Exercício Computacional 1: Tomografia Computadorizada
MAP3122 - Quadrimestral 2021

Gabriel Pereira de Carvalho NUSP: 11257668

Fevereiro 2021

Contents

1 Exercício 1

1.1 Eliminação de Gauss Jordan

O primeiro passo desse exercício computacional foi escrever um método para resolver sistemas lineares em Python. O algoritmo escolhido foi a Eliminação de Gauss Jordan com Condensação Pivotal para evitar preocupações com questões de convergência que surgem na escolha de um método iterativo.

Como a implementação requer o uso de ponto flutuante, para comparar o pivô com zero não basta simplesmente compará-los porque erros de aproximação fazer com que um sistema impossível não seja detectado. Para resolver esse problema, foi adotado que dois números são iguais se, e somente se, o valor absoluto de sua diferença é menor do que $EPS = 10^{-9}$.

Segue o código utilizado na realização desse trabalho, cuidadosamente comentado.

```
def ElimGauss(A, b): #resolve sistema de m equacoes e m variaveis

    #dimensao da matriz quadrada A
    m = np.size(A, 0)
    #criar vetor coluna das solucoes
    x = np.empty(m)

    #Vamos implementar com Condensacao Pivotal
    for coluna in range(0, m):

        #selecionando pivo
        sel = coluna
        for i in range(coluna + 1, m):
            if(abs(A[i][coluna]) > abs(A[sel][coluna])):
                sel = i

        #vamos trocar as linhas
        A[[coluna, sel]] = A[[sel, coluna]]
        b[[coluna,sel]] = b[[sel, coluna]]

        #se pivo eh nulo, sistema nao tem solucao
        if(abs(A[coluna][coluna]) < EPS):
            return False

        #hora de eliminar
        for linha in range(coluna + 1, m):
            multiplicador = A[linha][coluna]/A[coluna][coluna]
            A[linha] -= multiplicador*A[coluna]
            b[linha] -= multiplicador*b[coluna]

    linha = m - 1 #retrosubstituicao: comecamos na ultima linha e subimos
    while linha > -1:
        x[linha] = b[linha]
        for anterior in range(linha+1, m):
            x[linha] -= A[linha][anterior]*x[anterior]
        x[linha] /= A[linha][linha]
        linha -= 1
```

```
    return x
```

1.2 Construindo a matriz de coeficientes

Nossa tarefa é resolver o sistema $(A^TA + \delta I_{n^2})f_\delta = A^Tp$.

Para isso, precisamos construir a matriz A para valores de n genéricos.

É possível observar que a matriz pode ser dividida em blocos nxn. De fato, vemos a matriz Identidade I_n repetida na matriz A. Com essa observação podemos usar o produto de Kronecker para construir cada um desses blocos de maneira eficiente.

```
def constroeA(n):#constroe matriz A

    #na metade superior da matriz, vemos a identidade repetidamente
    auxiliar1 = np.zeros((2, n))
    for i in range(0, n):
        auxiliar1[0][i] = 1

    #na metada inferior, queremos produzir fileiras de uns
    auxiliar2 = np.ones(n)
    auxiliar3 = np.zeros((2*n, n))
    for i in range(n, 2*n):
        auxiliar3[i][i-n] = 1

    #somamos a decomposicao de cada metade para obter o todo
    A = np.kron(auxiliar1, np.identity(n)) + np.kron(auxiliar3, auxiliar2)

    return A
```

1.3 Regularização Exercício 1

O método da regularização consiste em encontrar uma solução aproximada suave compatível com os dados da observação. Note que ao buscar uma solução suave, estamos acrescentando uma informação não dada em nosso problema inverso, que por definição é mal-posto. De certa forma, se trata de uma condição física. Dentro do contexto da tomografia, não se esperam oscilações entre células vizinhas da imagem de raios-X, especialmente conforme aumentamos o valor de n e consequentemente a definição da imagem.

Pela Regra de Cramer, sabemos que o sistema é possível e determinado se, e somente se, o determinante da matriz de coeficientes é não nulo. Portanto, para que a solução do sistema normal esteja bem definida é necessário ter determinante não nulo e, para evitar erros de aproximação, distante do zero.

Mas note que, se delta for muito alto, vetores coluna com norma euclidiana pequena que são distantes da imagem original podem fornecer um valor menor de $E(f) = ||Af - p||^2 + \delta||f||^2$. Portanto, o ideal é balancear a contribuição dos dois termos.

Na Tabela 1, apresentamos os determinantes calculados para cada valor de n e δfornecidos nos dados do exercício. Vale observar que valores muito pequenos de determinantes são praticamente indestinguíveis de zero, porque ao trabalhar com valores tão pequenos deve ser considerada a influência de erros de arrendondamento no valor obtido.

n	$\delta = 0$	$\delta = 10^{-3}$	$\delta = 10^{-2}$	$\delta = 10^{-1}$
5	$0,0$	$3,91 * 10^{-42}$	$3,97 * 10^{-26}$	$4.62 * 10^{-10}$
10	$0,0$	$2 * 10^{-224}$	$2,04 * 10^{-143}$	$2,40 * 10^{-62}$
30	$0,0$	$0,0$	$0,0$	$0,0$

Table 1: Determinante das matriz de coeficientes do sistema normal

1.4 Implementação do exercício 1

Utilizando os dois métodos descritos nas seções anteriores, podemos escrever uma solução concisa para o exercício 1. Segue o código cuidadosamente comentado

```
def exercicio1(p, n, delta):

    A = constroeA(n)
    At = np.copy(A)
    At = np.transpose(At)

    #vamos obter sistema normal a partir dos dados
    A_normal = np.add(np.matmul(At, A), np.multiply(delta, np.identity(n*n)))
    p_normal = np.matmul(At, p)

    #vamos calcular os determinates pedidos aqui
    det = np.linalg.det(A_normal)
    print(n, delta, det)


    fsol = ElimGauss(A_normal, p_normal)

    #foi encontrado pivo nulo durante eliminacao
    if(type(fsol) == 'bool'):
        print("SISTEMA IMPOSSIVEL")

    #transformar vetor coluna em matriz no formato dado
    fsol = np.reshape(fsol, (n,n))
    fsol = np.transpose(fsol)

    #hora de gerar imagem da solucao
    print(fsol)
    mp.pyplot.matshow(fsol)
    mp.pyplot.show()

#vamos importar o arquivo p1.npy
print("Por digite o caminho do arquivo p1.npy")
caminho_p = input()
p = np.load(caminho_p)

#obter parametro n
n = round(np.size(p, 0)/2)

#vamos gerar solucao do problema para cada delta
delta = np.array([10**-3, 10**-2, 10**-1])
for i in range(0, np.size(delta,0)):
    exercicio1(p, n, delta[i])

#Agora vamos exibir a imagem original
print("Por favor digite o caminho do arquivo im.png")
caminho_f = input()
f = mp.pyplot.imread(caminho_f)
```

```
print(f)
mp.pyplot.matshow(f)
mp.pyplot.show()
```

1.5 Imagens Obtidas Exercício 1

Apesar de os valores de f mudarem conforme alteramos o valor de δ, as imagens obtidas para um mesmo valor de n foram as mesmas.

1.5.1 Imagens para n=5

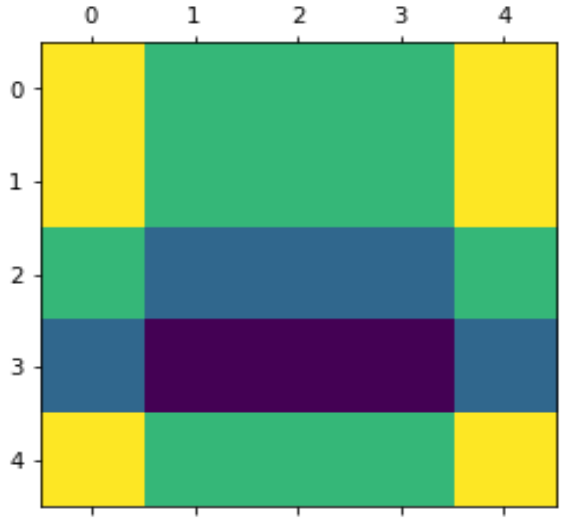

Figure 1: Solução gerada para n=5 e $\delta = 10^{-3}$

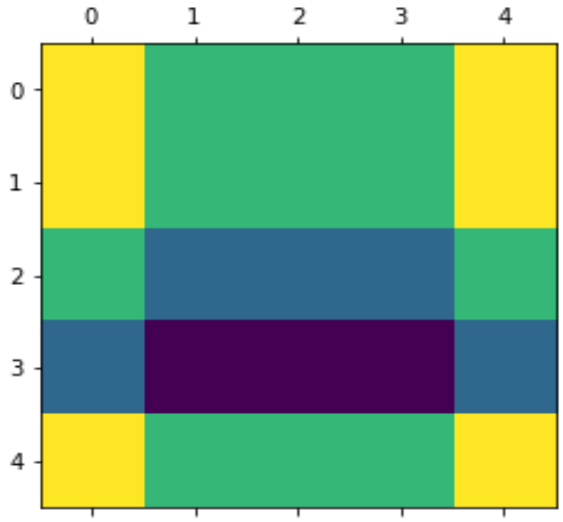

Figure 2: Solução gerada para n=5 e $\delta = 10^{-2}$

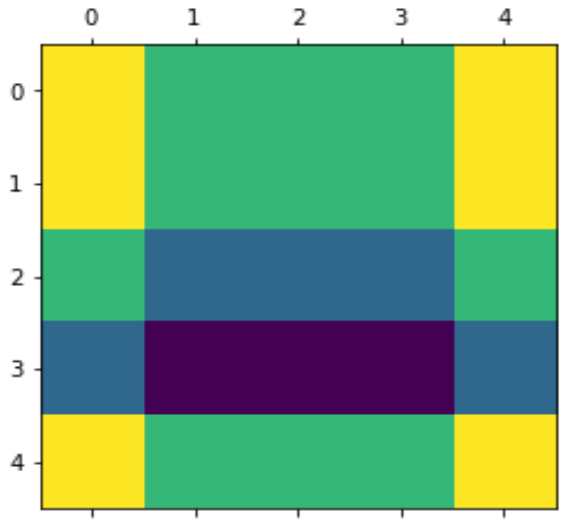

Figure 3: Solução gerada para n=5 e $\delta = 10^{-1}$

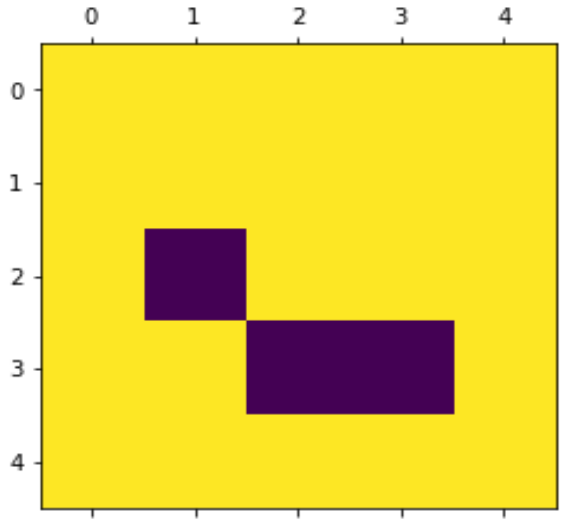

Figure 4: Imagem original para n=5

1.5.2 Imagens para n=10

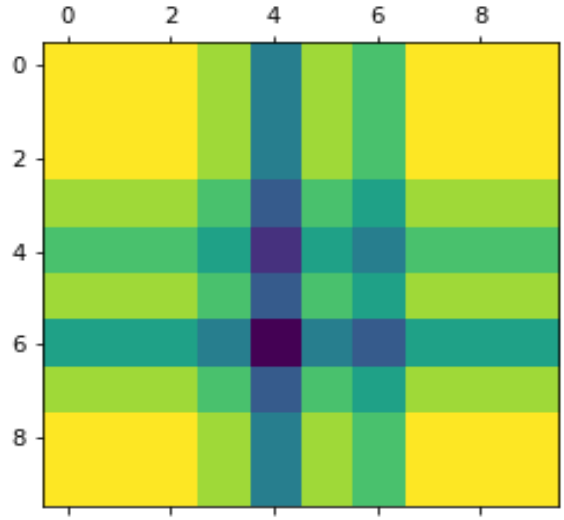

Figure 5: Solução gerada para n=10 e $\delta = 10^{-3}$

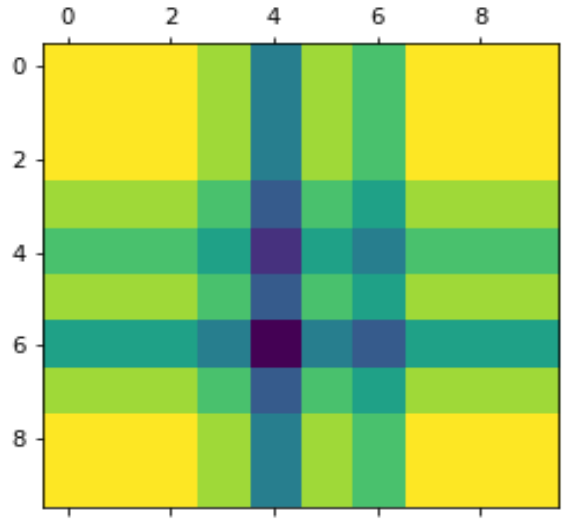

Figure 6: Solução gerada para n=10 e $\delta = 10^{-2}$

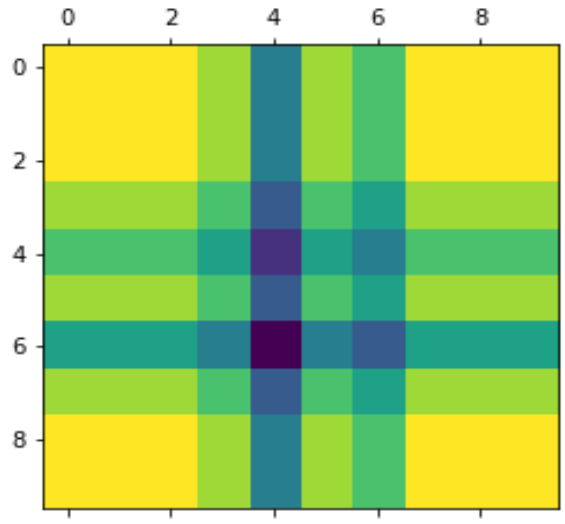

Figure 7: Solução gerada para n=10 e $\delta = 10^{-1}$

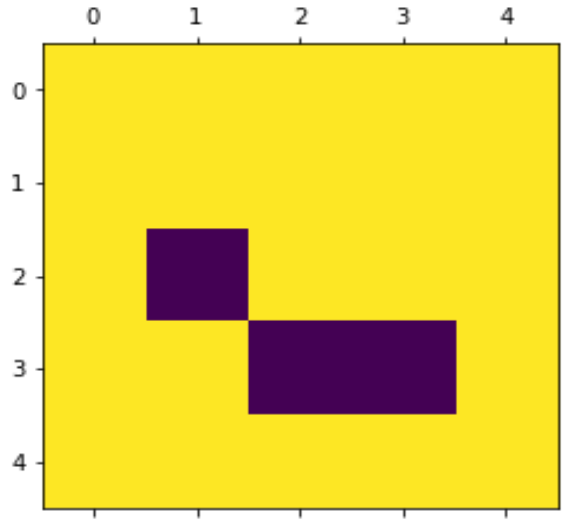

Figure 8: Imagem original para n=10

1.5.3 Imagens para n=30

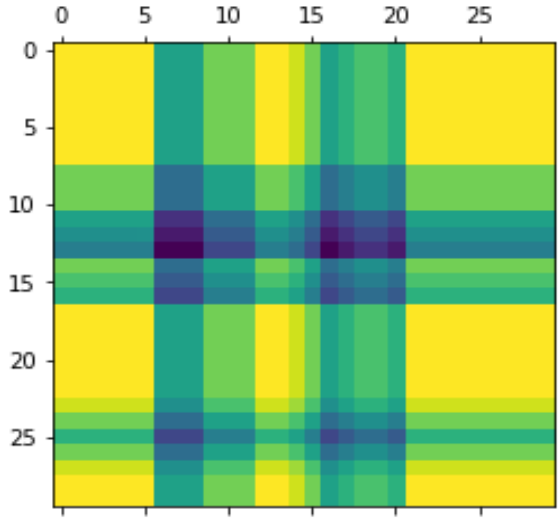

Figure 9: Solução gerada para n=30 e $\delta = 10^{-3}$

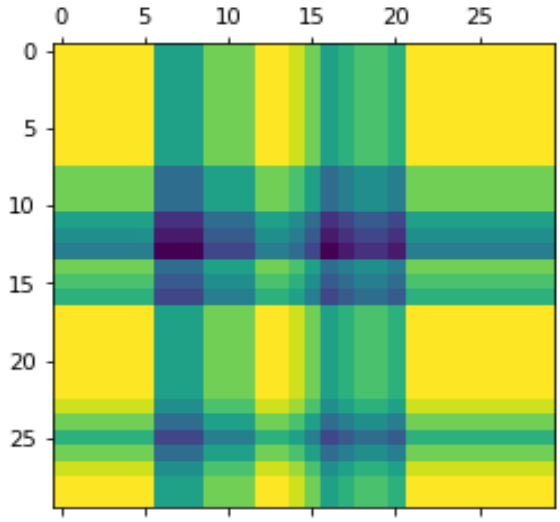

Figure 10: Solução gerada para n=30 e $\delta = 10^{-2}$

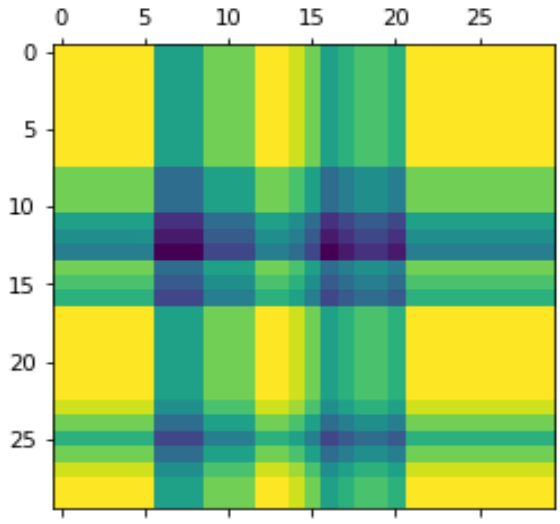

Figure 11: Solução gerada para n=30 e $\delta = 10^{-1}$

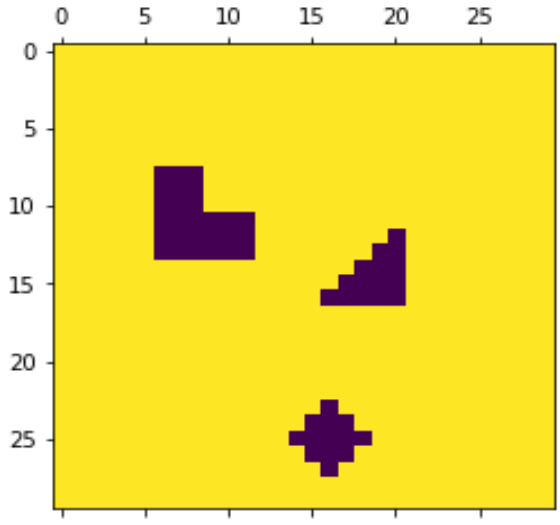

Figure 12: Imagem original para n=30

2 Exercício 2

O desafio do exercício 2 é a implementação do método de construção da matriz A, que cresceu em complexidade ao acrescentarmos projeções diagonais.

Note que cada f_i contribui para exatamente uma linha, uma coluna, uma diagonal descrescente e uma diagonal crescente. Portanto, podemos buscar padrões para cada uma dessas categorias e analisar separadamente cada f_i.

Para o caso das diagonais, observe que os elementos de cada diagonal formam uma progrssão aritmética, portanto basta saber resolver os casos base e escrever uma função recursiva para determinar a diagonal de qualquer elemento.

Essa é uma abordagem incremental para a construção da matriz.

Segue o código cuidadosamente comentado.

```
def diag_decr(i, n):
    if(i%n == 0): #atingimos borda superior
        return n - 1 + i/n
    elif i < n :#atingimos borda lateral
        return n-i-1
    else:
        return diag_decr(i - n - 1, n) #chamamos elemento anterior da progress o

def diag_cres(i, n):
    if i<n :#atingimos borda lateral
        return i
    elif i%n==n-1:#atingimos borda inferior
        return n-1 + i/n
    else:
        return diag_cres(i - n + 1, n) #chamamos elemento anterior da progress o

def constroeA(n):#constroe matriz A
    n = int(n)
    A = np.zeros((6*n-2,n*n))
    #vamos pensar na contribuicao de cada f_i
    for i in range(0, n*n):

        #vamos determinar qual linha f_i contribui
        linha =  int(i%n)
        A[linha][i] += 1

        #vamos determinar qual coluna f_i contribui
        coluna = int(i/n)
        A[n + coluna][i] += 1

        #vamos determinar qual diagonal descrescente f_i contribui
        A[int(2*n + diag_decr(i,n))][i] += 1

        #vamos determinar qual diagonal crescente f_i contribui
        A[int(4*n-1+diag_cres(i,n))][i] += 1

    return A
```

2.1 Regularização Exercício 2

Começamos apresentando a tabela com valores do determinante.

n	$\delta = 0$	$\delta = 10^{-3}$	$\delta = 10^{-2}$	$\delta = 10^{-1}$
5	$0,0$	$0,873$	$9,36 * 10^{+3}$	$1,84 * 10^{+8}$
10	$0,0$	$1,26 * 10^{-107}$	$1,4 * 10^{-58}$	$4,04 * 10^{-9}$
30	$0,0$	$0,0$	$0,0$	$0,0$

Table 2: Determinante das matriz de coeficientes do sistema normal

Vale observar que para um mesmo par (n, δ) obtivemos valores maiores de determinante agora. Isso ocorre porque temos mais condições sobre a matriz f agora, portanto o sistema está melhor determinado.

2.2 Imagens Obtidas Exercício 2

2.2.1 Imagens para n=5

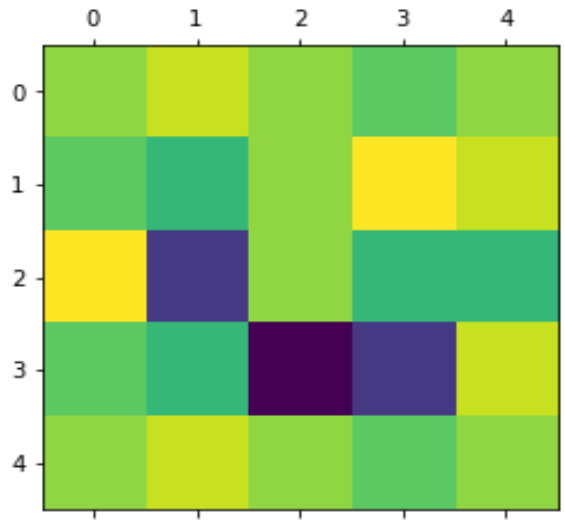

Figure 13: Solução gerada para n=5 e $\delta = 10^{-3}$

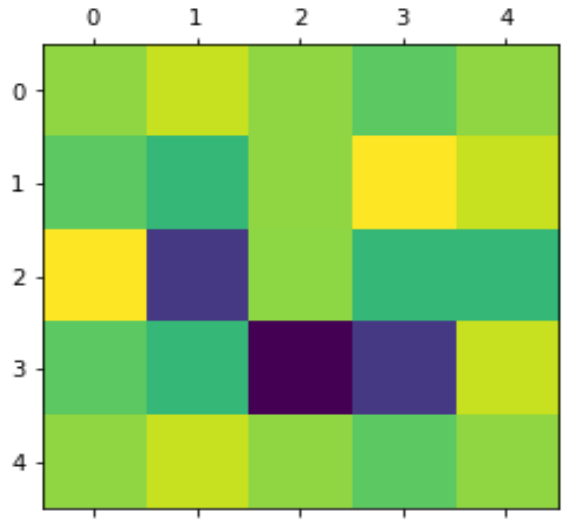

Figure 14: Solução gerada para n=5 e $\delta = 10^{-2}$

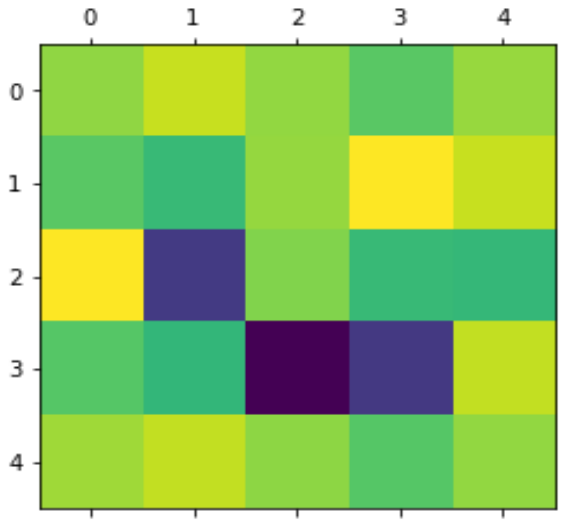

Figure 15: Solução gerada para n=5 e $\delta = 10^{-1}$

Uma grande diferença entre o exercício 1 e o exercício 2 é que agora, para valores diferentes de δ é possível observar pequenas diferenças nas imagens mesmo que para um mesmo valor de δ. Em particular observe o canto inferior esquerdo nesse conjunto de imagens.

2.2.2 Imagens para n=10

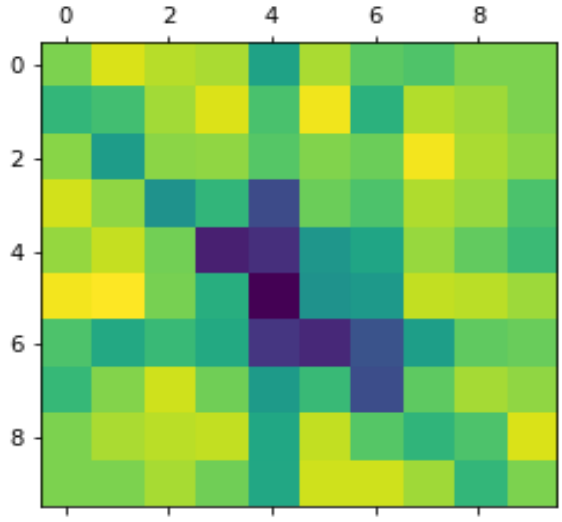

Figure 16: Solução gerada para n=10 e $\delta = 10^{-3}$

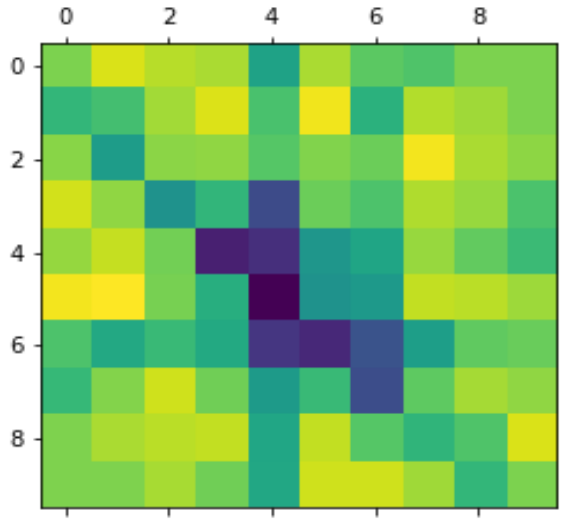

Figure 17: Solução gerada para n=10 e $\delta = 10^{-2}$

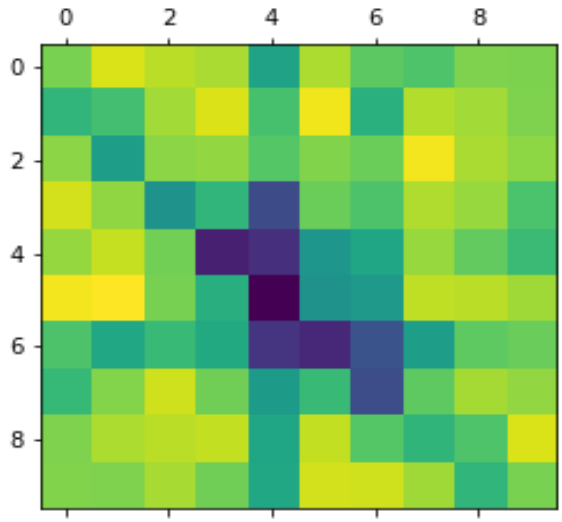

Figure 18: Solução gerada para n=10 e $\delta = 10^{-1}$

2.2.3 Imagens para n=30

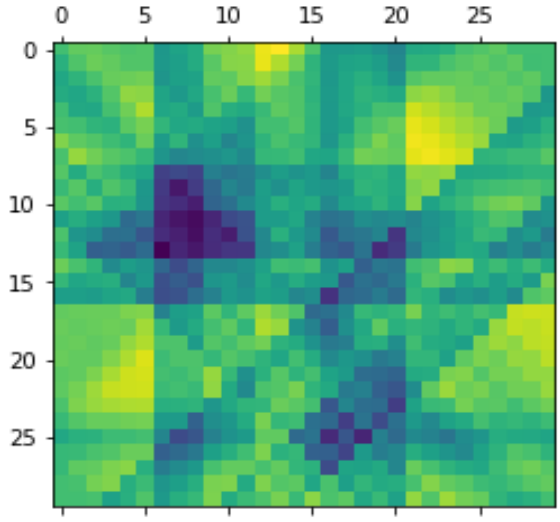

Figure 19: Solução gerada para n=30 e $\delta = 10^{-3}$

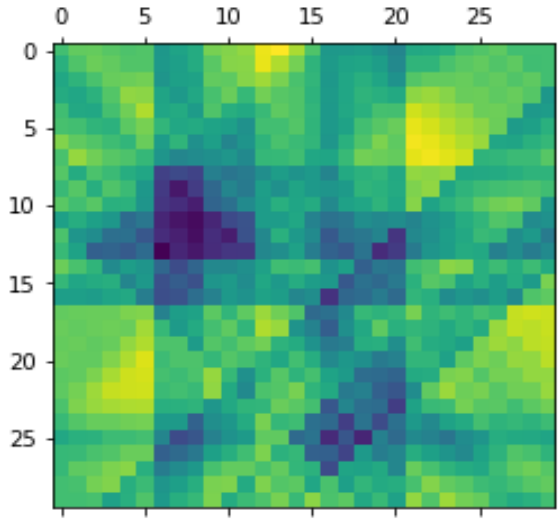

Figure 20: Solução gerada para n=30 e $\delta = 10^{-2}$

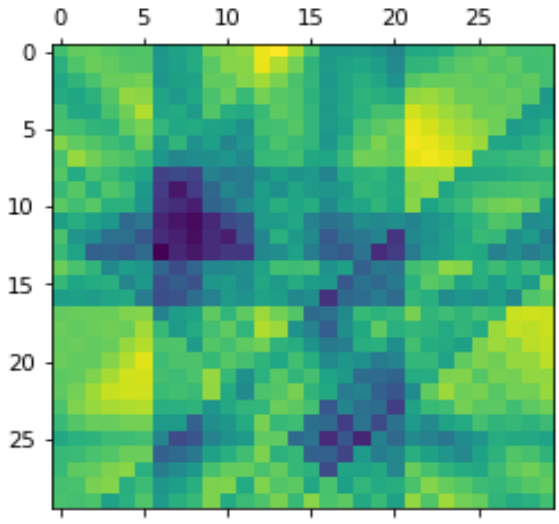

Figure 21: Solução gerada para n=30 e $\delta = 10^{-1}$

2.3 Cálculo do Erro

Para estimar a qualidade da aproximação da imagem original estamos interessados em calcular o erro relativo

$$L^2 = 100 * \frac{\sqrt{\sum_{j=1}^{n^2}(f_j - f_j^*)^2}}{\sqrt{\sum_{j=1}^{n^2} f_j^{*2}}}$$

Para efeito de comparação, e para entender melhor os resultados obtidos nessa seção, foi calculado também o erro relativo para as soluções obtidas no exercício 1.

Uma observação importante é que para comparar a imagem original com o vetor coluna obtido através da Eliminação Gaussiana, foi necessário rearranjar os elementos da matriz. Devido a ordem dos elementos é necessário tomar a transposta antes de aplicar a função rearrange.

```
f = mp.pyplot.imread(caminho_f)
f = np.transpose(f)
f = np.reshape(f, (n*n))
```

O código utilizado para cálculo do erro por outro lado é bastante direto.

```
numerador = 0
denominador = 0
for i in range(0, np.size(f, 0)):
    numerador += (f[i] - fsol[i])**2
    denominador += f[i]**2
numerador = numerador**0.5 #tiramos a raiz
denominador = denominador**0.5
erro = 100*numerador/denominador
print("Erro para ", delta, "eh ", erro)
```

Seguem as duas tabelas com os valores de erro relativo calculados.

n	$\delta = 10^{-3}$	$\delta = 10^{-2}$	$\delta = 10^{-1}$
5	28,28	28,28	28,30
10	22,94	22,94	22,94
30	21,91	21,91	21,91

Table 3: Cálculo do Erro Relativo para Exercício 1

n	$\delta = 10^{-3}$	$\delta = 10^{-2}$	$\delta = 10^{-1}$
5	$13,48$	$13,49$	$13,61$
10	$16,29$	$16,29$	$16,30$
30	$19,92$	$19,92$	$19,92$

Table 4: Cálculo do Erro Relativo para Exercício 2

Como era esperado, acrescentar informação relativa às diagonais reduziu bastante o erro relativo das soluções obtidas no Exercício 2. Como as soluções para um mesmo n são muito próximas, vemos que os erros também são muito próximos.

3 Conclusões

É interessante observar que a qualidade das soluções para o problema indireto aumentou drasticamente conforme o número de informações restringindo o problema também aumentou.

Começamos com um sistema indeterminado, onde o universo de soluções era infinito, portanto era impossível estabelecer uma bijeção entre os dados nos arquivos .npy e uma imagem única.

Formulando um problema de mínimos quadrados, foi imposta uma restrição sobre a norma euclidiana em duas direções. Obtivemos uma solução suave ao longo dos eixos x, y e próxima da imagem original.

No Exercício 2, acrescentamos ainda mais informação ao problema com as diagonais. Agora nossa restrição sobre a norma euclidiana passou a atuar em 4 direções. Obtivemos um sistema maior e mais complexo, e como consequência o tempo de computação aumentou. No entanto, a resolução das imagens também aumentou drasticamente.

Foi muito interessante ver como o Método dos Mínimos Quadrados pode ser aplicado para obter aproximações excelentes para um problema inicialmente mal definido.

4 Referências Bibliográficas

1. RAMOS, Fernando Manuel; MARTINS, Marcos Nogueira; DIAS, Mauro da Silva; GONÇALVES, Odair Lelis; HELENE, Otaviano A.M. **Problemas Inversos em Física Experimental**.
2. VELHO, Haroldo Fraga de Campos. **Problemas Inversos: Conceitos Básicos e Aplicações**.
3. Kronecker Product: Wolfram MathWorld
4. NumPy Documentation - NumPy v1.20 Manual
5. towardsdatascience.com **The Ultimate Begginer's Guide to Numpy**
6. pythonforengineers.com **An Introduction to Numpy and Matplotlib**
7. kite.com **How to plot a 2d array with Matplotlib in Python**