


# Laboratório 3 – Sistemas Lineares

- 
- Efrain Marcelo
  - Fernando Lucas
  - Teophilo Vitor

# Decomposição LU

➤ Para a decomposição teremos a solução utilizando o algoritmo a seguir:

➤  $LU \cdot x = b$

➤ 
$$\begin{bmatrix} 1 & 0 & 0 \\ m1 & 1 & 0 \\ m2 & m2 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} b1 \\ b2 \\ b3 \end{bmatrix}$$

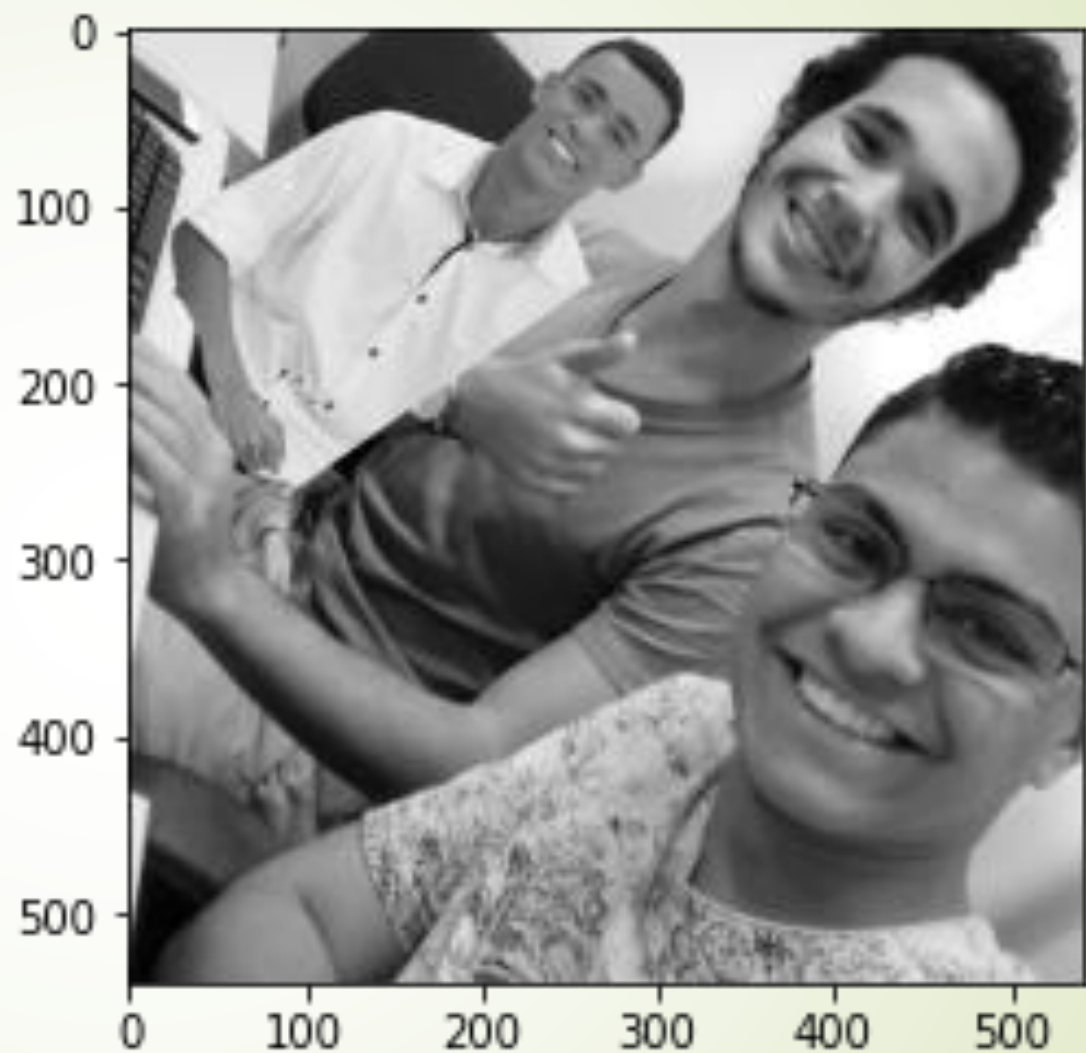
➤ Tendo  $U \cdot x = y$  ficaremos com o sistema  $L \cdot y = b$  e por fim tendo  $y$ :

➤ 
$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} y1 \\ y2 \\ y3 \end{bmatrix}, \text{ tendo os valores de } y \text{ encontramos } X \text{ solução}$$

Imagem  
original



Imagem antes de  
acrescentar o  
ruído



Matrizes  
resultantes L e U

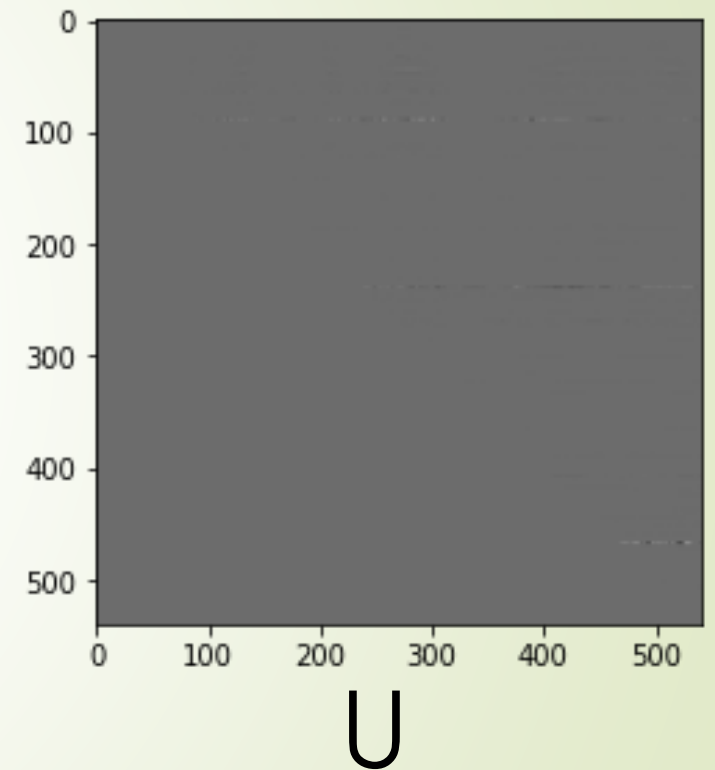
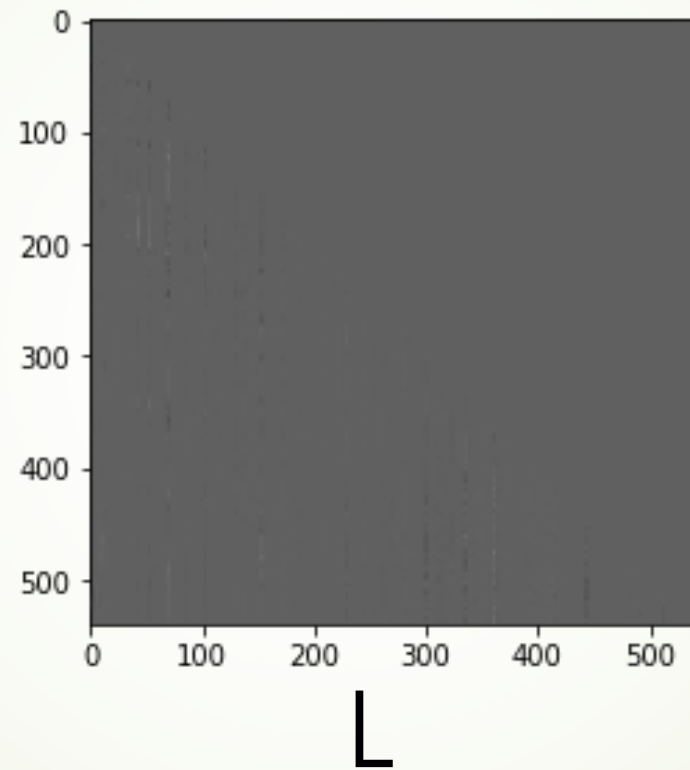
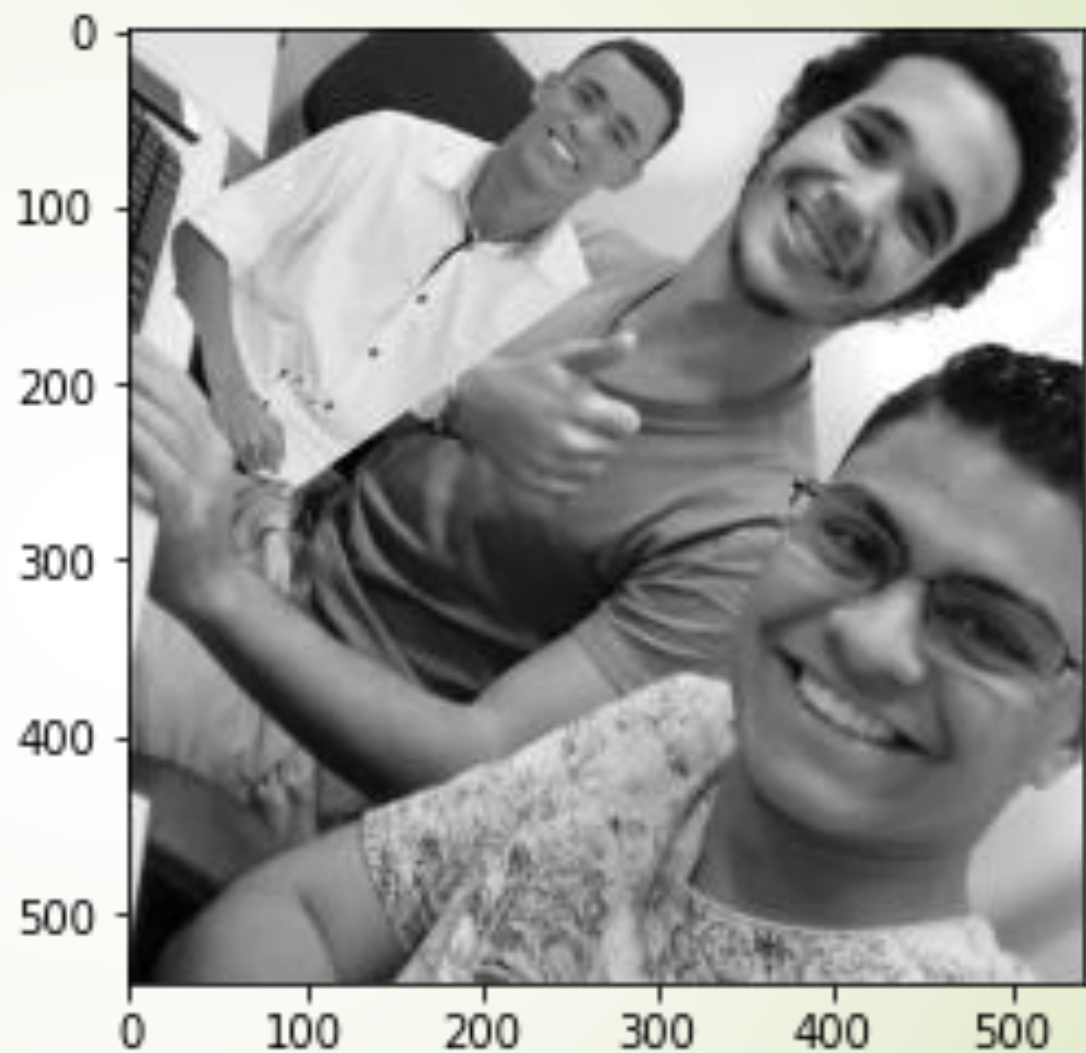




Imagem resultante  
da decomposição  
LU



# Cálculo decomposição LU

- Função rgb responsável por converter a imagem para branco e cinza
- Função LU responsável por criar as matrizes L e U

```
[1] import matplotlib.pyplot as plt
import numpy as np
import math as ma
# Função para tranformar a nossa imagem em preto e Branco
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [1/3, 1/3, 1/3])
#Função que retorna L e U
def LU(B):
    imgcinza = rgb2gray(B)
    h = imgcinza.shape
    t = h[0]
    A = imgcinza + np.random.rand(h[0],h[1])
    G = imgcinza + np.random.rand(h[0],h[1])
    for l in range(0,t) :
        for i in range(l+1, t):
            fator = G[i,l]/G[l,l]
            for j in range(t):
                G[i,j] = G[i,j] - (fator * G[l,j])
    #Criamos nossa Matriz L com base nas Matrizes A e G
    L = np.zeros([t,t])
    for i in range(0,t):
        for j in range(0,t):
            soma = 0
            for k in range(0,j):
                soma += L[i,k]*G[k,j]
            L[i,j] = (A[i,j] - soma)/G[j,j]
    return L,G
```

# Aplicando a função

- ▶ Aplicamos a função e realizamos o produto LU como já explicado para obter a matriz resultante, a nossa imagem

```
#Lendo nossa Imagem
img = plt.imread('FotoGrupo5_Definitiva.jpg')
plt.imshow(img)
plt.show()
#aplicando a função na nossa img e obtendo duas variaveis
(L , U) = LU(img)
imgcinza = rgb2gray(img) + np.random.rand(540,540)
# Fazendo o produto de L x U = imgcinza
F = L.dot(U)
print("Com ruído")
plt.imshow( imgcinza , cmap= "gray" )
plt.show()
print("Matriz L")
plt.imshow( L , cmap= "gray" )
plt.show()
print("Matriz U")
plt.imshow( U , cmap= "gray" )
plt.show()
print("Matriz L x U")
plt.imshow( F , cmap= "gray" )
plt.show()
```



## PARTE 2

JACOBI

$$\begin{cases} x_1^{(k+1)} = \left( b_1 - (a_{12}x_2^{(k)} + \dots + a_{1n}x_n^{(k)}) \right) / a_{11}, \\ x_2^{(k+1)} = \left( b_2 - (a_{21}x_1^{(k)} + \dots + a_{2n}x_n^{(k)}) \right) / a_{22}, \\ \vdots \\ x_n^{(k+1)} = \left( b_n - (a_{n1}x_1^{(k)} + \dots + a_{n,n-1}x_{n-1}^{(k)}) \right) / a_{nn}, \end{cases}$$

GAUSS-SEIDEL

$$\begin{cases} x_1^{(k+1)} = \left( b_1 - (a_{12}x_2^{(k)} + \dots + a_{1n}x_n^{(k)}) \right) / a_{11}, \\ x_2^{(k+1)} = \left( b_2 - (a_{21}x_1^{(k+1)} + \dots + a_{2n}x_n^{(k)}) \right) / a_{22}, \\ \vdots \\ x_n^{(k+1)} = \left( b_n - (a_{n1}x_1^{(k+1)} + \dots + a_{n,n-1}x_{n-1}^{(k+1)}) \right) / a_{nn}, \end{cases}$$

## Montagem da matriz C e b

```
import numpy as np
p = int(input())
n = int(input())
b = np.array([1715, 1941, 1944, 1797, 1829, 1878, 1209, 2061, 1845, 1992,
              1715, 1941, 1944, 1797, 1829, 1878, 1209, 2061, 1845, 1992])
C = np.zeros((12,12))
for i in range(0, 12, 1) :
    for j in range(0, 12, 1) :
        if i==j :
            C[i, j] = 10
        elif abs(i-j) > 1 :
            C[i, j] = 1
        elif abs(i-j) == 1 :
            C[i, j] = 0
C[11, 0] = 0
C[11, 9] = 0
C[0, 2] = 0
C[0, 4] = 0
C[1, 11] = 0
C[3, 4] = 1
C[3, 10] = 0
C[3, 11] = 0
C[10, 11] = 1
```

# Método de Jacobi

$$\begin{cases} M_1^{(k+1)} = (b_1 - (C_{12}M_2^{(k)} + \dots + C_{1n}M_n^{(k)})) / C_{11} \\ M_2^{(k+1)} = (b_2 - (C_{21}M_1^{(k)} + \dots + C_{2n}M_n^{(k)})) / C_{22} \\ \vdots \\ M_n^{(k+1)} = (b_n - (C_{n1}M_1^{(k)} + \dots + C_{n,n-1}M_{n-1}^{(k)})) / C_{nn} \end{cases}$$

```
x = np.zeros([len(b)])
cont = 0
erro = 1
while erro >= 10**(-p) and cont < n :
    x1 = np.array(x)
    for i in range(0, len(b)) :
        soma = 0
        for j in range(0, len(b)) :
            if j != i :
                soma = soma + C[i,j]*x1[j] #Soma dos Xn termos.
        x[i] = (b[i] - soma)/C[i,i] #X(k+1), obtido isolando em cada equação do sistema.
    cont+=1
    erro = max((abs(x-x1)))
r = [chr(int(np.round(c))) for c in x] #Transformação para tabela ASCII
```

Não converge para o critério das linhas.

Converge para o critério das colunas.

['e', 'u', 'r', 'e', 'c', 'a', ' ', 'v', 'i', 'v', 'e', '!']

92

# Método de Gauss-Seidel

$$\begin{cases} M_1^{(k+1)} = (b_1 - (C_{12}M_2^{(k)} + \dots + C_{1n}M_n^{(k)})) / C_{11} \\ M_2^{(k+1)} = (b_2 - (C_{21}M_1^{(k+1)} + \dots + C_{2n}M_n^{(k+1)})) / C_{22} \\ \vdots \\ M_n^{(k+1)} = (b_n - (C_{n1}M_1^{(k+1)} + \dots + C_{n,n-1}M_{n-1}^{(k+1)})) / C_{nn} \end{cases}$$

```
x = np.zeros([len(b)])
cont = 0
erro = 1
while erro >= 10**(-p) and cont < n :
    x1 = np.array(x)
    for i in range(0, len(b)) :
        soma = 0
        for j in range(0, len(b)) :
            if j != i :
                soma = soma + C[i,j]*x[j] #Soma dos Xn termos.
        x[i] = (b[i] - soma)/C[i,i] #X(k+1), obtido isolando em cada equação do sistema.
    cont+=1
    erro = max((abs(x-x1)))
r = [chr(int(np.round(c))) for c in x] #Transformação para tabela ASCII
```

Converge para Sassenfeld.

Não converge para o critério das linhas.

```
['e', 'u', 'r', 'e', 'c', 'a', ' ', 'v', 'i', 'v', 'e', '!']
10
```

# Teste de convergência

```
cc = np.zeros((12,1)) #vetor auxiliar para critério das colunas
cl = np.zeros((12,1)) #vetor auxiliar para critério das linhas
#for para critério das linhas, verifica se a soma de cada elemento da linha é maior que o respectivo C[i, i]
for i in range(0, 12, 1):
    cc[i] = C[i, :].sum() - C[i, i]
if max(cc) < 0 :
    print("Converge para o critério das linhas.")
else :
    print("Não converge para o critério das linhas.")

#for para critério das colunas, critério das linhas falha pois a soma de pelo menos uma das linhas é maior que o C[i,i]
for i in range(0, 12, 1) :
    for j in range(0, 12, 1) :
        C[i, j] = C[i, j]/10
for j in range(0, 12) :
    cc[j] = C[:, j].sum() - C[i, i]
if max(cc) < 1 :
    print("Converge para o critério das colunas.")
else :
    print("Não converge para o critério das colunas.")
print(r)
print(cont)
```

```
betas = np.zeros((12, 1)) #Vetor auxiliar Sassenfeld
for i in range(len(C)):
    beta = 0
    for j in range(len(C)):
        if i!=j and i==0:
            beta = beta + C[i, j] #Primeiro beta
        elif i!=j and i!=0:
            beta = beta + C[i, j]*betas[j] #Beta posteriores
    beta = beta/C[i, i] #Divisão pelo elemento da diagonal principal
    betas[i] = beta #Atribuição do vetor auxiliar ao beta respectivo
if max(betas) < 1: #Verificação da condição de satisfação para Sassenfeld
    print("Converge para Sassenfeld.")
else :
    print("Não converge para Sassenfeld")
```





OBRIGADO  
PELA  
ATENÇÃO