Calculo Numérico - Tarefa 6

Mateus Ferreira Olaso - DRE: 120056682

Fevereiro 2022

Questão 1

Essa questão e a questão 3 foi feita em conjunto com os alunos Matheus Silva e João Matheus.

i) A função "resolve_diagonal" receberá uma matriz quadrada diagonal nxn A e uma matriz coluna b de tamanho n. Ela achará os elementos da matriz coluna x que respeitam a equação $A * x = b$:

$$\begin{bmatrix} a_{1,1} & 0 & 0 & \cdots & 0 & 0 \\ 0 & a_{2,2} & 0 & \cdots & 0 & 0 \\ 0 & 0 & a_{3,3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-1,n-1} & 0 \\ 0 & 0 & 0 & \cdots & 0 & a_{n,n} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Para cada elemento de x, teremos $x_i = \frac{b_i}{a_{i,i}}$, para i $1 \leq i \leq n$. Nossa saída, então, será:

$$\begin{bmatrix} \frac{b_1}{a_{1,1}} \\ \frac{b_2}{a_{2,2}} \\ \vdots \\ \frac{b_n}{a_{n,n}} \end{bmatrix}$$

A ordem de complexidade do programa ficará na casa de n ($O(n)$), que será o número de multiplicações que devemos fazer. Na página seguinte, o código e seus exemplos:

```julia
"""
Função que resolve e retorna Ax = b, quando A é uma matriz quadrada diagonal.
Entradas: Matriz A, uma matriz quadrada diagonal; Vetor B, uma matriz coluna.
Saídas: Vetor x, cujo elementos respeitam a equação Ax = b.
Complexidade: O(n)
"""
function resolve_diagonal(A, b)
    n = length(b) # Obtém o tamanho de b e armazena em n
    x = zeros(n) # Cria um vetor x de tamanho n
    for i = 1:n
        x[i] = b[i]/A[i,i] # Para cada i, de 1 a n, faz x_1 = b_1/A_1,1
    end
    return x # Retorna o vetor x
end
```

```
resolve_diagonal
```

```julia
A = [1 0 0;
     0 2 0;
     0 0 -4]
b = [1, 1, 1]
x = resolve_diagonal(A, b)
A*x ≈ b
```

```
true
```

```julia
A = [7 0 0 0;
     0 -8 0 0;
     0 0 6 0;
     0 0 0 5]
b = [5, 14, 16, 8]
x = resolve_diagonal(A, b)
A*x ≈ b
```

```
true
```

```julia
A = [5 0;
     0 4]
b = [0, -10]
x = resolve_diagonal(A, b)
A*x ≈ b
```

```
true
```

ii) A função "resolve_triangular_superior" receberá uma matriz quadrada triangular superior nxn A e uma matriz coluna b de tamanho n. Ela achará os elementos da matriz coluna x que respeitam a equação $A * x = b$:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ 0 & 0 & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & a_{n,n} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Para cada elemento de x, teremos $x_i = \frac{b_i - \sum_{k=i+1}^{n} a_{i,k} * x_k}{a_{i,i}}$, para i $1 \leq i \leq n$. Nossa saída, então, será:

$$\begin{bmatrix} \frac{b_1 - \sum_{k=2}^{n} a_{1,k} * x_k}{a_{1,1}} \\ \frac{b_2 - \sum_{k=3}^{n} a_{2,k} * x_k}{a_{2,2}} \\ \vdots \\ \frac{b_n}{a_{n,n}} \end{bmatrix}$$

A ordem de complexidade do programa ficará na casa de n ($O(n^2)$), por percorrermos todas as linhas para formar os valores de x. Na página seguinte, o código e seus exemplos:

```julia
"""
Função que resolve e retorna Ax = b, quando A é uma matriz quadrada triangular superior.
Entradas: Matriz A, uma matriz quadrada triangular superior; Vetor B, uma matriz coluna.
Saídas: Vetor x, cujo elementos respeitam a equação Ax = b.
Complexidade: O(n^2)
"""
function resolve_triangular_superior(A, b)
    n = length(b) # Obtém o tamanho de b e armazena em n
    x = zeros(n) # Cria um vetor x de tamanho n
    for i = n:-1:1 # Faz i, que começa com valor n e diminui até 1
        x[i] = b[i] # Armazenda em x_i o valor b_i
        for j = i+1:n # Dentro do for, faz o b_i menos o somatório de A_i,j * x_j, e armazenar em x_i,
                      # Com j que começa com valor i+1 e cresce até n
            x[i] = x[i] - (A[i,j]*x[j])
        end
        x[i] = x[i]/A[i, i] # Por fim, dividir o o x_i atual por A_i,i, e armazenar em x_i
    end
    return x # Retorna o vetor x
end
```

```
resolve_triangular_superior
```

```julia
A = [1 2 3;
     0 4 5;
     0 0 6]
b = [1, 1, 1]
x = resolve_triangular_superior(A, b)
A*x ≈ b
```

```
true
```

```julia
A = [10 -2 34 5;
     0 -20 1 52;
     0 0 -5 7;
     0 0 0 3]
b = [11, 0, 10, -24]
x = resolve_triangular_superior(A, b)
A*x ≈ b
```

```
true
```

```julia
A = [-1 20;
     0 5]
b = [17, -41]
x = resolve_triangular_superior(A, b)
A*x ≈ b
```

```
true
```

$iii)$A função "resolve_triangular_inferior" receberá uma matriz quadrada triangular inferior nxn A e uma matriz coluna b de tamanho n. Ela achará os elementos da matriz coluna x que respeitam a equação $A * x = b$:

$$\begin{bmatrix} a_{1,1} & 0 & 0 & \cdots & 0 & 0 \\ a_{2,1} & a_{2,2} & 0 & \cdots & 0 & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & 0 \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Para cada elemento de x, teremos $x_i = \frac{b_i - \sum_{k=1}^{i-1} a_{i,k} * x_k}{a_{i,i}}$, para i $1 \leq i \leq n$. Nossa saída, então, será:

$$\begin{bmatrix} \frac{b_1}{a_{1,1}} \\ \frac{b_2 - \sum_{k=1}^{1} a_{2,k} * x_k}{a_{2,2}} \\ \vdots \\ \frac{b_n - \sum_{k=1}^{n-1} a_{2,k} * x_k}{a_{n,n}} \end{bmatrix}$$

A ordem de complexidade do programa ficará na casa de n ($O(n^2)$), por percorrermos todas as linhas para formar os valores de x. Na página seguinte, o código e seus exemplos:

```julia
"""
Função que resolve e retorna Ax = b, quando A é uma matriz quadrada triangular inferior.
Entradas: Matriz A, uma matriz quadrada triangular inferior; Vetor B, uma matriz coluna.
Saídas: Vetor x, cujo elementos respeitam a equação Ax = b.
Complexidade: O(n^2)
"""
function resolve_triangular_inferior(A, b)
    n = length(b) # Obtém o tamanho de b e armazena em n
    x = zeros(n) # Cria um vetor x de tamanho n
    for i = 1:n # Faz i, que começa com valor 1 e cresce até n
        x[i] = b[i] # Armazenda em x_i o valor b_i
        for j = 1:(i - 1)# Dentro do for, faz o b_i menos o somatório de A_i,j * x_j, e armazenar em x_i,
                         # Com j que começa com valor 1 e cresce até i - 1
            x[i] = x[i] - (A[i,j]*x[j])
        end
        x[i] = x[i]/A[i, i] # Por fim, dividir o o x_i atual por A_i,i, e armazenar em x_i
    end
    return x # Retorna o vetor x
end
```

```
resolve_triangular_inferior
```

```julia
A = [1 0 0;
     2 3 0;
     4 5 6]
b = [1, 1, 1]
x = resolve_triangular_inferior(A, b)
A*x ≈ b
```

```
true
```

```julia
A = [11 0 0 0;
     -2 34 0 0;
     14 55 89 0;
    -5 12 19 3]
b = [41, 17, -1, 5]
x = resolve_triangular_inferior(A, b)
A*x ≈ b
```

```
true
```

```julia
A = [-24 0 ;
     1 58]
b = [7, 62]
x = resolve_triangular_inferior(A, b)
A*x ≈ b
```

```
true
```

$iv)$A função "decomposicao_LU" tem como entrada uma matriz quadrada nxn A, e retorna duas matrizes, uma triangular inferior quadrada nxn L (que contém uma diagonal principal com todos os elementos igual a 1), e uma triangular superior quadrada nxn U, que respeita a equação $L * U = A$:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ f_{2,1} & 1 & 0 & \cdots & 0 & 0 \\ f_{3,1} & f_{3,2} & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ f_{n-1,1} & f_{n-1,2} & f_{n-1,3} & \cdots & 1 & 0 \\ f_{n,1} & f_{n,2} & f_{n,3} & \cdots & f_{n,n-1} & 1 \end{bmatrix} * \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & u_{2,2} & u_{2,3} & \cdots & u_{2,n-1} & u_{2,n} \\ 0 & 0 & u_{3,3} & \cdots & u_{3,n-1} & u_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & u_{n,n} \end{bmatrix}$$

Temos a matriz L formada por elementos $f_{i,j}$, onde $1 \leq j < i \leq n$. Esses são achados pelo valor que utilizamos para realizar a eliminação gaussiana na matriz A, que, depois de feita, é guardada em U.

O funcionamento da função acontece da seguinte forma: primeiro, é selecionado um pivô, que faz parte da diagonal principal de U (exceto o último, já que não há linhas abaixo dele para fazer a eliminação). Esse é utilizado para achar um valor que zera o elemento de mesma coluna do pivô de cada linha da matriz U (as manipulações são feitas na matriz U, uma cópia de A). Chamando esses elementos de $f_{i,j}$ em L, temos que são calculados da seguinte forma:

$f_{i,j} = \frac{U_{i,j}}{U_{j,j}}$, onde $1 \leq j < i \leq n$.

Por fim, vamos fazer a eliminação gaussiana nas linhas em U, que inicia com os mesmos valores de A. Temos, em cada linha de U:

$LinhaU_i = LinhaU_i - f_{i,j} * LinhaU_j$, onde $1 \leq j < i \leq n$.

Ao termos como pivô todos os elementos da diagonal principal de U, realizando a eliminação em todas as linhas, vamos ter em L uma matriz quadrada triangular inferior e em U uma matriz quadrada triangular superior, que quando multiplicadas, resultam em A.

A ordem de complexidade do programa ficará na casa de n ($O(n^3)$), por selecionarmos o valor da diagonal principal, percorrer as linhas faltantes e os elementos de cada linha, realizando a eliminação. Abaixo e na próxima página, o código e seus exemplos:

```
"""
Função que recebe uma matriz A e retorna duas matrizes: L, uma matriz quadrada triangular inferior,
e U, uma matriz quadrada triangular superior. Essas, quando multiplicadas, equivalem a A (L*U = A)
Entradas: Matriz A, uma matriz quadrada;
Saídas: Uma matriz quadrada triangular inferior, L; Uma matriz quadrada triangular superior, U.
Complexidade: O(n^3)
"""
function decomposicao_LU(A)
    n, = size(A) # Obtém o número de linhas e colunas em A e armazena em n
    L = zeros(n, n) # Cria uma matriz quadrada L de tamanho nxn vazia
    U = copy(A) # Copia a matriz A em U
    v = 0 # Variável auxiliar
    for i = 1:n # Preenche a diagonal principal da matriz L com o número 1
        L[i,i] = 1
    end
    for i = 1:(n - 1) # For que seleciona o 'pivo' para realiza a eliminação de Gauss, e formular valores de L
        pivo = U[i,i] # Salva o pivo retirado da matriz A
        for j = i + 1:n # Realiza o for, com j representando as linhas abaixo da linha do pivo, até n
            v = U[j, i]/pivo # Atualiza a variável auxiliar v para zerar os elementos em A de mesma coluna do pivo
            for k = 1:n # Realiza o for, passando por todos os itens da linha para realizar a eliminação gaussiana
                U[j,k] = U[j,k] - U[i,k]*(v) # Realiza a eliminação gaussiana, zerando sempre os elementos da coluna do pivo
            end
            L[j, i] = v # Armazena o valor calculado para a eliminação gaussiana
        end
    end
    return L, U # Retorna L, como uma matriz quadrada triangular inferior,
                # e U, como uma matriz quadrada triangular superior
end
```

```
A = [3.0 2.0 4.0;
     1.0 1.0 2.0;
     4.0 3.0 -2.0]
L, U = decomposicao_LU(A)
L*U ≈ A
```

```
true
```

```
A = [1.0 2.0 3.0;
     4.0 -2.0 6.0;
     1.0 4.0 5.0]
L, U = decomposicao_LU(A)
L*U ≈ A
```

```
true
```

```
A = [8.0 -4.0 -2.0;
     -4.0 10.0 -2.0;
     -2.0 -2.0 10.0]
L, U = decomposicao_LU(A)
L*U ≈ A
```

```
true
```

v) A função "inversa_LU" tem como entrada uma matriz quadrada nxn A, e retorna a sua inversa A^{-1}, que naturalmente respeita a equação $A * A^{-1} = I$:

Conseguimos achar a inversa da matriz A utilizando da decomposição LU, já que, sabendo que $L * U = A$, temos:

$A * A^(-1) = I \rightarrow L * U * A^(-1) = I$, onde tomamos $U * A^{-1} = y$. Logo,

$L * y = I$, onde, ao acharmos y, podemos solucionar A^{-1}.

A função funciona da seguinte forma: primeiro, utilizamos a função "decomposicao_LU" com entrada A, obtendo uma matriz triangular inferior quadrada L e uma matriz triangular superior quadrada U, que respeitam $L * U = A$. Então, utilizamos a função "resolve_matriz_inferior", tendo como entrada a matriz L e uma coluna i da matriz identidade, onde $1 \leq i \leq n$, obtendo a coluna i da matriz y que respeita $L * y = I$. Depois, utilizamos a função "resolve_matriz_superior", tendo como entrada a matriz U e uma coluna i da matriz y, obtendo a coluna i da matriz A^{-1}, que respeita $U * A^{-1} = y$. Ao realizar isso com as n colunas das matrizes, obtemos a matriz inversa de A, que é retornada.

A ordem de complexidade do programa ficará na casa de n ($O(n^3)$), por chamarmos n vezes as funções "resolve_matriz_inferior" e "resolve_matriz_superior", que tem complexidade $O(n^2)$. Nas próximas páginas, o código e seus exemplos:

```julia
"""
Função que recebe uma matriz A e retorna A invertida.
(Utiliza o método A*A^(-1) = I <-> L*U*A^(-1) = I, onde temos U*A^(-1) = y)
Entradas: Matriz A, uma matriz quadrada;
Saídas: A_inv, Uma matriz quadrada que é a inversa de A.
Complexidade: O(n^3)
"""

function inversa_LU(A)
    L, U = decomposicao_LU(A) # 'Desmonta' a matriz A em duas matrizes,
                              # L (quadrada triangular inferior) e U (quadrada triangular superior)
    n, = size(A) # Obtém o número de linhas e colunas em A e armazena em n
    y = zeros(n, n) # Cria uma matiz y, que será intermediária, e a preenche com zeros
    A_inv = zeros(n, n) # Cria a matriz que sera retornada, a inversa de A, e preenche com zeros
    ident = zeros(n, n) # Cria uma matriz identidade, e a preenche com zeros
    for i = 1:n # For para preencher a diagonal da matriz identidade com 1's
        ident[i,i] = 1
    end
    for i = 1:n # For que completa a matriz A invertida, das colunas 1 a n
        y[:,i] = resolve_triangular_inferior(L, ident[:,i]) # Resolve o triangular inferior com a coluna i da matriz identidade,
                                                            # no molde L*y_:,i = I_:,i
        A_inv[:, i] = resolve_triangular_superior(U, y[:,i])# Resolve o triangular superior com a coluna i da matriz y,
                                                            # no molde U*A_:,i = y_:,i
    end
    return A_inv # Retorna a matriz A invertida
end
```

```
inversa_LU (generic function with 1 method)
```

```julia
# Matriz identidade usada para os testes
ident = [1 0 0;
         0 1 0;
         0 0 1]
```

```
3×3 Matrix{Int64}:
 1  0  0
 0  1  0
 0  0  1
```

```julia
A = [3.0 2.0 4.0;
    1.0 1.0 2.0;
    4.0 3.0 -2.0]
A_inv = inversa_LU(A)
A*A_inv ≈ ident
```

```
true
```

```julia
A = [1.0 2.0 3.0;
    4.0 -2.0 6.0;
    1.0 4.0 5.0]
A_inv = inversa_LU(A)
A*A_inv ≈ ident
```

```
true
```

```julia
A = [8.0 -4.0 -2.0;
    -4.0 10.0 -2.0;
    -2.0 -2.0 10.0]
A_inv = inversa_LU(A)
A*A_inv ≈ ident
```

```
true
```

Questão 2

a) Por aproximações da primeira e segunda derivadas, temos o seguinte:

$y'(x_k) \approx \frac{y(x_{k+1})-y(x_k)}{h}$ ou $y'(x_k) \approx \frac{y(x_k)-y(x_{k-1})}{h}$

$y"(x_k) \approx \frac{y'(x_{k+1})-y'(x_{k-1})}{2h}$, que pode ser escrito como:

$(*1)y"(x_k) \approx \frac{y(x_{k+1})-2y(x_k)+y(x_{k-1})}{2h^2}$

Como temos n igual 6, e temos nosso x final (x_7) igual a 10, enquanto nosso x inicial (x_1) é igual a 0, temos h como:

$h * n = x_7 - x_1 \rightarrow h = \frac{10}{6} = \frac{5}{3}$

Então, vamos ter para valores de x e das suas respectivas segundas derivadas:

$x_2 = x_1 + h = \frac{5}{3} | y"(x_2) = 4 * \frac{5}{3} = \frac{20}{3}$

$x_3 = x_1 + 2h = \frac{10}{3} | y"(x_3) = 4 * \frac{10}{3} = \frac{40}{3}$

$x_4 = x_1 + 3h = \frac{15}{3} = 5 | y"(x_4) = 4 * 5 = 20$

$x_5 = x_1 + 4h = \frac{20}{3} | y"(x_5) = 4 * \frac{20}{3} = \frac{80}{3}$

$x_6 = x_1 + 5h = \frac{25}{3} | y"(x_6) = 4 * \frac{25}{3} = \frac{100}{3}$

Seguindo (*1), vamos ter que:

$y"(x_2) = \frac{20}{3} \approx \frac{y(x_3)-2y(x_2)+y(x_1)}{2(\frac{5}{3})^2} \rightarrow \frac{20}{3} * 2 * \frac{25}{9} \approx y(x_3) - 2y(x_2) + 5 \rightarrow$

$\frac{1000}{27} - 5 \approx y(x_3) - 2y(x_2)$

$y"(x_3) = \frac{40}{3} \approx \frac{y(x_4)-2y(x_3)+y(x_2)}{2(\frac{5}{3})^2} \rightarrow \frac{40}{3} * 2 * \frac{25}{9} \approx y(x_4) - 2y(x_3) + y(x_2) \rightarrow$

$\frac{2000}{27} \approx y(x_4) - 2y(x_3) + y(x_2)$

$y"(x_4) = 20 \approx \frac{y(x_5)-2y(x_4)+y(x_3)}{2(\frac{5}{3})^2} \rightarrow 20 * 2 * \frac{25}{9} \approx y(x_5) - 2y(x_4) + y(x_3) \rightarrow$

$\frac{1000}{9} \approx y(x_5) - 2y(x_4) + y(x_3)$

$y"(x_5) = \frac{80}{3} \approx \frac{y(x_6)-2y(x_5)+y(x_4)}{2(\frac{5}{3})^2} \rightarrow \frac{80}{3} * 2 * \frac{25}{9} \approx y(x_6) - 2y(x_5) + y(x_4) \rightarrow$

$\frac{4000}{27} \approx y(x_6) - 2y(x_5) + y(x_4)$

$y"(x_6) = \frac{100}{3} \approx \frac{y(x_7)-2y(x_6)+y(x_5)}{2(\frac{5}{3})^2} \rightarrow \frac{100}{3} * 2 * \frac{25}{9} \approx 20 - 2y(x_6) + y(x_5) \rightarrow$

$\frac{5000}{27} - 20 \approx -2y(x_6) + y(x_5)$

Então, formamos o seguinte sistema:

$$\begin{cases} y(x_3) - 2y(x_2) = \frac{1000}{27} - 5 \\ y(x_4) - 2y(x_3) + y(x_2) = \frac{2000}{27} \\ y(x_5) - 2y(x_4) + y(x_3) = \frac{1000}{9} \\ y(x_6) - 2y(x_5) + y(x_4) = \frac{4000}{27} \\ -2y(x_6) + y(x_5) = \frac{5000}{27} - 20 \end{cases}$$

b) Com o sistema que formamos na letra A, podemos organizar-lo em uma multiplicação de matrizes no molde Ay = b, onde:

$$A = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix}$$

$$y = \begin{bmatrix} y(x_2) \\ y(x_3) \\ y(x_4) \\ y(x_5) \\ y(x_6) \end{bmatrix}$$

$$b = \begin{bmatrix} \frac{2000}{27} - 5 \\ \frac{4000}{27} \\ \frac{1000}{9} \\ \frac{4600}{27} \\ \frac{5000}{27} - 20 \end{bmatrix}$$

Então, aplicamos tais matrizes ao Julia, onde conseguimos obter os resultados dos valores desejados (próxima página):

```julia
A = [-2 1 0 0 0;
     1 -2 1 0 0;
     0 1 -2 1 0;
     0 0 1 -2 1;
     0 0 0 1 -2]
```

```
5×5 Matrix{Int64}:
 -2   1   0   0   0
  1  -2   1   0   0
  0   1  -2   1   0
  0   0   1  -2   1
  0   0   0   1  -2
```

```julia
b = [1000/27 - 5;
     2000/27;
     1000/9;
     4000/27;
     5000/27 - 20]
```

```
5-element Vector{Float64}:
  32.03703703703704
  74.07407407407408
 111.11111111111111
 148.14814814814815
 165.1851851851852
```

```julia
y = A\b
```

```
5-element Vector{Float64}:
 -208.54938271604937
 -385.0617283950617
 -487.50000000000006
 -478.82716049382725
 -322.00617283950623
```

Logo, temos $y(x_2) = 5$, $y(x_3) \approx -209$, $y(x_4) \approx 385$, $y(x_5) \approx -488$ e $y(x_6) \approx -322$

c) Aplicando esses x que obtemos, e os y que obtemos, podemos usar as funções criadas nas ultimas listas para fazer a interpolação polinomial de grau 3:

```
function vandermonde(x, grau) # Função que gera a matriz para acharmos os coeficientes
    n = length(x) # Pega o número de pontos
    V = zeros(n, grau + 1) # Gera uma matriz zerada de altura n e largura d
    for i = 1:n
        for j = 1:(grau + 1)
            V[i,j] = x[i]^(j - 1) # Preenche a matriz com os valores de x elevados
        end
    end
    return V
end
```

```
vandermonde (generic function with 1 method)
```

```
function regressao(x, y, grau) #Por regressão, calcula-se os coeficientes
    V = vandermonde(x, grau) # Calcula a matriz
    c = V\y # Acha os coeficientes
    return c
end
```

```
regressao (generic function with 1 method)
```

```
function gerar_a_funcao(x,y,grau) # Função que gera a função de cada grau
    c = regressao(x, y, grau) # Obtém os coeficientes
    h(x) = sum(c[j]*x^(j - 1) for j = 1:(grau + 1)) # Monta a função
    return h
end
```

```
gerar_a_funcao (generic function with 1 method)
```

Então, com os dados que temos, adicionando os valores de $x_1 = 0$ e $x_7 = 10$ ao vetor x, e $y(x_1) = 5$ e $y(x_7) = 20$ ao vetor y, temos a seguinte função:

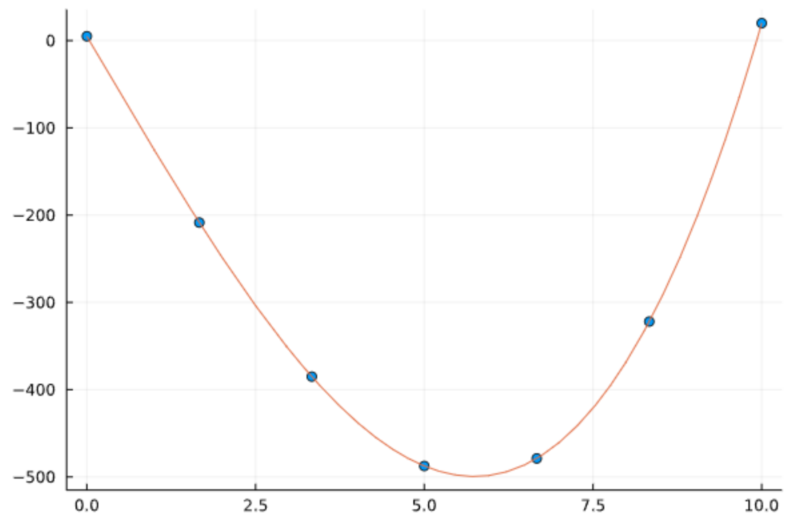
```
y = [5, y[1], y[2], y[3], y[4], y[5], 20]
```

```
7-element Vector{Float64}:
    5.0
 -208.54938271604937
 -385.0617283950617
 -487.50000000000006
 -478.82716049382725
 -322.00617283950623
   20.0
```

```
x = [0, 5/3, 10/3, 5, 20/3, 25/3, 10]
```

```
7-element Vector{Float64}:
  0.0
  1.6666666666666667
  3.3333333333333335
  5.0
  6.666666666666667
  8.333333333333334
 10.0
```

```
g = gerar_a_funcao(x, y, 3)
scatter(x, y, leg = false)
plot!(g)
```



Então, podemos achar $y(3.2345)$:

```
g(2.2345)
```

```
-274.7058010484999
```

Logo, $y(2.2345) \approx -275$.

Questão 3

Essa questão e a questão 1 foi feita em conjunto com os alunos Matheus Silva e João Matheus.

a) Como cada ponto é composto pela média da temperatura dos pontos ao lado dele, temos que:

$x_1 = \frac{15+5+x_2+x_3}{4} \rightarrow x_1 - \frac{x_2}{4} - \frac{x_3}{4} = 5$

$x_2 = \frac{15+35+x_1+x_4}{4} \rightarrow x_1 - \frac{x_1}{4} - \frac{x_4}{4} = \frac{50}{4}$

$x_3 = \frac{10+5+x_1+x_4}{4} \rightarrow x_3 - \frac{x_1}{4} - \frac{x_4}{4} = \frac{15}{4}$

$x_4 = \frac{10+35+x_2+x_3}{4} \rightarrow x_4 - \frac{x_2}{4} - \frac{x_3}{4} = \frac{45}{4}$

Logo, temos as matrizes:

$$A * x = b \rightarrow \begin{bmatrix} 1 & -\frac{1}{4} & -\frac{1}{4} & 0 \\ -\frac{1}{4} & 1 & 0 & -\frac{1}{4} \\ -\frac{1}{4} & 0 & 1 & -\frac{1}{4} \\ 0 & -\frac{1}{4} & -\frac{1}{4} & 1 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 5 \\ \frac{50}{4} \\ \frac{15}{4} \\ \frac{45}{4} \end{bmatrix}$$

b) Utilizando as funções criadas na questão 1, temos:

```
A = [1 -1/4 -1/4 0; -1/4 1 0 -1/4; -1/4 0 1 -1/4; 0 -1/4 -1/4 1]
```

```
4×4 Matrix{Float64}:
  1.0   -0.25  -0.25   0.0
 -0.25   1.0    0.0   -0.25
 -0.25   0.0    1.0   -0.25
  0.0   -0.25  -0.25   1.0
```

```
b = [5 50/4 15/4 45/4]
```

```
1×4 Matrix{Float64}:
 5.0  12.5  3.75  11.25
```

```
L, U = decomposicao_LU(A)
```

```
([1.0 0.0 0.0 0.0; -0.25 1.0 0.0 0.0; -0.25 -0.06666666666666667 1.0 0.0; 0.0 -0.26666666666666666 -0.2857142857142857 1.0],
[1.0 -0.25 -0.25 0.0; 0.0 0.9375 -0.0625 -0.25; 0.0 0.0 0.9333333333333333 -0.26666666666666666; 0.0 0.0 0.0 0.857142857142857
2])
```

```
# A*x = b, onde A = L*U
# L*y = b, onde obtemos y para achar x
# U*x = y, obtendo x
y = resolve_triangular_inferior(L, b)
x = resolve_triangular_superior(U, y)
```

```
4-element Vector{Float64}:
 13.125
 20.625
 11.875
 19.375
```

Logo, temos que $x_1 = 13.125$, $x_2 = 20.625$, $x_3 = 11.875$ e $x_4 = 19.375$ (em graus Celsius).

c) Primeiro, montamos a matriz A com os pontos x_1 a x_{25}. As equações são do mesmo formato da letra a):

```
A = [1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
     -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
     0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
     0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
     0 0 0 -1/4 1 0 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
     -1/4 0 0 0 0 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
     0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0 0;
     0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0 0;
     0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0 0;
     0 0 0 0 -1/4 0 0 0 -1/4 1 0 0 0 0 -1/4 0 0 0 0 0 0 0 0 0 0;
     0 0 0 0 0 -1/4 0 0 0 0 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0 0;
     0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0 0;
     0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0 0;
     0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0 0 0 0;
     0 0 0 0 0 0 0 0 0 -1/4 0 0 -1/4 1 0 0 0 0 0 -1/4 0 0 0 0 0;
     0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 0 1 -1/4 0 0 0 -1/4 0 0 0 0;
     0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0 0;
     0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0 0;
     0 0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0 0 -1/4 0;
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 0 0 0 0 -1/4;
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 0 1 -1/4 0 0 0;
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0 0;
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4 0;
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1 -1/4;
     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1/4 0 0 0 -1/4 1;]
```

Então, montamos a matriz b. Note que a mesma tera alguns valores 0, dado o fato de vários pontos terem como referências vizinhas vértices a serem achados também:

```
b = [45/4 ;
    20/4;
    20/4 ;
    20/4;
    40/4;
    25/4;
    0;
    0;
    0;
    20/4;
    25/4;
    0;
    0;
    0;
    20/4;
    25/4;
    0;
    0;
    0;
    20/4;
    55/4;
    30/4;
    30/4;
    30/4;
    50/4]
```

Por fim, utilizando as funções criadas na questão 1, temos:

```
# A*x = b, onde A = L*U
# L*y = b, onde obtemos y para achar x
# U*x = y, obtendo x
L, U = decomposicao_LU(A)
y = resolve_triangular_inferior(L, b)
x = resolve_triangular_superior(U, y)
```

```
25-element Vector{Float64}:
 22.544910929891554
 21.518050214897478
 20.87762318174459
 20.320043856439405
 19.899858605080322
 23.661593504668744
 22.64966674795377
 21.672398655641462
 20.502693638932705
 19.27939056388188
 24.45179634082966
 23.74662461660739
 22.659611053934803
 20.738941479768073
 16.715010011514508
 25.398967242042485
 25.225424323711348
 24.48047946372228
 23.078451214690272
 21.016764301255613
 26.918648303628935
 27.275625972473247
 26.958431262552672
 26.07761961401516
 24.27359597881769
```

Sendo essa a temperatura dos vértices, em Celsius, dos vértices do interior do quadrado.

d) Para achar o número de nós na qual a discretização começa a demorar mais de 2 minutos utilizando a decomposição LU, montamos uma função, "resolve_lago", que pede o número de linhas e a temperatura das margens do lago, e retorna a temperatura de cada ponto do lago:

```
#item 3)d)

function resolve_lago(n_linhas, margem_sup, margem_dir, margem_inf, margem_esq)
    #declarando a matriz e o vetor (Ax = b)
    A = zeros((n_linhas)^2,(n_linhas)^2)
    b = zeros(n_linhas^2,1)

    for i=1:(n_linhas)^2
        A[i,i] = 1   #preenchendo a diagonal

        if((i+1)%n_linhas == 1) #se isso acontecer, estamos olhando para a margem direita,
                                #afinal só há n_linha pontos em cada linha
            b[i] += margem_dir/4  #então esse valor é uma constante, que fica do outro lado da equação
        else
            A[i,i+1] = -1/4     #se o ponto estiver "no meio" do quadrado, colocamos o -1/4
                                # na matriz, já que está multiplicando uma incógnita
        end


        if((i-1)%n_linhas == 0)   #se isso acontecer, estamos na margem esquerda
            b[i] += margem_esq/4

        else
            A[i,i-1] = -1/4
        end

        if((i+n_linhas)> n_linhas^2) #nesse caso, estamos nos ultimos pontos do quadrado,
            b[i] += margem_inf/4     #na margem inferior

        else
            A[i,i+n_linhas] = -1/4
        end

        if((i-n_linhas) < 1)
            b[i] += margem_sup/4    #nesse caso, estamos nos primeiros pontos do quadrado, na
                                    #margem superior
        else
            A[i,i-n_linhas] = -1/4
        end

    end

    #resolvendo o sistema resultante
    L, U = decomposicao_LU(A)
    y = resolve_triangular_inferior(L, b)
    x = resolve_triangular_superior(U, y)
    return x
end
```

```
resolve_lago (generic function with 1 method)
```

```
x = resolve_lago(52,15,35,10,5) #52 linhas, ou seja, 2704 pontos (52^2), já leva 2 minutos
length(x) #só pro output não ser um vetor de milhares de linhas
```

```
2704
```

Questão 4

a) Vamos começar analisando cada "ponto" por qual os canos passam e se redistribuem. Temos 8 pontos, de A a H. Vamos organizar de forma que a esquerda seja a chegada de água ao ponto, e a direita a saída de água do ponto, depois reorganizando para que possamos montar a matriz A:

A: $x_1 = 5000 + 2000$

B: $x_2 = 1500 + 2000$

C: $x_3 = 8000 + 1000$

D: $x_4 = 30000 + x_1 \rightarrow -x_1 + x_4 = 30000$

E: $x_5 = 3000 + x_3 \rightarrow -x_3 + x_5 = 3000$

F: $x_6 = x_2 + x_4 \rightarrow -x_2 - x_4 + x_6 = 0$

G: $x_7 = 3000 + x_5 \rightarrow -x_5 + x_7 = 3000$

H: $x_8 = 500 + x_6 + x_7 \rightarrow -x_6 - x_7 + x_8 = 500$

Com isso, podemos montar as matrizes A e b:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} 7000 \\ 3500 \\ 9000 \\ 30000 \\ 3000 \\ 0 \\ 3000 \\ 500 \end{bmatrix}$$

Por fim, podemos aplicar as variáveis que criamos na questão 1:

```
A = [1 0 0 0 0 0 0 0;
     0 1 0 0 0 0 0 0;
     0 0 1 0 0 0 0 0;
    -1 0 0 1 0 0 0 0;
     0 0 -1 0 1 0 0 0;
     0 -1 0 -1 0 1 0 0;
     0 0 0 0 -1 0 1 0;
     0 0 0 0 0 -1 -1 1;]
```

```
8×8 Matrix{Int64}:
  1   0   0   0   0   0   0  0
  0   1   0   0   0   0   0  0
  0   0   1   0   0   0   0  0
 -1   0   0   1   0   0   0  0
  0   0  -1   0   1   0   0  0
  0  -1   0  -1   0   1   0  0
  0   0   0   0  -1   0   1  0
  0   0   0   0   0  -1  -1  1
```

```
b = [7000; 3500; 9000; 30000; 3000; 0; 3000; 500]
```

```
8-element Vector{Int64}:
  7000
  3500
  9000
 30000
  3000
     0
  3000
   500
```

```
# A*x = b, onde A = L*U
# L*y = b, onde obtemos y para achar x
# U*x = y, obtendo x
L, U = decomposicao_LU(A)
y = resolve_triangular_inferior(L, b)
x = resolve_triangular_superior(U, y)
```

```
8-element Vector{Float64}:
  7000.0
  3500.0
  9000.0
 37000.0
 12000.0
 40500.0
 15000.0
 56000.0
```

Então, temos $x_1 = 7000l/m$, $x_2 = 3500l/m$, $x_3 = 9000l/m$, $x_4 = 37000l/m$, $x_5 = 12000l/m$, $x_6 = 40500l/m$, $x_7 = 15000l/m$ e $x_8 = 56000l/m$.

b) Caso a modelagem seja feita com o cano (faz o L) x_9, dois "pontos" se alteram em comparação com a modelagem original, D e E:

D: $x_4 = 30000 + x_1 + x_9 \rightarrow -x_1 + x_4 - x_9 = 30000$

E: $x_5 + x_9 = 3000 + x_3 \rightarrow -x_3 + x_5 + x_9 = 3000$

Então, ficamos com a matriz A (note que b se mantém a mesma da modelagem anterior):

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 \end{bmatrix}$$

Porém, ao aplicarmos isso a nossa função de decomposição LU, vamos obter o mesmo resultado da letra a). Não só isso, mas se fizermos a prova real da função (L*U = A), vamos obter falso (próxima página):

```julia
A = [1 0 0 0 0 0 0 0 0;
     0 1 0 0 0 0 0 0 0;
     0 0 1 0 0 0 0 0 0;
    -1 0 0 1 0 0 0 0 -1;
     0 0 -1 0 1 0 0 0 1;
     0 -1 0 -1 0 1 0 0 0;
     0 0 0 0 -1 0 1 0 0;
     0 0 0 0 0 -1 -1 1 0;]
```

```
8×9 Matrix{Int64}:
  1   0   0   0   0   0   0  0   0
  0   1   0   0   0   0   0  0   0
  0   0   1   0   0   0   0  0   0
 -1   0   0   1   0   0   0  0  -1
  0   0  -1   0   1   0   0  0   1
  0  -1   0  -1   0   1   0  0   0
  0   0   0   0  -1   0   1  0   0
  0   0   0   0   0  -1  -1  1   0
```

```julia
# A*x = b, onde A = L*U
# L*y = b, onde obtemos y para achar x
# U*x = y, obtendo x
L, U = decomposicao_LU(A)
y = resolve_triangular_inferior(L, b)
x = resolve_triangular_superior(U, y)
```

```
8-element Vector{Float64}:
  7000.0
  3500.0
  9000.0
 37000.0
 12000.0
 40500.0
 15000.0
 56000.0
```

```julia
L*U ≈ A
```

```
false
```

Por que isso acontece? O motivo é porque nossa função foi projetada para lidar com matrizes quadradas, enquanto a matriz A que achamos com a modelagem é 8x9. Ele consegue trabalhar com valores dentro das primeiras 8 colunas, ignorando a última, mas não leva em consideração o cano x_9 e erra o calculo de L e U.