Aluna: Livia Barbosa Fonseca
DRE: 118039721

Questão 1:

Questão realizada pelos alunos Augusto Guimarães - DRE:119025393, Luiz Rodrigo Lace - DRE: 118049873 e Livia Barbosa Fonseca - DRE: 118039721.

i) Resolve diagonal

```
# Matriz Diagonal
# Recebe como parâmetro duas matrizes A e b e retorna x tal que Ax = b.
#(A deve ser uma matriz diagonal)
function resolve_diagonal(A, b)
    #Pega tamanho da matriz A
    n = size(A, 1)

    #Cria matriz x zerada do tamanho (nx1)
    x = zeros(n, 1)

    #Lopping que realiza a resolução do sistema (x=b/a)
    for i = 1:n
        x[i] = b[i] / A[i, i]
    end

    #Retorna x tal que Ax = b
    x
end
```

Realizando o primeiro teste temos:

```
#Testando a função matriz diagonal
#criando matriz A
A = [2 0 0 ;0 3 0; 0 0 4]
#Gerando valores aleatorios para a matriz x
x_verdadeiro=randn(3,1)

#Multiplicando os números aleatorios pela matriz A (b = Ax)
b= A*x_verdadeiro

#Aplicando a função
x_calculado=resolve_diagonal(A ,b)

#Conferindo se o valor é o esperado
norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

Podemos observar que o valor esperado é encontrado

Realizando o segundo teste encontramos:

```
#Testando a função matriz diagonal
#criando matriz A
A = [10 0 0 0 0;0 3 0 0 0; 0 0 7 0 0; 0 0 0 9 0; 0 0 0 0 6]

#Gerando valores aleatorios para a matriz x
x_verdadeiro=randn(5,1)


#Multiplicando os números aleatorios pela matriz A (b = Ax)
b= A*x_verdadeiro

#Aplicando a função
x_calculado=resolve_diagonal(A,b)

#Conferindo se o valor é o esperado
norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

Realizando o terceiro teste:

```julia
#Testando a função matriz diagonal
#criando matriz A
A = [10 0 0 0 0 0 0;0 8 0 0 0 0 0; 0 0 17 0 0 0 0; 0 0 0 99 0 0 0; 0 0 0 0 66 0 0; 0 0 0 0 0 111 0; 0 0 0 0 0 0 7 ]

#Gerando valores aleatorios para a matriz x
x_verdadeiro=randn(7,1)

#Multiplicando os números aleatorios pela matriz A (b = Ax)
b= A*x_verdadeiro

#Aplicando a função
x_calculado=resolve_diagonal(A,b)

#Conferindo se o valor é o esperado
norm(x_verdadeiro-x_calculado) < 0.000000000001
```

Julia

```
true
```

ii) resolve triangular superior

```julia
# Matriz Diagonal Superior
# Recebe como parâmetro duas matrizes A e b e retorna x tal que Ax = b.
# A deve ser triangular superior
function resolve_triangular_superior(A, b)
   #Recebe tamanho da matriz A
    n = size(A, 1)

   #Cria matriz zerada de x(nx1)
    x = zeros(n, 1)

   #Matriz triangular superior, então pegamos a matriz ao "contrário" para iniciarmos a substituição dos valores e solução
    for i = reverse(1:n)
        #x recebe o que está em b
        x[i] = b[i]

        #Diminui dos outros coeficientes (substituindo)
        for j = reverse(i+1:n)
            x[i] -= A[i, j] * x[j]
        end

       #Realiza a divisão do coeficiente pela 'resposta'
        x[i] /= A[i, i]
    end
```

```
    #Retorna a matriz x de Ax=b
    x
end
```

Agora vamos realizar um teste para conferirmos se essa função está funcionando como o esperado:

```
    #matriz triangular superior
    A = [2 2 2 ;0 3 3; 0 0 4]

    #Gerando valores aleatorios para a matriz x
    x_verdadeiro=randn(3,1)

    #Encontrando b por meio de b = Ax
    b = A*x_verdadeiro

    #Chamando a função
    x_calculado=resolve_triangular_superior(A ,b)

    ##Conferendo se o valor é o esperado
    norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

Outro exemplo:

```
    #matriz triangular superior
    A = [10 244 421 33 1;0 3 33 22 11; 0 0 7 6 5; 0 0 0 92 1; 0 0 0 0 68]

    #Gerando valores aleatorios para a matriz x
    x_verdadeiro=randn(5,1)

    #Encontrando b por meio de b = Ax
    b = A*x_verdadeiro

    #Chamando a função
    x_calculado=resolve_triangular_superior(A,b)

    ##Conferendo se o valor é o esperado
    norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

Mais um exemplo:

```
#matriz triangular superior
Teste3 = [10 20 30 40 50 60 70;0 82 2 3 40 1 1;
0 0 17 20 30 40 110; 0 0 0 99 33 55 44;
0 0 0 0 66 11 22; 0 0 0 0 0 111 3; 0 0 0 0 0 0 7]

#Gerando valores aleatorios para a matriz x
x_verdadeiro=randn(7,1)

#Encontrando b por meio de b = Ax
b= Teste3*x_verdadeiro

#Chamando a função
x_calculado=resolve_triangular_superior(Teste3,b)

##Conferindo se o valor é o esperado
norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

iii) resolve triangular inferior

```
# Matriz Diagonal Inferior
# Recebe como parâmetro duas matrizes A e b e retorna x tal que Ax = b.
# A deve ser triangular inferior
function resolve_triangular_inferior(A, b)
    #Recebe tamanho da matriz A
    n = size(A, 1)

    #Cria matriz zerada de x(nx1)
    x = zeros(n, 1)

       for i = 1:n
       #Dizemos que x é igual a b, pois é triangular inferior
        x[i] = b[i]
```

```
        #Diminui dos outros coeficientes (substituindo)
        for j = 1:i-1
            x[i] -= A[i, j] * x[j]
        end
       #Realiza a divisão do valor do coeficiente pela 'resposta'
        x[i] /= A[i, i]
    end
    #Retorna a matriz x de Ax=b
    x
end
```

Agora vamos fazer alguns exemplos:

```
#matriz triangular superior
Teste3 = [10 20 30 40 50 60 70;0 82 2 3 40 1 1;
0 0 17 20 30 40 110; 0 0 0 99 33 55 44;
0 0 0 0 66 11 22; 0 0 0 0 0 111 3; 0 0 0 0 0 0 7]

#Gerando valores aleatorios para a matriz x
x_verdadeiro=randn(7,1)

#Encontrando b por meio de b = Ax
b= Teste3*x_verdadeiro

#Chamando a função
x_calculado=resolve_triangular_superior(Teste3,b)

##Conferindo se o valor é o esperado
norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

Outro exemplo:

```
#matriz triangular inferior
A = [10 0 0 0 0;3 3 0 0 0; 7 6 5 0 0; 1 92 33 1 0; 244 421 33 1 68]

#Gerando valores aleatorios para a matriz x
x_verdadeiro=randn(5,1)

#Encontrando b por meio de b = Ax
b= A*x_verdadeiro

#Chamando a função
x_calculado=resolve_triangular_inferior(A,b)

#Conferindo se o valor é o esperado
norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

Outro exemplo:

```
#matriz triangular inferior
A = [1 0 0; 3 2 0; 1 4 3]

#Gerando valores aleatorios para a matriz x
x_verdadeiro=randn(3,1)

#Encontrando b por meio de b = Ax
b = A*x_verdadeiro

#Chamando a função
x_calculado=resolve_triangular_inferior(A ,b)

#Conferindo se o valor é o esperado
norm(x_verdadeiro-x_calculado) < 0.000000000001
```

```
true
```

iv) decomposição LU

```
#Decomposição LU
#Recebe uma matriz a e retorna as matrizes L e U.
#Código baseado nas observações feitas em sala de aula e
#nas explicações encontradas no site:
https://www.ime.unicamp.br/~pjssilva/pdfs/notas_de_aula/ms211/Sistemas_
Lineares.pdf
function decomposicao_lu(A)
    #Cria matriz U e passa a copia de A
    U = copy(A)
    #Pega o tamanho de A
    n = size(A, 1)
    #Cria matriz L, identidade
    L = Matrix{Float64}(I, n, n)

    #Colocando zeros abaixo da posição i x i usando a linha i
    for i = 1:n
        for j = i+1:n
            coef=U[j, i] / U[i, i]
            L[j, i] = coef
            U[j, :] -= coef * U[i, :]
        end
    end

    #Retornando matriz L (triangular inferior),U (Triangular superior)
    L, U
end
```

Montando os exemplos temos:

```
#Matriz que será decomposta
k = [2 3 1 1; 4 7 4 3; 4 7 6 4; 6 9 9 8]

#Chamando a função
l,U=decomposicao_lu(k)
```

✓ 0.4s

```
([1.0 0.0 0.0 0.0; 2.0 1.0 0.0 0.0; 2.0 1.0 1.0 0.0; 3.0 0.0 3.0 1.0], [2 3 1 1; 0 1 2 1; 0 0 2 1; 0 0 0 2])
```

```
#Printando matriz l
l
```

✓ 0.2s

```
4x4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 2.0  1.0  0.0  0.0
 2.0  1.0  1.0  0.0
 3.0  0.0  3.0  1.0
```

```
#Printando matriz u
U
```

✓ 0.3s

```
4x4 Matrix{Int64}:
 2  3  1  1
 0  1  2  1
 0  0  2  1
 0  0  0  2
```

Outro exemplo:

```
#Matriz que será decomposta
k = [3 6 4 2]

#Chamando a função
l,U=decomposicao_lu(k)
```

✓ 0.3s

```
([1.0], [3 6 4 2])
```

```
l
```

✓ 0.3s

```
1x1 Matrix{Float64}:
 1.0
```

```
U
```

✓ 0.1s

```
1x4 Matrix{Int64}:
 3  6  4  2
```

Outro exemplo:

```
#Matriz que será decomposta
k = [1 2 3; 5 4 3; 7 5 4]

#Chamando a função
l,U=decomposicao_lu(k)
```
✓ 0.3s

```
([1.0 0.0 0.0; 5.0 1.0 0.0; 7.0 1.5 1.0], [1 2 3; 0 -6 -12; 0 0 1])
```

```
l
```
✓ 0.3s

```
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 5.0  1.0  0.0
 7.0  1.5  1.0
```

```
U
```
✓ 0.2s

```
3×3 Matrix{Int64}:
 1   2    3
 0  -6  -12
 0   0    1
```

v) Como podemos usar LU para achar a inversa de uma matriz An×n?. Escreva uma função em Julia que recebe uma matriz quadrada e retorna a sua inversa. Qual é a complexidade do seu algoritmo?

```
#Função que recebe uma matriz T e retorna sua inversa
function MatrizInversa(T)

    #Pega o tamanho de t
    n = size(T,1)

    #Cria matriz identidade
    identidade = Matrix{Float64}(I,n,n)

    #Realiza a decomposição lu
    L,U = decomposicao_lu(T)

    #Matriz triangular inferior L dividida pela identidade
    y = L\identidade
```

```
    #Matriz triangular superior U dividida pela identidade
    x=U\y

    #Retorna a inversa de t
    return x
end
```

Exemplos:

```
    a =[1 4 -3; 2 7 0; -1 8 -9]

    #Inversa de a
    MatrizInversa(a)
```

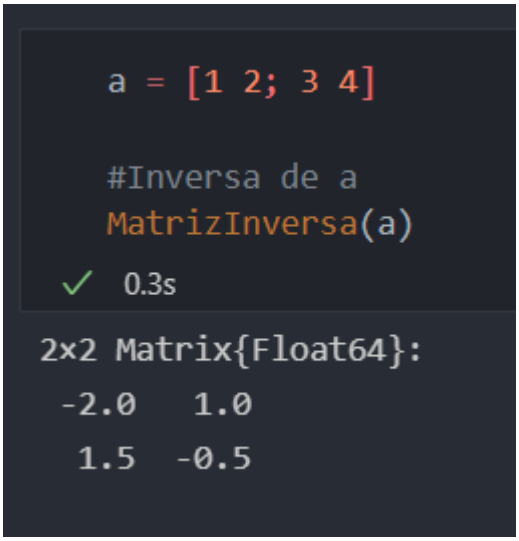
✓ 0.5s

```
3×3 Matrix{Float64}:
  1.05       -0.2   -0.35
 -0.3         0.2    0.1
 -0.383333    0.2    0.0166667
```

```
    a = [1 -2; 4 6]

    #Inversa de a
    MatrizInversa(a)
```

✓ 0.5s

```
2×2 Matrix{Float64}:
  0.428571   0.142857
 -0.285714   0.0714286
```



A complexidade do algoritmo é $O(n^3)$.

Questão 2:

a) Dado o problema de valor de contorno $y''(x) = 4x$ com $y(0) = 5$ e $y(10) = 20$. Vamos montar o sistema linear que se aproxima pelo método das diferenças finitas com n = 6 intervalos de discretização.

Primeiramente, vamos calcular as aproximações de y' e y'' para podermos substituir no PVC:

$$y' \approx y_{k+1} - y_{k-1}/2h$$

Sabendo que podemos calcular y'' com ajuda de y', temos:

$$y'' \approx y'(x_{k+1}) - y'(x_{k-1})/2h$$
$$y'' \approx (((y(x_{k+1}) - y(x_k))/h) - (y(x_k) - y(x_{k-1}))/h)/2h$$
$$y'' \approx (y(x_{k+1}) - 2y(x_k) + y(x_{k-1})/2h^2$$

Agora vamos substituir o y'' na equação dada pela questão:

$$y''(x) = 4x$$
$$(y(x_{k+1}) - 2y(x_k) + y(x_{k-1}))/2h^2 = 4x_k$$

Vamos calcular o valor de h para colocar na fórmula:

$$n = (b - a)/h$$
$$6 = (10 - 0)/h$$
$$h = 10/6$$

Sabendo o número e tamanho dos intervalos de discretização e as condições de contorno, podemos montar a seguinte tabela:

k	x_k	y_k
0	0	5
1	10/6	
2	(10/6)*2	
3	(10/6)*3	
4	(10/6)*4	
5	(10/6)*5	
6	10	20

Agora, vamos encontrar os valores de y_k.

Sabemos que teremos 6 intervalos de discretização, então podemos escrever:
k = 1:
$(y(x_2) - 2y(x_1) + 5) = 8(10/6)^2x_1$

k = 2:
$(y(x_3) - 2y(x_2) + y(x_1)) = 8(10/6)^2x_2$

k = 3:
$(y(x_4) - 2y(x_3) + y(x_2)) = 8(10/6)^2x_3$

k = 4:
$(y(x_5) - 2y(x_4) + y(x_3)) = 8(10/6)^2x_4$

k = 5:
$(y(x_6) - 2y(x_5) + y(x_4)) = 8(10/6)^2x_5$

k =6:
$(10 - 2y(x_6) + y(x_5)) = 8(10/6)^2x_6$

Para simplificar, podemos dizer que P = 8*(10/6)^2. Sendo assim, podemos escrever as matrizes:

$$\begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix} * \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} P_{x1} - 5 \\ P_{x2} \\ P_{x3} \\ P_{x4} \\ P_{x5} - 10 \end{pmatrix}$$

b) Agora vamos resolver o sistema linear obtido no item anterior. Para isso, vamos utilizar o método de Gauss Seidel, uma forma iterativa para resolução de sistemas lineares. Esse método nos pede inicialmente um chute inicial, onde a partir dele realizaremos o seguinte cálculo informado abaixo de forma iterativa, até xk está muito próximo de xk+1.

Queremos resolver o sistema do tipo Ax = b, para isso, vamos realizar transformações algébricas para encontrar o ponto fixo e resolver o sistema:

$$Ax = b$$

Quebrando a matriz A em duas, onde uma delas é inversível (k e M)

$$(M - k)x = b$$

Quebrando em duas partes

$$Mx - kx = b$$

$$Mx = Kx + b$$

Como M é inversível, podemos fazer a seguinte transformação

$$\mathbf{x = M^{-1}(kx + b)}$$

Vamos utilizar $\mathbf{x = M^{-1}(kx + b)}$ para a construção do método iterativo no Julia:

```
#Dado duas matrizes A(nxn) e b(nxn, retorna um vetor de coeficientes x
tais que Ax ≈ b
function Gauss_Seidel(A,b)

    #Guarda o tamanho da matriz em n
    n,=size(A)

    #Guarda a diagonal da matriz A em M
    M =Diagonal(A)

    #Zera a diagonal da A e guarda em K
    K=A-M
```

```
    #chute inicial
    x = randn(n,1)

    #Iteração que realiza a aproximação de Ax ≈ b
    for i=1:500
        x = inv(M)*(-K*x+b)
    end
    #Retornando os coeficientes x
    return x
end
```

Para a construção da matriz, vamos utilizar o seguinte função no julia:

```
#Função que cria uma matriz(nxn)com todos os valores da maior diagonal
iguais a -2 e as outras duas maiores iguais a 1
function criação_da_matriz(n)

   #Cria uma matriz A de zeros
    A=zeros(n,n)

    #Criando o "topo" e a "base" da matriz
    A[1,1]=-2
    A[1,2]=1
    A[n,n-1]=1
    A[n,n]=-2

    #Criando o restante
    for i= 2:(n-1)
        A[i,i]=-2
        A[i,i+1]=1
        A[i,i-1]=1
    end

   #Retorna essa matriz criada
    return A
end
```

Agora vamos passar os valores do problema

```
A = criação_da_matriz(5) #Criando matriz 5x5
x = [0; 10/6; (10/6)*2; (10/6)*3; (10/6)*4; (10/6)*5; 10] #pontos x_k
p = ((10/6)^2)*8
B = [(p*x[2])-5; p*x[3]; p*x[4]; p*x[5]; (p*x[6])-20]
```

Chamando a função de Gauss_Seidel temos:

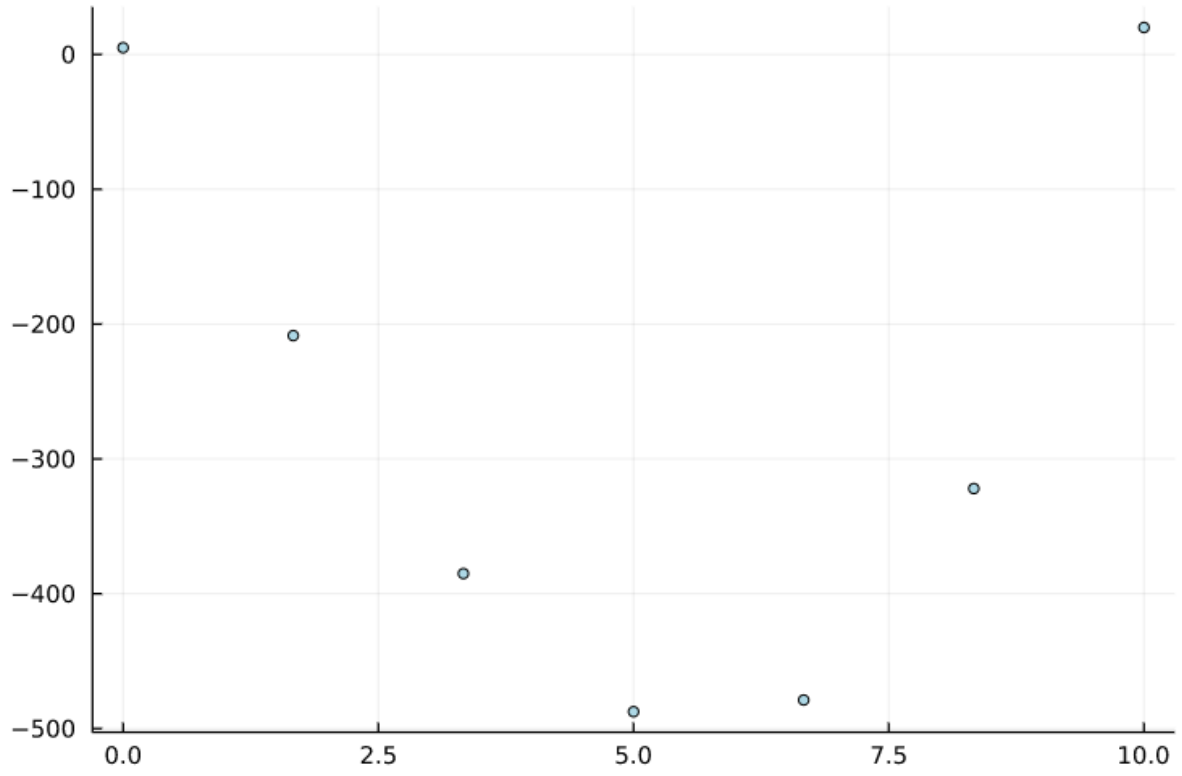
```
Gauss_Seidel(A,B)
✓ 0.4s
5x1 Matrix{Float64}:
 -208.54938271604937
 -385.06172839506166
 -487.4999999999999
 -478.8271604938271
 -322.0061728395061
```

Completando a tabela mostrada anteriormente temos os seguintes pontos:

k	x_k	y_k
0	0	5
1	10/6	-208.54938271604937
2	(10/6)*2	-385.06172839506166
3	(10/6)*3	-487.4999999999999
4	(10/6)*4	-478.8271604938271
5	(10/6)*5	-322.0061728395061
6	10	20

Plotando os pontos temos:

```
y = [5;-208.54938271604937;-385.06172839506166;-487.4999999999999;
-478.8271604938271;-322.0061728395061;20]
x = [0; 10/6; (10/6)*2; (10/6)*3; (10/6)*4; (10/6)*5; 10]
scatter(x, y, c=:lightblue, ms=3, leg=false)
```



(c) Agora vamos utilizar a interpolação polinomial de grau três para descobrir y(3.2345). Para isso, vamos utilizar a seguinte função no julia:

```
#Função que dado um conjunto de pontos, realiza uma interpolação de
grau 3 e retorna um vetor contendo os coeficientes do polinômio
interpolador calculado
function interpolação3(x,y)
#Cria a matriz V (primeira linha: todos os pontos ‘x’ elevados a 0,
#segunda linha: todos os pontos de ‘x’ elevados a 1 e terceira
#linha: todos os pontos de ‘x’ elevados a 2. Quarta linha: todos os 'x'
#elevados a 3)
    V=[x.^0 x.^1 x.^2 x.^3]
    #Calcula o sistema linear Vc = y
    c=V\y
    return c #Retorna o vetor dos coeficientes do polinômio
end
```

Sabendo que os pontos são:

```
y = [5; -208.54938271604937;-385.06172839506166;-487.4999999999999;
-478.8271604938271;-322.0061728395061;20]
x = [0; 10/6; (10/6)*2; (10/6)*3; (10/6)*4; (10/6)*5; 10]
```

Chamando a função temos:

```
c = interpolação3(x,y)
```

✓ 0.5s

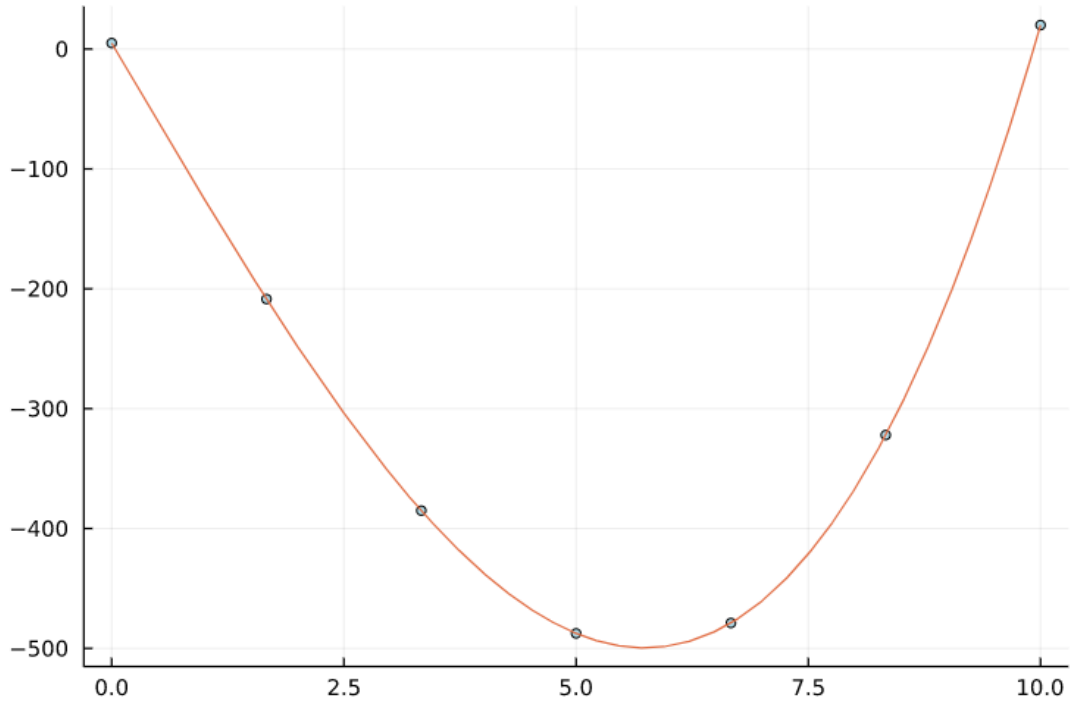
```
4-element Vector{Float64}:
     5.000000000000004
  -131.83333333333331
    -1.6012051278138663e-14
     1.3333333333333348
```

Agora vamos achar a função com os coeficientes dados:

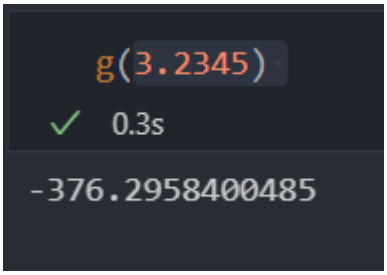
```
g(x) = c[1] + c[2]*x + c[3]*(x^2) + c[4]*(x^3)
```

Plotando o gráfico e os pontos temos:

```
scatter(x, y, c=:lightblue, ms=3, leg=false)#Pontos
plot!(g,0,10) #Gráfico
```



Agora descobrindo a g(3.2345):

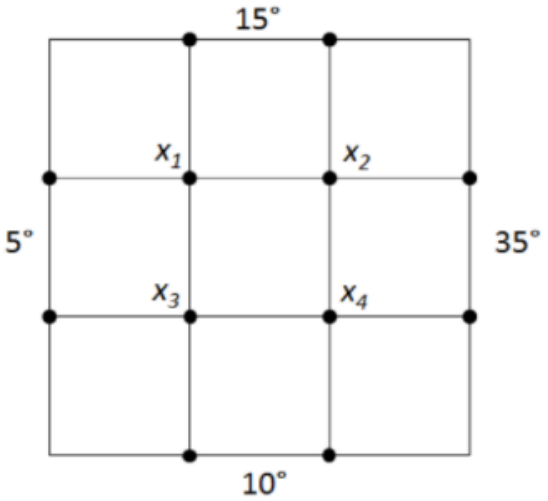


Temos que g(3.2345)= -376.2958400485

Questão 3:

Questão realizada pelos alunos Augusto Guimarães - DRE:119025393, Luiz Rodrigo Lace - DRE: 118049873 e Livia Barbosa Fonseca - DRE: 118039721.

a) Nesse exercício queremos descobrir a temperatura em diferentes lugares no interior de um lago. Sabendo que quando o calor está em equilíbrio, a temperatura em cada vértice no interior do lago é média das temperaturas dos 4 vértices vizinhos. Vamos modelar o problema dado como um sistema linear do tipo Ax = b.



Com base nos dados informados pelo problema, vamos obter o seguinte conjunto de equações:

$$x1 = \frac{x2+x3+5+15}{4}$$
$$x2 = \frac{x1+x4+15+35}{4}$$
$$x3 = \frac{x1+x4+5+10}{4}$$
$$x4 = \frac{x2+x3+35+10}{4}$$

Arrumando essas equações obtemos:

$$4x1 = x2 + x3 + 5 + 15 \quad => \quad -4x1 + x2 + x3 = -20$$
$$4x2 = x1 + x4 + 15 + 35 \quad => \quad -4x2 + x1 + x4 = -50$$
$$4x3 = x1 + x4 + 5 + 10 \quad => \quad -4x3 + x1 + x4 = -15$$
$$4x4 = x2 + x3 + 35 + 10 \quad => \quad -4x4 + x2 + x3 = -45$$

Com essas equações, podemos montar as seguintes matrizes que atendem o formato o formato Ax = B

$$\begin{pmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -20 \\ -50 \\ -15 \\ -45 \end{pmatrix}$$

b) Nesse item vamos determinar a temperatura nos 4 vértices do interior do quadrado com o LU. Para isso, vamos utilizaremos as funções resolve_triangular_superior*(A, b), resolve_triangular_inferior(A, b)* e *decomposicao_lu(A)* que foram modeladas na questão 1 para resolver o sistema pelo método lu:

Passando os valores das matrizes mencionadas anteriormente e chamando a função sistema_denso para resolver o sistema linear, temos:

```
# Função que dado A e b de um sistema do tipo Ax=b retorna x
function sistema_denso(A,b)
    #Realiza a decomposição LU
    L,U=decomposicao_lu(A)
    #Resolve matriz triangular inferior
    y=resolve_triangular_inferior(L,b)
    #Resolve matriz triangular superior
    x=sresolve_triangular_superior(U,y)
    #Retorna x de Ax=b
    return x
end
```

```
#Matriz A de Ax=b
A = [-4.0 1 1 0; 1 -4 0 1; 1 0 -4 1; 0 1 1 -4]
#Matriz b de Ax=b
b = [-20;-50;-15;-45]
```

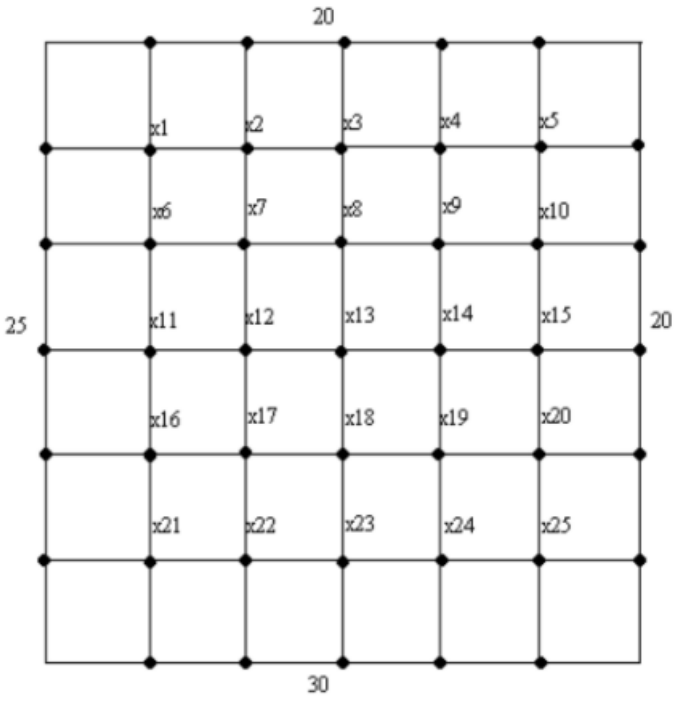
```
#Resolvendo o sistema Ax=b via LU
sistema_denso(A,b)
```

Dessa forma obtemos:

```
sistema_denso(A,b)
✓ 0.8s
4×1 Matrix{Float64}:
 13.125
 20.625
 11.875
 19.375
```

Assim temos que x1 = 13.125, x2 = 20.625, x3 = 11.875, x4 = 19.375.

c) Agora temos que determinar a temperatura dos vértices do interior do lago via método LU, sabendo que a temperatura da margem mudou e o lago foi discretamente aumentado, como mostra a figura abaixo.



Podemos observar que temos 25 pontos no interior do lago, ou seja, temos que escrever 25 equações, sendo uma equação para cada ponto do lago. Sendo assim, para facilitar, construímos duas funções no julia que nos devolve as matrizes modeladas para o problema.

Primeiramente, estudamos o padrão que a matriz A (Ax=b) estava estabelecendo e descobrimos que seria o mesmo da matriz A do exercício anterior. Em seguida, descobrimos que este é um problema clássico, quando possuímos um grid que representa um sistema físico em equilíbrio Nestes casos, encontramos um “padrão fundamental” que a matriz estudada possui, e observamos que ela segue o mesmo desenho para todas as discretizações a partir dela, tendo em mente que o número de linhas da matriz deve ser o mesmo número de colunas, sendo estes quadrados perfeitos.

Dessa forma, montamos a seguinte função que nos devolve a matriz A do sistema Ax=b.

```
Função que dado um n desejado cria uma matriz (nxn) no modelo do nosso
problema (funciona somente para n's que são quadrados perfeitos)
function criação_da_matriz(n)
   #Cria uma matriz de zeros(nxn)
    A=zeros(n,n)

    rq=Int(sqrt(n))

    #Passando os valores na primeira e última linha da matriz
    A[1,1]=-4
    A[1,2]=1
    A[n,n-1]=1
    A[n,n]=-4

    #tridiagonal
    for i= 2:(n-1)
        A[i,i]=-4
        A[i,i+1]=1
        A[i,i-1]=1

        if (i%rq == 0)
            A[i,i+1] = 0
        end
    end

    # 1's de baixo
    for i = (rq+1) : n
        A[i, i-rq] = 1
    end

    # 1's de cima
```

```
    for i = 1 : n - rq
        A[i, i+rq] = 1
    end

    # passando 1 para 0 a cada rq
    for i = rq : n-rq
        if i % rq == 0
            A[i+1,i] = 0
        end
    end

    #Retorna a matriz criada
    return A
end
```

Para facilitar, criamos também uma função chamada de matriz_resultado que constrói uma matriz nx1 com os resultados do nosso sistema linear, ou seja, estamos encontrando a matriz b de Ax=b .

```
#Função que encontra a matriz b de (Ax=b) do problema modelado (só
funciona para n's que são quadrados perfeitos)
function matriz_resultado(n)
   #Cria uma matriz de zeros(nxn)
    A=zeros(n,1)

    rq=Int(sqrt(n))

    #canto superior esquerdo
    A[1,1]  = -45

    #canto superior direito
    A[rq,1] = -40

    #canto inferior esquerdo
    A[n-rq+1,1] = -55

    #canto inferior direito
    A[n,1] = -50

    #cima
    for i = 2:rq-1
        A[i,1] = -20
    end
```

```
    #lado esquerdo
    for i = 1 : rq-2
        A[(i*rq)+1,1] = -25
    end

    #lado direito
    for i = 2 : rq-1
        A[(i*rq),1] = -20
    end

    #baixo
    for i = n-rq+2 : n-1
        A[i,1] = -30
    end

    #Retorna a matriz criada
    return A
end
```

Vamos criar as matrizes mencionadas*:
Aqui estamos criando a matriz A (25x25):

```
A = criação_da_matriz(25)
```

```
25×25 Matrix{Float64}:
 -4.0   1.0   0.0   0.0   0.0   1.0  …   0.0   0.0   0.0   0.0   0.0   0.0
  1.0  -4.0   1.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   1.0  -4.0   1.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   1.0  -4.0   1.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   1.0  -4.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  1.0   0.0   0.0   0.0   0.0  -4.0  …   0.0   0.0   0.0   0.0   0.0   0.0
  0.0   1.0   0.0   0.0   0.0   1.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   1.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   1.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   1.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   1.0  …   0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      1.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0  …   0.0   1.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   1.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   1.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      1.0   0.0   0.0   0.0   1.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0     -4.0   0.0   0.0   0.0   0.0   1.0
  0.0   0.0   0.0   0.0   0.0   0.0  …   0.0  -4.0   1.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   1.0  -4.0   1.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   1.0  -4.0   1.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   1.0  -4.0   1.0
  0.0   0.0   0.0   0.0   0.0   0.0      1.0   0.0   0.0   0.0   1.0  -4.0
```

**Utilizando o Júpiter Notebook nessa etapa, pois no VsCode limita ainda mais a visualização das matrizes*

Agora, vamos criar a matriz b (1x25):

```
b = matriz_resultado(25)
```

```
: 25×1 Matrix{Float64}:
  -45.0
  -20.0
  -20.0
  -20.0
  -40.0
  -25.0
    0.0
    0.0
    0.0
  -20.0
  -25.0
    0.0
    0.0
    0.0
  -20.0
  -25.0
    0.0
    0.0
    0.0
  -20.0
  -55.0
  -30.0
  -30.0
  -30.0
  -50.0
```

Para solucionarmos esse sistema via método LU é necessário o uso das funções que foram criadas no exercício 1, resolve_triangular_superior*(A, b), resolve_triangular_inferior(A, b)* ***e decomposicao_lu(A)*** e a função *sistema_denso(A,b)* utilizada no item anterior*:*

```
In [19]: A = criação_da_matriz(25)
         b = matriz_resultado(25)
         sistema_denso(A,b)
```

```
Out[19]: 25×1 Matrix{Float64}:
          22.656565656565654
          21.762286324786324
          21.28651903651904
          20.89359945609946
          20.46969696969697
          23.863976301476303
          23.106060606060602
          22.49019036519036
          21.818181818181817
          20.98518842268842
          24.69327894327894
          24.30778943278943
          23.75
          22.903749028749036
          21.65287490287491
          25.601350038850036
          25.68181818181818
          25.29827117327117
          24.393939393939394
          22.722562160062164
          27.030303030303024
          27.519862082362078
          27.367327117327115
          26.65117521367521
          24.84343434343434
```

Dessa forma, podemos observar que encontramos os valores de x_1 até x_k para o nosso problema. Sendo o primeiro número retornado o x_1 e o x_{25} o último.

d)Nessa questão queremos descobrir o maior número de nós que conseguimos discretizar e rodar na nossa máquina em menos de 2 minutos usando as funções criadas para a resolução de um sistema via método LU.

Para descobrirmos o tempo que a discretização em questão está demorando, vamos utilizar a seguinte biblioteca:

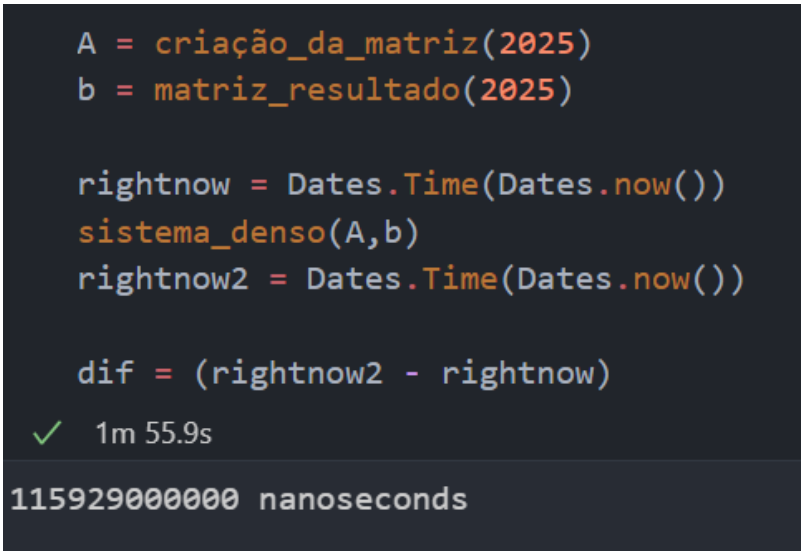
```
using Dates
#Importando a biblioteca
import Dates
```

Para calcularmos o tempo que a função está demorando, vamos marcar o tempo do início do processo e quando o processo foi finalizado e em seguida realizar um tempo menos o outro, essa diferença será o tempo que a função está demorando para ser rodada.

Vamos começar chutando n = 2025 (lembrando que n é um quadrado perfeito):

```
A = criação_da_matriz(2025)
b = matriz_resultado(2025)

#Inicio do processo
rightnow = Dates.Time(Dates.now())

#Chamando a função que calcula o sistema
sistema_denso(A,b)

#Final do processo
rightnow2 = Dates.Time(Dates.now())

#Duração
dif = (rightnow2 - rightnow)
```

Obtemos:



Sabendo que 1 minuto vale 6e+10 nanosegundo, temos:

```
115929000000/6e+10
[28]  ✓ 0.5s
...   1.93215
```

Encontramos que com n = 2025, nossa função demora 1,93215 minutos.

Agora vamos tentar com n = 2116:

```
A = criação_da_matriz(2116)
b = matriz_resultado(2116)

rightnow = Dates.Time(Dates.now())
sistema_denso(A,b)
rightnow2 = Dates.Time(Dates.now())

dif = (rightnow2 - rightnow)
✓ 2m 19.8s
139792000000 nanoseconds
```

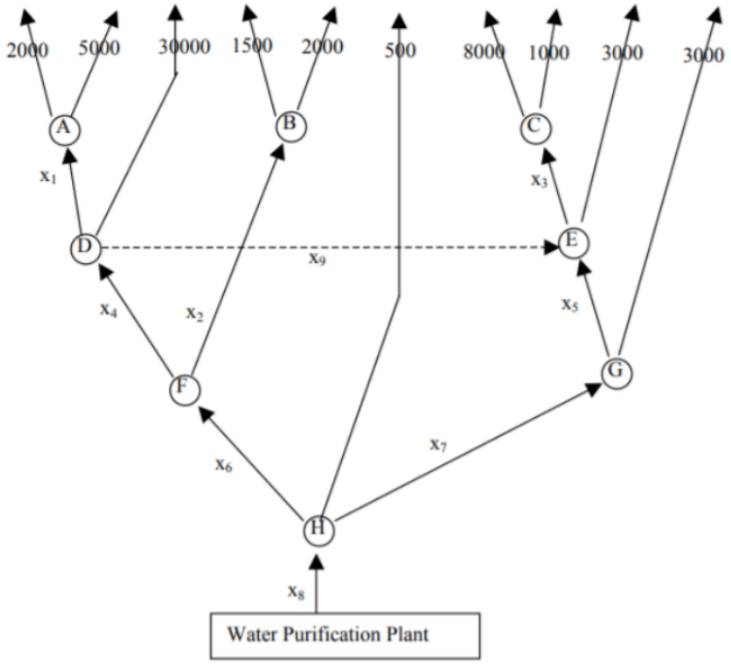
Transformando em minuto temos:

```
139792000000/6e+10
✓ 0.2s
2.3298666666666668
```

Podemos observar que com n = 2116, o tempo estipulado foi maior do que 2 minutos. Dessa forma, o maior numero de nos que conseguimos discretizar em menos de 2 minutos é 2025.

Questão 4:

a) Nesse exercício, vamos fazer a modelagem do seguinte sistema abaixo e o resolver, sem o cano x9, com ajuda do LU.



A modelagem vai ser feita de forma que iremos analisar no ponto de cada cidade quanto de água entra e o quanto sai, criando assim, nossos sistemas lineares:

Cidade A => x1 = 2000 + 5000
Cidade B => x2 = 2000 + 1500
Cidade C => x3 = 8000 + 1000
Cidade D => x4 = x1 + 30000
Cidade E => x5 = x3 + 3000
Cidade F => x6 = x2 + x4
Cidade G => x7 = x5 + 3000
Cidade H => x8 = x6 + x7

Podemos observar que:

x1 = 7000
x2 = 3500
x3 = 9000

Com esses valores conseguimos substituir e encontrar todos os valores dos canos, sem a necessidade de utilizar o método de LU, visto que temos um sistema de equações muito simples.

x4 = x1 + 30000
x4 = 7000+ 30000 = 37000

x5 = x3 + 3000
x5 = 9000+ 3000 = 12000

x6 = x2 + x4
x6 = 3500 + 37000 = 40500

x7 = x5 + 3000
x7 = 12000 + 3000 = 15000

x8 = x6 + x7 + 500
x8 = 40500 + 15000 + 500 = 56000

Agora vamos fazer esse mesmo cálculo com o método de fatoração Lu. Para isso, vamos construir as matrizes do sistema do tipo Ax=b:

como foi pedido no exercício. Para isso, utilizaremos as funções resolve_triangular_superior*(A, b), resolve_triangular_inferior(A, b)* e *decomposicao_lu(A)* que foram modeladas na questão 1 e a função *sistema_denso(A,b) c*riada no exercício 3 para resolver esse sistema pelo método lu:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} 7000 \\ 3500 \\ 9000 \\ 30000 \\ 3000 \\ 0 \\ 3000 \\ 500 \end{pmatrix}$$

```
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]

B =[7000;3500;9000;30000;3000;0;3000;500]
sistema_denso(A,B)
```

✓ 0.5s

```
8×1 Matrix{Float64}:
  7000.0
  3500.0
  9000.0
 37000.0
 12000.0
 40500.0
 15000.0
 56000.0
```

Podemos observar que x1 = 7000.0, x2 = 3500.0, x3 = 9000.0, x4 = 37000.0, x5 = 12000.0, x6 = 40500.0, x7 = 15000.0, x8 = 56000.Obtemos o mesmo resultado anteriormente.

(b) Agora vamos fazer a modelagem utilizando o cano x9 e tentar utilizar o método de lu para resolver. Para criarmos os sistemas, vamos utilizar o mesmo processo descrito anteriormente:

Cidade A
x1 = 2000+5000
x1 = 7000

Cidade B
x2 = 1500+2000
x2 = 3500

Cidade C

x3 = 8000+1000
x3 = 9000

Cidade D
x4 = x1 +x9+30000
x4 - x1 - x9 = 30000

Cidade E
x5+x9 = x3+3000
x5+x9-x3 = 3000

Cidade F
x6 = x4 + x2
x6-x4-x2=0

Cidade G
x7 = x5+3000
x7-x5=3000

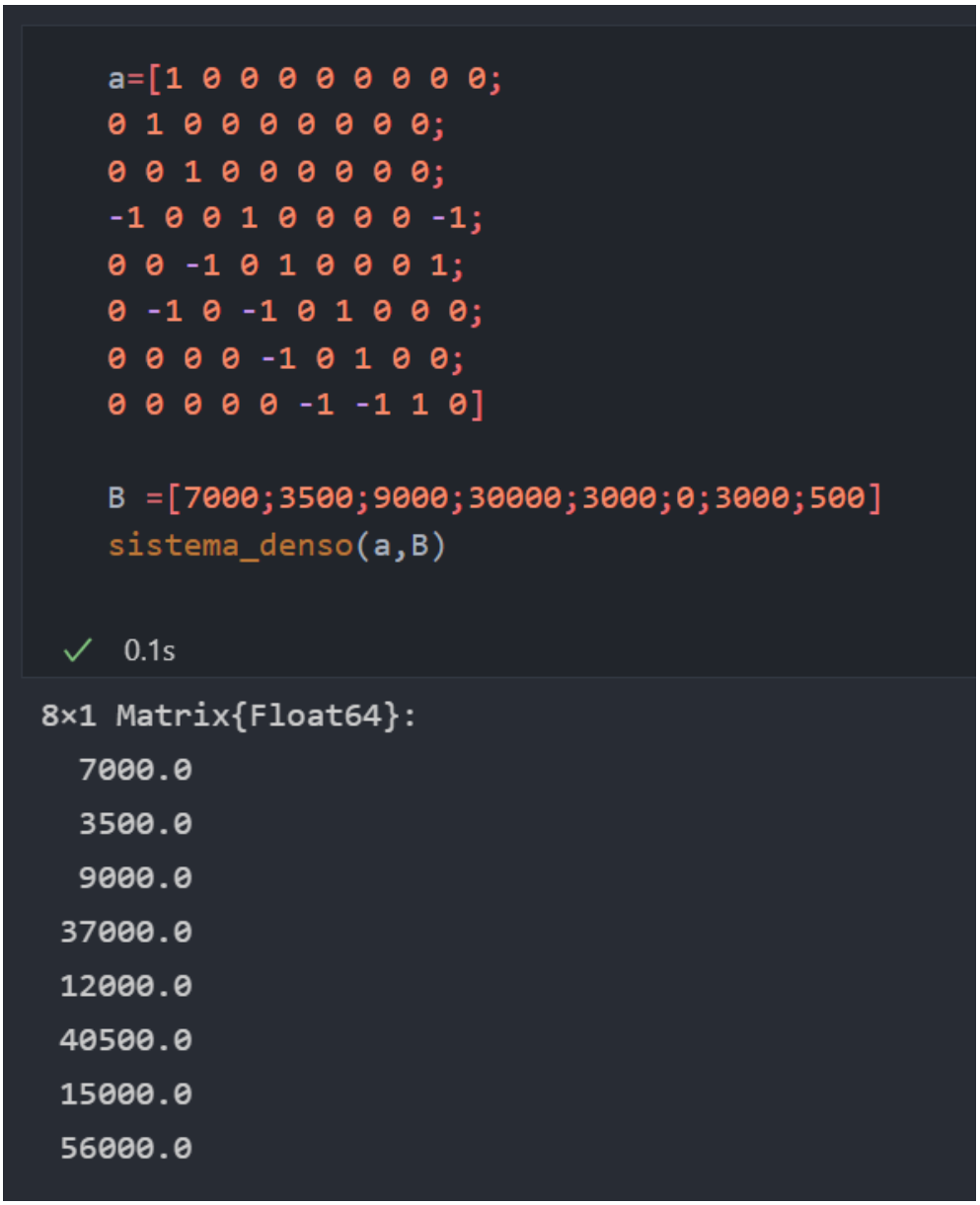
Cidade H
x8 = x6 + x7 + 500
x8-x6-x7=500

Montando a matriz desse sistema, temos:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} 7000 \\ 3500 \\ 9000 \\ 30000 \\ 3000 \\ 0 \\ 3000 \\ 500 \end{pmatrix}$$

Vamos utilizar o método de Lu para resolver esse sistema. Podemos observar que não iremos conseguir resolvê-lo, visto que temos uma matriz de 9 colunas por 8 linhas e uma matriz de 8 linhas por 1 coluna. Dessa forma, não conseguiremos resolver nosso problema de forma desejada. Vamos descobrir o que acontece quando passamos as funções do júlia para resolver essa

questão. Utilizando as mesmas funções mencionadas anteriormente para resolver um sistema via método lu, temos:



```
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]

B =[7000;3500;9000;30000;3000;0;3000;500]
sistema_denso(a,B)
```

✓ 0.1s

```
8×1 Matrix{Float64}:
  7000.0
  3500.0
  9000.0
 37000.0
 12000.0
 40500.0
 15000.0
 56000.0
```

Podemos observar que não encontramos um valor para x9 e que os valores dos demais canos continuaram iguais ao problema anterior. Sendo assim, não conseguimos encontrar o resultado desse sistema do tipo Ax=b via método LU.