

✓ Lab 1 - BCC406/PCC177

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Pacote *NumPy*

Prof. Eduardo e Prof. Pedro

Objetivos:

- Uso de *NumPy*.

Data da entrega : 10/12/2024

- Complete o código (marcado com 'ToDo') e quando requisitado, escreva textos diretamente nos notebooks. Onde tiver *None*, substitua pelo seu código.
- Execute todo notebook e salve tudo em um PDF **nomeado** como "NomeSobrenome-LabX.pdf"
- Envie o PDF e o o *.ipynb* via [Formulário Google](#).

Sugestão de leitura:

- Ler [Capítulo 2 do livro texto](#). Dê ênfase para as seções 2.3 e 2.4. **Sugerimos fortemente** abrir com o Colab e executar estas duas seções passo a passo.

✓ *NumPy*

NumPy é uma das bibliotecas mais populares para computação científica. Ela foi desenvolvida para dar suporte a operações com *arrays* de N dimensões e implementa métodos úteis para operações de álgebra linear, geração de números aleatórios, etc.

✓ Criando arrays (5pt)

```
1 # Primeiramente, vamos importar a biblioteca
2 import numpy as np
```

```
1 # Usaremos a função zeros para criar um array de uma dimensão de tamanho 5
2 np.zeros(5)
```

```
→ array([0., 0., 0., 0., 0.])
```

```
1 # Da mesma forma, para criar um array de duas dimensões:
2 np.zeros((3,4))
```

```
→ array([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

```
1 # ToDo: Crie um array de três dimensões com o shape (3, 3, 3)
2 np.zeros((3, 3, 3))
```

```
array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

✓ Vocabulário comum (25pt)

- Em *NumPy*, cada dimensão é chamada eixo (***axis***).
- Um array é uma lista de axis e uma lista de tamanho dos axis é o que chamamos de **shape** do array.
 - Por exemplo, o shape da matrix acima é (3, 4) .
- O tamanho (***size***) de uma array é o número total de elementos, por exemplo, no array 2D acima = $3 * 4 = 12$.

```
1 # Criando e mostrando o array
2 a = np.zeros((3,4))
3 a
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
1 # Verificando o shape do array
2 a.shape
```

```
(3, 4)
```

```
1 # Verificando a quantidade de dimensões de um array
2 a.ndim
```

```
2
```

```
1 # Verificando a quantidade de elemntos no array
2 a.size
```

```
12
```


```
1 # ToDo : Criar um array de 3 dimensões, de shape (2,3,4) e mostrar o shape, quantidade de dimensões e o número de elemetos
2 arr = np.zeros((2, 3, 4))
3 arr.shape, arr.ndim, arr.size
```

 `((2, 3, 4), 3, 24)`

```
1 # ToDo : Criar um array de 3 dimensões mas trocando a função zeros por ones e mostrar o shape, quantidade de dimensões e o número de elementos
2 arr = np.ones((2, 3, 4))
3 arr.shape, arr.ndim, arr.size
```

 `((2, 3, 4), 3, 24)`


```
1 # ToDo : Criar um array de 3 dimensões mas trocando a função zeros por full e mostrar o shape, quantidade de dimensões e o número de elementos
2 arr = np.full((2, 3, 4), 5)
3 print(arr)
4 arr.shape, arr.ndim, arr.size
```



```
[[[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]

 [[5 5 5 5]
  [5 5 5 5]
  [5 5 5 5]]]
((2, 3, 4), 3, 24)
```

```
1 # ToDo : Criar um array de 3 dimensões mas trocando a função zeros por empty e mostrar o shape, quantidade de dimensões e o número de elementos
2 arr = np.empty((2, 3, 4))
3 print(arr)
4 arr.shape, arr.ndim, arr.size
```



```
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
((2, 3, 4), 3, 24)
```


ToDo: O que você pode dizer sobre cada uma das quatro funções que você usou?

Todas as funções têm a finalidade de criar um array com determinadas dimensões. Todavia, o `np.zeros` cria o array somente com valores

✓ O comando *np.arange* (5pt)

Você pode criar um array usando a função `arange`, similar a função `range` do Python.

```
1 # Criando um array
2 np.arange(1, 5)
```

 `array([1, 2, 3, 4])`

```
1 # Para criar com ponto flutuante
2 np.arange(1.0, 5.0)
```

```
↵ array([1., 2., 3., 4.])
```

```
1 # ToDo : crie um array com arange, variando de 1 a 5, com um passo de 0.5
2 np.arange(1, 5, 0.5)
```

```
↵ array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

✓ Os comandos *np.rand* e *np.randn* (5pt)

O *NumPy* tem várias funções para criação de números aleatórios. Estas funções são muito úteis para inicialização dos pesos das redes neurais. Por exemplo, abaixo criamos uma matrix (3, 4) inicializada com números em ponto flutuante (*floats*) e distribuição uniforme:

```
1 np.random.rand(3,4)
```

```
↵ array([[0.7602757 , 0.02111394, 0.32905191, 0.7960685 ],
         [0.50795559, 0.63346452, 0.74581834, 0.55521584],
         [0.20389059, 0.01919086, 0.43726752, 0.68728086]])
```

Abaixo um matriz inicializada com distribuição gaussiana ([normal distribution](#)) com média 0 e variância 1

```
1 np.random.randn(3,4)
```

```
↵ array([[ -0.16063135, -0.88306013, -0.0101034 , -0.70084827],
         [ 0.87521827,  0.06817194,  0.09125965, -1.63923991],
         [ 0.03520767, -0.42240487,  1.00899785, -0.5030477 ]])
```

```
1 # ToDo : crie um array aleatório com o shape (1, 2, 3, 4)
2 np.random.rand(1, 2, 3, 4)
```

```
↵ array([[[[0.64874423, 0.2436377 , 0.20717099, 0.94130833],
           [0.06044594, 0.23421054, 0.22104799, 0.83949689],
           [0.09506331, 0.28749473, 0.14611944, 0.33257559]],
          [[0.14678961, 0.36113759, 0.59564543, 0.93870563],
           [0.13541196, 0.00763982, 0.53572155, 0.68410892],
           [0.03403972, 0.60074604, 0.09140654, 0.8426105 ]]]])
```

✓ A biblioteca *Matplotlib* (5pt)

Vamos usar a biblioteca matplotlib (para mais detalhes veja o [tutorial de matplotlib](#)) para plotar dois arrays de tamanho 10.000, um inicializado com distribuição normal e o outro com uniforme

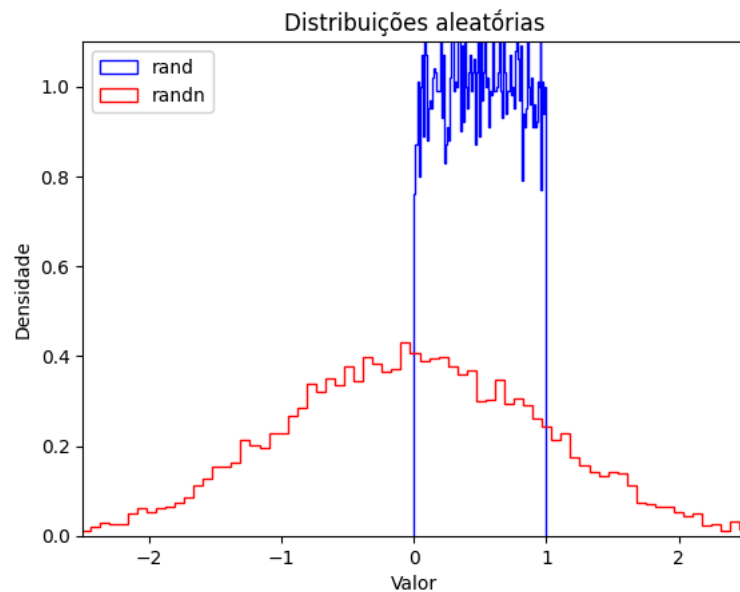
```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
```

Primeiro os dados que serão plotados precisam ser criados

```
1 array_a = np.random.rand(10_000, ) # ToDo : criar um array de shape (10.000,)  
2 array_b = np.random.randn(10_000, )# ToDo : criar um array de shape (10.000,)
```

Depois eles podem ser plotados

```
1 plt.hist(array_a, density=True, bins=100, histtype="step", color="blue", label="rand")  
2 plt.hist(array_b, density=True, bins=100, histtype="step", color="red", label="randn")  
3 plt.axis([-2.5, 2.5, 0, 1.1])  
4 plt.legend(loc = "upper left")  
5 plt.title("Distribuições aleatórias")  
6 plt.xlabel("Valor")  
7 plt.ylabel("Densidade")  
8 plt.show()
```



▼ Tipo de dados (*dtype*) (5pt)

Você pode ver qual o tipo de dado pelo atributo `dtype`. Verifique abaixo:

```
1 c = np.arange(1, 5)  
2 print(c.dtype, c)
```



```
int64 [1 2 3 4]
```

```
1 # ToDo: Crie um array aleatório de shape (2, 3, 4) e verifique o seu tipo
2 arr = np.random.rand(2, 3, 4)
3 print(arr.dtype)
```

↩ float64

Tipos disponíveis: int8, int16, int32, int64, uint8|16|32|64, float16|32|64 e complex64|128. Veja a [documentação](#) para a lista completa.

▼ Atributo *itemsize* (5pt)

O atributo `itemsize` retorna o tamanho em bytes

```
1 e = np.arange(1, 5, dtype=np.complex64)
2 e.itemsize
```

↩ 8

```
1 # Na memória, um array é armazenado de forma contígua
2 f = np.array([[1,2],[1000, 2000]], dtype=np.int32)
3 f.data
```

↩ <memory at 0x7c377c8b0e10>

```
1 # ToDo: Crie arrays de shape (2, 2) dos tipos int8, int64, float16, float64, complex64 e complex128
2
3 types = [ np.int8, np.int64, np.float16, np.float64, np.complex64, np.complex128]
4
5 for t in types:
6     arr = np.empty((2,2)).astype(t)
7     print(arr.dtype, arr.itemsize, arr.data)
```

↩ int8 1 <memory at 0x7a22c0ae1970>
int64 8 <memory at 0x7a22c0ae1970>
float16 2 <memory at 0x7a22c0ae1970>
float64 8 <memory at 0x7a22c0ae1970>
complex64 8 <memory at 0x7a22c0ae1970>
complex128 16 <memory at 0x7a22c0ae1970>

ToDo: O que você pode dizer sobre esses arrays criados?

Os arrays são similares, mas com os tipos diferentes. Ademais, os tamanhos deles em bytes também serão variados de acordo com o t

▼ *Reshaping* (5pt)

Alterar o shape de uma array é muito fácil com NumPy e muito útil para adequação das matrizes para métodos de machine learning. Contudo, o tamanho (size) não pode ser alterado.

```
1 # O número de dimensões também é chamado de rank
2 g = np.arange(24)
3 print(g)
4 print("Rank:", g.ndim)
```

```
→ [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
Rank: 1
```

```
1 g.shape = (6, 4)
2 print(g)
3 print("Rank:", g.ndim)
```

```
→ [[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]
   [12 13 14 15]
   [16 17 18 19]
   [20 21 22 23]]
Rank: 2
```

```
1 g.shape = (2, 3, 4)
2 print(g)
3 print("Rank:", g.ndim)
```

```
→ [[[ 0  1  2  3]
     [ 4  5  6  7]
     [ 8  9 10 11]]

    [[12 13 14 15]
     [16 17 18 19]
     [20 21 22 23]]]
Rank: 3
```

Mudando o formato do dado (*reshape*)

```
1 g2 = g.reshape(4,6)
2 print(g2)
3 print("Rank:", g2.ndim)
```

```
→ [[ 0  1  2  3  4  5]
   [ 6  7  8  9 10 11]
   [12 13 14 15 16 17]
   [18 19 20 21 22 23]]
Rank: 2
```

```
1 # Pode-se alterar diretamente um item da matriz, pelo índice
2 g2[1, 2] = 999
3 g2
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7, 999,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

Repare que o objeto 'g' foi modificado também!

1 g

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [999,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
```

Todas as operações aritméticas comuns podem ser feitas com o *ndarray*

```
1 a = np.array([14, 23, 32, 41])
2 arr = np.array([5, 4, 3, 2])
3 print("a + b =", a + arr)
4 print("a - b =", a - arr)
5 print("a * b =", a * arr)
6 print("a / b =", a / arr)
7 print("a // b =", a // arr)
8 print("a % b =", a % arr)
9 print("a ** b =", a ** arr)
```

```
a + b = [19 27 35 43]
a - b = [ 9 19 29 39]
a * b = [70 92 96 82]
a / b = [ 2.8      5.75      10.66666667 20.5      ]
a // b = [ 2  5 10 20]
a % b = [4 3 2 1]
a ** b = [537824 279841 32768 1681]
```

Repare que a multiplicação acima **NÃO** é uma multiplicação de matrizes

Arrays devem ter o mesmo shape, caso contrário, NumPy vai aplicar a regra de *broadcasting* (Ver seção 2.1.3 do [livro texto](#)). Pesquise sobre a operação e broadcasting do NumPy e explique com suas palavras, abaixo:

ToDo: Explique o conceito de *broadcasting*.

Operações em numpy, geralmente, são feitas de elemento a elemento, o que faria com que os itens devessem ter o mesmo shape. Mas c

✓ Iteração e Concatenação de arrays de *NumPy* (5pt)

Repare que você pode iterar pelos `ndarrays`, e que ela é feita pelos *axis*.

```
1 c = np.arange(24).reshape(2, 3, 4) # Um array 3D (coposto de duas matrizes de 3x4)
2 c
```

```
↵ array([[[ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
          [16, 17, 18, 19],
          [20, 21, 22, 23]]])
```

```
1 for m in c:
2     print("Item:")
3     print(m)
```

```
↵ Item:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Item:
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

```
1 for i in range(len(c)): # Observe que len(c) == c.shape[0]
2     print("Item:")
3     print(c[i])
```

```
↵ Item:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Item:
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

```
1 # Para iterar por todos os elementos
2 for i in c.flat:
3     print("Item:", i)
```

```
↵ Item: 0
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
Item: 6
Item: 7
Item: 8
Item: 9
Item: 10
Item: 11
Item: 12
```

```

Item: 13
Item: 14
Item: 15
Item: 16
Item: 17
Item: 18
Item: 19
Item: 20
Item: 21
Item: 22
Item: 23

```

Também é possível concatenar ndarrays, e isso pode ser feito em um eixo específico.

```

1 # Pode-se concatenar arrays pelos axis
2 q1 = np.full((3,4), 1.0)
3
4 q2 = np.full((4,4), 2.0)
5
6 q3 = np.full((3,4), 3.0)
7
8 q = np.concatenate((q1, q2, q3), axis=0)
9
10 q

```

```

↩ array([[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [2., 2., 2., 2.],
         [2., 2., 2., 2.],
         [2., 2., 2., 2.],
         [2., 2., 2., 2.],
         [2., 2., 2., 2.],
         [3., 3., 3., 3.],
         [3., 3., 3., 3.],
         [3., 3., 3., 3.]])

```

```

1 # ToDo: imprima o shape resultante da concatenação dos arrays de shape a = (2, 3, 4) e b = (2, 3, 4) em cada eixo separadamente
2 a = np.zeros((2, 3, 4))
3 b = np.ones((2, 3, 4))
4
5 print("Axis 0:", np.concatenate((a, b), axis=0))
6 print("Axis 1:", np.concatenate((a, b), axis=1))
7 print("Axis 2:", np.concatenate((a, b), axis=2))
8

```

```

↩ Axis 0: [[[0. 0. 0. 0.]
          [0. 0. 0. 0.]
          [0. 0. 0. 0.]]

          [[0. 0. 0. 0.]
          [0. 0. 0. 0.]
          [0. 0. 0. 0.]]

          [[1. 1. 1. 1.]
          [1. 1. 1. 1.]
          [1. 1. 1. 1.]]]

```

```

[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Axis 1: [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Axis 2: [[0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]]

[[0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]
 [0. 0. 0. 0. 1. 1. 1. 1.]]

```

✓ Transposta (5pt)

```

1 m1 = np.arange(10).reshape(2,5)
2 m1

```

```

↔ array([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]])

```

```

1 # ToDo : imprima a matriz transposta de m1
2 print(m1.T)

```

```

↔ [[0 5]
    [1 6]
    [2 7]
    [3 8]
    [4 9]]

```

✓ Produto de matrizes (5pt)

```

1 n1 = np.arange(10).reshape(2, 5)
2 n1

```

```

↔ array([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]])

```

```

1 n2 = np.arange(15).reshape(5, 3)
2 n2

```

```
↵ array([[ 0,  1,  2],
         [ 3,  4,  5],
         [ 6,  7,  8],
         [ 9, 10, 11],
         [12, 13, 14]])
```

```
1 n1.dot(n2)
```

```
↵ array([[ 90, 100, 110],
         [240, 275, 310]])
```

```
1 # ToDo: Crie um array de 1 a 25 com shape (5, 5) e faça a multiplicação por uma matriz de zeros de (5, 1).
```

```
2 a = np.arange(1, 26).reshape((5, 5))
```

```
3 z = np.zeros((5, 1))
```

```
4
```

```
5 np.dot(a, z)
```

```
↵ array([[0.],
         [0.],
         [0.],
         [0.],
         [0.]])
```

▼ Matriz Inversa (5pt)

```
1 import numpy.linalg as linalg
```

```
2
```

```
3 m3 = np.array([[1,2,3],[5,7,11],[21,29,31]])
```

```
4
```

```
5 m3
```

```
↵ array([[ 1,  2,  3],
         [ 5,  7, 11],
         [21, 29, 31]])
```

```
1 linalg.inv(m3)
```

```
↵ array([[-2.31818182,  0.56818182,  0.02272727],
         [ 1.72727273, -0.72727273,  0.09090909],
         [-0.04545455,  0.29545455, -0.06818182]])
```

ToDo: O que a função `linalg.inv` faz?

A função ``linalg.inv`` calcula a matriz inversa da matriz quadrada passada. Caso a matriz não seja quadrada, ocorrerá o erro `LinAl`

▼ Matriz identidade (5pt)

```
1 m3.dot(linalg.inv(m3))

array([[ 1.00000000e+00, -1.66533454e-16,  0.00000000e+00],
       [ 6.31439345e-16,  1.00000000e+00, -1.38777878e-16],
       [ 5.21110932e-15, -2.38697950e-15,  1.00000000e+00]])
```

```
1 # ToDo: Crie uma matriz identidade de tamanho (5, 5)
2 np.identity(5)
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

✓ Exercícios (15pt)

Para os próximos exercícios, use o numpy.

Questão 1: Escreva uma função recursiva que calcule o determinante de uma matriz $n \times n$ usando o teorema de Laplace.

```
1 from math import isclose

1 # Seu código aqui
2 def determinante(matriz):
3     n = matriz.shape[0]
4
5     if n == 2:
6         return matriz[0][0] * matriz[1][1] - (matriz[0][1] * matriz[1][0])
7
8     sum = 0
9     # A primeira coluna será fixa e serão removidas as linhas
10    for i in range(n):
11        submatriz_1 = matriz[0:i, 1:n]
12        submatriz_2 = matriz[i+1:n, 1:n]
13        submatriz = np.concatenate((submatriz_1, submatriz_2))
14        cofator = ((-1)**i) * matriz[i][0] * determinante(submatriz)
15        sum += cofator
16    return sum
17
18 m = np.array([
19     [4,5,-3,0],
20     [2,-1,3,1],
21     [1,-3,2,1],
22     [0,2,-2,5]
23 ])
24
25 det = determinante(m)
26 print(det)
27 isclose(linalg.det(m), det)
```

↗ 210
True

Questão 2: Escreva um programa que calcule a solução de um sistema de equações lineares por meio da regra de Cramer. Seu programa deve iniciar lendo o número de equações e variáveis, e, logo após, ler as matrizes de entrada do teclado coeficiente a coeficiente. Para o cálculo dos determinantes, você pode utilizar a função escrita no exercício 1, ou a função `det` do pacote `numpy`.

```
1 # Seu código aqui
2 def leitura():
3     num_equacoes = int(input("Número de equações: "))
4     num_variaveis = int(input("Número de variáveis: "))
5
6     if num_equacoes != num_variaveis:
7         raise ValueError("O número de equações deve ser igual ao de variáveis")
8
9     # Sistema do tipo Ax=B
10    A = np.empty((num_equacoes, num_variaveis))
11    B = np.empty((num_equacoes,1))
12
13    for i in range(A.shape[0]):
14        for j in range(A.shape[1]):
15            A[i][j] = float(input(f"Coeficiente [{i+1}, {j+1}]: "))
16
17    for i in range(B.shape[0]):
18        B[i][0] = float(input(f"Termo B {i+1}: "))
19
20    matriz_final = np.concatenate((A, B), axis = 1)
21    return matriz_final
22
23 def cramer(matriz: np.array):
24     A = matriz.T[:-1].T
25     B = matriz.T[-1].T
26
27     det_A = determinante(A)
28     resultados = []
29
30     for i in range(matriz.shape[0]):
31         A_copia = A.T.copy()
32         A_copia[i] = B
33         det_temp = determinante(A_copia.T) / det_A
34         resultados.append(det_temp)
35
36     return resultados
37
38 def sistema_pelo_metodo_de_cramer():
39     matriz = leitura()
40     print(cramer(matriz))
41
42 sistema_pelo_metodo_de_cramer()
```

↗ Número de equações: 2
Número de variáveis: 2
Coeficiente [1, 1]: 1
Coeficiente [1, 2]: 2

```

Coeficiente [2, 1]: 3
Coeficiente [2, 2]: -5
Termo B 1: 5
Termo B 2: 4
[3.0, 1.0]

```

Questão 3: Implemente uma função que resolva sistemas de equações lineares através do método de eliminação de Gauss. Rode a função para algum exemplo, e compare com a solução obtida com o código da questão 2. Meça o tempo de execução para verificar qual algoritmo executa mais rápido.

```

1 # Seu código aqui
2 def eliminacao_gauss(matriz: np.array):
3     if determinante(matriz) == 0:
4         raise ValueError ("Determinante zero implica em infinitas soluções, logo a matriz é inválida")
5
6     n = matriz.shape[0]
7     matriz_copy = matriz.copy()
8
9     for i in range(n):
10         pivo = matriz_copy[i][i]
11         matriz_copy[i] = matriz_copy[i] / pivo # Dividindo a linha atual pelo pivo
12
13         # aplicando Linha[i] -m * Linha[j]
14         for linha in range(i+1, n):
15             m = matriz_copy[linha][i] / matriz_copy[i][i]
16             matriz_copy[linha] = matriz_copy[linha] - m * matriz_copy[i]
17
18     # Apos obter a matriz triangular superior, aplicamos o metodo direto indo da ultima variavel ate a primeira
19     # para obter os valores de cada termo
20     Xs = []
21     xn = matriz_copy[-1][-1] / matriz_copy[-1][-2]
22     Xs.append(xn)
23
24     for i in reversed(range(n - 1)):
25         somatorio = 0
26         for j in range(i+1, n):
27             somatorio += matriz_copy[i][j] * Xs[-(n-j)]
28         xi = (matriz_copy[i][-1] - somatorio) / matriz_copy[i][i]
29         Xs.insert(0, xi)
30
31     return Xs

```

```

1 import time
2 matriz = np.array([
3     [2, 3, -4, 0, 8],
4     [1, -1, 0, -1, 0],
5     [0, 1, 1, 1, 2],
6     [1, 2, 2, 4, 8]
7 ]).astype(np.float64)
8
9 tempos = np.empty((2, 10), dtype=np.float64)
10 resultados = np.empty((2, matriz.shape[0]), dtype=np.float64)
11 # Executando ambos os códigos 10 vezes para possuir uma média dos tempos de execução
12 for i in range(10):

```

```
13 for index, metodo in enumerate([cramer, eliminacao_gauss]):
14     inicio = time.time()
15     resultado = metodo(matriz)
16     final = time.time()
17
18     tempos[index][i] = final - inicio
19     if i == 9:
20         resultados[index] = resultado
21
22 tempos = np.average(tempos, axis = 1)
23 print(f"Cramer: {tempos[0]}s")
24 print(f"Gauss : {tempos[1]}s")
25 print("Os valores de tempo são uma média de 10 execuções de cada algoritmo para a mesma matriz\n")
26
27 melhor = "Gauss"
28 pior   = "Cramer"
29 if tempos[0] < tempos[1]:
30     melhor = "Cramer"
31     pior = "Gauss"
32 print(f"O método de {melhor} foi mais rápido do que o método de {pior}\n")
33
34 print(f"Resultados Cramer: {resultados[0]}")
35 print(f"Resultados Gauss : {resultados[1]}")
36
37 resultados_semelhantes = True
38 for (c, g) in zip(resultados[0], resultados[1]):
39     if not isclose(c, g):
40         resultados_semelhantes = False
41
42 if resultados_semelhantes:
43     print("Os resultados obtidos de ambos os métodos são equivalentes")
44 else:
45     print("Os resultados dos métodos divergem")
```



```
Cramer: 0.0008433341979980468s
Gauss : 0.00020754337310791016s
Os valores de tempo são uma média de 10 execuções de cada algoritmo para a mesma matriz
```

O método de Gauss foi mais rápido do que o método de Cramer

```
Resultados Cramer: [ 2.0952381  1.14285714 -0.0952381  0.95238095]
Resultados Gauss : [ 2.0952381  1.14285714 -0.0952381  0.95238095]
Os resultados obtidos de ambos os métodos são equivalentes
```