ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

8° LISTA DE ALGORITMO

1. Implemente os algoritmos de resolução de sistemas lineares por substituição direta e reversa;

$LUx = Pb$.

Agora podemos resolver essa equação resolvendo dois sistemas lineares triangulares. Definimos $y = Ux$, onde x é o vetor solução desejado. Primeiro, resolvemos o sistema triangular inferior

$$\boxed{Ly = Pb} \tag{28.5}$$

para o vetor incógnita y por um método denominado "substituição direta". Depois de resolvido para y, resolvemos o sistema triangular superior

$$\boxed{Ux = y} \tag{28.6}$$

para a incógnita x por um método denominado "substituição reversa". Como a matriz de permutação P é inversível (Exercício D.2-3), multiplicamos ambos os lados da equação (28.4) por P^{-1} dá $P^{-1}PA = P^{-1}LU$, de modo que

$$A = P^{-1}LU. \tag{28.7}$$

$$\begin{array}{lcl}
y_1 & = & b_{\pi[1]}, \\
l_{21}y_1 + y_2 & = & b_{\pi[2]}, \\
l_{31}y_1 + l_{32}y_2 + y_3 & = & b_{\pi[3]}, \\
 & \vdots & \\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n & = & b_{\pi[n]}.
\end{array}$$

A primeira equação nos diz que $y_1 = b_{p[1]}$. Conhecendo o valor de y_1 podemos substituí-lo na segunda equação, o que dá

$$y_2 = b_{\pi[2]} - l_{21}y_1 .$$

Agora, podemos substituir y_1 e y_2 na terceira equação, obtendo

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) .$$

Em geral, substituímos $y_1, y_2, ..., y_{i-1}$ "diretamente" na i-ésima equação para resolver para y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j .$$

Agora, que resolvemos para y, resolvemos para x na equação (28.6) usando ***substituição reversa***, que é semelhante à substituição direta. Aqui, resolvemos primeiro a n-ésima equação e trabalhamos em sentido contrário até a primeira equação. Como a substituição direta, esse processo é executado no tempo (n_2). Visto que U é triangular superior, podemos reescrever o sistema (28.6) como

$$
\begin{array}{rcl}
u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n & = & y_1 , \\
u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n & = & y_2 , \\
 & \vdots & \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n & = & y_{n-2} , \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n & = & y_{n-1} , \\
u_{n,n}x_n & = & y_n .
\end{array}
$$

Assim, podemos resolver sucessivamente para x_n, x_n-1, ..., x_1, da seguinte maneira:

$$
\begin{array}{rcl}
x_n & = & y_n/u_{n,n} , \\
x_{n-1} & = & (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} , \\
x_{n-2} & = & (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} , \\
\vdots & &
\end{array}
$$

ou, em geral,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii} .$$

Dados P, L, U e b, o procedimento LUP-Solve resolve para x combinando substituição direta e substituição reversa. O pseudocódigo considera que a dimensão n aparece no atributo $L.linhas$ e que a matriz de permutação P é representada pelo arranjo p.

LUP-Solve(L, U, π, b)

$n = L.linhas$

sejam x e y novos vetores de comprimento n

for $i = 1$ **to** n

$\quad y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$

for $i = n$ **downto** 1

$\quad x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}$

return x

Exemplo do livro implementado em python.

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0{,}2 & 1 & 0 \\ 0{,}6 & 0{,}5 & 1 \end{pmatrix} ,$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0{,}8 & -0{,}6 \\ 0 & 0 & 2{,}5 \end{pmatrix} ,$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} .$$

$$y = \begin{pmatrix} 8 \\ 1{,}4 \\ 1{,}5 \end{pmatrix}$$

$$x = \begin{pmatrix} -1{,}4 \\ 2{,}2 \\ 0{,}6 \end{pmatrix}$$

```python
import numpy as np

def substituicao_direta(L, P, B):
    n = len(L)
    y = np.zeros(n)                     # criando a matriz y
    b = np.dot(P, B)                    # fazendo o produto da matriz permutação e a matriz B
    for i in range(n):
        y[i] = b[i]                     # implementando o somatório que realiza a substituição direta
        for j in range(i):
            y[i] -= L[i][j] * y[j]
    return y

def substituicao_reversa(U, y):
    n = len(U)
    x = np.zeros(n)                    # criando a matriz solução x
    for i in range(n - 1, -1, -1):     # Começa da última linha e retrocede
        x[i] = y[i]                    # implementando o somatório que realiza a substitução reversa
        for j in range(i + 1, n):
            x[i] -= U[i][j] * x[j]
        x[i] /= U[i][i]
    return x
```

```python
if __name__ == "__main__":

    L = np.array([[1, 0, 0],
                  [0.2, 1, 0],
                  [0.6, 0.5, 1]])  # Matriz triangular inferior unitária
    P = np.array([[0, 0, 1],
                  [1, 0, 0],
                  [0, 1, 0]])  # Matriz de permutação (índices)
    B = np.array([3, 7, 8])  # Vetor B

    U = np.array([[5, 6, 3],
                  [0, 0.8, -0.6],
                  [0, 0, 2.5]])  # Matriz triangular superior U

    y = substituicao_direta(L, P, B)
    x = substituicao_reversa(U, y)
    print("Valor de Y:\t Y =", y)
    print("Solução do sistema linear:\t X =",x)
```

```
/home/alessandro/PycharmProjects/pythonProject/geraMatriz/venv/bin/python /home/alessandro/PycharmProjects/pythonProject/lista_8/sub_dir_rev.py
Valor de Y:  Y = [8.  1.4 1.5]
Solução do sistema linear:   X = [-1.4  2.2  0.6]


Process finished with exit code 0
```

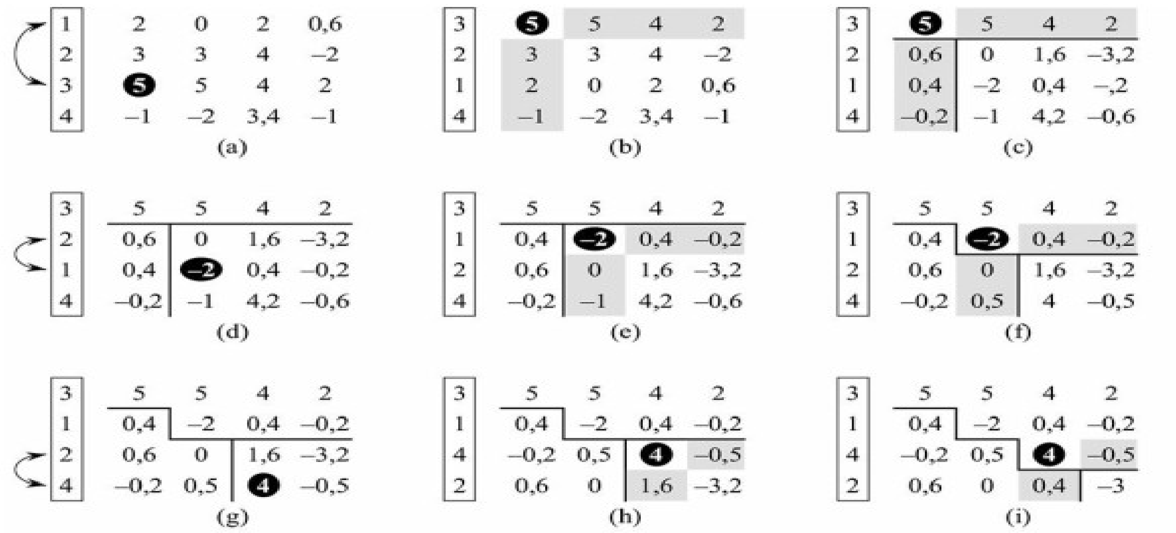
2. Implemente o algoritmo de decomposição LUP

Calculando uma decomposição LUP

Em geral, quando resolvemos um sistema de equações lineares $Ax = b$, temos de pivotar em elementos de A que estão fora da diagonal para evitar divisão por zero. Claro que dividir por zero seria desastroso. Porém, também queremos evitar dividir por um valor pequeno mesmo que A seja não singular — porque isso pode produzir instabilidades numéricas. Então, tentamos pivotar em um valor grande.

A matemática por trás da decomposição LUP é semelhante à da decomposição LU. Lembre-se de que temos uma matriz $n \times n$ não singular A e desejamos encontrar uma matriz de permutação P, uma matriz triangular inferior unitária L e uma matriz triangular superior U, tais que $PA = LU$. Antes de particionar a matriz A, como fizemos para a decomposição LU, passamos um elemento não nulo; digamos a_{k1}, de algum lugar na primeira coluna até a posição (1, 1) da matriz. Para estabilidade numérica, escolhemos a_{k1} como o elemento na primeira coluna que tem o maior valor absoluto. (A primeira coluna não pode conter somente zeros porque A seria singular, já que seu determinante seria zero pelos Teoremas D.4 e D.5.) Para preservar o conjunto de equações, trocamos a linha 1 com a linha k, o que equivale a multiplicar A por uma matriz de permutação Q à esquerda (Exercício D.14). Assim, podemos escrever QA como

$$QA = \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix}.$$



$$\underbrace{\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{P} \underbrace{\begin{pmatrix} 2 & 0 & 2 & 0,6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -1 & 3,4 & -1 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0,4 & 1 & 0 & 0 \\ -0,2 & 0,5 & 1 & 0 \\ 0,6 & 0 & 0,4 & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0,4 & -0,2 \\ 0 & 0 & 4 & -0,5 \\ 0 & 0 & 0 & -3 \end{pmatrix}}_{U}$$

```
LUP-COMPOSITION(A)
    n = A.linhas
    seja π[1 .. n] um novo arranjo
    for i = 1 to n
        π[i] = i
    for k = 1 to n
        p = 0
        for i = k to n
            if |a_ik| > p
                p = |a_ik|
                k' = i
        if p == 0
            error "matriz singular"
        trocar π[k] por π[k']
        for i = 1 to n
            trocar a_ki por a_k'i
        for i = k + 1 to n
            a_ik = a_ik/a_kk
            for j = k + 1 to n
                a_ij = a_ij − a_ik a_kj
```

Implementando o exemplo do Livro em liguaguem Python

```python
import numpy as np

def lup_decomposition(A):
    n = len(A)
    U = np.zeros((n,n))
    a = np.copy(A)
    P = np.identity(n)  # Matriz de permutação inicialmente é uma matriz de identidade
    L = np.identity(n)  # Matriz L inicializada como um matriz identidade
    for k in range(n):
        linha_pivot = np.argmax(np.abs(a[k:, k])) + k  # Encontre o índice da linha de maior modulo (pivô)
        if linha_pivot != k:
            P[[k, linha_pivot]] = P[[linha_pivot, k]]  # Troca de linhas na matriz de permutação
            a[[k, linha_pivot]] = a[[linha_pivot, k]]  # Troca de linhas na matriz A
        for i in range(k + 1, n):
            fator = a[i, k] / a[k, k]  # monta o vetor v dividindo os elementos da coluna pelo elemento pivô
            a[i,k] = fator
            for j in range(k + 1, n):
                a[i, j] -= fator * a[k, j]
    for i in range(0,n):                    # laço para formar a matriz L e U de forma separada
        for j in range(0,n):
            if (i>j):                       # valores abaixo da diagonal principal
                L[i,j] = a[i,j]
            if (j>=i):                      # incluem a diagonal principal e valores acima dela
                U[i,j] = a[i,j]
    return P, U, L, A  # Retornar a matriz de permutação e a matriz U (após a decomposição)
```

```python
if __name__ == "__main__":
    #A = np.array([[3.0, -4.0, 1.0],
    #               [1.0, 2.0, 2.0],
    #              [4.0, 0.0, -3.0]])

    A = np.array([[2.0, 0.0, 2.0, 0.6],
                  [3.0, 3.0, 4.0, -2.0],
                  [5.0, 5.0, 4.0, 2.0],
                  [-1.0, -2.0, 3.4, -1.0]])

    P, U, L, A = lup_decomposition(A)

    print("Matriz P (Matriz de Permutação):")
    print(P)
    print("Matriz U (Matriz Triangular Superior):")
    print(U)
    print("Matriz L (Matriz Triangular Inferior):")
    print(L)
    print("Matriz A (Matriz Original):")
    print(A)
    print("Matriz LU (Matriz Original):")
    print(np.dot(L, U))
    print("Matriz PA (Matriz Original):")
    print(np.dot(P, A))
```

```
Matriz P (Matriz de Permutação):
[[0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]]
Matriz U (Matriz Triangular Superior):
[[ 5.   5.   4.   2. ]
 [ 0.  -2.   0.4 -0.2]
 [ 0.   0.   4.  -0.5]
 [ 0.   0.   0.  -3. ]]
Matriz L (Matriz Triangular Inferior):
[[ 1.   0.   0.   0. ]
 [ 0.4  1.   0.   0. ]
 [-0.2  0.5  1.   0. ]
 [ 0.6 -0.   0.4  1. ]]
Matriz A (Matriz Original):
[[ 2.   0.   2.   0.6]
 [ 3.   3.   4.  -2. ]
 [ 5.   5.   4.   2. ]
 [-1.  -2.   3.4 -1. ]]
```

3. Use os algoritmos que implementou nessa lista e crie uma rotina para calcular a inversa de uma matriz nxn não-singular qualquer, verificando o resultado com o auxílio algum algoritmo de multiplicação de matrizes que já implementou.

Normalmente, não utilizamos inversas de matrizes para resolver sistemas de equações lineares na prática; em vez disso, preferimos usar técnicas numericamente mais estáveis como a decomposição LUP. Porém, às vezes, precisamos calcular a inversa de uma matriz.

Calculando a inversa de uma matriz a partir de uma decomposição LUP

Suponha que temos uma decomposição LUP de uma matriz A na forma de três matrizes L, U e P tais que $PA = LU$. Usando LUP-SOLVE, podemos resolver uma equação da forma $Ax = b$ no tempo (n_2). Visto que a decomposição LUP depende de A, mas não de b, podemos executar LUP-SOLVE para um segundo conjunto de equações da forma $Ax = b'$ no tempo adicional (n_2). Em geral, tão logo tenhamos a decomposição LUP de A, podemos resolver, no tempo (kn_2), k versões da equação $Ax = b$ cuja única diferença é o termo b.

Podemos considerar a equação

$$AX = I_n, \tag{28.10}$$

que define a matriz X, a inversa de A, como um conjunto de n equações distintas da forma $Ax = b$. Para sermos precisos, seja X_i a i-ésima coluna de X, e lembre-se de que o vetor unitário e_i é a i-ésima coluna de I_n. Então, podemos resolver a equação (28.10) para X, usando a decomposição LUP de A para resolver cada equação

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix} \text{ e } A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix}. \tag{28.11}$$

separadamente para X_i . Tão logo tenhamos a decomposição LUP, podemos calcular cada uma das n colunas X_i no tempo $(n\ _2)$ e, portanto, podemos calcular X pela decomposição LUP de A no tempo (n_3). Visto que podemos determinar a decomposição LUP de A no tempo (n_3), podemos calcular a inversa A-1 de uma matriz A no tempo (n_3).

Nessa questão é solicitado que, além da construção do algoritmo que para o cálculo da matriz inversa, se desenvolva uma rotina para verificar a legitimidade da matriz gerada no código. Dessa forma adotei a ideia dado pelo professor, que foi de multiplicar a matriz resultante com a matriz de entrada do problema (matriz A), afim de obter uma matriz identidade, conforme teoria.

Esse foi o primeiro passo, um segundo passo criar uma matriz identidade, e substrair pela matriz resultante do produto mencionado anteriormente, matriz identidade gerado pelo algoritmo, assim teremos como resultado uma matriz nula. O que na prática não aconteceu, devido ainstabilidade numérica como veremos na implementação. No entanto, isso não quer dizer que o código esteja errado, os valores de ponto flutuante são infimos, muito próximos de zero.

```python
import numpy as np

def decomposicao_LUP(A):
    n = len(A)
    U = np.zeros((n,n))
    a = np.copy(A)
    P = np.identity(n)  # Matriz de permutação inicialmente é uma matriz de identidade
    L = np.identity(n)  # Matriz L inicializada como um matriz identidade
    for k in range(n):
        linha_pivot = np.argmax(np.abs(a[k:, k])) + k  # Encontre o índice da linha de maior modulo (pivô)
        if linha_pivot != k:
            P[[k, linha_pivot]] = P[[linha_pivot, k]]  # Troca de linhas na matriz de permutação
            a[[k, linha_pivot]] = a[[linha_pivot, k]]  # Troca de linhas na matriz A
        for i in range(k + 1, n):
            fator = a[i, k] / a[k, k]  # monta o vetor v dividindo os elementos da coluna pelo elemento pivô
            a[i,k] = fator
            for j in range(k + 1, n):
                a[i, j] -= fator * a[k, j]
    for i in range(0,n):                       # laço para formar a matriz L e U de forma separada
        for j in range(0,n):
            if (i>j):                          # valores abaixo da diagonal principal
                L[i,j] = a[i,j]
            if (j>=i):                         # incluem a diagonal principal e valores acima dela
                U[i,j] = a[i,j]
    return L, U, P                             # Retornar a matriz de permutação e a matriz U (após a decomposição)

def substituicao_direta(L, P, B):
    n = len(L)
    y = np.zeros(n)                    # criando a matriz y
    b = np.dot(P, B)                   # fazendo o produto da matriz permutação e a matriz B
    for i in range(n):
        y[i] = b[i]                    # implementando o somatório que realiza a substituição direta
        for j in range(i):
            y[i] -= L[i][j] * y[j]
    return y

def substituicao_reversa(U, y):
    n = len(U)
    x = np.zeros(n)                   # criando a matriz solução x
    for i in range(n - 1, -1, -1):    # Começa da última linha e retrocede
        x[i] = y[i]                   # implementando o somatório que realiza a substitução reversa
        for j in range(i + 1, n):
            x[i] -= U[i][j] * x[j]
        x[i] /= U[i][i]               # CONFERIR ESSA LINHA
    return x

def inverse_matrix(A):
    n = len(A)
    L, U, P = decomposicao_LUP(A) # Decomposição LUP
    A_inv = np.zeros((n, n))

    for i in range(n):                        # Calculando as colunas da matriz inversa
        b = np.zeros(n)                       # Cria o vetor b com valores zeros
        b[i] = 1                              # Monta o vetor b como identidade
        y = substituicao_direta(L, P, b)      # Calcula o valor de y por substituição direta
        x = substituicao_reversa(U, y)        # Calcula o vetor de x por substituição reversa
        A_inv[:, i] = x                       # fixa a coluna na posição i e preenche as linhas com o reultados de x
    return A_inv                              # Retorna a matriz Inversa

if __name__ == "__main__":
    A = np.array([[3.0, -4.0, 1.0],
                  [1.0, 2.0, 2.0],
                 [4.0, 0.0, -3.0]])
    n = len(A)
    A_inv = inverse_matrix(A)
    print("Matriz Inversa:")
    print(A_inv)
    print("\nConferência:")
    X = np.dot(A, A_inv)          # Multiplica a matriz original e a matriz inversa, esperando a matriz identidade
    Y = np.zeros((n, n))          # Cria a matriz auxiliar Y com zeros para realizar a rotina de verificação.
    for i in range(0,n):
        for j in range (0,n):
            if (i==j):
                Y[i][j] = 1       # Preenche Y com a matriz identidade
    print(X-Y)                    # faz a subtração do resultado e a matriz identidade para veririfcação do resultado
```

```
/home/alessandro/PycharmProjects/pythonProject/geraMatriz/venv/bin/python /home/alessandro/PycharmProjects/pythonProject/lista_8/Matriz_inversa.py
Matriz Inversa:
[[ 0.08571429  0.17142857  0.14285714]
 [-0.15714286  0.18571429  0.07142857]
 [ 0.11428571  0.22857143 -0.14285714]]

Conferência:
[[ 0.00000000e+00 -8.32667268e-17 -5.55111512e-17]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00]]

Process finished with exit code 0
```

Como pode-se observar a matriz com o nome de conferência deveria ser uma matriz nula, no entanto, pode-se observar que os valores da primeira linha e colunas 2 e 3 apresentam valores não nulos. No entanto, se observarmos bem são valores extremamente pequenos, na ordem de 10^{-17}, ou seja podemos considera-los como sendo 0.

Obs: Essa operação foi para uma matriz 3x3, a medida que aumenramos a dimensão da matriz esse valor residual tende a aumentar.