UNIVERSIDADE DO VALE DO ITAJAÍ
Escola Politécnica
Ciência da Computação

Professor: Felipe Viel

ANDRÉ HILLESHEIM MEYER
FILIPI SCALVIN DA COSTA

FILTRAGEM ESPACIAL X FILTRAGEM NO DOMÍNIO DA FREQUÊNCIA

Itajaí
2024

SUMÁRIO

1. INTRODUÇÃO

O objetivo deste trabalho é realizar uma análise comparativa entre diferentes técnicas de filtragem de imagens, aplicando filtros tanto no domínio espacial quanto no domínio da frequência. A proposta é investigar o desempenho das técnicas de filtragem e compará-las utilizando métricas quantitativas: PSNR (Pico de Relação Sinal-Ruído), RMSE (Raiz do Erro Quadrático Médio) e MSE (Erro Quadrático Médio).

A análise será dividida em dois cenários principais:

1. **Comparação entre Filtro Espacial de Esmaecimento Gaussiano e Filtros Passa-Baixa Ideal e Gaussiano no Domínio da Frequência**

   Nesta etapa, serão aplicados dois tipos de filtros: no domínio espacial, será utilizado o filtro de esmaecimento gaussiano; no domínio da frequência, serão implementados filtros passa-baixa ideal e gaussiano. A comparação dos resultados de filtragem será baseada nas métricas PSNR, RMSE e MSE, buscando avaliar a eficácia de cada técnica.

2. **Comparação entre Filtro Sobel e Filtros Passa-Alta Ideal e Gaussiano, com Referência ao Filtro Canny**

   A segunda parte do projeto envolve uma análise entre o filtro Sobel, que detecta bordas no domínio espacial, e os filtros passa-alta ideal e gaussiano, aplicados no domínio da frequência. A imagem resultante do filtro de detecção de bordas de Canny será usada como referência para comparar a saída do filtro Sobel e dos filtros passa-alta. As métricas utilizadas para avaliar a precisão e o desempenho de cada abordagem serão PSNR, RMSE e MSE.

2. IMPLEMENTAÇÃO

2.1 Implementação da Convolução e do Filtro Gaussiano

```python
def add_padding(img: np.ndarray, padding_height: int, padding_width: int):
    n, m = img.shape

    padded_img = np.zeros((n + padding_height * 2, m + padding_width * 2))
    padded_img[padding_height: n + padding_height, padding_width: m + padding_width] = img

    return padded_img
```

Figura 1: Imagem da implementação da função ‘add_padding’

Esta função recebe uma imagem representada como um array NumPy (img) e adiciona uma borda ao redor dela. A quantidade de padding é determinada pelos parâmetros padding_height e padding_width, que representam o número de pixels adicionados vertical e horizontalmente, respectivamente.

Parâmetros:

- img: Array NumPy 2D representando a imagem.
- padding_height: Quantidade de pixels de padding a ser adicionada nas bordas superior e inferior.
- padding_width: Quantidade de pixels de padding a ser adicionada nas bordas laterais.

Saída:

- Retorna uma nova imagem (array) com as dimensões aumentadas para comportar o padding. As novas áreas de padding são preenchidas com zeros (preto).

```python
def conv2d(img: np.ndarray, kernel: np.ndarray, padding=True) -> np.ndarray:
    # Dimensions
    k_height, k_width = kernel.shape
    img_height, img_width = img.shape

    # Create a padded version of the image to handle edges
    if padding:
        pad_height = k_height // 2
        pad_width = k_width // 2
        padded_img = add_padding(img, pad_height, pad_width)
    else:
        padded_img = img

    # Initialize an output image with zeros
    output = np.zeros((img_height, img_width), dtype=float)

    # Perform convolution
    for i_img in range(img_height):
        for j_img in range(img_width):
            for i_kernel in range(k_height):
                for j_kernel in range(k_width):
                    output[i_img, j_img] += padded_img[i_img + i_kernel, j_img + j_kernel] * kernel[i_kernel, j_kernel]

    output = np.clip(output, 0, 255)  # Ensure values are within the range [0, 255]
    return output.astype(np.uint8)
```

Figura 2: Imagem da implementação da função ‘conv2d’

Essa função realiza a operação de convolução 2D na imagem com um kernel fornecido. Ela serve para aplicar efeitos como suavização, detecção de bordas, entre outros.

Parâmetros:

- img: Array 2D representando a imagem de entrada.
- kernel: Array 2D que representa o filtro (ou kernel) que será aplicado à imagem.
- padding: Indica se o padding será aplicado antes da convolução (padrão: True).

Saída:

- Retorna a imagem resultante após aplicar a convolução, com valores normalizados e ajustados para o intervalo de [0, 255].

A função primeiro adiciona padding à imagem (caso padding=True). Depois, percorre a imagem e, para cada pixel, aplica o kernel (ou seja, multiplica os valores da janela local da imagem pelos valores correspondentes do kernel e soma esses produtos). O resultado da convolução é armazenado em um novo array.
O valor resultante é limitado para ficar no intervalo de [0, 255], o que representa o intervalo típico de intensidade de pixels em imagens em tons de cinza.

```python
def conv2d_sharpening(img: np.ndarray, kernel: np.ndarray, padding=True) -> np.ndarray:
    # Dimensions
    k_height, k_width = kernel.shape
    img_height, img_width = img.shape

    # Create a padded version of the image to handle edges
    if padding:
        pad_height = k_height // 2
        pad_width = k_width // 2
        padded_img = add_padding(img, pad_height, pad_width)
    else:
        padded_img = img

    # Initialize an output image with zeros
    output = np.zeros((img_height, img_width), dtype=float)

    # Perform convolution
    for i_img in range(img_height):
        for j_img in range(img_width):
            for i_kernel in range(k_height):
                for j_kernel in range(k_width):
                    output[i_img, j_img] += padded_img[i_img + i_kernel, j_img + j_kernel] * kernel[i_kernel, j_kernel]

    output = np.clip(output, 0, 255)  # Ensure values are within the range [0, 255]
    return output.astype(np.uint8)
```

Figura 3: Imagem da implementação da função ‘conv2d_sharpening’

Essa função é uma versão bem parecida da conv2d, mas ela é usada especificamente para operações de "sharpening". Isso implica que, ao invés de usar qualquer kernel, essa função pode ser usada com kernels específicos para aumentar o contraste ou destacar bordas da imagem.

Parâmetros:

- img: Array 2D representando a imagem de entrada.
- kernel: Array 2D que representa o filtro (ou kernel) que será aplicado à imagem.
- padding: Indica se o padding será aplicado antes da convolução (padrão: True).

Saída:

- Retorna a imagem após a convolução, com a expectativa de que o kernel usado tenha efeito de nitidez.

```python
def conv2d_with_lookup_table(img: np.ndarray, lookup_table: np.ndarray, padding=True, k_height=3, k_width=3) -> np.ndarray:
    if padding:
        padded_img = add_padding(img, k_height // 2, k_width // 2)
    else:
        padded_img = img

    img_height, img_width = img.shape

    # Initialize an output image with zeros
    output = np.zeros((img_height, img_width), dtype=float)

    # Perform convolution
    for i_img in range(img_height):
        for j_img in range(img_width):
            for i_kernel in range(k_height):
                for j_kernel in range(k_width):
                    p = padded_img[i_img + i_kernel, j_img + j_kernel]
                    output[i_img, j_img] += p * lookup_table[int(p)]

    output = np.clip(output, 0, 255)
    return output.astype(np.uint8)
```

Figura 4: Imagem da implementação da função ‘conv2d_with_lookup_table’

Essa função é similar à conv2d, mas em vez de aplicar um kernel diretamente, ela usa uma tabela de consulta para multiplicar cada valor de pixel da imagem. A tabela de consulta contém valores pré-calculados que são usados para modificar cada pixel durante a operação de convolução.

Parâmetros:

- img: Array 2D representando a imagem.
- lookup_table: Array 1D onde o valor de cada pixel é multiplicado pelo valor correspondente da tabela.
- padding: Indica se o padding será aplicado (padrão: True).
- k_height: Altura do kernel (padrão: 3).
- k_width: Largura do kernel (padrão: 3).

Saída:

- Retorna a imagem após a aplicação da convolução usando a tabela de lookup.

Ao invés de multiplicar diretamente pelos valores do kernel, a função consulta a tabela lookup_table com base no valor do pixel atual e usa o valor da tabela para fazer a convolução.

```python
def gauss_create(sigma=1, size_x=3, size_y=3) -> np.ndarray:
    '''
    Create normal (gaussian) distribuiton
    '''
    x, y = np.meshgrid(np.linspace(-1,1,size_x), np.linspace(-1,1,size_y))
    calc = 1/((2*np.pi*(sigma**2)))
    exp = np.exp(-(((x**2) + (y**2))/(2*(sigma**2))))

    return exp*calc
```

Figura 5: Imagem da implementação da função ‘gauss_create’’

Essa função gera o filtro Gaussiano, que serve para suavizar ou desfocar a imagem.

Parâmetros:

- sigma: Desvio padrão da distribuição Gaussiana (quanto maior, mais desfocada a imagem resultante).
- size_x: Tamanho do kernel no eixo X.
- size_y: Tamanho do kernel no eixo Y.

Saída:

- Retorna um kernel Gaussiano (matriz 2D) de tamanho size_x x size_y.

A função cria uma grade de coordenadas (x, y) e calcula os valores do filtro Gaussiano usando a fórmula da distribuição Gaussiana bidimensional. A função retorna a matriz normalizada que será utilizada nos filtros de convolução.

```python
def gauss_filter(img : np.ndarray, padding=True) -> np.ndarray:
    gaus_3x3 = gauss_create(sigma=1, size_x=3, size_y=3)

    return conv2d(img = img, kernel = gaus_3x3, padding=padding)
```

Figura 6: Imagem da implementação da função ‘gauss_filter’

Essa função aplica o filtro Gaussiano gerado pela gauss_create a uma imagem.

Parâmetros:

- img: Array 2D representando a imagem.
- padding: Indica se o padding será aplicado à imagem antes da convolução (padrão: True).

Saída:

- Retorna a imagem suavizada pelo filtro Gaussiano.

A função primeiro cria um kernel Gaussiano 3x3 usando a função gauss_create. Em seguida, aplica a convolução 2D à imagem usando o filtro Gaussiano, por meio da função conv2d.

2.2 Implementação do Filtro Passa-baixa ideal

```python
import numpy as np
import cv2

IDEAL_TYPE = 0
GAUSS_TYPE = 1

def create_low_pass_filter(shape, center, radius, lpType=GAUSS_TYPE, n=2) -> np.ndarray:
    rows, cols = shape[:2]
    r, c = np.mgrid[0:rows:1, 0:cols:1]
    c -= center[0]
    r -= center[1]
    d = np.sqrt(np.power(c, 2.0) + np.power(r, 2.0))  # distância ao centro
    lpFilter = np.zeros((rows, cols), np.float32)

    if lpType == IDEAL_TYPE:  # Ideal low-pass filter
        lpFilter[d <= radius] = 1
    elif lpType == GAUSS_TYPE:  # Gaussian low-pass filter
        lpFilter = np.exp(-d**2 / (2 * (radius**2)))

    # Retorna o filtro em um formato de dois canais (para imagem complexa)
    lpFilter_matrix = np.zeros((rows, cols, 2), np.float32)
    lpFilter_matrix[:, :, 0] = lpFilter
    lpFilter_matrix[:, :, 1] = lpFilter

    return lpFilter_matrix
```

Figura 7: Imagem da implementação da função ‘create_low_pass_filter’’

IDEAL_TYPE: Constante para representar o filtro passa-baixa ideal.
GAUSS_TYPE: Constante para representar o filtro passa-baixa gaussiano.

Essa função cria um filtro passa-baixa, que é usado para suavizar a imagem, removendo detalhes de alta frequência.

Parâmetros:

- shape: O formato da imagem (número de linhas e colunas).
- center: As coordenadas do centro do filtro, normalmente o ponto médio da imagem.
- radius: O raio de corte do filtro (região afetada pelo filtro).
- lpType: O tipo de filtro passa-baixa a ser aplicado (ideal ou gaussiano).

- n: Um parâmetro opcional para configurar outros tipos de filtros, mas não é usado neste caso.

Operações:

1. Criação da malha de coordenadas: Cria dois arrays (r e c) que representam as coordenadas de cada ponto da imagem em relação ao centro especificado.

2. Distância ao centro: Calcula a distância euclidiana de cada ponto em relação ao centro da imagem.

3. Criação do filtro:

    - Para o filtro ideal, o valor é 1 para os pontos dentro do raio e 0 fora dele.
    - Para o filtro gaussiano, os valores diminuem suavemente conforme a distância ao centro aumenta, aplicando uma função exponencial.

4. Formato complexo: O filtro é convertido para uma matriz com dois canais, necessária para ser usada em uma transformada de Fourier bidimensional.

```python
def low_pass_filter(img: np.ndarray, radius=60, lpType=GAUSS_TYPE) -> np.ndarray:
    # Dimensões da imagem
    rows, cols = img.shape
    crow, ccol = rows // 2, cols // 2

    # Transformada de Fourier (DFT)
    image_f32 = np.float32(img)
    dft = cv2.dft(image_f32, flags=cv2.DFT_COMPLEX_OUTPUT) # type: ignore

    # Shift do centro da DFT
    dft_shift = np.fft.fftshift(dft)

    # Criar filtro passa-baixa e aplicar
    mask = create_low_pass_filter(dft_shift.shape[:2], center=(ccol, crow), radius=radius, lpType=lpType)

    # Aplicar a máscara de passa-baixa
    fshift = dft_shift * mask

    # Shift inverso e DFT inversa
    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])

    # Normalização da imagem resultante
    filtered_img = np.abs(img_back)
    filtered_img -= filtered_img.min()
    filtered_img = (filtered_img * 255) / filtered_img.max()
    filtered_img = filtered_img.astype(np.uint8)

    return filtered_img
```

Figura 8: Imagem da implementação da função ‘low_pass_filter’

Parâmetros:

- img: A imagem de entrada (em escala de cinza).
- radius: O raio do filtro passa-baixa.
- lpType: O tipo de filtro passa-baixa (ideal ou gaussiano).

Operações:

1. Dimensões da imagem: As dimensões da imagem são obtidas para calcular o centro e aplicar a transformada de Fourier corretamente.

2. Transformada de Fourier:

    - A imagem é convertida para um formato de ponto flutuante (float32) e a transformada de Fourier é aplicada usando a função cv2.dft, que gera uma versão da imagem no domínio da frequência (com valores complexos).

3. Deslocamento do centro da DFT:

    - A função np.fft.fftshift é usada para mover as frequências baixas para o centro da imagem transformada, o que facilita a aplicação do filtro.

4. Aplicação do filtro:

    - Um filtro passa-baixa é gerado usando a função create_low_pass_filter, com base nas dimensões da imagem transformada e no tipo de filtro escolhido.

    - O filtro é então aplicado à imagem no domínio da frequência.

5. Transformada inversa:

    - A transformada inversa de Fourier é realizada usando cv2.idft para trazer a imagem de volta do domínio da frequência para o domínio espacial.

    - A magnitude da imagem resultante é calculada usando cv2.magnitude.

6. Normalização:

    - A imagem resultante é normalizada para garantir que os valores estejam no intervalo correto de 0 a 255.

    - A normalização é feita subtraindo o valor mínimo, multiplicando por 255 e dividindo pelo valor máximo.

7. Retorno: A imagem filtrada é retornada como uma matriz do tipo uint8.

2.3 Implementação do Filtro Passa-alta ideal

```python
import numpy as np
import cv2

IDEAL_TYPE = 0
GAUSS_TYPE = 1

def create_high_pass_filter(shape, center, radius, lpType=GAUSS_TYPE, 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 = np.zeros(shape, np.float32)
    if lpType == IDEAL_TYPE:  # Ideal high pass filter
        lpFilter[d >= radius] = 1
    elif lpType == GAUSS_TYPE: # Gaussian Highpass Filter
        lpFilter = 1.0 - np.exp(-d**2 / (2 * (radius**2)))

    lpFilter_matrix = np.zeros((rows, cols, 2), np.float32)
    lpFilter_matrix[:, :, 0] = lpFilter
    lpFilter_matrix[:, :, 1] = lpFilter

    return lpFilter_matrix
```

Figura 9: Imagem da implementação da função ‘create_high_pass_filter’

IDEAL_TYPE: Constante para representar o filtro passa-baixa ideal.
GAUSS_TYPE: Constante para representar o filtro passa-baixa gaussiano.

Essa função cria uma matriz que representa um filtro passa-altas, que será aplicado no domínio da frequência da imagem.

Parâmetros:

- shape: Dimensão da imagem (número de linhas e colunas).
- center: Coordenadas do centro do filtro.
- radius: Raio do filtro (define a frequência de corte).
- lpType: Tipo de filtro (Ideal ou Gaussiano).
- n: Parâmetro opcional, não utilizado nesse caso.

Operações:

1. r, c = np.mgrid[0:rows:1, 0:cols:1]: Cria uma grade com as coordenadas (linhas e colunas) da imagem.
2. c -= center[0] e r -= center[1]: Desloca o centro das coordenadas para o centro da imagem.
3. d = np.power(c, 2.0) + np.power(r, 2.0): Calcula a distância ao centro da imagem.
4. Dependendo do lpType, o filtro passa-altas é criado:

    - Para o filtro Ideal, o valor é 1 fora do raio definido.
    - Para o filtro Gaussiano, a equação Gaussiana é aplicada para criar uma transição suave nas bordas do filtro.

5. A matriz lpFilter_matrix é preenchida com a versão complexa do filtro (dois canais), necessária para a Transformada de Fourier.

```python
def high_pass_filter(img: np.ndarray, radius=60, lpType=GAUSS_TYPE) -> np.ndarray:
    # Dimensões da imagem
    rows, cols = img.shape
    crow, ccol = rows // 2, cols // 2

    # Transformada de Fourier (DFT)
    image_f32 = np.float32(img)
    dft = cv2.dft(image_f32, flags=cv2.DFT_COMPLEX_OUTPUT) # type: ignore

    # Shift do centro da DFT
    dft_shift = np.fft.fftshift(dft)

    # Criar filtro passa-baixa e aplicar
    mask = create_high_pass_filter(dft_shift.shape[:2], center=(ccol, crow), radius=radius, lpType=lpType)

    # Aplicar a máscara de passa-baixa
    fshift = dft_shift * mask

    # Shift inverso e DFT inversa
    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])

    # Normalização da imagem resultante
    filtered_img = np.abs(img_back)
    filtered_img -= filtered_img.min()
    filtered_img = (filtered_img * 255) / filtered_img.max()
    filtered_img = filtered_img.astype(np.uint8)

    return filtered_img
```

Figura 10: Imagem da implementação da função ‘high_pass_filter’

Esta função aplica o filtro passa-alta à imagem no domínio da frequência.

Parâmetros:

- img: Imagem em escala de cinza a ser processada.
- radius: Raio de corte do filtro passa-altas.

- lpType: Tipo de filtro (Ideal ou Gaussiano).

Operações:

1. Transformada de Fourier (DFT):

    - A imagem é convertida para float32 para ser processada.
    - A função cv2.dft() realiza a DFT (Transformada Discreta de Fourier), convertendo a imagem do domínio espacial para o domínio da frequência.

2. Deslocamento do centro da DFT:

    - O espectro da DFT é centralizado utilizando np.fft.fftshift(), de forma que as frequências baixas fiquem no centro da imagem.

3. Criação do filtro passa-altas:

    - A função create_high_pass_filter é chamada para gerar o filtro passa-altas com base no tipo de filtro e no raio definido.

4. Aplicação do filtro:

    - A máscara de filtro passa-altas é multiplicada pelo espectro de Fourier da imagem.

5. Transformada Inversa de Fourier (IDFT):

    - O espectro filtrado é deslocado de volta utilizando np.fft.ifftshift().
    - A função cv2.idft() realiza a transformada inversa, convertendo o espectro filtrado de volta para o domínio espacial.

6. Magnitude e normalização:

- A magnitude da imagem resultante é calculada.

- A imagem é normalizada para o intervalo [0, 255] para poder ser exibida corretamente.

2.4 Implementação do Filtro de Sobel

```python
def sobel_sharpening(img : np.ndarray) -> np.ndarray:
    kernel_sobel_vertical = np.array(([-1,-2,-1],[0,0,0],[1,2,1]))
    kernel_sobel_horizontal = np.array(([-1,0,1],[-2,0,2],[-1,0,1]))

    img_sobel_1 = conv2d_sharpening(img, kernel_sobel_vertical)
    img_sobel_2 = conv2d_sharpening(img, kernel_sobel_horizontal)

    sobel = np.hypot(img_sobel_1, img_sobel_2)
    sobel = np.clip(sobel / np.max(sobel) * 255, 0, 255).astype(np.uint8)

    return sobel
```

Figura 11: Imagem da implementação da função ‘sobel_sharpening’

Essa função aplica o filtro Sobel a uma imagem para realçar suas bordas. A imagem de entrada é um array numpy bidimensional que representa uma imagem em tons de cinza.

Parâmetros:

- img: Uma imagem em escala de cinza representada como um array numpy.

Saída:

- Uma nova imagem onde as bordas foram realçadas.

kernel_sobel_vertical = np.array(([-1, -2, -1], [0, 0, 0], [1, 2, 1]))
kernel_sobel_horizontal = np.array(([-1, 0, 1], [-2, 0, 2], [-1, 0, 1]))

Essas duas matrizes, conhecidas por kernels ou máscaras, são responsáveis por detectar bordas na imagem em duas direções diferentes:

- kernel_sobel_vertical: Detecta bordas na direção vertical, ou seja, mudanças de intensidade ao longo das linhas da imagem.
- kernel_sobel_horizontal: Detecta bordas na direção horizontal, ou seja, mudanças de intensidade ao longo das colunas da imagem.

Esses kernels são aplicados à imagem para calcular o gradiente de intensidade de pixel em ambas as direções.

img_sobel_1 = conv2d_sharpening(img, kernel_sobel_vertical)
img_sobel_2 = conv2d_sharpening(img, kernel_sobel_horizontal)

A função conv2d_sharpening é responsável por aplicar a convolução da imagem com cada um dos kernels Sobel.

- img_sobel_1: Contém o resultado da convolução da imagem com o kernel vertical. Isso realça as bordas verticais da imagem.
- img_sobel_2: Contém o resultado da convolução da imagem com o kernel horizontal. Isso realça as bordas horizontais da imagem.

sobel = np.hypot(img_sobel_1, img_sobel_2)

A função np.hypot combina os resultados das duas convoluções anteriores (vertical e horizontal) utilizando a fórmula da hipotenusa.

Isso calcula a magnitude do gradiente em cada pixel, ou seja, a força e direção da borda. Assim, são combinadas as bordas detectadas nas direções vertical e horizontal para formar uma imagem final que contém as bordas da imagem original.

sobel = np.clip(sobel / np.max(sobel) * 255, 0, 255).astype(np.uint8)

Após calcular a magnitude do gradiente, o resultado pode ter valores que excedem o intervalo [0, 255] (que é o intervalo de intensidade de pixels para imagens de 8 bits). Portanto, é necessário normalizar a imagem:

- sobel / np.max(sobel) * 255: A magnitude do gradiente é dividida pelo valor máximo encontrado e escalada para que os valores fiquem no intervalo [0, 255].

- np.clip(..., 0, 255): Garante que nenhum valor ultrapasse o intervalo válido de intensidade.

- astype(np.uint8): Converte os valores para o formato inteiro de 8 bits (uint8), que é o formato padrão para imagens.

2.5 Implementação do Filtro de Canny

```python
def canny_edge_detection(img: np.ndarray, low_threshold: int, high_threshold: int, magnitude_scale: float = 1.0) -> np.ndarray:
    img_blurred = gauss_filter(img, padding=True)

    # Função sobel_sharpening para calcular a magnitude dos gradientes
    magnitude = sobel_sharpening(img_blurred)

    # Escalar a magnitude para ajustar a sensibilidade
    magnitude = np.clip(magnitude * magnitude_scale, 0, 255)

    # Cálculo dos gradientes para obter os ângulos
    kernel_sobel_vertical = np.array(([-1,-2,-1],[0,0,0],[1,2,1]))
    kernel_sobel_horizontal = np.array(([-1,0,1],[-2,0,2],[-1,0,1]))

    grad_x = conv2d_sharpening(img_blurred, kernel_sobel_horizontal)
    grad_y = conv2d_sharpening(img_blurred, kernel_sobel_vertical)

    angle = np.arctan2(grad_y, grad_x) * (180 / np.pi) % 180
```

Figura 12: Imagem da implementação da função ‘canny_edge_detection’

```python
    # Non-maximum Suppression
    nms = np.zeros_like(magnitude, dtype=np.uint8)
    img_height, img_width = img.shape
    for i in range(1, img_height - 1):
        for j in range(1, img_width - 1):
            region = magnitude[i-1:i+2, j-1:j+2]
            angle_deg = angle[i, j]

            if (0 <= angle_deg < 22.5) or (157.5 <= angle_deg <= 180):
                neighbors = [region[1, 0], region[1, 2]]
            elif 22.5 <= angle_deg < 67.5:
                neighbors = [region[0, 2], region[2, 0]]
            elif 67.5 <= angle_deg < 112.5:
                neighbors = [region[0, 1], region[2, 1]]
            else:
                neighbors = [region[0, 0], region[2, 2]]

            if magnitude[i, j] >= max(neighbors):
                nms[i, j] = magnitude[i, j]

    # Double Threshold
    strong_edges = (nms > high_threshold).astype(np.uint8)
    weak_edges = ((nms >= low_threshold) & (nms <= high_threshold)).astype(np.uint8)

    # Hysteresis
    edges = np.zeros_like(img, dtype=np.uint8)
    strong_i, strong_j = np.where(strong_edges == 1)
    weak_i, weak_j = np.where(weak_edges == 1)

    edges[strong_i, strong_j] = 255
```

Figura 13: Imagem continuação da implementação da função ‘canny_edge_detection’

```
    for i, j in zip(weak_i, weak_j):
        if np.any(strong_edges[i-1:i+2, j-1:j+2]):
            edges[i, j] = 255

    return edges
```

Figura 14: Imagem continuação da implementação da função ‘canny_edge_detection’

Este código implementa o algoritmo de Detecção de Bordas de Canny, uma técnica amplamente usada para detectar contornos em imagens. O processo envolve várias etapas, como suavização da imagem, cálculo de gradientes, supressão não máxima, limiares duplos e rastreamento por histerese.

Parâmetros:

- img: A imagem de entrada em formato de matriz numpy.
- low_threshold e high_threshold: Os limiares baixo e alto usados na etapa de threshold duplo.
- magnitude_scale: Um fator opcional para escalar a magnitude dos gradientes.

Operações:

1. Suavização com Filtro Gaussiano:

   img_blurred = gauss_filter(img, padding=True)

   - A imagem de entrada é suavizada com um filtro Gaussiano para reduzir o ruído e facilitar a detecção de bordas reais. A função gauss_filter realiza essa operação aplicando convolução com um kernel Gaussiano.

2. Cálculo da Magnitude dos Gradientes:

   magnitude = sobel_sharpening(img_blurred)
   magnitude = np.clip(magnitude * magnitude_scale, 0, 255)

   - O operador Sobel é utilizado (via sobel_sharpening) para calcular os gradientes verticais e horizontais da imagem, que representam as bordas.
   - A magnitude desses gradientes é calculada e escalada por magnitude_scale para ajustar a sensibilidade da detecção de bordas.
   - np.clip limita os valores para que se mantenham entre 0 e 255 (escala de níveis de cinza).

3. Cálculo dos Gradientes e Ângulos:

```
grad_x = conv2d_sharpening(img_blurred, kernel_sobel_horizontal)
grad_y = conv2d_sharpening(img_blurred, kernel_sobel_vertical)
angle = np.arctan2(grad_y, grad_x) * (180 / np.pi) % 180
```

- grad_x e grad_y são os gradientes da imagem nas direções horizontal e vertical, obtidos aplicando o kernel de Sobel.
- A função np.arctan2(grad_y, grad_x) é usada para calcular o ângulo de inclinação das bordas em cada pixel, que é então convertido para graus (entre 0° e 180°).

4. Supressão Não Máxima (Non-Maximum Suppression):

- Supressão Não Máxima (NMS) é aplicada para afinar as bordas. Apenas os pixels que são máximos locais ao longo da direção do gradiente são mantidos. Isso ajuda a eliminar falsos positivos.
- Cada pixel é comparado com seus vizinhos ao longo da direção do ângulo calculado, e somente os picos (valores maiores que seus vizinhos) são mantidos.

5. Threshold Duplo:

```
strong_edges = (nms > high_threshold).astype(np.uint8)
weak_edges = ((nms >= low_threshold) & (nms <= high_threshold)).astype(np.uint8)
```

- A imagem resultante da NMS é submetida a um threshold duplo.
- Fortes bordas são definidas como os pixels cujos valores estão acima do high_threshold.
- Fracas bordas são os pixels cujos valores estão entre o low_threshold e o high_threshold.

6. Rastreio por Histerese:

- Rastreio por histerese conecta as bordas fracas aos pixels de bordas fortes, garantindo a continuidade das bordas.
- Se um pixel de borda fraca estiver conectado a uma borda forte, ele é promovido a uma borda forte (valor 255). Caso contrário, é descartado.

2.6 Implementação das Métricas

```python
def calculate_mse(original: np.ndarray, compressed: np.ndarray) -> float:
    err = np.sum((original.astype(float) - compressed.astype(float)) ** 2)
    err /= float(original.shape[0] * original.shape[1])
    return err
```

Figura 15: Imagem da implementação da função ‘calculate_mse’

Essa é a função do Erro Médio Quadrático (MSE) entre duas imagens, que é uma medida do quanto as duas imagens diferem pixel a pixel. Quanto menor o valor, mais parecidas as imagens são.

Operações:

1. Subtração das imagens: A diferença entre os pixels correspondentes da imagem original e da imagem comprimida é calculada.
    - original.astype(float) - compressed.astype(float): As imagens são convertidas para o tipo float para garantir que os valores de pixels não sejam truncados durante a operação de subtração.
2. Elevação ao quadrado: O quadrado da diferença é calculado para que tanto as diferenças positivas quanto negativas sejam tratadas igualmente.
    - (... ** 2): Cada diferença é elevada ao quadrado.
3. Soma de todos os erros: Todos os erros ao quadrado são somados.
    - np.sum(...): Calcula a soma de todas as diferenças quadráticas entre os pixels.
4. Normalização: O erro total é dividido pelo número total de pixels na imagem para obter a média do erro quadrático.
    - err /= float(original.shape[0] * original.shape[1]): Isso divide o erro acumulado pelo número total de pixels (altura vezes largura da imagem).
5. Retorno: A função retorna o valor do MSE. Valores mais baixos indicam maior similaridade entre as imagens.

```python
def calculate_rmse(original: np.ndarray, compressed: np.ndarray) -> float:
    err = calculate_mse(original, compressed)
    return np.sqrt(err)
```

Figura 16: Imagem da implementação da função ‘calculate_rmse’

Essa função calcula o Erro Médio Quadrático da Raiz (RMSE), que é a raiz quadrada do MSE. O RMSE dá uma métrica mais intuitiva e na mesma unidade dos valores de pixel, sendo útil para entender a magnitude do erro. Valores mais baixos indicam imagens mais similares, mas o RMSE é mais interpretável pois está na mesma unidade dos valores de pixel.

Operações:

1. Chamada para MSE: A função primeiro chama a calculate_mse para obter o MSE entre as duas imagens.
    - err = calculate_mse(original, compressed): Calcula o MSE.
2. Raiz quadrada: O RMSE é então calculado como a raiz quadrada do MSE.
    - np.sqrt(err): Calcula a raiz quadrada do MSE.
3. Retorno: O RMSE é retornado, dando uma noção da média do erro em termos absolutos (na escala dos valores dos pixels).

```python
def calculate_psnr(original: np.ndarray, compressed: np.ndarray) -> float:
    mse = calculate_mse(original, compressed)
    if mse == 0: # imagens iguais
        return float('inf') # PSNR infinito
    max_pixel = 255.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return float(psnr)
```

Figura 17: Imagem da implementação da função ‘calculate_psnr’

Essa função calcula a Razão de Sinal para Ruído de Pico (PSNR), que é uma métrica usada para medir a qualidade de reconstrução de uma imagem. O PSNR compara o valor máximo possível de um pixel (neste caso, 255 para imagens de 8 bits) com o ruído gerado pela compressão ou transformação da imagem. Valores maiores indicam imagens de melhor qualidade.

Operações:

1. Chamada para MSE: Primeiro, a função calcula o MSE entre a imagem original e a imagem comprimida.

    - mse = calculate_mse(original, compressed): Calcula o MSE.

2. Verificação de MSE: Se o MSE for 0, significa que as imagens são idênticas, então o PSNR é infinito.

    - if mse == 0: Verifica se as imagens são idênticas.
    - return float('inf'): Retorna infinito se o MSE for zero.

3. Cálculo do PSNR:

    - O max_pixell é o valor máximo possível de um pixel (255 para imagens de 8 bits) e o MSE é o erro calculado. - max_pixel = 255.0: Define o valor máximo possível para um pixel. - psnr = 20 * np.log10(max_pixel / np.sqrt(mse)): Aplica a fórmula para calcular o PSNR.

3. RESULTADOS

Para a execução dos filtros, foi escolhida a imagem exibida na Figura 18. É uma imagem cujo tipo de arquivo é PNG e possui a resolução 512x512.

Figura 18: Imagem referência

3.1 Esmaecimento

Foram utilizados filtros de suavização e esmaecimento tanto no domínio espacial quanto no domínio da frequência para fins de comparação. No domínio espacial, foi aplicado o filtro Gaussiano, enquanto no domínio da frequência foram usados os filtros passa-baixa Ideal e Gaussiano. A execução do filtro Gaussiano no domínio espacial levou 970 milissegundos, resultando na imagem exibida na Figura 19.

Figura 19: Imagem com filtro de esmaecimento gaussiano

No domínio da frequência, o filtro passa-baixa Ideal teve um tempo de execução de 24 milissegundos, enquanto o filtro Gaussiano foi ainda mais rápido, com um tempo de 15 milissegundos. Os resultados dessas operações são mostrados nas Figuras 20 e 21, respectivamente.

Figura 20: Imagem com filtro passa-baixa ideal

Figura 21: Imagem com filtro passa-baixa gaussiano

Observa-se o fenômeno de *ringing* na imagem filtrada com o passa-baixa Ideal, mesmo com a aplicação única desse filtro, enquanto o filtro Gaussiano apresentou um resultado mais suave e contínuo.

3.1.1 Métricas

Vide na figura 22 a tabela com as métricas resultantes da comparação da aplicação do filtro espacial de esmaecimento Gaussiano com os filtros de domínio da frequência passa-baixa Ideal e Gaussiana.

Métrica	Passa-baixa Ideal	Passa-baixa Gaussiana
PSNR	22.737952375533062	20.457128028836053
RMSE	18.605552094901768	24.19266874695418
MSE	346.1665687561035	585.2852210998535

Figura 22: Tabela para comparação do filtro Gaussiano com os filtros passa-baixa Ideal e Gaussiano

Em termos de qualidade de imagem, o filtro passa-baixa Ideal apresenta o maior PSNR (22.74), indicando uma menor diferença em relação ao filtro espacial Gaussiano. No entanto, também apresenta o efeito ringing, uma distorção visual indesejada, o que pode afetar a percepção de suavidade.

O filtro passa-baixa Gaussiano, embora tenha um PSNR menor (20.46), resulta em uma suavização mais consistente e sem artefatos visuais perceptíveis, sendo mais adequado para manter a integridade da imagem. Quanto aos tempos de execução, os filtros no domínio da frequência são extremamente rápidos em comparação ao filtro espacial Gaussiano. Isso faz dos filtros de frequência escolhas mais eficientes para aplicações em tempo real, onde a rapidez é crucial, especialmente o filtro passa-baixa Gaussiano, que oferece um bom equilíbrio entre qualidade de suavização e desempenho.

Portanto, embora o filtro passa-baixa Ideal ofereça uma alta métrica de PSNR, sua distorção visual o torna menos adequado em situações onde a qualidade perceptual é importante. O filtro passa-baixa Gaussiano, por outro lado, se destaca como uma alternativa eficiente e com melhor suavidade visual, além de ser o mais rápido entre os três.

3.2 Destaque de bordas

A mesma imagem utilizada para os filtros de esmaecimento foi utilizada para os filtros de realce de bordas. O filtro espacial escolhido para a comparação foi o Sobel, enquanto os filtros de domínio da frequência foram o passa-alta Ideal e o passa-alta Gaussiano. Cada um deles será comparado com o filtro Canny. A imagem com o filtro de referência aplicado pode ser visualizada na Figura 23.

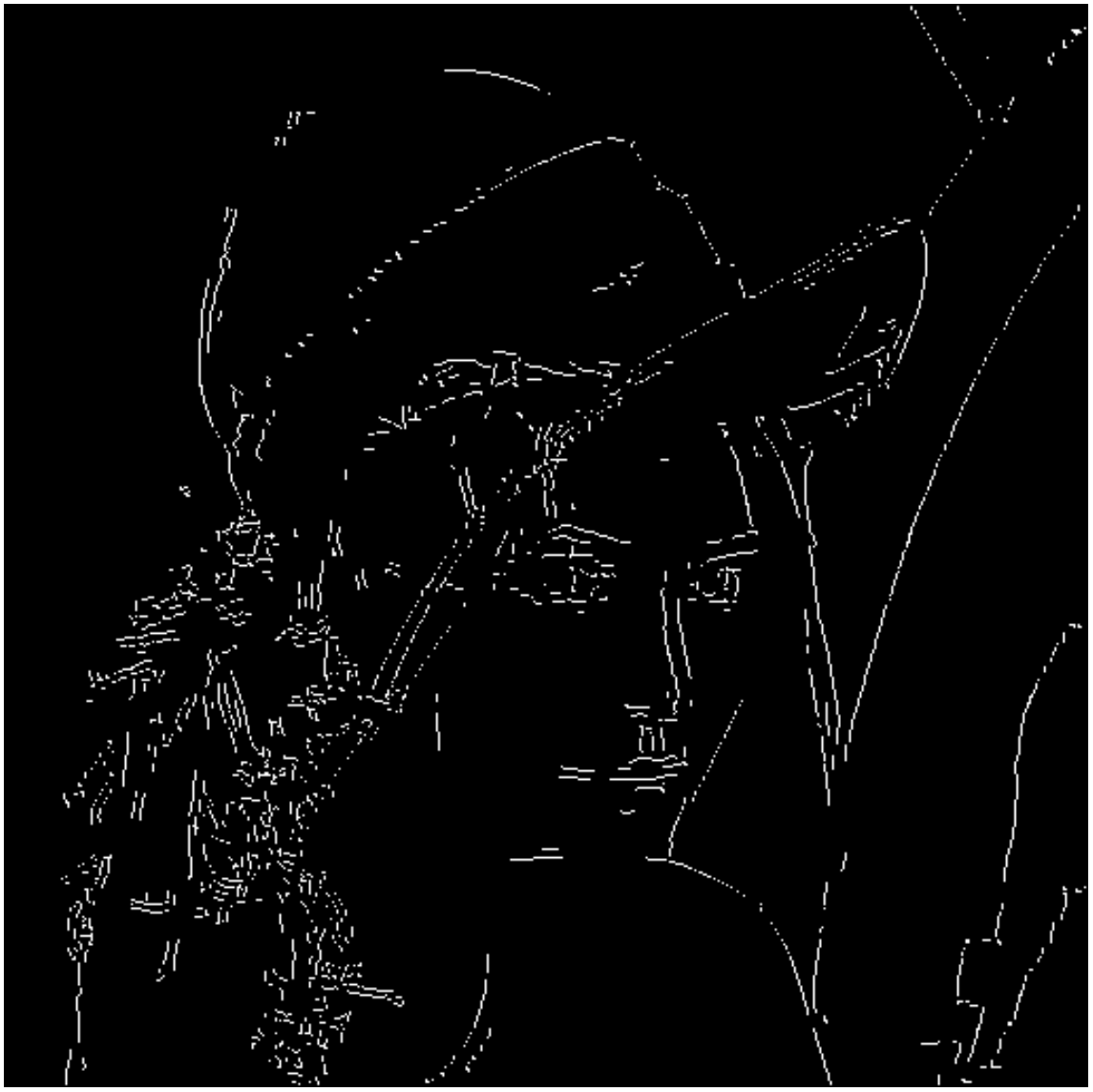

Figura 23: Imagem com filtro Canny

O filtro Sobel executou em 1,99 segundos e apresentou o resultado exibido na Figura 24.

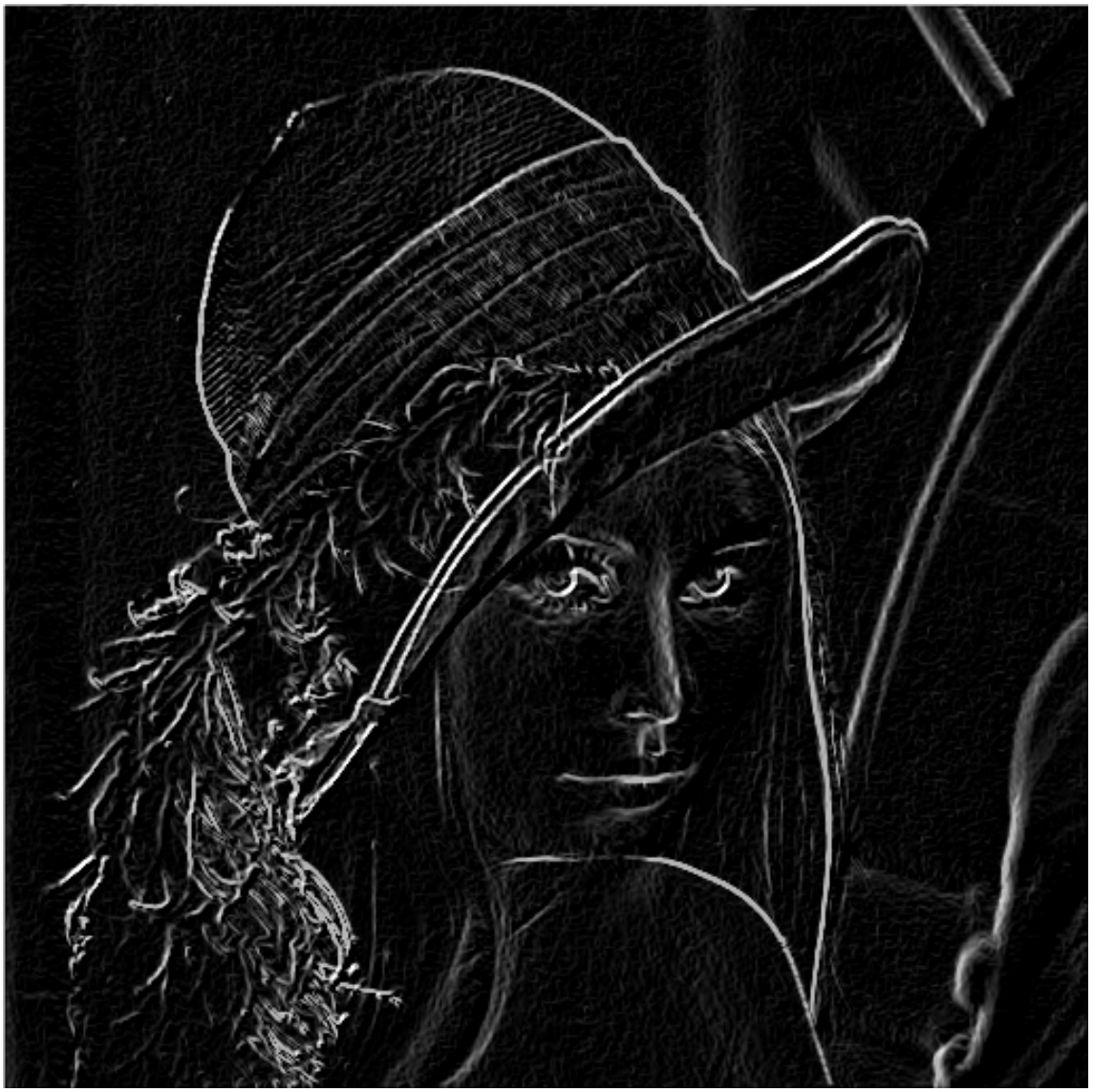

Figura 24: Imagem com filtro Sobel

No domínio da frequência, o filtro passa-alta Ideal teve um tempo de execução de 26 milissegundos, enquanto o filtro Gaussiano foi ainda mais rápido, com um tempo de 21 milissegundos. Os resultados dessas operações são mostrados nas Figuras 25 e 26, respectivamente.

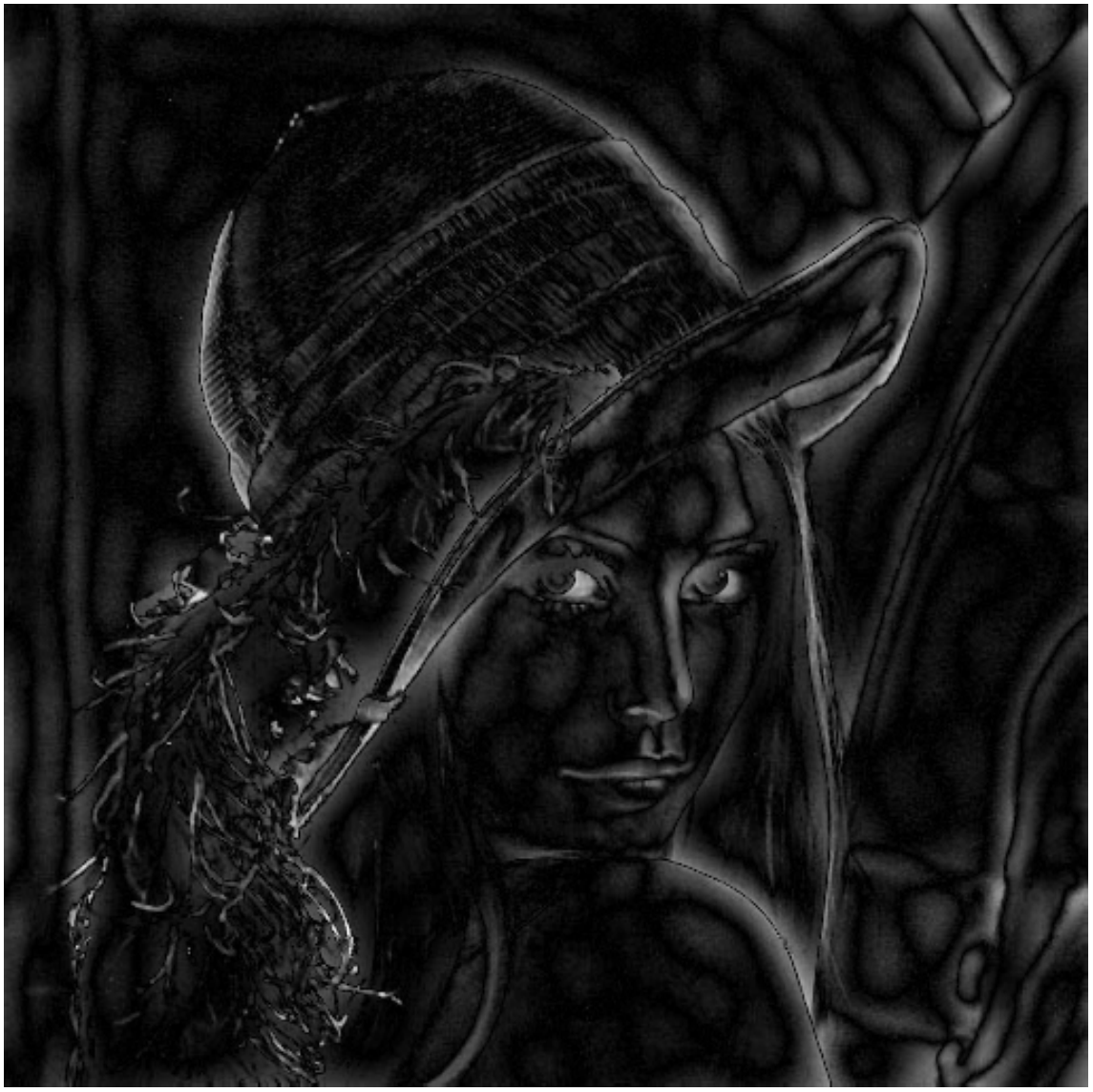

Figura 25: Imagem com filtro passa-alta Ideal

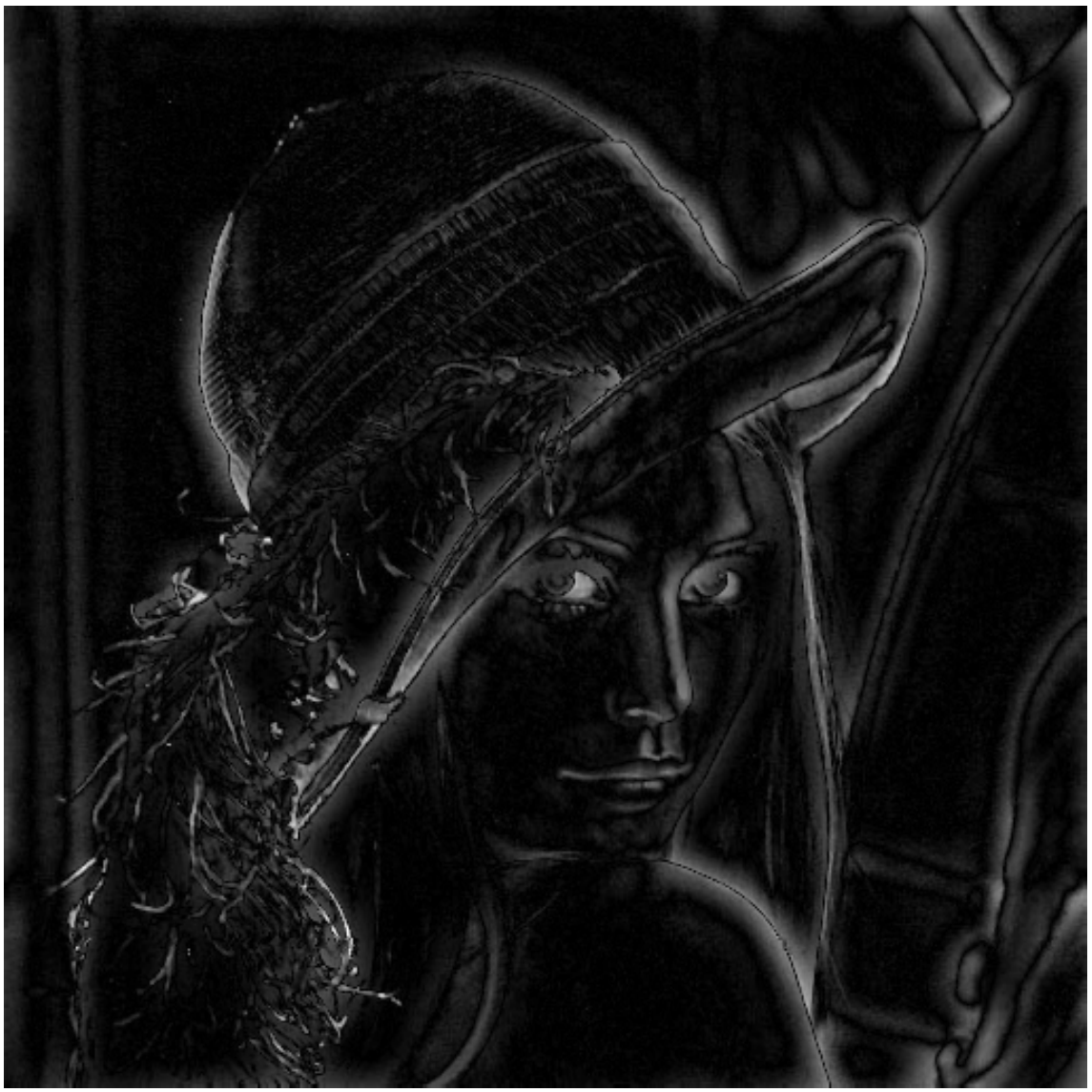

Figura 26: Imagem com filtro passa-alta Gaussiana

Visualmente, é possível apontar que o Sobel apresenta um resultado mais bruto, com bordas serrilhadas, enquanto os filtros de domínio da frequência tem bordas suaves. Também vale apontar o efeito de *ringing* inverso do passa-alta ideal, com manchas claras nas partes sem borda.

3.2.1 Métricas

Observe os valores que as métricas apresentaram na comparação dos filtros espacial e de domínio da frequência com o Canny na Figura 27.

Métrica	Sobel	Passa-alta Ideal	Passa-alta Gaussiana
PSNR	14.63904908613	14.69604367439858	15.044883036513976
RMSE	47.27023116437	46.96107140958126	45.11240999630865
MSE	2234.474754333	2205.342227935791	2035.1295356750488

Figura 27: Tabela para comparação do filtro Sobel com os filtros passa-alta Ideal e Gaussiano

Em termos de PSNR, o filtro passa-alta Gaussiano apresenta o maior valor (15.04), o que indica uma menor diferença em relação ao Canny, enquanto os filtros Sobel (14.64) e passa-alta Ideal (14.70) têm valores semelhantes, com o Sobel sendo ligeiramente pior em relação à precisão da borda. A MSE e RMSE seguem o mesmo padrão, mostrando que o passa-alta Gaussiano consegue um melhor equilíbrio entre a preservação de detalhes e a suavização das bordas.

Nos tempos de execução, o Sobel se destaca por ser significativamente mais lento (1.99 segundos), enquanto os filtros de passa-alta são muito mais rápidos. O passa-alta Ideal mais rápido que o Sobel, e o Gaussiano é ainda mais eficiente. Essa diferença de performance torna os filtros de passa-alta mais indicados para aplicações que requerem processamento rápido, como sistemas em tempo real. Em resumo, o filtro passa-alta Gaussiano apresenta o melhor equilíbrio entre qualidade e velocidade, com um PSNR ligeiramente superior e execução rápida. O Sobel, embora seja um método robusto, é consideravelmente mais lento, o que pode limitar sua utilização em situações onde o tempo de processamento é crucial.

4. CONCLUSÃO

Com base nas comparações realizadas entre os diferentes filtros de imagem, tanto no domínio da frequência quanto no domínio espacial, é possível tirar conclusões valiosas sobre o desempenho de cada técnica. No caso da comparação entre o filtro Sobel e os filtros passa-alta Ideal e Gaussiano, utilizando o filtro Canny como referência, o filtro passa-alta Gaussiano apresentou o melhor equilíbrio entre qualidade e tempo de processamento. Seu valor de PSNR (15.04) foi o mais alto, indicando uma menor distorção em relação ao filtro de referência. Além disso, o filtro Gaussiano também foi o mais eficiente em termos de tempo de execução, o que o torna uma opção ideal para aplicações em tempo real. O Sobel, apesar de ser um método robusto, mostrou-se consideravelmente mais lento, o que limita sua aplicabilidade em cenários onde a rapidez é um fator determinante.

Já na comparação entre os filtros passa-baixa Ideal e Gaussiano, observou-se que o filtro passa-baixa Ideal obteve um valor superior de PSNR (22.74), indicando uma qualidade de imagem teoricamente melhor. No entanto, ele apresenta o efeito ringing, que acaba sendo uma distorção indesejada que afeta a suavidade visual da imagem. Por outro lado, o filtro passa-baixa Gaussiano, embora com um PSNR menor (20.46), demonstrou ser mais consistente, proporcionando uma suavização sem artefatos perceptíveis. Além disso, os filtros no domínio da frequência, em especial o passa-baixa Gaussiano, mostraram-se extremamente rápidos em comparação ao filtro espacial Gaussiano, reforçando sua eficiência para aplicações que exigem processamento em tempo real.

Portanto, é possível concluir que, para aplicações em que a qualidade de borda e a velocidade de processamento são igualmente importantes, o filtro passa-alta Gaussiano se destaca como a melhor opção. Por outro lado, para tarefas que envolvem a suavização de imagens sem comprometer a percepção visual, o filtro passa-baixa Gaussiano é o mais adequado, especialmente quando se considera a ausência de artefatos como o ringing. No geral, os filtros baseados no domínio da frequência, tanto para passa-baixa quanto para passa-alta, são os mais eficientes em termos de tempo de execução, tornando-os altamente recomendados para sistemas que exigem rapidez e precisão na filtragem de imagens.

5. REPOSITÓRIO

https://github.com/Ripdt/Spatial-VS-FreqDomain