Relatório - Atividade Prática 02

UNIVERSIDADE FEDERAL DO PIAUÍ - UFPI
DC/CCN037 - Processamento Digital de Imagens
Professor: Kelson Romulo Teixeira
Aluno: Amarildo Junior

Sumário

Introdução

Para este trabalho prático da disciplina Processamento Digital de Imagens, da Universidade Federal do Piauí, foi escolhida a linguagem Python e algumas bibliotecas disponíveis para manipular dados de uma imagem, como a Pillow. O trabalho consiste em aplicar conceitos ensinados em sala de aula, como Filtragem Espacial e Morfologia Matemática, resolvendo as questões dadas.

Questão 01

A questão 01 solicitava realizar determinadas operações na imagem “*lena_gray.bmp*”.

a. Laplaciano

   Como mostra o Anexo 1.1, essa operação foi realizada pela função `filtro_laplaciano(imagem)`, que recebe uma imagem como parâmetro. A imagem pode ser a aberta com o método `Image.open()` que recebe o caminho da imagem como parâmetro. Em código é definido o filtro que será utilizado, podendo ser com o centro 4, e quatro 1’s formando uma cruz, para que a soma total resulte em 0, como -4, onde o resto dos valores serão -1 formando uma cruz e 0 nos cantos do filtro. É criada uma variável `imagem_saida` que recebe inicialmente uma imagem do mesmo tamanho da imagem de entrada e em escala de cinza, aberta com o metodo `Image.new()` da biblioteca Pillow .

   Desconsiderandos as bordas da imagem, onde não é possível é aplicar o filtro, a imagem é percorrida a partir da linha 1 e coluna 1 até a penúltima linha e coluna, pois o filtro utilizado é 3x3. A cada pixel a variável `soma` é inicializada com valor 0 e os pixels vizinhos são percorridos juntamente com o pixel central. Com o valor da variável `soma` atualizado, é necessário verificar se o pixel central é positivo ou negativo, para que seja necessário ajustar o valor para somá-lo à imagem original. Após somar o valor com a imagem original, a função retorna `imagem_saida`.

Segue abaixo as imagem resultantes da aplicação de cada filtro somada à imagem original.

É possível perceber um realce em cada imagem e que o com o filtro onde o centro é 8 ou -8, a imagem teve um realce ainda maior.

Laplaciano com o centro +4

Laplaciano com o centro -4

Laplaciano com o centro +8

Laplaciano com o centro -8

b. Unsharp Masking

Baseado na aplicação do filtro, foram utilizadas duas funções: `filtro_media(imagem)` que recebe uma imagem como parâmetro e utilizada para suavizar a imagem e a função `unsharp_masking(imagem)` que faz a operação final, também recebendo uma imagem como parâmetro, mostradas no anexo 1.2 e 1.3, respectivamente.

A função `unsharp_marsking()` começa suavizando a imagem de entrada através da função `filtro_media()`, que assim como no laplaciano percorre a imagem ignorando as bordas, e em cada pixel soma o valor dos vizinhos, porém no final divide pela soma dos valores ‘1’ no filtro. Após isso, a imagem de entrada é percorrida e o valor da mascara é calculado, subtraindo o valor do pixel da imagem de entrada com o correspondente da imagem suavizada. E então, o valor do pixel imagem de entrada somado ao da máscara é colocado na imagem de saída e ao fim dos laços de repetição, a função retorna a imagem de saída.

Abaixo estão o resultado após aplicar unsharp_masking:

Aplicação de unsharp masking

c. Filtragem Highboost

Por se tratar de uma variação do *“unsharp_masking”*, a filtragem highboost consiste em fazer o mesmo procedimento da questão anterior, diferenciando apenas pela presença de uma constante K, que multiplica o valor da máscara antes de somá-la a imagem original. Por isso, o código das duas funções são tão parecidos, praticamente idênticos, alterando apenas no fator de multiplicação K, uma vez que se K = 1, o processo é *unsharp_masking*. O código referente à função `highboost_filtering(imagem, k)` que recebe uma imagem e o valor de K como parâmetro está no anexo 1.4.

A seguir estão os resultados da aplicação da função highboost_filtering à imagem *“lena_gray.bmp”* com diferentes valores de K.

Foi possível perceber que quanto mais aumentava o K, até o número definido na aplicação, maior era riqueza de detalhes da imagem.

Highboost filtering com K = 2

Highboost filtering com K = 3

Highboost filtering com K = 5

Highboost filtering com K = 10

d. Detecção de bordas

   i. Prewitt

   Para a aplicar a detecção de bordas utilizando os operadores de Prewitt foi utilizada a função `filtro_prewitt(imagem)` que recebe uma imagem como parâmetro e descrita no anexo 1.5. Primeiramente, em código, é possível definir qual filtro será utilizado, onde os valores de 1 e -1 estão dispostos de forma horizontal e vertical. A imagem de saída é criada em escala de cinza e a imagem é percorrida da linha 1 e coluna 1 até a penúltima linha e coluna, em outras palavras, ignorando as bordas e considerando que o filtro é 3x3. A variável `soma` é inicializada em 0 e a cada pixel, os vizinhos-4 e vizinhos-8 são percorridos e multiplicados pelo correspondente na máscara, onde o resultado é adicionado à variável `soma`. Ao fim, o pixel correspondente ao pixel central recebe o valor da variável e quando todos os pixels

forem percorridos, a função retorna a imagem de saída. Seguem abaixo as máscaras aplicadas e o resultado obtido para cada uma.

- máscara a

$$\begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

- máscara b

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

Resultado da operação de Prewitt utilizando a máscara a

Resultado da operação de Prewitt utilizando a máscara b

ii. Sobel

Pode-se considerar que o processo de percorrer a imgem e aplicação do filtro para colocar o valor resultante no correspondente ao pixel central é o mesmo. O que irá mudar será nos valores das máscaras. A função para aplicar o procedimento é a filtro_sobel(imagem), situada no anexo 1.6. Assim, estas serão as máscaras aplicadas e o resultado obtido para cada uma.

- máscara a

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

máscara b

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Resultado da operação de Sobel utilizando a máscara a

Resultado da operação de Sobel utilizando a máscara b

iii. Diferença entre os dois métodos

Com os resultados obtidos, é possível perceber que os dois detectam as bordas, como esperado, porém a diferença das máscaras é que em Sobel utiliza-se -2 e 2 em duas posições específicas, onde o Prewitt utilizava apenas valores -1 e 1. Então, nas imagens resultantes destacam-se as bordas, mas as imagens resultantes de Sobel possuem as bordas ainda mais eminentes que as imagens de Prewitt.

Questão 02

A questão dois solicita a implementação do filtro da média com as máscaras abaixo na imagem *“lena_ruido.bmp”* (mostrada a seguir) e comparar com os resultados da implementação do filtro da mediana:

Imagem da lena_gray corrompida com ruído

$$\frac{1}{5}\begin{bmatrix}0 & 1 & 0\\ 1 & 1 & 1\\ 0 & 1 & 0\end{bmatrix} \quad \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \quad \frac{1}{32}\begin{bmatrix}1 & 3 & 1\\ 3 & 16 & 3\\ 1 & 3 & 1\end{bmatrix} \quad \frac{1}{8}\begin{bmatrix}0 & 1 & 0\\ 1 & 4 & 1\\ 0 & 1 & 0\end{bmatrix}$$

Máscaras para a aplicação do filtro de média

Por se tratar de uma questão diferente, a função para aplicar filtro de média foi diferente da presente no anexo 1.2. Nesse caso, a função utilizada foi a função `filtro_media()` do anexo 2.1 que recebe como parâmetro uma imagem e uma máscara para aplicar na imagem. Primeiramente, por se tratar de um filtro de média, a máscara é percorrida para se obter a soma de todos os valores, guardando assim o divisor da operação após aplicação da máscara. Então a imagem a percorrida, desconsiderando bordas e a cada pixel percorrem-se os vizinhos multiplicando o seu valor ao correspondente na máscara. Esses valores são somados e ao final divide-se pelo fator de divisão obtido anteriormente, restando apenas adicionar este resultado ao pixel central. Ao percorrer todos os pixels a função retorna a imagem de saída. Os resultados da aplicação de cada máscara estão a seguir:

Resultado da aplicação da máscara com fator $\frac{1}{5}$

Resultado da aplicação da máscara com fator $\frac{1}{9}$

Resultado da aplicação da máscara com fator $\frac{1}{32}$

Resultado da aplicação da máscara com fator $\frac{1}{8}$

Para a aplicação do filtro da mediana, foram utilizadas duas funções: a principal `filtro_mediana()` , anexo 2.2, que recebe uma imagem como parâmetro e `mediana()` , descrita no anexo 2.3, que recebe uma lista como parâmetro e retorna a mediana dos valores. Na função para aplicar o filtro, basicamente a imagem é percorrida sobre os eixos x e y, descartando onde o filtro 3x3 não pode ser aplicado, e a cada pixel cria-se uma lista, inicialmente vazia, que será preenchida com os valores de todos os pixels correspondentes à mascara 3x3. Ao fim pe calculada a mediana dos valores com a função auxiliar citada anteriormente e o resultado é armazenado ao pixel na imagem de saída correspondente ao pixel central. Por se tratar de um processo empírico, o filtro foi aplicado mais de uma vez e os resultados obtidos foram os seguintes:

Aplicação do filtro de mediana pela primeira vez

Aplicação do filtro de mediana duas vezes

Aplicação do filtro de mediana três vezes

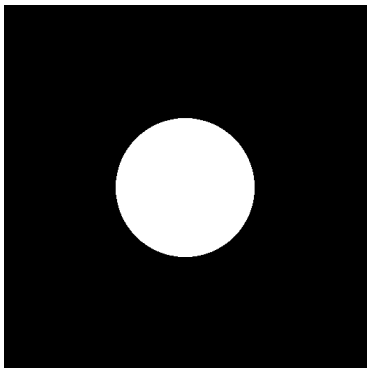
Aplicação do filtro de mediana quatro vezes

> Ilustramente, é notório que o filtro de mediana se sobressaiu ao filtro de média, considerando principalmente o tipo de ruído na imagem original. Entretanto, mesmo que não mostrados os resultados neste relatório, o filtro de média também foi aplicado mais de uma vez para cada um dos valores, mas para todos a imagem escureceu ainda mais e não removeu ruído.

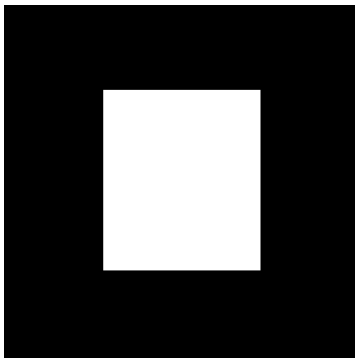
Questão 03

Esta questão solicita funções para realizar as operações morfológicas de União, Interseção e Diferença, considerando que as imagens são objetos brancos em um fundo preto.

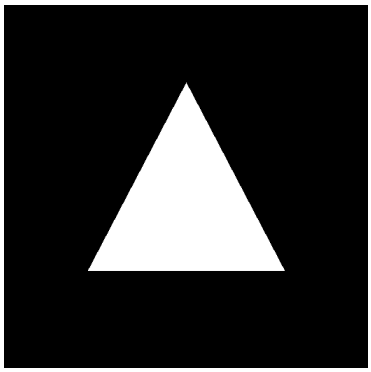
Para efeitos de prática e visualização de resultados, foram criadas as seguintes imagens manualmente:



círculo branco em um fundo preto



quadrilátero branco em um fundo preto

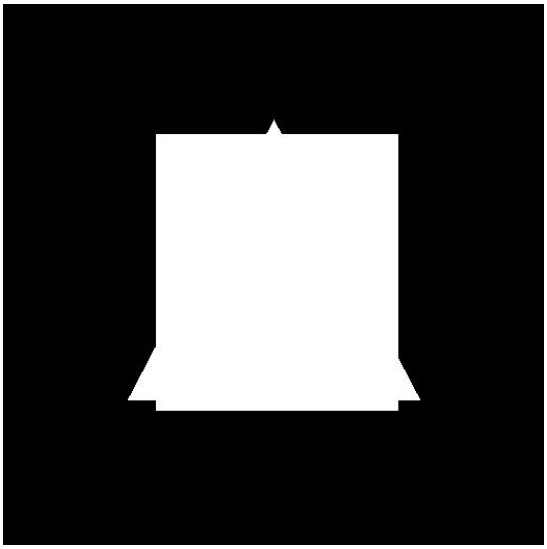


triângulo branco em um fundo preto

a. União

   Para a aplicar essa operação foi criada a função `uniao()` , situada no anexo 3.1, que recebe como parâmetro duas imagens. Primeiramente, tendo a certeza de que as duas imagens possuem o mesmo tamanho uma imagem de saída é criada com todos os pixels pretos e o valor de intensidade máxima é calculado através da função `max_intensidade()` - anexo 3.2, que basicamente percorre uma imagem recebida como parâmetro e retorna o maior valor dentre os pixels. São realizados dois laços de repetição para percorrer os eixos x e y e os pixels correspondentes à posição [x, y] são comparados, caso um dos dois sejam iguais a intensidade máxima, ou seja, possuem como valor a cor branca, então o pixel na mesma posição da imagem de saída recebe branco como seu valor.

   Os resultados dessa operação nas imagens citadas anteriormente seguem abaixo:



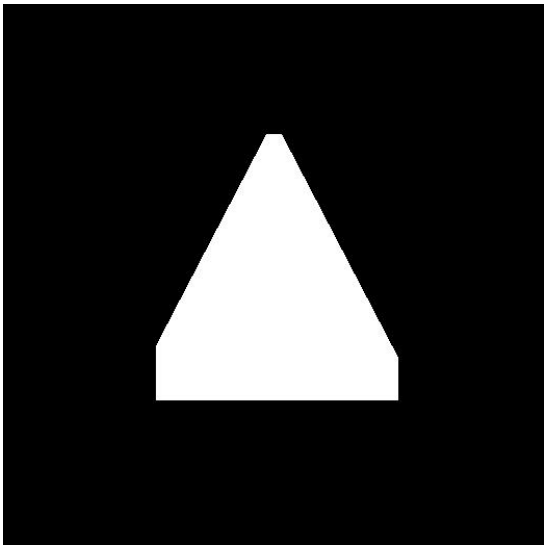
União entre as imagens do quadrilátero e triângulo

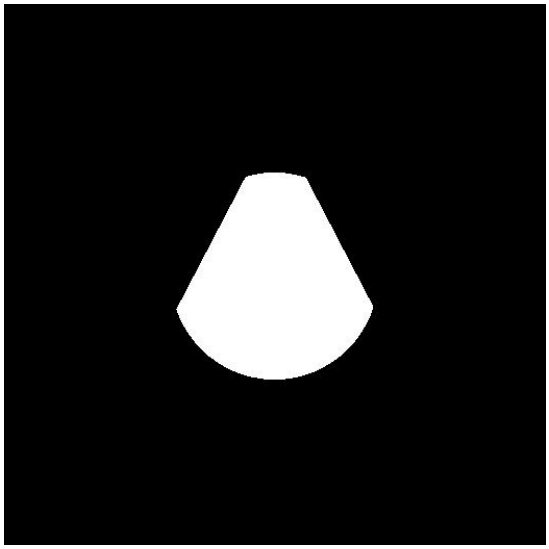
União entre as imagens do círculo e triângulo

b. Interseção

É inegável que a união e interseção são operações bem diferentes, mas que o processo de percorrer as imagens é praticamente o mesmo. Assim, a função `intersecao()` - anexo 3.3 -compara o valor dos pixels das duas imagens e se os dois corresponderem ao valor de intensidade máxima, o pixel na posição [x, y] na imagem de saída tem seu valor alterado para a intensidade máxima, caso contrário o pixel permanece com o pixel preto. Ao final dos laços a função retorna a imagem de saída.

Segue abaixo os resultados da aplicação dessa função nas imagens citadas para visualizar o resultado:





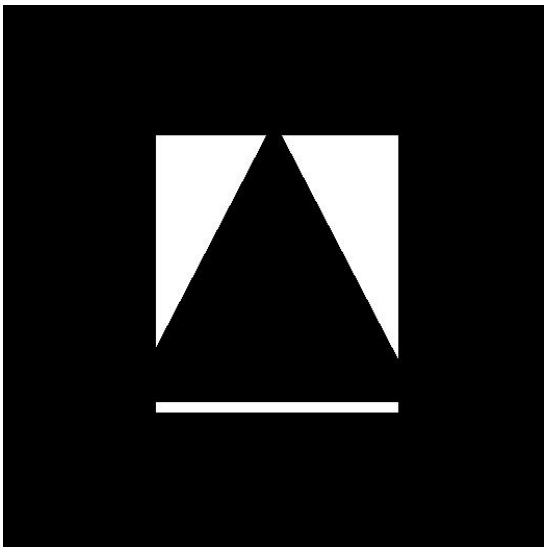
Interseção entre as imagens do círculo e triângulo

Interseção entre as imagens do quadrilátero e triângulo

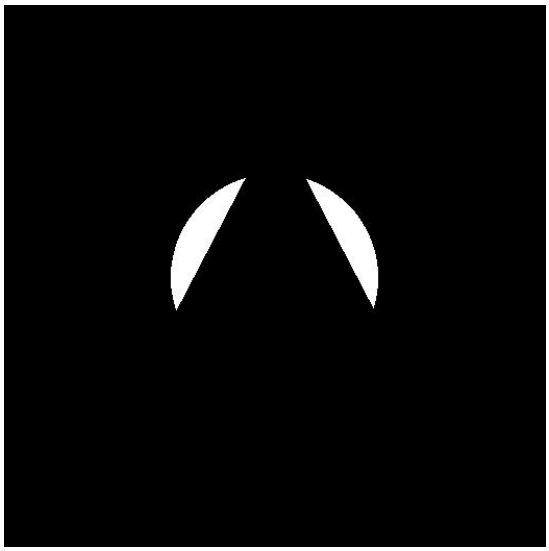
c. Diferença

Para aplicar a operação de diferença foi utilizada a função `diferenca()` - anexo 3.4. Esta função recebe como parâmetro duas imagens e assim como nas anteriores certifica que elas tem o mesmo tamanho. Após isso, o valor do pixel [x, y] na imagem de saída irá corresponder a diferença dos pixels na mesma posição da imagem 1 e imagem 2, respectivamente. Ao final, a função retorna a imagem de saída.

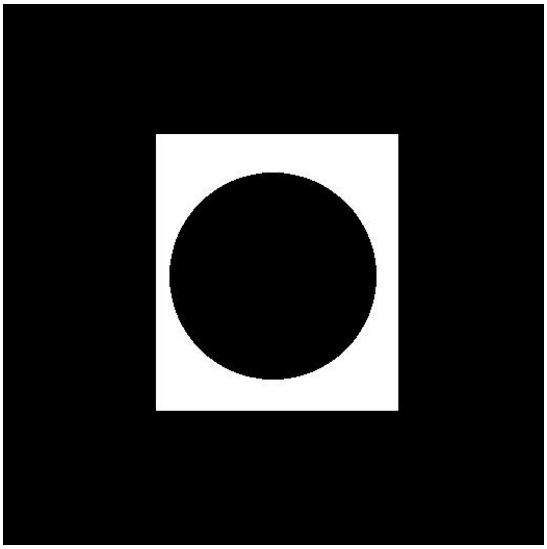
Abaixo estão os resultados da aplicação dessa operação nas imagens de quadrilátero, triângulo e círculo.



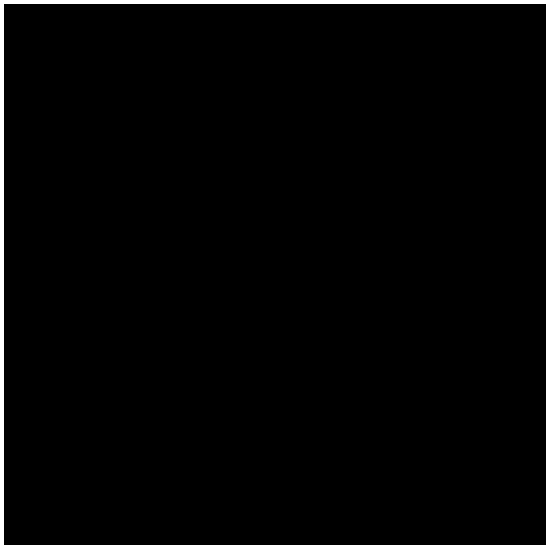
Diferença entre as imagens do quadrilátero e triângulo, respectivamente



Diferença entre as imagens do círculo e triângulo, respectivamente



Diferença entre as imagens do quadrilátero e círculo, respectivamente

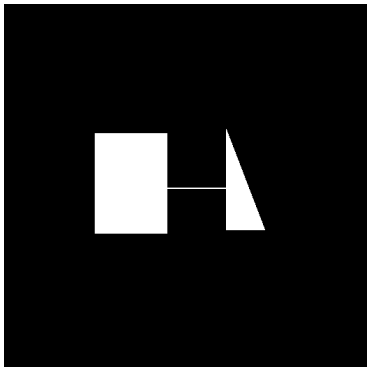


Diferença entre as imagens do círculo e quadrilátero, respectivamente

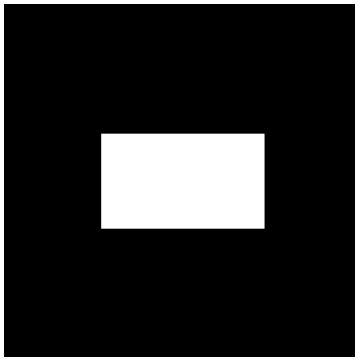
Questão 04

Assim como na questão 03, a questão 04 requer a criação de funções para realizar operações morfológicas, desta vez: Dilatação, Erosão, Abertura e Fechamento.

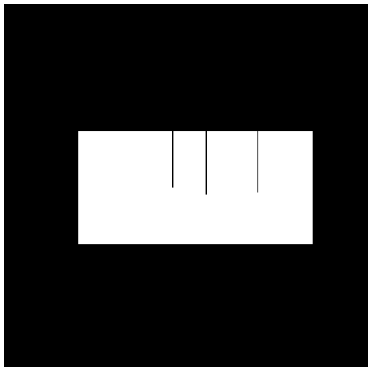
Para efeitos de prática e visualização de resultados, para essa questão também foram criadas algumas imagens manualmente:



Duas formas conectadas por um traço de largura 3 px.



Retângulo 1 - totalmente preenchido



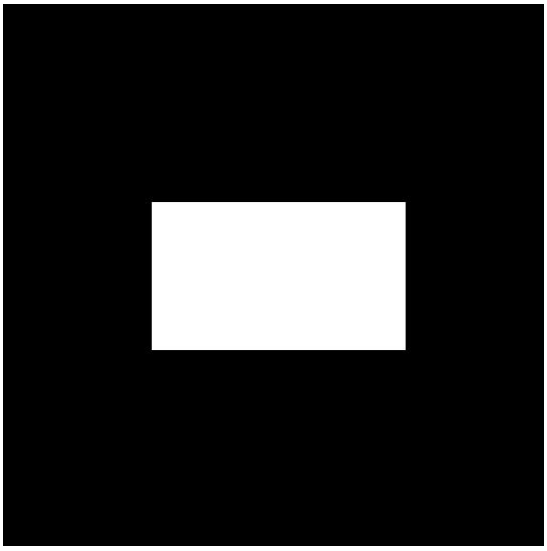
Retângulo 2 - com algumas partes misturadas ao fundo

a. Dilatação

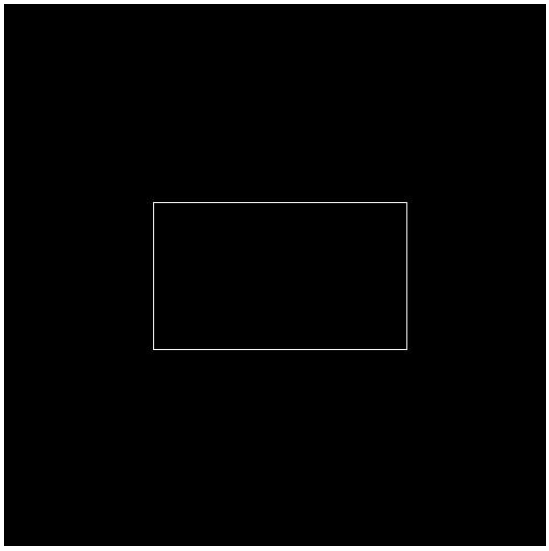
   Para realizar a operação de dilatação foi criada a função `dilatacao()` - anexo 4.1 - que recebe como parâmetro uma imagem, um elemento estruturante e seu centro. De início, é aberta uma imagem em escala de cinza do mesmo tamanho da imagem de entrada, com o

fundo totalmente preto. A intensidade máxima da imagem é obtida através da função `max_intensidade()` , situada no anexo 3.2. E então a imagem é percorrida e para os pixels em que o valor for igual a intensidade máxima da imagem, ou seja, pertencente ao objeto, será considerado o pixel central e os vizinhos serão percorridos de acordo com a correspondência do elemento estruturante. Onde se encontrar 1 no elemento estruturante o pixel na imagem de saída receberá o valor de intensidade máxima. Ao final a função retornará a imagem de saída.

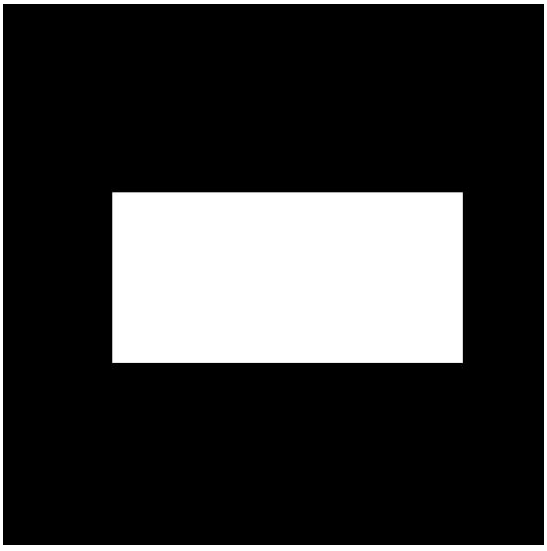
Segue abaixo os exemplos da dilatação aplicada aos retângulos com um elemento estruturante 3x3 composto apenas por 1’s:



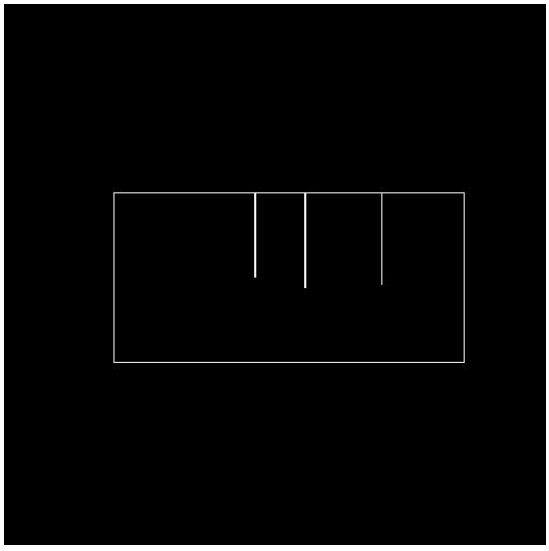
Retângulo 1 dilatado com um elemento 3x3 composto apenas por 1’s



Diferença entre o Retângulo 1 dilatado e a imagem com o retângulo original



Retângulo 2 dilatado com um elemento 3x3 composto apenas por 1’s

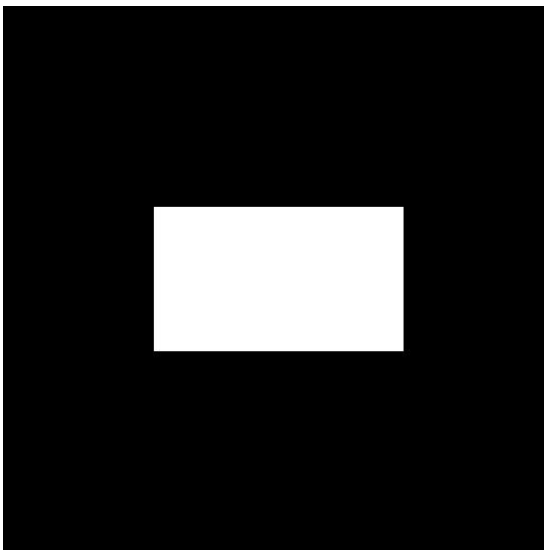


Diferença entre o Retângulo 2 dilatado e a imagem com o retângulo original

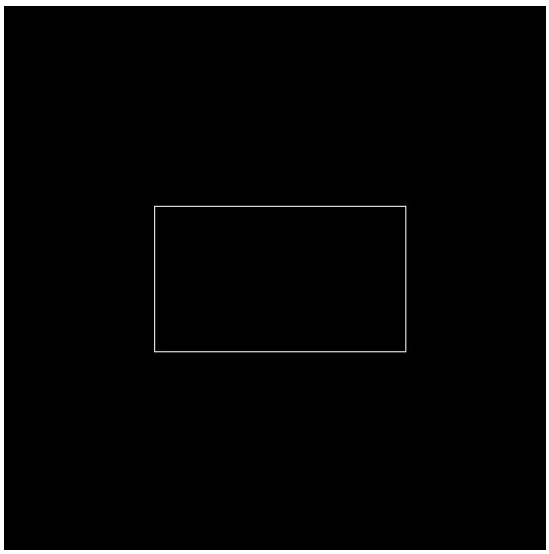
b. Erosão

A função criada para realizar essa operação foi a de `erosao()` , anexo 4.2, que recebe como parâmetro uma imagem, um elemento estruturante e o seu centro. Assim como na dilatação, o processo de criar uma imagem de saída com fundo preto e tamanho igual ao da entrada, percorrer a imagem de entrada é o mesmo, a diferença é o que considerar quando percorrer todos os pixels. Nesse caso, primeiro uma variável booleana `elemento_est_encontrado` é criada inicialmente com o valor True e para cada pixel da imagem de entrada os vizinhos são percorridos de acordo com o elemento estruturante. Se em determinado momento o elemento estruturante tiver o valor 1 e o pixel correspondente não fazer parte do objeto, então a variável `elemento_est_encontrado` irá para False. Após percorrer os pixels da imagem de entrada, caso a variável seja igual a True, significa que o elemento estrurante corresponde àquela região da imagem, então apenas o pixel central na imagem de saída recebe o valor de intensidade máxima. No fim da função, a imagem de saída é retornada.

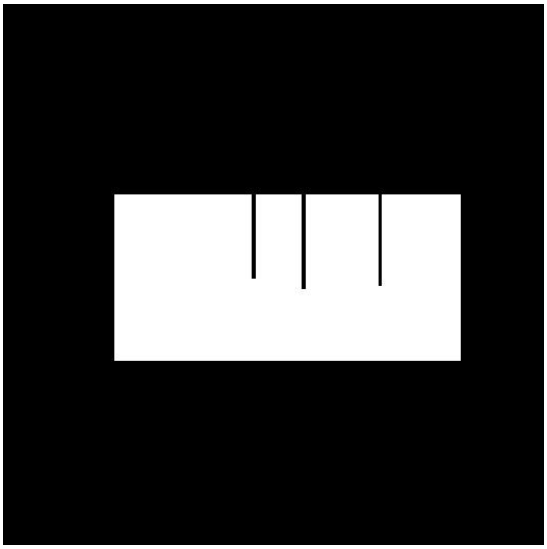
Aqui estão alguns exemplos da erosão aplicada aos retângulos com um elemento estruturante 3x3 composto apenas por 1’s:



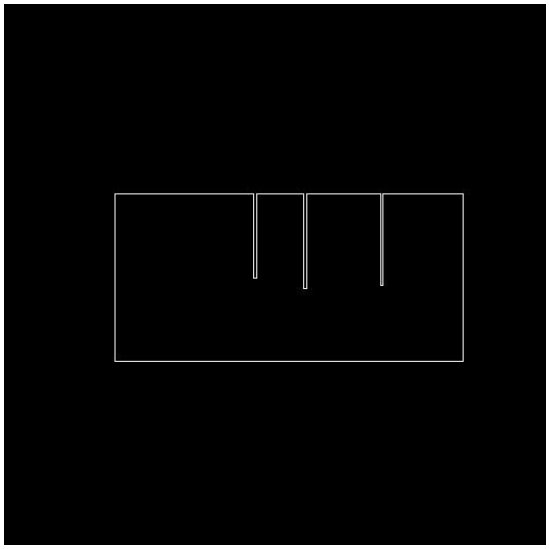
Retângulo 1 erodido com um elemento 3x3 composto apenas por 1’s



Diferença entre a imagem original com o Retângulo 1 erodido



Retângulo 2 erodido com um elemento 3x3 composto apenas por 1’s



Diferença entre a imagem original com o Retângulo 2 erodido

c. Abertura

Por se tratar de uma operação que combina as outras duas operações anteriores, a função `abertura()` , anexo 4.3, recebe como parâmetro uma imagem, um elemento estruturante e o seu centro. Basicamente é feita uma erosão na imagem de entrada (anexo 4.2) e com o resultado desta é feita uma dilatação (anexo 4.1). Então esta imagem de saída é retornada.

A primeira imagem exemplo desta questão recebeu um processo de abertura, com um elemento estruturante 3x3 composto apenas por 1’s:

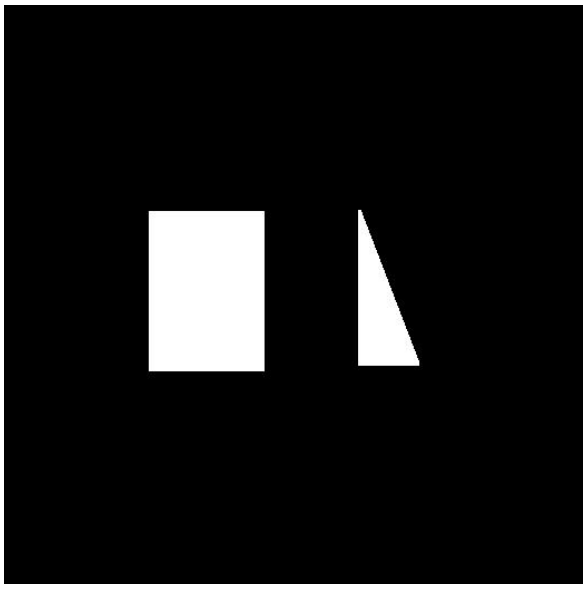


Imagem de uma abertura na figura com elementos conectados por um traço de 3 pixels

d. Fechamento

Assim como na abertura, o fechamento também é uma combinação das operações de erosão e dilatação, mudando apenas a ordem em que são aplicadas quando comparada à abertura. Foi criada a função `fechamento()` - anexo 4.4 - que recebe uma imagem, um elemento estruturante e seu centro, e de maneira simples: é feita uma dilatação na imagem de entrada (anexo 4.1) e com o resultado desta é feita uma erosão (anexo 4.2). Ao final, a função retorna a imagem de saída.

Foi realizado um fechamento na imagem contendo o Retângulo 2, com buracos, utilizando um elemento estruturante 3x3 contendo apenas 1:

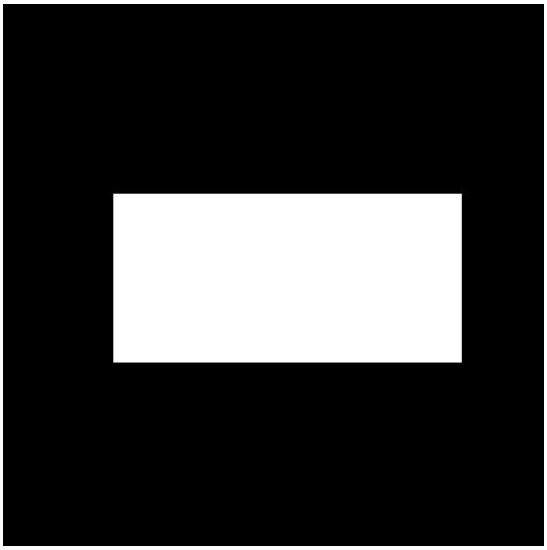
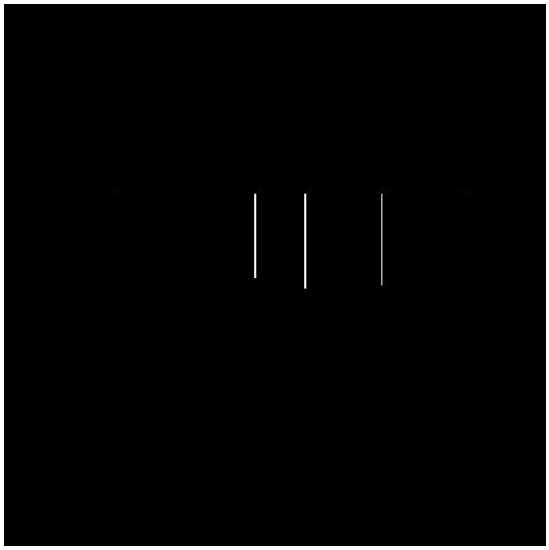


Imagem do retângulo 2 onde foi realizado um fechamento com elemento estruturante 3x3 contendo apenas 1



Diferença entre a imagem onde foi realizado o fechamento e a imagem original. Percebe-se que sobraram apenas os buracos que foram preenchidos

Questão 05

A questão 5 solicita que sejam realizados alguns procedimentos em cima da imagem “quadro.png”.

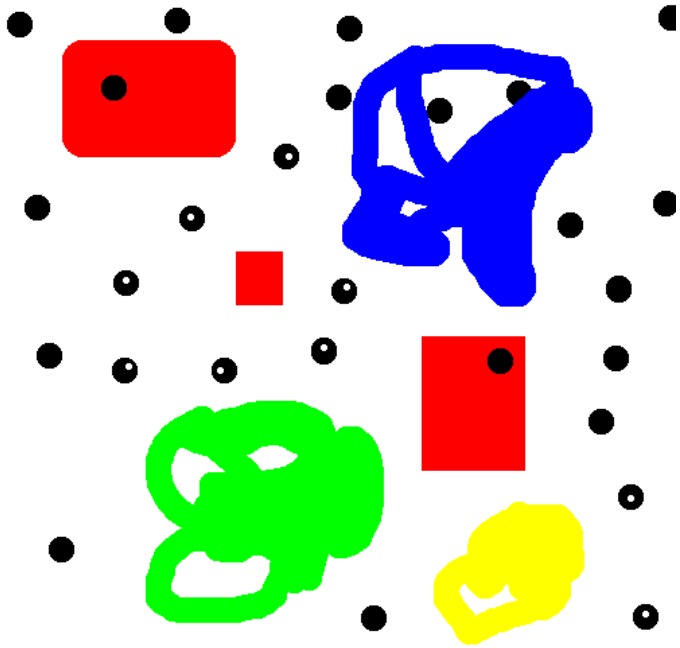


Imagem “quadro.png” original

a) Preencher todos os buracos dos objetos pretos

Nessa alternativa, para manter a organização foi necessário o uso de algumas funções criadas como auxiliares. A função principal `letra_a()` se encontra no anexo 5.1. Primeiramente, como o processo consiste em preencher apenas os buracos dos objetos pretos, então foi ideal separá-los da imagem original com a função `separar_elementos_cor()` - anexo 5.2 -passando como parâmetros uma imagem e a cor dos pixels a serem destacados. Após isso, os buracos dos objetos pretos serão preenchidos através da função `fechamento()` (anexo 4.4) que recebe como parâmetro a imagem com os elementos separados, um elemento estruturante e seu centro, para que seja possível realizar um fechamento na imagem e retorná-la. Com a imagem dos buracos pretos preenchidos, basta somá-la à imagem original.

Segue abaixo o resultado final da operação:

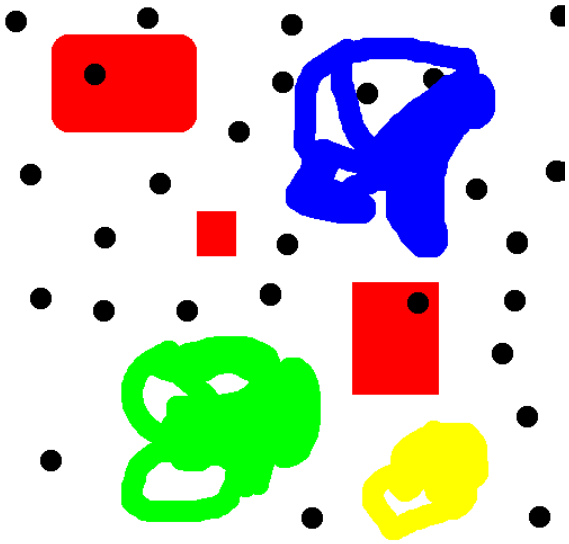


Imagem “quadro.png” com os elementos pretos completamente preenchidos.

b) Eliminar todos e somente os objetos pretos

A alternativa foi resolvida apenas utilizando a função `remover_elementos_cor()` (anexo 5.4), que recebe uma imagem e a cor dos elementos que irá remover na imagem. Assim, a imagem é percorrida e os elementos que não são da cor especificada permanecem na imagem de saída e os que são, nesse caso, acabam sendo considerados como parte do fundo da imagem (branco).

O resultado após a realização dessa operação foi:

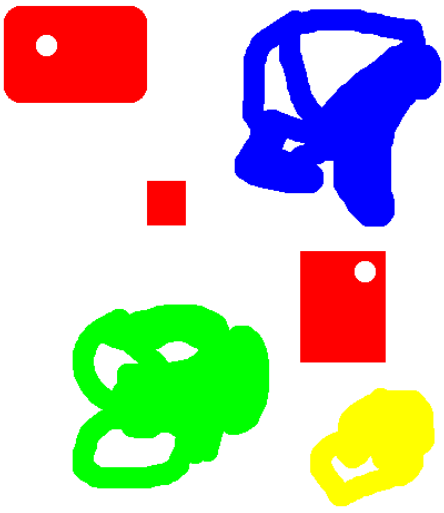


Imagem “quadro.png” sem os objetos pretos.

c) Preencher os buracos dos objetos de cor: azul, amarelo e verde

Essa alternativa gerou um pouco de dificuldade, na questão de identificar em que partes poderiam ser considerados um buraco. Nesse caso, foi utilizada a função `letra_c()` (anexo 5.5) que recebe uma imagem como parâmetro e a converte pra RGB. Primeiramente, os elementos de cada cor citada são separados em imagens diferentes com a função já citada `separar_elementos_cor()`. Após isso, com a função `preencher_regioes()` (anexo 5.6) é possível procurar regiões que tenham pixels brancos e estejam cercadas (por cima, baixo, esquerda, direita) por pixels da cor do objeto, que nesse caso poderia ser considerado, de forma rústica, um buraco no objeto. Então a função sai preenchendo os valores pelas coordenadas que definem o intervalo em que o buraco está presente e retorna a imagem com o objeto totalmente preenchido. Depois disso, só resta somar as imagens com cada objeto preenchido à imagem original, o que será feito com a função adaptada `somar_imagens()` (anexo5.7) que recebe duas imagens como parâmetros e sobrepõe o objeto da segunda na primeira. Entretanto, por haver casos em que essa verificação de existências de buracos não é completa, o objeto fica suavemente maior, não sendo totalmente ideal.

Segue abaixo o resultado da aplicação da função:

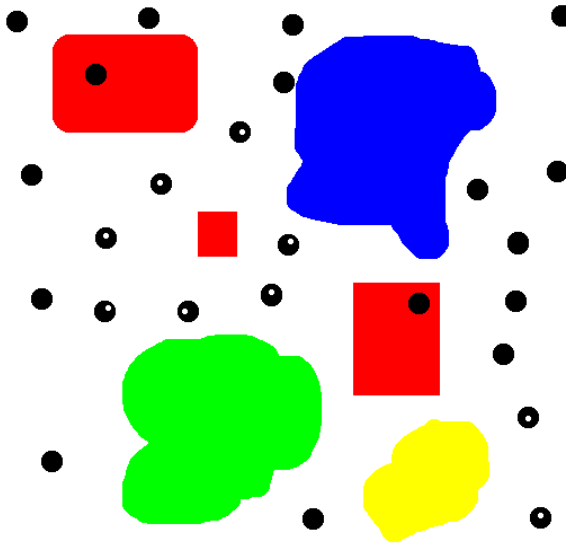


Imagem “quadro.png” sem os objetos amarelo, azul e verde preenchidos.

Anexos

Anexo 1.1 : Função para aplicar o filtro laplaciano em uma imagem

```
def filtro_laplaciano(imagem):
    filtro = [[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]
    # filtro = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]

    # filtro = [[0, -1, 0], [-1, 4, -1], [0, -1, 0]]
    # filtro = [[0, 1, 0], [1, -4, 1], [0, 1, 0]]

    imagem_saida = Image.new("L", imagem.size)

    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
            # Soma entre o produto do filtro e a imagem
            soma = 0
            for i in range(3):
                for j in range(3):
                    soma += imagem.getpixel((x + i - 1, y + j - 1)) * filtro[i][j]
            if (filtro[len(filtro) // 2][len(filtro) // 2] < 0):
                soma = -soma
            imagem_saida.putpixel((x, y), imagem.getpixel((x, y)) + soma)

    return imagem_saida
```

Anexo 1.2 : Código referente à aplicação do filtro da média

```
def filtro_media(imagem):
    filtro = [[1, 1, 1], [1, 1, 1], [1, 1, 1]]

    imagem_saida = Image.new("L", imagem.size)

    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
            sum = 0
            for i in range(3):
                for j in range(3):
                    sum += image.getpixel((x + i - 1, y + j - 1)) * filtro[i][j]
            imagem_saida.putpixel((x, y), int (sum / 9))

    return imagem_saida
```

Anexo 1.3 : Código para aplicar unsharp_masking

```
def unsharp_masking(imagem):
    imagem_suavizada = filtro_media(imagem)
    imagem_saida = Image.new("L", imagem.size)
```

```
    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
            mascara_g = imagem.getpixel((x, y)) - imagem_suavizada.getpixel((x, y))
            imagem_saida.putpixel((x, y), int(imagem.getpixel((x, y)) + mascara_g))

    return imagem_saida
```

Anexo 1.4: Função para aplicar highboost filtering

```
def highboost_filtering(imagem, k):

    filtro = [[1, 1, 1], [1, 1, 1], [1, 1, 1]]

    imagem_suavizada = filtro_media(imagem, filtro)
    imagem_saida = Image.new("L", imagem.size)

    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
            mascara_g = imagem.getpixel((x, y)) - imagem_suavizada.getpixel((x, y))
            imagem_saida.putpixel((x, y), int(imagem.getpixel((x, y)) + mascara_g *
 k))

    return imagem_saida
```

Anexo 1.5: Detecção de bordas com Prewitt

```
def filtro_prewitt(imagem):
    # filtro = [[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]
    filtro = [[-1, -1, -1], [0, 0, 0], [1, 1, 1]]
    imagem_saida = Image.new("L", imagem.size)
    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
            soma = 0
            for i in range(3):
                for j in range(3):
                    soma += imagem.getpixel((x + i - 1, y + j - 1)) * filtro[i][j]
            imagem_saida.putpixel((x, y), int(soma))
    return imagem_saida
```

Anexo 1.6: Detecção de bordas utilizando Sobel

```
def filtro_sobel(imagem):
    filtro = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
    # filtro = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]
    imagem_saida = Image.new("L", imagem.size)
    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
```

```
            soma = 0
            for i in range(3):
                for j in range(3):
                    soma += imagem.getpixel((x + i - 1, y + j - 1)) * filtro[i][j]
            imagem_saida.putpixel((x, y), int(soma))
    return imagem_saida
```

Anexo 2.1: Aplicação do filtro de média

```
def filtro_media(imagem, mascara):
    n = 0

    for mascara_i in range(len(mascara)):
        for mascara_j in range(len(mascara[0])):
            n += mascara[mascara_i][mascara_j]

    imagem_saida = Image.new("L", imagem.size)

    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
            sum = 0
            for i in range(3):
                for j in range(3):
                    sum += imagem.getpixel((x + i - 1, y + j - 1)) * mascara[i][j]
            imagem_saida.putpixel((x, y), int (sum / 9))

    return imagem_saida
```

Anexo 2.2: Aplicação do filtro da mediana

```
def filtro_mediana(imagem):
    imagem_saida = Image.new("L", imagem.size)

    px = imagem.load()

    for x in range(1, imagem.width - 1):
        for y in range(1, imagem.height - 1):
            lista = []
            for i in range(3):
                for j in range(3):
                    lista.append(px[x + i - 1, y + j - 1])
            imagem_saida.putpixel((x, y), mediana(lista))

    return imagem_saida
```

Anexo 2.3: Calcula a mediana a partir de uma lista de valores

```
def mediana(lista):
    lista.sort()
    if len(lista) % 2 == 0:
        return (lista[len(lista) // 2] + lista[len(lista) // 2 - 1]) / 2
    else:
        return lista[len(lista) // 2]
```

Anexo 3.1: Operação de União entre duas imagens

```
def uniao(imagem1, imagem2):
    if imagem1.size > imagem2.size:
        imagem2.resize(imagem1.size)
    elif imagem2.size > imagem1.size:
        imagem1.resize(imagem2.size)

    imagem3 = Image.new('L', imagem1.size, "black")

    intensidade_max = max_intensidade(imagem1)

    px1 = imagem1.load()
    px2 = imagem2.load()

    for x in range(imagem3.width):
        for y in range(imagem3.height):
            if px1[x, y] == intensidade_max or px2[x, y] == intensidade_max:
                imagem3.putpixel((x, y), intensidade_max)

    return imagem3
```

Anexo 3.2: Função que retorna o maior valor de intensidade em uma imagem

```
def max_intensidade(imagem):
    max = 0
    for x in range(imagem.width):
        for y in range(imagem.height):
            if imagem.getpixel((x, y)) > max:
                max = imagem.getpixel((x, y))
    return max
```

Anexo 3.3: Operação de Interseção entre duas imagens

```
def intersecao(imagem1, imagem2):
    if imagem1.size > imagem2.size:
        imagem2.resize(imagem1.size)
    elif imagem2.size > imagem1.size:
```

```
        imagem1.resize(imagem2.size)

    imagem3 = Image.new('L', imagem1.size)

    intensidade_max = max_intensidade(imagem1)

    px1 = imagem1.load()
    px2 = imagem2.load()

    for x in range(imagem3.width):
        for y in range(imagem3.height):
            if px1[x, y] == intensidade_max and px2[x, y] == intensidade_max:
                imagem3.putpixel((x, y), intensidade_max)

    return imagem3
```

Anexo 3.4: Operação de Diferença entre duas imagens

```
def diferenca(imagem1, imagem2):
    if imagem1.size > imagem2.size:
        imagem2.resize(imagem1.size)
    elif imagem2.size > imagem1.size:
        imagem1.resize(imagem2.size)

    imagem3 = Image.new('L', imagem1.size)

    px1 = imagem1.load()
    px2 = imagem2.load()

    for x in range(imagem3.width):
        for y in range(imagem3.height):
            imagem3.putpixel((x, y), px1[x, y] - px2[x, y])

    return imagem3
```

Anexo 4.1: Função para aplicar a operação de Dilatação

```
def dilatacao(imagem, elemento_estruturante, centro):
    imagem_saida = Image.new("L", imagem.size)

    intensidade_max = max_intensidade(imagem)

    for x in range(imagem_saida.width):
        for y in range(imagem_saida.height):
            if(imagem.getpixel((x, y)) == intensidade_max):
                for i in range(len(elemento_estruturante)):
                    for j in range(len(elemento_estruturante[i])):
                        if(elemento_estruturante[i][j] == 1):
                            imagem_saida.putpixel((x + i - centro[0], y + j - centro
[1]), intensidade_max)
```

```
    return imagem_saida
```

Anexo 4.2: Função para aplicar a operação de Erosão

```
def erosao(imagem, elemento_estruturante, centro):
    imagem_saida = Image.new("L", imagem.size)

    intensidade_max = max_intensidade(imagem)
    for x in range(imagem_saida.width):
        for y in range(imagem_saida.height):
            if imagem.getpixel((x, y)) == intensidade_max:
                elemento_est_encontrado = True
                for i in range(len(elemento_estruturante)):
                    for j in range(len(elemento_estruturante[i])):
                        if elemento_estruturante[i][j] == 1:
                            if not imagem.getpixel((x + i - centro[0], y + j - centro
[1])) == intensidade_max:
                                elemento_est_encontrado = False

                if elemento_est_encontrado:
                    imagem_saida.putpixel((x, y), intensidade_max)
    return imagem_saida
```

Anexo 4.3: Função para aplicar a operação de Abertura

```
def abertura(imagem, elemento_estruturante, centro):
    imagem_erodida = erosao(imagem, elemento_estruturante, centro)
    imagem_saida = dilatacao(imagem_erodida, elemento_estruturante, centro)
    return imagem_saida
```

Anexo 4.4: Função para aplicar a operação de Fechamento

```
def fechamento(imagem, elemento_estruturante, centro):
    imagem_dilatada = dilatacao(imagem, elemento_estruturante, centro)
    imagem_saida = erosao(imagem_dilatada, elemento_estruturante, centro)
    return imagem_saida
```

Anexo 5.1: Função principal da alternativa ‘a’ da questão 5

```
def letra_a(imagem):
    elemento_est = [[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1,
 1], [1, 1, 1, 1, 1]]
    centro = (2, 2)
```

```
    elementos_separados = separar_elementos_cor(imagem, (0, 0, 0))

    buracos_preenchidos = preencher_buracos(elementos_separados, elemento_est, centro)

    imagem_final = somar_imagens(imagem, buracos_preenchidos)

    return imagem_final
```

Anexo 5.2: Função para destacar apenas os elementos de uma determinada cor

```
def separar_elementos_cor(imagem, cor):
    imagem_saida = Image.new("RGB", imagem.size, "white")
    pixels = imagem.load()

    for x in range(imagem.width):
        for y in range(imagem.height):
            if pixels[x, y] == cor:
                imagem_saida.putpixel((x, y), cor)

    return imagem_saida
```

Anexo 5.3: Função auxiliar para somar duas imagens

```
def somar_imagens(imagem1, imagem2, cor):
    imagem_saida = imagem1.copy()
    pixels1 = imagem1.load()
    pixels2 = imagem2.load()

    for x in range(imagem_saida.width):
        for y in range(imagem_saida.height):
            if pixels1[x, y] != pixels2[x, y] and pixels2[x, y] == cor:
                imagem_saida.putpixel((x, y), pixels2[x, y])

    return imagem_saida
```

Anexo 5.4: Função para remover os elementos de uma determinada cor

```
def remover_elementos_cor(imagem, cor):
    imagem_saida = Image.new("RGB", imagem.size, "white")
    pixels = imagem.load()

    for x in range(imagem.width):
        for y in range(imagem.height):
            if pixels[x, y] != cor:
                imagem_saida.putpixel((x, y), pixels[x, y])

    return imagem_saida
```

Anexo 5.5: Função principal da alternativa ‘c’ da questão 5

```
def letra_c(imagem):
    imagem = imagem.convert("RGB")

    # # preenchendo buracos verdes
    elementos_verdes = separar_elementos_cor(imagem, (0, 255, 0))
    verdes_preenchidos = preencher_regioes(elementos_verdes, (0, 255, 0))

    elementos_azuis = separar_elementos_cor(imagem, (0, 0, 255))
    azuis_preenchidos = preencher_regioes(elementos_azuis, (0, 0, 255))

    elementos_amarelos = separar_elementos_cor(imagem, (255, 255, 0))
    amarelos_preenchidos = preencher_regioes(elementos_amarelos, (255, 255, 0))

    soma = somar_imagens(imagem, azuis_preenchidos)
    soma = somar_imagens(soma, amarelos_preenchidos)
    soma = somar_imagens(soma, verdes_preenchidos)

    return soma
```

Anexo 5.6: Função para preencher regiões com objetos de determinada cor

```
def preencher_regioes(imagem, cor):
    imagem_saida = imagem.copy()

    pixels = imagem.load()

    for x in range(imagem.width):
        for y in range(imagem.height):
            buraco_dir = False
            buraco_esq = False
            buraco_cima = False
            buraco_baixo = False
            pixel_buraco = []
            if pixels[x, y] == cor:
                primeira_exe = True
                for i in range(x+1, imagem.width):
                    if pixels[i, y] == cor:
                        if primeira_exe:
                            primeira_exe = False
                            continue
                        else:
                            buraco_dir = True
                            pixel_buraco.append(i)
                            break

                if buraco_dir:
                    primeira_exe = True
```

```
                    for j in range(y+1, imagem.height):
                        if pixels[i, y] == cor:
                            if primeira_exe:
                                primeira_exe = False
                                continue
                            else:
                                buraco_baixo = True
                                pixel_buraco.append(j)
                                break

                if buraco_dir and buraco_baixo:

                    primeira_exe = True
                    for j in range(y-1, 0, -1):
                        if pixels[i, y] == cor:
                            if primeira_exe:
                                primeira_exe = False
                                continue
                            else:
                                buraco_cima = True
                                pixel_buraco.append(j)
                                break

                if buraco_dir and buraco_baixo and buraco_cima:

                    primeira_exe = True
                    for i in range(x-1, 0, -1):
                        if pixels[i, y] == cor:
                            if primeira_exe:
                                primeira_exe = False
                                continue
                            else:
                                buraco_esq = True
                                pixel_buraco.append(i)
                                break
                        primeira_exe = False

            if buraco_dir and buraco_baixo and buraco_esq and buraco_cima:
                for i in range(pixel_buraco[3], pixel_buraco[0]):
                    for j in range(pixel_buraco[2], pixel_buraco[1]):
                        imagem_saida.putpixel((i, j), cor)
            pixel_buraco.clear()

    return imagem_saida
```

Anexo 5.7: Função para sobrepor um objeto de uma em outra

```
def somar_imagens(imagem1, imagem2):
    imagem_saida = imagem1.copy()
    pixels2 = imagem2.load()

    for i in range(imagem_saida.width):
        for j in range(imagem_saida.height):
            if pixels2[i, j] != (255, 255, 255):
```

```
                imagem_saida.putpixel((i, j), pixels2[i, j])

    return imagem_saida
```