# Álgebra Linear

Jeann Rocha

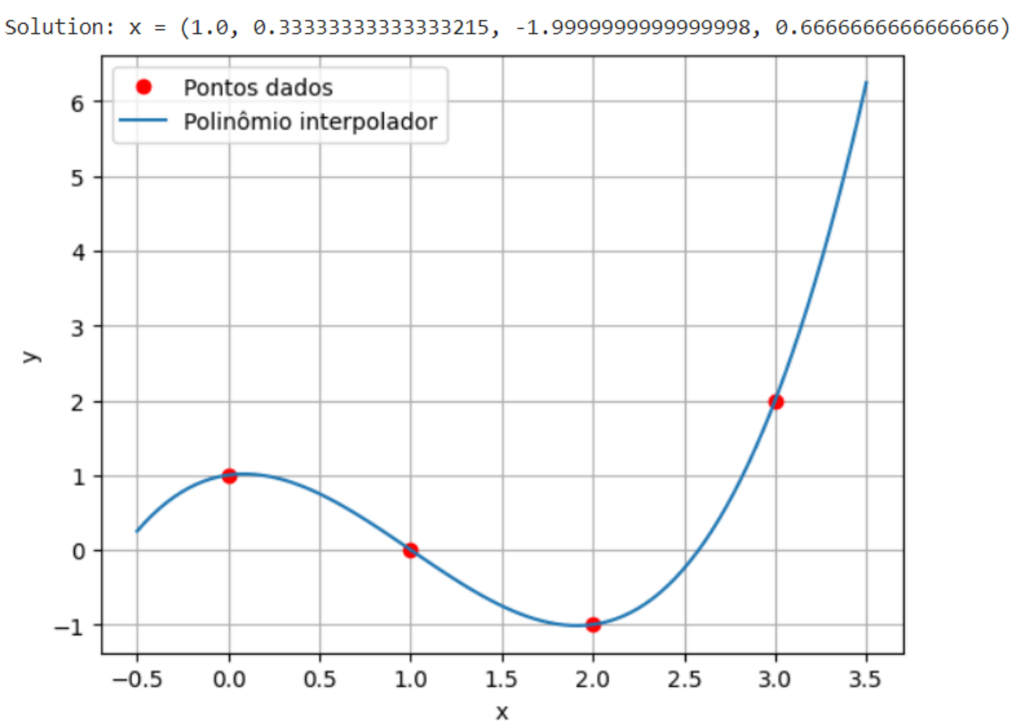
March 2024

## 0 Lista de Exercícios

**Exercício 0.1.** Se existir tal polinômio, ele é da forma $P(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ e para os pontos informados, temos o sistema

$$\begin{cases} a_0 & = P(0) = 1 \\ a_0 + a_1 + a_2 + a_3 & = P(1) = 0 \\ a_0 + 2a_1 + 4a_2 + 8a_3 & = P(2) = -1 \\ a_0 + 3a_1 + 9a_2 + 27a_3 & = P(3) = 2 \end{cases}$$

Resolvendo o sistema, encontramos uma única solução e, portanto, existe o polinômio interpolador. Utilizando a linguagem Python para resolver o sistema e plotar o gráfico do polinômio interpolador e os pontos indicados, obtemos o seguinte resultado



Para fins de verificação, o código utlizado foi o seguinte:

```
import numpy as np
import matplotlib.pyplot as plt

# reported data
points = np.array([[0, 1], [1, 0], [2, -1], [3, 2]])
A = np.array([[1, 0, 0, 0], [1, 1, 1, 1], [1, 2, 4, 8],
    [1, 3, 9, 27]])
b = np.array([1, 0, -1, 2])

# solution of Ax = b
coefficients = np.linalg.solve(A, b)

print(f"Solution: x = {coefficients[0],
    coefficients[1], coefficients[2], coefficients[3]}")

x = np.linspace(-0.5, 3.5, 100)
y = coefficients[0] + coefficients[1] * x +
    coefficients[2] * x**2 + coefficients[3] * x**3

# ploting results
plt.plot(points[:, 0], points[:, 1], 'ro',
    label='Pontos dados')
plt.plot(x, y, label='Polinomio interpolador')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

**Exercício 0.2.** a) Podemos transformar o problema em questão no seguinte sistema linear

$$\begin{cases} x + y + 3z &= 3.8 \\ x + 2y + 4z &= 5.2 \\ 2x + 3y + 7z &= 9 \end{cases}$$

onde a primeira linha representa o valor aplicado na empresa $A$, a segunda representa o valor aplicado em $B$ e a terceira em $C$, todas em escala de 1 para 1000, segundo a vontade de Asdrúbal. Trata-se de um sistema possível e indeterminado, isto é, com infinitas soluções, as quais são descritas na resolução por escalonamento abaixo:

$$\left(\begin{array}{ccc|c} 1 & 1 & 3 & 3.8 \\ 1 & 2 & 4 & 5.2 \\ 2 & 3 & 7 & 9 \end{array}\right) \xrightarrow[L3 \leftarrow L3 - 2L1]{L2 \leftarrow L2 - L1} \left(\begin{array}{ccc|c} 1 & 1 & 3 & 3.8 \\ 0 & 1 & 1 & 1.4 \\ 0 & 1 & 1 & 1.4 \end{array}\right) \xrightarrow{L3 \leftarrow L3 - L2}$$

$$\begin{pmatrix} 1 & 1 & 3 & | & 3.8 \\ 0 & 1 & 1 & | & 1.4 \\ 0 & 0 & 0 & | & 0 \end{pmatrix}$$

Fixando $z = t$ como variável livre, na linha 2, temos $y + z = 1.4 \Leftrightarrow y = 1.4 - t$ e, por conseguinte, na linha 1, $x + y + 3z = 3.8 \Leftrightarrow x = 3.8 - (1.4 - t) - 3t \Leftrightarrow x = 2.4 - 2t$. Logo, o conjunto solução para o sistema é $\{(2.4 - 2t, 1.4 - t, t) | t \in \mathbb{R}\}$. Dado que $x, y, z$ são $\geq 0$, temos $t \geq 0, 1.4 - t \geq 0 \Leftrightarrow t \leq 1.4$ e $2.4 - 2t \geq 0 \Leftrightarrow t \leq 1.2$, donde as possíveis escolhas de Asdrúbal são dadas pela terna $(2400 - 2t, 1400 - t, t)$, com $0 \leq t \leq 1200$ (em escala normal).

b) Para que $y > x$ (em escala normal), devemos ter $1400 - t > 2400 - 2t \Leftrightarrow t > 1000$. Logo, Asdrúbal pode ter mais quotas do fundo $y$ do que do que quotas do fundo $x$ desde que $1000 < t < 1200$, neste contexto.

**Exercício 0.3.** i. Temos

$$\overline{E} \cdot E_{13}(1) \cdot E_{11}(1/3) \cdot B \cdot E_{11}(2) \cdot P_{14} \cdot J \cdot K$$

onde

- $E_{ij}(m)$ é a matriz identidade com o elemento da posição $ij$ trocado por $m$
- $P_{ij}$ é a matriz identidade com as linhas $i$ e $j$ trocadas de lugar
- $\overline{E} = E_{12}(-1)E_{32}(-1)E_{42}(-1)$ é a matriz identidade com $-1$ nas posições $12, 32$ e $42$
- $J$ é a matriz identidade com 1 na posição 34 e 0 na posição 44.
- $K$ é a matriz $4 \times 3$ descrita pela identidade $4 \times 4$ sem a primeira coluna

Mais explicitamente, o produto em questão é

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1/3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$
$$\cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

ii. Basta definir $A = \overline{E} \cdot E_{31}(1) \cdot E_{11}(1/3)$ e $C = E_{11}(2) \cdot P_{14} \cdot J \cdot K$. Explicitamente, temos

$$A \cdot B \cdot C = \begin{bmatrix} 1/3 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

**Exercício 0.4.** Efetuando o escalonamento da matriz, obtemos

$$\begin{pmatrix} 1 & -1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 2 \end{pmatrix} \xrightarrow{L3 \leftarrow L3 - L1} \begin{pmatrix} 1 & -1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \xrightarrow{L3 \leftarrow L3 - L2} \begin{pmatrix} 1 & -1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Como a terceira linha da matriz escalonada possui uma linha de zeros, podemos concluir que ela não possui inversa.

**Exercício 0.5.** Sendo $L = [a_{ij}]_{n \times n}$ triangular inferior, temos $a_{ij} = 0$ se $i < j$. Assim, sejam $b = (b_1, ..., b_n)$ e $x = (x_1, ..., x_n)$ tais que $Lx = b$. Então considere o algoritmo recursivo descrito a seguir:

- Tome $x_1 = \dfrac{b_1}{a_{11}}$
- Já obtidos $x_1, ..., x_m$ $(m < n)$, tome $x_{m+1} = \dfrac{b_{m+1} - a_{m+1,1}x_1 - ... - a_{m+1,m}x_m}{a_{m+1,m+1}}$

Para o primeiro passo, existe 1 operação de produto (dividir por $a_{11}$). Para o segundo passo, existem 2 operações de produto (multiplicar por $a_{21}$ e dividir por $a_{22}$). Para o terceiro, existem 3 (multiplicar por $a_{31}$, por $a_{32}$ e dividir por $a_{33}$). Para o quarto, existem 4 (multiplicar por $a_{41}$, por $a_{42}$, por $a_{43}$ e dividir por $a_{44}$), e assim por diante, até o $n-$ésimo passo, onde existem $n$ operações de produto (multiplicar por $a_{n1}$, por $a_{n2}$, ..., por $a_{n-1,n}$ e dividir por $a_{nn}$). Portanto, a ordem de complexidade desse algoritmo é

$$1 + 2 + ... + n = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} = O(n).$$

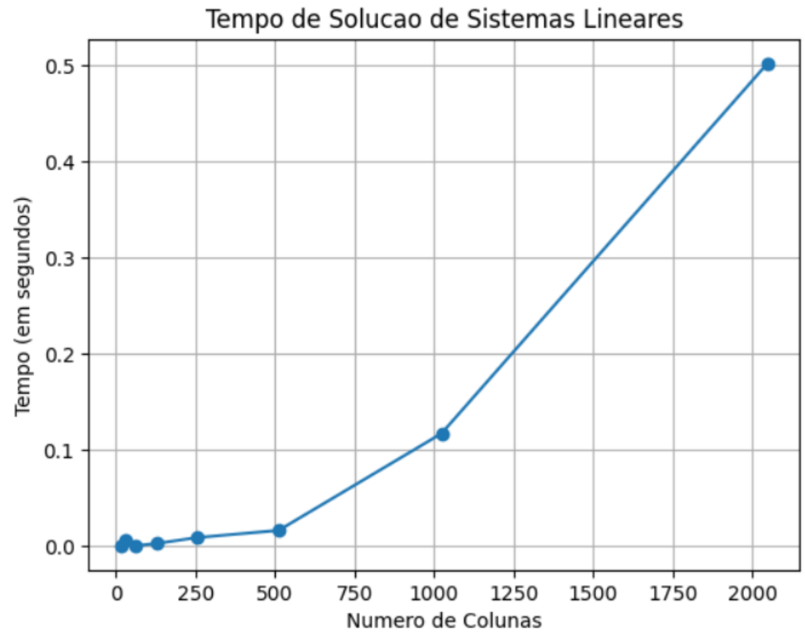
**Exercício 0.6.** Utilizando a linguagem Python para resolver um sistema linear gerado aleatoriamente $Ax = b$ para os valores de número de colunas fornecidos $(16, 32, ..., 2048)$, obtemos os seguintes tempos de execução:

Tempos de Execucao: [0.0006, 0.0061, 0.0007, 0.0029, 0.0091, 0.0165, 0.1174, 0.5023]

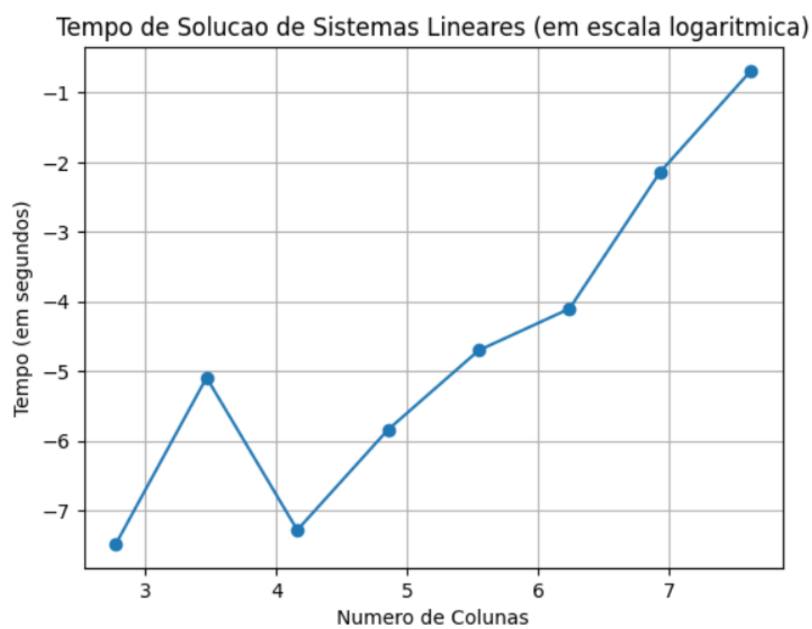
Este código não é $O(n^3)$, pois ao dobrar o número de colunas, seria esperado que o tempo de execução octuplicasse, já que $n \mapsto n^3$ e $2n \mapsto (2n)^3 = 8n^3$. Mas,

1. $8 \cdot 0.0006 = 0.0048 = 0.0061 - 0.0013$
2. $8 \cdot 0.0061 = 0.0488 = 0.0007 + 0.0481$
3. $8 \cdot 0.0007 = 0.0056 = 0.0029 + 0.0027$
4. $8 \cdot 0.0029 = 0.0232 = 0.0091 + 0.0141$
5. $8 \cdot 0.0165 = 0.132 = 0.0165 + 0.1155$
6. $8 \cdot 0.1174 = 0.9392 = 0.5023 + 0.4369$

Ou seja, a sequência de erros vai se tornando cada vez maior, o que significa que $n^3$ não é uma boa ordem de complexidade para este caso. Graficamente, temos:




Tomando o log em ambos os eixos, obtemos um aspecto aparentemente linear, o que dá a entender que a ordem de complexidade não é $O(n^3)$, mas é ainda uma potência, isto é, $\exists \alpha \in \mathbb{R}$ tal que a ordem de complexidade é $O(n^\alpha)$, como segue abaixo:




Para fins de verificação, o código utlizado foi o seguinte:

```python
import numpy as np
import time
import matplotlib.pyplot as plt

# number of columns of the matrix
n_cols = [16, 32, 64, 128, 256, 512, 1024, 2048]

time_exec = [] # time of execution (in seconds)

for n in n_cols:
    # generate a random matriz n x n and a random vector b
    A = np.random.rand(n, n)
    b = np.random.rand(n, 1)

    # time of execution of the system Ax = b
    time_i = time.time()
    x = np.linalg.solve(A, b)
    time_f = time.time()
    time_exec.append(round(time_f- time_i, 6))

print(f"Tempos de Execucao: {time_exec}")

# ploting the time x n_cols graph
plt.figure()
plt.plot(n_cols, time_exec, marker='o')
plt.xlabel('Numero de Colunas')
plt.ylabel('Tempo (em segundos)')
plt.title('Tempo de Solucao de Sistemas Lineares')
plt.grid(True)
plt.show()

# ploting the log(time) x log(n_cols) graph
plt.figure()
plt.plot(np.log(n_cols), np.log(time_exec), marker='o')
plt.xlabel('Numero de Colunas')
plt.ylabel('Tempo (em segundos)')
plt.title('Tempo de Solucao de Sistemas Lineares (em escala logaritmica)')
plt.grid(True)
plt.show()
```