Tarefa 1 - Decomposição LU para Matrizes Tridiagonais - MAP3121

Data de entrega: 01/05/2022

- Rodrigo Gebara Reis - NUSP: 11819880
- Victor Rocha da Silva - NUSP: 11223782

Decomposição LU:

Supondo que A é uma matriz triangularizável pelo Método de Eliminação de Gauss sem trocas de linhas, podemos escrever:

$$A = LU,$$

onde L é uma matriz triangular inferior, e U é uma matriz triangular superior. Então:

$$a_{ij} = \sum_{k=1}^{n} l_{ik} u_{kj}$$

Como L é triangular inferior, $l_{ik} = 0$ se $k > i$; e como U é triangular superior, $u_{kj} = 0$ se $k > j$. Assim, podemos escrever:

$$a_{ij} = \sum_{k=1}^{\min\{i,j\}} l_{ik} u_{kj} \qquad (1)$$

Para aplicar o Método de Eliminação de Gauss no sistema $Ax = b$, supondo-se que A é triangularizável sem troca de linhas, podemos aplicar o seguinte algoritmo:

para $j = 1, \ldots, n$:
  para $i = j + 1, \ldots, n$:
    $l_i = l_i - \frac{a_{ij}}{a_{jj}} \cdot l_j$
fim

onde l_k representa a linha k da matriz expandida $[A|b]$.

Posteriormente, encontra-se o vetor x de baixo para cima, resolvendo as equações.

In [ ]:
```
import numpy as np
import time

# Entradas: a matriz extendida A|b, tal que A array n x n, b array 1 x n (Ax = b)
def gauss(a):
    n = a.shape[0]

    a = np.array(a, dtype = float)
    A = np.copy(a[:,:-1])
    b = np.copy(a[:,n]).reshape((n))

    start = time.time()

    # Eliminação (Escalonamento)
    for i in range(n):
        for j in range(i+1, n):
            a[j,:] = a[j,:] - a[j, i]/a[i, i]*a[i,:]
```

```python
    # Solução
    x = np.zeros(n)
    x[n-1] = a[n-1, n]/a[n-1, n-1]
    for i in range(n-2,-1,-1):
        x[i] = (a[i, n] - np.dot(a[i, i+1:n], x[i+1:n]))/a[i, i]

    end = time.time()

    print("Solucao: ")
    print(x)

    print("Residuo: ")
    print(np.max(np.abs(A@x - b)))

    print("Tempo: ")
    print(end - start)

    return x
```

In [ ]:
```python
# Teste:
a = np.array([[3, 2, -1, 0], [1, 3, 1, 1], [2, 2, -2, 2]], dtype = float) #Matriz expandida [

gauss(a)
```

```
Solucao:
[-1.  1. -1.]
Residuo:
7.771561172376096e-16
Tempo:
0.0
```

Out[ ]:
```
array([-1.,  1., -1.])
```

Pode-se provar, também, que as matrizes L e U, triangulares inferior e superior, tais que $A = LU$, podem ser obtidas da seguinte forma:

- U é a matriz triangular superior obtida pelo Método de Eliminação de Gauss;
- L é a matriz triangular inferior que contém os multiplicadores $m_{ij} = \frac{a_{ij}}{a_{jj}}$, utilizados na Eliminação de Gauss, na posição ij. Além disso, possui todos os elementos da diagonal principal iguais a 1.

Utilizando o fato de que $l_{ii} = 1$, $li, i+1 = 0$, e $u_{i+1,i} = 0$, podemos encontrar L e U de outra maneira. Observe:

- Sabemos que

$$a_{ii} = \sum_{k=1}^{i} l_{ik} u_{ki} \qquad (2)$$

  para $i \in [1, n]$, de acordo com a expressão (1). Assim, $a_{11} = l_{11}u_{11} \Rightarrow u_{11} = a_{11}$

- Se $i < j$,

$$a_{ij} = \sum_{k=1}^{i} l_{ik} u_{kj} \qquad (3)$$

  Se $i = 1$: $a_{1j} = l_{11}u_{1j} \Rightarrow u_{1j} = a_{1j}$, para $j \in [2, n]$

Com isso, já definimos toda a primeira linha de U.

- Se $j < i$,

$$a_{ij} = \sum_{k=1}^{j} l_{ik}u_{kj} \qquad (4)$$

Para $j = 1$: $a_{i1} = l_{i1}u_{11} \Rightarrow l_{i1} = \frac{a_{i1}}{u_{11}} = \frac{a_{i1}}{a_{11}}$, para $i \in [2, n]$

Como sabemos que $l_{11} = 1$ por hipótese, temos definida toda a primeira coluna de L.

- Substituindo $i = 2$ em (2), temos: $a_{22} = \sum_{k=1}^{2} l_{2k}u_{k2} = l_{21}u_{12} + l_{11}u_{22} \Rightarrow u_{22} = a_{22} - l_{21}u_{12}$
- Fazendo $i = 2$ em (3): $a_{2j} = \sum_{k=1}^{2} l_{2k}u_{kj} = l_{21}u_{1j} + l_{22}u_{2j} \Rightarrow u_{2j} = a_{2j} - l_{21}u_{1j}$, $j \in [3, n]$

Sabe-se que $u_{21} = 0$, então encontramos toda a segunda linha de U.

- Agora fazemos $j = 2$ em (4): $a_{i2} = \sum_{k=1}^{2} l_{ik}u_{k2} = l_{i1}u_{12} + l_{i2}u_{22} \Rightarrow l_{i2} = (a_{i2} - l_{i1}u_{12}) \cdot \frac{1}{u_{22}}$, $i \in [3, n]$

Dessa forma, já determinados l_{12}, l_{22}, u_{22}, l_{i1} e u_{12}, temos a segunda coluna de L.

Com tudo isso, podemos chegar ao algoritmo proposto:

- Utilizando (2): $a_{kk} = \sum_{s=1}^{k} l_{ks}u_{sk} \Rightarrow u_{kk} = \frac{1}{l_{kk}}(a_{kk} - \sum_{s=1}^{k-1} l_{ks}u_{sk}) = a_{kk} - \sum_{s=1}^{k-1} l_{ks}u_{sk}$
- De (3): $a_{kj} = \sum_{s=1}^{k} l_{ks}u_{sj} \Rightarrow u_{kj} = \frac{1}{l_{kk}}(a_{kj} - \sum_{s=1}^{k-1} l_{ks}u_{sj}) = a_{kj} - \sum_{s=1}^{k-1} l_{ks}u_{sj}$
- A partir de (4): $a_{ik} = \sum_{s=1}^{k} l_{is}u_{sk} \Rightarrow l_{ik} = \frac{1}{u_{kk}}(a_{ik} - \sum_{s=1}^{k-1} l_{is}u_{sk})$

Reunindo as proposições, finalmente temos:

para $i = 1, \ldots, n$:

$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$, $j = i, \ldots, n$

$l_{ji} = \left(a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki}\right) / u_{ii}$, $j = i+1, \ldots, n$

fim

In [ ]:
```
def decomposicaoLU(A):
    n = A.shape[0]
    L, U = np.zeros((n, n)), np.zeros((n, n))

    for i in range(n):
        U[i, i:n] = A[i, i:n] - L[i, :i]@U[:i, i:n]
        L[i+1:n, i] = (A[i+1:n, i] - L[i+1:n,:i]@U[:i, i])/U[i,i]

    np.fill_diagonal(L,1) # Completa a diagonal de L com 1

    return L, U
```

Testando a funcionalidade do algoritmo acima para uma matriz A genérica:

$$A = \begin{bmatrix} 3 & 2 & 4 \\ 1 & 1 & 2 \\ 4 & 3 & -2 \end{bmatrix} \Rightarrow L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{4}{3} & 1 & 1 \end{bmatrix}; \; U = \begin{bmatrix} 3 & 2 & 4 \\ 0 & \frac{1}{3} & \frac{2}{3} \\ 0 & 0 & -8 \end{bmatrix}$$

In [ ]:
```
A = np.array([[3,2,4],[1,1,2],[4,3,-2]])
L, U = decomposicaoLU(A)

print('\nL = ')
print(L)
```

```
print('\nU = ')
print(U)
```

```
L =
[[1.         0.         0.        ]
 [0.33333333 1.         0.        ]
 [1.33333333 1.         1.        ]]

U =
[[ 3.          2.          4.        ]
 [ 0.          0.33333333  0.66666667]
 [ 0.          0.         -8.        ]]
```

In [ ]:
```
print('A = ')
print(L@U)
```

```
A =
[[ 3.  2.  4.]
 [ 1.  1.  2.]
 [ 4.  3. -2.]]
```

Vemos que a implementação desse algoritmo permite obter de maneira muito mais simples a decomposição LU, sem que seja necessário calcular explicitamente pelo Método da Eliminação de Gauss as matrizes L e U.

Tarefa - Parte 1: decomposição LU de uma matriz tridiagonal $A\,(n \times n)$:

Para fazer um uso mais eficiente de memória, sabendo das características das matrizes L e U, podemos armazená-las em uma única matriz.

Ao realizar o processo da eliminação de Gauss, automaticamente encontramos a matriz U e os termos da matriz L. Curiosamente, para zerar um termo de uma linha da matriz A, é necessário o cálculo de $m_{ij} = \dfrac{a_{ij}}{a_{jj}}$, como apresentado no código da Eliminação de Gauss. Esses multiplicadores são justamente os termos de L. Assim, alteramos a matriz original de tal forma que, à medida que a eliminação de Gauss é realizada, a matriz U é automaticamente gerada, e, nos termos zerados, armazenamos os multiplicadores.

No final da execução, os termos da diagonal principal e da metade triangular superior serão os elementos da matriz U, e o restante (metade triangular inferior), da matriz L. Observe:

In [ ]:
```
def LU_completo(A):

    n = A.shape[0]

    start = time.time()

    LU = np.eye(n) # Inicia matriz identidade de ordem n

    for i in range(n):
        #Varre linhas superiores (Upper)
        LU[i,i:] = A[i,i:]-LU[i,:i] @ LU[:i,i:]
        #Varre colunas inferiores (Lower)
        LU[(i+1):,i] = (A[(i+1):,i]- LU[(i+1):,:i] @ LU[:i,i]) / LU[i,i]

    U = np.triu(LU)
    L = np.tril(LU)
    np.fill_diagonal(L, 1)

    end = time.time()
```

```
    print("Tempo: ")
    print(end - start)

    print("L = \n", L, "\n U = \n", U, "\n LU = \n", LU)
    return LU
```

Para a mesma matriz $A = \begin{bmatrix} 3 & 2 & 4 \\ 1 & 1 & 2 \\ 4 & 3 & -2 \end{bmatrix}$

In [ ]:
```
A = np.array([[3,2,4],[1,1,2],[4,3,-2]])

LU_completo(A)
```

```
Tempo:
0.0
L =
 [[1.         0.         0.        ]
 [0.33333333 1.         0.        ]
 [1.33333333 1.         1.        ]]
 U =
 [[ 3.          2.          4.        ]
 [ 0.          0.33333333  0.66666667]
 [ 0.          0.         -8.        ]]
 LU =
 [[ 3.          2.          4.        ]
 [ 0.33333333  0.33333333  0.66666667]
 [ 1.33333333  1.         -8.        ]]
```

Out[ ]:
```
array([[ 3.        ,  2.        ,  4.        ],
       [ 0.33333333,  0.33333333,  0.66666667],
       [ 1.33333333,  1.        , -8.        ]])
```

Depois de feita a decomposição LU, é muito mais simples e rápido resolver sistemas do tipo $Ax = b$. Como $A = LU$, temos:

$$LUx = b \Leftrightarrow L(Ux) = b$$

Podemos fazer uma mudança de variável, tal que $(Ux) = y$, que resulta em $Ly = b$.

Ou seja, resolver $Ax = b$ equivale a resolver dois sistemas mais simples, um triangular inferior, e outro triangular superior:

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

Matrizes Tridiagonais

Matrizes tridiagonais são aquelas que possuem elementos não nulos apenas na diagonal principal, e em uma acima e outra abaixo da principal, isto é: $\begin{cases} a_{ij} = 0, & \text{se } |i-j| > 1, \\ a_{ij} \neq 0, & \text{caso contrário.} \end{cases}$

$$A = \begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix}$$

Observando o formato da matriz tridiagonal, podemos guardar toda a informação contida nela em apenas três vetores:

- $a = (0, a_2, a_3, \ldots, a_{n-1}, a_n)$
- $b = (b_1, b_2, b_3, \ldots, b_{n-1}, b_n)$
- $c = (c_1, c_2, \ldots, c_{n-1}, 0)$

Podemos implementar um código para representar matrizes tridiagonais inteiras a partir dos vetores a, b e c:

In [ ]:
```
# Função auxiliar para estruturar matrizes tridiagonais (cíclicas ou não)

def createMatrix(a, b, c):
  n = a.shape[0]

  A = np.zeros([n, n])
  A[0, n-1] = a[0]
  A[n-1, 0] = c[n-1]
  A[n-1, n-1] = b[n-1]
  for i in range(0, n-1):
    A[i, i] = b[i]
    A[i+1, i] = a[i+1]
    A[i, i+1] = c[i]

  return A
```

Além disso, devido a esse formato específico, as matrizes L e U, triangulares, tais que $A = LU$, também possuem características especiais.

Na dedução da decomposição LU, encontra-se matrizes M_i tais que $\prod M_i A = U$, com U triangular superior. A multiplicação pelas matrizes M_i tem o efeito de zerar os termos da coluna i abaixo da diagonal principal de A. No entanto, no caso de uma matriz tridiagonal, os únicos elementos alterados, além dos zerados, é a própria diagonal principal: a diagonal imediatamente acima (vetor c) permanece inalterada. Observe, para o caso de $A_{3\times 3}$:

$$\begin{bmatrix} 1 & 0 & 0 \\ -\frac{a_2}{b_1} & 1 & 0 \\ -\frac{0}{a_2} & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 & c_1 & 0 \\ a_2 & b_2 & c_2 \\ 0 & a_3 & b_3 \end{bmatrix} = \begin{bmatrix} b_1 & c_1 & 0 \\ 0 & \tilde{b_2} & c_2 \\ 0 & a_3 & b_3 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 & c_1 & 0 \\ a_2 & b_2 & c_2 \\ 0 & a_3 & b_3 \end{bmatrix} = \begin{bmatrix} b_1 & c_1 & 0 \\ 0 & \tilde{b_2} & c_2 \\ 0 & a_3 & b_3 \end{bmatrix}$$

Similarmente, para zerar a_3, faríamos:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -m_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 & c_1 & 0 \\ a_2 & b_2 & c_2 \\ 0 & a_3 & b_3 \end{bmatrix} = \begin{bmatrix} b_1 & c_1 & 0 \\ 0 & \tilde{b_2} & c_2 \\ 0 & 0 & \tilde{b_3} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ 0 & -m_{32} & 1 \end{bmatrix} \begin{bmatrix} b_1 & c_1 & 0 \\ a_2 & b_2 & c_2 \\ 0 & a_3 & b_3 \end{bmatrix} = \begin{bmatrix} b_1 & c_1 & 0 \\ 0 & \tilde{b_2} & c_2 \\ 0 & 0 & \tilde{b_3} \end{bmatrix}$$

Note, também, que podemos escrever $A = \prod M_i^{-1} U$. Essas matrizes M_i, no entanto, têm uma propriedade interessante, decorrente do simples fato de que $M_i M_i^{-1} = I_n$: a inversa é idêntica à matriz original, trocando-se o sinal do termo $m_{k,k-1}$. Assim, teríamos, em um caso geral:

Assim, de modo geral, teríamos:

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & & & \\ m_{21} & 1 & & & \\ & \ddots & \ddots & & \\ & & m_{n-1,n-2} & 1 & \\ & & & m_{n,n-1} & 1 \end{bmatrix} \begin{bmatrix} b_1 & c_1 & & & \\ & \tilde{b_2} & c_2 & & \\ & & \ddots & \ddots & \\ & & & \tilde{b_{n-1}} & c_{n-} \\ & & & & \tilde{b_n} \end{bmatrix}$$

Tendo A decomposta em duas matrizes, uma triangular inferior, e outra triangular superior, podemos escrever:

$$L = \begin{bmatrix} 1 & 0 & & & \\ m_{21} & 1 & & & \\ & \ddots & \ddots & & \\ & & m_{n-1,n-2} & 1 & \\ & & & m_{n,n-1} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & & & \\ l_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & l_{n-1} & 1 & \\ & & & l_n & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} b_1 & c_1 & & & \\ & \tilde{b_2} & c_2 & & \\ & & \ddots & \ddots & \\ & & & \tilde{b_{n-1}} & c_{n-1} \\ & & & & \tilde{b_n} \end{bmatrix} = \begin{bmatrix} u_1 & c_1 & & & \\ & u_2 & c_2 & & \\ & & \ddots & \ddots & \\ & & & u_{n-1} & c_{n-1} \\ & & & & u_n \end{bmatrix}$$

Finalmente, observamos as seguintes propriedades:

- Únicos elementos de U que podem ser não nulos são U_{ii} e $U_{i,i+1}$
- Únicos multiplicadores que podem ser não nulos são $L_{i+1,i}$
- $U_{i,i+1} = c_i$, $u_i = U_{ii}$ e $l_{i+1} = L_{i+1,i}$.

Com essas propriedades, o custo computacional fica extremamente reduzido, uma vez que podemos evitar realizar contas cujos resultados sabemos que serão iguais a zero.

Sabendo que $A = LU$, realizando as devidas multiplicações, encontramos:

- $b_1 = u_1 \Rightarrow u_1 = b_1$
- $a_2 = l_2 \cdot u_1$, $a_3 = 0 \cdot c_1 + l_3 \cdot u_2 \Rightarrow a_k = l_k \cdot u_{k-1} \Rightarrow l_k = a_k / u_{k-1}$
- $b_2 = l_2 \cdot c_1 + u_2$, $b_3 = l_3 \cdot c_2 + u_3 \Rightarrow b_k = l_k \cdot c_{k-1} + u_k \Rightarrow u_k = b_k - l_k c_{k-1}$

Assim, temos a base do algoritmo para encontrar as matrizes L e U com um custo computacional reduzido:

$u_1 = b_1$
para $i = 2, \dots, n$:
    $l_i = a_i / u_{i-1}$

$u_i = b_i - l_i \cdot c_{i-1}$

fim

Assim, a matriz U está armazenada em dois vetores: u e c (já que $U_{i,i+1} = c_i$), e L está armazenada em l.

In [ ]:
```
import numpy as np
import time

# Decompõe uma matriz A tridiagonal em uma matriz triangular inferior (L) e uma triangular su
# Como A é tridiagonal, é caracterizada por três vetores (a, b, c), são fornecidos na entrada
# A função retorna os arrays numpy (vetores) l e u que caracterizam as matrizes L e U
def decompLU(a, b, c):
  n = a.shape[0]
  l, u = np.zeros(n), np.zeros(n) # Inicia os vetores l e u

  l[0] = 0 # Assim como a[0] = 0, definimos l[0] = 0
  u[0] = b[0]

  # Segue as formulações definidas no texto acima
  for i in range(1, n):
    l[i] = a[i]/u[i-1]
    u[i] = b[i] - l[i]*c[i-1]

  return(l, u)
```

Vamos testar o código com um exemplo: tomamos a matriz

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

ou seja, $a = (0, 3, 2, 1)$, $b = (1, 4, 3, 3)$, e $c = (4, 1, 4, 0)$.

In [ ]:
```
a = np.array([0, 3, 2, 1])
b = np.array([1, 4, 3, 3])
c = np.array([4, 1, 4, 0])

l, u = decompLU(a, b, c)

L = createMatrix(l, np.ones(len(l)), np.zeros(len(l)))
U = createMatrix(np.zeros(len(u)), u, c)

print("L = \n", L, "\nU = \n", U, "\nLU = \n", L@U)
```

```
L =
 [[ 1.          0.          0.          0.        ]
 [ 3.          1.          0.          0.        ]
 [ 0.         -0.25        1.          0.        ]
 [ 0.          0.          0.30769231  1.        ]]
U =
 [[ 1.          4.          0.          0.        ]
 [ 0.         -8.          1.          0.        ]
 [ 0.          0.          3.25        4.        ]
 [ 0.          0.          0.          1.76923077]]
LU =
 [[1. 4. 0. 0.]
 [3. 4. 1. 0.]
 [0. 2. 3. 4.]
 [0. 0. 1. 3.]]
```

A partir disso, conseguimos resolver sistemas $Ax = d$ com matrizes A tridiagonais. Vamos usar a abordagem de resolução de dois sistemas triangulares, como discutido acima.

Aqui, podemos fazer uso das propriedades de L e U quando A é tridiagonal. Para o primeiro sistema, $Ly = d$, temos:

$$\begin{bmatrix} 1 & 0 & & & \\ l_2 & 1 & 0 & & \\ 0 & \ddots & \ddots & \ddots & \\ & 0 & l_{n-1} & 1 & 0 \\ & \ddots & 0 & l_n & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}$$

Ou seja, $y_1 = d_1$, $l_2 \cdot y_1 + y_2 = d_2 \Leftrightarrow y_2 = d_2 - l_2 \cdot y_1$ ...

Sem perda de generalidade, para $2 \leq i \leq n$, $y_i = d_i - l_i \cdot y_{i-1}$.

Já para o sistema $Ux = y$, temos:

$$\begin{bmatrix} u_1 & c_1 & & & \\ 0 & u_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & 0 & u_{n-1} & c_{n-1} \\ & & & 0 & u_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}$$

Isto é, de baixo para cima, $u_n \cdot x_n = y_n \Leftrightarrow x_n = y_n / u_n$ e $u_{n-1} \cdot x_{n-1} + c_{n-1} \cdot x_n = y_{n-1} \Leftrightarrow x_{n-1} = \frac{y_{n-1} - c_{n-1} \cdot x_n}{u_{n-1}}$

Assim, para $n - 1 \leq i \leq 1$, $x_i = \frac{y_i - c_i \cdot x_{i+1}}{u_i}$

In [ ]:

```
# São aplicadas as formulações apresentadas no texto para resolver um sistema do tipo Ax = d,
# A sendo uma matriz tridiagonal.
# Na entrada são fornecidos os três vetores (a, b, c) que definem a matriz A, além do vetor d
# A função devolve o vetor solução x

def resolveSistemaTridiagonal(a, b, c, d):
  n = a.shape[0]
  # Decompõe a matriz tridiagonal em uma triangular inferior, caracterizada pelo vetor l,
  # e uma triangular superior, caracterizada pelos vetores u e c
  l, u = decompLU(a, b, c)

  #Ly = d
  y = np.zeros(n)

  y[0] = d[0]
  for i in range(1, n):
    y[i] = d[i] - l[i]*y[i-1]

  #Ux = y
  x = np.zeros(n)

  x[n-1] = y[n-1]/u[n-1]
  for i in range(n-2, -1, -1):
    x[i] = (y[i]-c[i]*x[i+1])/u[i]

  return x
```

Podemos realizar um teste com o seguinte sistema:

$$\begin{bmatrix} 10 & 2 & 0 & 0 \\ 3 & 10 & 4 & 0 \\ 0 & 1 & 7 & 5 \\ 0 & 0 & 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$$

Vamos imprimir a **solução** do sistema linear tridiagonal $Ax = d$, o **resíduo** $(d - A\bar{x})$ para a solução aproximada $\bar{x}$ encontrada e o **tempo** de execução para a solução:

In [ ]:
```
a = np.array([0, 3, 1, 3])
b = np.array([10, 10, 7, 4])
c = np.array([2, 4, 5, 0])
d = np.array([3, 4, 5, 6])

start = time.time()
x = resolveSistemaTridiagonal(a, b, c, d)
end = time.time()

print("\nSolucao: ")
print(x)

A = createMatrix(a, b, c)

print("\nResiduo: ")
print(np.max(np.abs(A@x-d)))

print("\nTempo: ")
print(end - start)
```

```
Solucao:
[ 0.14877589  0.75612053 -1.00188324  2.25141243]

Residuo:
1.7763568394002505e-15

Tempo:
0.0
```

Sistemas Tridiagonais Cíclicos

Uma matriz triangular cíclica tem um formato muito semelhante às tridiagonais comuns, à exceção de dois elementos: o último da primeira linha; e o último da primeira coluna:

$$A = \begin{bmatrix} b_1 & c_1 & & & a_1 \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ c_n & & & a_n & b_n \end{bmatrix}$$

Note que a matriz $n-1 \times n-1$ formada pelas linhas e colunas de 1 a $n-1$ é tridiagonal. Vamos denotá-la por T. Além disso, definimos $v = (a_1, 0, \ldots, 0, c_{n-1})^t$ e $w = (c_n, 0, \ldots, 0, a_n)^t$.

Sabe-se que $x = (x_1, \ldots, x_n)^t$, mas será útil dividir esse vetor em $\tilde{x} = (x_1, \ldots, x_{n-1})^t$ e x_n. Faremos o mesmo para o vetor d.

Com isso, o sistema $Ax = d$ pode ser escrito como:

$$\begin{cases} T\tilde{x} + x_n v = \tilde{d} \\ w^t\tilde{x} + x_n b_n = d_n \end{cases}$$

que tem solução:

$$\begin{cases} x_n = \frac{d_n - c_n\tilde{y}_1 - a_n\tilde{y}_{n-1}}{b_n - c_n\tilde{z}_1 - a_n\tilde{z}_{n-1}} \\ \tilde{x} = \tilde{y} - x_n\tilde{z} \end{cases}$$

em que $\tilde{y}$ é solução de $T\tilde{y} = \tilde{d}$, e $\tilde{z}$ é solução de $T\tilde{z} = v$. É de extrema importância compreender a dimensão de v. Note que ele deve ter a mesma dimensão de $\tilde{z}$, que, por sua vez, é $n - 1 \times 1$, já que está sendo multiplicado pela matriz T, que é $n - 1 \times n - 1$. Logo, o vetor v deve ter dimensão $n - 1 \times 1$.

Tarefa - Parte 2: algoritmo para a resolução de um sistema linear tridiagonal cíclico:

(usando a resolução de sistemas tridiagonais)

In [ ]:
```
# Aplica as formulações apresentadas no texto para resolver um sistema tridiagonal cíclico Ax
# Como A é uma matriz tridiagonal, são fornecidos na entrada os três vetores que a definem (a
# A função devolve o vetor solução x
def resolveSistemaCiclico(a, b, c, d):
  n = a.shape[0]

  # Montar matriz T
  at = a[0:n-1].copy()
  at[0] = 0
  bt = b[0:n-1].copy()
  ct = c[0:n-1].copy()
  ct[-1] = 0

  # Montar vetor d~
  dt = d[0:n-1].copy()

  # Montar vetor v
  v = np.zeros(n-1)
  v[0] = a[0]
  v[n-2] = c[n-2]

  # Montar vetor w
  w = np.zeros(n)
  w[0] = c[n-1]
  w[-1] = a[n-1]

  # Solução sistema Ty = d~
  y = resolveSistemaTridiagonal(at, bt, ct, dt)

  # Solução sistema Tz = v
  z = resolveSistemaTridiagonal(at, bt, ct, v)

  x_n = (d[n-1]-c[n-1]*y[0]-a[n-1]*y[n-2])/(b[n-1]-c[n-1]*z[0]-a[n-1]*z[n-2])

  x = y - x_n*z

  x = np.append(x, x_n)

  return x
```

Teste:

É válido implementar uma função de teste para o algoritmo de resolução do sistema linear tridiagonal cíclico, sendo os elementos dos vetores a, b, c e d fornecidos pela própria tarefa:

- $a_i = \dfrac{2i-1}{4i}$, $1 \leq i \leq n-1$, $a_n = \dfrac{2n-1}{2n}$;
- $c_i = 1 - a_i$, $1 \leq i \leq n$;
- $b_i = 2$, $1 \leq i \leq n$
- $d_i = \cos(\dfrac{2\pi i^2}{n^2})\; 1 \leq i \leq n$

In [ ]:
```
# Cria os vetores sugeridos para o teste que definem a matriz tridiagonal cíclica a ser resol
# n = tamanho da matriz utilizada para teste
def criaMatrizTeste(n):
    a = np.zeros(n)
    b = np.zeros(n)
    c = np.zeros(n)
    d = np.zeros(n)

    for i in range(1, n):
        a[i-1] = (2*i-1)/(4*i)
        c[i-1] = 1-a[i-1]
        b[i-1] = 2
        d[i-1] = np.cos((2*(np.pi)*(i**2))/(n**2))

    a[n-1] = (2*n-1)/(2*n)
    c[n-1] = 1-a[n-1]
    b[n-1] = 2
    d[n-1] = 1

    return a,b,c,d
```

Vamos imprimir a **solução** do sistema linear tridiagonal cíclico $Ax = d$, o **resíduo** $(A\bar{x} - d)$ para a solução aproximada $\bar{x}$ encontrada e o **tempo** de execução para a solução:

In [ ]:
```
# n = tamanho da matriz utilizada para teste
def teste(n):
  a, b, c, d = criaMatrizTeste(n)

  start = time.time()

  x = resolveSistemaCiclico(a, b, c, d)

  end = time.time()

  print('n =',n)

  print("\nSolução: ")
  print(x)

  A = createMatrix(a, b, c)
  print("\nResíduo: ")
  print(np.max(np.abs(A@x - d)))

  print("\nTempo: ")
  print(end-start)

  return
```

Teste usando $n = 20$ (sugestão da tarefa):

In [ ]:
```
teste(20)
```

```
n = 20

Solução:
[ 0.33031512  0.33369784  0.33082061  0.32458573  0.3105381   0.28498139
  0.24375728  0.18349137  0.10274415  0.00360629 -0.10669724 -0.2147279
 -0.30113746 -0.34330813 -0.32097501 -0.22451082 -0.0638644   0.12580676
  0.28713644  0.35589205]

Resíduo:
2.220446049250313e-16

Tempo:
0.0
```

É possível perceber um resíduo bem pequeno, o que é um forte indicador de que o código realmente está funcionando.

Além disso, o tempo de execução está pequeno, mostrando que o código está relativamente otimizado.

Teste usando $n = 10000$:

(apenas para mostrar que o tempo de execução está ok e que o resíduo continua pequeno mesmo para matrizes grandes)

In [ ]:
```
teste(10000)
```

```
n = 10000

Solução:
[0.33333332 0.33333334 0.33333333 ... 0.33333237 0.33333316 0.33333342]

Resíduo:
2.220446049250313e-16

Tempo:
0.030917167663574 22
```

Referências

- Equipe de MAP3121. Decomposição LU para Matrizes Tridiagonais.
- Peixoto, Pedro Silva. Anotações de aula de MAP3121.
- GAUSSIAN Elimination, LU-Factorization, and Cholesky Factorization. Disponível em: https://www.cis.upenn.edu/~cis515/cis515-11-sl3.pdf. Acesso em 19/04/2022.
- LU Factorizations. Disponível em: https://math.okstate.edu/people/binegar/4513-F98/ 4513-l10.pdf. Acesso em 20/04/2022.