Aluno: Luiz Rodrigo Lacé Rodrigues
DRE: 118049873

Questão realizada pelos alunos:
Augusto Guimarães - DRE:119025393
Luiz Rodrigo Lacé Rodrigues - DRE: 118049873
Livia Barbosa Fonseca - DRE: 118039721.

1. No primeiro exercício vamos começar a criar uma biblioteca de cálculo Numérico. Nessa questão você pode trabalhar individualmente ou com um grupo de até 3 alunos.

(a) Sua biblioteca deve estar bem comentada (grande parte da avaliação vai ser baseada nos comentários).
(b) Além de completar o código, não esqueça de escrever os objetivos das funçoes (especificações) e suas entradas e saídas.
(c) Forneça também 3 exemplos para cada função.
(d) Generalize os códigos das funções abaixo que fizemos na aula 15 para matrizes n × n (não precisa se preocupar com divisões por zero nem com troca de linhas ou colunas).

i. resolve diagonal

In [2]:
```
# 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
    for i = 1:n
        x[i] = b[i] / A[i, i]
    end

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

Teste 1:

```
In [7]: #Teste da matriz diagonal
        #criando matriz A
        Teste1 = [2 0 0 ;0 3 0; 0 0 4]
        #Gerando valores aleatorios para a matriz x
        x_verdadeiro=randn(3,1)

        #Multiplicando os numeros aleatorios pela matriz A (b=AX)
        b= Teste1*x_verdadeiro

        #Aplicando a função
        x_calculado=resolve_diagonal(Teste1 ,b)

        #Conferindo se o valor é o esperado
        norm(x_verdadeiro-x_calculado) < 0.000000000001
Out[7]: true
```

Observamos que o valor esperado é encontrado

Teste 2:

```
: #Testando a função matriz diagonal
  #Criando a matriz A
  Teste2= [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]

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

  #Multiplicando os números aleatorios pela matriz A (b=Ax)
  b= Teste2*x_verdadeiro

  #Aplicando a função
  x_calculado=resolve_diagonal(Teste2,b)


  #Verificando se o valor é o esperado
  norm(x_verdadeiro-x_calculado) < 0.000000000001
: true
```

Teste 3

```
#Testando a frunção matriz diagonal
#criando matriz A
Teste3 = [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 ]

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

#Multiplicando os números aleatorios pela matriz A (b=Ax)
b= Teste3*x_verdadeiro

#Aplicando a função
x_calculado=resolve_diagonal(Teste3,b)

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

```
true
```

ii. resolve triangular superior

In [9]:
```
# 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
```

Out[9]:
```
resolve_triangular_superior (generic function with 1 method)
```

Realizando os testes:

Teste 1:

```
#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)

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

```
true
```

Teste 2:

```
#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)

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

```
true
```

Teste 3:

A = [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]

```
#matriz triangular superior
A = [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

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

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

#Chamando a função
x_calculado=resolve_triangular_superior(A,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
```

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

Realizando testes:

Teste 1:

```
In [19]: #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
```

```
Out[19]: true
```

Teste 2:

```
#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
```

Teste 3:

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

# 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

```
#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
```

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

Realizando os testes:

Teste1:

```
# Matriz que vamos decompor
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)
```

```
([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])
```

```
#matriz l
l
```

```
4×4 Array{Float64,2}:
 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
```

```
#matriz U
U
```

```
4×4 Array{Int64,2}:
 2  3  1  1
 0  1  2  1
 0  0  2  1
 0  0  0  2
```

Teste 2:

In [45]:
```
# Matriz que vamos decompor
k = [3 6 4 2]

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

Out[45]:
```
([1.0], [3 6 4 2])
```

In [46]:
```
#matriz l
l
```

Out[46]:
```
1×1 Array{Float64,2}:
 1.0
```

In [47]:
```
#matriz U
U
```

Out[47]:
```
1×4 Array{Int64,2}:
 3  6  4  2
```

Teste3:

```
# Matriz que vamos decompor
k = [1 2 3; 5 4 3; 7 5 4]

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

```
([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])
```

```
#matriz l
l
```

```
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 5.0  1.0  0.0
 7.0  1.5  1.0
```

```
#matriz U
U
```

```
3×3 Array{Int64,2}:
 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?

Complexidade do algoritmo O(n^3)

```
#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{Int64}(I,n,n)

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

    #Matriz triangular
    y = L\identidade
    x=U\y
    #Retorna a inversa de t
    return x
end
```

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

Testes:

```
#Matriz A
a = [1 4 -3; 2 7 0; -1 8 -9]
#Matriz inversa de A
MatrizInversa(a)
```

```
3×3 Array{Float64,2}:
  1.05      -0.2  -0.35
 -0.3        0.2   0.1
 -0.383333   0.2   0.0166667
```

```
#Matriz A
a = [1 -2; 4 6]
#Matriz inversa de A
MatrizInversa(a)
```

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

```
#Matriz A
a = [1 2; 3 4]
#Matriz inversa de A
MatrizInversa(a)
```

```
2×2 Array{Float64,2}:
 -2.0   1.0
  1.5  -0.5
```

2. Dado o PVC y″(x) = 4x com y(0) = 5 e y(10) = 20

(a) Monte o sistema linear que aproxima pelo método de diferenças finitas com n = 6 intervalos na discretização.

Primeiramente temos os seguintes dados no enunciado:
y″(x) = 4x
y(0) = 5
y(10) = 20
n = 6

Sabendo que as fórmulas para obter y’(x) e y’’(x) são

$$y'(x) = \frac{Y_{i+1} - Y_{i-1}}{2h}$$

$$y''(x) = \frac{Y'_{i+1} - Y'_{i-1}}{2h}$$

$$y''(x) = \frac{\frac{Y_{i+1} - Y_i}{h} - \frac{Y_i - Y_{i-1}}{h}}{2h}$$

$$y''(x) = \frac{Y_{i+1} - 2Y_i + Y_{i-1}}{2h^2}$$

Sendo h o tamanho do intervalo.

Como temos o número de intervalos, podemos achar o tamanho de cada um:

$$n = \frac{b-a}{h} -> h = \frac{b-a}{n} -> h = \frac{10-0}{6} = \frac{10}{6}$$

Fazendo agora a devida substituição de y''(x) em sua expressão com o que temos do enunciado, chegamos em:

$$4x = \frac{Y_{i+1} - 2Y_{i-1} + Y_{i-1}}{2h^2} -> 8xh^2 = Y_{i+1} - 2Y_{i-1} + Y_{i-1}$$

$$-> Y_{i+1} - 2Y_{i-1} + Y_{i-1} = 8x(\frac{10}{6})^2$$

A partir dessa expressão vamos montar o sistema linear, temos apenas que seguir os valores da seguinte tabela:

i	Xi	Yi
0	0	5
1	(10/6) * 1	Y1
2	(10/6) * 2	Y2
3	(10/6) * 3	Y3
4	(10/6) * 4	Y4
5	(10/6) * 5	Y5
6	10	20

Como os valores que não conhecemos são Y1 até Y5, é justamente para descobri-los que vamos montar o sistema linear.

As expressões terão as seguintes caras:

i=1

$$Y(x_2) - 2 * Y(x_1) + Y(x_0) = 8 * x_1 * (\frac{10}{6})^2$$

Como $Y(x_0) = Y(0) = 5$, temos

$$Y(x_2) - 2 * Y(x_1) + 5 = 8 * x_1 * (\frac{10}{6})^2$$

$$Y(x_2) - 2 * Y(x_1) = 8 * x_1 * (\frac{10}{6})^2 - 5$$

i=2

$$Y(x_3) - 2 * Y(x_2) + Y(x_1) = 8 * x_2 * (\frac{10}{6})^2$$

i=3

$$Y(x_4) - 2 * Y(x_3) + Y(x_2) = 8 * x_3 * (\frac{10}{6})^2$$

i=4

$$Y(x_5) - 2 * Y(x_4) + Y(x_3) = 8 * x_4 * (\frac{10}{6})^2$$

i=5

$$Y(x_6) - 2 * Y(x_5) + Y(x_4) = 8 * x_5 * (\frac{10}{6})^2$$

Como $Y(x_6) = Y(10) = 20$, temos

$$20 - 2 * Y(x_5) + Y(x_4) = 8 * x_5 * (\frac{10}{6})^2$$

$$-2 * Y(x_5) + Y(x_4) = 8 * x_5 * (\frac{10}{6})^2 - 20$$

Com as devidas expressões formadas, podemos jogá-las em uma matriz para montar o sistema linear, temos:

Como a expressão: $8 * x * (\frac{10}{6})^2$ se repete muitas vezes, podemos reduzi-la para P(x).

$$\begin{vmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{vmatrix} * \begin{vmatrix} y1 \\ y2 \\ y3 \\ y4 \\ y5 \end{vmatrix} = \begin{vmatrix} P(x1) - 5 \\ P(x2) \\ P(x3) \\ P(x4) \\ P(x5) - 10 \end{vmatrix}$$

(b) Resolva o sistema linear obtido no item anterior no Julia.

Para resolver esse sistema linear, podemos utilizar o método Gauss_Seidel, que recebe duas matrizes e que resolve sistema lineares de forma iterativa,no qual necessita um “chute inicial”. A partir do chute inicial, esse método tenta aproximar o valor anterior até o proximo. Quando xk está proximo de xk+1, ele para e retorna os coeficientes de Ax=B

Podemos dividir a matriz A em duas (M e K), sendo M inversível:

(M - K)x = B
Mx - Kx = B
Mx = Kx+B

Como M é inversível, temos então temos que:
x = M^{-1} (Kx + B)

Passando esse raciocínio para o julia, temos:

In [ ]:
```
function Gauss_Seidel(A,b)
    #Tamanho da matriz A
    n,=size(A)

    #Diagonal da matriz A
    M = Diagonal(A)

    #Diagonal da A zerada
    K=A-M

    #chute inicial
    x = randn(n,1)

    #Aproximando de Ax = b
    for i=1:500
        x = inv(M)*(-K*x+b)
    end
    #retorna os coeficientes
    return x
end
```

Para construirmos a nossa matriz A (a primeira mostrada anteriormente), vamos utilizar o método criação_da_matriz:

In [1]:
```
function criação_da_matriz(n)
    A=zeros(n,n)

    #"manual"
    A[1,1]=-2
    A[1,2]=1
    A[n,n-1]=1
    A[n,n]=-2

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

Esse método recebe o tamanho da matriz a retorna uma matriz nxn com a estrutura mostrada anteriormente.

Montando a matriz B:

In [1]:
```
#calculando os pontos de x, sabendo que y(0) = 5 e y(10) = 20 e n=6. Como h=(b-a)/n; h=10/6
x = [0; 10/6; (10/6)*2; (10/6)*3; (10/6)*4; (10/6)*5; 10]
```

Out[1]:
```
7-element Array{Float64,1}:
  0.0
  1.6666666666666667
  3.3333333333333335
  5.0
  6.666666666666667
  8.333333333333334
 10.0
```

In [15]:
```
p = ((10/6)^2)*8
```

Out[15]:
```
22.222222222222225
```

In [16]:
```
B = [(p*x[2])-5; p*x[3]; p*x[4]; p*x[5]; (p*x[6])-20]
```

Out[16]:
```
5-element Array{Float64,1}:
  32.037037037037045
  74.07407407407409
 111.11111111111113
 148.14814814814818
 165.18518518518522
```

A nossa matriz A terá a seguinte cara:

In [4]:
```
A = criação_da_matriz(5)
```

Out[4]:
```
5×5 Array{Float64,2}:
 -2.0   1.0   0.0   0.0   0.0
  1.0  -2.0   1.0   0.0   0.0
  0.0   1.0  -2.0   1.0   0.0
  0.0   0.0   1.0  -2.0   1.0
  0.0   0.0   0.0   1.0  -2.0
```

Agora basta passar as matrizes A e B para o método Gauss_Seidel

```
In [18]: Gauss_Seidel(A,B)
```

```
Out[18]: 5x1 Array{Float64,2}:
          -208.54938271604937
          -385.06172839506166
          -487.4999999999999
          -478.8271604938271
          -322.0061728395061
```

Achamos então que:

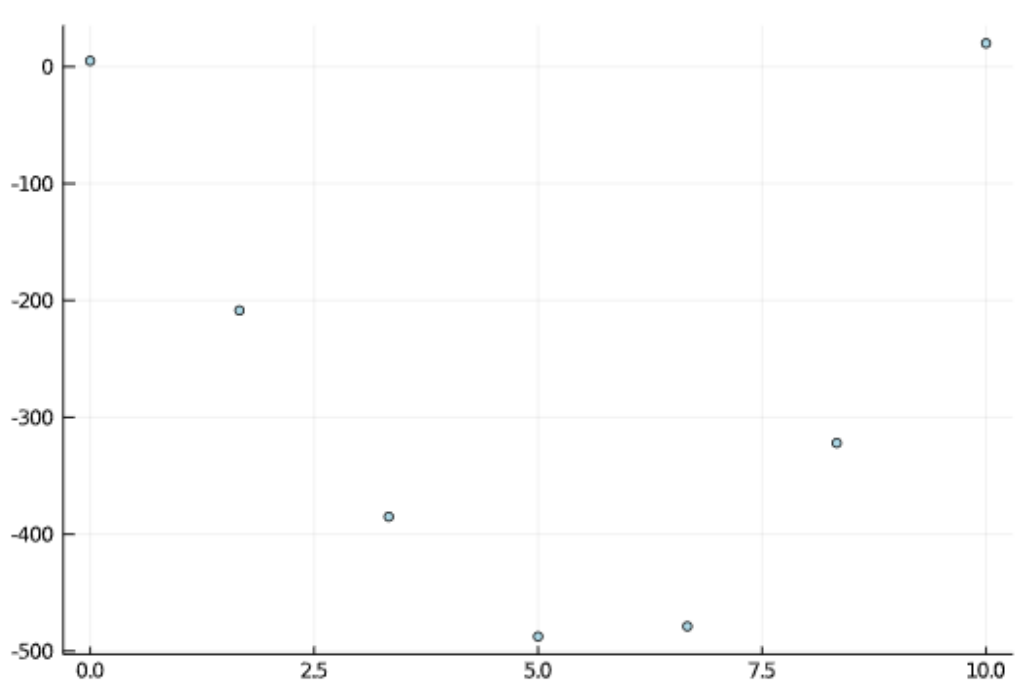
Y1= -208.54938271604937
Y2= -385.06172839506166
Y3= -487.4999999999999
Y4= -478.8271604938271
Y5= -322.0061728395061

(c) Use interpolação polinomial (pode ser com grau = 2 ou grau = 3)
para descobrir y(3.2345).

Passando os pontos de x e y, que descobrimos no exercício anterior, em vetores e plotando:

```
In [53]: x = [0; 10/6; (10/6)*2; (10/6)*3; (10/6)*4; (10/6)*5; 10]          #pontos de x
         y = [ 5;  -208.54938271604937; -385.06172839506166; -487.4999999999999; -478.8271604938271; -322.0061728395061; 20] #pontos de y
         scatter(x, y, c=:lightblue, ms=3, leg=false) #plotando os pontos
```

Out[53]:



Para calcular a interpolação de grau 2, temos o método interpolação2:

In [47]:
```
# fazendo interpolação de grau 2
function interpolação2(x,y)
    #criar a matriz V
    V=[x.^0 x.^1 x.^2]   # calcula a potencia ponto a ponto de x
    c=V\y                #resolve o sistema linear
    return c             #retorna os coeficientes
end
```

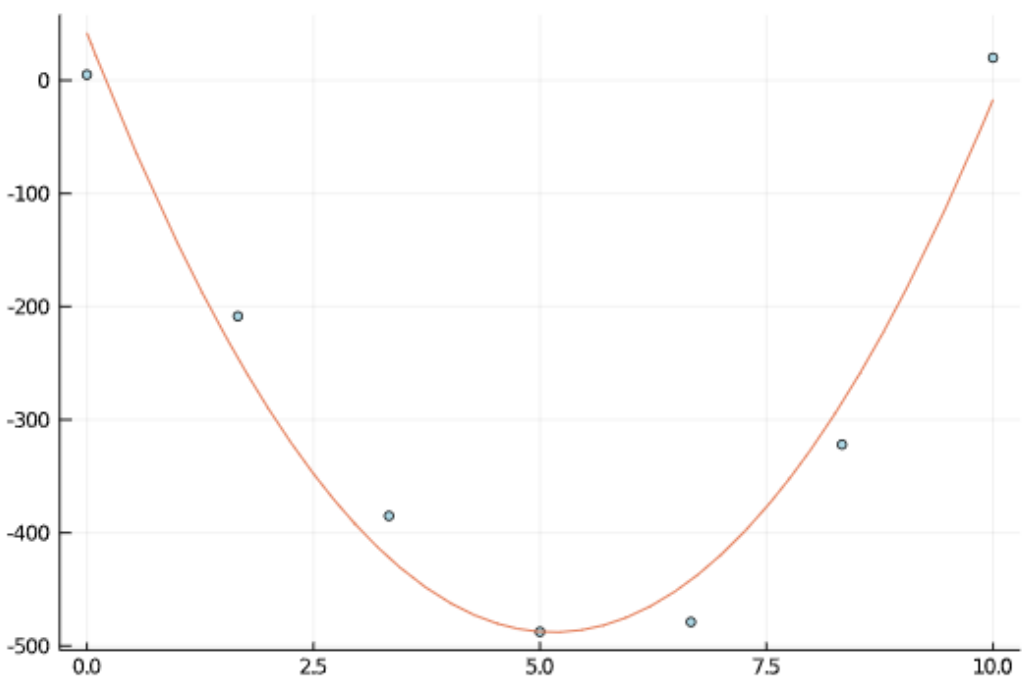
Pegando os coeficientes retornados, montando um polinômio (poli2) e plotando, temos:

In [49]:
```
c2 = interpolação2(x,y)
poli2(x) = c2[1] + c2[2]*x + c2[3]*x^2
```

Out[49]:
```
poli2 (generic function with 1 method)
```

In [50]:
```
scatter(x, y, c=:lightblue, ms=3, leg=false)
plot!(poli2,0,10)
```

Out[50]:



Fazendo o mesmo para interpolação de grau 3, temos o método interpolação3:

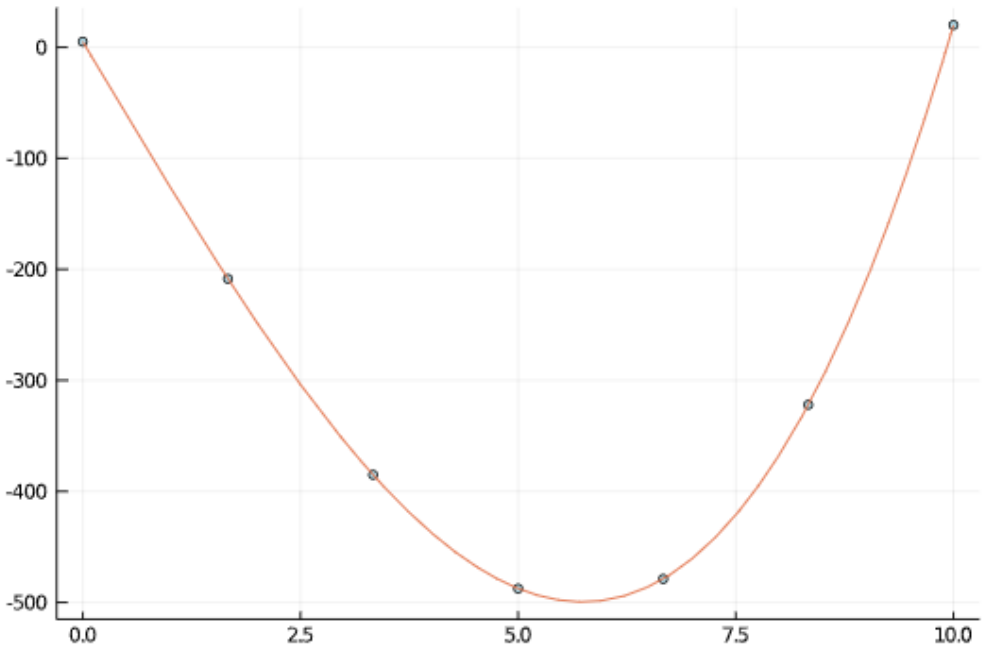
In [48]:
```
# fazendo interpolação de grau 3
function interpolação3(x,y)
    #criar a matriz V
    V=[x.^0 x.^1 x.^2 x.^3]  # calcula a potencia ponto a ponto de x
    c=V\y                    #resolve o sistema linear
    return c #retorna os coeficientes
end
```

Montando o polinômio (poli3) e plotando:

```
In [51]: c3 = interpolação3(x,y)
         poli3(x) = c3[1] + c3[2]*x + c3[3]*x^2 + c3[4]*x^3
Out[51]: poli3 (generic function with 1 method)
```

```
In [52]: scatter(x, y, c=:lightblue, ms=3, leg=false)
         plot!(poli,0,10)
```

Out[52]:

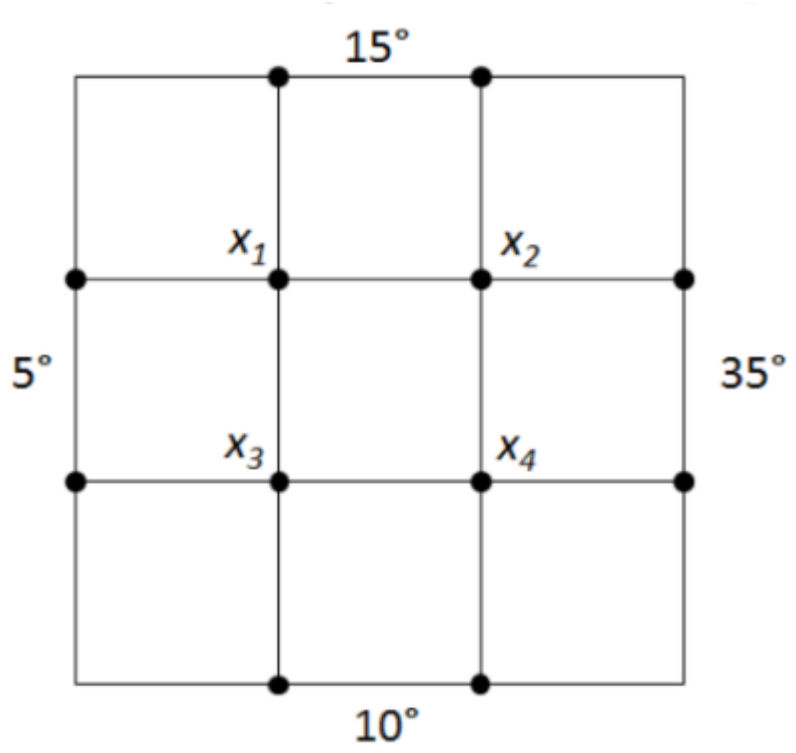


Como a interpolação de grau 3 se aproxima mais dos pontos vamos utilizar esse método para calcular y(3.2345), ou seja, vamos calcular poli3(3.2345).

```
In [54]: poli3(3.2345)
Out[54]: -376.29584004849994
```

Questão realizada pelos alunos:
Augusto Guimarães - DRE:119025393
Luiz Rodrigo Lacé Rodrigues - DRE: 118049873
Livia Barbosa Fonseca - DRE: 118039721.

3. Problema da temperatura de um lago - Nessa questão você pode trabalhar individualmente ou com um grupo de até 3 alunos.
Queremos descobrir a temperatura em diferentes lugares no interior de um lago (vértices x1, x2, x3 e x4), mas só conseguimos medir a temperatura de 5, 10, 15 e 35 graus Celsius nas margens (laterais do quadrado na figura abaixo). Quando o calor está em equilíbrio, a temperatura em cada vértice no interior do lago é a média das temperaturas dos 4 vértices vizinhos.



(a) Modele o problema como um sistema linear Ax = b.

A partir do que foi enunciado no problema, podemos modelar as equações da seguinte forma:

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

Manipulando as equações, nós temos o seguinte sistema linear:

$$4x1 = x2 + x3 + 5 + 15$$
$$4x2 = x1 + x4 + 15 + 35$$
$$4x3 = x1 + x4 + 5 + 10$$
$$4x4 = x2 + x3 + 35 + 10$$

=

$$-4x1 + x2 + x3 = -20$$
$$-4x2 + x1 + x4 = -50$$
$$-4x3 + x1 + x4 = -15$$
$$-4x4 + x2 + x3 = -45$$

Dessa forma, vamos montar a matriz do formato Ax = B, para encontrarmos os coeficientes x.

```
|-4.0 1 1 0 |     | x1 |     | -20  |
|   1 -4 0 1|  *  | x2 |  =  |  -50  |
|   1 0 -4 1|     | x3 |     | -15  |
|  0 1 1 -4 |     | x4 |     | -45  |
```

(b) Determine a temperatura nos 4 vértices do interior do quadrado com o LU.

Para esse exercício vamos encontrar os valores de x com o método LU.
Utilizaremos então os métodos ***resolve_triangular_inferior(A, b), resolve_triangular_superior(A, b) e decomposicao_lu(A)*** *do* exercício 1 dessa lista:

Vamos utilizar o método sistema_denso(A,B) que recebe as matrizes A e B e retorna os valores de x:

```
# 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=sistema_triangular_inferior(L,b)

    #Resolve matriz triangular superior
    x=sistema_triangular_superior(U,y)

    #Retorna x de Ax=b
    return x
end
```

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

Colocando os valores da nossa matriz A e B no vetor e chamando as matrizes no método **sistema_denso(A,b)** no julia:

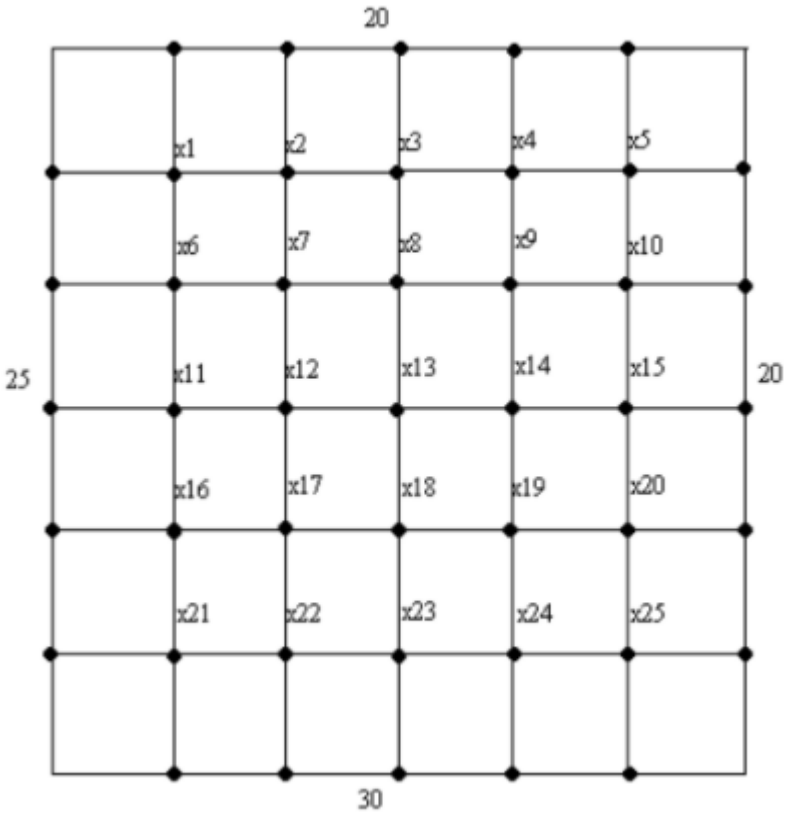
```
#Matriz A de Ax=b
A = [-4.0 1 1 0; 1 -4 0 1; 1 0 -4 1; 0 1 1 -4]
#Matriz b de Ax=b
b = [-20;-50;-15;-45]

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

```
4×1 Array{Float64,2}:
 13.125
 20.625
 11.875
 19.375
```

Temos então que:
x1=13.125
x2=20.625
x3=11.875
x4=19.375

(c) Agora a temperatura das margens mudou e queremos discretizar o lago ainda mais (figura abaixo). Determine a temperatura dos vértices do interior do quadrado com o LU.



Podemos observar que temos 25 pontos no interior do lago, ou seja, logo teremos 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. Tanto a matriz A quanto a B

Estudamos o padrão que a matriz A (Ax=b) estava estabelecendo e descobrimos que seria o mesmo da matriz A do exercício anterior. 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.

O método para a matriz A ficou da seguinte forma:

```
#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
```

Desenvolvemos também o método chamada de matriz_resultado(n) 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
```

Para solucionarmos esse sistema via método LU é necessário o uso dos métodos 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*:*

```
A = criação_da_matriz(25)
B = matriz_resultado(25)
sistema_denso(A,B)
```

```
25×1 Array{Float64,2}:
 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
```

Sendo o primeiro desses valores x1 e o último x25.

Criando a matriz A com 25 pontos:

```
A = criação_da_matriz(25)
```

```
25×25 Array{Float64,2}:
 -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
```

Criando a matriz B para 25 pontos:

```
B = matriz_resultado(25)
```

```
25×1 Array{Float64,2}:
 -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
```

(d) Você consegue discretizar com mais nós? Qual é o maior número de nós que você consegue discretizar e rodar na sua máquina em menos de 2 minutos usando LU.

Para a tomada de tempo, vamos utilizar a seguinte biblioteca Dates

Para calcularmos o tempo que o método sistema_denso está demorando, vamos marcar o tempo do início do processo e quando o processo foi finalizado, diferença será o tempo que a função está demorando para ser rodada

Para n = 2116 nós, teremos:

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

#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)
```

```
108140000000 nanoseconds
```

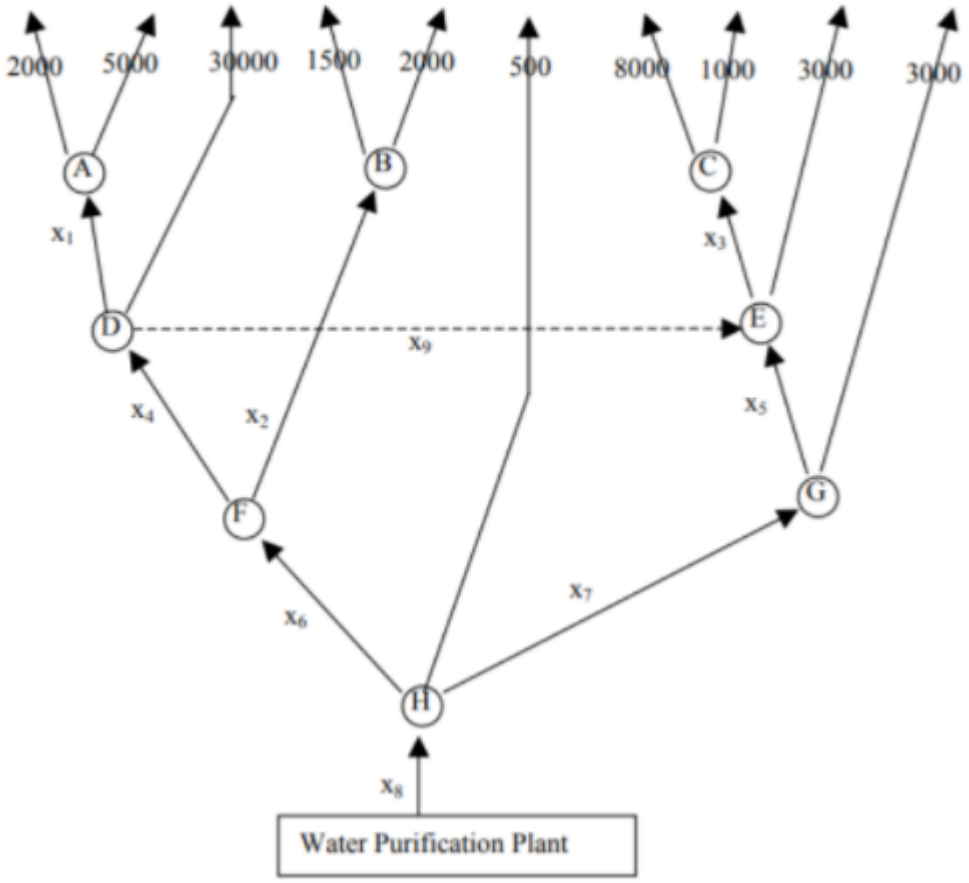
Sabendo que 6e+10 nanosegundos equivale a 1 min, temos:

```
108140000000 /60000000000
```

```
1.8023333333333333
```

Logo, é possível calcular 2116 em 1 minuto e 48 segundos. Para mais pontos, o tempo extrapola 2 minutos de execução

4. Problema de distribuição de água. Uma companhia de distribuição de água recebeu as novas demandas de fluxo, medidas em litros por minuto (números no topo da figura), dos bairros que ela atende para 2022. Ela precisa determinar o fluxo de água em cada cano (as arestas da figura).



(a) Faça a modelagem sem o cano pontilhado x9 e resolva o sistema linear com o decomposição LU.

Fazendo a modelagem de cada ponto, somando o que entra e diminuindo o que sai:

A: x1 - 7000
B: x2 - 3500
C: x3 - 9000
D: x4 - x1 - 30000
E: x5 - x3 - 3000
F: x6 - x4 - x2
G: x7 - x5 - 3000
H: x8 - x6 - x7 - 500

Ou seja, teremos um sistema onde:
x1 = 7000
x2 = 3500
x3 = 9000
x4 - x1 = 30000
x5 - x3 = 3000
 x6 - x4 - x2 = 0
x7 - x5 = 3000
x8 - x6 - x7 = 500

Montando as matrizes teremos:

```
|1 0 0 0 0 0 0 0    |      |x1|         |7000  |
| 0 1 0 0 0 0 0 0   |      |x2|         |3500  |
| 0 0 1 0 0 0 0 0   |   *  |x3|     =   |9000  |
|-1 0 0 1 0 0 0 0   |      |x4|         |30000 |
| 0 0 -1 0 1 0 0 0  |      |x5|         |3000  |
| 0 -1 0 -1 0 1 0 0 |      |x6|         |0     |
| 0 0 0 0 -1 0 1 0  |      |x7|         |3000  |
| 0 0 0 0 0 -1 -1 1 |      |x8|         |500   |
```

Passando isso para o Julia no formato Ax=B, teremos os vetores A e B para formar as nossas matrizes, que terão essa cara:

```
In [40]: 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]
```

Para resolvermos esse sistema, vamos nos utilizar do método sistema_denso(a,b) já utilizado e explicado nos exercícios anteriores.

```
In [41]: sistema_denso(A,B)

Out[41]: 8x1 Array{Float64,2}:
           7000.0
           3500.0
           9000.0
          37000.0
          12000.0
          40500.0
          15000.0
          56000.0
```

Dessa forma, encontramos que:
x1 = 7000
x2 = 3500
x3 = 9000
x4 = 37000
x4 = 12000
x6 = 40500
x7 = 15000
x8 = 56000

(b) Faça a modelagem com o cano pontilhado x9 e tente resolver o sistema linear com o LU. O que aconteceu?

Modelando agora com o cano pontilhado x9, teremos:

A: x1 - 7000
B: x2 - 3500
C: x3 - 9000
D: x4 - x1 - 30000 - x9
E: x5 - x3 - 3000 + x9
F: x6 - x4 - x2
G: x7 - x5 - 3000
H: x8 - x6 - x7 - 500

temos então o sistema:

x1 = 7000
x2 = 3500
x3 = 9000
x4 - x1 - x9 = 30000
x5 - x3 + x9 = 3000
x6 - x4 - x2 = 0
x7 - x5 = 3000
x8 - x6 - x7 = 500

Montando as matrizes do tipo Ax=B, teremos:

```
|1 0 0 0 0 0 0 0  0  |      |x1|          |7000 |
| 0 1 0 0 0 0 0 0 0  |      |x2|          |3500 |
| 0 0 1 0 0 0 0 0 0  |      |x3|     =    |9000 |
|-1 0 0 1 0 0 0 0  -1 |   * |x4|          |30000|
| 0 0 -1 0 1 0 0 0 1 |      |x5|          |3000 |
| 0 -1 0 -1 0 1 0 0 0|      |x6|          |0    |
| 0 0 0 0 -1 0 1 0  0|      |x7|          |3000 |
|0 0 0 0 0 -1 -1 1  0|      |x8|          |500  |
                            |x9|
```

Novamente passando para vetores no Julia, teremos:

```
In [ ]: 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]
```

E agora utilizando o método sistema_denso(a,b):

```
In [46]: sistema_denso(A,B)
Out[46]: 8×1 Array{Float64,2}:
           7000.0
           3500.0
           9000.0
          37000.0
          12000.0
          40500.0
          15000.0
          56000.0
```

O que acontece é que a nossa Matriz A tem 9 colunas enquanto a nossa matriz B tem 8 linhas, dessa forma o julia otimiza para que possamos ainda sim achar os coeficientes sem erro, mas assim não conseguiremos achar o valor para x9. Por isso foi retornado exatamente o mesmo número de coeficientes com os mesmos valores.