Equação do Calor - Parte 2

Disciplina: Métodos Numéricos em Termofluidos

Professor: Adriano Possebon Rosa

Departamento de Engenharia Mecânica
Faculdade de Tecnologia
Universidade de Brasília

1 Introdução

Vimos, nas últimas aulas, que a equação do calor pode ser resolvida **diretamente**, usando o método numérico das diferenças finitas, quando o problema é **transiente** e é usada uma **formulação explícita** para a aproximação das derivadas.

No entanto, em **problemas permanentes e em problemas transientes com formulação implícita**, as equações de diferenças obtidas se tornam **acopladas**, fazendo com que seja necessário resolver um **sistema de equações lineares** no caso permanente e um sistema para cada passo de tempo, no caso transiente.

Nessa aula vamos estudar esses problemas que envolvem sistemas de equações lineares e vamos ver como é possível resolvê-los sem a montar a matriz **A**.

Esse estudo será **fundamental** para podermos resolver o problema da cavidade cisalhante.

2 Equação do Calor Unidimensional Permanente

Considere novamente a equação do calor para o caso **unidimensional permanente** ($T = T(x)$),

$$\frac{d^2T}{dx^2} = 0 \ , \tag{1}$$

para $0 < x < 1$ e com condições de contorno dadas por

$$T(x = 0) = 0 \qquad \text{e} \qquad T(x = 1) = 1 \ . \tag{2}$$

A EDF (Equação de Diferenças Finitas, uma para cada ponto) desse problema é dada por (os passos para obter essa equação estão nas aulas anteriores)

$$T_{i+1} - 2T_i + T_{i-1} = 0 \ . \tag{3}$$

Vamos analisar um caso particular em que o domínio é dividido em 5 partes iguais, ou seja, $N = 5$. A figura (1) mostra o nosso domínio discretizado. Pelas condições de contorno, os pontos dos extremos já estão definidos. Assim,

$$T_0 = 0 \qquad \text{e} \qquad T_5 = 1\ . \tag{4}$$

Temos que descobrir, portanto, os valores de T_1, T_2, T_3 e T_4.

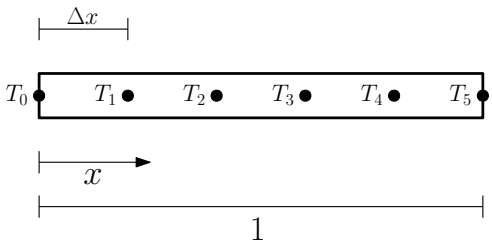

Figura 1: Barra discretizada com comprimento 1.

A equação geral, para qualquer ponto i, é a equação (3). Vamos abrir essa equação em cada ponto:

$$\begin{aligned} i = 1 \qquad & T_2 - 2T_1 + T_0 = 0 \\ i = 2 \qquad & T_3 - 2T_2 + T_1 = 0 \\ i = 3 \qquad & T_4 - 2T_3 + T_2 = 0 \\ i = 4 \qquad & T_5 - 2T_4 + T_3 = 0 \end{aligned} \tag{5}$$

Somando essas 4 equações com as condições de contorno, resulta no seguinte conjunto de equações lineares acopladas:

$$\begin{aligned} T_0 &= 0 \\ T_2 - 2T_1 + T_0 &= 0 \\ T_3 - 2T_2 + T_1 &= 0 \\ T_4 - 2T_3 + T_2 &= 0 \\ T_5 - 2T_4 + T_3 &= 0 \\ T_5 &= 1 \end{aligned} \tag{6}$$

Na forma matricial, esse sistema pode ser representado como

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{7}$$

Podemos deixar assim, ou podemos substituir os valores dos pontos dos cantos nas equações dos seus vizinhos. Isso é feito da seguinte maneira:

$$\begin{aligned} T_2 - 2T_1 &= 0 \\ T_3 - 2T_2 + T_1 &= 0 \\ T_4 - 2T_3 + T_2 &= 0 \\ -2T_4 + T_3 &= -1 \end{aligned} \tag{8}$$

Na forma matricial, esse sistema pode ser reescrito como:

$$\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} \tag{9}$$

E agora? Como resolver esse sistema de equações? Vimos alguns métodos iterativos na aula anterior. Assim, basta substituir as matrizes **A** e **b** nas funções que você criou. No entanto, vamos ver como resolver esse problema **sem montar explicitamente** a matriz **A**.

Voltando no método de Jacobi, vimos que ele é obtido ao isolarmos a componente i na EDF (equação 3), resultando em um processo iterativo. Assim:

$$T_i = \left(\frac{T_{i+1} + T_{i-1}}{2} \right) . \tag{10}$$

O processo iterativo é dado por

$$T_i^{(ITER+1)} = \left(\frac{T_{i+1}^{(ITER)} + T_{i-1}^{(ITER)}}{2} \right) , \tag{11}$$

em que $ITER = 0, 1, \cdots$ corresponde ao valor da iteração. Pronto, **esse é o método de Jacobi**. Estamos resolvendo exatamente o mesmo sistema linear de (9), mas agora sem montar a matriz **A** explicitamente.

Para deixar a variação entre uma iteração e a próxima bem visível, vamos adicionar e subtrair $T_i^{(ITER)}$ no lado direito da equação (11):

$$T_i^{(ITER+1)} = T_i^{(ITER)} + \frac{1}{2} \left(T_{i+1}^{(ITER)} - 2T_i^{(ITER)} + T_{i-1}^{(ITER)} \right) . \tag{12}$$

Esse é o processo iterativo que vamos implementar para resolver o problema. A equação (12) **é válida para pontos do interior do domínio que não estão em contato com as fronteiras**. Para inserir as condições de contorno no problema, **temos que abrir a equação** para os pontos 1 e 4.
(**Atenção:** esses procedimentos serão feitos em todos os problemas. Primeiro obtemos uma equação geral e depois temos que abrir para os pontos em contato com as fronteiras para podermos incluir as condições de contorno.)

Vamos abrir a equação do ponto $i = 1$:

$$T_1^{(ITER+1)} = T_1^{(ITER)} + \frac{1}{2} \left(T_2^{(ITER)} - 2T_1^{(ITER)} + T_0^{(ITER)} \right) . \tag{13}$$

Mas $T_0 = 0$, pelas condições de contorno. Assim:

$$T_1^{(ITER+1)} = T_1^{(ITER)} + \frac{1}{2} \left(T_2^{(ITER)} - 2T_1^{(ITER)} \right) . \tag{14}$$

Vamos abrir a equação do ponto $i = 4$:

$$T_4^{(ITER+1)} = T_4^{(ITER)} + \frac{1}{2} \left(T_5^{(ITER)} - 2T_4^{(ITER)} + T_3^{(ITER)} \right) . \tag{15}$$

Mas $T_5 = 1$, pelas condições de contorno. Assim:

$$T_4^{(ITER+1)} = T_4^{(ITER)} + \frac{1}{2}\left(1 - 2T_4^{(ITER)} + T_3^{(ITER)}\right) \ . \tag{16}$$

Note que as equações são diferentes para os pontos 1 e 4, pois já incluem informações sobre as condições de contorno.

Um algoritmo para a implementação dessa solução é apresentado abaixo.

Algoritmo 1:

Entradas: N (número de divisões do domínio), *tol* (tolerância), $T[0..N]$ (vetor com $N+1$ elementos, com índices de 0 a N, *float*)
Saída: T

Passos:

1. $T[0] \leftarrow 0$
2. $T[N] \leftarrow 1$
3. **declare** $R[0..N]$: zeros, *float*
4. $T^{AUX} \leftarrow T$
5. $iter \leftarrow 0$
6. **faça**:
7. **para** $i = 1, N-1$ **faça**: (obs: i vai de 1 a $N-1$, incluindo 1 e $N-1$.)
8. **se** $i = 1$:
9. $R[i] = 0.5(T[i+1] - 2T[i])$
10. **senão se** $i = N-1$:
11. $R[i] = 0.5(1 - 2T[i] + T[i-1])$
12. **senão**
13. $R[i] = 0.5(T[i+1] - 2T[i] + T[i-1])$
14. $T^{AUX}[i] \leftarrow T[i] + R[i]$
15. $T \leftarrow T^{AUX}$
16. $iter \leftarrow iter + 1$
17. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

Atenção: é necessário sempre abrir as equações para os pontos que estão em contato com as fronteiras e resolver as equações para esses pontos separadamente, como foi feito aqui. Isso vai ser muito importante na implementação do problema da cavidade cisalhante, na parte de mecânica dos fluidos. No entanto, aqui nesse caso como **as condições de**

contorno são de Dirichlet (pra própria variável, e não para suas derivadas), podemos **simplificar o algoritmo acima** eliminando o condicional.

Isso só é possível porque os elementos da matriz dos coeficientes não mudam para os pontos próximos às paredes. **Já quando as condições de contorno são de Neumann (da derivada), aí tem que resolver de maneira especial para os pontos próximos às paredes.**

Veja abaixo:

Algoritmo 2:

Entradas: N (número de divisões do domínio), *tol* (tolerância), $T[0..N]$ (vetor com $N+1$ elementos, com índices de 0 a N, *float*)
Saída: T

Passos:

1. $T[0] \leftarrow 0$
2. $T[N] \leftarrow 1$
3. **declare** $R[0..N]$: zeros, *float*
4. $T^{AUX} \leftarrow T$
5. $iter \leftarrow 0$
6. **faça**:
7. **para** $i = 1, N-1$ **faça**:
8. $R[i] = 0.5(T[i+1] - 2T[i] + T[i-1])$
9. $T^{AUX}[i] \leftarrow T[i] + R[i]$
10. $T \leftarrow T^{AUX}$
11. $iter \leftarrow iter + 1$
12. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

2.1 Resolvendo com Gauss-Seidel e SOR

Vimos que o método de Jacobi pode ser melhorado com pequenas modificações, que resultam nos métodos de Gauss-Seidel e SOR. No método de Gauss-Seidel temos:

$$T_i^{(ITER+1)} = T_i^{(ITER)} + \frac{1}{2}\left(T_{i+1}^{(ITER)} - 2T_i^{(ITER)} + T_{i-1}^{(ITER+1)}\right) . \tag{17}$$

Ou seja, o valor da temperatura que acabou de ser atualizado na iteração atual, já é aproveitado na hora de calcular o seu vizinho. O que muda pro SOR é temos uma constante ω multiplicando o resíduo:

$$T_i^{(ITER+1)} = T_i^{(ITER)} + \frac{\omega}{2}\left(T_{i+1}^{(ITER)} - 2T_i^{(ITER)} + T_{i-1}^{(ITER+1)}\right) . \tag{18}$$

Para implementar esses métodos a partir do método de Jacobi é bem simples: basta atualizar o valor da temperatura em um dado ponto no mesmo vetor T **(por quê?)**. **Veja o algoritmo abaixo** (lembre-se de que o método de Gauss-Seidel é recuperado quando $\omega = 1$).

Algoritmo 3:

Entradas: N (número de divisões do domínio), tol (tolerância), ω (fator do SOR), $T[0..N]$ (vetor com $N+1$ elementos, com índices de 0 a N, *float*)
Saída: T

Passos:

1. $T[0] \leftarrow 0$
2. $T[N] \leftarrow 1$
3. **declare** $R[0..N]$: zeros, *float*
4. $iter \leftarrow 0$
5. **faça**:
6. **para** $i = 1, N-1$ **faça**:
7. $R[i] = 0.5(T[i+1] - 2T[i] + T[i-1])$
8. $T[i] \leftarrow T[i] + \omega R[i]$
9. $iter \leftarrow iter + 1$
10. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

Faça alguns testes para ω entre 1 e 2. Veja como o número de iterações muda com ω.

2.2 Condição de Contorno de Derivada

Considere agora o seguinte problema:

$$\frac{d^2T}{dx^2} = 0 \ , \tag{19}$$

para $0 < x < 1$ e com condições de contorno dadas por

$$T(x=0) = 0 \qquad \text{e} \qquad \left.\frac{dT}{dx}\right|_{(x=1)} = 1 \ . \tag{20}$$

Note que agora temos uma condição de contorno para a derivada no lado direito da barra. No nosso exemplo com $N = 5$, conhecemos T_0 mas não conhecemos mais T_5.

O processo iterativo, obtido a partir da EDF, é dado por

$$T_i^{(ITER+1)} = T_i^{(ITER)} + \frac{1}{2}\left(T_{i+1}^{(ITER)} - 2T_i^{(ITER)} + T_{i-1}^{(ITER)}\right) \ . \tag{21}$$

Essa equação é implementada dessa forma para os pontos internos que não estão em contato com as fronteiras. No entanto, para os pontos que estão em contato com os pontos da fronteira, temos que fazer uma análise particular.

Para $i = 1$:

$$T_1^{(ITER+1)} = T_1^{(ITER)} + \frac{1}{2}\left(T_2^{(ITER)} - 2T_1^{(ITER)} + T_0^{(ITER)}\right) . \tag{22}$$

Mas $T_0 = 0$. Assim, a equação para $i = 1$ se torna:

$$T_1^{(ITER+1)} = T_1^{(ITER)} + \frac{1}{2}\left(T_2^{(ITER)} - 2T_1^{(ITER)}\right) . \tag{23}$$

Para $i = 4$:

$$T_4^{(ITER+1)} = T_4^{(ITER)} + \frac{1}{2}\left(T_5^{(ITER)} - 2T_4^{(ITER)} + T_3^{(ITER)}\right) . \tag{24}$$

Não sabemos o valor de T_5. No entanto, sabemos que

$$\left.\frac{dT}{dx}\right|_{(x=1)} = 1 . \tag{25}$$

Vimos 3 métodos pra fazer essa aproximação: o método de primeira ordem, o de segunda ordem e o de segunda ordem com o *ghost point*. Vamos aplicar primeiro o método de **primeira ordem**. Nesse caso:

$$\left.\frac{dT}{dx}\right|_{(x=1)} \approx \frac{T_5 - T_4}{\Delta x} . \tag{26}$$

Assim:

$$\frac{T_5 - T_4}{\Delta x} = 1 \qquad \rightarrow \qquad T_5 = \Delta x + T4 \tag{27}$$

Substituindo esse valor de T_5 de volta em (24):

$$T_4^{(ITER+1)} = T_4^{(ITER)} + \frac{1}{2}\left(\Delta x + T_4^{(ITER)} - 2T_4^{(ITER)} + T_3^{(ITER)}\right) . \tag{28}$$

Ou:

$$T_4^{(ITER+1)} = T_4^{(ITER)} + \frac{1}{2}\left(\Delta x - T_4^{(ITER)} + T_3^{(ITER)}\right) . \tag{29}$$

Agora temos uma equação especial para $i = 4$, e isso pode ser implementado no algoritmo com um condicional. **Note que em condições de contorno de Neumann, as constantes que multiplicam o termo principal mudam, o que não ocorre nas condições de Dirichlet.**

Veja o algoritmo abaixo (aqui é feita a implementação já com o SOR). Note que o valor de T_5 é atualizado em cada iteração.

Algoritmo 4:

Entradas: N (número de divisões do domínio), *tol* (tolerância), ω (fator do SOR), $T[0..N]$ (vetor com $N + 1$ elementos, com índices de 0 a N, *float*)

Saída: T

Passos:

1. $dx \leftarrow 1/N$
2. **declare** $R[0..N]$: zeros, *float*
3. $iter \leftarrow 0$
4. **faça**:
5. **para** $i = 1, N-1$ **faça**:
6. **se** $i = N-1$:
7. $R[i] = 0.5\,(dx - T[i] + T[i-1])$
8. **senão**:
9. $R[i] = 0.5(T[i+1] - 2T[i] + T[i-1])$
10. $T[i] \leftarrow T[i] + \omega R[i]$
11. $T[N] = T[N-1] + dx$
12. $iter \leftarrow iter + 1$
13. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

Vamos resolver esse problema agora usando a ideia do ***ghost point***. Neste caso, a aproximação para a derivada em $x = 1$ é de segunda ordem, dada por

$$\left.\frac{dT}{dx}\right|_{(x=1)} \approx \frac{T_6 - T_4}{2\Delta x}\ , \tag{30}$$

em que x_6 é o ***ghost point*** e T_6 é o valor da temperatura nesse ponto. Da condição de contorno:

$$\frac{T_6 - T_4}{2\Delta x} = 1 \qquad \rightarrow \qquad T_6 = 2\Delta x + T_4 \tag{31}$$

A equação para T_4 vai ser a equação geral, sem nenhuma modificação:

$$T_4^{(ITER+1)} = T_4^{(ITER)} + \frac{1}{2}\left(T_5^{(ITER)} - 2T_4^{(ITER)} + T_3^{(ITER)}\right)\ . \tag{32}$$

Agora, porém, teremos uma equação para T_5, e é aí que entra a condição de contorno:

$$T_5^{(ITER+1)} = T_5^{(ITER)} + \frac{1}{2}\left(T_6^{(ITER)} - 2T_5^{(ITER)} + T_4^{(ITER)}\right)\ . \tag{33}$$

Substituindo T_6:

$$T_5^{(ITER+1)} = T_5^{(ITER)} + \frac{1}{2}\left(2\Delta x - 2T_5^{(ITER)} + 2T_4^{(ITER)}\right)\ . \tag{34}$$

Um algoritmo que usa esse método é apresentado abaixo. Note que já está sendo utilizado o SOR para resolver o sistema de equações lineares resultante.

Algoritmo 5:

Entradas: N (número de divisões do domínio), tol (tolerância), ω (fator do SOR), $T[0..N]$ (vetor com $N+1$ elementos, com índices de 0 a N, *float*)
Saída: T

Passos:

1. $dx \leftarrow 1/N$
2. **declare** $R[0..N]$: zeros, *float*
3. $iter \leftarrow 0$
4. **faça**:
5. **para** $i = 1, N$ **faça**:
6. **se** $i = N$:
7. $R[i] = 0.5\,(2dx - 2T[i] + 2T[i-1])$
8. **senão**:
9. $R[i] = 0.5(T[i+1] - 2T[i] + T[i-1])$
10. $T[i] \leftarrow T[i] + \omega R[i]$
11. $iter \leftarrow iter + 1$
12. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

3 Problema do Calor Bidimensional Permanente

Vamos agora para o problema **bidimensional permanente**. Vamos resolver a equação de Laplace em um quadrado de lado 1:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0\,, \qquad 0 < x < 1\,, \qquad 0 < y < 1\,, \tag{35}$$

com as seguintes condições de contorno:

$$T(x,0) = 0\,, \qquad 0 \le x \le 1 \tag{36}$$

$$T(x,1) = 0\,, \qquad 0 \le x \le 1 \tag{37}$$

$$T(0,y) = 0\,, \qquad 0 < y < 1 \tag{38}$$

$$T(1, y) = 1\,, \qquad 0 < y < 1\ . \tag{39}$$

O domínio e as condições de contorno são apresentados na figura (2).

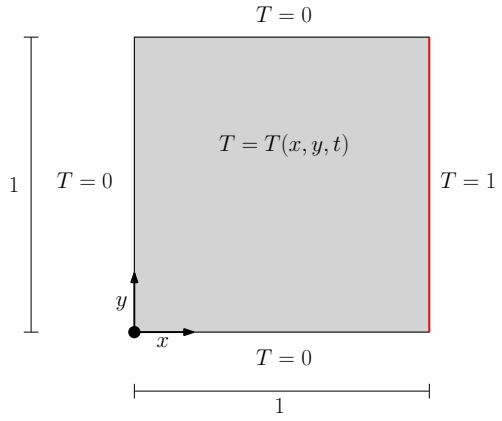

Figura 2: Placa quadrada de lado 1.

Queremos encontrar $T(x, y)$ que satisfaça, simultaneamente, a equação de Laplace e as condições de contorno. **Lembre-se de que em problemas permanentes não existe tempo.**

O domínio discretizado é apresentado na figura (3). Estamos considerando aqui $N_x = 5$ e $N_y = 5$, onde N_x é o número de divisões do domínio na direção x e N_y é o número de divisões na direção y. Temos ainda

$$\Delta x = \frac{1}{N_x} \qquad \text{e} \qquad \Delta y = \frac{1}{N_y}\ . \tag{40}$$

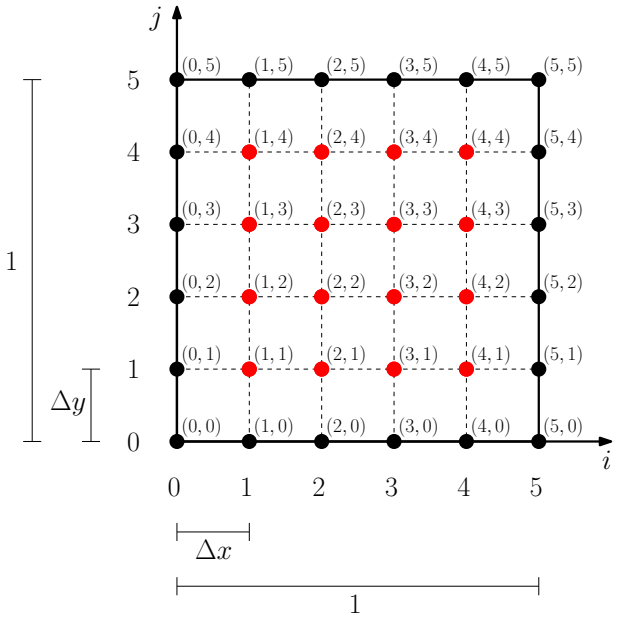

Figura 3: Domínio da placa discretizado. Os pontos pretos pertencem às paredes, enquanto os pontos vermelhos fazem parte do interior do domínio. Nesse caso: $N_x = N_y = 5$.

No domínio discretizado, as temperaturas nos pontos assinalados representam uma aproximação numérica para a temperatura exata. Assim,

$$T_{i,j} \approx T(x_i, y_j) \ , \tag{41}$$

em que $x_i = i\Delta x$ e $y_j = j\Delta y$. A EDF nesse caso (após obtenção das derivadas por meio de séries de Taylor) é

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} = 0 \ . \tag{42}$$

Ou:

$$T_{i+1,j} - 2T_{i,j} + T_{i-1,j} + \left(\frac{\Delta x^2}{\Delta y^2}\right)(T_{i,j+1} - 2T_{i,j} + T_{i,j-1}) = 0 \ . \tag{43}$$

Se $\Delta x = \Delta y$, temos:

$$T_{i+1,j} - 2T_{i,j} + T_{i-1,j} + T_{i,j+1} - 2T_{i,j} + T_{i,j-1} = 0 \ . \tag{44}$$

Ou:

$$T_{i+1,j} + T_{i-1,j} - 4T_{i,j} + T_{i,j+1} + T_{i,j-1} = 0 \ . \tag{45}$$

Atenção: essa equação só vale para $\Delta x = \Delta y$.

A equação (45) é a nossa equação principal geral. Essa é a equação para cada ponto (i, j) do interior do domínio. Note que os pontos das fronteiras já estão resolvidos. No nosso exemplo com $N_x = N_y = 5$, **temos que resolver 16 equações para calcular a temperatura em 16 pontos.**

Para um ponto do interior sem contato com as fronteiras, a equação (45) é aplicada diretamente. Considere o ponto $(2, 3)$. Neste caso:

$$T_{3,3} + T_{1,3} - 4T_{2,3} + T_{2,4} + T_{2,2} = 0 \ . \tag{46}$$

Para incluir as condições de fronteira, temos que abrir todos os pontos que estão em contato com as fronteiras. Temos 2 tipos de pontos: os pontos de canto (pontos $(1, 1)$, $(4, 1)$, $(1, 4)$ e $(4, 4)$), que estão em contato com duas paredes ao mesmo tempo; e os pontos que estão em contato apenas com uma parede. Vamos abrir a EDF para o ponto $(1, 1)$:

$$T_{2,1} + T_{0,1} - 4T_{1,1} + T_{1,2} + T_{1,0} = 0 \ . \tag{47}$$

Os dois ponto em azul são pontos da parede, e têm temperatura nula. Assim, a equação para o ponto $(1, 1)$ se torna:

$$T_{2,1} - 4T_{1,1} + T_{1,2} = 0 \ . \tag{48}$$

Para o ponto $(4, 1)$:

$$T_{5,1} + T_{3,1} - 4T_{4,1} + T_{4,2} + T_{4,0} = 0 \ . \tag{49}$$

Mas $T_{5,1} = 1$ e $T_{4,0} = 0$. Assim:

$$T_{3,1} - 4T_{4,1} + T_{4,2} = -1 \ . \tag{50}$$

Agora vamos considerar o ponto $(3, 4)$, que está em contato apenas com uma parede:

$$T_{4,4} + T_{2,4} - 4T_{3,4} + T_{3,5} + T_{3,3} = 0 \ . \tag{51}$$

Mas $T_{3,5} = 0$. Assim, a equação para esse ponto se torna:

$$T_{4,4} + T_{2,4} - 4T_{3,4} + T_{3,3} = 0 \ . \tag{52}$$

Continuando nesse processo, teremos uma equação para cada ponto do interior do domínio, totalizando 16 equações que formarão o nosso sistema. Esse sistema tem a seguinte representação matricial:

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{array}\right]
\left[\begin{array}{c} T_{1,1} \\ T_{2,1} \\ T_{3,1} \\ T_{4,1} \\ \hline T_{1,2} \\ T_{2,2} \\ T_{3,2} \\ T_{4,2} \\ \hline T_{1,3} \\ T_{2,3} \\ T_{3,3} \\ T_{4,3} \\ \hline T_{1,4} \\ T_{2,4} \\ T_{3,4} \\ T_{4,4} \end{array}\right]
=
\left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \\ 0 \\ 0 \\ \hline -1 \\ -1 \\ -1 \\ -1 \end{array}\right]$$

Esse é o sistema que estamos tentando resolver. Mas para resolvê-lo com os métodos iterativos que vimos anteriormente, não é necessário montar a matriz $\mathbf{A}$, basta saber o efeito de $\mathbf{A}$ sobre um vetor qualquer. (Obs.: note que a matriz $\mathbf{A}$ tem $(N-1)^2 \times (N-1)^2$ elementos.) Vamos desenvolver **o método de Jacobi**, que consiste simplesmente em isolar o termo principal da equação e aplicar um método iterativo.
Considere novamente a EDF geral:

$$T_{i+1,j} + T_{i-1,j} - 4T_{i,j} + T_{i,j+1} + T_{i,j-1} = 0 \ . \tag{53}$$

Essa é a equação para o ponto (i, j). Isolando esse ponto na equação, temos:

$$T_{i,j} = \frac{1}{4}\left(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}\right) \ . \tag{54}$$

Vamos usar essa equação para gerar um método iterativo:

$$T_{i,j}^{(ITER+1)} = \frac{1}{4}\left(T_{i+1,j}^{(ITER)} + T_{i-1,j}^{(ITER)} + T_{i,j+1}^{(ITER)} + T_{i,j-1}^{(ITER)}\right) \ . \tag{55}$$

Adicionando e subtraindo $T_{i,j}^{(ITER)}$ no lado direito:

$$T_{i,j}^{(ITER+1)} = T_{i,j}^{(ITER)} + \frac{1}{4}\left(T_{i+1,j}^{(ITER)} + T_{i-1,j}^{(ITER)} - 4T_{i,j}^{(ITER)} + T_{i,j+1}^{(ITER)} + T_{i,j-1}^{(ITER)}\right) \ . \tag{56}$$

Essa é a equação para o método iterativo que vai resolver o sistema linear e a equação do calor permanente bidimensional. As condições de contorno são inseridas nessa equação.

Abaixo é apresentado um algoritmo para esse caso.

Atenção: esse algoritmo resolve apenas o problema apresentado, com as respectivas condições de contorno e com $N_x = N_y = N$. Para casos gerais em que $N_x \neq N_y$, você tem que voltar na equação (43) e continuar o desenvolvimento a partir dela.

Algoritmo 6:

Entradas: N (número de divisões do domínio em cada direção, $N_x = N_y = N$), tol (tolerância), $T[0..N, 0..N]$ (matriz com duas dimensões com $(N+1) \times (N+1)$ elementos, com índices de 0 a N em cada dimensão, *float*)
Saída: T

Passos:

1. **para** $i = 0, N$ **faça**:
2. $\quad T[i, 0] \leftarrow 0$
3. $\quad T[i, N] \leftarrow 0$
4. **para** $j = 1, N-1$ **faça**:
5. $\quad T[0, j] \leftarrow 0$
6. $\quad T[N, j] \leftarrow 1$
7. **declare** $R[0..N, 0..N]$: zeros, *float*
8. $T^{AUX} \leftarrow T$
9. $iter \leftarrow 0$
10. **faça**:
11. $\quad$ **para** $i = 1, N-1$ **faça**:
12. $\quad\quad$ **para** $j = 1, N-1$ **faça**:
13. $\quad\quad\quad$ **se** $i = 1$ **e** $j = 1$:
14. $\quad\quad\quad\quad R[i, j] = 0.25\,(T[i+1, j] - 4T[i, j] + T[i, j+1])$
15. $\quad\quad\quad$ **senão se** $i = 1$ **e** $j = N-1$:
16. $\quad\quad\quad\quad R[i, j] = 0.25\,(T[i+1, j] - 4T[i, j] + T[i, j-1])$
17. $\quad\quad\quad$ **senão se** $i = N-1$ **e** $j = 1$:
18. $\quad\quad\quad\quad R[i, j] = 0.25\,(1 + T[i-1, j] - 4T[i, j] + T[i, j+1])$
19. $\quad\quad\quad$ **senão se** $i = N-1$ **e** $j = N-1$:
20. $\quad\quad\quad\quad R[i, j] = 0.25\,(1 + T[i-1, j] - 4T[i, j] + T[i, j-1])$

21. **senão se** $i = 1$:
22. $R[i,j] = 0.25\left(T[i+1,j] - 4T[i,j] + T[i,j+1] + T[i,j-1]\right)$
23. **senão se** $i = N-1$:
24. $R[i,j] = 0.25\left(T[i-1,j] - 4T[i,j] + T[i,j+1] + T[i,j-1]\right)$
25. **senão se** $j = 1$:
26. $R[i,j] = 0.25\left(T[i+1,j] + T[i-1,j] - 4T[i,j] + T[i,j+1]\right)$
27. **senão se** $j = N-1$:
28. $R[i,j] = 0.25\left(T[i+1,j] + T[i-1,j] - 4T[i,j] + T[i,j-1]\right)$
29. **senão**:
30. $R[i,j] = 0.25\left(T[i+1,j] + T[i-1,j] - 4T[i,j] + T[i,j+1] + T[i,j-1]\right)$
31. $T^{AUX}[i,j] \leftarrow T[i,j] + R[i,j]$
32. $T \leftarrow T^{AUX}$
33. $iter \leftarrow iter + 1$
34. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

Atenção: é necessário sempre abrir as equações para os pontos que estão em contato com as fronteiras e resolver as equações para esses pontos separadamente, como foi feito aqui. Isso vai ser muito importante na implementação do problema da cavidade cisalhante, na parte de mecânica dos fluidos. No entanto, aqui nesse caso como **as condições de contorno são de Dirichlet** (pra própria variável, e não para suas derivadas), podemos **simplificar o algoritmo acima** eliminando o condicional.

Isso só é possível porque os elementos da matriz dos coeficientes não mudam para os pontos próximos às paredes. **Já quando as condições de contorno são de Neumann (da derivada), aí tem que resolver de maneira especial para os pontos próximos às paredes.**

Veja abaixo:

Algoritmo 7:

Entradas: N (número de divisões do domínio em cada direção, $N_x = N_y = N$), tol (tolerância), $T[0..N, 0..N]$ (matriz com duas dimensões com $(N+1) \times (N+1)$ elementos, com índices de 0 a N em cada dimensão, *float*)
Saída: T

Passos:

1. **para** $i = 0, N$ **faça**:
2. $T[i,0] \leftarrow 0$

3. $T[i, N] \leftarrow 0$
4. **para** $j = 1, N - 1$ **faça**:
5. $T[0, j] \leftarrow 0$
6. $T[N, j] \leftarrow 1$
7. **declare** $R[0..N, 0..N]$: zeros, *float*
8. $T^{AUX} \leftarrow T$
9. $iter \leftarrow 0$
10. **faça**:
11. **para** $i = 1, N - 1$ **faça**:
12. **para** $j = 1, N - 1$ **faça**:
13. $R[i, j] = 0.25\left(T[i + 1, j] + T[i - 1, j] - 4T[i, j] + T[i, j + 1] + T[i, j - 1]\right)$
14. $T^{AUX}[i, j] \leftarrow T[i, j] + R[i, j]$
15. $T \leftarrow T^{AUX}$
16. $iter \leftarrow iter + 1$
17. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

Abaixo temos um algoritmo muito parecido com o algoritmo acima, mas com o método SOR.

Algoritmo 8:

Entradas: N (número de divisões do domínio em cada direção, $N_x = N_y = N$), tol (tolerância), ω (fator do SOR), $T[0..N, 0..N]$ (matriz com duas dimensões com $(N + 1) \times (N + 1)$ elementos, com índices de 0 a N em cada dimensão, *float*)
Saída: T

Passos:

1. **para** $i = 0, N$ **faça**:
2. $T[i, 0] \leftarrow 0$
3. $T[i, N] \leftarrow 0$
4. **para** $j = 1, N - 1$ **faça**:
5. $T[0, j] \leftarrow 0$
6. $T[N, j] \leftarrow 1$
7. **declare** $R[0..N, 0..N]$: zeros, *float*

8. $iter \leftarrow 0$
9. **faça**:
10. **para** $i = 1, N-1$ **faça**:
11. **para** $j = 1, N-1$ **faça**:
12. $R[i,j] = 0.25\left(T[i+1,j] + T[i-1,j] - 4T[i,j] + T[i,j+1] + T[i,j-1]\right)$
13. $T[i,j] \leftarrow T[i,j] + \omega R[i,j]$
14. $iter \leftarrow iter + 1$
15. **se** $max(abs(R)) < tol$: **pare**

Fim do Algoritmo

4 Problema do Calor Unidimensional Transiente

Vamos resolver novamente o problema **transiente** de condução de calor em uma barra:

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} \ , \qquad 0 < x < 1, \qquad t > 0 \ , \tag{57}$$

com

$$T(x,0) = 0 \ , \qquad 0 < x < 1 \tag{58}$$

$$T(0,t) = 0 \ , \qquad t \geq 0 \tag{59}$$

$$T(1,t) = 1 \ , \qquad t \geq 0 \ , \tag{60}$$

Agora, o ponto numérico T_i^k aproxima a solução no ponto $x_i = i\Delta x$ e no instante de tempo $t_k = k\Delta t$, ou seja,

$$T_i^k \approx T(x_i, t_k) \ . \tag{61}$$

O domínio discretizado é apresentado na figura (1). Vamos considerar que a barra está dividida em $N = 5$ partes iguais. Já resolvemos esse problema utilizando o método de diferenças finitas com uma formulação explícita das equações de diferenças finitas (EDF). Agora vamos resolver esse problema com uma **aproximação implícita.**
A EDF é nesse caso é obtida a partir de uma expansão usando série de Taylor **para trás no tempo e centrada no espaço**, resultando em

$$\frac{T_i^{k+1} - T_i^k}{\Delta t} = \frac{T_{i+1}^{k+1} - 2T_i^{k+1} + T_{i-1}^{k+1}}{\Delta x^2} \ . \tag{62}$$

Essa formulação é chamada de **BTCS**: *backward-time centered-space*. **Por que usar um método implícito?** Vantagem: ele é incondicionalmente estável, então não tem mais a restrição de $\Delta t < \Delta x^2/2$. Desvantagem: temos que resolver um sistema de equações em cada passo de tempo. Em certas situações vale a pena, especialmente quando Δx é pequeno.

A condição de contorno no nosso problema é $T_0^k = 0$ e $T_N^k = 1$ para todo $k \geq 0$ e a condição inicial é $T_i^0 = 0$ para $0 < i < N$.

Para resolver a equação (62), vamos utilizar o método de Jacobi. Para isso, vamos isolar nossa incógnita principal, T_i^{k+1} e gerar um método iterativo a partir disso. Isolando:

$$T_i^{k+1} = T_i^k + \frac{\Delta t}{\Delta x^2}\left(T_{i+1}^{k+1} + T_{i-1}^{k+1}\right) - \frac{\Delta t}{\Delta x^2}\left(2T_i^{k+1}\right) \ . \tag{63}$$

Vamos definir

$$\lambda = \frac{\Delta t}{\Delta x^2} \ . \tag{64}$$

Assim:

$$T_i^{k+1}(1+2\lambda) = T_i^k + \lambda\left(T_{i+1}^{k+1} + T_{i-1}^{k+1}\right) \ . \tag{65}$$

Finalmente:

$$T_i^{k+1} = \frac{1}{(1+2\lambda)}\left[T_i^k + \lambda\left(T_{i+1}^{k+1} + T_{i-1}^{k+1}\right)\right] \ . \tag{66}$$

Essa última equação nos sugere um método iterativo. Queremos encontrar a temperatura em $k+1$. Assim:

$$T_i^{k+1(ITER+1)} = \frac{1}{(1+2\lambda)}\left[T_i^k + \lambda\left(T_{i+1}^{k+1(ITER)} + T_{i-1}^{k+1(ITER)}\right)\right] \ . \tag{67}$$

Atenção: não confundir **tempo** com **iteração**. Em cada passo de tempo temos que resolver um sistema de equações, ou seja, **temos que iterar várias vezes até a convergência em cada passo de tempo**. **Note que T_i^k na equação (67) é uma constante**, já é conhecida, e não vai mudar durante as iterações dentro de um tempo fixo.

A equação (67) é a equação que vamos utilizar pra resolver o problema. Para deixá-la um pouco mais elegante e também para possibilitar uma implementação fácil do SOR, vamos somar e subtrair $T_i^{k+1(ITER)}$ no lado direito:

$$\begin{aligned} T_i^{k+1(ITER+1)} &= T_i^{k+1(ITER)} + \\ &+ \frac{1}{(1+2\lambda)}\left[T_i^k + \lambda\left(T_{i+1}^{k+1(ITER)} + T_{i-1}^{k+1(ITER)}\right) - (1+2\lambda)T_i^{k+1(ITER)}\right] \ . \end{aligned} \tag{68}$$

Assim, o nosso método iterativo pode ser apresentado como

$$R_i^{(ITER)} = \frac{1}{(1+2\lambda)}\left[T_i^k + \lambda\left(T_{i+1}^{k+1(ITER)} + T_{i-1}^{k+1(ITER)}\right) - (1+2\lambda)T_i^{k+1(ITER)}\right] \ . \tag{69}$$

$$T_i^{k+1(ITER+1)} = T_i^{k+1(ITER)} + R_i^{(ITER)} \tag{70}$$

Esse método de resolução é o de Jacobi. **O método SOR é dado por:**

$$R_i^{(ITER)} = \frac{1}{(1+2\lambda)}\left[T_i^k + \lambda\left(T_{i+1}^{k+1(ITER)} + T_{i-1}^{k+1(ITER+1)}\right) - (1+2\lambda)T_i^{k+1(ITER)}\right] \ . \tag{71}$$

$$T_i^{k+1(ITER+1)} = T_i^{k+1(ITER)} + \omega R_i^{(ITER)} \ . \tag{72}$$

A formulação BTCS associada com o método SOR é apresentada no algoritmo abaixo.

Algoritmo 9:

Entradas: N (número de divisões do domínio), tol (tolerância), ω (fator do SOR), $T[0..N]$ (vetor com $N+1$ elementos, com índices de 0 a N, *float*), dt (passo de tempo), t_{final} (tempo final)
Saída: T

Passos:

1. $dx = 1/N$
2. $\lambda = dt/dx^2$
3. $T[0] \leftarrow 0$
4. $T[N] \leftarrow 1$
5. **declare** $R[0..N]$: zeros, *float*
6. $T^{NOVO} \leftarrow T$
7. $t \leftarrow 0$
8. **faça**:
9. $\quad iter \leftarrow 0$
10. $\quad$ **faça**:
11. $\quad\quad$ **para** $i = 1, N-1$ **faça**:
12. $\quad\quad\quad R[i] = \frac{1}{(1+2\lambda)}\left[T[i] + \lambda\left(T^{NOVO}[i+1] + T^{NOVO}[i-1]\right) - (1+2\lambda)T^{NOVO}[i]\right]$
13. $\quad\quad\quad T^{NOVO}[i] \leftarrow T^{NOVO}[i] + \omega R[i]$
14. $\quad\quad iter \leftarrow iter + 1$
15. $\quad\quad$ **se** $max(abs(R)) < tol$: **pare**
16. $\quad T \leftarrow T^{NOVO}$
17. $\quad t \leftarrow t + dt$
18. $\quad$ **se** $t \geq t_{final}$: **pare**

Fim do Algoritmo

4.1 Crank-Nicolson

A formulação BTCS é de primeira ordem no tempo e de segunda ordem no espaço, ou seja, o erro nesse método é $O(\Delta t, \Delta x^2)$. Uma formulação que possui basicamente o mesmo custo computacional mas que é $O(\Delta t^2, \Delta x^2)$ é a formulação de Crank-Nicolson:

$$\frac{T_i^{k+1} - T_i^k}{\Delta t} = \frac{1}{2}\left(\frac{T_{i+1}^k - 2T_i^k + T_{i-1}^k}{\Delta x^2} + \frac{T_{i+1}^{k+1} - 2T_i^{k+1} + T_{i-1}^{k+1}}{\Delta x^2}\right) . \tag{73}$$

A implementação dessa formulação pode ser feita a partir dos passos apresentados para a formulação BTCS. Vou deixar essa implementação para você.

5 Problema do Calor Bidimensional Transiente

Vamos voltar agora no problema **transiente** de condução de calor em uma placa:

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}, \quad 0 < x < 1, \quad 0 < y < 1, \quad t > 0 \tag{74}$$

$$T(x, 0, t) = 0, \qquad 0 \leq x \leq 1, \quad t \geq 0 \tag{75}$$

$$T(x, 1, t) = 0, \qquad 0 \leq x \leq 1, \quad t \geq 0 \tag{76}$$

$$T(0, y, t) = 0, \qquad 0 < y < 1, \quad t \geq 0 \tag{77}$$

$$T(1, y, t) = 1, \qquad 0 < y < 1, \quad t \geq 0 \tag{78}$$

$$T(x, y, 0) = 0, \qquad 0 < x < 1, \quad 0 < y < 1 \tag{79}$$

Usando a formulação BTCS para aproximar as derivadas, temos:

$$\frac{T_{i,j}^{k+1} - T_{i,j}^k}{\Delta t} = \frac{T_{i+1,j}^{k+1} - 2T_{i,j}^{k+1} + T_{i-1,j}^{k+1}}{\Delta x^2} + \frac{T_{i,j+1}^{k+1} - 2T_{i,j}^{k+1} + T_{i,j-1}^{k+1}}{\Delta y^2} , \tag{80}$$

Para encontrar o valor de $T_{i,j}^{k+1}$, vamos usar o método de Jacobi. Pra isso, vamos isolar esse termo na equação (80), resultando, após alguns passos, em:

$$T_{i,j}^{(k+1)} = \frac{1}{(1+4\lambda)}\left[T_{i,j}^k + \lambda\left(T_{i+1,j}^k + T_{i-1,j}^k + T_{i,j+1}^k + T_{i,j-1}^k\right)\right] \tag{81}$$

Aqui, $\lambda = \Delta t/\Delta x^2 = \Delta t/\Delta y^2$. Estamos considerando $\Delta x = \Delta y$. O processo iterativo é dado por:

$$R_{i,j}^{(ITER)} = \frac{1}{(1+4\lambda)}\Bigg[T_{i,j}^k + \lambda\Big(T_{i+1,j}^{k+1(ITER)} + T_{i-1,j}^{k+1(ITER)} + T_{i,j+1}^{k+1(ITER)} + T_{i,j-1}^{k+1(ITER)}\Big) - (1+4\lambda)\Big(T_{i,j}^{k+1(ITER)}\Big)\Bigg] \tag{82}$$

$$T_{i,j}^{k+1(ITER+1)} = T_{i,j}^{k+1(ITER)} + R_{i,j}^{(ITER)} \tag{83}$$

O SOR é obtido a partir das mudanças feitas abaixo:

$$R_{i,j}^{(ITER)} = \frac{1}{(1+4\lambda)}\Bigg[T_{i,j}^{k} + \lambda\Big(T_{i+1,j}^{k+1(ITER)} + T_{i-1,j}^{k+1(ITER+1)} + T_{i,j+1}^{k+1(ITER)} + T_{i,j-1}^{k+1(ITER+1)}\Big) - (1+4\lambda)\Big(T_{i,j}^{k+1(ITER)}\Big)\Bigg] \tag{84}$$

$$T_{i,j}^{k+1(ITER+1)} = T_{i,j}^{k+1(ITER)} + \omega R_{i,j}^{(ITER)} \tag{85}$$

O algoritmo a seguir apresenta os passos para a resolução da equação do calor nessas condições, usando o SOR.

Algoritmo 10:

Entradas: N (número de divisões do domínio em cada direção, $N_x = N_y = N$), tol (tolerância), ω (fator do SOR), $T[0..N, 0..N]$ (matriz com duas dimensões com $(N+1) \times (N+1)$ elementos, com índices de 0 a N em cada dimensão, *float*), dt (passo de tempo), t_{final} (tempo final)
Saída: T

Passos:

1. **para** $i = 0, N$ **faça**:
2. $\quad T[i, 0] \leftarrow 0$
3. $\quad T[i, N] \leftarrow 0$
4. **para** $j = 1, N-1$ **faça**:
5. $\quad T[0, j] \leftarrow 0$
6. $\quad T[N, j] \leftarrow 1$
7. $dx = 1/N$
8. $\lambda = dt/dx^2$
9. **declare** $R[0..N, 0..N]$: zeros, *float*
10. $T^{NOVO} \leftarrow T$
11. $t \leftarrow 0$
12. **faça**:
13. $\quad iter \leftarrow 0$
14. $\quad$ **faça**:

15. **para** $i = 1, N-1$ **faça**:

16. **para** $j = 1, N-1$ **faça**:

17. $$R[i,j] = \frac{1}{(1+4\lambda)}\Big[T[i,j] + \lambda\Big(T^{(NOVO)}[i+1,j] + T^{(NOVO)}[i-1,j] + \\ + T^{(NOVO)}[i,j+1] + T^{(NOVO)}[i,j-1]\Big) - (1+4\lambda)\Big(T^{(NOVO)}[i,j]\Big)\Big]$$

18. $T^{NOVO}[i,j] \leftarrow T^{NOVO}[i,j] + \omega R[i,j]$

19. $iter \leftarrow iter + 1$

20. **se** $max(abs(R)) < tol$: **pare**

21. $T \leftarrow T^{NOVO}$

22. $t \leftarrow t + dt$

23. **se** $t \geq t_{final}$: **pare**

Fim do Algoritmo

5.1 Crank-Nicolson

Nesse caso, a formulação de Crank-Nicolson é dadar por:

$$\frac{T_{i,j}^{k+1} - T_{i,j}^{k}}{\Delta t} = \frac{1}{2}\left(\frac{T_{i+1,j}^{k} - 2T_{i,j}^{k} + T_{i-1,j}^{k}}{\Delta x^2} + \frac{T_{i,j+1}^{k} - 2T_{i,j}^{k} + T_{i,j-1}^{k}}{\Delta y^2} + \right. \\ \left. + \frac{T_{i+1,j}^{k+1} - 2T_{i,j}^{k+1} + T_{i-1,j}^{k+1}}{\Delta x^2} + \frac{T_{i,j+1}^{k+1} - 2T_{i,j}^{k+1} + T_{i,j-1}^{k+1}}{\Delta y^2}\right). \tag{86}$$

O algoritmo e a implementação dessa formulação serão cobradas no Trabalho 2.

Equação do Calor Parte 2

March 24, 2021

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1 Algoritmo 1

[2]:
```
N = 5
tol = 1.e-5
T = np.zeros(N+1,float)

T[0] = 0.0
T[N] = 1.0

R = np.zeros(N+1,float)
T_aux = np.copy(T)

iter = 0

while True:
    for i in range(1,N):
        if i == 1:
            R[i] = 0.5*(T[i+1] - 2.0*T[i])
        elif i == N-1:
            R[i] = 0.5*(1.0 - 2.0*T[i] + T[i-1])
        else:
            R[i] = 0.5*(T[i+1] - 2.0*T[i] + T[i-1])
        T_aux[i] = T[i] + R[i]
    T = np.copy(T_aux)
    iter += 1
    #print(iter,T)
    if np.max(np.abs(R)) < tol: break
print(iter,T)
```

```
49 [0.         0.19998764 0.39998382 0.59998    0.79999    1.        ]
```

2 Algoritmo 2

[3]:
```
N = 5
tol = 1.e-5
T = np.zeros(N+1,float)

T[0] = 0.0
T[N] = 1.0

R = np.zeros(N+1,float)
T_aux = np.copy(T)

iter = 0

while True:
    for i in range(1,N):
        R[i] = 0.5*(T[i+1] - 2.0*T[i] + T[i-1])
        T_aux[i] = T[i] + R[i]
    T = np.copy(T_aux)
    iter += 1
    #print(iter,T)
    if np.max(np.abs(R)) < tol: break
print(iter,T)
```

```
49 [0.         0.19998764 0.39998382 0.59998    0.79999    1.        ]
```

3 Algoritmo 3

[4]:
```
N = 5
tol = 1.e-5
T = np.zeros(N+1,float)
omega = 1.0

T[0] = 0.0
T[N] = 1.0

R = np.zeros(N+1,float)

iter = 0

while True:
    for i in range(1,N):
        R[i] = 0.5*(T[i+1] - 2.0*T[i] + T[i-1])
        T[i] += omega*R[i]
    iter += 1
    #print(iter,T)
```

```
    if np.max(np.abs(R)) < tol: break
print(iter,T)
```

```
26 [0.         0.19998764 0.39998382 0.59998691 0.79999346 1.        ]
```

4 Algoritmo 4

[5]:
```
N = 5
tol = 1.e-5
T = np.zeros(N+1,float)
omega = 1.5

dx = 1.0/N

R = np.zeros(N+1,float)

iter = 0

while True:
    for i in range(1,N):
        if i == N-1:
            R[i] = 0.5*(dx - T[i] + T[i-1])
        else:
            R[i] = 0.5*(T[i+1] - 2.0*T[i] + T[i-1])
        T[i] = T[i] + omega*R[i]
    T[N] = T[N-1] + dx
    iter += 1
    #print(iter,T)
    if np.max(np.abs(R)) < tol: break
print(iter,T)
```

```
42 [0.         0.19998003 0.3999656  0.59995655 0.79995238 0.99995238]
```

5 Algoritmo 5

[6]:
```
N = 5
tol = 1.e-5
T = np.zeros(N+1,float)
omega = 1.5

dx = 1.0/N

R = np.zeros(N+1,float)

iter = 0
```

```
while True:
    for i in range(1,N+1):
        if i == N:
            R[i] = 0.5*(2.0*dx - 2.0*T[i] + 2.0*T[i-1])
        else:
            R[i] = 0.5*(T[i+1] - 2.0*T[i] + T[i-1])
        T[i] = T[i] + omega*R[i]
    iter += 1
    #print(iter,T)
    if np.max(np.abs(R)) < tol: break
print(iter,T)
```

```
26 [0.         0.19998447 0.39997616 0.59997351 0.79997487 0.99997868]
```

6 Algoritmo 7

[7]:
```
N = 5
tol = 1.e-5
T = np.zeros((N+1,N+1),float)

for i in range(N+1):
    T[i,0] = 0.0
    T[i,N] = 0.0

for j in range(1,N):
    T[0,j] = 0.0
    T[N,j] = 1.0

R = np.zeros((N+1,N+1),float)

T_aux = np.copy(T)

iter = 0

while True:
    for i in range(1,N):
        for j in range(1,N):
            R[i,j] = 0.25*(T[i+1,j] + T[i-1,j] - 4.0*T[i,j] + T[i,j+1] +␣
 ↪T[i,j-1])
            T_aux[i,j] = T[i,j] + R[i,j]
    T = np.copy(T_aux)
    iter += 1
    if np.max(np.abs(R)) < tol: break
print(iter,T)

Temp = np.copy(T)
```

```
43 [[0.         0.         0.         0.         0.         0.        ]
 [0.         0.04544012 0.07194636 0.07194636 0.04544012 0.        ]
 [0.         0.10982514 0.17041678 0.17041678 0.10982514 0.        ]
 [0.         0.22346151 0.32950769 0.32950769 0.22346151 0.        ]
 [0.         0.45453103 0.59467363 0.59467363 0.45453103 0.        ]
 [0.         1.         1.         1.         1.         0.        ]]
```

7 Algoritmo 8

[8]:
```
N = 100
tol = 1.e-5
T = np.zeros((N+1,N+1),float)
omega = 1.92

for i in range(N+1):
    T[i,0] = 0.0
    T[i,N] = 0.0

for j in range(1,N):
    T[0,j] = 0.0
    T[N,j] = 1.0

R = np.zeros((N+1,N+1),float)

iter = 0

while True:
    for i in range(1,N):
        for j in range(1,N):
            R[i,j] = 0.25*(T[i+1,j] + T[i-1,j] - 4.0*T[i,j] + T[i,j+1] +␣
 ↪T[i,j-1])
            T[i,j] = T[i,j] + omega*R[i,j]
    iter += 1
    #print(iter,T)
    if np.max(np.abs(R)) < tol: break
print(iter,T)
```

```
266 [[0.00000000e+00 0.00000000e+00 0.00000000e+00 … 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.07280551e-04 2.14512528e-04 … 2.17681861e-04
  1.08904725e-04 0.00000000e+00]
 [0.00000000e+00 2.14736346e-04 4.29373393e-04 … 4.35620918e-04
  2.17937942e-04 0.00000000e+00]
 …
 [0.00000000e+00 3.02127393e-01 4.99560136e-01 … 4.99561719e-01
  3.02128205e-01 0.00000000e+00]
 [0.00000000e+00 4.99890033e-01 6.97432741e-01 … 6.97433522e-01
```

```
 4.99890433e-01 0.00000000e+00]
[0.00000000e+00 1.00000000e+00 1.00000000e+00 … 1.00000000e+00
 1.00000000e+00 0.00000000e+00]]
```

8 Algoritmo 9

[9]:
```
N = 5
tol = 1.e-5
T = np.zeros(N+1,float)
omega = 1.3
t_final = 1.5
dt = 0.01

dx = 1.0/N
lamda = dt/(dx*dx)

T[0] = 0.0
T[N] = 1.0

R = np.zeros(N+1,float)

T_novo = np.copy(T)

t = 0.0

while True:
    iter = 0
    while True:
        for i in range(1,N):
            a1 = (1.0 + 2.0*lamda)
            R[i] = (1.0/a1)*(T[i] + lamda*(T_novo[i+1]+T_novo[i-1]) -␣
↪a1*T_novo[i])
            T_novo[i] = T_novo[i] + omega*R[i]
        iter += 1
        if np.max(np.abs(R)) < tol: break
    T = np.copy(T_novo)
    t += dt
    #print(t, iter, T)
    if t >= t_final-0.1*dt: break
print(t, iter, T)
```

```
1.500000000000001 1 [0.         0.19999981 0.39999971 0.59999972 0.79999984 1.
]
```

9 Algoritmo 10

[10]:
```
N = 5
tol = 1.e-5
T = np.zeros((N+1,N+1),float)
omega = 1.3
t_final = 1.0
dt = 0.01

for i in range(N+1):
    T[i,0] = 0.0
    T[i,N] = 0.0

for j in range(1,N):
    T[0,j] = 0.0
    T[N,j] = 1.0

dx = 1.0/N
lamda = dt/(dx*dx)

R = np.zeros((N+1,N+1),float)

T_novo = np.copy(T)

t = 0.0

while True:
    iter = 0
    while True:
        for i in range(1,N):
            for j in range(1,N):
                a1 = (1.0 + 4.0*lamda)
                R[i,j] = (1.0/a1)*(T[i,j] +
                        lamda*(T_novo[i+1,j]+T_novo[i-1,j] +␣
 ↪T_novo[i,j+1]+T_novo[i,j-1])
                        - a1*T_novo[i,j])
                T_novo[i,j] = T_novo[i,j] + omega*R[i,j]
        iter += 1
        if np.max(np.abs(R)) < tol: break
    T = np.copy(T_novo)
    t += dt
    #print(t, iter, T)
    if t >= t_final-0.1*dt: break
print(t, iter, T)
```

```
1.0000000000000007 1 [[0.          0.          0.          0.          0.          0.
]
```

```
[0.         0.04545454 0.07196969 0.07196969 0.04545454 0.        ]
[0.         0.10984848 0.17045454 0.17045454 0.10984848 0.        ]
[0.         0.22348485 0.32954545 0.32954545 0.22348485 0.        ]
[0.         0.45454545 0.59469697 0.59469697 0.45454545 0.        ]
[0.         1.         1.         1.         1.         0.        ]]
```

10 Acelerando o Código

O Python é uma ótima linguagem de programação, pois é fácil de programar (boa para iniciantes) e permite que você faça tudo em um único código ou linguagem (rodar a parte pesada e fazer os gráficos e análise dos dados, ou seja, processamento e pós-processamento).

O problema é que o Python é lento. **Muito lento. Lento de verdade.** Principalmente na hora de fazer *loops* dentro de *loops*. E agora vamos começar a perceber isso, especialmente nos problemas 2D. Volte no código do algoritmo 8. Para $N = 5$ não tem nenhum problema, é quase instantâneo. Mas queremos soluções melhores, com valores maiores de N. Tente rodar com $N = 100$ ou $N = 200$. Demora um tempinho. Isso é o **custo computacional.** De agora em diante em nosso curso só vai piorando, pois vamos ter que resolver um sistema como esse em cada passo de tempo.

O que fazer? Bom, alguns meses atrás eu diria que você teria que mudar para o **Fortran** ou para o **C++**. No entanto, descobri recentemente que existem várias maneiras de acelerar o código em Python e deixá-lo pelo menos com um tempo de execução próximo a essas linguagens mais rápidas.

Algumas opções para fazer isso é usando uma das bibliotecas abaixo:

1. Numpy
2. Cython
3. Pythran
4. Numba

Você pode substituir os *loops* por funções do Numpy. Não conheço muito bem nem o Cython e nem o Pythran. O problema é que em todas essas 3 opções você tem que fazer modificações significativas no seu código (se você estiver implementando da forma como estamos implementando).

Já a biblioteca **Numba** permite que você deixe o código praticamente como o original. É necessário apenas que você coloque a parte mais demorada dentro de uma **função**. Mais alguns detalhes são necessários para otimizar o processo, mas basicamente é isso.

Abaixo vou implementar os códigos dos algoritmos 8 e 10 usando o Numba.

10.1 Otimizando o Código do Algoritmo 8 com Numba

Código original:

[11]:
```
import timeit

start = timeit.default_timer()

N = 100
tol = 1.e-5
T = np.zeros((N+1,N+1),float)
```

```
omega = 1.9

for i in range(N+1):
    T[i,0] = 0.0
    T[i,N] = 0.0

for j in range(1,N):
    T[0,j] = 0.0
    T[N,j] = 1.0

R = np.zeros((N+1,N+1),float)

iter = 0

while True:
    for i in range(1,N):
        for j in range(1,N):
            R[i,j] = 0.25*(T[i+1,j] + T[i-1,j] - 4.0*T[i,j] + T[i,j+1] +␣
 ↪T[i,j-1])
            T[i,j] = T[i,j] + omega*R[i,j]
    iter += 1
    #print(iter,T)
    if np.max(np.abs(R)) < tol: break
print(iter,T)

end = timeit.default_timer()

print(f'\nO custo computacional foi de {end-start:6.4f} s.')
```

```
330 [[0.00000000e+00 0.00000000e+00 0.00000000e+00 … 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.07219781e-04 2.14374719e-04 … 2.17143725e-04
  1.08636261e-04 0.00000000e+00]
 [0.00000000e+00 2.14600465e-04 4.29070046e-04 … 4.34548042e-04
  2.17402722e-04 0.00000000e+00]
 …
 [0.00000000e+00 3.02126855e-01 4.99559063e-01 … 4.99561081e-01
  3.02127887e-01 0.00000000e+00]
 [0.00000000e+00 4.99889765e-01 6.97432206e-01 … 6.97433205e-01
  4.99890276e-01 0.00000000e+00]
 [0.00000000e+00 1.00000000e+00 1.00000000e+00 … 1.00000000e+00
  1.00000000e+00 0.00000000e+00]]

O custo computacional foi de 6.0957 s.
```

Código com Numba:

[12]:
```
from numba import jit, njit

start = timeit.default_timer()

N = 100
tol = 1.e-5
T = np.zeros((N+1,N+1),float)
omega = 1.9

# Estou declarando a matriz R
# fora da função, pois o numba
# fica mais rápido dessa forma.
R = np.zeros((N+1,N+1),float)

# Aqui temos a função e antes dela
# colocamos um decorador. Pode ser
# o jit ou o njit. O njit é mais
# restrito mas funciona de verdade.
#@jit

@njit
def resolver_calor_2D_Perm(N,tol,omega,R,T):

    for i in range(N+1):
        T[i,0] = 0.0
        T[i,N] = 0.0

    for j in range(1,N):
        T[0,j] = 0.0
        T[N,j] = 1.0

    iter = 0

    while True:
        for i in range(1,N):
            for j in range(1,N):
                R[i,j] = 0.25*(T[i+1,j] + T[i-1,j] - 4.0*T[i,j] + T[i,j+1] +␣
 ↪T[i,j-1])
                T[i,j] = T[i,j] + omega*R[i,j]
        iter += 1
        #print(iter,T)
        if np.max(np.abs(R)) < tol: break
    print(iter,T)

    return T

T = resolver_calor_2D_Perm(N,tol,omega,R,T)
```

```
end = timeit.default_timer()

print(f'\nO custo computacional foi de {end-start:6.4f} s.')
```

```
330 [[0.00000000e+00 0.00000000e+00 0.00000000e+00 … 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.07219781e-04 2.14374719e-04 … 2.17143725e-04
  1.08636261e-04 0.00000000e+00]
 [0.00000000e+00 2.14600465e-04 4.29070046e-04 … 4.34548042e-04
  2.17402722e-04 0.00000000e+00]
 …
 [0.00000000e+00 3.02126855e-01 4.99559063e-01 … 4.99561081e-01
  3.02127887e-01 0.00000000e+00]
 [0.00000000e+00 4.99889765e-01 6.97432206e-01 … 6.97433205e-01
  4.99890276e-01 0.00000000e+00]
 [0.00000000e+00 1.00000000e+00 1.00000000e+00 … 1.00000000e+00
  1.00000000e+00 0.00000000e+00]]

O custo computacional foi de 0.8072 s.
```

Atenção: primeiro teste o seu código em o Numba. Coloque a parte mais lenta em uma função e teste o código. Se tudo estiver funcionando bem, aí sim você colocar o decorador @njit antes da função.

10.2 Otimizando o Código do Algoritmo 10 com Numba

Código original:

[13]:
```
start = timeit.default_timer()

N = 50
tol = 1.e-5
T = np.zeros((N+1,N+1),float)
omega = 1.9
t_final = 1.0
dt = 0.01

for i in range(N+1):
    T[i,0] = 0.0
    T[i,N] = 0.0

for j in range(1,N):
    T[0,j] = 0.0
    T[N,j] = 1.0

dx = 1.0/N
lamda = dt/(dx*dx)
```

```
R = np.zeros((N+1,N+1),float)

T_novo = np.copy(T)

t = 0.0

while True:
    iter = 0
    while True:
        for i in range(1,N):
            for j in range(1,N):
                a1 = (1.0 + 4.0*lamda)
                R[i,j] = (1.0/a1)*(T[i,j] +
                        lamda*(T_novo[i+1,j]+T_novo[i-1,j] +␣
 ↪T_novo[i,j+1]+T_novo[i,j-1])
                        - a1*T_novo[i,j])
                T_novo[i,j] = T_novo[i,j] + omega*R[i,j]
        iter += 1
        if np.max(np.abs(R)) < tol: break
    T = np.copy(T_novo)
    t += dt
    #print(t, iter, T)
    if t >= t_final-0.1*dt: break

print(t, iter, T)

end = timeit.default_timer()

print(f'\nO custo computacional foi de {end-start:6.4f} s.')
```

```
1.0000000000000007 1 [[0.00000000e+00 0.00000000e+00 0.00000000e+00 …
0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 4.37824488e-04 8.73850242e-04 … 8.73993797e-04
  4.37900923e-04 0.00000000e+00]
 [0.00000000e+00 8.77464890e-04 1.75132155e-03 … 1.75159379e-03
  8.77609900e-04 0.00000000e+00]
 …
 [0.00000000e+00 3.01472532e-01 4.98248356e-01 … 4.98248379e-01
  3.01472544e-01 0.00000000e+00]
 [0.00000000e+00 4.99562087e-01 6.96775814e-01 … 6.96775825e-01
  4.99562092e-01 0.00000000e+00]
 [0.00000000e+00 1.00000000e+00 1.00000000e+00 … 1.00000000e+00
  1.00000000e+00 0.00000000e+00]]

O custo computacional foi de 9.9196 s.
```

Código com Numba:

[14]:
```
start = timeit.default_timer()

N = 50
tol = 1.e-5
T = np.zeros((N+1,N+1),float)
omega = 1.9
t_final = 1.0
dt = 0.01

for i in range(N+1):
    T[i,0] = 0.0
    T[i,N] = 0.0

for j in range(1,N):
    T[0,j] = 0.0
    T[N,j] = 1.0

dx = 1.0/N
lamda = dt/(dx*dx)

R = np.zeros((N+1,N+1),float)

T_novo = np.copy(T)

t = 0.0

#@jit
@njit
def resolver_calor_2D_Tran(N,tol,omega,dt,t_final,t,lamda,R,T_novo,T):

    while True:
        iter = 0
        while True:
            for i in range(1,N):
                for j in range(1,N):
                    a1 = (1.0 + 4.0*lamda)
                    R[i,j] = (1.0/a1)*(T[i,j] +
                            lamda*(T_novo[i+1,j]+T_novo[i-1,j] +␣
 ↪T_novo[i,j+1]+T_novo[i,j-1])
                            - a1*T_novo[i,j])
                    T_novo[i,j] = T_novo[i,j] + omega*R[i,j]
            iter += 1
            if np.max(np.abs(R)) < tol: break
        T = np.copy(T_novo)
        t += dt
```

```
        #print(t, iter, T)
        if t >= t_final-0.1*dt: break
    print(t, iter, T)
    return T

T = resolver_calor_2D_Tran(N,tol,omega,dt,t_final,t,lamda,R,T_novo,T)

end = timeit.default_timer()

print(f'\nO custo computacional foi de {end-start:6.4f} s.')
```

```
1.0000000000000007 1 [[0.00000000e+00 0.00000000e+00 0.00000000e+00 …
0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 4.37824488e-04 8.73850242e-04 … 8.73993797e-04
  4.37900923e-04 0.00000000e+00]
 [0.00000000e+00 8.77464890e-04 1.75132155e-03 … 1.75159379e-03
  8.77609900e-04 0.00000000e+00]
 …
 [0.00000000e+00 3.01472532e-01 4.98248356e-01 … 4.98248379e-01
  3.01472544e-01 0.00000000e+00]
 [0.00000000e+00 4.99562087e-01 6.96775814e-01 … 6.96775825e-01
  4.99562092e-01 0.00000000e+00]
 [0.00000000e+00 1.00000000e+00 1.00000000e+00 … 1.00000000e+00
  1.00000000e+00 0.00000000e+00]]

O custo computacional foi de 0.3338 s.
```