Paulo Eduardo Borges do Vale
Questão 1.a Laplacian

Neste código, vamos usar a biblioteca OpenCV para realizar a detecção de bordas em uma imagem em escala de cinza. A detecção de bordas é um processo que identifica os pontos de uma imagem onde há uma mudança brusca de intensidade ou cor. As bordas podem ser usadas para realçar as características de uma imagem, como formas, contornos, objetos, etc.

Para detectar as bordas, vamos usar o filtro Laplaciano, que é um operador diferencial que calcula a segunda derivada da intensidade da imagem. O filtro Laplaciano é sensível a ruídos, mas pode produzir bordas finas e bem definidas.

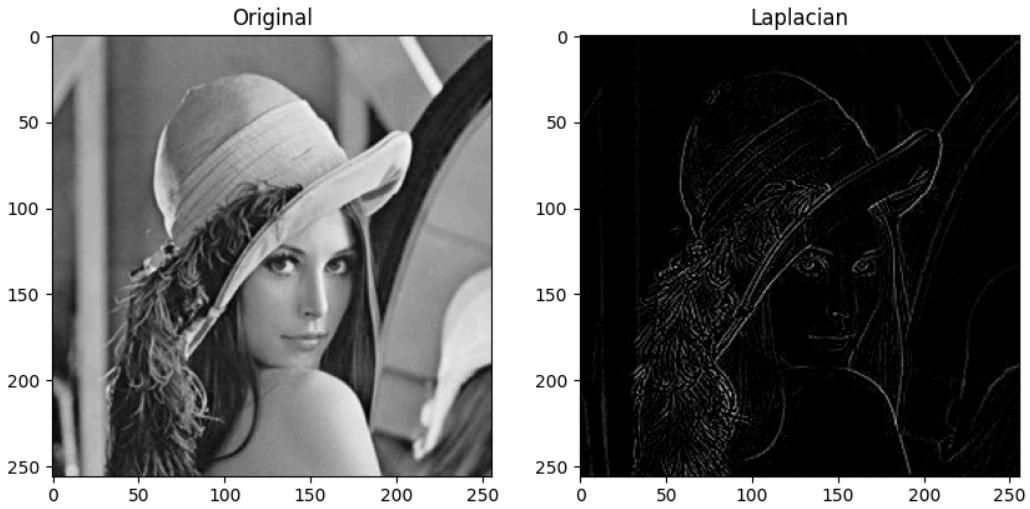
O código consiste nos seguintes passos:

- Importar as bibliotecas cv2 e matplotlib.pyplot, que são usadas para manipular imagens e plotar gráficos, respectivamente.
- Carregar a imagem “lena_gray.bmp” em escala de cinza, usando a função cv2.imread. A imagem é uma versão em tons de cinza da famosa imagem de Lena, que é usada frequentemente como exemplo em processamento de imagens.
- Aplicar o filtro Laplaciano na imagem, usando a função cv2.Laplacian. O resultado é uma imagem com as bordas destacadas em branco sobre um fundo preto.
- Mostrar a imagem original e a imagem com bordas detectadas usando a biblioteca matplotlib.pyplot. Para isso, usamos as funções plt.figure, plt.subplot, plt.imshow, plt.title e plt.show, que permitem criar uma figura com dois subplots, mostrar as imagens em escala de cinza, adicionar títulos e exibir o gráfico na tela.

```
import cv2
import matplotlib.pyplot as plt

# Carregar a imagem "lena_gray.bmp" em escala de cinza
img = cv2.imread("lena_gray.bmp", cv2.IMREAD_GRAYSCALE)

# Aplicar o filtro Laplaciano na imagem
edges = cv2.Laplacian(img, -1)

# Mostrar a imagem original e a imagem com bordas detectadas usando
Matplotlib
plt.figure(figsize=(10, 5))
```

```
plt.subplot(1, 2, 1)
plt.imshow(img, cmap='gray')
plt.title('Original')

plt.subplot(1, 2, 2)
plt.imshow(edges, cmap='gray')
plt.title('Laplacian')

plt.show()
```

Resultado



Questão 1.b Unsharp masking

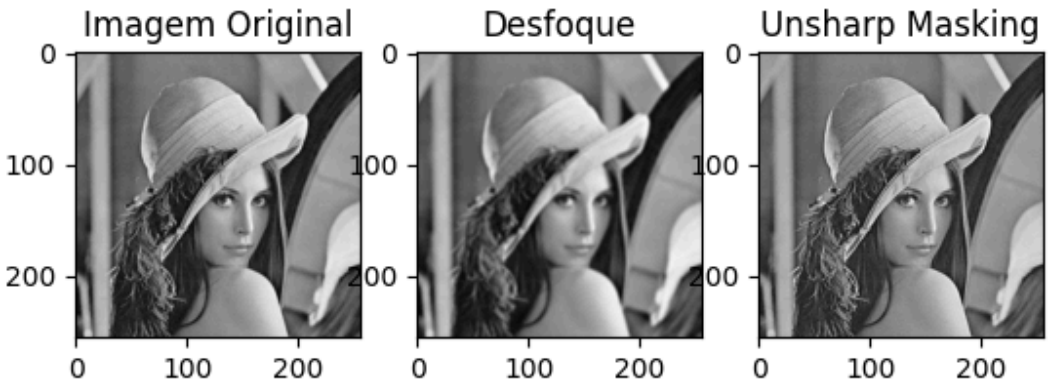
Neste código, vamos usar a biblioteca OpenCV para realizar o realce de uma imagem em escala de cinza. O realce de imagem é um processo que visa melhorar a qualidade visual de uma imagem, aumentando o contraste, a nitidez, a iluminação, etc.

Para realçar a imagem, vamos usar a técnica de Unsharp Masking, que consiste em subtrair uma versão desfocada da imagem original da própria imagem original, multiplicada por um fator de ganho. O resultado é uma imagem com as bordas e os detalhes mais evidentes.

O código consiste nos seguintes passos:

- Importar as bibliotecas cv2, numpy e matplotlib.pyplot, que são usadas para manipular imagens, realizar operações matemáticas e plotar gráficos, respectivamente.
- Carregar a imagem “lena_gray.bmp” em escala de cinza, usando a função cv2.imread. A imagem é uma versão em tons de cinza da famosa imagem de Lena, que é usada frequentemente como exemplo em processamento de imagens.
- Aplicar um desfoque à imagem original, usando a função cv2.GaussianBlur. O desfoque é um filtro que suaviza a imagem, reduzindo o ruído e as irregularidades. O desfoque é feito com um núcleo gaussiano de tamanho 5x5 e desvio padrão 0.
- Calcular a máscara de realce (Unsharp Mask), usando a função cv2.addWeighted. A máscara é obtida pela combinação linear da imagem original, multiplicada por 1.5, e da imagem desfocada, multiplicada por -0.5. O resultado é uma imagem com as diferenças entre a imagem original e a imagem desfocada, amplificadas pelo fator de ganho 1.5.
- Exibir a imagem original, o desfoque e o resultado do Unsharp Masking, usando a biblioteca matplotlib.pyplot. Para isso, usamos as funções plt.subplot, plt.imshow, plt.title e plt.show, que permitem criar uma figura com três subplots, mostrar as imagens em escala de cinza, adicionar títulos e exibir o gráfico na tela.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Carregando a imagem
imagem = cv2.imread('lena_gray.bmp', cv2.IMREAD_GRAYSCALE)

# Aplicando um desfoque à imagem original
desfoque = cv2.GaussianBlur(imagem, (5, 5), 0)

# Calculando a máscara de realce (Unsharp Mask)
mascara = cv2.addWeighted(imagem, 1.5, desfoque, -0.5, 0)

# Exibindo a imagem original, o desfoque e o resultado do Unsharp
Masking
plt.subplot(131), plt.imshow(imagem, cmap='gray'), plt.title('Imagem
Original')
plt.subplot(132), plt.imshow(desfoque, cmap='gray'),
plt.title('Desfoque')
plt.subplot(133), plt.imshow(mascara, cmap='gray'), plt.title('Unsharp
Masking')
plt.show()
```



Questão 1.c Filtragem Highboost

Neste código, vamos usar a biblioteca OpenCV para realizar a filtragem highboost em uma imagem em escala de cinza. A filtragem highboost é uma técnica que visa realçar as bordas e os detalhes de uma imagem, aumentando o contraste e a nitidez.

Para realizar a filtragem highboost, vamos usar os seguintes passos:

- Importar a biblioteca cv2, que é usada para manipular imagens.
- Carregar a imagem “lena_gray.bmp” em escala de cinza, usando a função cv2.imread. A imagem é uma versão em tons de cinza da famosa imagem de Lena, que é usada frequentemente como exemplo em processamento de imagens.
- Aplicar o filtro da média com tamanho 5x5 na imagem, usando a função cv2.blur. O filtro da média é um filtro que suaviza a imagem, reduzindo o ruído e as irregularidades. O filtro da média é feito com um núcleo retangular de tamanho 5x5, que calcula a média dos pixels vizinhos.
- Aplicar o filtro passa-alta subtraindo a imagem suavizada da imagem original, usando a função cv2.subtract. O filtro passa-alta é um filtro que destaca as bordas e os detalhes da imagem, realçando as diferenças de intensidade. O filtro passa-alta é obtido pela subtração da imagem original pela imagem suavizada, que é o inverso do filtro passa-baixa.

- Aplicar a filtragem highboost com fator 2, usando a função cv2.addWeighted. A filtragem highboost é uma combinação linear da imagem original, multiplicada por um fator de ganho maior que 1, e da imagem passa-alta, multiplicada por 1. O resultado é uma imagem com as bordas e os detalhes mais evidentes, amplificados pelo fator de ganho 2.
- Salvar a imagem highboost, usando a função cv2.imwrite. A imagem highboost é salva com o nome "lena_gray_highboost.bmp", no mesmo formato e na mesma pasta da imagem original.

```
import cv2

# Carregar a imagem em escala de cinza
imagem = cv2.imread("lena_gray.bmp", cv2.IMREAD_GRAYSCALE)

# Aplicar o filtro da média com tamanho 5x5
imagem_suavizada = cv2.blur(imagem, (5, 5))

# Aplicar o filtro passa-alta subtraindo a imagem suavizada da imagem
original
imagem_passa_alta = cv2.subtract(imagem, imagem_suavizada)

# Aplicar a filtragem highboost com fator 2
imagem_highboost = cv2.addWeighted(imagem, 2, imagem_passa_alta, 1, 0)

# Salvar a imagem highboost
cv2.imwrite("lena_gray_highboost.bmp", imagem_highboost)
```

Questão 1.d Prewitt

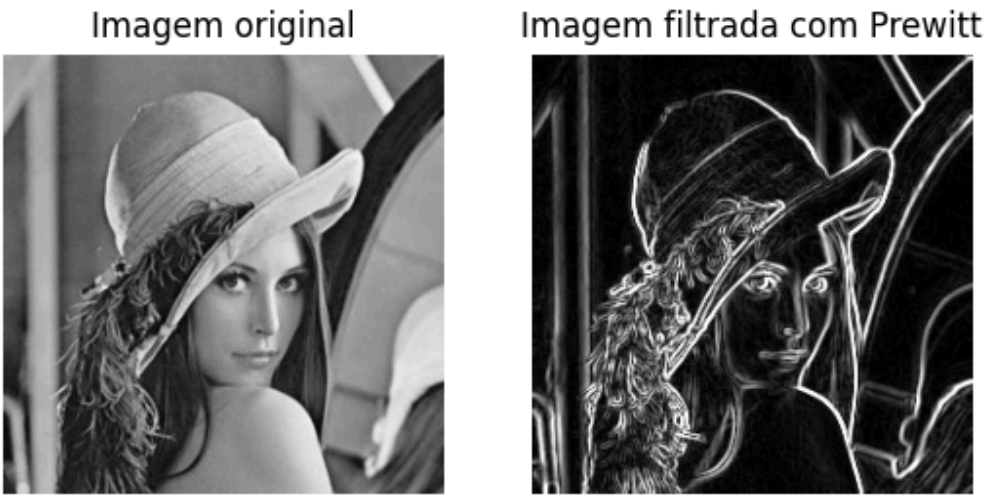
Neste código, vamos usar a biblioteca OpenCV para aplicar o operador Prewitt em uma imagem em escala de cinza. O operador Prewitt é um operador de detecção de bordas que calcula a primeira derivada da intensidade da imagem em duas direções ortogonais, horizontal e vertical. O operador Prewitt é sensível a ruídos, mas pode produzir bordas finas e bem definidas.

O código consiste nos seguintes passos:

- Importar as bibliotecas cv2, numpy e matplotlib.pyplot, que são usadas para manipular imagens, realizar operações matemáticas e plotar gráficos, respectivamente.
- Ler a imagem “lena_gray.bmp” em escala de cinza, usando a função cv2.imread. A imagem é uma versão em tons de cinza da famosa imagem de Lena, que é usada frequentemente como exemplo em processamento de imagens.
- Definir o kernel do operador Prewitt, que é uma matriz 3x3 que contém os coeficientes do operador. O kernel é definido como um array do tipo float32, usando a biblioteca numpy.
- Aplicar o filtro Prewitt na imagem, usando a função cv2.filter2D. Essa função realiza uma convolução da imagem com o kernel, gerando uma nova imagem com os valores do gradiente. O filtro Prewitt é aplicado na direção horizontal, usando o kernel original, e na direção vertical, usando a transposta do kernel, que é obtida com o método kernel.T.
- Calcular a magnitude do gradiente, usando a função cv2.magnitude. Essa função recebe as imagens com os gradientes horizontal e vertical, e retorna uma imagem com a raiz quadrada da soma dos quadrados dos gradientes. A magnitude do gradiente é uma medida da variação da intensidade da imagem, e indica a presença de bordas.
- Converter a imagem para o tipo uint8, usando a função cv2.convertScaleAbs. Essa função converte a imagem do tipo float64 para o tipo uint8, que é o formato mais comum para imagens em escala de cinza. A função também ajusta os valores da imagem para o intervalo entre 0 e 255, usando uma escala linear.
- Exibir a imagem original e a imagem filtrada, usando a biblioteca matplotlib.pyplot. Para isso, usamos as funções plt.subplot, plt.imshow, plt.title, plt.axis e plt.show, que permitem criar uma figura com dois subplots, mostrar as imagens em escala de cinza, adicionar títulos, remover os eixos e exibir o gráfico na tela.

```python
# Importar as bibliotecas necessárias
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Ler a imagem "lena_gray.bmp" em tons de cinza
img = cv2.imread("lena_gray.bmp", cv2.IMREAD_GRAYSCALE)
```

```
# Definir o kernel do operador Prewitt
kernel = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]],
dtype=np.float32)

# Aplicar o filtro Prewitt na imagem
prewitt_x = cv2.filter2D(img, cv2.CV_64F, kernel)
prewitt_y = cv2.filter2D(img, cv2.CV_64F, kernel.T) # transposta do
kernel

# Calcular a magnitude do gradiente
prewitt = cv2.magnitude(prewitt_x, prewitt_y)

# Converter a imagem para o tipo uint8
prewitt = cv2.convertScaleAbs(prewitt)

# Exibir a imagem original e a imagem filtrada
plt.subplot(121)
plt.imshow(img, cmap="gray")
plt.title("Imagem original")
plt.axis("off")
plt.subplot(122)
plt.imshow(prewitt, cmap="gray")
plt.title("Imagem filtrada com Prewitt")
plt.axis("off")
plt.show()
```



Questão 1.e Sobel

Neste código, vamos usar a biblioteca OpenCV para aplicar o operador Sobel em uma imagem em escala de cinza. O operador Sobel é um operador de detecção de bordas que calcula a primeira derivada da intensidade da imagem em duas direções ortogonais, horizontal e vertical. O operador Sobel é mais robusto a ruídos do que o operador Prewitt, pois usa pesos maiores para os pixels centrais.

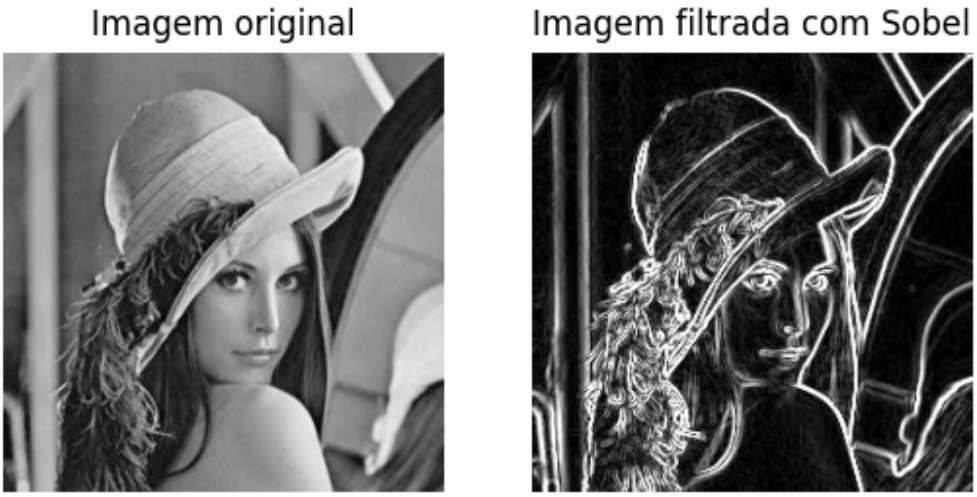
O código consiste nos seguintes passos:

- Importar as bibliotecas cv2, numpy e matplotlib.pyplot, que são usadas para manipular imagens, realizar operações matemáticas e plotar gráficos, respectivamente.
- Ler a imagem “lena_gray.bmp” em escala de cinza, usando a função cv2.imread. A imagem é uma versão em tons de cinza da famosa imagem de Lena, que é usada frequentemente como exemplo em processamento de imagens.
- Definir os kernels do operador Sobel, que são duas matrizes 3x3 que contêm os coeficientes do operador. Os kernels são definidos como arrays do tipo float32, usando a biblioteca numpy. O kernel_x é usado para a direção horizontal, e o kernel_y é usado para a direção vertical.
- Aplicar o filtro Sobel na imagem, usando a função cv2.filter2D. Essa função realiza uma convolução da imagem com os kernels, gerando duas novas imagens com os valores do gradiente. O filtro Sobel é aplicado na direção horizontal, usando o kernel_x, e na direção vertical, usando o kernel_y.

- Calcular a magnitude do gradiente, usando a função cv2.magnitude. Essa função recebe as imagens com os gradientes horizontal e vertical, e retorna uma imagem com a raiz quadrada da soma dos quadrados dos gradientes. A magnitude do gradiente é uma medida da variação da intensidade da imagem, e indica a presença de bordas.
- Converter a imagem para o tipo uint8, usando a função cv2.convertScaleAbs. Essa função converte a imagem do tipo float64 para o tipo uint8, que é o formato mais comum para imagens em escala de cinza. A função também ajusta os valores da imagem para o intervalo entre 0 e 255, usando uma escala linear.
- Exibir a imagem original e a imagem filtrada, usando a biblioteca matplotlib.pyplot. Para isso, usamos as funções plt.subplot, plt.imshow, plt.title, plt.axis e plt.show, que permitem criar uma figura com dois subplots, mostrar as imagens em escala de cinza, adicionar títulos, remover os eixos e exibir o gráfico na tela.

```
# Importar as bibliotecas necessárias
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Ler a imagem "lena_gray.bmp" em tons de cinza
img = cv2.imread("lena_gray.bmp", cv2.IMREAD_GRAYSCALE)

# Definir o kernel do operador Sobel
kernel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]],
dtype=np.float32)
kernel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]],
dtype=np.float32)

# Aplicar o filtro Sobel na imagem
sobel_x = cv2.filter2D(img, cv2.CV_64F, kernel_x)
sobel_y = cv2.filter2D(img, cv2.CV_64F, kernel_y)

# Calcular a magnitude do gradiente
sobel = cv2.magnitude(sobel_x, sobel_y)

# Converter a imagem para o tipo uint8
sobel = cv2.convertScaleAbs(sobel)

# Exibir a imagem original e a imagem filtrada
plt.subplot(121)
plt.imshow(img, cmap="gray")
```

```
plt.title("Imagem original")
plt.axis("off")
plt.subplot(122)
plt.imshow(sobel, cmap="gray")
plt.title("Imagem filtrada com Sobel")
plt.axis("off")
plt.show()
```



Questão 1.f

Neste código, vamos usar a biblioteca OpenCV para aplicar diferentes filtros em uma imagem em escala de cinza que contém ruído. O objetivo é comparar os efeitos dos filtros na redução do ruído e no realce da imagem. Os filtros que vamos usar são:

- Filtro 1: um filtro linear que usa um kernel 3x3 com valores 1 nas posições adjacentes ao centro e 0 nas demais, dividido por 5. Esse filtro é uma variação do filtro da média, que suaviza a imagem e reduz o ruído.

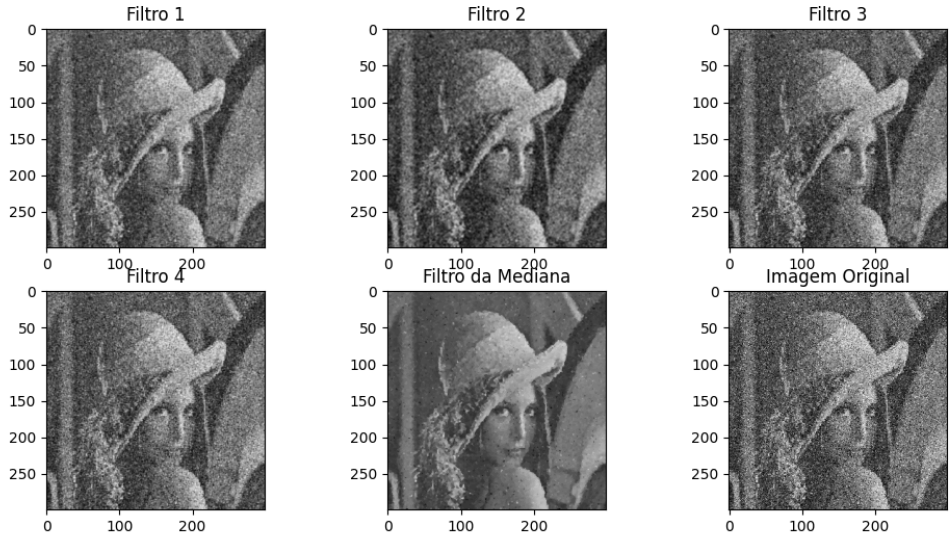
- Filtro 2: um filtro linear que usa um kernel 3x3 com valores 1 em todas as posições, dividido por 9. Esse filtro é o filtro da média, que também suaviza a imagem e reduz o ruído, mas com um efeito mais intenso que o filtro 1.
- Filtro 3: um filtro linear que usa um kernel 3x3 com valores 1 nas bordas, 3 nas posições adjacentes ao centro e 16 no centro, dividido por 32. Esse filtro é uma variação do filtro gaussiano, que suaviza a imagem e reduz o ruído, mas preservando mais as bordas que o filtro da média.
- Filtro 4: um filtro linear que usa um kernel 3x3 com valores 0 nas bordas, 1 nas posições adjacentes ao centro e 4 no centro, dividido por 8. Esse filtro é uma variação do filtro laplaciano, que realça as bordas e os detalhes da imagem, mas também aumenta o ruído.
- Filtro da mediana: um filtro não linear que usa uma janela 3x3 e substitui cada pixel pela mediana dos pixels vizinhos. Esse filtro é eficaz para reduzir o ruído do tipo sal e pimenta, que consiste em pixels brancos e pretos espalhados pela imagem.

O código consiste nos seguintes passos:

- Importar as bibliotecas cv2, numpy e matplotlib.pyplot, que são usadas para manipular imagens, realizar operações matemáticas e plotar gráficos, respectivamente.
- Ler a imagem “lena_ruido.bmp” em escala de cinza, usando a função cv2.imread. A imagem é uma versão em tons de cinza da famosa imagem de Lena, que é usada frequentemente como exemplo em processamento de imagens, mas com ruído do tipo sal e pimenta adicionado.
- Definir os kernels dos filtros lineares, que são matrizes 3x3 que contêm os coeficientes dos filtros. Os kernels são definidos como arrays do tipo float32, usando a biblioteca numpy.
- Aplicar os filtros na imagem, usando a função cv2.filter2D. Essa função realiza uma convolução da imagem com os kernels, gerando novas imagens filtradas. O filtro da mediana é aplicado usando a função cv2.medianBlur, que recebe a imagem e o tamanho da janela como parâmetros.
- Converter as imagens para o tipo uint8, usando a função cv2.convertScaleAbs. Essa função converte as imagens do tipo float64 para o tipo uint8, que é o formato mais comum para imagens em escala de cinza. A função também ajusta os valores das imagens para o intervalo entre 0 e 255, usando uma escala linear.
- Exibir a imagem original e as imagens filtradas, usando a biblioteca matplotlib.pyplot. Para isso, usamos as funções plt.subplot, plt.imshow, plt.title, plt.axis e plt.show, que permitem criar uma figura com seis subplots, mostrar as imagens em escala de cinza, adicionar títulos, remover os eixos e exibir o gráfico na tela.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Função para calcular métricas
# Carregar a imagem
imagem_original = cv2.imread('lena_ruido.bmp', cv2.IMREAD_GRAYSCALE)
```

```
# Aplicar os filtros
kernel1 = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]]) / 5.0
kernel2 = np.ones((3, 3), np.float32) / 9.0
kernel3 = np.array([[1, 3, 1], [3, 16, 3], [1, 3, 1]]) / 32.0
kernel4 = np.array([[0, 1, 0], [1, 4, 1], [0, 1, 0]]) / 8.0

resultado1 = cv2.filter2D(imagem_original, -1, kernel1)
resultado2 = cv2.filter2D(imagem_original, -1, kernel2)
resultado3 = cv2.filter2D(imagem_original, -1, kernel3)
resultado4 = cv2.filter2D(imagem_original, -1, kernel4)

# Aplicar o filtro da mediana
resultado_mediana = cv2.medianBlur(imagem_original, 3)

# Mostrar as imagens
plt.figure(figsize=(12, 6))

plt.subplot(231), plt.imshow(resultado1, cmap='gray'), plt.title('Filtro
1')
plt.subplot(232), plt.imshow(resultado2, cmap='gray'), plt.title('Filtro
2')
plt.subplot(233), plt.imshow(resultado3, cmap='gray'), plt.title('Filtro
3')
plt.subplot(234), plt.imshow(resultado4, cmap='gray'), plt.title('Filtro
4')
plt.subplot(235), plt.imshow(resultado_mediana, cmap='gray'),
plt.title('Filtro da Mediana')
plt.subplot(236), plt.imshow(imagem_original, cmap='gray'),
plt.title('Imagem Original')

plt.show()
```



Questão 2.a

Neste código, vamos usar a biblioteca OpenCV para realizar a operação de união entre duas imagens binárias. A união é uma operação morfológica que combina os objetos das duas imagens, resultando em uma imagem que contém todos os pixels brancos das imagens originais. A união é útil para juntar partes de objetos que foram separadas por algum processo anterior.

A operação morfológica de união é definida como a operação lógica OR entre os pixels das duas imagens. Ou seja, se pelo menos um dos pixels for branco, o pixel resultante será branco; se ambos os pixels forem pretos, o pixel resultante será preto. A operação de união pode ser realizada usando a função cv2.bitwise_or, que recebe duas imagens como argumentos e retorna a imagem resultante da união.

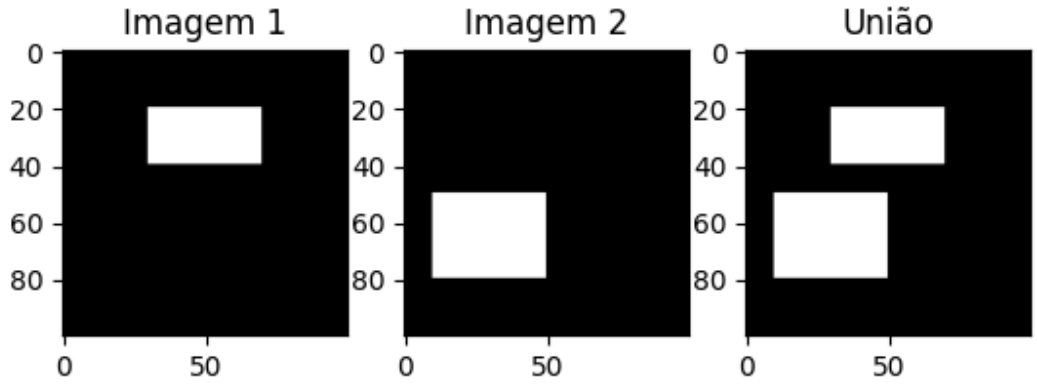
O código consiste nos seguintes passos:

- Importar as bibliotecas cv2, numpy e matplotlib.pyplot, que são usadas para manipular imagens, realizar operações matemáticas e plotar gráficos, respectivamente.
- Definir uma função chamada morfologia_uniao, que recebe duas imagens como parâmetros e retorna a imagem resultante da união entre elas. A função faz as seguintes verificações e operações:

    - Verifica se as imagens têm o mesmo tamanho, usando a função assert. Se as imagens tiverem tamanhos diferentes, a função gera um erro e interrompe a execução do código.
    - Binariza as imagens, usando a função cv2.threshold. Essa função converte as imagens em imagens binárias, onde os pixels com valor maior que 1 são transformados em 255 (branco) e os pixels com valor menor ou igual a 1 são transformados em 0 (preto). Isso é feito para garantir que os objetos das imagens sejam brancos em um fundo preto, como é esperado pela operação de união.
    - Realiza a operação de união, usando a função cv2.bitwise_or. Essa função aplica a operação lógica OR entre os pixels das duas imagens, gerando uma nova imagem que contém os pixels brancos de ambas as imagens.
    - Retorna a imagem resultante da união.
- Exemplo de uso da função:
    - Suponha que você tenha duas imagens binárias (0 para fundo preto, 255 para objeto branco) de tamanho 100x100, armazenadas nas variáveis imagem1 e imagem2.
    - Cria objetos nas imagens, atribuindo o valor 255 a alguns pixels. Na imagem1, cria um retângulo branco na região [20:40, 30:70]. Na imagem2, cria outro retângulo branco na região [50:80, 10:50].
    - Aplica a operação de união, chamando a função morfologia_uniao e passando as imagens como argumentos. A função retorna uma imagem que contém os dois retângulos brancos, unidos em uma única forma.
    - Exibe as imagens em uma única tela, usando a biblioteca matplotlib.pyplot. Para isso, usa as funções plt.subplot, plt.imshow, plt.title e plt.show, que permitem criar uma figura com três subplots, mostrar as imagens em escala de cinza, adicionar títulos e exibir o gráfico na tela.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morfologia_uniao(imagem1, imagem2):
```

```
    # Certifique-se de que as imagens têm o mesmo tamanho
    assert imagem1.shape == imagem2.shape, "As imagens devem ter o mesmo
tamanho"

    # Binarizar as imagens, assumindo que os objetos são brancos em um
fundo preto
    _, binarizada1 = cv2.threshold(imagem1, 1, 255, cv2.THRESH_BINARY)
    _, binarizada2 = cv2.threshold(imagem2, 1, 255, cv2.THRESH_BINARY)

    # Realizar a operação de união
    uniao = cv2.bitwise_or(binarizada1, binarizada2)

    return uniao

# Exemplo de uso:
# Suponha que você tenha duas imagens binárias (0 para fundo preto, 255
para objeto branco)
imagem1 = np.zeros((100, 100), dtype=np.uint8)
imagem2 = np.zeros((100, 100), dtype=np.uint8)

# Criar objetos nas imagens (objetos brancos)
imagem1[20:40, 30:70] = 255
imagem2[50:80, 10:50] = 255

# Aplicar a operação de união
resultado_uniao = morfologia_uniao(imagem1, imagem2)

# Exibir as imagens em uma única tela usando matplotlib
plt.subplot(1, 3, 1), plt.imshow(imagem1, cmap='gray'),
plt.title('Imagem 1')
plt.subplot(1, 3, 2), plt.imshow(imagem2, cmap='gray'),
plt.title('Imagem 2')
plt.subplot(1, 3, 3), plt.imshow(resultado_uniao, cmap='gray'),
plt.title('União')
plt.show()
```



Questão 2.b

O código apresenta uma função chamada morfologia_intersecao que recebe duas imagens como parâmetros e retorna a interseção entre elas. A interseção é uma operação morfológica que consiste em obter a região comum entre dois objetos binários. Para isso, o código utiliza as seguintes etapas:

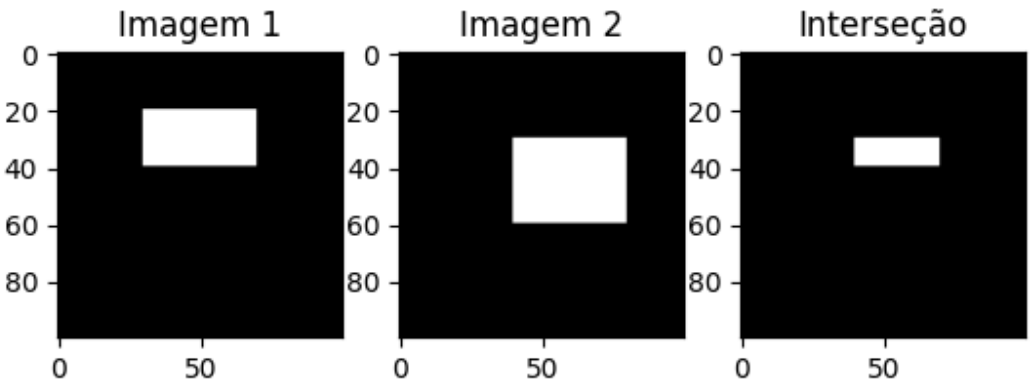
- Importar as bibliotecas cv2, numpy e matplotlib.pyplot, que são usadas para manipular imagens, realizar operações matemáticas e exibir gráficos, respectivamente.
- Binarizar as imagens, ou seja, transformá-las em imagens com apenas dois valores possíveis: 0 para o fundo preto e 255 para o objeto branco. Isso é feito usando a função cv2.threshold, que recebe uma imagem, um limiar, um valor máximo e um tipo de limiarização. Neste caso, o limiar é 1, o valor máximo é 255 e o tipo é cv2.THRESH_BINARY, que significa que os pixels com valor maior que o limiar serão atribuídos ao valor máximo e os demais serão atribuídos a zero.
- Realizar a operação de interseção usando a função cv2.bitwise_and, que recebe duas imagens e realiza a operação lógica AND entre elas, pixel a pixel. Isso significa que o resultado terá valor 255 apenas nos pixels em que ambas as imagens têm valor 255, ou seja, na região comum entre os objetos.
- Exibir as imagens usando a biblioteca matplotlib.pyplot, que permite criar uma janela com três subplots, cada um contendo uma imagem e um título. A função plt.imshow é usada para exibir a

imagem em escala de cinza, usando o parâmetro cmap='gray'. A função plt.title é usada para atribuir um título ao subplot. A função plt.show é usada para mostrar a janela na tela.

O código também apresenta um exemplo de uso da função morfologia_intersecao, criando duas imagens com objetos brancos em formas retangulares e aplicando a função para obter a interseção entre eles. O resultado é uma imagem com um objeto branco em forma de quadrado, que corresponde à região comum entre os retângulos.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morfologia_intersecao(imagem1, imagem2):
    # Certifique-se de que as imagens têm o mesmo tamanho
    assert imagem1.shape == imagem2.shape, "As imagens devem ter o mesmo
tamanho"

    # Binarizar as imagens, assumindo que os objetos são brancos em um
fundo preto
    _, binarizada1 = cv2.threshold(imagem1, 1, 255, cv2.THRESH_BINARY)
    _, binarizada2 = cv2.threshold(imagem2, 1, 255, cv2.THRESH_BINARY)

    # Realizar a operação de interseção
    intersecao = cv2.bitwise_and(binarizada1, binarizada2)

    return intersecao

# Exemplo de uso:
# Suponha que você tenha duas imagens binárias (0 para fundo preto, 255
para objeto branco)
imagem1 = np.zeros((100, 100), dtype=np.uint8)
imagem2 = np.zeros((100, 100), dtype=np.uint8)

# Criar objetos nas imagens (objetos brancos)
imagem1[20:40, 30:70] = 255
imagem2[30:60, 40:80] = 255

# Aplicar a operação de interseção
resultado_intersecao = morfologia_intersecao(imagem1, imagem2)
```

```
# Exibir as imagens em uma única tela usando matplotlib
plt.subplot(1, 3, 1), plt.imshow(imagem1, cmap='gray'),
plt.title('Imagem 1')
plt.subplot(1, 3, 2), plt.imshow(imagem2, cmap='gray'),
plt.title('Imagem 2')
plt.subplot(1, 3, 3), plt.imshow(resultado_intersecao, cmap='gray'),
plt.title('Interseção')
plt.show()
```



Questão 2.c

Neste código, vamos definir uma função chamada diferenca, que recebe duas imagens como parâmetros e retorna a imagem resultante da subtração dos valores dos pixels correspondentes. A função diferenca pode ser usada para comparar duas imagens e destacar as diferenças entre elas.

O código consiste nos seguintes passos:

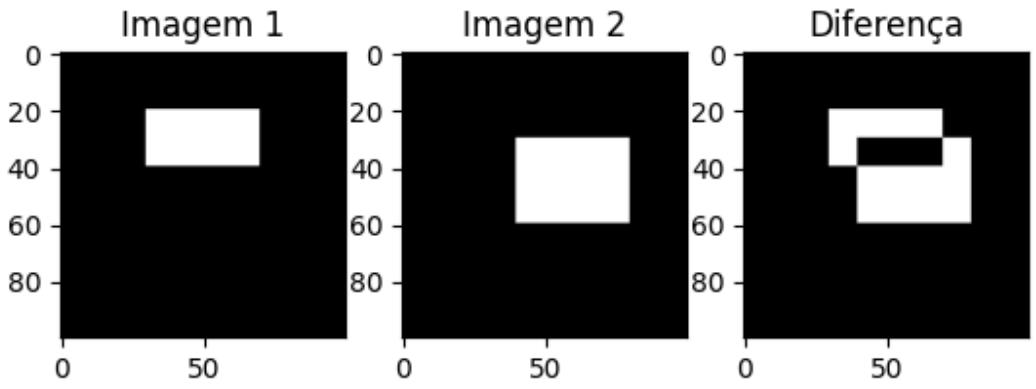
- Verificar se as imagens têm o mesmo tamanho, usando a instrução if. Se as imagens tiverem tamanhos diferentes, a função gera um erro e interrompe a execução do código, usando a função raise ValueError.
- Criar uma imagem vazia para armazenar o resultado, usando a função np.zeros_like. Essa função cria uma imagem com o mesmo tamanho e tipo da imagem original, mas com todos os pixels com valor zero.
- Percorrer cada pixel das imagens, usando dois laços for aninhados. Os laços for variam os índices i e j, que representam as coordenadas dos pixels nas imagens.
- Calcular a subtração dos valores dos pixels correspondentes, usando a operação de subtração (-). A subtração é feita entre os pixels das imagens originais, que são acessados usando a notação de colchetes ([i, j]). O resultado da subtração é atribuído ao pixel da imagem resultante, que também é acessado usando a notação de colchetes ([i, j]).
- Retornar a imagem resultante, usando a instrução return.

O código também mostra um exemplo de uso da função, criando duas imagens binárias com objetos brancos e aplicando a operação de diferença entre elas. O resultado é uma imagem que contém as diferenças entre os dois objetos, em tons de cinza. O código também exibe as imagens em uma única tela, usando a biblioteca matplotlib.pyplot.

```
def diferenca(im1, im2):
  # Verificar se as imagens têm o mesmo tamanho
  if im1.shape != im2.shape:
    raise ValueError("As imagens devem ter o mesmo tamanho")
  # Criar uma imagem vazia para armazenar o resultado
  im_diferenca = np.zeros_like(im1)
  # Percorrer cada pixel das imagens
  for i in range(im1.shape[0]):
    for j in range(im1.shape[1]):
      # Calcular a subtração dos valores dos pixels correspondentes
      im_diferenca[i, j] = im1[i, j] - im2[i, j]
  # Retornar a imagem resultante
  return im_diferenca
```

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morfologia_diferenca(imagem1, imagem2):
    # Certifique-se de que as imagens têm o mesmo tamanho
    assert imagem1.shape == imagem2.shape, "As imagens devem ter o mesmo tamanho"

    # Binarizar as imagens, assumindo que os objetos são brancos em um fundo preto
    _, binarizada1 = cv2.threshold(imagem1, 1, 255, cv2.THRESH_BINARY)
    _, binarizada2 = cv2.threshold(imagem2, 1, 255, cv2.THRESH_BINARY)

    # Realizar a operação de diferença
    diferenca = cv2.absdiff(binarizada1, binarizada2)

    return diferenca

# Exemplo de uso:
# Suponha que você tenha duas imagens binárias (0 para fundo preto, 255 para objeto branco)
imagem1 = np.zeros((100, 100), dtype=np.uint8)
imagem2 = np.zeros((100, 100), dtype=np.uint8)

# Criar objetos nas imagens (objetos brancos)
imagem1[20:40, 30:70] = 255
imagem2[30:60, 40:80] = 255

# Aplicar a operação de diferença
resultado_diferenca = morfologia_diferenca(imagem1, imagem2)

# Exibir as imagens em uma única tela usando matplotlib
plt.subplot(1, 3, 1), plt.imshow(imagem1, cmap='gray'),
plt.title('Imagem 1')
plt.subplot(1, 3, 2), plt.imshow(imagem2, cmap='gray'),
plt.title('Imagem 2')
plt.subplot(1, 3, 3), plt.imshow(resultado_diferenca, cmap='gray'),
```

```
plt.title('Diferença')
plt.show()
```



Questão 3.a

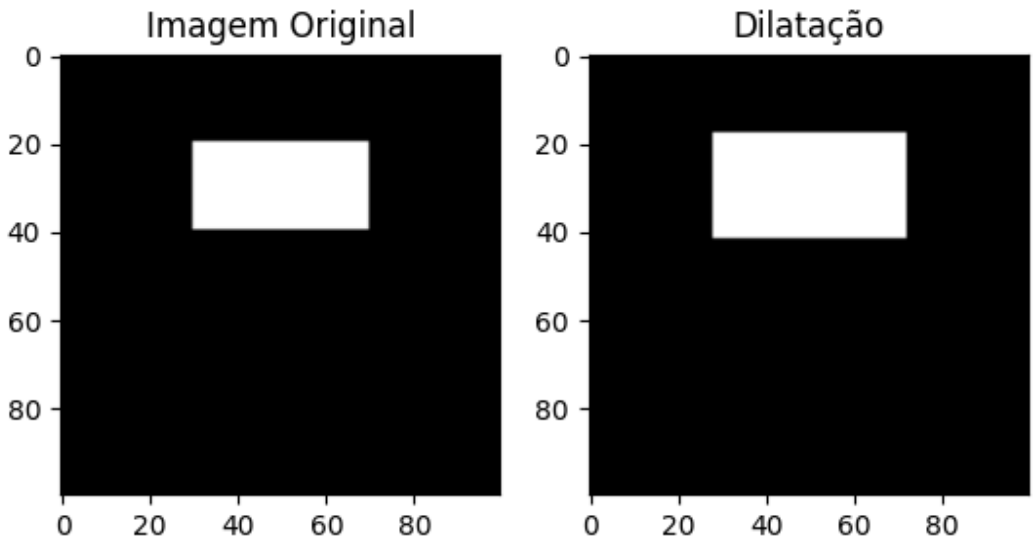
A dilatação é uma operação que aumenta o tamanho dos objetos brancos em uma imagem, preenchendo as lacunas e conectando as partes desconectadas. A dilatação depende de um **elemento estruturante**, que é uma matriz binária que define a forma e o tamanho da dilatação. O elemento estruturante é posicionado sobre cada pixel da imagem, e se pelo menos um dos pixels do elemento estruturante coincidir com um pixel branco da imagem, o pixel central do elemento estruturante é definido como branco na imagem de saída. O centro do elemento estruturante é chamado de **âncora** e pode ser especificado pelo usuário. A operação de dilatação pode ser repetida várias vezes, aumentando o efeito da dilatação.

O código define uma função chamada `morfologia_dilatacao` que recebe três parâmetros: `imagem`, `elemento_estruturante` e `centro`. A função converte a imagem em uma imagem binária usando a função `cv2.threshold`, que define todos os pixels com valor maior que 1 como 255 (branco) e todos os pixels com valor menor ou igual a 1 como 0 (preto). Em seguida, a função usa a função `cv2.dilate` para realizar a operação de dilatação na imagem binarizada, usando o elemento estruturante e a âncora fornecidos pelo usuário. A função retorna a imagem dilatada como resultado.

O código também mostra um exemplo de uso da função `morfologia_dilatacao`. Primeiro, o código cria uma imagem binária de 100 por 100 pixels, com todos os pixels pretos. Em seguida, o código cria um objeto branco na imagem, desenhando um retângulo branco de 20 por 40 pixels. Depois, o código cria um elemento estruturante de 5 por 5 pixels, com todos os pixels brancos. O código define o centro do elemento estruturante como o pixel na posição (2, 2). Por fim, o código aplica a função `morfologia_dilatacao` na imagem com o objeto, usando o elemento estruturante e o centro definidos. O código exibe a imagem original e a imagem dilatada usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. A imagem dilatada mostra que o objeto branco ficou maior e mais arredondado, de acordo com o elemento estruturante usado.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morfologia_dilatacao(imagem, elemento_estruturante, centro):
    # Certifique-se de que a imagem é binária (objetos brancos em fundo
preto)
    _, binarizada = cv2.threshold(imagem, 1, 255, cv2.THRESH_BINARY)

    # Realizar a operação de dilatação
    dilatacao = cv2.dilate(binarizada, elemento_estruturante,
anchor=centro, iterations=1)

    return dilatacao

# Exemplo de uso:
```

```
# Suponha que você tenha uma imagem binária (0 para fundo preto, 255
para objeto branco)
imagem = np.zeros((100, 100), dtype=np.uint8)

# Criar um objeto na imagem (objeto branco)
imagem[20:40, 30:70] = 255

# Criar um elemento estruturante (por exemplo, um quadrado)
elemento_estruturante = np.ones((5, 5), dtype=np.uint8)

# Definir o centro do elemento estruturante
centro_elemento_estruturante = (2, 2)

# Aplicar a operação de dilatação
resultado_dilatacao = morfologia_dilatacao(imagem,
elemento_estruturante, centro_elemento_estruturante)

# Exibir a imagem original e o resultado da dilatação
plt.subplot(1, 2, 1), plt.imshow(imagem, cmap='gray'), plt.title('Imagem
Original')
plt.subplot(1, 2, 2), plt.imshow(resultado_dilatacao, cmap='gray'),
plt.title('Dilatação')
plt.show()
```



Questão 3.b

A erosão é uma operação que diminui o tamanho dos objetos brancos em uma imagem, removendo os pixels das bordas e separando as partes conectadas. A erosão depende de um **elemento estruturante**, que é uma matriz binária que define a forma e o tamanho da erosão. O elemento estruturante é posicionado sobre cada pixel da imagem, e se todos os pixels do elemento estruturante coincidirem com os pixels brancos da imagem, o pixel central do elemento estruturante é definido como branco na imagem de saída. O centro do elemento estruturante é chamado de **âncora** e pode ser especificado pelo usuário. A operação de erosão pode ser repetida várias vezes, aumentando o efeito da erosão.

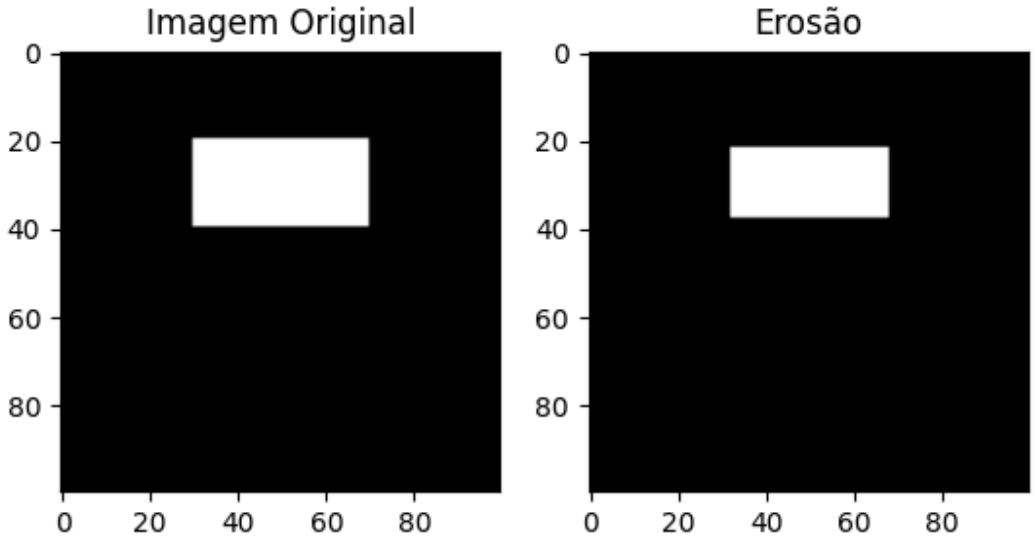
O código define uma função chamada `morfologia_erosao` que recebe três parâmetros: `imagem`, `elemento_estruturante` e `centro`. A função converte a imagem em uma imagem binária usando a função `cv2.threshold`, que define todos os pixels com valor maior que 1 como 255

(branco) e todos os pixels com valor menor ou igual a 1 como 0 (preto). Em seguida, a função usa a função `cv2.erode` para realizar a operação de erosão na imagem binarizada, usando o elemento estruturante e a âncora fornecidos pelo usuário. A função retorna a imagem erodida como resultado.

O código também mostra um exemplo de uso da função `morfologia_erosao`. Primeiro, o código cria uma imagem binária de 100 por 100 pixels, com todos os pixels pretos. Em seguida, o código cria um objeto branco na imagem, desenhando um retângulo branco de 20 por 40 pixels. Depois, o código cria um elemento estruturante de 5 por 5 pixels, com todos os pixels brancos. O código define o centro do elemento estruturante como o pixel na posição (2, 2). Por fim, o código aplica a função `morfologia_erosao` na imagem com o objeto, usando o elemento estruturante e o centro definidos. O código exibe a imagem original e a imagem erodida usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. A imagem erodida mostra que o objeto branco ficou menor e mais quadrado, de acordo com o elemento estruturante usado.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morfologia_erosao(imagem, elemento_estruturante, centro):
    # Certifique-se de que a imagem é binária (objetos brancos em fundo
preto)
    _, binarizada = cv2.threshold(imagem, 1, 255, cv2.THRESH_BINARY)

    # Realizar a operação de erosão
    erosao = cv2.erode(binarizada, elemento_estruturante, anchor=centro,
iterations=1)

    return erosao

# Exemplo de uso:
# Suponha que você tenha uma imagem binária (0 para fundo preto, 255
para objeto branco)
imagem = np.zeros((100, 100), dtype=np.uint8)

# Criar um objeto na imagem (objeto branco)
```

```
imagem[20:40, 30:70] = 255

# Criar um elemento estruturante (por exemplo, um quadrado)
elemento_estruturante = np.ones((5, 5), dtype=np.uint8)

# Definir o centro do elemento estruturante
centro_elemento_estruturante = (2, 2)

# Aplicar a operação de erosão
resultado_erosao = morfologia_erosao(imagem, elemento_estruturante,
centro_elemento_estruturante)

# Exibir a imagem original e o resultado da erosão
plt.subplot(1, 2, 1), plt.imshow(imagem, cmap='gray'), plt.title('Imagem
Original')
plt.subplot(1, 2, 2), plt.imshow(resultado_erosao, cmap='gray'),
plt.title('Erosão')
plt.show()
```



Questão 3.c

A abertura é uma operação que suaviza as bordas dos objetos brancos em uma imagem, removendo as saliências e os ruídos. A abertura é definida como a erosão seguida da dilatação da imagem pelo mesmo elemento estruturante. A erosão diminui o tamanho dos objetos brancos, enquanto a dilatação restaura o tamanho original dos objetos que não foram completamente eliminados pela erosão. A abertura depende de um **elemento estruturante**, que é uma matriz binária que define a forma e o tamanho da abertura. O centro do elemento estruturante é chamado de **âncora** e pode ser especificado pelo usuário.

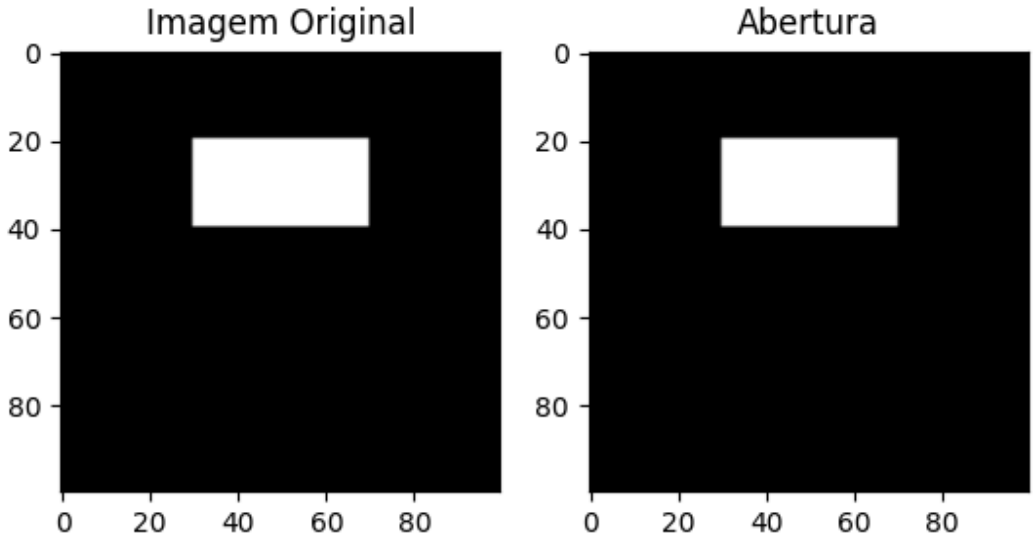
O código define uma função chamada `morfologia_abertura` que recebe três parâmetros: `imagem`, `elemento_estruturante` e `centro`. A função converte a imagem em uma imagem binária usando a função `cv2.threshold`, que define todos os pixels com valor maior que 1 como 255 (branco) e todos os pixels com valor menor ou igual a 1 como 0 (preto). Em seguida, a função

usa a função `cv2.morphologyEx` com o parâmetro `cv2.MORPH_OPEN` para realizar a operação de abertura na imagem binarizada, usando o elemento estruturante e a âncora fornecidos pelo usuário. A função retorna a imagem aberta como resultado.

O código também mostra um exemplo de uso da função `morfologia_abertura`. Primeiro, o código cria uma imagem binária de 100 por 100 pixels, com todos os pixels pretos. Em seguida, o código cria um objeto branco na imagem, desenhando um retângulo branco de 20 por 40 pixels. Depois, o código cria um elemento estruturante maior de 9 por 9 pixels, com todos os pixels brancos. O código define o centro do elemento estruturante como o pixel na posição (4, 4). Por fim, o código aplica a função `morfologia_abertura` na imagem com o objeto, usando o elemento estruturante e o centro definidos. O código exibe a imagem original e a imagem aberta usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. A imagem aberta mostra que o objeto branco ficou mais suave e sem saliências, de acordo com o elemento estruturante usado.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morfologia_abertura(imagem, elemento_estruturante, centro):
    # Certifique-se de que a imagem é binária (objetos brancos em fundo
preto)
    _, binarizada = cv2.threshold(imagem, 1, 255, cv2.THRESH_BINARY)

    # Realizar a operação de abertura
    abertura = cv2.morphologyEx(binarizada, cv2.MORPH_OPEN,
elemento_estruturante, anchor=centro)

    return abertura

# Exemplo de uso:
# Suponha que você tenha uma imagem binária (0 para fundo preto, 255
```

```
para objeto branco)
imagem = np.zeros((100, 100), dtype=np.uint8)

# Criar um objeto na imagem (objeto branco)
imagem[20:40, 30:70] = 255

# Criar um elemento estruturante maior (por exemplo, um quadrado de 9x9)
elemento_estruturante = np.ones((9, 9), dtype=np.uint8)

# Definir o centro do elemento estruturante
centro_elemento_estruturante = (4, 4)

# Aplicar a operação de abertura
resultado_abertura = morfologia_abertura(imagem, elemento_estruturante,
centro_elemento_estruturante)

# Exibir a imagem original e o resultado da abertura
plt.subplot(1, 2, 1), plt.imshow(imagem, cmap='gray'), plt.title('Imagem
Original')
plt.subplot(1, 2, 2), plt.imshow(resultado_abertura, cmap='gray'),
plt.title('Abertura')
plt.show()
```



Questão 3.d

O código é um exemplo de como realizar a operação de morfologia matemática chamada **fechamento** em uma imagem binária usando a biblioteca OpenCV em Python. O fechamento é uma operação que preenche as lacunas e os buracos dos objetos brancos em uma imagem, conectando as partes desconectadas. O fechamento é definido como a dilatação seguida da erosão da imagem pelo mesmo elemento estruturante. A dilatação aumenta o tamanho dos objetos brancos, enquanto a erosão restaura o tamanho original dos objetos que não foram completamente expandidos pela dilatação. O fechamento depende de um **elemento estruturante**, que é uma matriz binária que define a forma e o tamanho do fechamento. O centro do elemento estruturante é chamado de **âncora** e pode ser especificado pelo usuário.

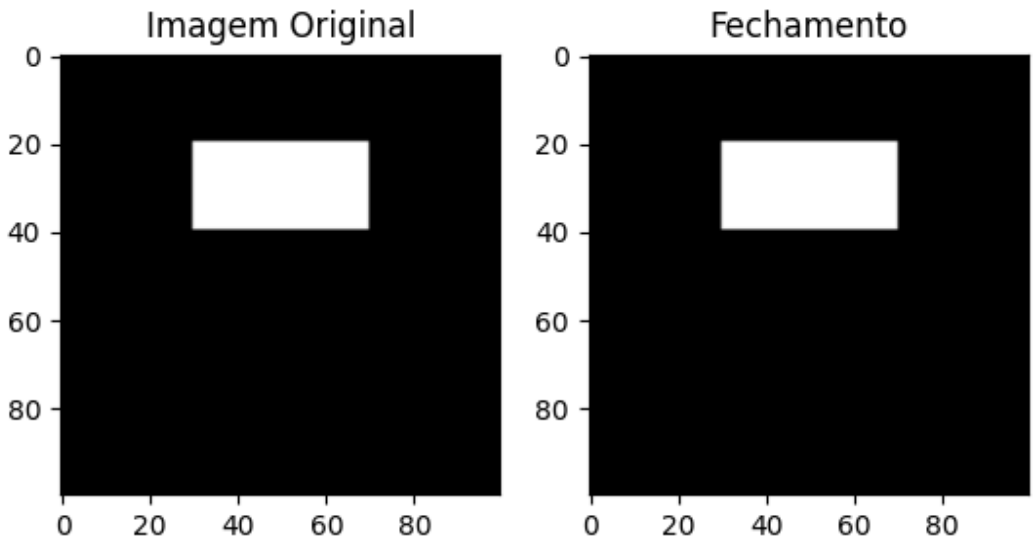
O código define uma função chamada `morfologia_fechamento` que recebe três parâmetros: `imagem`, `elemento_estruturante` e `centro`. A função converte a imagem em uma imagem

binária usando a função `cv2.threshold`, que define todos os pixels com valor maior que 1 como 255 (branco) e todos os pixels com valor menor ou igual a 1 como 0 (preto). Em seguida, a função usa a função `cv2.morphologyEx` com o parâmetro `cv2.MORPH_CLOSE` para realizar a operação de fechamento na imagem binarizada, usando o elemento estruturante e a âncora fornecidos pelo usuário. A função retorna a imagem fechada como resultado.

O código também mostra um exemplo de uso da função `morfologia_fechamento`. Primeiro, o código cria uma imagem binária de 100 por 100 pixels, com todos os pixels pretos. Em seguida, o código cria um objeto branco na imagem, desenhando um retângulo branco de 20 por 40 pixels. Depois, o código cria um elemento estruturante de 5 por 5 pixels, com todos os pixels brancos. O código define o centro do elemento estruturante como o pixel na posição (2, 2). Por fim, o código aplica a função `morfologia_fechamento` na imagem com o objeto, usando o elemento estruturante e o centro definidos. O código exibe a imagem original e a imagem fechada usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. A imagem fechada mostra que o objeto branco ficou sem lacunas e buracos, de acordo com o elemento estruturante usado.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morfologia_fechamento(imagem, elemento_estruturante, centro):
    # Certifique-se de que a imagem é binária (objetos brancos em fundo
preto)
    _, binarizada = cv2.threshold(imagem, 1, 255, cv2.THRESH_BINARY)

    # Realizar a operação de fechamento
    fechamento = cv2.morphologyEx(binarizada, cv2.MORPH_CLOSE,
elemento_estruturante, anchor=centro)

    return fechamento

# Exemplo de uso:
# Suponha que você tenha uma imagem binária (0 para fundo preto, 255
para objeto branco)
imagem = np.zeros((100, 100), dtype=np.uint8)
```

```
# Criar um objeto na imagem (objeto branco)
imagem[20:40, 30:70] = 255

# Criar um elemento estruturante (por exemplo, um quadrado)
elemento_estruturante = np.ones((5, 5), dtype=np.uint8)

# Definir o centro do elemento estruturante
centro_elemento_estruturante = (2, 2)

# Aplicar a operação de fechamento
resultado_fechamento = morfologia_fechamento(imagem,
elemento_estruturante, centro_elemento_estruturante)

# Exibir a imagem original e o resultado do fechamento
plt.subplot(1, 2, 1), plt.imshow(imagem, cmap='gray'), plt.title('Imagem
Original')
plt.subplot(1, 2, 2), plt.imshow(resultado_fechamento, cmap='gray'),
plt.title('Fechamento')
plt.show()
```



Questão 4.a

O código é um exemplo de como preencher buracos em uma imagem binária usando a biblioteca OpenCV em Python. Preencher buracos é uma tarefa útil para restaurar imagens danificadas ou remover ruídos indesejados. Preencher buracos pode ser feito usando uma operação de morfologia matemática chamada **fechamento**, que é a dilatação seguida da erosão da imagem pelo mesmo elemento estruturante. A dilatação aumenta o tamanho dos objetos brancos, enquanto a erosão restaura o tamanho original dos objetos que não foram completamente expandidos pela dilatação. O fechamento depende de um **elemento estruturante**, que é uma matriz binária que define a forma e o tamanho do fechamento. O centro do elemento estruturante é chamado de **âncora** e pode ser especificado pelo usuário.

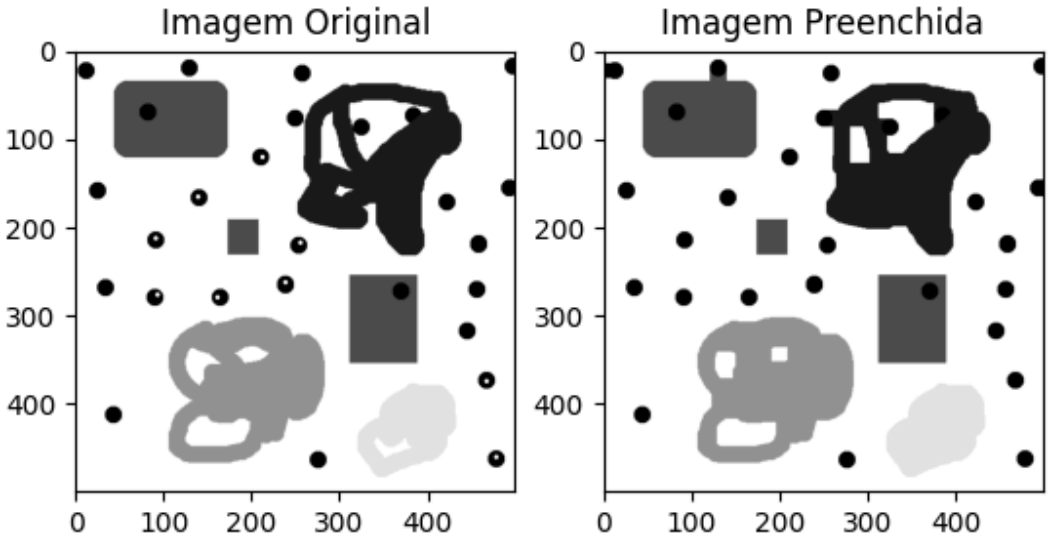
O código define uma função chamada `preencher_buracos` que recebe um parâmetro: `imagem`. A função inverte a imagem usando a função `cv2.bitwise_not`, que troca os valores dos pixels (0

por 255 e vice-versa). Isso é feito para que os objetos sejam pretos e o fundo seja branco, facilitando a aplicação do fechamento. Em seguida, a função cria um elemento estruturante de 15 por 15 pixels, com todos os pixels brancos. O tamanho e a forma do elemento estruturante podem ser ajustados conforme a necessidade. O código define o centro do elemento estruturante como o pixel na posição (7, 7). Depois, a função usa a função `cv2.morphologyEx` com o parâmetro `cv2.MORPH_CLOSE` para realizar a operação de fechamento na imagem invertida, usando o elemento estruturante e a âncora fornecidos pelo usuário. A função inverte novamente a imagem para obter a imagem final com os buracos preenchidos. A função retorna a imagem final como resultado.

O código também mostra um exemplo de uso da função `preencher_buracos`. O código carrega uma imagem em escala de cinza chamada ‘quadro.png’, que mostra um quadro branco com alguns buracos pretos. O código verifica se a imagem foi carregada corretamente, caso contrário, exibe uma mensagem de erro. Se a imagem foi carregada, o código aplica a função `preencher_buracos` na imagem do quadro. O código exibe a imagem original e a imagem preenchida usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. A imagem preenchida mostra que os buracos pretos foram eliminados, de acordo com o elemento estruturante usado.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def preencher_buracos(imagem):
    # Inverter a imagem (objetos pretos em fundo branco)
    imagem_invertida = cv2.bitwise_not(imagem)

    # Criar um elemento estruturante (pode ser ajustado conforme
necessário)
    elemento_estruturante = np.ones((15, 15), dtype=np.uint8)

    # Aplicar a operação de fechamento para preencher buracos
    imagem_preenchida = cv2.morphologyEx(imagem_invertida,
cv2.MORPH_CLOSE, elemento_estruturante)
```

```
    # Inverter novamente para obter a imagem final
    imagem_final = cv2.bitwise_not(imagem_preenchida)

    return imagem_final

# Carregar a imagem
imagem_quadro = cv2.imread('quadro.png', cv2.IMREAD_GRAYSCALE)

# Verificar se a imagem foi carregada corretamente
if imagem_quadro is None:
    print("Erro ao carregar a imagem.")
else:
    # Preencher buracos na imagem
    imagem_preenchida = preencher_buracos(imagem_quadro)

    # Exibir as imagens original e preenchida
    plt.subplot(1, 2, 1), plt.imshow(imagem_quadro, cmap='gray'),
plt.title('Imagem Original')
    plt.subplot(1, 2, 2), plt.imshow(imagem_preenchida, cmap='gray'),
plt.title('Imagem Preenchida')
    plt.show()
```



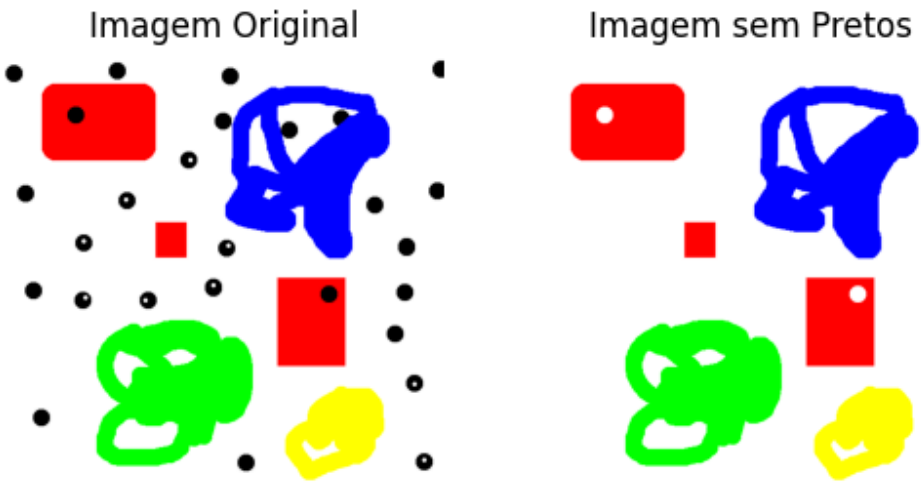
Questão 4.b

O código é um exemplo de como remover objetos pretos de uma imagem colorida usando a biblioteca OpenCV em Python. Remover objetos pretos é uma tarefa útil para limpar imagens que possuem buracos ou manchas pretas que atrapalham a visualização. Remover objetos pretos pode ser feito usando uma técnica simples de mascaramento, que consiste em selecionar os pixels de uma determinada cor e substituí-los por outra cor.

O código carrega uma imagem em cores chamada ‘quadro.png’, que mostra um quadro branco com alguns buracos pretos. O código converte a imagem em escala de cinza usando a função `cv2.cvtColor`, que permite mudar o espaço de cores de uma imagem. Em seguida, o código cria uma máscara para os pixels pretos usando a comparação `imagem_cinza == 0`, que retorna uma matriz booleana com True para os pixels que possuem valor zero (preto) e False para os demais. Depois, o código aplica a máscara na imagem original usando a função `np.where`, que permite escolher valores de duas matrizes de acordo com uma condição. Nesse caso, a condição é a

máscara booleana, e as matrizes são 255 (branco) e a imagem original. Assim, o código substitui os pixels pretos por brancos na imagem original. Por fim, o código exibe a imagem original e a imagem sem objetos pretos usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Carregando a imagem
imagem = cv2.imread('quadro.png')

# Convertendo a imagem para escala de cinza
imagem_cinza = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)

# Criando uma máscara para os pixels pretos
mascara_preta = (imagem_cinza == 0)

# Aplicando a máscara na imagem original
imagem_sem_pretos = np.where(mascara_preta[:, :, None], 255, imagem)

# Exibindo a imagem original
plt.subplot(1, 2, 1)
plt.title('Imagem Original')
plt.imshow(cv2.cvtColor(imagem, cv2.COLOR_BGR2RGB))
plt.axis('off')

# Exibindo a imagem sem objetos pretos
plt.subplot(1, 2, 2)
plt.title('Imagem sem Pretos')
plt.imshow(cv2.cvtColor(imagem_sem_pretos, cv2.COLOR_BGR2RGB))
plt.axis('off')

# Mostrando os gráficos
plt.show()
```



Questão 4.c

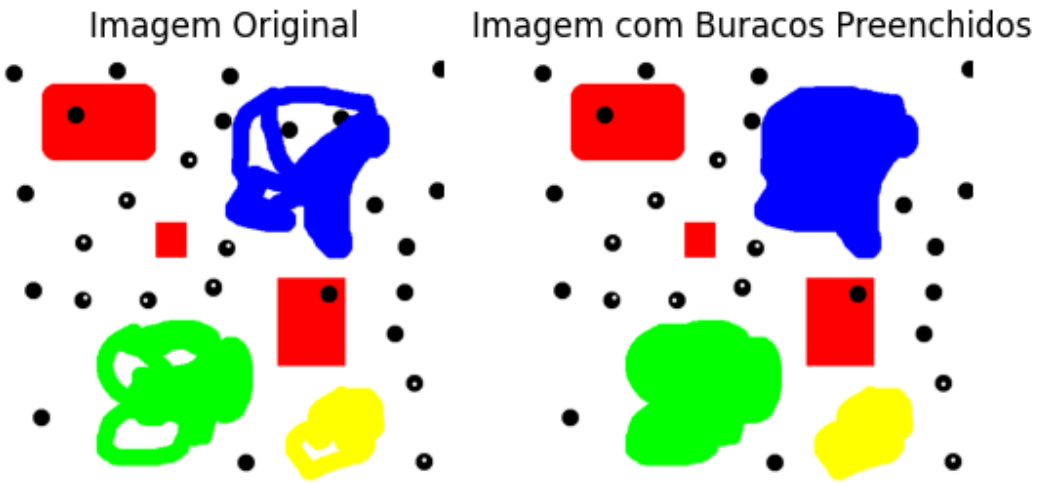
Remover objetos pretos é uma tarefa útil para limpar imagens que possuem buracos ou manchas pretas que atrapalham a visualização. Remover objetos pretos pode ser feito usando uma técnica simples de mascaramento, que consiste em selecionar os pixels de uma determinada cor e substituí-los por outra cor.

O código carrega uma imagem em cores chamada ‘quadro.png’, que mostra um quadro branco com alguns buracos pretos. O código converte a imagem em escala de cinza usando a função `cv2.cvtColor`, que permite mudar o espaço de cores de uma imagem. Em seguida, o código cria uma máscara para os pixels pretos usando a comparação `imagem_cinza == 0`, que retorna uma matriz booleana com True para os pixels que possuem valor zero (preto) e False para os demais. Depois, o código aplica a máscara na imagem original usando a função `np.where`, que permite escolher valores de duas matrizes de acordo com uma condição. Nesse caso, a condição é a máscara booleana, e as matrizes são 255 (branco) e a imagem original. Assim, o código substitui os

pixels pretos por brancos na imagem original. Por fim, o código exibe a imagem original e a imagem sem objetos pretos usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Carregando a imagem
imagem = cv2.imread('quadro.png')

# Exibindo a imagem original
plt.subplot(1, 2, 1)
plt.title('Imagem Original')
plt.imshow(cv2.cvtColor(imagem, cv2.COLOR_BGR2RGB))
plt.axis('off')
# Convertendo a imagem para o espaço de cores HSV
imagem_hsv = cv2.cvtColor(imagem, cv2.COLOR_BGR2HSV)

# Definindo as faixas de cores para azul, amarelo e verde
faixa_azul = np.array([100, 50, 50]), np.array([140, 255, 255])
faixa_amarelo = np.array([20, 100, 100]), np.array([30, 255, 255])
faixa_verde = np.array([40, 50, 50]), np.array([80, 255, 255])

# Criando máscaras para os objetos de cor azul, amarelo e verde
mascara_azul = cv2.inRange(imagem_hsv, *faixa_azul)
mascara_amarelo = cv2.inRange(imagem_hsv, *faixa_amarelo)
mascara_verde = cv2.inRange(imagem_hsv, *faixa_verde)

# Preenchendo os buracos usando a função cv2.fillPoly
cv2.fillPoly(imagem, [cv2.findNonZero(mascara_azul)], color=(255, 0, 0))
cv2.fillPoly(imagem, [cv2.findNonZero(mascara_amarelo)], color=(0, 255,
255))
cv2.fillPoly(imagem, [cv2.findNonZero(mascara_verde)], color=(0, 255,
0))

# Salvando a imagem com buracos preenchidos
```

```
cv2.imwrite('imagem_com_buracos_preenchidos.png', imagem)

# Exibindo a imagem com buracos preenchidos
plt.subplot(1, 2, 2)
plt.title('Imagem com Buracos Preenchidos')
plt.imshow(cv2.cvtColor(imagem, cv2.COLOR_BGR2RGB))
plt.axis('off')

# Mostrando os gráficos
plt.show()
```



Questão 4.d

O fecho convexo de um conjunto de pontos é o menor polígono convexo que contém todos os pontos do conjunto1. O fecho convexo pode ser usado para simplificar a forma dos objetos, eliminar ruídos ou buracos, ou medir propriedades geométricas como área, perímetro ou orientação2.

O código define uma função chamada `encontrar_fecho_convexo` que recebe dois parâmetros: `imagem` e `cor`. A função converte a imagem para o espaço de cor HSV (matiz, saturação e valor), que é mais adequado para trabalhar com cores do que o espaço de cor RGB (vermelho, verde e azul)3. A função usa a função `cv2.cvtColor` para realizar a conversão. Em seguida, a função define as faixas de cor para cada cor possível (‘azul’, ‘amarelo’ ou ‘verde’), usando arrays numpy que representam os valores mínimos e máximos de matiz, saturação e valor para cada cor. Se a cor fornecida não for suportada, a função lança um erro. Depois, a função cria uma máscara usando a função `cv2.inRange`, que retorna uma imagem binária com os pixels que estão dentro da faixa de cor definida como brancos e os demais como pretos. A máscara serve para isolar os objetos da cor desejada na imagem. Em seguida, a função encontra os contornos na máscara usando a função `cv2.findContours`, que retorna uma lista de arrays numpy que representam os pontos que formam os contornos dos objetos na máscara. Por fim, a função encontra e desenha o fecho convexo para cada contorno usando a função `cv2.convexHull`, que retorna um array numpy que representa os pontos que formam o fecho convexo de um contorno. A função retorna uma lista de arrays numpy que representam os fechos convexos de todos os contornos.

O código também mostra um exemplo de uso da função `encontrar_fecho_convexo`. O código carrega uma imagem chamada ‘imagem_com_buracos_preenchidos.png’, que mostra um quadro branco com alguns objetos de cores diferentes. O código encontra o fecho convexo para cada cor usando a função `encontrar_fecho_convexo` com os parâmetros ‘azul’, ‘amarelo’ e ‘verde’. O código cria uma imagem vazia do mesmo tamanho que a imagem original. O código desenha o fecho convexo na imagem vazia usando a função `cv2.drawContours`, que recebe uma lista de arrays numpy que representam os fechos convexos, uma cor e uma espessura para desenhar os contornos. O código usa as cores azul, amarelo e verde para desenhar os fechos convexos das respectivas cores. Por fim, o código mostra a imagem original e a imagem com os fechos convexos usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. A imagem com os fechos convexos mostra que os objetos de cada cor foram simplificados por um polígono convexo que contém todos os pontos do objeto.

```
import cv2
```

```
import numpy as np
import matplotlib.pyplot as plt

# Função para encontrar o fecho convexo de objetos de uma cor específica
def encontrar_fecho_convexo(imagem, cor):
    # Converte a imagem para o espaço de cor HSV
    hsv = cv2.cvtColor(imagem, cv2.COLOR_BGR2HSV)

    # Define as faixas de cor (nesse exemplo, substitua pelos valores
corretos)
    if cor == 'azul':
        lower_range = np.array([90, 50, 50])
        upper_range = np.array([130, 255, 255])
    elif cor == 'amarelo':
        lower_range = np.array([20, 100, 100])
        upper_range = np.array([30, 255, 255])
    elif cor == 'verde':
        lower_range = np.array([40, 50, 50])
        upper_range = np.array([80, 255, 255])
    else:
        raise ValueError("Cor não suportada")

    # Cria uma máscara usando a faixa de cor
    mascara = cv2.inRange(hsv, lower_range, upper_range)

    # Encontra contornos na máscara
    contornos, _ = cv2.findContours(mascara, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    # Encontra e desenha o fecho convexo para cada contorno
    fecho_convexo = []
    for contorno in contornos:
        hull = cv2.convexHull(contorno)
        fecho_convexo.append(hull)

    return fecho_convexo

# Carrega a imagem
imagem = cv2.imread('imagem_com_buracos_preenchidos.png')
```

```
# Encontra o fecho convexo para cada cor
azul_fecho_convexo = encontrar_fecho_convexo(imagem, 'azul')
amarelo_fecho_convexo = encontrar_fecho_convexo(imagem, 'amarelo')
verde_fecho_convexo = encontrar_fecho_convexo(imagem, 'verde')

# Cria uma imagem vazia
imagem_resultado = np.zeros_like(imagem)

# Desenha o fecho convexo na imagem resultante
cv2.drawContours(imagem_resultado, azul_fecho_convexo, -1, (255, 0, 0),
2)
cv2.drawContours(imagem_resultado, amarelo_fecho_convexo, -1, (0, 255,
255), 2)
cv2.drawContours(imagem_resultado, verde_fecho_convexo, -1, (0, 255, 0),
2)

# Mostra a imagem resultante usando Matplotlib
plt.imshow(cv2.cvtColor(imagem_resultado, cv2.COLOR_BGR2RGB))
plt.title('Fecho Convexo para Azul, Amarelo e Verde')
plt.show()
```

Questão 4.e

O esqueleto de um objeto é uma representação simplificada do objeto que preserva sua forma, topologia e conectividade1. O esqueleto pode ser usado para análise de forma, reconhecimento de objetos, extração de características, entre outras aplicações2.

O código carrega uma imagem chamada ‘imagem_com_buracos_preenchidos.png’, que mostra um quadro branco com alguns objetos de cores diferentes. O código converte a imagem para o espaço de cor HSV (matiz, saturação e valor), que é mais adequado para trabalhar com cores do que o espaço de cor RGB (vermelho, verde e azul)3. A função usa a função `cv2.cvtColor` para realizar a conversão. Em seguida, o código define os intervalos de cor para azul, amarelo e verde, usando arrays numpy que representam os valores mínimos e máximos de matiz, saturação e valor para cada cor. Depois, o código segmenta as cores usando a função `cv2.inRange`, que retorna uma

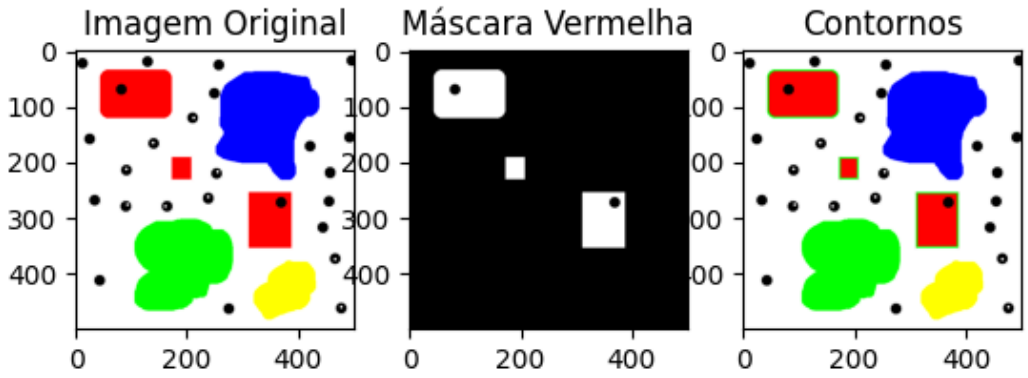
imagem binária com os pixels que estão dentro do intervalo de cor definido como brancos e os demais como pretos. A segmentação serve para isolar os objetos de cada cor na imagem.

O código define uma função chamada `find_skeleton` que recebe um parâmetro: `mask`. A função encontra os contornos na máscara usando a função `cv2.findContours`, que retorna uma lista de arrays numpy que representam os pontos que formam os contornos dos objetos na máscara. Em seguida, a função cria uma imagem em branco para desenhar os contornos usando a função `np.zeros_like`. Depois, a função desenha os contornos na imagem em branco usando a função `cv2.drawContours`, que recebe uma lista de arrays numpy que representam os contornos, uma cor e uma espessura para desenhar os contornos. Por fim, a função encontra o esqueleto dos contornos usando a função `cv2.ximgproc.thinning`, que aplica um algoritmo de afinamento iterativo para obter uma representação de um pixel de espessura dos contornos4. A função retorna a imagem com o esqueleto como resultado.

O código também mostra um exemplo de uso da função `find_skeleton`. O código encontra o esqueleto para cada cor usando a função `find_skeleton` com as máscaras de azul, amarelo e verde. O código exibe os resultados usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. As imagens com os esqueletos mostram que os objetos de cada cor foram simplificados por uma linha de um pixel de espessura que preserva sua forma e conectividade.

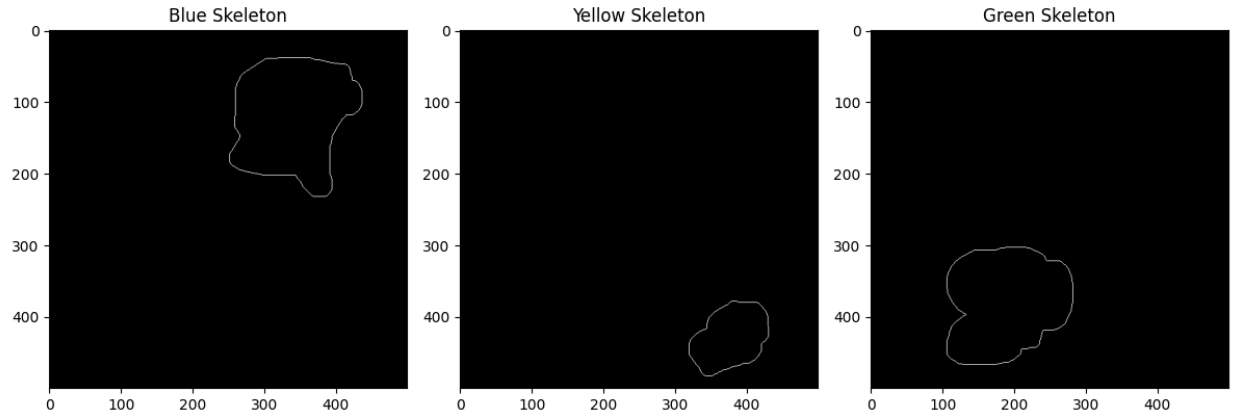
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Leitura da imagem
image = cv2.imread('imagem_com_buracos_preenchidos.png')

# Conversão da imagem para o espaço de cores HSV
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Definindo os intervalos de cor para azul, amarelo e verde
lower_blue = np.array([100, 50, 50])
upper_blue = np.array([140, 255, 255])
```

```
lower_yellow = np.array([20, 100, 100])
upper_yellow = np.array([30, 255, 255])

lower_green = np.array([40, 50, 50])
upp
```




```
er_green = np.array([80, 255, 255])

# Segmentação de cores
mask_blue = cv2.inRange(hsv, lower_blue, upper_blue)
mask_yellow = cv2.inRange(hsv, lower_yellow, upper_yellow)
mask_green = cv2.inRange(hsv, lower_green, upper_green)

# Encontrar o esqueleto dos objetos
def find_skeleton(mask):
    # Encontrar contornos
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

```
    # Criar uma imagem em branco para desenhar os contornos
    skeleton = np.zeros_like(mask)

    # Desenhar os contornos na imagem em branco
    cv2.drawContours(skeleton, contours, -1, 255, 1)

    # Encontrar o esqueleto
    skeleton = cv2.ximgproc.thinning(skeleton)

    return skeleton

# Encontrar o esqueleto para cada cor
skeleton_blue = find_skeleton(mask_blue)
skeleton_yellow = find_skeleton(mask_yellow)
skeleton_green = find_skeleton(mask_green)

# Exibir os resultados usando Matplotlib
plt.figure(figsize=(12, 4))

plt.subplot(131)
plt.title('Blue Skeleton')
plt.imshow(skeleton_blue, cmap='gray')

plt.subplot(132)
plt.title('Yellow Skeleton')
plt.imshow(skeleton_yellow, cmap='gray')

plt.subplot(133)
plt.title('Green Skeleton')
plt.imshow(skeleton_green, cmap='gray')

plt.tight_layout()
plt.show()
```



Questão 4.f

A transformada hit-or-miss é uma operação que detecta uma determinada configuração (ou padrão) em uma imagem binária, usando o operador de erosão morfológica e um par de elementos estruturantes disjuntos1. O resultado da transformada hit-or-miss é o conjunto de posições onde o primeiro elemento estruturante se encaixa no primeiro plano da imagem de entrada, e o segundo elemento estruturante não se encaixa em nenhum ponto1.

O código carrega uma imagem chamada ‘imagem_com_buracos_preenchidos.png’, que mostra um quadro branco com alguns objetos de cores diferentes. O código converte a imagem para o espaço de cor HSV (matiz, saturação e valor), que é mais adequado para trabalhar com cores do que o espaço de cor RGB (vermelho, verde e azul)2. A função usa a função `cv2.cvtColor` para realizar a conversão. Em seguida, o código extrai apenas os objetos vermelhos na imagem original, usando a função `cv2.inRange`, que retorna uma imagem binária com os pixels que estão dentro de um intervalo de cor definido como brancos e os demais como pretos. O intervalo de cor para o vermelho é definido pelos arrays numpy `lower_red` e `upper_red`, que representam os valores mínimos e máximos de matiz, saturação e valor para o vermelho.

Depois, o código aplica a transformada hit-or-miss à máscara vermelha, usando a função `cv2.morphologyEx`, que realiza operações morfológicas em uma imagem binária, usando um elemento estruturante e um tipo de operação. O tipo de operação é definido pelo parâmetro `cv2.MORPH_HITMISS`, que indica a transformada hit-or-miss. O elemento estruturante é definido pelo array numpy `kernel`, que representa uma matriz binária de 3 por 3 pixels, com o valor 1 nos pixels centrais e adjacentes, e o valor 0 nos pixels diagonais. Esse elemento estruturante serve

para detectar os pixels que estão no centro de um quadrado de 3 por 3 pixels, e que não têm vizinhos diagonais.

Em seguida, o código encontra os contornos na imagem resultante, usando a função `cv2.findContours`, que retorna uma lista de arrays numpy que representam os pontos que formam os contornos dos objetos na imagem. Depois, o código desenha os contornos na imagem original, usando a função `cv2.drawContours`, que recebe uma lista de arrays numpy que representam os contornos, uma cor e uma espessura para desenhar os contornos. O código usa a cor verde para desenhar os contornos.

Por fim, o código exibe a imagem original, a máscara vermelha, e a imagem com contornos, usando a biblioteca matplotlib.pyplot, que permite visualizar imagens em gráficos. A imagem com contornos mostra que a transformada hit-or-miss detectou os pixels que correspondem ao padrão definido pelo elemento estruturante, e que estão no centro dos objetos vermelhos na imagem original.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Carregar a imagem
imagem = cv2.imread('imagem_com_buracos_preenchidos.png')

# Converter a imagem para escala de cinza
imagem_gray = cv2.cvtColor(imagem, cv2.COLOR_BGR2GRAY)

# Extrair apenas os objetos vermelhos na imagem original
lower_red = np.array([0, 0, 100])
upper_red = np.array([100, 100, 255])
mask_red = cv2.inRange(imagem, lower_red, upper_red)

# Aplicar a transformada hit-or-miss à máscara vermelha
kernel = np.array([[0, 1, 0],
                   [1, 1, 1],
                   [0, 1, 0]], dtype=np.uint8)
```

```
resultado_hit_or_miss = cv2.morphologyEx(mask_red, cv2.MORPH_HITMISS,
kernel)

# Encontrar contornos na imagem resultante
contornos, _ = cv2.findContours(resultado_hit_or_miss,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Desenhar contornos na imagem original
imagem_contornos = imagem.copy()
cv2.drawContours(imagem_contornos, contornos, -1, (0, 255, 0), 2)

# Exibir a imagem original, a máscara vermelha, e a imagem com contornos
plt.subplot(1, 3, 1), plt.imshow(cv2.cvtColor(imagem,
cv2.COLOR_BGR2RGB)), plt.title('Imagem Original')
plt.subplot(1, 3, 2), plt.imshow(mask_red, cmap='gray'),
plt.title('Máscara Vermelha')
plt.subplot(1, 3, 3), plt.imshow(cv2.cvtColor(imagem_contornos,
cv2.COLOR_BGR2RGB)), plt.title('Contornos')

plt.show()
```