Eduardo Slomp Arán e João Pedro Gubertt

FILTRAGEM ESPACIAL X FILTRAGEM NO DOMÍNIO DA FREQUÊNCIA

Trabalho apresentado aos Cursos de Ciências da Computação da UNIVALI - Universidade do Vale do Itajaí, para a disciplina Processamento de Imagens.

Prof: Felipe Viel

Itajaí - Santa Catarina
2024

SUMÁRIO

1 INTRODUÇÃO

Neste trabalho foram implementados os filtros de processamento de imagens, utilizando das métricas de PSNR, MSE e RMSE para medir e analisar a qualidade de cada imagem. Foram aplicados filtros como Gaussiano e Sobel, filtros de frequência, incluindo passa-baixa e passa-alta nas variantes ideal e Gaussiana. Com os resultados obtidos foi possível determinar, de acordo com as métricas utilizadas, qual das determinadas imagens tiveram um melhor desempenho.

2 RESULTADOS

Resultados obtidos a partir das métricas de qualidade de imagem (MSE, PSNR e RMSE).

Imagem original:

Imagem Gaussian:

Comparação de resultados:

Gaussiano:

MSE: **513.73** | PSNR: **21.02** | RMSE: **22.67**

Passa-Baixa Ideal:

MSE: **137.40** | PSNR: **26.75** | RMSE: **11.72**

Passa-Baixa Gaussiano (d0=35):

MSE: **123.89** | PSNR: **27.20** | RMSE: **11.13**

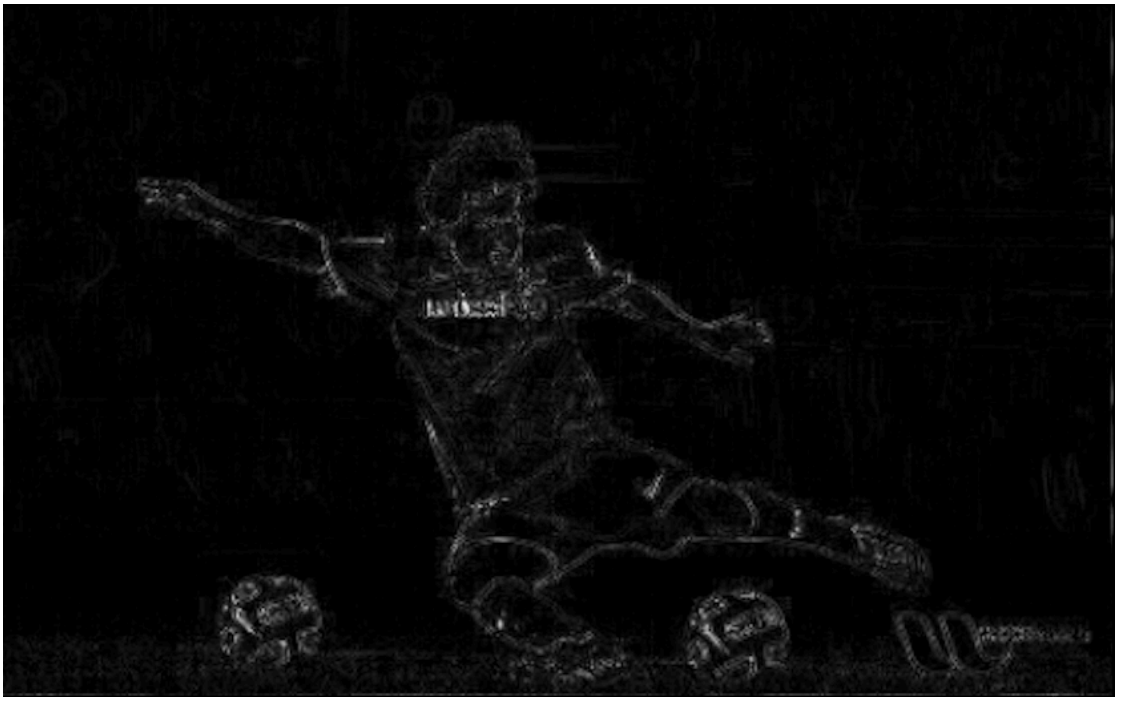
Resultados com Canny:

Passa-Alta Ideal:

MSE: **3244.39** | PSNR: **31.12** | RMSE: **56.96**

Passa-Alta Gaussiano do = 80:

MSE: **3121.16** | PSNR: **33.02** | RMSE: **55.87**



Sobel:

MSE: **2305.54** | PSNR: **29.11** | RMSE: **48.02**

Diante dos resultados obtidos, pode-se observar que filtros passa-baixa ideal e gaussiano possuem MSE e RMSE inferiores, indicando melhor preservação dos detalhes e menor distorção, com PSNR acima de 26. Porém, os filtros passa-alta (ideal e gaussiano) apresentaram valores elevados de MSE e RMSE, sugerindo aumento de ruídos, além do passa-alta gaussiano, com PSNR maior, superou o filtro Sobel na detecção de bordas.

3 CÓDIGO IMPLEMENTADO

Métricas:

```python
def MSE(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return err

def RMSE(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return np.sqrt(err)

def PSNR(original, compressed):
    mse = np.mean((original - compressed) ** 2)
    if(mse == 0):  # MSE is zero means no noise is present in the signal .
                   # Therefore PSNR have no importance.
        return 100
    max_pixel = 255.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return psnr
```

Passa-Baixa/Passa-Alta:

```
def createPB(shape, center, radius, lpType=2, n=2):
    rows, cols = shape[:2]
    r, c = np.mgrid[0:rows:1, 0:cols:1]
    c -= center[0]
    r -= center[1]
    d = np.power(c, 2.0) + np.power(r, 2.0)
    lpFilter_matrix = np.zeros(shape, np.float32)
    if lpType == 0:  # ideal low-pass filter
        lpFilter = np.copy(d)
        lpFilter[lpFilter < pow(radius, 2.0)] = 1
        lpFilter[lpFilter >= pow(radius, 2.0)] = 0
    elif lpType == 1: #Butterworth low-pass filter
        lpFilter = 1.0 / (1 + np.power(np.sqrt(d)/radius, 2*n))
    elif lpType == 2: # Gaussian low pass filter
        lpFilter = np.exp(-d/(2*pow(radius, 2.0)))
    lpFilter_matrix[:, :, 0] = lpFilter
    lpFilter_matrix[:, :, 1] = lpFilter
    return lpFilter_matrix

def createPA(shape, center, radius, lpType=2, n=2):
    rows, cols = shape[:2]
    r, c = np.mgrid[0:rows:1, 0:cols:1]
    c -= center[0]
    r -= center[1]
    d = np.power(c, 2.0) + np.power(r, 2.0)
    lpFilter_matrix = np.zeros(shape, np.float32)
    if lpType == 0:  # Ideal high pass filter
        lpFilter = np.copy(d)
        lpFilter[lpFilter < pow(radius, 2.0)] = 0
        lpFilter[lpFilter >= pow(radius, 2.0)] = 1
    elif lpType == 1: #Butterworth Highpass Filters
        lpFilter = 1.0 - 1.0 / (1 + np.power(np.sqrt(d)/radius, 2*n))
    elif lpType == 2: # Gaussian Highpass Filter
        lpFilter = 1.0 - np.exp(-d/(2*pow(radius, 2.0)))
    lpFilter_matrix[:, :, 0] = lpFilter
    lpFilter_matrix[:, :, 1] = lpFilter
    return lpFilter_matrix
```

```python
#Passa-baixa Ideal
print('passa-baixa ideal')
maskPB = createPB(dft_shift.shape, center=(int(ncols/2), int(nrows/2)), radius=35, lpType=0, n=2) #Quanto maior o radius aqui, menos borrada
#maskPB = createPB(dft_shift.shape, center=maxLoc, radius=35, lpType=1, n=2)
filtered_freqPB = dft_shift*maskPB
f_ishiftPB = np.fft.ifftshift(filtered_freqPB)  #inversa da fft
img_backPB = cv2.idft(f_ishiftPB)        #inversa da dft
img_backPB = cv2.magnitude(img_backPB[:,:,0],img_backPB[:,:,1])  #recuperando a imagem capturando a magnitude (intesidade)
img_backPB = np.array(img_backPB, dtype=np.float32)

#Normalizando a imagem
print('normalizando a imagem')
img_backPB -= img_backPB.min()
img_backPB = img_backPB * 255 / img_backPB.max()
img_backPB = img_backPB.astype(np.uint8)

#Valores das métricas
print('valores das métricas')
mse_PB = MSE(img_backPB, image_f32)
rmse_PB = RMSE(img_backPB, image_f32)
psnr_PB =  PSNR(img_backPB, image_f32)

#Passa-baixa Gaussiano
print('passa-baixa gaussiano')
d0PBG = 35 # quanto maior menor o blur
maskPBG = createPB(dft_shift.shape, center=(int(ncols/2), int(nrows/2)), radius=d0PBG, lpType=2, n=2)
filtered_freqPBG = dft_shift*maskPBG
f_ishiftPBG = np.fft.ifftshift(filtered_freqPBG)  #inversa da fft
img_backPBG = cv2.idft(f_ishiftPBG)        #inversa da dft
img_backPBG = cv2.magnitude(img_backPBG[:,:,0],img_backPBG[:,:,1])  #recuperando a imagem capturando a magnitude (intesidade)
```

```python
#Passa-Alta Ideal
print("passa alta ideal")
maskPA = createPA(dft_shift.shape, center=(int(ncols/2), int(nrows/2)), radius=35, lpType=0, n=2)
filtered_freq = dft_shift*maskPA
f_ishift = np.fft.ifftshift(filtered_freq)  #inversa da fft
img_backPA = cv2.idft(f_ishift)        #inversa da dft
img_backPA = cv2.magnitude(img_backPA[:,:,0],img_backPA[:,:,1])  #recuperando a imagem capturando a magnitude (intesidade)
img_backPA = np.array(img_backPA, dtype=np.float32)

#Normalizando a img
print("normalizando a imagem")
img_backPA = np.array(img_backPA, dtype=np.float32)
img_backPA -= img_backPA.min()
img_backPA = img_backPA * 255 / img_backPA.max()
img_backPA = img_backPA.astype(np.uint8)

#Valores métricas
print("valores metricas")
mse_backPA = MSE(img_backPA, canny_img)
rmse_backPA = RMSE(img_backPA, canny_img)
psnr_backPA =  PSNR(img_backPA, canny_img)

#Passa-Alta Gaussiano
print("passa alto gaussiano")
d0PAG = 80
maskPAG = createPA(dft_shift.shape, center=(int(ncols/2), int(nrows/2)), radius=d0PAG, lpType=2, n=2) #d0 = 80
filtered_freq = dft_shift*maskPAG
f_ishiftPAG = np.fft.ifftshift(filtered_freq)  #inversa da fft
img_backPAG = cv2.idft(f_ishiftPAG)        #inversa da dft
img_backPAG = cv2.magnitude(img_backPAG[:,:,0],img_backPAG[:,:,1])  #recuperando a imagem capturando a magnitude (intesidade)
img_backPAG = np.array(img_backPAG, dtype=np.float32)

#Normalizando a img
print("normalizando a imagem")
img_backPAG = np.array(img_backPAG, dtype=np.float32)
img_backPAG -= img_backPAG.min()
img_backPAG = img_backPAG * 255 / img_backPAG.max()
img_backPAG = img_backPAG.astype(np.uint8)
```