**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 Filipe José Americano Albeny

Rio de Janeiro, RJ
Junho de 2025

# 1ª Questão

# Ford - 11.1

Seja a matriz:

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

Vamos encontrar a fatoração $A = LU$, onde:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

## Passo 1: Eliminação de Gauss

Aplicamos a eliminação de Gauss à matriz $A$, sem pivotamento:

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

Eliminamos os elementos abaixo do pivô $a_{11} = 1$:

- $l_{21} = \frac{2}{1} = 2 \quad$ (L2 := L2 $\;2 \times$ L1)
- $l_{31} = \frac{3}{1} = 3 \quad$ (L3 := L3 $\;3 \times$ L1)

$$\text{Nova L2} = [2\ 5\ 4] - 2 \cdot [1\ 2\ 3] = [0\ 1\ -2]$$
$$\text{Nova L3} = [3\ 5\ 4] - 3 \cdot [1\ 2\ 3] = [0\ -1\ -5]$$

Matriz após a primeira eliminação:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -2 \\ 0 & -1 & -5 \end{bmatrix}$$

Agora eliminamos o elemento $a_{32} = -1$ (abaixo do pivô da 2ª coluna):

- $l_{32} = \frac{-1}{1} = -1 \quad$ (L3 := L3 + L2)

$$\text{Nova L3} = [0 \ -1 \ -5] + [0 \ 1 \ -2] = [0 \ 0 \ -7]$$

Matriz triangular superior $U$:

$$U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & -7 \end{bmatrix}$$

## Passo 2: Construção de $L$

Os multiplicadores usados na eliminação formam os elementos abaixo da diagonal de $L$:

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

## Resultado Final

A fatoração $LU$ da matriz $A$ é:

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & -7 \end{bmatrix}$$

# 2ª Questão

## [FORD] Algoritmo 11.2

### Código Python Completo

Listing 1: Script

```
import numpy as np
import scipy.linalg

def lu_decomp_partial_pivoting(A):
    """
    Decomposicao LU com pivotamento parcial (Algoritmo
       de Gauss).
    Retorna L, U, P tal que PA = LU.
    """
    A = A.copy().astype(float)
    n = A.shape[0]
    L = np.eye(n)
    P = np.eye(n)

    for i in range(n):
        pivot_index = np.argmax(np.abs(A[i:, i])) + i
        if pivot_index != i:
            A[[i, pivot_index]] = A[[pivot_index, i]]
            P[[i, pivot_index]] = P[[pivot_index, i]]
            if i > 0:
                L[[i, pivot_index], :i] = L[[pivot_index
                    , i], :i]
        for j in range(i+1, n):
            L[j, i] = A[j, i] / A[i, i]
            A[j] -= L[j, i] * A[i]
    U = np.triu(A)
    return L, U, P

# === MATRIZ 3x3 ===
A1 = np.array([
    [2, 3, 1],
    [4, 7, 2],
    [6, 18, -1]
], dtype=float)

# === MATRIZ 4x4 ===
```

```
A2 = np.array([
    [14.,  4.,  5.,  3.],
    [ 4., 13.,  8.,  3.],
    [ 2.,  1.,  2., 11.],
    [ 1., 13.,  6.,  3.]
], dtype=float)

# === Funcao para exibir os resultados de uma matriz ===
def mostrar_resultados(A, nome):
    print("\n" + "="*10 + f" {nome} " + "="*10)
    print("Matriz A:")
    print(A)

    # Implementacao propria
    L1, U1, P1 = lu_decomp_partial_pivoting(A)
    print("\n--- Implementacao Propria ---")
    print("P =\n", P1)
    print("L =\n", L1)
    print("U =\n", U1)

    # Funcao SciPy
    P2, L2, U2 = scipy.linalg.lu(A)
    print("\n--- SciPy ---")
    print("P =\n", P2)
    print("L =\n", L2)
    print("U =\n", U2)

    # Validacao
    print("\nValidacao:")
    print("P @ A aproximadamente igual a L @ U (
       implementacao)?", np.allclose(P1 @ A, L1 @ U1))
    print("P @ A     L @ U (SciPy)?", np.allclose(np.dot
       (np.linalg.inv(P2), A), np.dot(L2, U2)))

# Mostrar os resultados para ambas as matrizes
mostrar_resultados(A1, "Matriz 3x3")
mostrar_resultados(A2, "Matriz 4x4")
```

## Resultado script:

**Matriz $A$:**

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 7 & 2 \\ 6 & 18 & -1 \end{bmatrix}$$

**Implementação Própria**

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.66666667 & 1 & 0 \\ 0.33333333 & 0.6 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 6 & 18 & -1 \\ 0 & -5 & 2.66666667 \\ 0 & 0 & -0.26666667 \end{bmatrix}$$

**SciPy**

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.66666667 & 1 & 0 \\ 0.33333333 & 0.6 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 6 & 18 & -1 \\ 0 & -5 & 2.66666667 \\ 0 & 0 & -0.26666667 \end{bmatrix}$$

**Validação:**

- $PA \approx LU$ (implementação)? **True**
- $PA \approx LU$ (SciPy)? **True**

## Matriz $4 \times 4$

**Matriz $A$:**

$$A = \begin{bmatrix} 14 & 4 & 5 & 3 \\ 4 & 13 & 8 & 3 \\ 2 & 1 & 2 & 11 \\ 1 & 13 & 6 & 3 \end{bmatrix}$$

**Implementação Própria**

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.07142857 & 1 & 0 & 0 \\ 0.28571429 & 0.93258427 & 1 & 0 \\ 0.14285714 & 0.03370787 & 0.83690987 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 14 & 4 & 5 & 3 \\ 0 & 12.71428571 & 5.64285714 & 2.78571429 \\ 0 & 0 & 1.30898876 & -0.45505618 \\ 0 & 0 & 0 & 10.8583691 \end{bmatrix}$$

**SciPy**

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.07142857 & 1 & 0 & 0 \\ 0.28571429 & 0.93258427 & 1 & 0 \\ 0.14285714 & 0.03370787 & 0.83690987 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 14 & 4 & 5 & 3 \\ 0 & 12.71428571 & 5.64285714 & 2.78571429 \\ 0 & 0 & 1.30898876 & -0.45505618 \\ 0 & 0 & 0 & 10.8583691 \end{bmatrix}$$

**Validação:**

- $PA \approx LU$ (implementação)? **True**
- $PA \approx LU$ (SciPy)? **True**

Neste experimento, aplicamos o algoritmo de eliminação de Gauss com pivotamento parcial sobre duas matrizes distintas, utilizando tanto uma implementação manual quanto a função `scipy.linalg.lu` da biblioteca SciPy. A seguir, analisamos os resultados.

## Matriz 1 – Caso 3×3

$$A_1 = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 7 & 2 \\ 6 & 18 & -1 \end{bmatrix}$$

Para essa matriz, ambas as abordagens produziram as mesmas matrizes de permutação $P$, triangular inferior $L$ e triangular superior $U$. Isso ocorre porque os pivôs em cada etapa da eliminação foram claramente definidos, sem ambiguidade. Portanto, o caminho seguido pelas duas implementações foi idêntico.

## Matriz 2 – Caso 4×4 com diferenças

$$A_2 = \begin{bmatrix} 14 & 4 & 5 & 3 \\ 4 & 13 & 8 & 3 \\ 2 & 1 & 2 & 11 \\ 1 & 13 & 6 & 3 \end{bmatrix}$$

Nesta matriz, observamos que as decomposições $PA = LU$ geradas pelas duas abordagens resultaram em matrizes $P$, $L$ e $U$ distintas, embora ambas satisfaçam corretamente a identidade $PA = LU$.

## Explicação das Diferenças

As diferenças observadas se devem a três fatores principais:

1. **Ambiguidade no pivotamento parcial:** quando há múltiplos elementos com valores absolutos próximos em uma coluna, a escolha do pivô pode variar entre implementações. A função do SciPy, por exemplo, pode priorizar o menor índice de linha, enquanto a implementação manual segue o critério do `np.argmax`.

2. **Não unicidade da decomposição LU com pivotamento:** embora a decomposição $A = LU$ seja única sob certas condições quando não há permutação, o mesmo não é verdade quando usamos pivotamento. Existem múltiplas decomposições válidas que satisfazem $PA = LU$ com diferentes $P$, $L$ e $U$.

3. **Erros numéricos:** pequenas diferenças nas casas decimais ($\sim 10^{-16}$) foram observadas, o que é esperado em ambientes de aritmética de ponto flutuante e não afeta a validade da decomposição.

## Validação

Em ambos os casos, verificamos numericamente que as decomposições são corretas utilizando:

$$\texttt{np.allclose(P @ A, L @ U)} \Rightarrow \text{True}$$

Portanto, mesmo que os fatores $L$, $U$ e $P$ variem, a identidade fundamental $PA = LU$ é preservada em ambas as abordagens.

# 3ª Questão

## [FORD] Exercício 14.14

Dada a matriz:

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

Queremos encontrar uma base ortonormal para o espaço coluna de $A$, utilizando o **processo de Gram-Schmidt**.

## Passo 1: Normalizar $v_1$

$$v_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \|v_1\| = \sqrt{1^2 + 1^2 + 1^2 + 1^2} = 2$$

$$e_1 = \frac{v_1}{\|v_1\|} = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

## Passo 2: Ortogonalizar $v_2$ em relação a $e_1$

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

$$\text{proj}_{e_1}(v_2) = \langle v_2, e_1 \rangle \cdot e_1 = (1) \cdot e_1 = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

$$u_2 = v_2 - \text{proj}_{e_1}(v_2) = \begin{bmatrix} -2.5 \\ -0.5 \\ 0.5 \\ 2.5 \end{bmatrix}$$

$$\|u_2\| = \sqrt{(-2.5)^2 + (-0.5)^2 + 0.5^2 + 2.5^2} = \sqrt{13} \approx 3.6056$$

$$e_2 = \frac{u_2}{\|u_2\|} = \begin{bmatrix} -0.6934 \\ -0.1387 \\ 0.1387 \\ 0.6934 \end{bmatrix}$$

## Base ortonormal

Portanto, a base ortonormal para o espaço coluna de $A$ é:

$$\left\{ \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}, \begin{bmatrix} -0.6934 \\ -0.1387 \\ 0.1387 \\ 0.6934 \end{bmatrix} \right\}$$

# 4ª Questão

## [FORD] Exercício 14.7 (a)

Seja

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \text{com } \det A = ad - bc > 0$$

**Passo 1:**

$$r_{11} = \sqrt{a^2 + c^2}, \quad e_1 = \frac{1}{\sqrt{a^2 + c^2}} \begin{bmatrix} a \\ c \end{bmatrix}$$

**Passo 2:**

$$r_{12} = \begin{bmatrix} b & d \end{bmatrix} \cdot \frac{1}{\sqrt{a^2 + c^2}} \begin{bmatrix} a \\ c \end{bmatrix} = \frac{ab + cd}{\sqrt{a^2 + c^2}}$$

$$u_2 = \begin{bmatrix} b \\ d \end{bmatrix} - r_{12} e_1 = \begin{bmatrix} b \\ d \end{bmatrix} - \frac{ab + cd}{a^2 + c^2} \begin{bmatrix} a \\ c \end{bmatrix}$$

$$= \frac{1}{a^2 + c^2} \begin{bmatrix} \det A \cdot (-c) \\ \det A \cdot a \end{bmatrix} = \frac{ad - bc}{a^2 + c^2} \begin{bmatrix} -c \\ a \end{bmatrix}$$

**Passo 3:**

$$r_{22} = \|u_2\|_2 = \frac{ad - bc}{\sqrt{a^2 + c^2}}, \quad e_2 = \frac{u_2}{r_{22}} = \frac{1}{\sqrt{a^2 + c^2}} \begin{bmatrix} -c \\ a \end{bmatrix}$$

**Resultado:**

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

# [FORD] Exercício 14.7 (b)

Sejam as colunas da matriz $A \in \mathbb{R}^{2\times 2}$:

$$\mathbf{v}_1 = \begin{bmatrix} a \\ c \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} b \\ d \end{bmatrix}$$

Se $\mathbf{v}_2$ for linearmente dependente de $\mathbf{v}_1$, então existe um escalar $\lambda \in \mathbb{R}$ tal que:

$$\mathbf{v}_2 = \lambda \mathbf{v}_1 \Rightarrow \begin{bmatrix} b \\ d \end{bmatrix} = \lambda \begin{bmatrix} a \\ c \end{bmatrix}$$

No processo de Gram-Schmidt, calculamos:

$$\mathbf{u}_2 = \mathbf{v}_2 - \text{proj}_{\mathbf{e}_1}(\mathbf{v}_2) = \mathbf{v}_2 - \langle \mathbf{v}_2, \mathbf{e}_1 \rangle \cdot \mathbf{e}_1$$

Mas se $\mathbf{v}_2$ for múltiplo de $\mathbf{v}_1$, então $\mathbf{v}_2$ está totalmente na direção de $\mathbf{e}_1$, e portanto:

$$\text{proj}_{\mathbf{e}_1}(\mathbf{v}_2) = \mathbf{v}_2 \Rightarrow \mathbf{u}_2 = \mathbf{v}_2 - \mathbf{v}_2 = \mathbf{0}$$

Então ao tentar calcular $\mathbf{e}_2$, temos:

$$\mathbf{e}_2 = \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} = \frac{0}{0} \quad \text{(indefinido)}$$

## Conclusão

O processo de Gram-Schmidt **falha** quando as colunas de $A$ são linearmente dependentes, pois a subtração da projeção gera o vetor nulo, e este não pode ser normalizado. Portanto, não é possível obter uma base ortonormal completa nesse caso.

# 5ª Questão

```
    # Codigo disponivel em:
    # https://github.com/albenyfjaa/alc-albeny
import numpy as np

def modified_gram_schmidt(A):
    """
    Implementa o Algoritmo 14.3 - Decomposicao QR usando
        Gram-Schmidt Modificado.

    Parametros:
    A : numpy.ndarray (m x n)
        Matriz de entrada (com m linhas e n colunas)

    Retorna:
    Q : numpy.ndarray (m x n)
        Matriz com colunas ortonormais
    R : numpy.ndarray (n x n)
        Matriz triangular superior
    """
    m, n = A.shape
    Q = np.zeros((m, n))
    R = np.zeros((n, n))

    for i in range(n):
        # Inicialmente, q_i recebe a i-esima coluna de A
        Q[:, i] = A[:, i]

        for j in range(i):
            # Projecao de Q[:, i] sobre Q[:, j]
            R[j, i] = np.dot(Q[:, j], Q[:, i])
            Q[:, i] -= R[j, i] * Q[:, j]

        # Normaliza Q[:, i]
        R[i, i] = np.linalg.norm(Q[:, i])
        Q[:, i] = Q[:, i] / R[i, i]

    return Q, R

# Matriz do Exercicio 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 = modified_gram_schmidt(A)

print("Q =\n", Q)
print("R =\n", R)
print("Reconstrucao A aproximadamente igual a Q @ R?\n",
    np.allclose(A, Q @ R))

# Decomposicao QR usando NumPy
Q, R = np.linalg.qr(A)

# Impressao dos resultados
print("Q (numpy.linalg.qr) =\n", Q)
print("\nR (numpy.linalg.qr) =\n", R)

# Verificacao: A aproximadamente igual a QR?
print("\nReconstrucao A aproximadamente igual a Q @ R?\n
   ", np.allclose(A, Q @ R))
```

## Resultados Script:

## Matrizes Resultantes

**Matriz $Q$:**

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

**Matriz $R$:**

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

## Reconstrução da Matriz $A$

$$A \approx Q \cdot R \quad \Rightarrow \quad \text{Reconstrução correta?} \quad \textbf{True}$$

## Comparação com $Q$ e $R$ obtidos diretamente de `numpy.linalg.qr`

**Matriz $Q$ (via `numpy.linalg.qr`):**

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

**Matriz $R$ (via `numpy.linalg.qr`):**

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

## Reconstrução com sinais invertidos

$$A \approx Q \cdot R \quad \text{(também satisfaz)} \quad \textbf{True}$$

*Nota:* os sinais opostos em $Q$ e $R$ não afetam a reconstrução, pois seus produtos continuam equivalentes à matriz $A$ original.

Aplicamos a decomposição QR à matriz abaixo, fornecida no Exercício 14.15 do livro de FORD, utilizando dois métodos distintos:

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

## Métodos Utilizados

1. **Gram-Schmidt Modificado (Algoritmo 14.3)**: implementado manualmente em Python.

2. `numpy.linalg.qr`: função embutida da biblioteca NumPy, baseada em algoritmos mais estáveis numericamente.

## Resultados Obtidos

Ambos os métodos produzem duas matrizes:

- $Q$: matriz com colunas ortonormais
- $R$: matriz triangular superior

Apesar de pequenas diferenças numéricas nos sinais de colunas ou em casas decimais, ambas as abordagens satisfazem a identidade fundamental:

$$A \approx QR \quad \text{(verificada com } \texttt{np.allclose(A, Q @ R)}\text{)}$$

## Conclusão

- O Algoritmo 14.3 (Gram-Schmidt Modificado) é eficaz para calcular uma base ortonormal das colunas de $A$.
- A função `numpy.linalg.qr` é numericamente mais robusta e confiável para casos mal-condicionados.
- Em ambos os métodos, a decomposição $A = QR$ foi validada com sucesso.

## Validação Numérica

```
np.allclose(A, Q @ R) → True
```

Portanto, os dois métodos são consistentes entre si, ainda que sujeitos a pequenas variações numéricas aceitáveis.