MINISTÉRIO DA DEFESA
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
Seção de Engenharia de Defesa (SE/10)

EE 600200: Álgebra Linear Computacional

Lista de Exercícios #4

Cap Suzane GAERTNER Martins

Rio de Janeiro, RJ
JUNHO de 2025

1ª Questão

[FORD] Exercício 11.1: Dada a matriz

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 4 \\ 3 & 5 & 4 \end{bmatrix}$$

encontre a fatoração LU de A passo a passo sem pivotamento.

Solução
Realizando a fatoração LU de A **sem pivotamento**, tal que $A = LU$, onde:

- L é uma matriz triangular inferior com 1's na diagonal.
- U é uma matriz triangular superior.

Passo 1: Inicializar as matrizes L e U

Inicialmente definimos:

$$U = A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 4 \\ 3 & 5 & 4 \end{bmatrix}$$

$$L = I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Passo 2: Zerando os elementos abaixo do pivô (1,1)

O pivô é $U_{11} = 1$.
Calculamos os multiplicadores:

$$m_{21} = \frac{U_{21}}{U_{11}} = \frac{2}{1} = 2$$

$$m_{31} = \frac{U_{31}}{U_{11}} = \frac{3}{1} = 3$$

Atualizando as linhas:

Linha 2: $U_2 = U_2 - m_{21} \cdot U_1 = [2, 5, 4] - 2 \cdot [1, 2, 3] = [0, 1, -2]$

Linha 3: $U_3 = U_3 - m_{31} \cdot U_1 = [3, 5, 4] - 3 \cdot [1, 2, 3] = [0, -1, -5]$

Atualizamos também L com os multiplicadores calculados:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix} \quad \text{e} \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -2 \\ 0 & -1 & -5 \end{bmatrix}$$

Passo 3: Zerando o elemento (3,2)

O pivô agora é $U_{22} = 1$.
Calculamos:

$$m_{32} = \frac{U_{32}}{U_{22}} = \frac{-1}{1} = -1$$

Atualizando a linha 3 de U:

$$U_3 = U_3 - m_{32} \cdot U_2 = [0, -1, -5] - (-1) \cdot [0, 1, -2] = [0, 0, -3]$$

Atualizando L:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix} \quad \text{e} \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & -3 \end{bmatrix}$$

Resultado Final

Portanto, a fatoração LU é:

$$A = LU$$

com

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & -3 \end{bmatrix}$$

Verificação

Verificamos que:

$$L \cdot U = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & -3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 4 \\ 3 & 5 & 4 \end{bmatrix}$$

Portanto, a fatoração está correta.

2ª Questão

Implemente o Algoritmo de Eliminação de Gauss com Pivoteamento Parcial [FORD, Algorithm 11.2] em Python. Faça comparações dos resultados do seu algoritmo implementado com a função scipy.linalg.lu. (Mas atenção... Há uma leve diferença entre as definições da matriz de permutação no livro do FORD e na implementação do SciPy)

Solução

Arquivo "Gauss_pivo.py"

```python
import numpy as np
from scipy.linalg import lu

def ludecomp(A):
    A = A.copy().astype(float)
    n = A.shape[0] #Verifica se a matriz é quadrada
    if A.shape[1] != n:
        raise ValueError("A matriz de entrada deve ser quadrada.")

    L = np.eye(n)
    P = np.eye(n)

    for i in range(n - 1):
        # Pivoteamento parcial
        max_row = np.argmax(abs(A[i:n, i])) + i
        if max_row != i:
            # Trocar linhas em A (somente da coluna i até o fim)
            A[[i, max_row], i:] = A[[max_row, i], i:]
            # Trocar linhas em P (linha inteira)
            P[[i, max_row], :] = P[[max_row, i], :]
            # Trocar linhas em L (somente nas colunas anteriores a i)
            L[[i, max_row], :i] = L[[max_row, i], :i]

        # Calcula os multiplicadores
        multipliers = A[i + 1:n, i] / A[i, i]
        L[i + 1:n, i] = multipliers

        # Atualiza a submatriz
        A[i + 1:n, i + 1:n] = A[i + 1:n, i + 1:n] - np.outer(multipliers, A[i, i + 1:])
```

```

        # Zera explicitamente abaixo do pivô
        A[i + 1:n, i] = 0.0

    U = A
    return L, U, P

# Exemplo de teste
A =  np . array ([[2 , 1 , 5] , [ 4 , 4 , -4] , [ 1 , 3 , 1]] , dtype = float )

# Usando a implementação própria
L1, U1, P1 = ludecomp(A)

# Usando scipy.linalg.lu
P2, L2, U2 = lu(A)

print("Matriz A:")
print(A)
print("\n--- Implementação própria ---")
print("P:")
print(P1)
print("L:")
print(L1)
print("U:")
print(U1)

print("\n--- Scipy LU ---")
print("P:")
print(P2)
print("L:")
print(L2)
print("U:")
print(U2)

# Verificação se PA = LU
print("\nVerificação da implementação própria:")
print("PA  LU ?", np.allclose(P1 @ A, L1 @ U1))

print("\nVerificação da Scipy:")
print("A  PLU ?", np.allclose(A, P2 @ L2 @ U2))
```

Análise da Decomposição LU: LUDECOMP vs. `scipy.linalg.lu`

1. Definições

A decomposição LU com pivoteamento parcial pode ser expressa de duas formas diferentes, dependendo da convenção adotada.

a) Convenção clássica (FORD)

A decomposição é feita na forma:

$$PA = LU$$

onde:

- P é uma matriz de permutação (atua à esquerda, permutando linhas de A),
- L é uma matriz triangular inferior (com diagonais iguais a 1),
- U é uma matriz triangular superior.

Esta é a convenção utilizada no algoritmo `LUDECOMP` descrito no pseudocódigo de referência.

b) Convenção da função `scipy.linalg.lu`

A decomposição é feita na forma:

$$A = PLU$$

onde:

- P é uma matriz de permutação que atua à **direita** na fatoração, ou seja, a permutação é aplicada diretamente sobre LU para obter A.

2. Relação entre as matrizes P

Ao comparar as duas formas, observa-se que as matrizes de permutação P estão diretamente relacionadas pela transposição:

$$P_{\text{scipy}} = P_{\text{LUDECOMP}}^{\top}$$

Isto ocorre porque:

- Na convenção $PA = LU$, a matriz P atua permutando linhas de A.

- Na convenção $A = PLU$, a permutação é aplicada após a multiplicação de LU, ou seja, equivale a uma permutação à direita.

A transposição da matriz de permutação troca permutação de linhas por permutação de colunas e vice-versa, o que explica essa diferença.

3. Verificação Computacional

Ao calcular a decomposição de uma matriz A utilizando ambas as abordagens, verifica-se que:

$$P_{\text{scipy}} \approx P_{\text{LUDECOMP}}^{\top}$$

e as relações fundamentais são satisfeitas:

$$P_{\text{LUDECOMP}} A = LU$$

e

$$A = P_{\text{scipy}} LU$$

```
--- Implementação própria ---
P:
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
L:
[[ 1.    0.    0.  ]
 [ 0.25  1.    0.  ]
 [ 0.5  -0.5   1.  ]]
U:
[[ 4.  4. -4.]
 [ 0.  2.  2.]
 [ 0.  0.  8.]]
```

(a) Matrizes geradas pela função LUDECOMP.

```
--- Scipy LU ---
P:
[[0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]]
L:
[[ 1.    0.    0.  ]
 [ 0.25  1.    0.  ]
 [ 0.5  -0.5   1.  ]]
U:
[[ 4.  4. -4.]
 [ 0.  2.  2.]
 [ 0.  0.  8.]]
```

(b) Matrizes geradas pela função scipy.linalg.lu.

```
Verificação da implementação própria:
PA ≈ LU ? True

Verificação da Scipy:
A ≈ PLU ? True
PS C:\Users\suzi_\OneDrive\Área de Trabalho\Mestrado\Alg Lin Comp>
```

Figura 2: Verificação.

4. Conclusão

Ambas as decomposições são matematicamente corretas e produzem os mesmos fatores L e U, desde que a matriz de permutação seja interpretada de acordo com a convenção adotada.

O entendimento dessa diferença é crucial para evitar erros de interpretação ao comparar resultados de diferentes bibliotecas e algoritmos de fatoração LU.

$3^{\underline{a}}$ Questão

[FORD] Exercício 14.14: Encontre uma base ortonormal para o espaço coluna da matriz A.

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

Solução

Passo 1: Obter uma base para o espaço coluna

As colunas de A são:

$$v_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} -2 \\ 0 \\ 1 \\ 3 \end{bmatrix}$$

Testamos a independência linear. Não existe escalar α tal que

$$\alpha \cdot v_1 = v_2$$

pois isso exigiria

$$\alpha = -2, 0, 1, 3$$

ao mesmo tempo, o que é impossível. Portanto, os vetores são linearmente independentes e formam uma base do espaço coluna.

Passo 2: Processo de Gram-Schmidt

- **Primeiro vetor ortonormal**

Tomamos

$$u_1 = v_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Calculamos a norma:

$$\|u_1\| = \sqrt{1^2 + 1^2 + 1^2 + 1^2} = \sqrt{4} = 2$$

Portanto, o primeiro vetor ortonormal é:

$$e_1 = \frac{u_1}{\|u_1\|} = \frac{1}{2}\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = \begin{bmatrix}0.5\\0.5\\0.5\\0.5\end{bmatrix}$$

- **Segundo vetor ortonormal**

  Calculamos a projeção de v_2 sobre u_1:

$$\text{proj}_{u_1}(v_2) = \frac{\langle v_2, u_1\rangle}{\langle u_1, u_1\rangle}u_1$$

O produto interno é:

$$\langle v_2, u_1\rangle = (-2)\cdot 1 + 0\cdot 1 + 1\cdot 1 + 3\cdot 1 = 2$$

e

$$\langle u_1, u_1\rangle = 4$$

Logo,

$$\text{proj}_{u_1}(v_2) = \frac{2}{4}\cdot\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = \begin{bmatrix}0.5\\0.5\\0.5\\0.5\end{bmatrix}$$

Calculamos o vetor ortogonal:

$$u_2 = v_2 - \text{proj}_{u_1}(v_2)$$

$$u_2 = \begin{bmatrix}-2\\0\\1\\3\end{bmatrix} - \begin{bmatrix}0.5\\0.5\\0.5\\0.5\end{bmatrix} = \begin{bmatrix}-2.5\\-0.5\\0.5\\2.5\end{bmatrix}$$

Calculamos sua norma:

$$\|u_2\| = \sqrt{(-2.5)^2 + (-0.5)^2 + 0.5^2 + 2.5^2}$$
$$= \sqrt{6.25 + 0.25 + 0.25 + 6.25} = \sqrt{13}$$

Portanto, o segundo vetor ortonormal é:

$$e_2 = \frac{u_2}{\sqrt{13}} = \begin{bmatrix}-\frac{2.5}{\sqrt{13}}\\-\frac{0.5}{\sqrt{13}}\\\frac{0.5}{\sqrt{13}}\\\frac{2.5}{\sqrt{13}}\end{bmatrix}$$

Resposta Final

Uma base ortonormal para o espaço coluna de A é:

$$\left\{ \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}, \begin{bmatrix} -\frac{2.5}{\sqrt{13}} \\ -\frac{0.5}{\sqrt{13}} \\ \frac{0.5}{\sqrt{13}} \\ \frac{2.5}{\sqrt{13}} \end{bmatrix} \right\}$$

$4^{\underline{a}}$ Questão

[FORD] Exercício 14.7:

(a) Seja $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ com determinante $det(A) = ad - bc > 0$. Encontre a decomposição QR de A.

(b) O processo de Gram-Schmidt falha se os vetores coluna forem linearmente dependentes. Usando o resultado do item (a), mostre como essa falha ocorre.

Solução
a) Decomposição QR

Passo 1: Primeiro vetor ortonormal

O primeiro vetor coluna de A é

$$v_1 = \begin{bmatrix} a \\ c \end{bmatrix}$$

Calculando sua norma:

$$r_{11} = \|v_1\| = \sqrt{a^2 + c^2}$$

O vetor ortonormal correspondente é

$$e_1 = \frac{v_1}{\|v_1\|} = \frac{1}{\sqrt{a^2 + c^2}} \begin{bmatrix} a \\ c \end{bmatrix}$$

Passo 2: Projeção do segundo vetor

O segundo vetor coluna é

$$v_2 = \begin{bmatrix} b \\ d \end{bmatrix}$$

Calculando o coeficiente:

$$r_{12} = e_1^T v_2 = \frac{ab + cd}{\sqrt{a^2 + c^2}}$$

Passo 3: Vetor ortogonal a e_1

Calculando:

$$u_2 = v_2 - r_{12} e_1$$

Explicitamente,

$$u_2 = \frac{ad - bc}{a^2 + c^2} \begin{bmatrix} -c \\ a \end{bmatrix}$$

Passo 4: Segundo vetor ortonormal

Norma de u_2:

$$r_{22} = \|u_2\| = \frac{ad - bc}{\sqrt{a^2 + c^2}}$$

Portanto,

$$e_2 = \frac{1}{\sqrt{a^2 + c^2}} \begin{bmatrix} -c \\ a \end{bmatrix}$$

Passo 5: Matrizes Q e R

A matriz Q é:

$$Q = \frac{1}{\sqrt{a^2 + c^2}} \begin{bmatrix} a & -c \\ c & a \end{bmatrix}$$

A matriz R é:

$$R = \frac{1}{\sqrt{a^2 + c^2}} \begin{bmatrix} a^2 + c^2 & ab + cd \\ 0 & ad - bc \end{bmatrix}$$

Resultado final

A decomposição QR é

$$A = QR = \frac{1}{a^2 + c^2} \begin{bmatrix} a & -c \\ c & a \end{bmatrix} \begin{bmatrix} a^2 + c^2 & ab + cd \\ 0 & ad - bc \end{bmatrix}$$

b) Quebra do Gram-Schmidt

O processo de Gram-Schmidt falha se as colunas de A forem linearmente dependentes, ou seja, quando:

$$\det(A) = ad - bc = 0$$

Nesse caso, o vetor v_2 é múltiplo de v_1:

$$v_2 = \lambda v_1$$

Ao aplicar Gram-Schmidt, a projeção de v_2 sobre q_1 é exatamente v_2:

$$u_2 = v_2 - \mathrm{proj}_{q_1}(v_2) = v_2 - v_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Portanto, sua norma é zero:

$$\|u_2\| = 0$$

e não é possível calcular:

$$q_2 = \frac{u_2}{\|u_2\|}$$

pois envolveria uma divisão por zero. Assim, o processo se quebra.

Conclusão

O processo de Gram-Schmidt funciona se, e somente se, as colunas de A forem linearmente independentes, o que ocorre quando:

$$\det(A) = ad - bc \neq 0$$

Se $\det(A) = 0$, o processo falha, pois a segunda norma se anula.

5ª Questão

Implemente o algoritmo da decomposição QR por ortogonalização de Gram-Schmidt modificado [FORD, Algorithm 14.3] em Python. Utilize o algoritmo implementado para resolver a questão [FORD] Exercício 14.15. Faça comparações dos resultados com outra função padronizada, tal como numpy.linalg.qr:

Solução

Arquivo "gram_schimdt.py"

```
import numpy as np

def mod_gram_schmidt(A):
    A = A.copy().astype(float)
    m, n = A.shape

    Q = np.zeros((m, n))
    R = np.zeros((n, n))

    for i in range(n):
        Q[:, i] = A[:, i]
        for j in range(i):
            R[j, i] = np.dot(Q[:, j], Q[:, i])
            Q[:, i] = Q[:, i] - R[j, i] * Q[:, j]
        R[i, i] = np.linalg.norm(Q[:, i])
        Q[:, i] = Q[:, i] / R[i, i]

    return Q, R

# [FORD] Exercício 14.15
A = np.array([
    [1, 9, 0, 5, 3, 2],
    [-6, 3, 8, 2, -8, 0],
    [3, 15, 23, 2, 1, 7],
    [3, 57, 35, 1, 7, 9],
    [3, 5, 6, 15, 55, 2],
    [33, 7, 5, 3, 5, 7]
], dtype=float)

Q, R = mod_gram_schmidt(A)

```

```
print("Matriz A:")
print(A)

print("\n--- Base Ortonormal (Colunas da Matriz Q) ---")
print("As colunas da matriz Q abaixo formam uma base ortonormal para o espaço gerado pelas colunas de A:")
print(np.round(Q, 5))

print("\nMatriz R (triangular superior):")
print(np.round(R, 5))


# Verificação das propriedades:
print("\nVerificação se A  QR:", np.allclose(A, Q @ R))
print("Verificação se Q^T Q  I:", np.allclose(Q.T @ Q, np.eye(Q.shape[1])))

# Executa a decomposição QR com a função do NumPy
Q_np, R_np = np.linalg.qr(A)

print("\n" + "="*50)
print("COMPARAÇÃO DOS RESULTADOS")
print("="*50 + "\n")

# --- Resultados do Gram-Schmidt Modificado ---
print("--- Resultados do Gram-Schmidt Modificado ---")
print("Matriz Q (MGS):")
print(np.round(Q, 5))
print("\nMatriz R (MGS):")
print(np.round(R, 5))

# --- Resultados da implementação do NumPy ---
print("\n--- Implementação do NumPy (numpy.linalg.qr) ---")
print("Matriz Q (NumPy):")
print(np.round(Q_np, 5))
print("\nMatriz R (NumPy):")
print(np.round(R_np, 5))

print("\n" + "="*50)
print("VERIFICAÇÃO DAS PROPRIEDADES")
print("="*50 + "\n")

# 1. Verificação da Reconstrução: A = QR
```

```python
reconstruction_mgs_ok = np.allclose(Q @ R, A)
reconstruction_np_ok = np.allclose(Q_np @ R_np, A)

print(f"Reconstrução A = QR (Função Gram-Schmidt Modificado): {reconstruction_mgs_ok}")
print(f"Reconstrução A = QR (Função NumPy): {reconstruction_np_ok}")

# 2. Verificação da Ortonormalidade: Q.T @ Q = I
orthonormality_mgs_ok = np.allclose(Q.T @ Q, np.identity(A.shape[1]))
orthonormality_np_ok = np.allclose(Q_np.T @ Q_np, np.identity(A.shape[1]))

print(f"\nOrtonormalidade de Q (Função Gram-Schmidt Modificado): {orthonormality_mgs_ok}")
print(f"Ortonormalidade de Q (Função NumPy): {orthonormality_np_ok}")

# 3. Comparação direta ignorando os sinais
# Comparamos os valores absolutos para ver se as bases são essencialmente as mesmas
q_abs_close = np.allclose(np.abs(Q), np.abs(Q_np))
r_abs_close = np.allclose(np.abs(R), np.abs(R_np))

print(f"\nOs valores absolutos das matrizes Q são próximos? {q_abs_close}")
print(f"Os valores absolutos das matrizes R são próximos? {r_abs_close}")

```

- [FORD, Exercício 14.15]: Use o algoritmo MODGRSCH para encontrar uma base ortonormal para as colunas da matriz A.

$$A = \begin{bmatrix} 1 & 9 & 0 & 5 & 3 & 2 \\ -6 & 3 & 8 & 2 & -8 & 0 \\ 3 & 15 & 23 & 2 & 1 & 7 \\ 3 & 57 & 35 & 1 & 7 & 9 \\ 3 & 5 & 6 & 15 & 55 & 2 \\ 33 & 7 & 5 & 3 & 5 & 7 \end{bmatrix}$$

Algoritmo Gram-Schmidt Modificado

O algoritmo calculou primeiramente a decomposição QR da matriz `A` usando a sua função `mod_gram_schmidt`.

Matriz A:

$$A = \begin{bmatrix} 1. & 9. & 0. & 5. & 3. & 2. \\ -6. & 3. & 8. & 2. & -8. & 0. \\ 3. & 15. & 23. & 2. & 1. & 7. \\ 3. & 57. & 35. & 1. & 7. & 9. \\ 3. & 5. & 6. & 15. & 55. & 2. \\ 33. & 7. & 5. & 3. & 5. & 7. \end{bmatrix}$$

Matriz Ortonormal Q Resultante (arredondada para 5 casas decimais):

$$Q = \begin{bmatrix} 0.02945 & 0.14632 & -0.37466 & 0.35977 & -0.69045 & 0.48083 \\ -0.17670 & 0.09108 & 0.37632 & 0.09368 & -0.60665 & -0.66488 \\ 0.08835 & 0.23497 & 0.80927 & -0.08915 & -0.07076 & 0.51876 \\ 0.08835 & 0.94899 & -0.18913 & -0.11134 & 0.14565 & -0.14911 \\ 0.08835 & 0.06496 & 0.16509 & 0.91541 & 0.33611 & -0.09884 \\ 0.97185 & -0.10141 & 0.00839 & -0.05886 & -0.12674 & -0.16008 \end{bmatrix}$$

Matriz R Resultante (arredondada para 5 casas decimais):

$$R = \begin{bmatrix} 33.95585 & 13.34085 & 9.10005 & 4.29970 & 11.92725 & 8.45215 \\ 0. & 58.82195 & 39.23021 & 3.00293 & 9.65422 & 9.89836 \\ 0. & 0. & 16.03682 & 2.81021 & 4.47247 & 3.60234 \\ 0. & 0. & 0. & 15.25115 & 49.51474 & 0.51215 \\ 0. & 0. & 0. & 0. & 21.58297 & -0.78034 \\ 0. & 0. & 0. & 0. & 0. & 1.93273 \end{bmatrix}$$

As propriedades da decomposição foram verificadas e confirmadas como corretas:

- **Reconstrução:** `A` é aproximadamente igual a `Q @ R`.
- **Ortonormalidade:** `Q.T @ Q` é aproximadamente igual à matriz identidade.

Comparação com o `linalg.qr` do NumPy

O código então comparou os resultados do algoritmo com aqueles da função nativa `linalg.qr` do NumPy.

Principais Conclusões:

- **Reconstrução e Ortonormalidade:** Tanto a sua implementação quanto a do NumPy produziram decomposições QR válidas, satisfazendo as propriedades de reconstrução e ortonormalidade.
  - `Reconstrução A = QR (Função Gram-Schmidt Modificado): True`
  - `Reconstrução A = QR (Função NumPy): True`
  - `Ortonormalidade de Q (Função Gram-Schmidt Modificado): True`
  - `Ortonormalidade de Q (Função NumPy): True`
- **Diferenças de Sinal:** As matrizes `Q` e `R` resultantes do algoritmo e da função do NumPy possuem sinais diferentes para algumas colunas e linhas. Isso é esperado, pois os sinais dos vetores na decomposição QR não são únicos. Desde que o produto `Q @ R` reconstrua `A`, a decomposição é válida.
- **Comparação de Valor Absoluto:** Os valores absolutos das matrizes `Q` e `R` de ambas as implementações são muito próximos, confirmando que elas representam essencialmente a mesma decomposição.
  - `Os valores absolutos das matrizes Q são próximos? True`
  - `Os valores absolutos das matrizes R são próximos? True`

Em conclusão, a implementação do algoritmo de Gram-Schmidt Modificado está funcionando corretamente e produz uma decomposição QR válida da matriz de entrada `A`, onde as colunas da matriz Q gerada são uma base ortonormal das colunas da matriz A.