

Projeto Filtros AdFG

Beatriz Evelbauer Simões

Nesse projeto, visamos comparar diferentes tipos de filtros com relação a diferentes ruídos em uma mesma imagem.

Os filtros propostos são o filtro do kernel da equação do calor e também o de regularização à Tikhonov, este sendo um caso particular de um filtro $ARMA(p, q)$.

Eles são definidos por:

$$\text{heat}(\lambda) = e^{-\tau\lambda}$$

onde τ é um parâmetro real a ser definido, e

$$\text{Tikhonov}(\lambda) = \frac{1}{1 + \gamma\lambda}$$

onde γ é um parâmetro real a ser escolhido.

Ambos os filtros tem implementação na biblioteca `PyGSP`, com a qual faremos a comparação de desempenho no final do trabalho.

Imports

```
In [ ]: pip install opencv-python
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: opencv-python in c:\users\beatriz\appdata\roaming\python\python311\site-packages (4.9.0.80)
Requirement already satisfied: numpy>=1.21.2 in c:\users\beatriz\appdata\roaming\python\python311\site-packages (from opencv-python) (1.26.4)
Note: you may need to restart the kernel to use updated packages.
```

```
In [ ]: # bibliotecas
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pygsp import graphs, filters, plotting
from PIL import Image, ImageFilter
import scipy.sparse as sp
import cv2 # para mudar o tamanho das fotos
```

```
In [ ]: # abrindo imagens
img_original = cv2.imread('cameraman_original.jpg')
img_ruido1 = cv2.imread('cameraman_ruido.jpg')
img_ruido2 = cv2.imread('cameraman_ruido2.jpg')

print("Tamanho original: ", img_original.shape[:2])
print("Tamanho ruido 1: ", img_ruido1.shape[:2])
print("Tamanho ruido 2: ", img_ruido2.shape[:2])
```

```
Tamanho original: (256, 256)
Tamanho ruido 1: (256, 256)
Tamanho ruido 2: (256, 256)
```

Reduzindo as imagens

```
In [ ]: img_original = cv2.resize(img_original, (120,120))
img_ruido1 = cv2.resize(img_ruido1, (120,120))
img_ruido2 = cv2.resize(img_ruido2, (120,120))

print("Tamanho original: ", img_original.shape[:2])
print("Tamanho ruido 1: ", img_ruido1.shape[:2])
print("Tamanho ruido 2: ", img_ruido2.shape[:2])
```

```
Tamanho original: (120, 120)
Tamanho ruido 1: (120, 120)
Tamanho ruido 2: (120, 120)
```

```
In [ ]: # transformando em escala de cinza
img_original = cv2.cvtColor(img_original, cv2.COLOR_BGR2GRAY)
img_ruido1 = cv2.cvtColor(img_ruido1, cv2.COLOR_BGR2GRAY)
img_ruido2 = cv2.cvtColor(img_ruido2, cv2.COLOR_BGR2GRAY)
```

```
In [ ]: plt.imsave('cameraman_original_reduzido.png', Image.fromarray(img_original,'L'), c
```

Transformando em sinal

```
In [ ]: sinal_original = np.array(img_original).reshape(120**2)
sinal_ruido1 = np.array(img_ruido1).reshape(120**2)
sinal_ruido2 = np.array(img_ruido2).reshape(120**2)
```

Implementação dos filtros e das métricas

Abaixo, estão definidas as funções utilizadas para calcular os filtros Heat e Tikhonov, bem como as métricas de Erro Quadrático Médio (MSE) e Peak Signal-to-Noise Ratio (PSNR).

Na ordem:

- Heat
- Tikhonov
- MSE
- PSNR

```
In [ ]: def heat(lam,tau=1):
    '''Calcula a função do filtro a partir de um vetor de autovalores.

    Parâmetros:
    - lambda (array) - vetor de autovalores

    Return:
    - H (array) - matriz diagonal cujos valores da diagonal são os h(lambda_i) e
    h() é a função que define o filtro
    ...'''
```

```
h_lambda = np.exp(-(tau * lam)) #h(Lam, tau)
H = np.diag(h_lambda)

return H
```

```
In [ ]: def tikhonov(lam,g=1):
    '''Calcula a matriz H da regularização à tikhonov a partir de
    um vetor de autovalores.

    Parâmetros:
    - lam (array): vetor de autovalores
    - g (float): default = 1, valor do parâmetro lambda

    Returns:
    - H (array): matriz diagonal cujas entradas são a função h()
    calculada sobre os autovalores correspondentes.'''
    h_lambda = 1/(1+ g*lam)
    H = np.diag(h_lambda)

    return H
```

```
In [ ]: def MSE(original,filtrado):
    '''Calcula o erro quadrático médio da filtração escolhida.

    Parâmetros:
    - original (array): sinal original, sem ruídos, para usar como
    referência no MSE
    - filtrado (array): sinal obtido com a filtração

    Returns:
    - mse (float): erro quadrático médio'''

    soma = sum((original - filtrado)**2) # soma dos quadrados
    n = len(filtrado)
    mse = soma/n
    return mse
```

```
In [ ]: def PSNR(original,filtrado,img=True):
    '''Calcula o Peak signal-to-noise ratio com relação a alguma filtração
    escolhida. O PSNR é a variação relativa do máximo valor possível
    do sinal em relação ao erro quadrático médio (norma L2 do erro),
    em escala logarítmica.

    Parâmetros:
    - original (array): sinal original
    - filtrado (array): sinal filtrado de acordo com a filtração de
    escolha

    Returns:
    - res (float): medida do PSNR'''
    mse = MSE(original,filtrado) # precisa dessa métrica
    if img:
        max_f = 255
    else:
        # definindo uma métrica alternativa para meu toy problem
        max_f = max(filtrado)
    res = 20*np.log10(max_f) - 10*np.log10(mse)
    return res
```

Grafo Clássico

Nessa seção, vamos fazer todas as contas para os filtros implementados aqui com o grafo construído de maneira clássica, isto é, conectando os pixels adjascentes.

Além disso, vamos dividir os resultados entre os filtros Heat e Tikhonov.

```
In [ ]: # primeiro a identificação das bordas
def bordas(img):
    '''Função que recupera os índices de cada uma das bordas de um array (ou ima

    Parâmetros:
    - img (array): matriz mxn que representa a imagem (ou qualquer outra coisa)

    Return:
    - left (list): índices da borda esquerda
    - right (list): índices da borda direita
    - up (list): índices da borda superior
    - down (list): índices da borda inferior'''
    # matriz m por n
    m, n = img.shape

    left = [i*m for i in range(n)]
    right = [i*m - 1 for i in range(1,n+1)]
    up = [i for i in range(m)]
    down = [n*(m-1) + i for i in range(m)]

    return left, right, up, down
```

```
In [ ]: def grafo_img_simples(img, adj = False, plot=True):
    '''Função que retorna uma instância do objeto graph do pygsp que
    representa uma imagem. Aqui o grafo é o mais simples possível, considerando
    adjascentes os pixels vizinhos.
```

Parâmetros:

```

- img (array): imagem a partir da qual construiremos o grafo
- adj (Bool): default = False, indica se retornamos ou não a matriz de adjacência do grafo criado
- plot (Bool): default = True, faz o desenho do grafo
...
# primeira coisa é pegar as bordas
left, right, up, down = bordas(img)
m,n = img.shape
#W = np.zeros((m*n,m*n))
W = []

for i in range(m*n):
    linha = np.zeros(m*n,dtype=np.int16)

    if i == 0: # canto superior esquerdo
        linha[1] = 1 # a direita
        linha[m] = 1 # abaixo
        linha[m+1] = 1 # diagonal p baixo

    elif i == m-1: # canto superior direito
        linha[i-1] = 1 # a esquerda
        linha[i+m] = 1 # abaixo
        linha[i+m-1] = 1 # diagonal p baixo

    elif i == n*(m-1): # canto inferior esquerdo
        linha[i+1] = 1 # a direita
        linha[i-m] = 1 # acima
        linha[i-m+1] = 1

    elif i == n*m -1: # canto inferior direito
        linha[i-1] = 1 # a esquerda
        linha[i-m] = 1 # acima
        linha[i-m-1] = 1

    elif i in up:
        linha[i-1] = 1
        linha[i+1] = 1
        linha[i+m] = 1
        linha[i+m-1] = 1
        linha[i+m+1] = 1

    elif i in left:
        linha[i+1] = 1
        linha[i-m] = 1
        linha[i+m] = 1
        linha[i-m+1] = 1
        linha[i+m+1] = 1

    elif i in right:
        linha[i-m] = 1
        linha[i+m] = 1
        linha[i-1] = 1
        linha[i-m-1] = 1
        linha[i+m-1] = 1

    elif i in down:
        linha[i-1] = 1
        linha[i+1] = 1
        linha[i-m] = 1

```

```

        linha[i-m+1] = 1
        linha[i-m-1] = 1

    else:
        linha[i-m-1] = 1
        linha[i-m] = 1
        linha[i-m+1] = 1
        linha[i-1] = 1
        linha[i+1] = 1
        linha[i+m-1] = 1
        linha[i+m] = 1
        linha[i+m+1] = 1

    W.append(linha)
W = sp.csc_matrix(np.array(W, dtype=np.int16))
#assert W.shape == (m*n,m*n)

# criando o grafo
G = graphs.Graph(W)

if adj:
    return G, W
elif plot:
    G.set_coordinates()
    G.plot(title=f"Grafo trivial para imagem {m}x{n}")
    return G
else:
    return G

```

Note que essa construção independe do sinal, então vamos fazê-la a partir da imagem original e computar somente os filtros a partir das imagens com ruído.

```
In [ ]: G_classico = grafo_img_simples(img_original, False, False)
```

Heat

Para a implementação desses filtros, foram testados 100 valores no intervalo $[0.00001, 1]$ para investigar o comportamento das métricas MSE e PSNR, como mostra o gráfico abaixo.

Conseguimos verificar uma tendência quadrática e a partir dela obter o valor que minimiza o erro quadrático médio.

```

In [ ]: # calculando autofunções
G_classico.compute_fourier_basis()

U = G_classico.U # autofunções
lam = G_classico.e # autovalores

aux1 = U.T @ sinal_ruido1
aux2 = U.T @ sinal_ruido2

```

2024-06-26 18:44:18,803:[WARNING](pygsp.graphs.graph.compute_fourier_basis): Computing the full eigendecomposition of a large matrix (14400 x 14400) may take some time.

```
In [ ]: # calculando para parametros dentro de [epsilon,1] para ver qual é o melhor
tau = np.linspace(1e-4,1,100)

heat_ruido1, heat_ruido2 = [], []
mse_heat1, mse_heat2, psnr_heat1, psnr_heat2 = [], [], [], []

for t in tau:
    r1 = U @ heat(lam, tau = t) @ aux1
    r2 = U @ heat(lam, tau = t) @ aux2

    # guardando resultados
    heat_ruido1.append(np.round(r1))
    heat_ruido2.append(np.round(r2))

    # metricas de erro
    mse_heat1.append(MSE(sinal_original,r1))
    mse_heat2.append(MSE(sinal_original,r2))
    psnr_heat1.append(PSNR(sinal_original,r1))
    psnr_heat2.append(PSNR(sinal_original,r2))
```

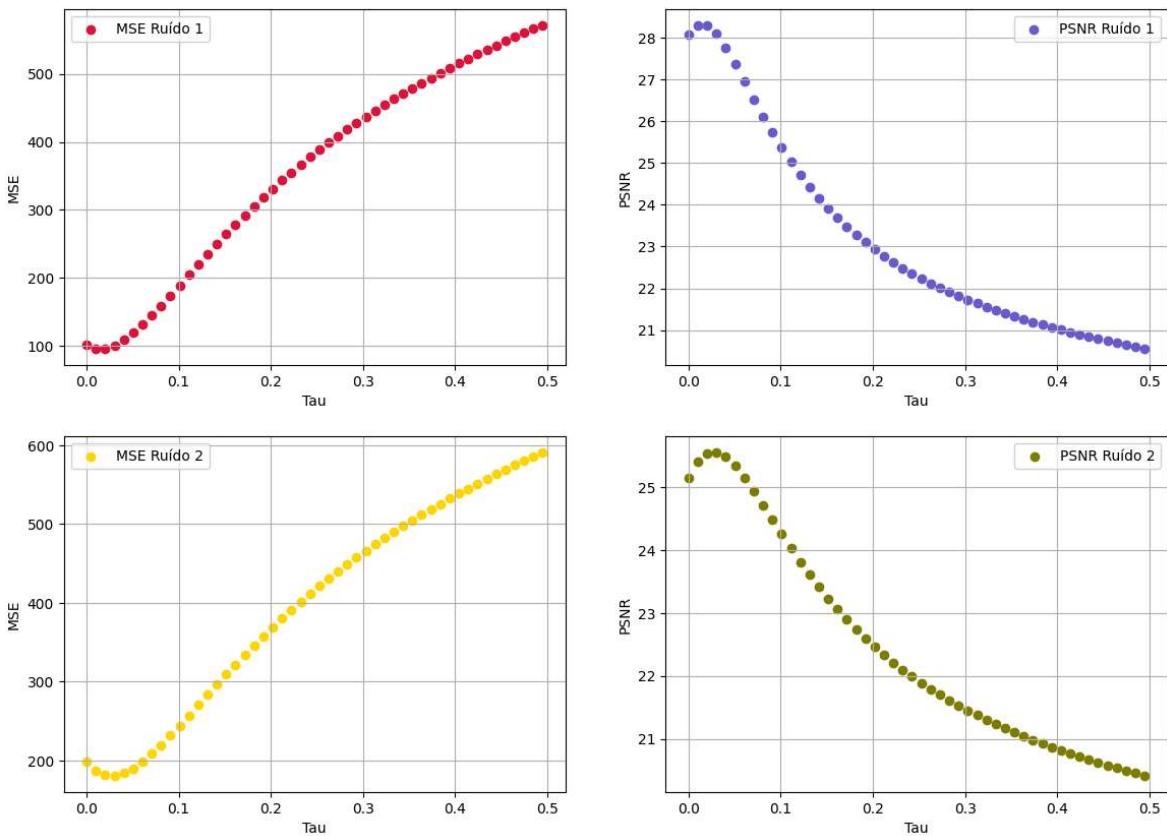
```
In [ ]: fig, ax = plt.subplots(figsize = (14,10), ncols = 2, nrows = 2)

ax[0,0].scatter(tau[:50], mse_heat1[:50], label = "MSE Ruído 1", color = "crimson")
ax[0,0].legend()
ax[0,0].set_xlabel("Tau")
ax[0,0].set_ylabel("MSE")
ax[0,0].grid()

ax[0,1].scatter(tau[:50], psnr_heat1[:50], label = "PSNR Ruído 1", color = "slateblue")
ax[0,1].legend()
ax[0,1].set_xlabel("Tau")
ax[0,1].set_ylabel("PSNR")
ax[0,1].grid()

ax[1,0].scatter(tau[:50],mse_heat2[:50], label = "MSE Ruído 2", color = "gold")
ax[1,0].legend()
ax[1,0].set_xlabel("Tau")
ax[1,0].set_ylabel("MSE")
ax[1,0].grid()

ax[1,1].scatter(tau[:50],psnr_heat2[:50], label = "PSNR Ruído 2", color = "olivedrab")
ax[1,1].legend()
ax[1,1].set_xlabel("Tau")
ax[1,1].set_ylabel("PSNR")
ax[1,1].grid()
```



```
In [ ]: # selecionando os parâmetros
ind1 = np.argmin(np.array(mse_heat1))
ind2 = np.argmin(np.array(mse_heat2))

tau1 = tau[ind1]
tau2 = tau[ind2]

heat_ruido1_otimo = heat_ruido1[ind1]
heat_ruido2_otimo = heat_ruido2[ind2]
```

A fim de comparação, indicamos abaixo os índices dos valores de τ que minimizam o MSE e maximizam o PSNR, respectivamente.

Note que utilizamos o `argmax` para o PSNR porque ele é minimizado sempre quando o parâmetro aumenta, pois quanto maior o parâmetro mais suavizado fica o sinal.

```
In [ ]: print("MSE 1)", ind1, "MSE2)", ind2, "PSNR 1)", np.argmax(np.array(psnr_heat1)), "PSNR 2)", np.argmax(np.array(psnr_heat2)))
```

MSE 1) 1 MSE2) 3 PSNR 1) 1 PSNR 2) 3

Tikhonov

O parâmetro γ foi variado de 0.01 a 4 e vamos selecionar o valor que minimiza o erro quadrático médio.

```
In [ ]: gamas = np.linspace(1e-6, 2, 50)

tik_ruido1, tik_ruido2 = [], []
mse_tik1, mse_tik2, psnr_tik1, psnr_tik2 = [], [], [], []

for t in gamas:
```

```
r1 = U @ tikhonov(lam, g = t) @ aux1
r2 = U @ tikhonov(lam, g = t) @ aux2

# guardando resultados
tik_ruido1.append(np.round(r1))
tik_ruido2.append(np.round(r2))

# metricas de erro
mse_tik1.append(MSE(sinal_original,r1))
mse_tik2.append(MSE(sinal_original,r2))
psnr_tik1.append(PSNR(sinal_original,r1))
psnr_tik2.append(PSNR(sinal_original,r2))
```

```
In [ ]: fig, ax = plt.subplots(figsize = (14,10), ncols = 2, nrows = 2)

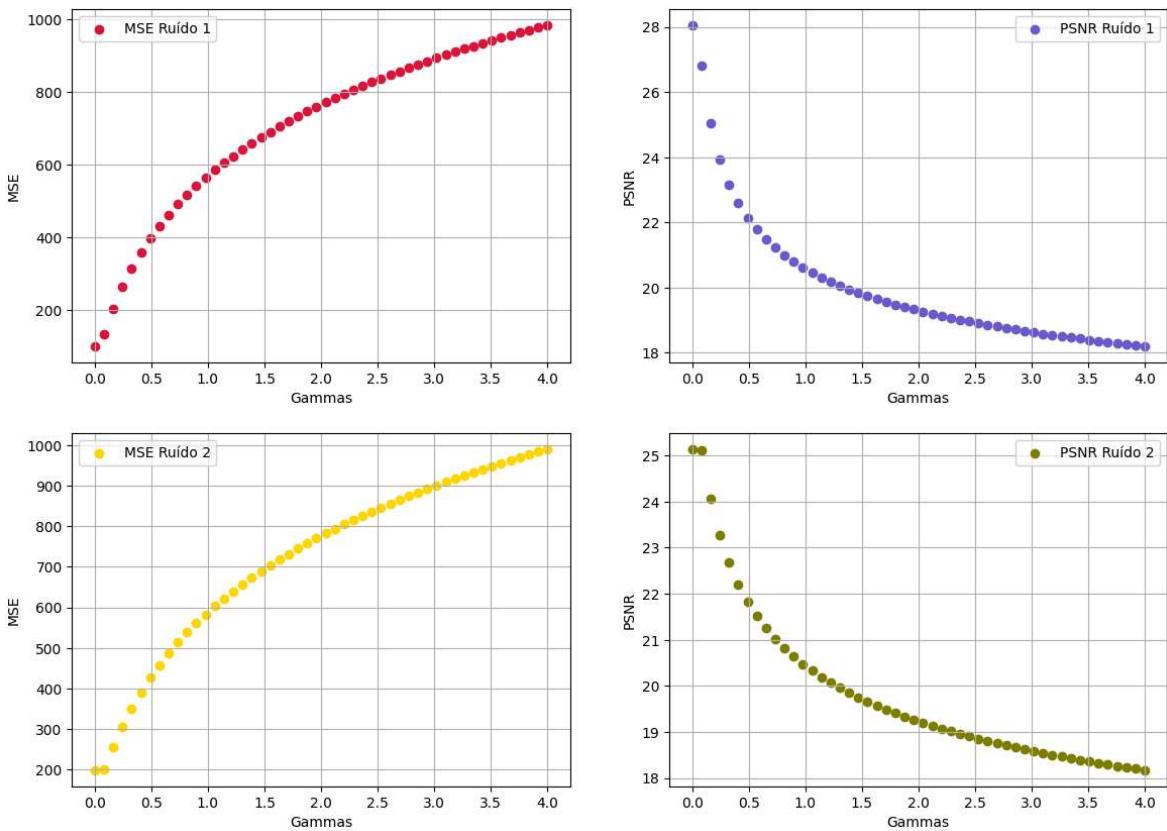
ax[0,0].scatter(gamas, mse_tik1, label = "MSE Ruído 1", color = 'crimson')
ax[0,0].set_xlabel("Gammas")
ax[0,0].set_ylabel("MSE")
ax[0,0].grid()
ax[0,0].legend()

ax[0,1].scatter(gamas, psnr_tik1, label = "PSNR Ruído 1", color = 'slateblue')
ax[0,1].set_xlabel("Gammas")
ax[0,1].set_ylabel("PSNR")
ax[0,1].grid()
ax[0,1].legend()

ax[1,0].scatter(gamas,mse_tik2, label = "MSE Ruído 2", color = 'gold')
ax[1,0].set_xlabel("Gammas")
ax[1,0].set_ylabel("MSE")
ax[1,0].grid()
ax[1,0].legend()

ax[1,1].scatter(gamas,psnr_tik2, label = "PSNR Ruído 2", color = 'olive')
ax[1,1].set_xlabel("Gammas")
ax[1,1].set_ylabel("PSNR")
ax[1,1].grid()
ax[1,1].legend()
```

```
Out[ ]: <matplotlib.legend.Legend at 0x1c0c0d3d310>
```



```
In [ ]: # selecionando os parâmetros
ind1_t = np.argmin(np.array(mse_tik1))
ind2_t = np.argmin(np.array(mse_tik2))

gama1 = gamas[ind1_t]
gamas2 = tau[ind2_t]

tik_ruido1_otimo = tik_ruido1[ind1_t]
tik_ruido2_otimo = tik_ruido2[ind2_t]
```

```
In [ ]: print("MSE 1)", ind1_t, "MSE2)", ind2_t, "PSNR 1)", np.argmax(np.array(psnr_tik1)),
MSE 1) 0 MSE2) 1 PSNR 1) 0 PSNR 2) 1
```

Resultados

Primeiro vamos comparar os resultados numéricos e depois a olho nu.

```
In [ ]: labels = [f"Heat Ótimo {tau[ind1]}", f"Tikhonov Ótimo ({gamas[ind1_t]})"]
resultados = pd.DataFrame(data = np.array([labels,
                                           [mse_heat1[ind1], mse_tik1[ind1_t]],
                                           [mse_heat2[ind2], mse_tik2[ind2_t]],
                                           [psnr_heat1[ind1], psnr_tik1[ind1_t]],
                                           [psnr_heat2[ind2], psnr_tik2[ind2_t]]]),
                           columns = ['Método', 'MSE Ruído 1', 'MSE Ruído 2', 'P
resultados
```

Out[]:

	Método	MSE Ruído 1	MSE Ruído 2	PSNR Ruído 1
0	Heat Ótimo 0.01019999999999999	96.13432628912464	180.9036620647914	28.302018737600786
1	Tikhonov Ótimo (1e-06)	101.63641856034351	181.7897113827877	28.06031007784385

Vemos que os melhores resultados para o MSE são atingidos conforme tomamos valores menores para τ no filtro heat - nesse caso 0.01 -, enquanto que os menores valores para o PSNR são do filtro tikhonov e heat com valores de parâmetro maiores (1 e 1.5 respectivamente). Isso possivelmente se deve ao fato destes dois últimos serem mais homogenizadores que os demais valores testados.

Além disso, note que o erro quadrático médio dos filtros heat é bem maior para tratar o ruído 2.

In []:

```
# salvando as imagens
# ruido 1
plt.imsave('cameraman_CR1H.png', heat_ruido1[ind1].reshape((120,120)), cmap='gray')

# tikhonov
plt.imsave('cameraman_CR1TIK.png', tik_ruido1[ind1_t].reshape((120,120)), cmap='gray')

# ruido 2
plt.imsave('cameraman_CR2H.png', heat_ruido2[ind2].reshape((120,120)), cmap='gray')

# tikhonov
plt.imsave('cameraman_CR2TIK.png', tik_ruido2[ind2_t].reshape((120,120)), cmap='gray')
```

Ruido 1



Ruido 2



Podemos concluir que o filtro heat com parâmetro $\tau = 0.01$ se saiu melhor na comparação a olho nu, possivelmente por ser o menos homogenizador dos filtros, i.e., preserva as diferenças de tonalidade entre vértices adjascentes.

Grafo com Pesos

Nessa seção, introduzimos um novo método para construção dos grafos, em que é aplicado um peso Gaussiano quando dois vértices são κ -próximos, ou seja,

$$a_{ij} = e^{-d^2/2\sigma^2}, \text{ se } |d(i,j)| < \kappa$$

A distância aqui utilizada é a diferença entre as cores de cada pixel em grayscale.

```
In [ ]: def grafo_peso_grayscale(img, th, kap, adj=False, plot=True):
    '''Função que constrói o grafo para uma imagem em escala de cinzas
    a partir do gradiente de cor. Os pesos atribuídos a cada vértice serão
    uma gaussiana de variância theta^2 avaliada em x = gray(i) - gray(j),
    onde gray() é o valor na escala de cinzas de cada pixel.

    Parâmetros:
    - img (array): matrix representando a imagem, suas entradas correspondem ao
    de cada vértice na escala de cinza
    - th (float): desvio padrão da Gaussiana
    - kap (float): threshold de conexão dos vértices'''

    m, n = img.shape
    img_array = np.array(img).reshape(n*m)
    N = m*n
    W = np.zeros((N,N))
    #W = sp.coo_matrix(shape=(N,N), dtype=np.float16)
    for i in range(N):
        for j in range(N):
            dist = abs(img_array[i] - img_array[j])
            #print(img[ij], img[jj], dist)
            if dist < kap and i!=j:
                W[i,j] = np.exp(-((dist*dist)/(2*th*th)), dtype=np.float32)
                #print(W[i,j])
                assert W[i,j] <= 1

    G = graphs.Graph(W)
    if adj:
        return G, W
    elif plot:
        G.set_coordinates()
        G.plot(title=f"Grafo com pesos $\kappaappa = $ {kap}")
        return G
    else:
        return G
```

Diferentemente do primeiro grafo, essa construção depende do valor do sinal para calcular as distâncias entre os vértices, então temos um grafo para cada ruído diferente.

```
In [ ]: G_ruido1_peso = grafo_peso_grayscale(img_ruido1, 1, 5, False, False)
G_ruido2_peso = grafo_peso_grayscale(img_ruido2, 1, 5, False, False)
```

```
C:\Users\mevel\AppData\Local\Temp\ipykernel_10120\622753297.py:20: RuntimeWarning
g: overflow encountered in scalar subtract
    dist = abs(img_array[i] - img_array[j])
```

```
In [ ]: G_ruido1_peso.compute_fourier_basis()
G_ruido1_peso.e[0] = 0
```

2024-06-28 09:40:54,180:[WARNING](pygsp.graphs.graph.compute_fourier_basis): Computing the full eigendecomposition of a large matrix (14400 x 14400) may take some time.

```
In [ ]: G_ruido2_peso.compute_fourier_basis()
G_ruido2_peso.e[0] = 0
```

2024-06-28 09:54:01,116:[WARNING](pygsp.graphs.graph.compute_fourier_basis): Computing the full eigendecomposition of a large matrix (14400 x 14400) may take some time.

```
In [ ]: U_peso1 = G_ruido1_peso.U
lam1 = G_ruido1_peso.e

U_peso2 = G_ruido2_peso.U
lam2 = G_ruido2_peso.e

aux1 = U_peso1.T @ sinal_ruido1
aux2 = U_peso2.T @ sinal_ruido2
```

Heat

```
In [ ]: # calculando para parametros dentro de [epsilon,1] para ver qual é o melhor
tau = np.linspace(1e-4,1,50)

heat_peso1 = [] # resultados da filtragem
heat_peso2 = []

# listas para guardar as métricas de erro
mse_peso1, mse_peso2 = [], []
psnr_peso1, psnr_peso2 = [], []
for t in tau:
    # calculando o sinal filtrado
    r1 = U_peso1 @ heat(lam1, tau = t) @ aux1
    r2 = U_peso2 @ heat(lam2, tau = t) @ aux2

    # guardando resultados
    heat_peso1.append(np.round(r1))
    heat_peso2.append(np.round(r2))

    # metricas de erro
    mse_peso1.append(MSE(sinal_original,r1))
    mse_peso2.append(MSE(sinal_original,r2))
    psnr_peso1.append(PSNR(sinal_original,r1))
    psnr_peso2.append(PSNR(sinal_original,r2))
```

```
In [ ]: fig, ax = plt.subplots(figsize = (14,10), ncols = 2, nrows = 2)

ax[0,0].scatter(tau, mse_peso1, label = "MSE Ruído 1", color = "crimson")
ax[0,0].set_xlabel("Tau")
ax[0,0].set_ylabel("MSE")
```

```

ax[0,0].grid()
ax[0,0].legend()

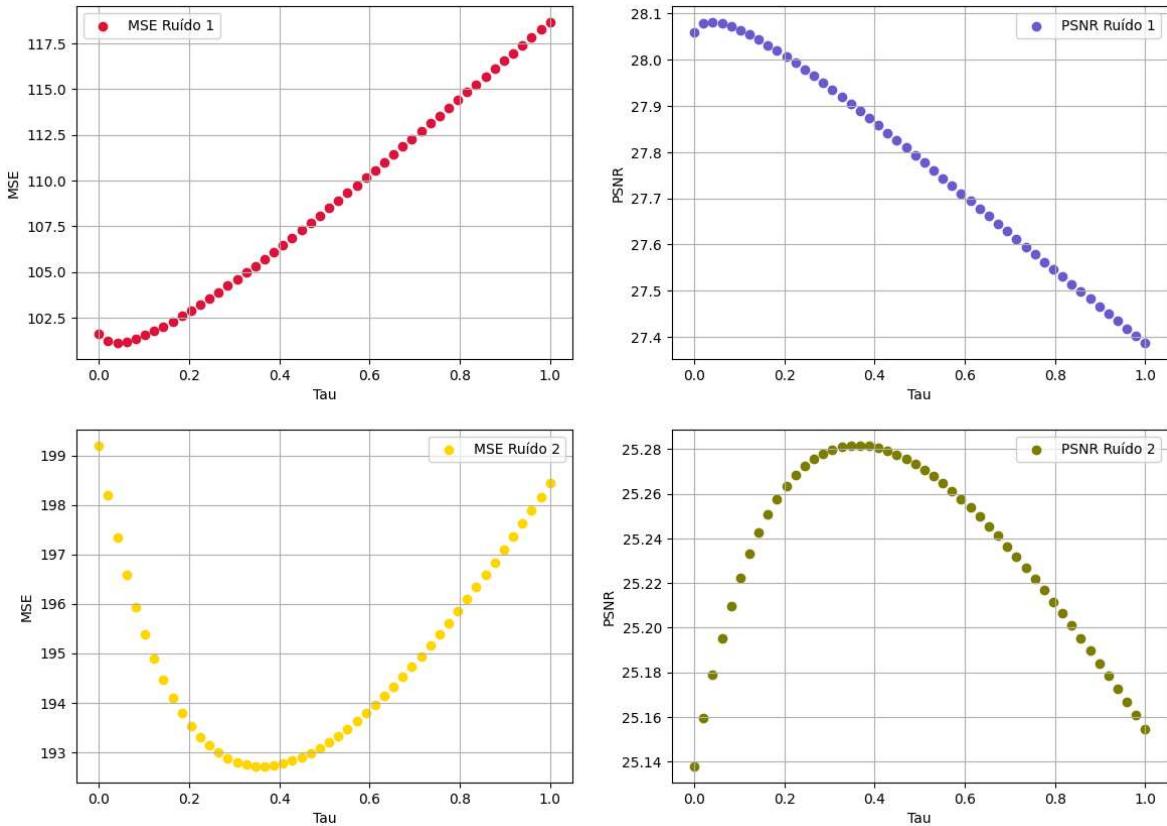
ax[0,1].scatter(tau, psnr_peso1, label = "PSNR Ruído 1", color = "slateblue")
ax[0,1].set_xlabel("Tau")
ax[0,1].set_ylabel("PSNR")
ax[0,1].grid()
ax[0,1].legend()

ax[1,0].scatter(tau,mse_peso2, label = "MSE Ruído 2", color = "gold")
ax[1,0].set_xlabel("Tau")
ax[1,0].set_ylabel("MSE")
ax[1,0].grid()
ax[1,0].legend()

ax[1,1].scatter(tau,psnr_peso2, label = "PSNR Ruído 2", color = "olive")
ax[1,1].set_xlabel("Tau")
ax[1,1].set_ylabel("PSNR")
ax[1,1].grid()
ax[1,1].legend()

```

Out[]: <matplotlib.legend.Legend at 0x1c0cda1a890>



```

In [ ]: # selecionando os parâmetros
ind1p = np.argmin(np.array(mse_peso1))
ind2p = np.argmin(np.array(mse_peso2))

tau1 = tau[ind1p]
tau2 = tau[ind2p]

heat_peso1_otimo = heat_peso1[ind1p]
heat_peso2_otimo = heat_peso2[ind2p]

```

```
In [ ]: print("MSE 1)",ind1p,"MSE2)", ind2p,"PSNR 1)", np.argmax(np.array(psnr_peso1)), "
```

```
MSE 1) 2 MSE2) 18 PSNR 1) 2 PSNR 2) 18
```

```
In [ ]: print("Tau 1: ", tau1, "Tau 2: ", tau2)
```

```
Tau 1: 0.04091224489795919 Tau 2: 0.36741020408163266
```

Tikhonov

```
In [ ]: gamas = np.linspace(1e-5,10,50)
```

```
tik_peso1, tik_peso2 = [], []
mse_tik_peso1, mse_tik_peso2, psnr_tik_peso1, psnr_tik_peso2 = [], [], [], []

for t in gamas:
    r1 = U_peso1 @ tikhonov(lam, g = t) @ aux1
    r2 = U_peso2 @ tikhonov(lam, g = t) @ aux2

    # guardando resultados
    tik_peso1.append(np.round(r1))
    tik_peso2.append(np.round(r2))

    # metricas de erro
    mse_tik_peso1.append(MSE(sinal_original,r1))
    mse_tik_peso2.append(MSE(sinal_original,r2))
    psnr_tik_peso1.append(PSNR(sinal_original,r1))
    psnr_tik_peso2.append(PSNR(sinal_original,r2))
```

```
In [ ]: fig, ax = plt.subplots(figsize = (14,10), ncols = 2, nrows = 2)
```

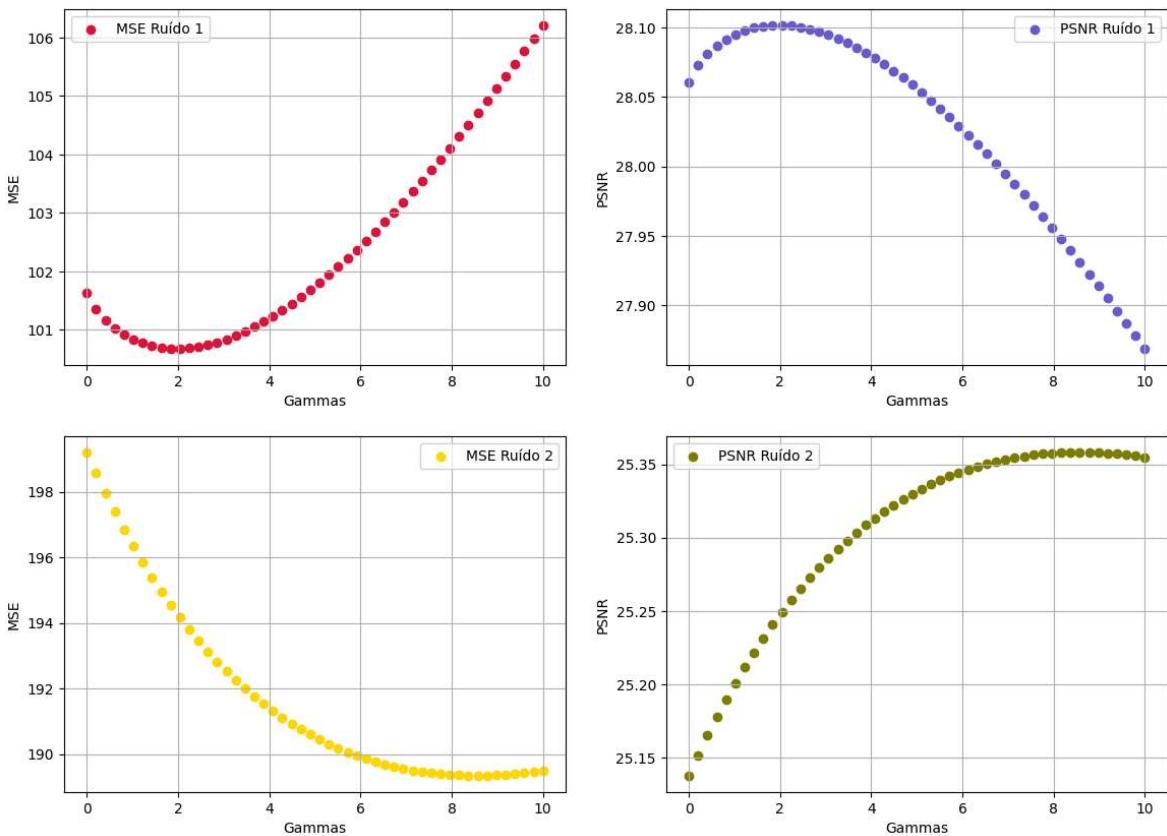
```
ax[0,0].scatter(gamas, mse_tik_peso1, label = "MSE Ruído 1", color = "crimson")
ax[0,0].set_xlabel("Gammas")
ax[0,0].set_ylabel("MSE")
ax[0,0].grid()
ax[0,0].legend()

ax[0,1].scatter(gamas, psnr_tik_peso1, label = "PSNR Ruído 1", color = "slateblue")
ax[0,1].set_xlabel("Gammas")
ax[0,1].set_ylabel("PSNR")
ax[0,1].grid()
ax[0,1].legend()

ax[1,0].scatter(gamas,mse_tik_peso2, label = "MSE Ruído 2", color = "gold")
ax[1,0].set_xlabel("Gammas")
ax[1,0].set_ylabel("MSE")
ax[1,0].grid()
ax[1,0].legend()

ax[1,1].scatter(gamas,psnr_tik_peso2, label = "PSNR Ruído 2", color = "olive")
ax[1,1].set_xlabel("Gammas")
ax[1,1].set_ylabel("PSNR")
ax[1,1].grid()
ax[1,1].legend()
```

```
Out[ ]: <matplotlib.legend.Legend at 0x1c0d968d410>
```



```
In [ ]: # selecionando os parâmetros
ind1_tp = np.argmin(np.array(mse_tik_pes01))
ind2_tp = np.argmin(np.array(mse_tik_pes02))

gama1 = gamas[ind1_tp]
gama2 = gamas[ind2_tp]

tik_pes01_otimo = tik_pes01[ind1_tp]
tik_pes02_otimo = tik_pes02[ind2_tp]
```

```
In [ ]: print("MSE 1)", ind1_tp, "MSE2)", ind2_tp, "PSNR 1)", np.argmax(np.array(psnr_tik_p
MSE 1) 10 MSE2) 42 PSNR 1) 10 PSNR 2) 42
```

```
In [ ]: print("Gamma 1: ", gama1, "Gamma 2: ", gama2)
```

```
Gamma 1: 2.0408242857142858 Gamma 2: 8.57143
```

Resultados

Como no caso anterior, vamos fazer os resultados numéricos primeiro.

```
In [ ]: labels = [f'Heat ótimo', 'Tikhonov ótimo']
resultados = pd.DataFrame(data = np.array([labels,
                                           [mse_heat1[ind1p], mse_tik1[ind1_tp],
                                            mse_heat2[ind2p], mse_tik2[ind2_tp]],
                                           [psnr_heat1[ind1p], psnr_tik1[ind1_t
                                           [psnr_heat2[ind2p], psnr_tik2[ind2_t
resultados
```

Out[]:	Método	MSE Ruído 1	MSE Ruído 2	PSNR Ruído 1	PSNR Ruído 2
0	Heat ótimo	96.35998110772452	346.0658956093014	28.29183654689091	22.7392155868
1	Tikhonov ótimo	358.35626078765534	731.9822778295965	22.587653645033193	19.4857979447

Veja que, novamente, os filtros que tem os menores parâmetros são aqueles que saem melhor nas métricas de erro quadrático médio (Heat 1 e Tikhonov 3). No entanto,

Agora, a comparação a olho nu.

```
In [ ]: # salvando as imagens
# ruido 1
plt.imsave('cameraman_PR1H.png', heat_peso1_otimo.reshape((120,120)), cmap='gray')

# tikhonov
plt.imsave('cameraman_PR1TIK.png', tik_peso1_otimo.reshape((120,120)), cmap='gray')

# ruido 2
plt.imsave('cameraman_PR2H.png', heat_peso2_otimo.reshape((120,120)), cmap='gray')

# tikhonov
plt.imsave('cameraman_PR2TIK.png', tik_peso2_otimo.reshape((120,120)), cmap='gray')
```

Ruido 1 - Pesos



Ruido 2 - Pesos



Note que, comparado aos resultados com o filtro clássico, as imagens são muito parecidas para diferentes filtrações, possivelmente porque a construção desse filtro já

separa bem os vértices com tonalidades mais distantes, não havendo a suavização do sinal entre eles quando não são adjascentes.

Ainda assim, podemos ver que para o caso do ruído 2, tanto o filtro Heat quanto Tikhonov não conseguiram filtrar muito bem o ruído no fundo da imagem.

Para o caso do ruído 1 os resultados são melhores e bem mais similares.

Comparação com o PyGSP

Como o filtro de regularização à Tikhonov não é implementado em nenhum método do PyGSP, vamos comparar apenas com o filtro heat.

Como o `compute_fourier_basis` é comum aos dois métodos, basta calcular $Uh(\lambda)U^T x$, onde:

- U é a matriz de autovetores de L
- $h(\lambda)$ é a função que define o filtro aplicada aos autovalores
- x é o sinal

Ruido 1

```
In [ ]: %%timeit
U @ heat(lam, tau = 1) @ U.T @ sinal_ruido1
1min 16s ± 1.68 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [ ]: %%timeit
U_peso1 @ heat(lam1, tau = 1) @ U_peso1.T @ sinal_ruido1
5min 45s ± 890 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [ ]: %%timeit
filters.Heat(G_classico,1).filter(sinal_ruido1,method='exact')
873 ms ± 65.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Ruido 2

```
In [ ]: %%timeit
U @ heat(lam, tau = 1) @ U.T @ sinal_ruido2
1min 14s ± 164 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [ ]: %%timeit
U_peso2 @ heat(lam2, tau = 1) @ U_peso2.T @ sinal_ruido2
1min 14s ± 195 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [ ]: %%timeit
filters.Heat(G_classico,1).filter(sinal_ruido2,method='exact')
827 ms ± 26.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Resultados

Vemos, claramente, que o método de filtro do PyGSP é mais otimizado.