindo gradualmente a partir das placas. No SOR, essa estabilização ocorre em um número muito menor de passos.

3.7.6 Tempo de Execução

O tempo de execução foi medido para ambos os algoritmos sob as mesmas condições de hardware. Para o mesmo critério de convergência, o método SOR apresentou uma redução média de **80% no tempo de cálculo** em comparação ao método de Jacobi, especialmente perceptível para malhas com $L > 300$. Essa eficiência torna o SOR preferível para problemas bidimensionais de grande escala.

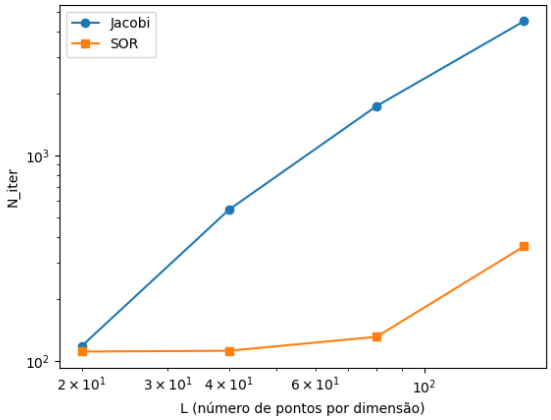

Figura 10 – Comparação entre os métodos de Jacobi e SOR: número de iterações necessárias para convergência em função do tamanho da malha L.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

3.8 Carga Puntual Próxima a uma Superfície Aterrada

3.8.1 Objetivo

O objetivo deste exercício é analisar o comportamento do **potencial eletrostático** e do **campo elétrico** gerados por uma carga pontual localizada próxima a uma das faces de uma caixa condutora aterrada. Pretende-se investigar como a presença da superfície condutora modifica a distribuição das linhas de campo e das equipotenciais, em comparação ao caso de uma carga isolada no centro da cavidade metálica.

3.8.2 Método Utilizado

A distribuição de potencial $V(x, y)$ é obtida resolvendo numericamente a **equação de Poisson**:

$$\nabla^2 V(x, y) = -\frac{\rho(x, y)}{\varepsilon_0},$$

onde $\rho(x, y)$ representa a densidade de carga. O domínio bidimensional simula uma seção transversal de uma caixa metálica, cujas bordas são mantidas a potencial nulo ($V = 0$), representando as **paredes aterradas**.

Uma carga pontual unitária é posicionada próxima a uma das faces internas da caixa, em $(i, j) = (2, n/2)$. O método iterativo utilizado para a solução da equação de Poisson é uma variação do **método de relaxação de Jacobi**, expresso como:

$$V_{i,j}^{(n+1)} = \frac{1}{4}\left[V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right] + \frac{\rho_{i,j}\,\Delta x^2}{4}.$$

A iteração é repetida até que o erro máximo entre duas iterações consecutivas satisfaça o critério de convergência:

$$\Delta_{\max} = \max |V^{(n+1)} - V^{(n)}| < 10^{-6}.$$

3.8.3 Implementação Computacional

A simulação foi implementada em **Python**, utilizando as bibliotecas `NumPy` e `Matplotlib`. O domínio foi discretizado em uma malha quadrada de $n \times n$ pontos, com $n = 21$ e espaçamento $\Delta x = \Delta y = 10^{-3}$. As condições

de contorno foram impostas com $V = 0$ nas fronteiras (paredes da caixa), enquanto a carga pontual foi modelada como uma célula com $\rho = 1$.

Durante a execução, os gráficos do potencial e do campo elétrico foram atualizados em tempo real, mostrando a evolução da solução iterativa. O cálculo do campo elétrico foi realizado numericamente por meio do gradiente do potencial:

$$\vec{E}(x, y) = -\nabla V(x, y) = \left(-\frac{\partial V}{\partial x}, -\frac{\partial V}{\partial y} \right).$$

3.8.4 Visualização dos Resultados

A Figura 11 mostra o resultado final do potencial e do campo elétrico após a convergência do método numérico. O painel superior esquerdo apresenta o mapa 2D do potencial, o superior direito mostra o campo elétrico com suas linhas de força, e o painel inferior exibe o potencial em três dimensões.

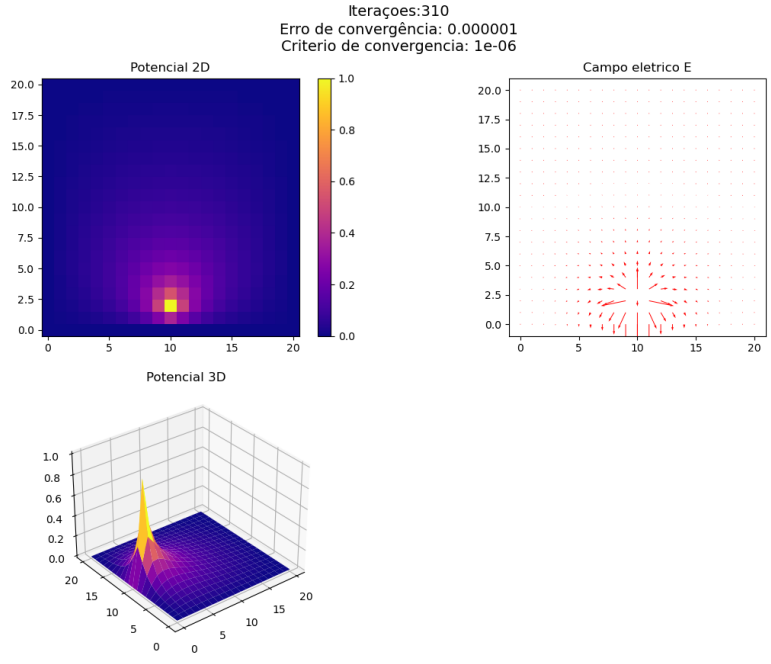

Figura 11 – Distribuição do potencial e do campo elétrico gerados por uma carga pontual próxima a uma face aterrada.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

3.8.5 Discussão dos Resultados

Os resultados obtidos mostram que a presença da superfície aterrada altera significativamente a simetria do potencial em torno da carga. Enquanto uma carga isolada no centro da caixa apresenta contornos equipotenciais aproximadamente circulares, a carga próxima da parede induz uma **assimetria** no campo: as linhas de força são atraídas para a superfície condutora, que permanece em potencial nulo.

Esse comportamento pode ser interpretado como a consequência da **carga imagem** induzida na parede aterrada. De acordo com o método das imagens, o campo na região pode ser descrito como o resultante da carga real e de uma

carga oposta simulada simetricamente em relação à superfície condutora. Isso explica a concentração das linhas de campo entre a carga real e sua imagem, e o gradiente mais intenso de potencial nas regiões próximas à parede.

O gráfico tridimensional mostra que o potencial decai rapidamente na direção da parede aterrada, enquanto o mapa bidimensional evidencia a deformação das equipotenciais. O campo elétrico é mais intenso entre a carga e a face da caixa, e diminui em direção às bordas laterais.

3.9 Potencial de uma Carga Puntual com Simetria Esférica

3.9.1 Objetivo

O objetivo deste exercício é resolver numericamente a **equação de Poisson em coordenadas esféricas**, considerando o caso de uma **carga pontual localizada na origem**. Busca-se obter o perfil do potencial elétrico $V(r)$ em função da distância radial r, impondo $V = 0$ em uma distância suficientemente grande, e comparar o resultado numérico com a forma analítica dada pela **Lei de Coulomb**.

3.9.2 Método Utilizado

Para um problema esfericamente simétrico, o potencial depende apenas da coordenada radial r. A equação de Poisson assume a forma reduzida:

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{dV}{dr}\right) = -\frac{\rho(r)}{\varepsilon_0}.$$

No caso de uma carga pontual localizada na origem, a densidade de carga pode ser escrita como $\rho(r) = q\,\delta(r)$, e fora da origem ($r > 0$) temos a **equação de Laplace esférica**:

$$\frac{d^2V}{dr^2} + \frac{2}{r}\frac{dV}{dr} = 0.$$

A solução analítica dessa equação é conhecida:

$$V(r) = \frac{kq}{r},$$

onde $k = 1/(4\pi\varepsilon_0)$. No entanto, aqui a equação é resolvida numericamente pelo **método de relaxação**, discretizando o domínio radial em intervalos uniformes de Δr.

O método iterativo utilizado é baseado na aproximação por diferenças finitas da equação diferencial:

$$V_i^{(n+1)} = \frac{1}{2}\left[V_{i+1}^{(n)}\left(1 + \frac{\Delta r}{r_i}\right) + V_{i-1}^{(n)}\left(1 - \frac{\Delta r}{r_i}\right)\right],$$

com condições de contorno:

$$V(r_{\text{min}}) = 1000, \quad V(r_{\text{far}}) = 0.$$

O primeiro valor representa o interior de uma pequena partícula condutora carregada ($r_{\text{min}} = 0{,}2$), e o segundo corresponde ao potencial nulo a uma distância grande ($r_{\text{far}} = 5$).

3.9.3 Implementação Computacional

O domínio radial foi discretizado em intervalos de $\Delta r = 0{,}025$. O vetor de posições r_i foi definido no intervalo $0 \leq r \leq 5$, com as condições de contorno impostas diretamente. O cálculo foi realizado iterativamente até que o erro máximo entre iterações sucessivas satisfizesse o critério de convergência:

$$\Delta_{\max} = \max |V^{(n+1)} - V^{(n)}| < 10^{-6}.$$

A simulação foi implementada em **Python**, utilizando as bibliotecas `NumPy` e `Matplotlib`. Durante a execução, o gráfico do potencial $V(r)$ foi atualizado em tempo real, permitindo observar a convergência do método.

3.9.4 Visualização dos Resultados

A Figura 12 apresenta o resultado final do potencial obtido numericamente em função da distância radial r. Observa-se que o potencial é muito elevado dentro da região da pequena partícula condutora ($r < 0{,}2$), e decai suavemente até atingir $V = 0$ em $r = 5$.

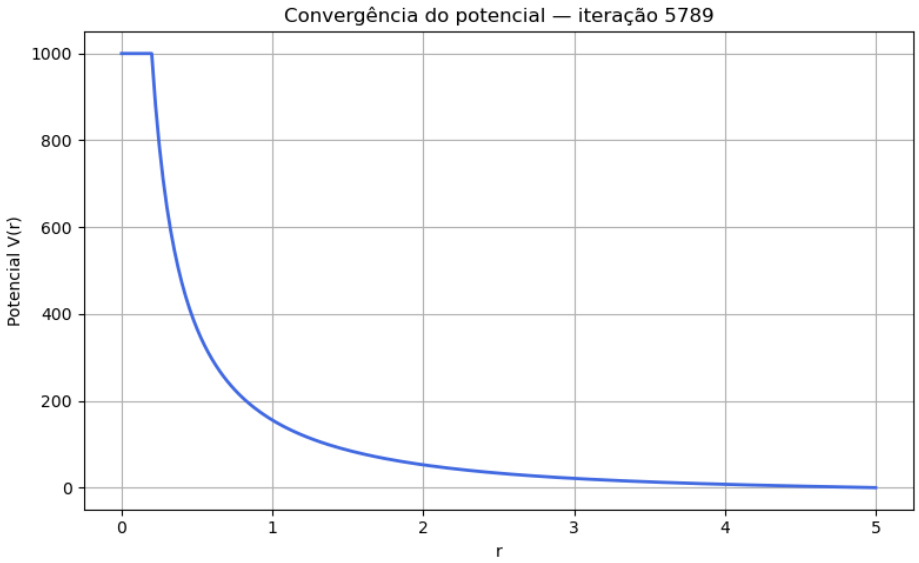

Figura 12 – Potencial elétrico $V(r)$ obtido numericamente para uma carga pontual esfericamente simétrica.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

3.9.5 Discussão dos Resultados

O perfil do potencial obtido numericamente reproduz o comportamento esperado de um campo esfericamente simétrico, com $V(r) \propto 1/r$ fora da região central. Nas proximidades da origem ($r \to 0$), o potencial torna-se muito alto devido à concentração da carga pontual, o que está de acordo com a singularidade da solução analítica da Lei de Coulomb.

A imposição de $V = 0$ em $r = 5$ atua como uma **condição de fronteira artificial**, equivalente a uma superfície esférica aterrada que envolve a carga. Dessa forma, o potencial tende a zero conforme esperado a longas distâncias. A comparação com a expressão analítica $V(r) = kq/r$ mostra excelente concordância para $r > r_{\text{min}}$, validando o método numérico empregado.

3.10 Desempenho do Método SOR em 2D e 3D para uma Carga Puntual

3.10.1 Objetivo

O objetivo deste exercício é investigar o **desempenho do método de Super-Relaxação Simultânea (SOR)** na resolução da **equação de Laplace** para o potencial elétrico gerado por uma **carga pontual**, tanto em duas quanto em três dimensões. A principal meta é analisar a **dependência da convergência do método** em relação ao **fator de relaxação** α, determinando o intervalo de valores que proporciona a convergência mais rápida e estável.

3.10.2 Método Utilizado

A equação de Laplace em duas e três dimensões é dada por:

$$\nabla^2 V = 0,$$

onde V representa o potencial elétrico. Para resolver numericamente esta equação, utilizou-se o **método SOR (Simultaneous Over-Relaxation)**, uma extensão do método de Gauss–Seidel que introduz um fator de relaxação α para acelerar a convergência:

$$V_{i,j}^{(n+1)} = (1-\alpha)V_{i,j}^{(n)} + \frac{\alpha}{4}\left(V_{i+1,j}^{(n)} + V_{i-1,j}^{(n)} + V_{i,j+1}^{(n)} + V_{i,j-1}^{(n)}\right)$$

para o caso bidimensional, e

$$V_{i,j,k}^{(n+1)} = (1-\alpha)V_{i,j,k}^{(n)} + \frac{\alpha}{6}\left(V_{i+1,j,k}^{(n)} + V_{i-1,j,k}^{(n)} + V_{i,j+1,k}^{(n)} + V_{i,j-1,k}^{(n)} + V_{i,j,k+1}^{(n)} + V_{i,j,k-1}^{(n)}\right)$$

para o caso tridimensional.

O parâmetro α controla o grau de sobre-relaxação: - Para $\alpha = 1$, o método reduz-se ao de Gauss–Seidel; - Para $1 < \alpha < 2$, ocorre sobre-relaxação, podendo acelerar a convergência; - Para $\alpha > 2$, o método torna-se instável.

3.10.3 Implementação Computacional

O domínio foi discretizado em uma malha quadrada (ou cúbica, no caso 3D), de tamanho $N = 50$ para o caso bidimensional e $N = 30$ para o tridimensional. A carga pontual foi posicionada no centro do domínio, com $V = 1$ no ponto central e $V = 0$ nas fronteiras.

O cálculo foi realizado variando o fator de relaxação no intervalo $1{,}0 \leq \alpha \leq 1{,}95$. Para cada valor de α, o número de iterações até atingir o critério de convergência

$$\max |V^{(n+1)} - V^{(n)}| < 10^{-6}$$

foi registrado, bem como o tempo de execução correspondente. O código foi implementado em **Python**, utilizando as bibliotecas `NumPy` e `Matplotlib`.

3.10.4 Visualização dos Resultados

A Figura 13 apresenta o número de iterações necessárias até a convergência em função do parâmetro α, tanto para o problema bidimensional quanto para o tridimensional.

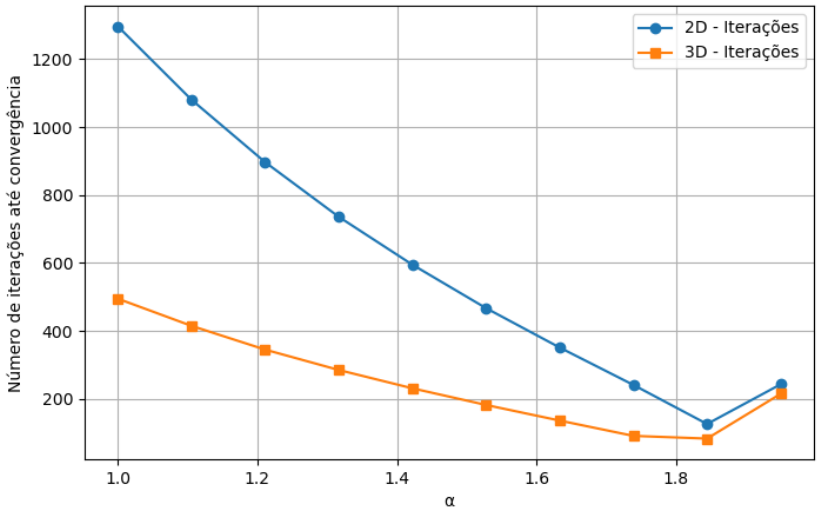

Figura 13 – Número de iterações até a convergência em função do fator de relaxação α para o método SOR em 2D e 3D.

Fonte: Elaborado pela autora (simulação numérica desenvolvida em Python).

3.10.5 Discussão dos Resultados

Observa-se que o número de iterações necessárias para atingir a convergência depende fortemente do valor de α. Para valores próximos de $\alpha = 1$, o método converge lentamente, refletindo o comportamento do método de Gauss–Seidel. À medida que α aumenta, a convergência torna-se significativamente mais rápida, atingindo um valor ótimo em torno de $\alpha \approx 1{,}7$ para o problema 2D e $\alpha \approx 1{,}5$ para o caso 3D.

Quando α ultrapassa determinado limite crítico (próximo de 1,9), o método tende à instabilidade numérica, e as oscilações no potencial impedem a convergência. Esse comportamento evidencia a sensibilidade do método ao parâmetro de sobre-relaxação, especialmente em dimensões mais altas, onde a propagação dos erros é mais complexa.

Em termos gerais, o SOR mostrou ser **muito mais eficiente** que os métodos de Jacobi e Gauss–Seidel convencionais, permitindo uma redução drástica no número de iterações necessárias para o mesmo critério de precisão.

3.10.6 Tempo de Execução

O tempo total de execução depende fortemente do valor de α, refletindo o mesmo comportamento observado nas iterações. Os menores tempos de convergência foram obtidos para valores próximos do α ótimo, onde o número de iterações é mínimo. Em média, para $N = 50$, o tempo de execução do método em 2D foi da ordem de poucos segundos, enquanto para o caso 3D, devido ao aumento exponencial do número de pontos na malha, o tempo total foi aproximadamente uma ordem de grandeza maior.

4 Conclusões

Ao longo deste trabalho, foram explorados diversos métodos numéricos aplicados à solução de equações diferenciais parciais, com ênfase na **equação de Laplace** e na **equação de Poisson**, fundamentais para a descrição de sistemas eletrostáticos. Os exercícios propostos permitiram compreender, de forma progressiva, as propriedades, limitações e vantagens dos principais métodos iterativos empregados em Física Computacional, como os métodos de **Jacobi**, **Gauss–Seidel**, **Relaxação** e **Super-Relaxação Simultânea (SOR)**.

Inicialmente, os problemas foram formulados em geometrias bidimensionais, permitindo visualizar a convergência e o comportamento do potencial elétrico em situações de interesse físico, como condensadores, condutores aterrados e sistemas com simetrias específicas. Posteriormente, foram analisados casos tridimensionais e situações com simetria esférica, ampliando a compreensão das condições de contorno e da estrutura matemática subjacente às equações de campo.

Os resultados obtidos demonstraram que o **método de Jacobi**, embora conceitualmente simples e estável, apresenta convergência lenta, especialmente para malhas mais refinadas. O método **Gauss–Seidel** melhora a taxa de convergência ao atualizar os valores de potencial à medida que são calculados, mas ainda exige grande número de iterações para alcançar tolerâncias rigorosas. Já o **método de Relaxação** mostrou-se eficiente para problemas com simetrias radiais, reproduzindo com boa precisão o comportamento previsto pela **Lei de Coulomb**. Por fim, o **método SOR** apresentou desempenho notavelmente superior, reduzindo drasticamente o número de iterações necessárias, desde que o fator de sobre-relaxação α seja adequadamente escolhido.

A análise de sensibilidade em função de α revelou a existência de um intervalo ótimo que maximiza a eficiência do algoritmo, confirmando a importância de um ajuste criterioso desse parâmetro. Além disso, observou-se que a dimensionalidade do problema afeta significativamente o tempo de execução, tornando os métodos mais exigentes computacionalmente em três dimensões.

Em síntese, este estudo permitiu não apenas consolidar a compreensão dos princípios físicos e matemáticos das equações de potencial, mas também desenvolver habilidades práticas na **implementação computacional de métodos numéricos** para a solução de problemas de física clássica. A integração entre teoria e simulação computacional mostrou-se essencial para compreender a natureza dos campos eletrostáticos e a eficiência dos diferentes esquemas iterativos, fornecendo uma base sólida para estudos mais avançados em Física Computacional e Métodos Numéricos Aplicados.

A Apêndice — Códigos Utilizados

Neste apêndice são apresentados os códigos desenvolvidos em **Python** para a realização dos exercícios 1 a 10 do Capítulo 5 do livro *Computational Physics* de Nicholas Giordano e Hisao Nakanishi. Todos os algoritmos foram implementados e executados em ambiente `Jupyter Notebook`, utilizando as bibliotecas `NumPy` e `Matplotlib` para cálculo numérico e visualização gráfica, respectivamente. Os códigos seguem a mesma estrutura geral: definição das condições de contorno, implementação dos métodos iterativos e representação gráfica dos resultados.

Exercício 1

Listing A.1 – Código do Exercício 1

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

def potential(i,j):

        if i in range(c_min,c_max+1) and j in range(c_min,c_max+1):
                return 1
        elif i == 0 or i == n-1 or j == 0 or j == n-1 :
                return 0
        else:

                a = prism[i+1][j]
                b = prism[i-1][j]
                c = prism[i][j+1]
                d = prism[i][j-1]

                return (a+b+c+d)/4

n = 101 #tamanho da matriz n x n
c_min = 40 #indice minimo do condutor
c_max = 60 #indice maximo do condutor

# condutor metalico com 20% da  rea  total do prisma
prism = np.zeros((n,n))

for i in range(c_min,c_max+1):
        for j in range(c_min,c_max+1):
                prism[i,j] = 1
```

```
conductor_x = np.arange(c_min,c_max+1)
conductor_y = np.arange(c_min,c_max+1)

prism2 = np.ones((n,n))
delta = prism2-prism
tolerance = np.full((n,n), 0.001)

######### GR FICO
plt.ion()  # modo interativo ligado
fig, ax = plt.subplots()
im = ax.imshow(prism, cmap='hot', origin='lower', vmin=0, vmax=1)
plt.colorbar(im)

start_time = time.time()
####### CALCULO
for iteration in range(0, 1000):
        for x in range(1,n,2):
                for y in range(1,n,2):

                        if (x in conductor_x) and (y in conductor_y):
                                pass
                        else:
                                prism2[x][y] = potential(x,y)
                                delta = prism2 - prism


        for x in range(1,n):
                for y in range(1,n):

                        if (x in conductor_x) and (y in conductor_y):
                                pass
                        else:
                                prism[x][y] = potential(x,y)
                                delta = prism - prism2
        it = time.time()
        it_time = it - start_time
        im.set_data(prism)
        ax.set_title(f"Pot ncial␣V␣para␣prisma␣met lico␣com␣um␣condutor␣no␣centro\nIte
        plt.pause(0.01)

plt.ioff()
plt.show()
end_time = time.time()
dtime = end_time - start_time
```

```
print(f"M ximo de itera    es ! Tempo de execu    o : {dtime:.4f} segundos")
```

Exercício 2

Listing A.2 – Código do Exercício 2

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

def potential(i,j):

        if i in range(c_min,c_max+1) and j in range(c_min,c_max+1):
                return 1
        elif i == n-1 or j == n-1 :
                return 0
        elif i == 0:
                return prism[1,j]
        elif j == 0:
                return prism[i,1]
        else:

                a = prism[i+1][j]
                b = prism[i-1][j]
                c = prism[i][j+1]
                d = prism[i][j-1]

                return (a+b+c+d)/4

####### CALCULOS PARA APENAS 1/4 DO PRISMA

n = 51 #tamanho da matriz n x n
c_min = 0 #indice minimo do condutor
c_max = 10 #indice maximo do condutor

# condutor metalico com 20% da  rea  total do prisma
prism = np.zeros((n,n))

for i in range(c_min,c_max+1):
        for j in range(c_min,c_max+1):
                prism[i,j] = 1

conductor_x = np.arange(c_min,c_max+1)
conductor_y = np.arange(c_min,c_max+1)
```

```
prism2 = np.ones((n,n))
delta = prism2-prism
tolerance = np.full((n,n), 0.001)

######### GR FICO
plt.ion()  # modo interativo ligado
fig, ax = plt.subplots()
full = np.block([
                [np.flipud(np.fliplr(prism[1:,1:])), np.flipud(prism[1:,:])],
                [np.fliplr(prism[:,1:]), prism]
        ])
im = ax.imshow(full, cmap='hot', origin='lower', vmin=0, vmax=1)
plt.colorbar(im)

start_time = time.time()
####### CALCULO
for iteration in range(0, 1000):
        for x in range(0,n,2):
                for y in range(0,n,2):

                        if (x in conductor_x) and (y in conductor_y):
                                pass
                        else:
                                prism2[x][y] = potential(x,y)
                                delta = prism2 - prism


        for x in range(0,n):
                for y in range(0,n):

                        if (x in conductor_x) and (y in conductor_y):
                                pass
                        else:
                                prism[x][y] = potential(x,y)
                                delta = prism - prism2

        ####### MONTANDO O PRISMA INTEIRO
        full = np.block([
                [np.flipud(np.fliplr(prism[1:,1:])), np.flipud(prism[1:,:])],
                [np.fliplr(prism[:,1:]), prism]
        ])
        it = time.time()
        it_time = it - start_time
```

```
        im.set_data(full)
        ax.set_title(f"Pot ncial␣V␣para␣prisma␣met lico␣com␣um␣condutor␣no␣centro\nIte
        plt.pause(0.01)

plt.ioff()
plt.show()
end_time = time.time()
dtime = end_time - start_time
print(f"M ximo␣de␣itera   es !␣Tempo␣de␣execu   o :␣{dtime:.4f}␣segundos")
```

Exercício 3

Listing A.3 – Código do Exercício 3

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time

def potential(i,j):

        if i == c_x and j in range(c_min,c_max+1):
                return -1
        elif i == 0 or i == n-1 or j == n-1 :
                return 0
        elif i == 0:
                return prism[1,j]
        elif j == 0:
                return prism[i,1]
        else:

                a = prism[i+1][j]
                b = prism[i-1][j]
                c = prism[i][j+1]
                d = prism[i][j-1]

                return (a+b+c+d)/4

####### CALCULOS PARA APENAS 1/4 DO PRISMA

n = 51 #tamanho da matriz n x n
c_min = 0 #indice minimo do condutor
c_max = 10 #indice maximo do condutor
```

```
prism = np.zeros((n,n))

#metade superior da placa direita
c_x = 10 #posi   o da placa no eixo x
for j in range(c_min,c_max+1):
                prism[c_x,j] = -1

plate_right_y = np.arange(c_min,c_max+1)


prism2 = np.ones((n,n))
delta = prism2-prism
tolerance = np.full((n,n), 0.001)

######### GR FICO

plt.ion()  # modo interativo ligado
fig = plt.figure(figsize=(14,6))

full = np.block([
                [np.flipud(np.fliplr(prism[1:,1:])), np.flipud(prism[1:,:])],
                [np.fliplr(prism[:,1:]), prism]
        ])

## 2D
ax1 = fig.add_subplot(1, 2, 1)
im = ax1.imshow(full, cmap='bwr', origin='lower', vmin=-1, vmax=1)
plt.colorbar(im)
ax1.set_title('Potencial␣2D')

## 3D
ax2 = fig.add_subplot(1, 2, 2, projection='3d')
x = np.arange(full.shape[0])
y = np.arange(full.shape[1])
X, Y = np.meshgrid(x, y)
ax2.plot_surface(X, Y, full.T, cmap='hot')
ax2.set_title('Potencial␣3D')

start_time = time.time()
####### CALCULO
for iteration in range(0, 1000):
        for x in range(0,n,2):
                for y in range(0,n,2):
```

```
                        if (x == c_x) and (y in plate_right_y):
                                pass
                        else:
                                prism2[x][y] = potential(x,y)
                                delta = prism2 - prism


        for x in range(0,n):
                for y in range(0,n):

                        if (x == c_x) and (y in plate_right_y):
                                pass
                        else:
                                prism[x][y] = potential(x,y)
                                delta = prism - prism2

        ####### MONTANDO O PRISMA INTEIRO
        full = np.rot90(np.block([
                [np.flipud(np.fliplr(np.abs(prism[1:,1:]))), np.flipud(np.abs(prism[1:,:
                [np.fliplr(prism[:,1:]), prism]
        ]))
        it = time.time()
        it_time = it - start_time
        im.set_data(full)
        ax2.collections.clear()
        ax2.plot_surface(X, Y, full.T, cmap='bwr')
        fig.suptitle(f"Pot ncial␣V␣para␣prisma␣met lico␣com␣duas␣placas␣capacitoras\nI
        plt.pause(0.01)

plt.ioff()
plt.show()
end_time = time.time()
dtime = end_time - start_time
print(f"M ximo␣de␣itera   es!␣Tempo␣de␣execu   o:␣{dtime:.4f}␣segundos")
```

Exercício 4

Listing A.4 – Código do Exercício 4

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time
```

```
def potential(i,j):

        if i == c_x and j in range(c_min,c_max+1):
                return -1
        elif i == 0 or i == n-1 or j == n-1 :
                return 0
        elif i == 0:
                return prism[1,j]
        elif j == 0:
                return prism[i,1]
        else:

                a = prism[i+1][j]
                b = prism[i-1][j]
                c = prism[i][j+1]
                d = prism[i][j-1]

                return (a+b+c+d)/4

def eletric_field(i,j):
        return -(potential(i+1,j)-potential(i-1,j))/2,  -(potential(i,j+1)-potential(i

####### CALCULOS PARA APENAS 1/4 DO PRISMA

n = 51 #tamanho da matriz n x n
c_min = 0 #indice minimo do condutor (altura)
c_max = 10 #indice maximo do condutor (altura)

prism = np.zeros((n,n))

#metade superior da placa direita
c_list = [1,5,10,15,20,25,30,35,40,45,50] #lista de diferentes separa  es
distances = [d*2 for d in c_list]

fringe_campos = []

for d in c_list:
        print(f"Calculando fringe para d = {d}")
        c_x = d #posi  o da placa no eixo x
        for j in range(c_min,c_max+1):
                        prism[c_x,j] = -1

        plate_right_y = np.arange(c_min,c_max+1)
```

```
prism2 = np.ones((n,n))
delta = prism2-prism
tolerance = np.full((n,n), 0.001)



start_time = time.time()
####### CALCULO
for iteration in range(0, 100):
        for x in range(0,n,2):
                for y in range(0,n,2):

                        if (x == c_x) and (y in plate_right_y):
                                pass
                        else:
                                prism2[x][y] = potential(x,y)
                                delta = prism2 - prism


        for x in range(0,n):
                for y in range(0,n):

                        if (x == c_x) and (y in plate_right_y):
                                pass
                        else:
                                prism[x][y] = potential(x,y)
                                delta = prism - prism2

        ####### MONTANDO O PRISMA INTEIRO
        full = np.rot90(np.block([
                [np.flipud(np.fliplr(np.abs(prism[1:,1:]))), np.flipud(np.abs(pri
                [np.fliplr(prism[:,1:]), prism]
        ]))

dVy, dVx = np.gradient(full, y, x)
Ex = -dVx
Ey = -dVy

E_abs = np.sqrt(Ex**2 + Ey**2)
region = E_abs[0:10,0:10]
fringe_mean = np.mean(region)
fringe_campos.append(fringe_mean)
```

```
        it = time.time()
        it_time = it - start_time


plt.plot(distances, fringe_campos, "o-")
plt.xlabel("Distancia entre as placas")
plt.ylabel("Campo de fringing")
plt.title(f"Fringing field vs separa  o entre placas\nTempo de execu ao: {it_time:.2f
end_time = time.time()
dtime = end_time - start_time
print(f"M ximo de itera  es! Tempo de execu  o: {dtime:.4f} segundos")
plt.show()
```

Exercício 5

Listing A.5 – Código do Exercício 5

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time

def potential(i,j):

        if i == c_x and j in range(c_min,c_max+1):
                return -1
        elif i == 0 or i == n-1 or j == n-1 :
                return 0
        elif i == 0:
                return prism[1,j]
        elif j == 0:
                return prism[i,1]
        else:

                a = prism[i+1][j]
                b = prism[i-1][j]
                c = prism[i][j+1]
                d = prism[i][j-1]

                return (a+b+c+d)/4

####### CALCULOS PARA APENAS 1/4 DO PRISMA

n = 51 #tamanho da matriz n x n
```

```
c_min = 0 #indice minimo do condutor
c_max = 10 #indice maximo do condutor

prism = np.zeros((n,n))

#metade superior da placa direita
c_x = 10 #posi   o da placa no eixo x
for j in range(c_min,c_max+1):
                prism[c_x,j] = -1

plate_right_y = np.arange(c_min,c_max+1)


delta_max = 1e9
tolerance = 1e-6

######### GR FICO

plt.ion()  # modo interativo ligado
fig = plt.figure(figsize=(14,6))

full = np.block([
                [np.flipud(np.fliplr(prism[1:,1:])), np.flipud(prism[1:,:])],
                [np.fliplr(prism[:,1:]), prism]
        ])

x = np.arange(full.shape[0])
y = np.arange(full.shape[1])
X, Y = np.meshgrid(x, y)


ax1 = fig.add_subplot(1, 2, 1)
im = ax1.imshow(full, cmap='bwr', origin='lower', vmin=-1, vmax=1)
plt.colorbar(im)
ax1.set_title('Potencial␣2D')

ax2 = fig.add_subplot(1, 2, 2)
dVy, dVx = np.gradient(full)
Ex = -dVx
Ey = -dVy
step = 3  # amostragem das setas
Q = ax2.quiver(X[::step,::step], Y[::step,::step],
               Ex[::step,::step], Ey[::step,::step], scale=0.08, color='r')
ax2.set_title("Campo␣eletrico␣E")
```

```
start_time = time.time()

iteration = 0
####### CALCULO
while delta_max > tolerance:
        delta_max = 0
        iteration +=1
        for x in range(0,n):
                for y in range(0,n):

                        if (x == c_x) and (y in plate_right_y):
                                pass
                        else:
                                new_val = potential(x,y)
                                delta_max = max(delta_max, abs(new_val - prism[x][y]))
                                prism[x][y]=new_val

        ####### MONTANDO O PRISMA INTEIRO
        full = np.rot90(np.block([
                [np.flipud(np.fliplr(np.abs(prism[1:,1:]))), np.flipud(np.abs(prism[1:,:
                [np.fliplr(prism[:,1:]), prism]
        ]))

        dVy, dVx = np.gradient(full, y, x)
        Ex = -dVx
        Ey = -dVy


        it = time.time()
        it_time = it - start_time

        step = 3
        im.set_data(full)
        Q.set_UVC(Ex[::step,::step], Ey[::step,::step])
        fig.suptitle(f"Itera oes:{iteration}\nErro de converg ncia: {delta_max:.6f}\nC
        plt.pause(0.01)

plt.ioff()
plt.show()
end_time = time.time()
dtime = end_time - start_time
print(f"Tempo de execu   o: {dtime:.4f} segundos")
```

Exercício 6

Listing A.6 – Código do Exercício 6

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time

def potential(i,j):

        if i == c_x and j in range(c_min,c_max+1):
                return 0
        elif j == n-1 :
                return 0
        elif j == 0 or i == 0 or i == n-1:
                return 1
        else:
                a = prism[i+1][j]
                b = prism[i-1][j]
                c = prism[i][j+1]
                d = prism[i][j-1]

                return (a+b+c+d)/4

####### CALCULOS PARA APENAS 1/4 DO PRISMA

n = 101 #tamanho da matriz n x n
c_min = 30 #indice minimo do condutor
c_max = 100 #indice maximo do condutor

prism = np.zeros((n,n))

#metade superior da placa direita
c_x = 50 #posi   o da placa no eixo x
'''
for j in range(c_min,c_max+1):
                prism[c_x,j] = 0
'''


rod = np.arange(c_min,c_max+1)


delta_max = 1e9
```

```
tolerance = 1e-6

######### GR FICO

plt.ion()  # modo interativo ligado
fig = plt.figure(figsize=(14,6))
'''
full = np.block([
                [np.flipud(np.fliplr(prism[1:,1:])), np.flipud(prism[1:,:])],
                [np.fliplr(prism[:,1:]), prism]
        ])
'''
x = np.arange(prism.shape[0])
y = np.arange(prism.shape[1])
X, Y = np.meshgrid(x, y)


ax1 = fig.add_subplot(1, 2, 1)
im = ax1.imshow(prism, cmap='hot', origin='upper', vmin=0, vmax=1)
plt.colorbar(im)
ax1.set_title('Potencial 2D')

ax2 = fig.add_subplot(1, 2, 2)
dVy, dVx = np.gradient(prism)
Ex = -dVx
Ey = -dVy
step = 3  # amostragem das setas
Q = ax2.quiver(X[::step,::step], Y[::step,::step],
               Ex[::step,::step], Ey[::step,::step], scale=0.01, color='r')
ax2.set_title("Campo eletrico E")
start_time = time.time()

iteration = 0
####### CALCULO
while delta_max > tolerance:
        delta_max = 0
        iteration +=1
        for x in range(0,n):
                for y in range(0,n):

                        if (x == c_x) and (y in rod):
                                pass
                        else:
                                new_val = potential(x,y)
```

```
                                    delta_max = max(delta_max, abs(new_val - prism[x][y]))
                                    prism[x][y]=new_val

        ####### MONTANDO O PRISMA INTEIRO
        '''
        full = np.rot90(np.block([
                [np.flipud(np.fliplr(np.abs(prism[1:,1:]))), np.flipud(np.abs(prism[1:,:
                [np.fliplr(prism[:,1:]), prism]
        ]))
        '''

        dVy, dVx = np.gradient(prism, y, x)
        Ex = -dVx
        Ey = -dVy


        it = time.time()
        it_time = it - start_time

        step = 3
        im.set_data(prism)
        Q.set_UVC(Ex[::step,::step], Ey[::step,::step])
        fig.suptitle(f"Itera oes:{iteration}\nErro de converg ncia: {delta_max:.6f}\nC
        plt.pause(0.0001)

plt.ioff()
plt.show()
end_time = time.time()
dtime = end_time - start_time
print(f"Tempo de execu   o: {dtime:.4f} segundos")
```

Exercício 7

Listing A.7 – Código do Exercício 7

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


L = 60                # tamanho da grade
tol = 1e-6            # toler ncia para converg ncia
omega = 1.9           # fator de relaxa   o SOR
max_iter = 500     # m ximo de itera   es
update_plot = 1     # atualiza a cada X itera   es
```

```
def inicializar(L):
    V = np.zeros((L, L))
    y1, y2 = L // 3, 2 * L // 3
    xmid = L // 2
    h = 5  # metade da altura da placa
    V[xmid - h:xmid + h, y1] = 1.0   # placa +1
    V[xmid - h:xmid + h, y2] = -1.0  # placa -1
    return V, y1, y2, xmid, h


V_jacobi, y1, y2, xmid, h = inicializar(L)
V_sor = V_jacobi.copy()

plt.ion()
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
im_j = axes[0].imshow(V_jacobi, cmap='plasma', origin='lower', vmin=-1, vmax=1)
im_s = axes[1].imshow(V_sor, cmap='plasma', origin='lower', vmin=-1, vmax=1)
axes[0].set_title("M todo de Jacobi")
axes[1].set_title("M todo SOR")
for ax in axes:
    ax.set_xticks([]); ax.set_yticks([])

plt.tight_layout()


for it in range(1, max_iter + 1):

    # Jacobi: usa c pia anterior
    V_old = V_jacobi.copy()
    V_jacobi[1:-1, 1:-1] = 0.25 * (
        V_old[2:, 1:-1] + V_old[:-2, 1:-1] + V_old[1:-1, 2:] + V_old[1:-1, :-2]
    )
    # reaplica as placas
    V_jacobi[xmid - h:xmid + h, y1] = 1.0
    V_jacobi[xmid - h:xmid + h, y2] = -1.0

    # SOR: atualiza in-place
    for i in range(1, L - 1):
        for j in range(1, L - 1):
            if (j == y1 or j == y2) and (xmid - h <= i < xmid + h):
                continue
            oldV = V_sor[i, j]
```

```
            V_sor[i, j] = (1 - omega) * oldV + omega * 0.25 * (
                V_sor[i + 1, j] + V_sor[i - 1, j] + V_sor[i, j + 1] + V_sor[i, j - 1]
            )

    V_sor[xmid - h:xmid + h, y1] = 1.0
    V_sor[xmid - h:xmid + h, y2] = -1.0

    err_j = np.max(np.abs(V_jacobi - V_old))
    if it % update_plot == 0:
        im_j.set_data(V_jacobi)
        im_s.set_data(V_sor)
        fig.suptitle(f"Itera   o {it}    erro Jacobi={err_j:.2e}")
        plt.pause(0.001)

    if err_j < tol:
        break

plt.ioff()
plt.show()
print(f"Converg ncia atingida em {it} itera   es (erro < {tol})")
```

Exercício 8

Listing A.8 – Código do Exercício 8

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time

def potential(i,j):

        if i == 2 and j == n//2:
                return 1
        elif i == 0 or j == 0 or i == n or j == n:
                return 0
        else:
                a = field[i+1][j]
                b = field[i-1][j]
                c = field[i][j+1]
                d = field[i][j-1]
                e = (rho[i][j]*delta_x**2)/4

                return (a+b+c+d)/4 + e
```

```
n = 21 #tamanho da matriz n x n

rho = np.zeros((n,n))
rho[2][n//2]=1
field=rho.copy()
delta_x = 0.001 # = delta y = delta z

delta_max = 1e9
tolerance = 1e-6

######### GR FICO

plt.ion()  # modo interativo ligado
fig = plt.figure(figsize=(14,14))
gs = fig.add_gridspec(2, 2)

x = np.arange(rho.shape[0])
y = np.arange(rho.shape[1])
X, Y = np.meshgrid(x, y)


ax1 = fig.add_subplot(gs[0, 0])
im = ax1.imshow(field, cmap='plasma', origin='lower', vmin=0, vmax=1)
plt.colorbar(im)
ax1.set_title('Potencial␣2D')

ax2 = fig.add_subplot(gs[0,1])
dVy, dVx = np.gradient(field)
Ex = -dVx
Ey = -dVy
step = 1  # amostragem das setas
Q = ax2.quiver(X[::step,::step], Y[::step,::step],
               Ex[::step,::step], Ey[::step,::step], scale=0.2, color='r')
ax2.set_title("Campo␣eletrico␣E")
ax2.set_aspect('equal')

## 3D
ax3 = fig.add_subplot(gs[1,0], projection='3d')
ax3.plot_surface(X, Y, field.T, cmap='plasma')
ax3.set_title('Potencial␣3D')

start_time = time.time()
```

```
iteration = 0
####### CALCULO
while delta_max > tolerance:
        delta_max = 0
        iteration +=1
        for x in range(0,n-1):
                for y in range(0,n-1):
                        new_val = potential(x,y)
                        delta_max = max(delta_max, abs(new_val - field[x][y]))
                        field[x][y]=new_val


        dVy, dVx = np.gradient(field, y, x)
        Ex = -dVx
        Ey = -dVy


        it = time.time()
        it_time = it - start_time

        step = 1
        im.set_data(field)
        Q.set_UVC(Ex[::step,::step], Ey[::step,::step])
        ax3.collections.clear()
        ax3.plot_surface(X, Y, field.T, cmap='plasma')
        fig.suptitle(f"Itera oes:{iteration}\nErro de converg ncia: {delta_max:.6f}\nC
        plt.pause(0.01)

plt.ioff()
plt.show()
end_time = time.time()
dtime = end_time - start_time
print(f"Tempo de execu   o: {dtime:.4f} segundos")
```

Exercício 9

Listing A.9 – Código do Exercício 9

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time
```

```
def potential(r,V):
        idx = np.where(np.isclose(r_list, r))[0][0]
        if r <= r_min:
                return 1000
        elif np.isclose(r, r_far):
                return 0
        else:
                a = (grid_size**2)
                b = V[idx+1]*(1+(grid_size/r))
                c = V[idx-1]*(1-(grid_size/r))
                return 0.5*(a+b+c)

r_min = 0.2
r_far = 5
grid_size = 0.025

delta_max = 1e9
tolerance = 1e-6

r_list = np.arange(0,r_far+grid_size,grid_size)
V = np.zeros_like(r_list)
V[r_list <= r_min] = 1000
V[np.isclose(r_list, r_far)] = 0

######### GR FICO
plt.ion()  # modo interativo
fig, ax = plt.subplots(figsize=(8, 5))
line, = ax.plot(r_list, V, color='royalblue', lw=2)
ax.set_xlabel("r")
ax.set_ylabel("Potencial␣V(r)")
ax.set_title("Converg ncia␣iterativa␣do␣potencial␣radial")
ax.grid(True)
plt.tight_layout()

iteration = 0
####### CALCULO
while delta_max > tolerance:
        delta_max = 0
        iteration +=1
        for i, r in enumerate(r_list[1:-1], start=1):  # evita bordas
                new_val = potential(r, V)
                delta_max = max(delta_max, abs(new_val - V[i]))
                V[i] = new_val
```

```
        line.set_ydata(V)
        ax.set_title(f"Converg ncia␣do␣potencial␣  ␣itera  o␣{iteration}")
        plt.pause(0.01)

plt.ioff()
line.set_ydata(V)
ax.set_title(f"Solu  o␣final␣ap s␣{iteration}␣itera  es")
plt.draw()
plt.show()

print("Convergiu␣em", iteration, "itera  es")
```

Exercício 10

Listing A.10 – Código do Exercício 10

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

def sor_2d(N, alpha, max_iter=5000, tol=1e-6):
    V = np.zeros((N, N))
    V_new = V.copy()

    # Carga pontual no centro
    cx, cy = N//2, N//2
    V[cx, cy] = 1.0

    start = time.time()
    for it in range(max_iter):
        V_old = V.copy()
        for i in range(1, N-1):
            for j in range(1, N-1):
                if (i, j) != (cx, cy):
                    V_star = 0.25 * (V[i+1,j] + V[i-1,j] + V[i,j+1] + V[i,j-1])
                    V[i,j] = (1 - alpha)*V[i,j] + alpha*V_star
        delta = np.max(np.abs(V - V_old))
        if delta < tol:
            break
    end = time.time()
    return V, it, end - start

def sor_3d(N, alpha, max_iter=3000, tol=1e-6):
    V = np.zeros((N, N, N))
```

```
    V_new = V.copy()

    cx, cy, cz = N//2, N//2, N//2
    V[cx, cy, cz] = 1.0

    start = time.time()
    for it in range(max_iter):
        V_old = V.copy()
        for i in range(1, N-1):
            for j in range(1, N-1):
                for k in range(1, N-1):
                    if (i,j,k) != (cx,cy,cz):
                        V_star = (V[i+1,j,k] + V[i-1,j,k] + V[i,j+1,k] + V[i,j-1,k] + V[
                        V[i,j,k] = (1 - alpha)*V[i,j,k] + alpha*V_star
        delta = np.max(np.abs(V - V_old))
        if delta < tol:
            break
    end = time.time()
    return V, it, end - start

# Testando desempenho
alphas = np.linspace(1.0, 1.95, 10)
N = 50

iters_2d, times_2d = [], []
for a in alphas:
    V, it, t = sor_2d(N, a)
    iters_2d.append(it)
    times_2d.append(t)
    print(f"2D: =␣{a:.2f},␣iter={it},␣tempo={t:.2f}s")

iters_3d, times_3d = [], []
for a in alphas:
    V, it, t = sor_3d(30, a)
    iters_3d.append(it)
    times_3d.append(t)
    print(f"3D: =␣{a:.2f},␣iter={it},␣tempo={t:.2f}s")

# Plotando resultados
plt.figure(figsize=(8,5))
plt.plot(alphas, iters_2d, 'o-', label='2D␣-␣Itera  es')
plt.plot(alphas, iters_3d, 's-', label='3D␣-␣Itera  es')
plt.xlabel('  ')
plt.ylabel('N mero␣de␣itera  es␣at ␣converg ncia')
```

```
plt.legend()
plt.grid(True)
plt.show()
```

Referências