

(Universidade de São Paulo)  
Trabalho 01 - Temperaturas no Grafo de Manhattan

Métodos do Cálculo Numérico I – SME0205

Docente: Antonio Castelo Filho

Julia Guazzelli Monteiro – 15465383

Vinícius de Sá Ferreira – 15491650

21 abr. 2025

## 1 Modelagem do problema

Suponha  $T : \mathbb{N} \times (\mathbb{R} \setminus \mathbb{R}^-) \rightarrow \mathbb{R}$  definida por  $T(x, t) = \text{temperatura no ponto } x \text{ ao tempo } t$ . Temos a equação do calor em 1D da seguinte forma

$$\frac{\partial T(x, t)}{\partial t} = \alpha \frac{\partial^2 T(x, t)}{\partial x^2}$$

onde  $\alpha > 0$  é a difusão térmica. Assim podemos fazer a aproximação por série de Taylor para um  $t_0$  fixo, ao redor de  $x$

$$\begin{aligned} T(x+1) &= \sum_{j=0}^{+\infty} \frac{T^{(j)}(x)(x+1-x)^j}{j!} = \sum_{j=0}^{+\infty} \frac{T^{(j)}(x)}{j!} = T(x) + T'(x) + \frac{T''(x)}{2} + \frac{T^{(3)}(x)}{6} + \frac{T^{(4)}(x)}{24} + \dots \\ T(x-1) &= \sum_{j=0}^{+\infty} \frac{T^{(j)}(x)(x-1-x)^j}{j!} = \sum_{j=0}^{+\infty} (-1)^j \frac{T^{(j)}(x)}{j!} = T(x) - T'(x) + \frac{T''(x)}{2} - \frac{T^{(3)}(x)}{6} + \frac{T^{(4)}(x)}{24} - \dots \\ T(x+1) + T(x-1) &= 2T(x) + T''(x) + \frac{T^{(4)}(x)}{12} + \dots \\ T''(x) &= T(x+1) - 2T(x) + T(x-1) - \frac{T^{(4)}(x)}{12} - \dots \\ \therefore T''(x) &= T(x+1) - 2T(x) + T(x-1) - \mathcal{O}(1) \end{aligned}$$

logo se definirmos  $T_i(t) := T(t, i)$ , teremos

$$\frac{dT_i(t)}{dt} = \alpha(T_{i+1}(t) - 2T_i(t) + T_{i-1}(t))$$

Portanto, se tivermos  $T(t) = \begin{bmatrix} T_1(t) \\ T_2(t) \\ T_3(t) \\ \vdots \\ T_n(t) \end{bmatrix}$ , podemos reescrever a equação em função do Laplaciano  $L$  do grafo, dado por  $L = G - A$ , com  $G$  a matriz diagonal com o grau de arestas para cada vértice e  $A$  a matriz

de adjacência. Note que

$$LT = \left( \begin{bmatrix} G_1 & 0 & 0 & \cdots & 0 \\ 0 & G_2 & 0 & \cdots & 0 \\ 0 & 0 & G_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & G_n \end{bmatrix} - \begin{bmatrix} & & & & \\ & & & & \\ & & A & & \\ & & & & \\ & & & & \end{bmatrix}_{n \times n} \right) \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ \vdots \\ T_n \end{bmatrix}_{n \times 1} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ \vdots \\ C_n \end{bmatrix}_{n \times 1}$$

em que, para  $i = 1, 2, 3, \dots, n$  teremos

$$C_i = G_i T_i - \sum_{j \rightarrow i} T_j$$

onde  $j \rightarrow i$  indica os vizinhos de  $i$ , aqueles que estão conectados com  $i$ . No caso 1D, o grau de um ponto  $i$  interno é de 2 e possui 2 pontos conexos, o sucessor e o antecessor, logo teremos a equação

$$\frac{dT_i(t)}{dt} = \alpha(T_{i+1}(t) - 2T_i(t) + T_{i-1}(t)) = -\alpha(2T_i(t) - T_{i-1}(t) - T_{i+1}(t)) = -\alpha(G_i T_i(t) - \sum_{j \rightarrow i} T_j(t)) = -\alpha(LT)_i$$

Portanto, temos

$$\frac{dT}{dt} = -\alpha LT \quad (*)$$

Uma forma de estender  $T(t)$  para  $\mathbb{R}$  é com  $T(t) = e^{(-\alpha Lt)}T(0)$ , onde  $T(0) = T_0$ . Assim

$$\frac{dT}{dt} = \frac{d}{dt}[e^{(-\alpha Lt)}T(0)] = -\alpha L e^{(-\alpha Lt)}T(0) = -\alpha L(e^{(-\alpha Lt)}T(0)) = -\alpha LT(t)$$

que satisfaz (\*). Por fim, se fizermos  $\lim_{t \rightarrow +\infty} T(t) = T_\infty = \begin{bmatrix} c \\ c \\ c \\ \vdots \\ c \end{bmatrix}$ ,  $c \in \mathbb{R}$ , logo  $\frac{dT}{dt} = 0$  e a equação fica da seguinte forma

$$\frac{dT}{dt} = -\alpha LT_\infty = \vec{0}$$

$$LT_\infty = \vec{0}$$

$$\begin{aligned} T_{i+1}(t) - 2T_i(t) + T_{i-1}(t) &= 0 \\ \therefore T_i(t) &= \frac{T_{i-1}(t) + T_{i+1}(t)}{2} \end{aligned}$$

Agora, podemos adicionar condições de contorno, então teremos para  $b$  o vetor com a temperatura em alguns pontos

$$\frac{dT}{dt} = -\alpha LT + b = 0$$

$$\alpha LT = b$$

Com a matriz de penalidades, ajustamos a contribuição resultando em

$$(L + P)T = Pb$$

## 2 Métodos diretos

### 2.1 Método de Cholesky

Suponhamos que  $M$  seja uma matriz triangular simples, mais precisamente, uma matriz pequena, triangular e bem-comportada, então, pela teoria, achar na  $M^{-1}$  explicitamente não seria algo muito complexo. Exemplo:

$$\begin{bmatrix} 2 & 1 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 4 \end{bmatrix}$$

Acharemos  $U^{-1}$  usando  $[U|I] \rightarrow [I|U^{-1}]$ :

$$\begin{aligned} [U|I] &= \left[ \begin{array}{ccc|ccc} 2 & 1 & 3 & 1 & 0 & 0 \\ 0 & 1 & 2 & 0 & 1 & 0 \\ 0 & 0 & 4 & 0 & 0 & 1 \end{array} \right] \rightarrow \left[ \begin{array}{ccc|ccc} 1 & 1/2 & 3/2 & 1/2 & 0 & 0 \\ 0 & 1 & 2 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1/4 \end{array} \right] \\ \left[ \begin{array}{ccc|ccc} 1 & 1/2 & 0 & 1/2 & 0 & -3/8 \\ 0 & 1 & 0 & 0 & 1 & -1/2 \\ 0 & 0 & 1 & 0 & 0 & 1/4 \end{array} \right] &\rightarrow \left[ \begin{array}{ccc|ccc} 1 & 0 & 0 & 1/2 & -1/2 & -1/8 \\ 0 & 1 & 0 & 0 & 1 & -1/2 \\ 0 & 0 & 1 & 0 & 0 & 1/4 \end{array} \right] \\ U^{-1} &= \left[ \begin{array}{ccc} 1/2 & -1/2 & -1/8 \\ 0 & 1 & -1/2 \\ 0 & 0 & 1/4 \end{array} \right] \end{aligned}$$

Mas se tivermos uma matriz geral (não necessariamente triangular), mas invertível, de entradas significativamente grandes (com linhas, colunas na casa dos milhares), então o número de operações para computar a inversa explicitamente pelo algoritmo de Gauss-Jordan (demonstrado de forma simplificada no exemplo acima com a matriz triangular) cresceria cubicamente com o tamanho  $N$  da matriz, ou seja, seria  $O(N^3)$  no que tange à análise assintótica. Pois, em termos simples, há  $O(N)$  passos principais (um para cada coluna/pivô) na eliminação, e em cada passo, se opera em  $O(N)$  linhas abaixo/acima do pivô, afetando  $O(N)$  elementos em cada linha - as colunas à direita. A complexidade vem de algo como  $O(N \text{ passos}) * O(N \text{ linhas afetadas}) * O(N \text{ operações por elemento}) = O(N^3)$ . Para o computador, calcular a inversa explicitamente é, portanto, custoso. Para o humano, bem, deixaremos a discussão de lado.

Para contra-atacar o alto custo da complexidade computacional na resolução do sistema  $Mx = k$ , assumindo que a matriz seja simétrica e def. positiva, podemos pensar em fatorizar a matriz em questão em  $M = LL^T$ , onde  $L$  é uma matriz triangular inferior (e  $L^T$  é triangular superior). A própria fatorização de Cholesky tem um custo da ordem de  $O(N^3)$  (embora geralmente com uma constante menor que a da inversão). Mas a vantagem crucial vem a seguir: em vez de calcular  $M^{-1}$ , resolvemos o sistema original  $(LL^T)x = k$  em duas etapas, introduzindo  $y = L^Tx$ :

1. Resolver  $Ly = k$  para  $y$  usando substituição **direta**.
2. Resolver  $L^Tx = y$  para  $x$  usando substituição **retroativa**.

Cada uma dessas etapas de substituição tem um custo muito menor, da ordem de  $O(N^2)$ . Portanto, após o custo inicial da fatorização ( $O(N^3)$ ), a solução do sistema em si é muito menos complexa ( $O(N^2)$ ) do que calcular a inversa completa ( $O(N^3)$ ) e depois multiplicar por  $k$  ( $O(N^2)$ ). É nisso que o algoritmo de Cholesky se baseia para ser eficiente na resolução de sistemas lineares.

#### 2.1.1 Conexão com o código

A fatoração da decomposição da matriz  $M$  no produto de uma matriz triangular inferior  $L$  e sua transposta  $L^T$  é realizada pela função abaixo no código.

```

    # Calcula a decomp. de Cholesky: M = L*L^T
151 L = la.cholesky(M, lower=True)
    
```

Depois de obter  $L$ , a solução é encontrada resolvendo dois sistemas triangulares sequencialmente e retornando o vetor  $x$ :

```

8   # Calcula a decomp. de Cholesky: M = L*L^T
7   L = la.cholesky(M, lower=True)
6
5   # Resolve a matriz triangular L*y = c
4   y = la.solve_triangular(L, c, lower=True)
3
2   # Resolve a transposta da matriz triâng L^T*x = y
1   x = la.solve_triangular(L.T, y, lower=False)
158
1   return x
2
NORMAL main.py

```

unix | utf-8 | python 61% 158:0

Embora a etapa de fatoração ainda domine a complexidade ( $O(N^3)$ ), a constante multiplicativa é menor do que em métodos como LU ou QR para matrizes densas, o cálculo de comparação do tempo de execução será discutido mais posteriormente.

## 2.2 Método LU

Fazendo uma breve comparação com o método de Cholesky, o algoritmo LU não requer que a matriz  $M$  seja simétrica ou definida positiva, basta que ela seja quadrada e invertível - isso generaliza as aplicações de uma forma significativa.

Porém mesmo para matrizes quadradas e invertíveis com entradas muito grandes, calcular a inversa  $M^{-1}$  diretamente (explicitamente) ou usar métodos muito básicos tende a ser computacionalmente caro (complexidade  $O(N^3)$ ) e propenso a erros numéricos.

A saída seria usar a decomposição LU que se consiste em fatorar a matriz  $M$  de forma a simplificar a resolução do sistema. E para evitar divisões por zero ou números muito pequenos, o processo pode incluir pivotamento, que troca as linhas da matriz durante a fatoração. Em suma, isso resulta na decomposição:

$$M = PLU$$

onde:

- $P$  é uma matriz de permutação;
- $L$  é uma matriz triangular inferior;
- $U$  é uma matriz triangular superior.

### 2.2.1 Resolução do Sistema com Decomposição LU

Com essa fatoração, o sistema  $MT = c$  se torna  $PLUT = c$ . Podemos resolver isso em duas etapas, envolvendo sistemas triangulares, que já sabemos que são mais fáceis e rápidos de resolver (complexidade  $O(N^2)$  cada):

1. Resolver  $Ly = Pc$  para achar um vetor intermediário  $y$ . (Note que aplicamos a permutação  $P$  ao vetor  $c$ ).
2. Resolver  $UT = y$  para achar a solução final  $T$

A etapa mais custosa ainda é a fatoração  $M = PLU$ , que tem complexidade  $O(N^3)$  (aproximadamente  $\frac{2}{3}N^3$  operações para matrizes densas). Em teoria, esse método tem a mesma ordem de complexidade de Cholesky, mas na prática LU geralmente requer mais operações por não explorar a simetria da matriz (quando presente).

### 2.2.2 Conexão com o código

Fatoração da matriz  $M$  no produto  $PLU$ :

```
2 def funcao_lu(M,c):
1 •   #Calculando a decomposicao LU: M= P*L*U, onde P eh uma matriz de permutacao
164 •   P,L,U = la.lu(M)
1
```

Ao obter  $P$ ,  $L$ , e  $U$ , encontramos  $T$  resolvendo os dois sistemas triangulares sequencialmente, como implementado na função `funcao_lu`:

```
2 •   #Resolvendo Ly=Pc
3 •   Pc= P @ c
4 •   y = la.solve_triangular(L, Pc, lower=True)
5
6 •   #Resolve Ux=y
7 •   x = la.solve_triangular(U, y, lower=False)
8
9 •   return x
10
COMMAND main.py | +
/la.lu
```

Parent 3730, Main T  
device-id: 'glib'  
irefox:3730): Glib  
as no property name0  
Parent 3730, Main T  
device-id: 'glib'

unix | utf-8 | python 63% 164:10 .b  
as no property name0  
Parent 3730, Main T

## 2.3 Método QR

De praxe, a decomposição QR é outro método fundamental para resolver sistemas lineares, porém com uma excelente estabilidade numérica, sendo particularmente robusto mesmo para matrizes mal-condicionadas. O método é aplicável a matrizes quadradas invertíveis gerais e não requer simetria ou definição positiva, também é a base para muitos algoritmos de autovalores e problemas de mínimos quadrados.

A ideia central da decomposição QR é fatorar a matriz  $M$  no produto de uma matriz ortogonal  $Q$  e uma matriz triangular superior  $R$ :

$$M = QR$$

onde:

- $Q$  é uma matriz ortogonal. Isso significa que suas colunas formam um conjunto orthonormal de vetores, e sua transposta é comutativa à sua inversa ( $Q^T Q = Q Q^T = I$ , onde  $I$  é a matriz identidade). Multiplicar por  $Q$  ou  $Q^T$  preserva normas e ângulos, o que contribui para a estabilidade numérica.
- $R$  é uma matriz triangular superior.

Com a fatoração  $M = QR$ , o sistema original  $MT = c$  se torna  $QRT = c$ . Para isolar  $T$ , multiplicamos ambos os lados pela transposta (que é a inversa) de  $Q$ :

$$\begin{aligned} Q^T(QRT) &= Q^T c \\ (Q^T Q)RT &= Q^T c \\ IRT &= Q^T c \\ RT &= Q^T c \end{aligned}$$

Isso resulta em um sistema triangular superior  $RT = Q^T c$ , que pode ser resolvido eficientemente por substituição retroativa (complexidade  $O(N^2)$ ).

A etapa computacionalmente mais intensiva é a própria fatoração  $M = QR$ . Existem diferentes algoritmos para computá-la (como Gram-Schmidt modificado, transformações de Householder ou rotações de Givens). Para matrizes densas, a complexidade da fatoração QR é geralmente  $O(N^3)$ , tipicamente com uma constante maior do que a da decomposição LU (aproximadamente  $\frac{4}{3}N^3$  operações usando Householder). Portanto, embora seja muito estável, tende a ser mais custoso que LU ou Cholesky (quando aplicável) para a simples solução de sistemas lineares quadrados.

### 2.3.1 Conexão com o código

Fatoração da matriz  $M$  no produto  $QR$ :

```
4
3 def funcao_qr(M, c):
2
1 •    #Calculando a decomp. QR: M=QR
178 •    Q, R = la.qr(M)
1
```

Após obter  $Q$  e  $R$ , a solução  $T$  é encontrada calculando  $Q^T c$  e depois resolvendo o sistema triangular  $RT = Q^T c$ , como implementado na função `funcao_qr`:

```
7
6 •    #Resolvendo Rx=Q.Tc
5 •    Qtc = Q.T @ c
4 •    x = la.solve_triangular(R, Qtc, lower=False)
3
2
1 •    return x
184
```

## 2.4 Resultados

Os tempos de execução registrados para cada método foram plotados no gráfico de barras, observada na [Figura 5](#). Ao fazer a análise, observa-se que, para a matriz  $M$  específica deste problema, que é simétrica e definida positiva (se trata de um grafo), o método de Cholesky apresentou o melhor desempenho, sendo o mais rápido entre as implementações baseadas em decomposição.

Isso está de acordo com a teoria e o que foi estudado nas análises dos métodos que conduzimos, pois Cholesky explora a simetria da matriz, resultando em aproximadamente metade das operações de ponto flutuante em comparação com a decomposição LU para matrizes densas (aproximadamente  $\frac{1}{3}N^3$  vs  $\frac{2}{3}N^3$  operações para a fatoração). A função `numpy.linalg.solve`, sendo uma aplicação altamente otimizada (frequentemente baseada em implementações LAPACK), apresentou um desempenho muito competitivo, potencialmente detectando a natureza da matriz e utilizando um solver apropriado internamente.

Além do mais, no que tange à análise minuciosa, a decomposição LU foi mais lenta que Cholesky, como esperado, por ser um método mais geral. A decomposição QR, embora robusta numericamente, tende a ser a mais custosa computacionalmente para a solução de sistemas lineares (com uma constante associada à complexidade  $O(N^3)$  tipicamente maior, como  $\frac{4}{3}N^3$  para métodos baseados em Householder), o que se refletiu em seu tempo de execução sendo o mais elevado entre os quatro métodos testados.

Em suma, para o sistema linear derivado deste problema específico, a decomposição de Cholesky é a escolha mais eficiente em termos de tempo de execução, devido às propriedades favoráveis da matriz  $M$ . A função genérica `numpy.linalg.solve` também se mostrou extremamente eficaz. A escolha entre LU e QR dependeria de um balanço entre generalidade, custo computacional e requisitos de estabilidade numérica, sendo QR preferível em casos de matrizes mal-condicionadas, apesar de seu maior custo.

### 3 Métodos iterativos

#### 3.1 Gauss-Jacobi

Queremos resolver o sistema  $Mx = c$ , em que  $M = (L + P)T$  e  $c = Pb$ . Assim, temos

$$\begin{aligned} Mx &= c \\ (M - D + D)x &= c \\ (M - D)x + Dx &= c \\ (M - D)x^{(k)} + Dx^{(k+1)} &= c \\ Dx^{(k+1)} &= (D - M)x^{(k)} + c \\ x^{(k+1)} &= D^{-1}(D - M)x^{(k)} + D^{-1}c \\ x^{(k+1)} &= (I - D^{-1}M)x^{(k)} + D^{-1}c \\ x^{(k+1)} &= Cx^{(k)} + g \end{aligned}$$

Como  $D$  é a diagonal de  $M$ , o determinante é o produtório da diagonal e  $M$  que é não-nulo, então  $D$  é inversível. O método convergiu em  $k = 1280$  iterações e demorou uma média de  $3\text{min}$  para uma tolerância de  $\pm 10^{-3}$ .

#### 3.2 Gauss-Seidel

Queremos resolver o sistema  $Mx = c$ , em que  $M = (L + P)T$  e  $c = Pb$ . Assim, façamos  $M = L + R$ , onde  $L$  é a matriz triangular inferior de  $M$  e  $R$ , a triangular superior sem a diagonal, então

$$\begin{aligned} Mx &= c \\ (L + R)x &= c \\ Lx + Rx &= c \\ Lx^{(k+1)} + Rx^{(k)} &= c \\ Lx^{(k+1)} &= -Rx^{(k)} + c \\ x^{(k+1)} &= (-L^{-1}R)x^{(k)} + L^{-1}c \\ x^{(k+1)} &= Cx^{(k)} + g \end{aligned}$$

Como  $L$  é triangular inferior, o determinante é o produtório da diagonal de  $M$  que é não-nulo, então  $L$  é inversível. O método convergiu em  $k = 657$  iterações e demorou uma média de  $2\text{min}$  para uma tolerância de  $\pm 10^{-3}$ .

#### 3.3 Gradientes conjugados

Queremos resolver o sistema  $Mx = c$ , em que  $M = (L + P)T$  e  $c = Pb$ . Considere a função

$$f(x) = \frac{1}{2}x^T Mx - c^T x$$

onde  $M$  é simétrica positiva definida (SPD). Note que

$$\begin{aligned} f(x) &= \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n m_{ij} x_i x_j - \sum_{i=1}^n c_i x_i \\ \frac{\partial f(x)}{\partial x_k} &= \frac{1}{2} \left( \sum_{i=1}^n \sum_{j=1}^n m_{ij} (x_i \delta_{ik} + x_j \delta_{jk}) \right) - c_k = \frac{1}{2} \left( \sum_{i=1}^n m_{ik} x_i + \sum_{j=1}^n m_{kj} x_j \right) - c_k \\ &= \frac{1}{2} \left( \sum_{j=1}^n m_{kj} x_j \right) - c_k = \sum_{j=1}^n m_{kj} x_j - c_k \\ \therefore \nabla f(x) &= Mx - c \end{aligned}$$

Assim, queremos encontrar  $x$  (que é único, porque  $M$  é SPD) tal que  $\nabla f(x) = 0 \iff Mx - c = 0 \iff Mx = c$ .

Começamos com um chute inicial  $x_0$ , então teremos um resíduo  $r_0 = c - Mx_0 = -\nabla f(x_0)$ . Teremos a direção inicial dada por  $p_0 = r_0$ , assim atualizamos

$$x_1 = x_0 + \alpha p_0$$

Temos que  $r_1^T \cdot r_0 = 0$ , porque são ortogonais, e  $r_1 = c - Mx_1$ , então

$$\begin{aligned} r_1^T \cdot r_0 &= 0 \\ (c - Mx_1)^T \cdot r_0 &= 0 \\ [c - M(x_0 + \alpha p_0)]^T \cdot r_0 &= 0 \\ (c - Mx_0)^T \cdot r_0 - \alpha(Mp_0)^T \cdot r_0 &= 0 \\ (c - Mx_0)^T \cdot r_0 - \alpha(Mp_0)^T \cdot p_0 &= 0 \\ (c - Mx_0)^T \cdot r_0 &= \alpha(Mp_0)^T \cdot p_0 \\ \alpha &= \frac{(c - Mx_0)^T \cdot r_0}{(Mp_0)^T \cdot p_0} \\ \alpha &= \frac{r_0^T \cdot r_0}{p_0^T \cdot M \cdot p_0} \end{aligned}$$

Agora fazemos  $r_1 = r_0 - \alpha(Mp_0)$  e a direção deve ser atualizada de forma conjugada, isto é,  $p_{k+1}^T \cdot M \cdot p_k = 0$ . Então podemos fazer

$$\begin{aligned} p_1 &= r_1 + \beta p_0 \\ p_1^T \cdot M \cdot p_0 &= 0 \\ (r_1 + \beta p_0)^T \cdot M \cdot p_0 &= 0 \\ r_1^T \cdot M \cdot p_0 + \beta(p_0^T \cdot M \cdot p_0) &= 0 \\ \beta(p_0^T \cdot M \cdot p_0) &= -r_1^T \cdot M \cdot p_0 \\ \beta &= -\frac{r_1^T \cdot M \cdot p_0}{p_0^T \cdot M \cdot p_0} = -\frac{r_1^T \cdot M \cdot p_0}{r_0^T \cdot M \cdot p_0} \end{aligned}$$

Como  $r_1 = r_0 - \alpha(Mp_0)$ , temos  $\frac{1}{\alpha}(r_0 - r_1) = M \cdot p_0$ , então

$$\begin{aligned} \beta &= -\frac{r_1^T \cdot \left[ \frac{1}{\alpha}(r_0 - r_1) \right]}{r_0^T \cdot \left[ \frac{1}{\alpha}(r_0 - r_1) \right]} = -\frac{\cancel{r_1^T \cdot r_0} - r_1^T \cdot r_1}{\cancel{r_0^T \cdot r_0} - \cancel{r_0^T \cdot r_1}} = -\frac{-r_1^T \cdot r_1}{r_0^T \cdot r_0} \\ \beta &= \frac{r_1^T \cdot r_1}{r_0^T \cdot r_0} \end{aligned}$$

Portanto, as etapas são

1. Chute inicial  $x_0$ ;
2.  $p_0 = r_0 = c - Mx_0$ ;
3.  $\alpha = \frac{r_k^T \cdot r_k}{p_k^T \cdot M \cdot p_k}$ ;
4.  $x_{k+1} = x_k + \alpha p_k$ ;
5.  $r_{k+1} = r_k - \alpha(M \cdot p_k)$ ;
6.  $\beta = \frac{r_{k+1}^T \cdot r_{k+1}}{r_k^T \cdot r_k}$ ;
7.  $p_{k+1} = r_{k+1} + \beta p_k$ ;
8. Volta para o passo 3, enquanto  $\|r_{k+1}\| > \text{tol}$ .

Por fim, o método convergiu em  $k = 255$  iterações e demorou uma média de 30s para uma tolerância de  $\pm 10^{-3}$ .

### 3.4 Resultados

Tolerância:  $\pm 10^{-3}$ .

	Gauss-Jacobi	Gauss-Seidel	Gradientes Conjugados
$k$	1280	657	255
tempo	3min	2min	30s

É possível observar os tempos exatos de execução na Figura 7.

## 4 Resultado obtido

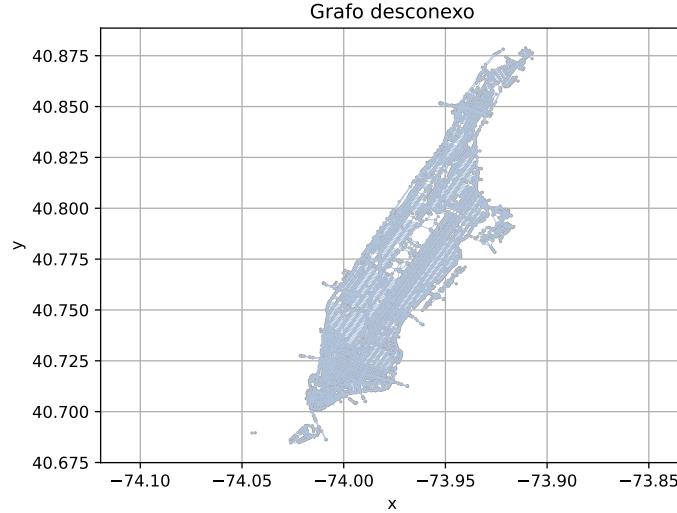


Figura 1

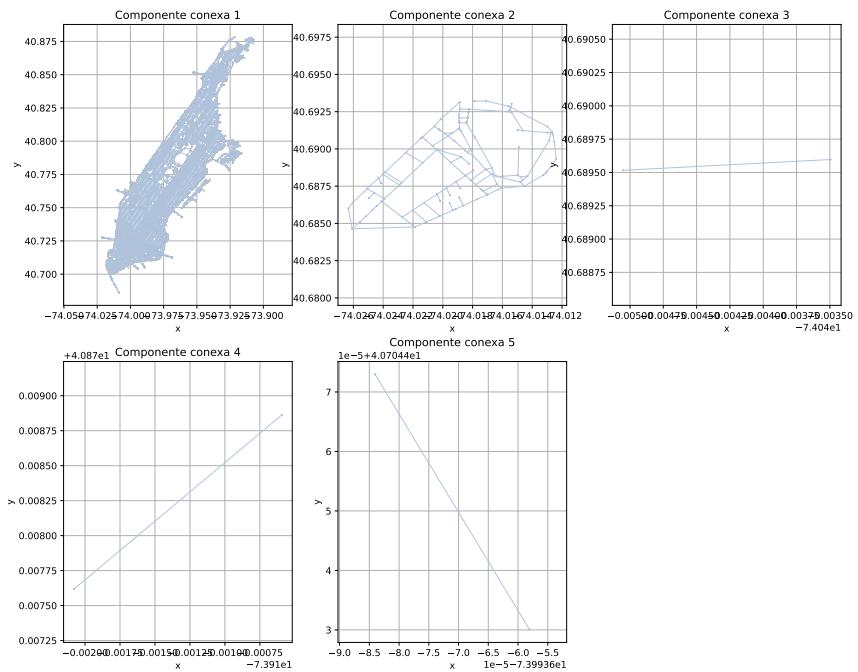


Figura 2

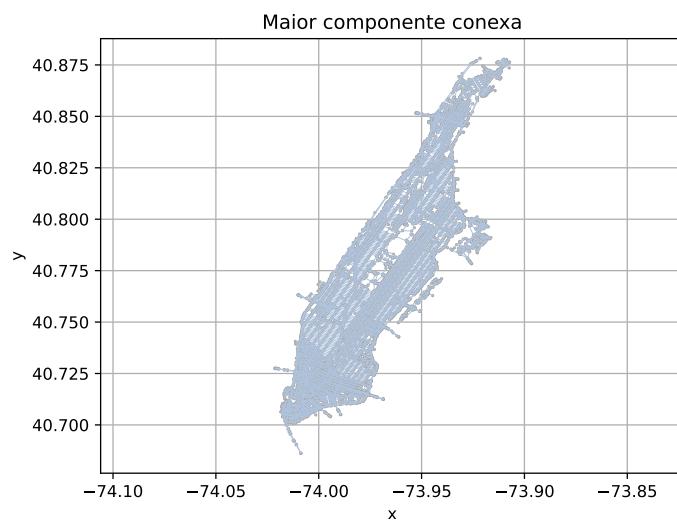


Figura 3

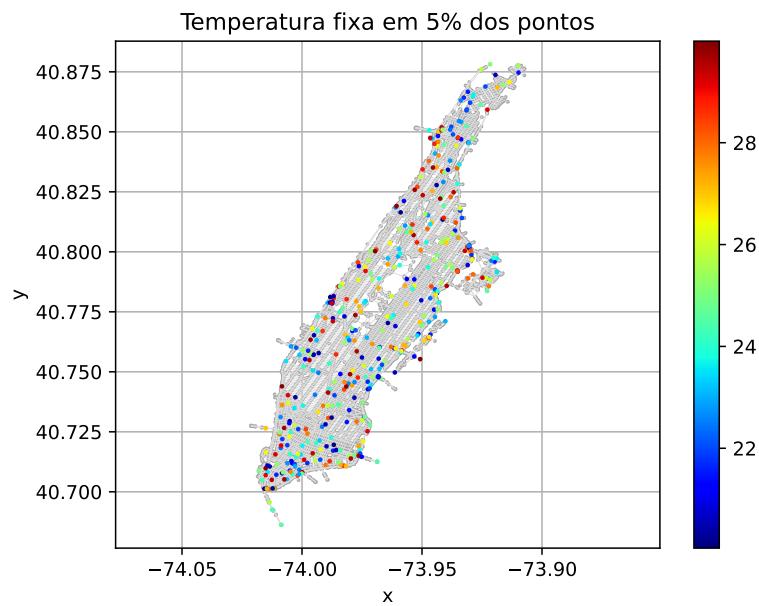


Figura 4

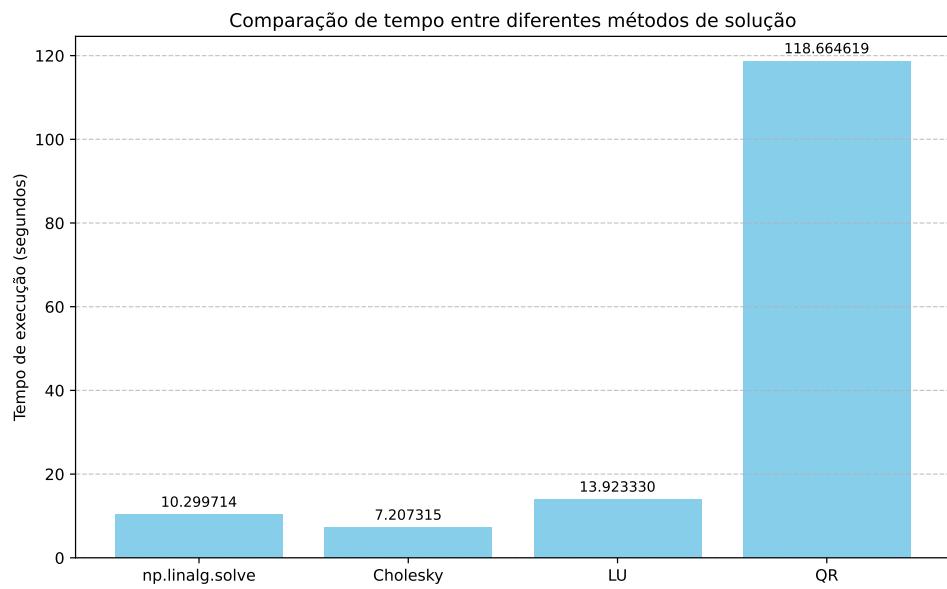


Figura 5

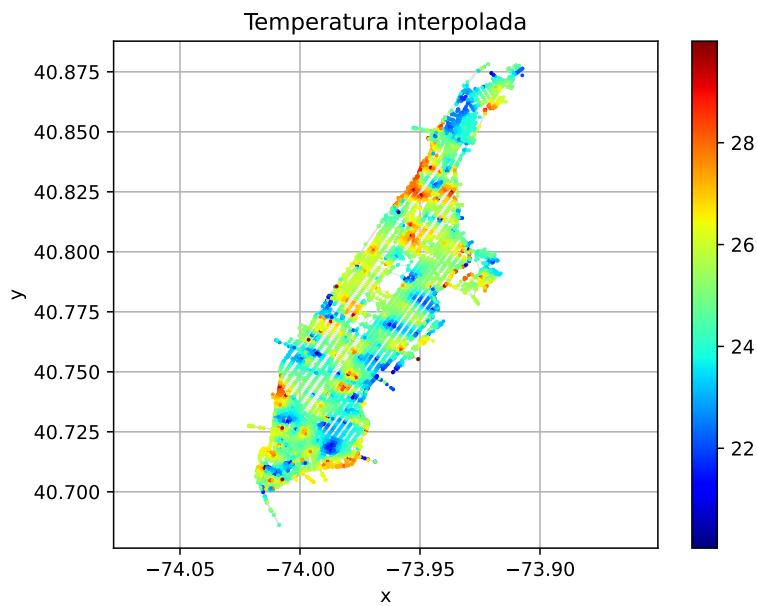


Figura 6

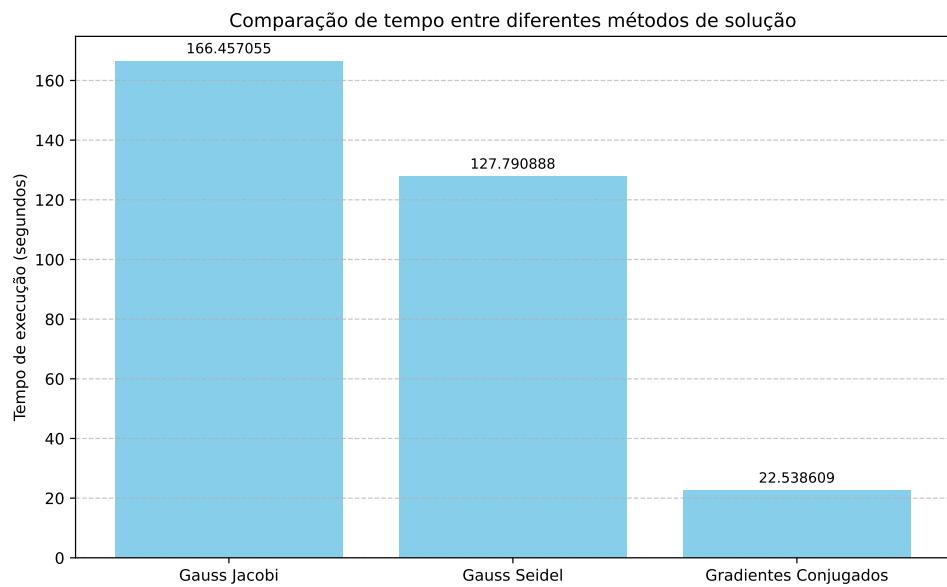


Figura 7