Processamento de Imagens - Projeto final

Marcos Heitor Carvalho de Oliveira

Dezembro - 2023

QUESTÕES

1ª Questão

Imagem	Ponto de Corte
doc1	50
doc2	70

Tabela 1 – Pontos de corte manuais para os documentos 1 e 2

Na tabela 1 obtemos manualmente o ponto de corte através da visualização no histograma das imagens doc1 e doc2.

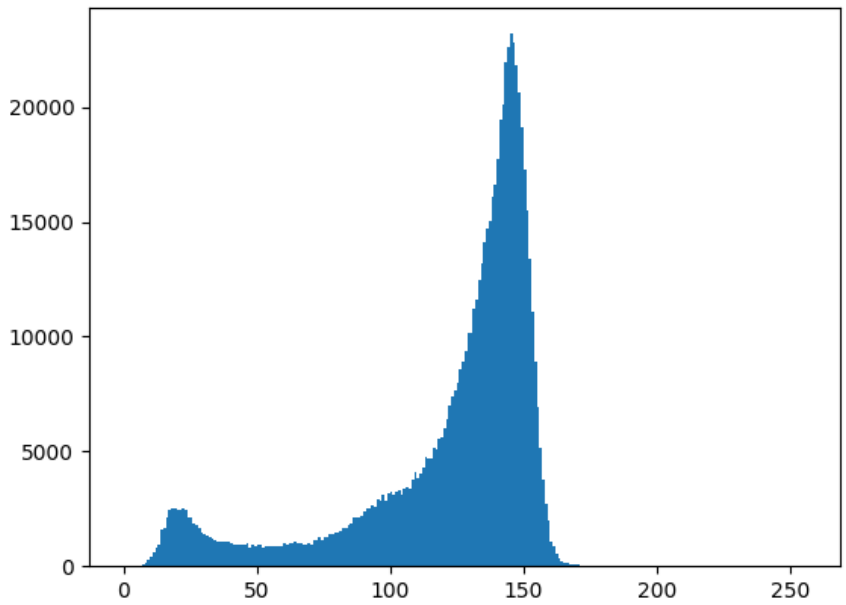

Figura 1 – Histograma para doc1.

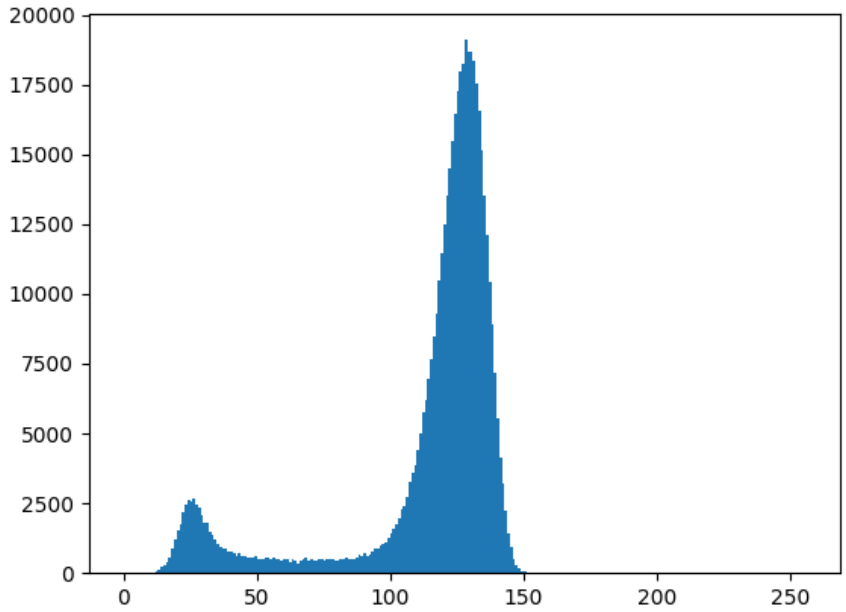

Figura 2 – Histograma para doc2.

Como podemos observar pelas figuras 1 e 2 um ponto de corte entre os dois toms mais frequentes da imagem deve gerar uma boa binarização. Escolhi os valores 50 e 70 pois foram os que mais me agradaram nas imagens binarizadas, como mostrado nas figuras 3 e 4 abaixo:

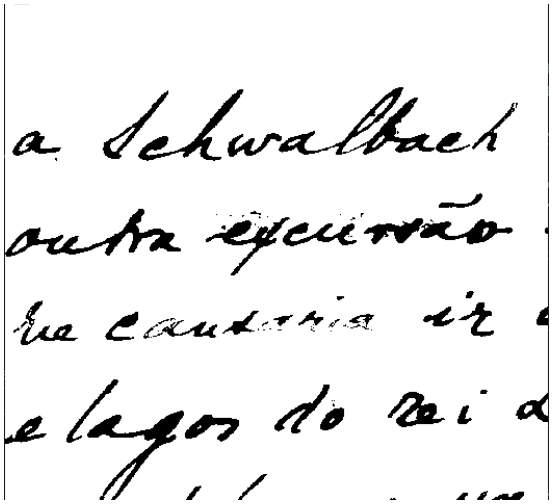

Figura 3 – Doc1 binarizado.

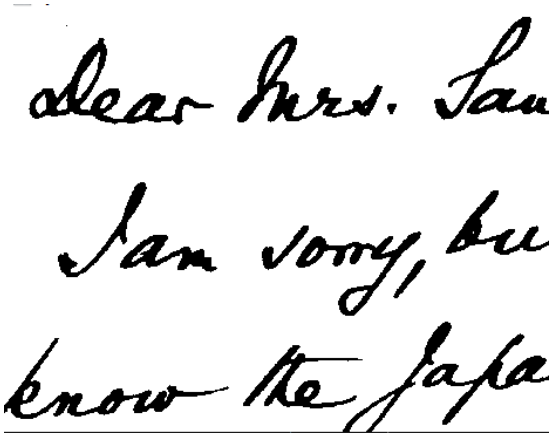

Figura 4 – Doc2 binarizado.

Trocando os pontos de corte obtivemos bons resultados, pois ambos os histogramas das imagens eram semelhantes. As duas imagens com os pontos de corte trocados estão como nas figuras 5 e 6 abaixo:

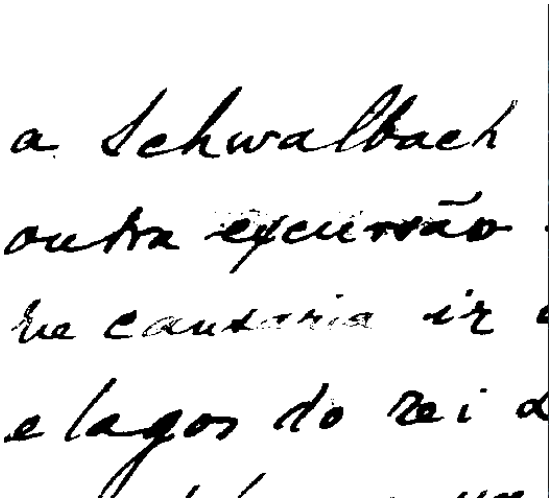

Figura 5 – Doc1 binarizado com o ponto de corte de doc2.

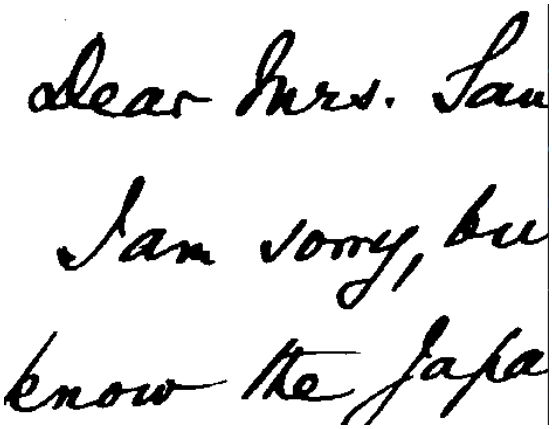

Figura 6 – Doc2 binarizado com o ponto de corte de doc1.

Para esse tipo de imagem textual, esperamos muitos pontos próximos de um tom branco (maior do que 127), onde esses pontos serão o background da imagem e outro aglomerado de pontos será nos tons mais escuros, representados pela tinta da escritura. Dito isso, um bom algoritmo de aproximação seria o de pegar o elemento que representa o 30^{o} percentil da imagem. Esperamos que tenhamos mais uma aglomeração de tons brancos, então o valor de corte deve estar perto do lado esquerdo do histograma, mas com um pequeno espaço entre os tons mais escuros. O algoritmo implementado pode ser visualizado na figura 7 abaixo

```python
# Gets the cut_point of a image based on the 30 percentile of the histogram distribution
def my_method_cut_point(image: np.ndarray) -> int:
    histogram, bins = np.histogram(image.ravel(), 256, [0, 256])

    percentile = np.percentile(histogram, 30)

    cut_point = (np.abs(histogram - percentile)).argmin()

    return np.uint8(cut_point)
```

Figura 7 – Função automatica de ponto de corte.

O algoritmo da figura 7 pega o histograma a imagem passada, e vê o valor da distribuição que equivale ao 30^{o} percentil, após isso procuramos qual index do histograma ([0, 255]) se aproxima do valor do percentil recolhido, então retornamos esse index como o ponto de corte.

Observando o histograma do documento 3 na imagem 8 podemos notar que o ponto de corte ideal deve estar em torno de 100 à 150, digamos 125. Nosso algoritmo retorna o ponto de corte de 130, que é bem próximo do ideal e que gera a binarização da imagem 9.

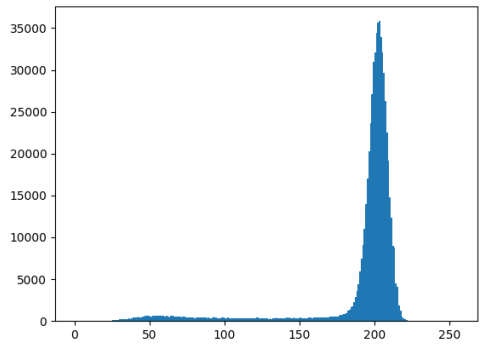

Figura 8 – Histograma do doc3.

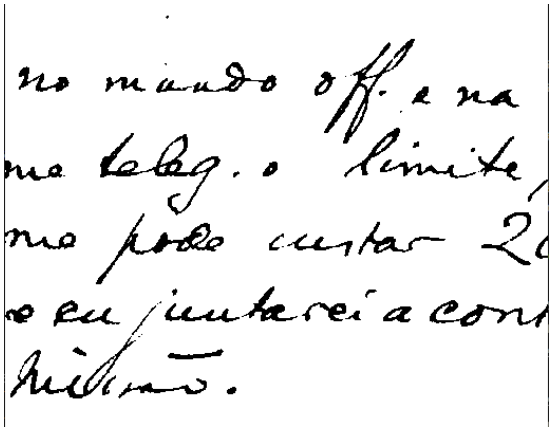

Figura 9 – Doc3 binarizado com o ponto de corte da função criada.

Código da questão 1:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from utils import *

# Images
DOC1_BMP_PATH = "./Imagens/Q1/doc1.bmp"
DOC2_BMP_PATH = "./Imagens/Q1/doc2.bmp"
DOC3_BMP_PATH = "./Imagens/Q1/doc3.bmp"


# Binarize image using @p cut_value and threshold
def binarize(image: np.ndarray, cut_value: int) -> np.ndarray:
    image[np.where(image <= cut_value)] = 0
    image[np.where(image > cut_value)] = 255

    return image


# Swaps the ideal cut point @p cut1 with @p doc2 and uses ideal cut @p cut2 in @p doc1
def swap_cut_point(doc1: np.ndarray, doc2: np.ndarray, cut1: int, cut2: int) -> None:
    show_image(binarize(doc1, cut2))
    show_image(binarize(doc2, cut1))


# Gets the cut_point of a image based on the 30 percentile of the histogram
    distribution
def my_method_cut_point(image: np.ndarray) -> int:
    histogram, bins = np.histogram(image.ravel(), 256, [0, 256])
```

```
    percentile = np.percentile(histogram, 30)

    cut_point = (np.abs(histogram - percentile)).argmin()

    return np.uint8(cut_point)


def main():
    # Load images
    images = read_images([DOC1_BMP_PATH, DOC2_BMP_PATH, DOC3_BMP_PATH])
    images_copy = images.copy()

    # Assert if image is grey scale
    assert_images_grey_scale(images)

    # Properly covert to gray scale to only work with one matrix
    for image in images:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Show histogram of all images
    show_images_hist(images)

    # Gets manual cut point from doc 1 and show images to user
    show_image(images[0])
    images[0] = binarize(images[0], 50)
    show_image(images[0])

    # Gets manual cut point from doc 2 and show images to user
    show_image(images[1])
    images[1] = binarize(images[1], 70)
    show_image(images[1])

    # Swaps cut points with doc 1 and doc 2 and show images to user
    swap_cut_point(images_copy[0], images_copy[1], 50, 70)

    # Apply my function on doc 3 and show images to user
    show_image(images[2])
    cut_point = my_method_cut_point(images[2])
    images[2] = binarize(images[2], cut_point)
    show_image(images[2])

    # Console log cut point
    print(cut_point)
    # Outputs 130


if __name__ == "__main__":
    main()
```

2ª Questão

Para esse tipo de imagem fica mais fácil de separar os tons, no entanto nosso algoritmo piora a performance. Aplicando os mesmos experimentos da questão 1, mas agora normalizando a imagem e aplicando a raiz quadrada conseguimos notar o seguinte padrão:

Imagem	Ponto de Corte
doc1	100
doc2	100

Tabela 2 – Pontos de corte manuais para os documentos 1 e 2 na questão 2

Os pontos de corte ideais retirados manualmente para os dois documentos ficaram novamente semelhantes. Se observarmos os histogramas dos documentos 1 e 2 ,respectivamente as imagens 10 e 11, podemos notar que ambos os histogramas tiveram um deslocamento para a direita, o que fez com que seus pontos de corte aumentarem.

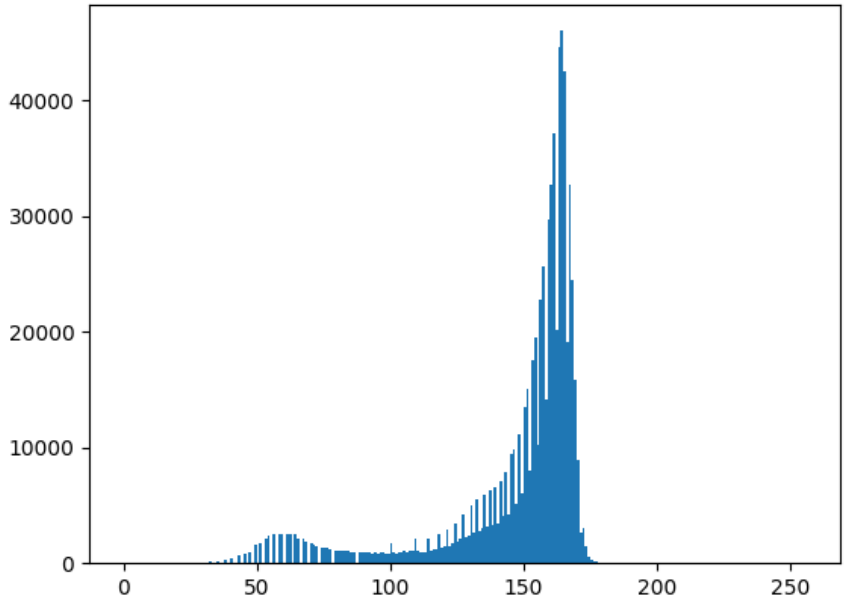

Figura 10 – Histograma para doc1.

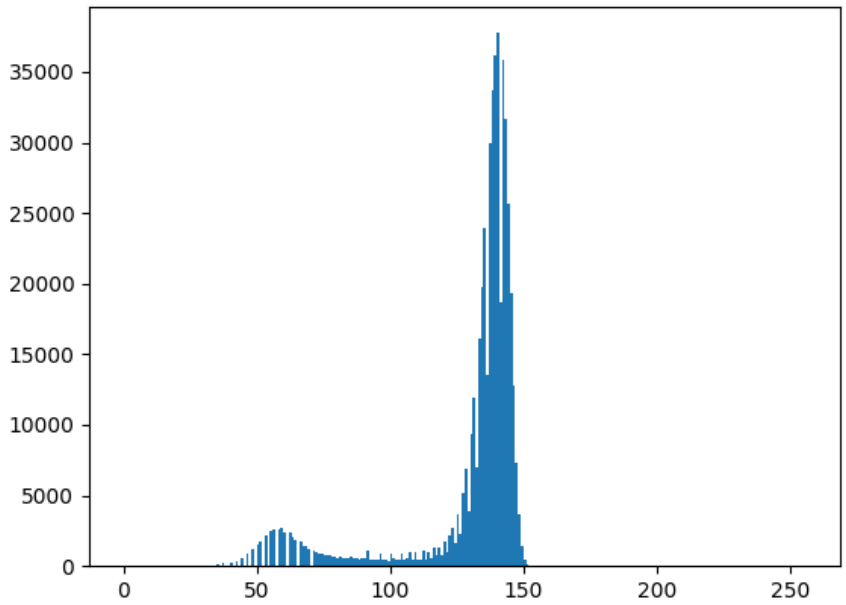

Figura 11 – Histograma para doc2.

Note que a binarização manual ainda gera bons resultados como mostrado nas figuras 12 e 13:

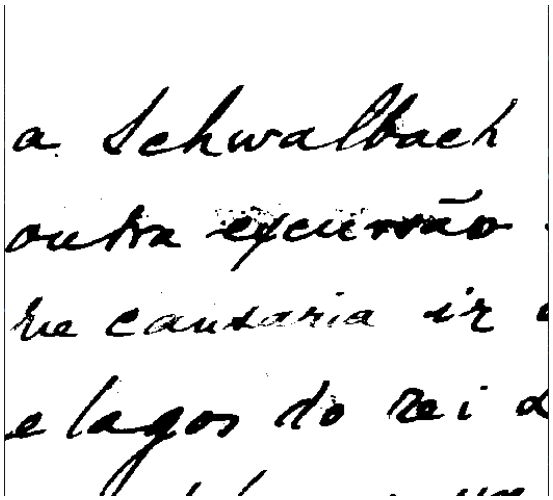

Figura 12 – Doc1 binarizado.

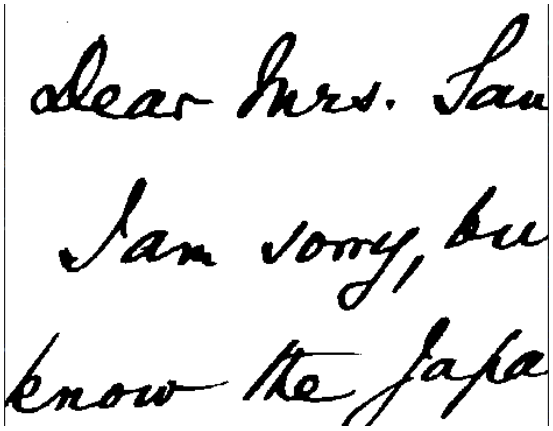

Figura 13 – Doc2 binarizado.

A troca dos pontos de corte também não influência na binarização, como mostrado nas figuras 5 e 6, pois ambos os histogramas continuam bem semelhantes.

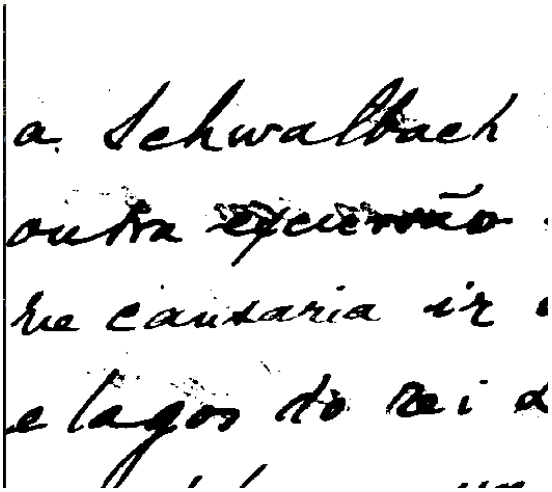

Figura 14 – Doc1 binarizado com o ponto de corte de doc2.

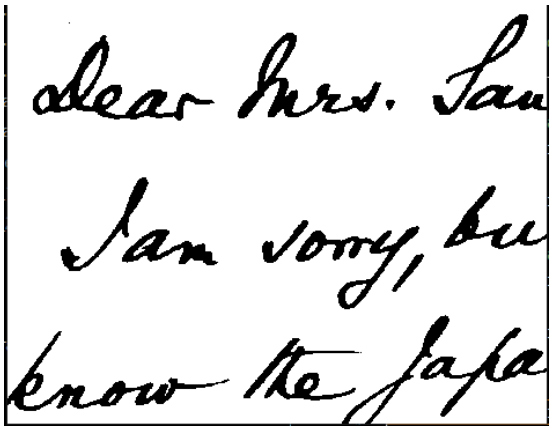

Figura 15 – Doc2 binarizado com o ponto de corte de doc1.

No entanto, quando aplicado o algoritmo desenvolvido na questão 1 nós obtemos os resultados das figuras 16 e 17:

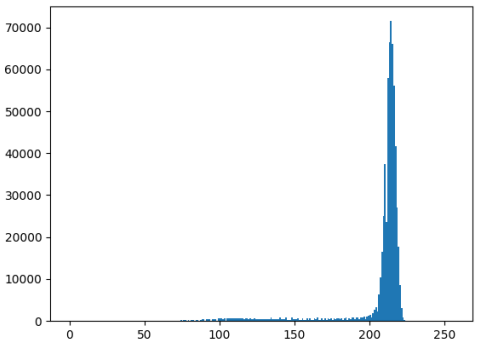

Figura 16 – Histograma do doc3.

Figura 17 – Doc3 binarizado com o ponto de corte da função criada.

O aconteceu nesse caso foi que: a operação de normalização fez com que gerassemos uma imagem de mais alto contraste, dessa nova imagem, o filtro de raiz quadrada vai fazer com a imagem normalizada tenha um realce nas diferenças de intensidades mais baixas, enquanto mantem as diferenças nas intensidades mais baixas. Em outras palavras, o filtro faz com que os tons mais escuros se espalhem mais pelo histograma diminuindo um pouco do ruido do *background* da imagem. Feita a desnormalização, podemos observar que agora o ponto de corte se aproxima mais do centro do histograma, mas como nosso algoritmo não tinha essa premissa, ele acaba gerando um resultado pior na binarização.

Código da questão 2:

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt
from q1 import swap_cut_point, binarize, my_method_cut_point
from utils import *

# Images
DOC1_BMP_PATH = "./Imagens/Q2/doc1.bmp"
DOC2_BMP_PATH = "./Imagens/Q2/doc2.bmp"
DOC3_BMP_PATH = "./Imagens/Q2/doc3.bmp"


# Normalize image by it's histogram
def normalize(image: np.ndarray) -> tuple[np.ndarray, int, int]:
    max_c = np.max(image.ravel())
    min_c = np.min(image.ravel())

    assert not np.isclose(max_c, min_c)

    return ((image - min_c) / (max_c - min_c)).astype("float64"), max_c, min_c


# Desnormalize image based on it previous max and min values
def desnormalize(image: np.ndarray, max_c: int, min_c: int) -> np.ndarray:
    return np.round(((image * (max_c - min_c)) + min_c)).astype("uint8")


def main():
    # Load images
    images = read_images([DOC1_BMP_PATH, DOC2_BMP_PATH, DOC3_BMP_PATH])

    # Assert if image is grey scale
    assert_images_grey_scale(images)

    # Properly covert to gray scale to only work with one matrix
    for image in images:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Get list of max and min values in each histogram
    maxs = []
    mins = []

    for i, image in enumerate(images):
        images[i], max_c, min_c = normalize(image)
        maxs.append(max_c)
        mins.append(min_c)
```

```
    # Apply sqrt filter
    images = list(map(np.sqrt, images))

    # Properly covert each image back to original domain
    for i, image in enumerate(images):
        images[i] = desnormalize(image, maxs[i], mins[i])

    # Show histogram of all images
    show_images_hist(images)

    # Gets manual cut point from doc 1 and show images to user
    show_image(images[0])
    images[0] = binarize(images[0], 100)
    show_image(images[0])

    # Gets manual cut point from doc 2 and show images to user
    show_image(images[1])
    images[1] = binarize(images[1], 100)
    show_image(images[1])

    # Swaps cut points with doc 1 and doc 2 and show images to user
    swap_cut_point(images[0], images[1], 100, 100)

    show_image(images[2])
    cut_point = my_method_cut_point(images[2])
    images[2] = binarize(images[2], cut_point)
    show_image(images[2])

    # Show cut point to user
    print(cut_point)
    # Outputs 0

if __name__ == "__main__":
    main()
```

3ª Questão

Já que não podemos usar *dithering*, uma forma de tentar reduzir o número de cores é através da redução do tamanho da paleta disponivel. Como queremos que as cores sejam agrupadas em tons semelhantes, o que fazemos é converter o modelo de cor para HSV antes de reduzir a palheta. Nesse modelo, fica mais fácil agruparmos as cores de tonalidades semelhantes pois elas ficam em intervalos de tonalidade proximas. Dessa forma, dividimos os 180 valores do canal de tonalidade (*hue*) em 8 intervalos, então, passamos em cada pixel da imagem e atribuimos aquele pixel ao valor intermediario do intervalo que a cor original pertence. Aplicando essa logica conseguimos como resultado as imagens 18, 19, 20 e 21:

Figura 18 – Imagem das araras com cores reduzidas.

Figura 19 – Imagem da Formula 1 com cores reduzidas.

Figura 20 – Imagem da planta cores reduzidas.

Figura 21 – Imagem do surf com cores reduzidas.

Para conseguir o fator de redução da técnica, basta contarmos quantas cores tinhamos antes da redução da paleta de cores e fazermos a razão com quantas cores temos após a redução. Fazendo isso, obtermos o resultado da 22. Como podemos observar, todos os valores de redução foram, aproximadamente, entre 0.5 e 0.33.

```
Reduction proporsion: 0.551
Reduction proporsion: 0.583
Reduction proporsion: 0.305
Reduction proporsion: 0.457
```

Figura 22 – Sumário da redução de cores para cada imagem.

Código da questão 3:

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt
from utils import *

# Images
ARARAS_BMP_PATH = "./Imagens/Q3/araras.bmp"
F1_BMP_PATH = "./Imagens/Q3/F1.bmp"
GREEN_WATER_BMP_PATH = "./Imagens/Q3/green-water.bmp"
SURF_51_BMP_PATH = "./Imagens/Q3/surf_51.bmp"


# Gets the total number of colors in one image
def get_colors(image: np.ndarray) -> dict:
    b, g, r = image[:, :, 0], image[:, :, 1], image[:, :, 2]
    colors_map = {}

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if ((b[i][j], g[i][j], r[i][j])) not in colors_map:
                colors_map[(b[i][j], g[i][j], r[i][j])] = 1
            else:
                colors_map[(b[i][j], g[i][j], r[i][j])] += 1

    return colors_map


# Reduces the color pallete of the image based where the color hits on the hsv
    reduced pallete
def reduce_color_range(image: np.ndarray) -> np.ndarray:
    image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    pallete_size = 8

    pallete = np.linspace(0, 179, endpoint=True, num=pallete_size).astype("uint8")

    for i in range(len(image)):
        for j in range(len(image[i])):
            for p in range(0, len(pallete) - 1, 1):
                if pallete[p] <= image[i][j][0] and image[i][j][0] <= pallete[p + 1]:
                    image[i][j][0] = np.uint8(
                        (int(pallete[p]) + int(pallete[p + 1])) // 2
                    )
                    break

    image = cv2.cvtColor(image, cv2.COLOR_HSV2BGR)
    return image
```

```python
def main():
    # Read images
    images = read_images(
        [ARARAS_BMP_PATH, F1_BMP_PATH, GREEN_WATER_BMP_PATH, SURF_51_BMP_PATH]
    )

    # For each image
    for i, image in enumerate(images):
        # Gets the original number of colors
        colors_size_before = len(get_colors(image))

        # Reduce the range of colors using hsv logic
        images[i] = reduce_color_range(image)

        # Gets the new number of colors
        colors_size_after = len(get_colors(images[i]))

        # Shows the proporsion that reduced
        print(f"Reduction proporsion: {(colors_size_after / colors_size_before):.3f}")
        # Outputs 0.551, 0.583, 0.305, 0.457

    show_images(images)


if __name__ == "__main__":
    main()
```

4ª Questão

Letra a). O resultado dessa operação é também uma imagem embaçada. O que aconteção são duas correlações, uma em relação as linhas da imagem e outra em relação as colunas. Podemos observar a imagem com o filtro box e a imagem resultante das duas correlações nas imagens 23 e 24 respectivamente.

Figura 23 – Imagem com filtro box aplicado.

Figura 24 – Imagem com filtro das mascaras h1 depois h2.

Letra b). Sim, as duas imagens são semelhantes.

Letra c). O resultado da convolução discreta de h1 e h2 pode ser observado na figura 25

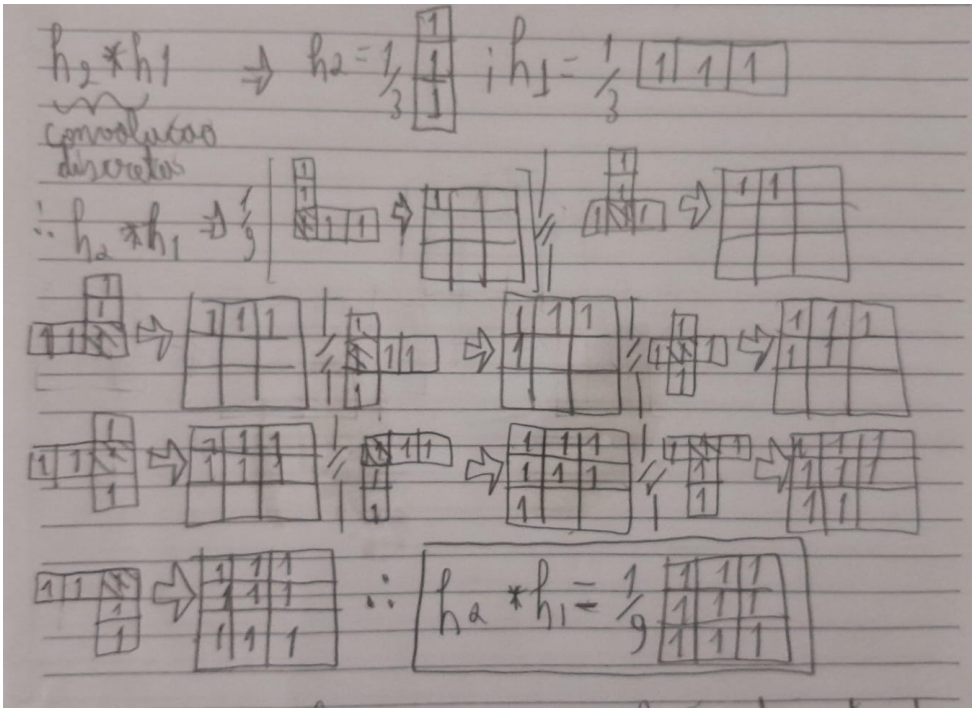

Figura 25 – Resultado da convolução discreta de h1 com h2

Como podemos observar, a convolução discreta de h2 com h1 gera o mesmo filtro box usado inicialmente na imagem. Podemos nos atentar ao fato que as operações de correlação e convolução (para os filtros lineares) são comutativas e associativas, portanto, mudar a ordem as operações leva ao mesmo resultado final.

Código da questão 4:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from utils import *
import scipy.signal as scs


# Image
CAMERAMEN_BMP_PATH = "./Imagens/Q4/cameraman.bmp"

# Filter kernels
BOX_FILTER_KERNEL = np.ones((3, 3)) * (1 / 9)
H1_FILTER_KERNEL = np.ones((1, 3)) * (1 / 3)
H2_FILTER_KERNEL = np.ones((3, 1)) * (1 / 3)


# Applies Correlation between a image and a kernel
def correlate2d(img: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    return scs.correlate2d(img, kernel, mode="same", boundary="symm")


# Applies Discrete convolution between a image and a kernel
def convolve2d(img: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    return scs.convolve2d(img, kernel, mode="full")


# Clips content of image to be inside rbg-8 range
def clip_gray_image(img: np.ndarray) -> np.ndarray:
    return np.clip(img, 0, 255).astype("uint8")


# Applies a Discrete convolution between the H1 filter kernel and the H2 filter kernel
def convolve_h1_h2() -> np.ndarray:
    return convolve2d(H1_FILTER_KERNEL, H2_FILTER_KERNEL)


def main():
    # Read image
    image = read_images([CAMERAMEN_BMP_PATH])[0]

    # Asserts grey scale
    assert_images_grey_scale([image])

    # Convert just to work with one matrix
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
    # Shows image
    show_image(image)

    # Apply box filter correlation
    image_box_filter = clip_gray_image(correlate2d(image, BOX_FILTER_KERNEL))
    show_image(image_box_filter)

    # Apply two correlations between the image with filter h1 then h2
    image_h1_then_h2 = clip_gray_image(
        correlate2d(correlate2d(image, H1_FILTER_KERNEL), H2_FILTER_KERNEL)
    )

    show_image(image_h1_then_h2)

    # Apply one discrete convolution and one correlation with the image
    image_h1_h2_combined = clip_gray_image(correlate2d(image, convolve_h1_h2()))
    show_image(image_h1_h2_combined)

    # Shows mean error between the two images
    print(np.abs(image_h1_h2_combined - image_box_filter).mean())


if __name__ == "__main__":
    main()
```

5ª Questão

Um modo de achar os contornos da imagem é aplicando os principios de detecção de borda. Inicialmente aplicamos um filtro Gaussiano para tentar eliminar os ruidos de alta frequência da imagem, obtendo como resultado a figura 26. Dito isso, aplicamos o algoritmo de binarização de dois picos obtendo a 27. Então, procuraremos os vetores de gradiente da imagem de tal forma e a maior variação indicará os pontos de maior troca de tons, portanto, as bordas esperadas do objeto. Para pegar essas bordas, aplicamos os filtros de Sobel o que nós leva à figura 28

Figura 26 – Imagem após aplicado o filtro Gaussiano

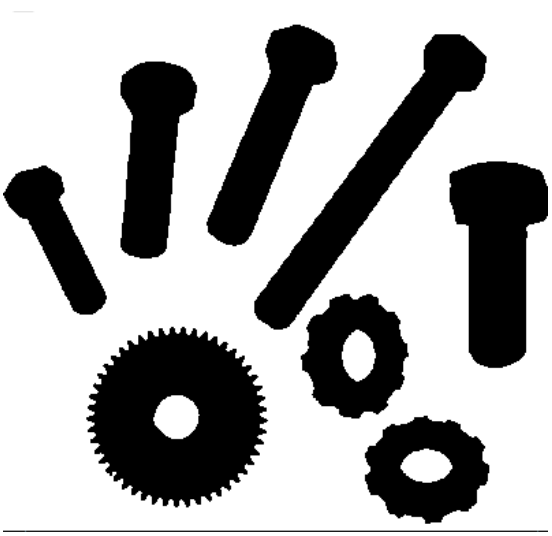

Figura 27 – Imagem binarizada.

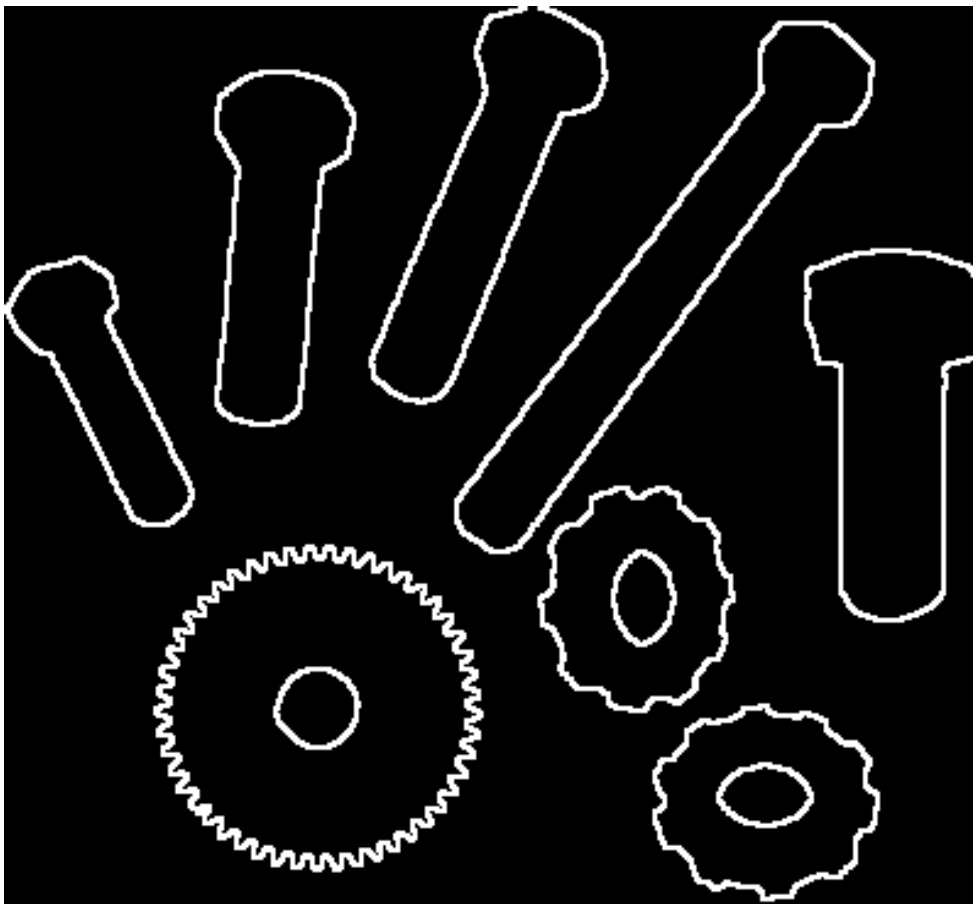

Figura 28 – Imagem com as bordas externas destacadas

Código da questão 5:

```
import numpy as np
```

```python
import cv2
import matplotlib.pyplot as plt
from utils import *
from scipy.ndimage import median_filter
from q7 import equalize
from q4 import convolve2d, correlate2d, clip_gray_image
from q8 import erosion, dilate
from q9 import complement_image
from q2 import normalize
from q1 import binarize, two_peaks


# Image
SCENE_BMP_PATH = "./Imagens/Q5/cena.bmp"

# Kernels
LAPLACIAN_FILTER_KERNEL = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
GAUSSIAN_FILTER_KERNEL = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]]) * (1 / 16)
SOBEL_FILTER_X_KERNEL = np.array(
    [
        [-1, 0, 1],
        [-2, 0, 2],
        [-1, 0, 1],
    ]
)
SOBEL_FILTER_Y_KERNEL = np.array(
    [
        [-1, -2, -1],
        [0, 0, 0],
        [1, 2, 1],
    ]
)

# Circles
C1_BMP_PATH = "./Imagens/Q5/C1.bmp"
C2_BMP_PATH = "./Imagens/Q5/C2.bmp"
C3_BMP_PATH = "./Imagens/Q5/C3.bmp"


# Some edge detection kernel
def edge_dection_kernel() -> np.ndarray:
    return convolve2d(LAPLACIAN_FILTER_KERNEL, GAUSSIAN_FILTER_KERNEL)


# Closes the image by dilate then erode
def close_image(image: np.ndarray) -> np.ndarray:
    circle_structure_element = (
        np.array(
```

```
            [
                [0, 1, 0],
                [1, 1, 1],
                [0, 1, 0],
            ],
            dtype="uint8",
        )
    ) * 255

    dilated_image = dilate(image, circle_structure_element)
    closed_image = erosion(image, circle_structure_element)

    return closed_image.astype("uint8")


# Open some image by eroding then dilating
def open_image(image: np.ndarray) -> np.ndarray:
    circle_structure_element = (
        np.array(
            [
                [1, 1],
                [1, 1],
            ],
            dtype="uint8",
        )
    ) * 255

    erode_image = erosion(image, circle_structure_element)
    opened_image = dilate(erode_image, circle_structure_element)

    return opened_image.astype("uint8")


def main():
    # Read image
    scene_image = read_images([SCENE_BMP_PATH])[0]

    # Assert grey scale
    assert_images_grey_scale(scene_image)

    # Convert to work with only one image
    scene_image = cv2.cvtColor(scene_image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian filter
    scene_image = clip_gray_image(correlate2d(scene_image, GAUSSIAN_FILTER_KERNEL))

    show_image(scene_image)
```

```
    # Binarize
    binarized_edge_image = clip_gray_image(
        binarize(scene_image, two_peaks(scene_image))
    )

    show_image(binarized_edge_image)

    # Apply Edge detection with Sobel
    sobel_x = correlate2d(binarized_edge_image, SOBEL_FILTER_X_KERNEL)

    sobel_y = correlate2d(binarized_edge_image, SOBEL_FILTER_Y_KERNEL)

    sobel = clip_gray_image(np.abs(sobel_x) + np.abs(sobel_y))

    show_image(sobel)


if __name__ == "__main__":
    main()
```

6ª Questão

Como apenas queremos detectar se temos um M&M azul na imagem e não necessitamos contar, uma forma de resolver o problema é usando uma lógica semelhante à da questão 3. Inicialmente convertemos a imagem para HSV e então verificamos para cada pixel se ele está presente dentro do intervalo de tonalidade do azul no espectro de cores. Se conseguirmos detectar que algum desse pixel respeita essa condição, dizemos que o M&M está presente.

Caso fossemos saber se existia um M&M vermelho E laranja a solução não seria tão fácil. Ambas as cores, vermelho e laranja, tem as suas tonalidades muito próximas, o efeito da saturação e o valor também faz com que os tons de vermelho e laranja sejam fácilmente confundidos caso exista alguma sombra na imagem ocasionada por um M&M sobreposto ao outro, o que torna faz com que esse problema requira mais processamento na imagem para corrigir esses casos.

Código da questão 6:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from utils import *

# Image
MM_BMP_PATH = "./Imagens/Q6/MM.bmp"


# Detect if image has hue value in the blue range
def detect_blue(image: np.ndarray) -> bool:
```

```
    blue_lower = np.ndarray([100, 100, 100])
    blue_upper = np.ndarray([130, 255, 255])

    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    for i in range(len(image)):
        for j in range(len(image[i])):
            if np.all(image[i][j] >= blue_lower) and np.all(image[i][j] <= blue_upper):
                return True

    return False


def main():
    # Read image
    mm_image = read_images([MM_BMP_PATH])[0]

    # Prints that it does have blue M&M's if we can detect blue and false otherwise
    if detect_blue(mm_image):
        print(f"It does have blue M&M's")
    else:
        print(f"It doesn't have blue M&M's")

    show_image(mm_image)


if __name__ == "__main__":
    main()
```

7ª Questão

A questão 7 envolve limpar falhas e ruidos o máximo possível para binarizar a imagem. Para fazer isso, fazemos primeiro uma analise de seu histograma como mostrado na figura 29. Como podemos ver, apenas binarizar a imagem agora acarretaria a alguns problemas, pois temos muita informação em regiões de tom cinza intermediario na imagem. Um primeiro passo para trabalhar com esse problema é equalizar a imagem. Equalizando nós separamos melhor a intensidade dos pixels e obtemos o histograma da imagem 30. Desse histograma nós temos a imagem 31 e o que podemos fazer é aplicar uma normalização e um filtro de potência quadrada. Com esse filtro nós conseguimos aumentar a intensidade da diferença dos tons da imagem obtendo o histograma da imagem **??**. Agora podemos aplicar um filtro Gaussiano para manter apenas as baixas frequências e removermos partes dos tons em cinza indesejados, assim obtendo a imagem 34. Por fim, podemos aplicar apenas um algoritmo de binarização como o de dois picos visto em sala, obtendo a imagem final 35.

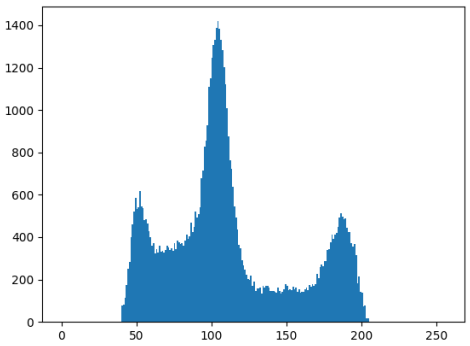

Figura 29 – Histograma da imagem.

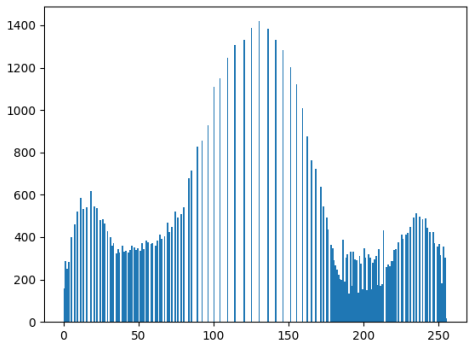

Figura 30 – Histograma equalizado da imagem.

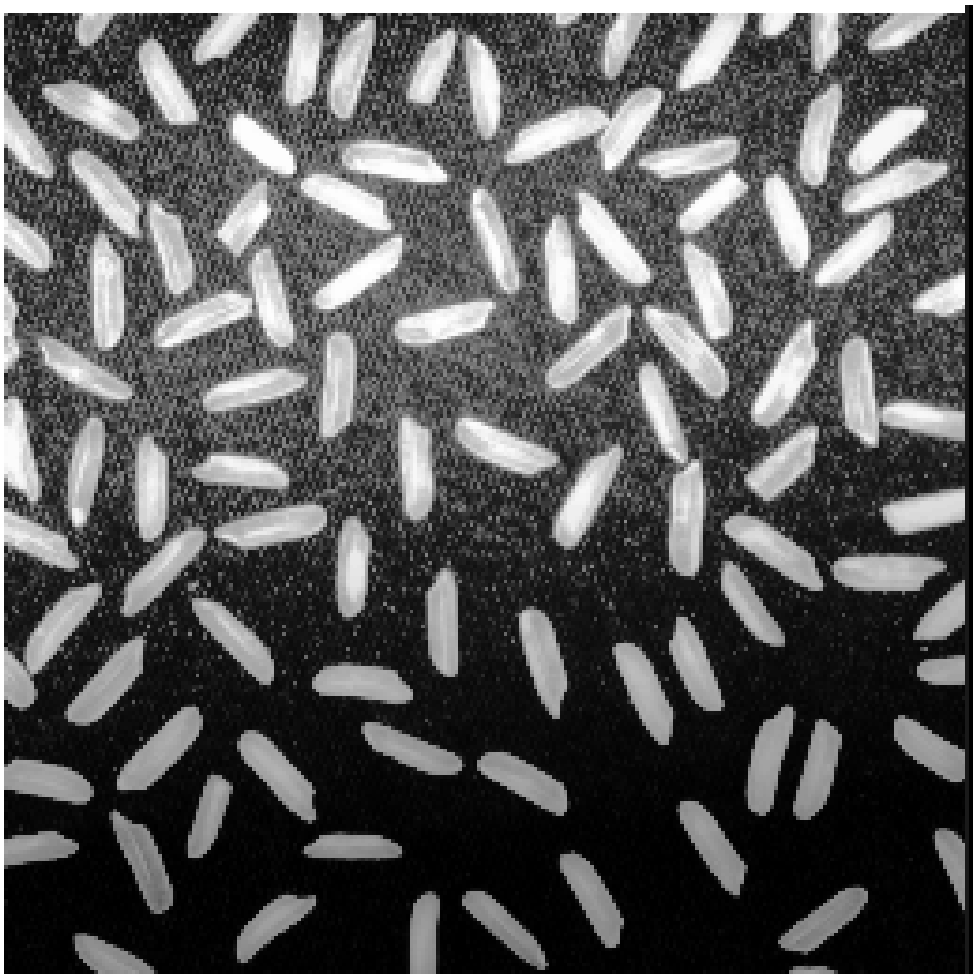

Figura 31 – Imagem dos arroz equalizada

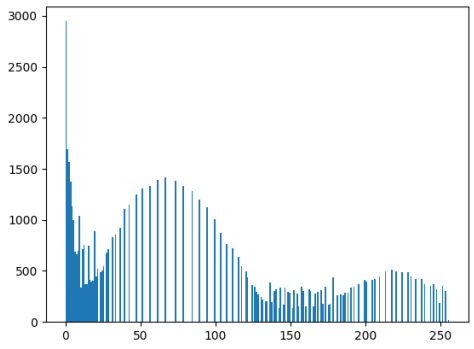

Figura 32 – Histograma normalizado e aplicado o filtro da potência ao quadrado.

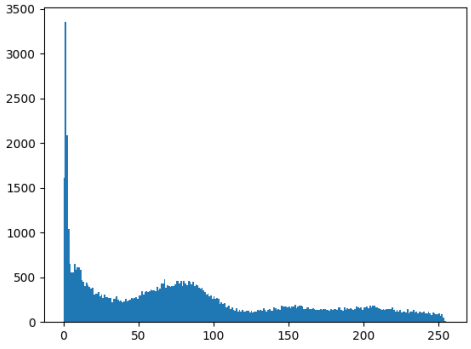

Figura 33 – Histograma após todas as operações e filtrado com um filtro Gaussiano.

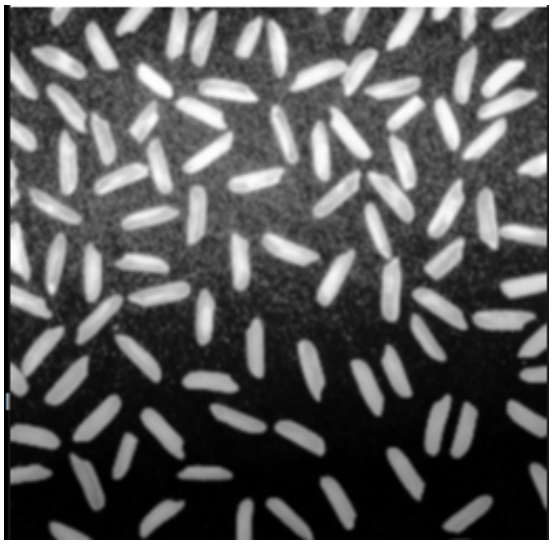

Figura 34 – Imagem após a aplicação do filtro Gaussiano

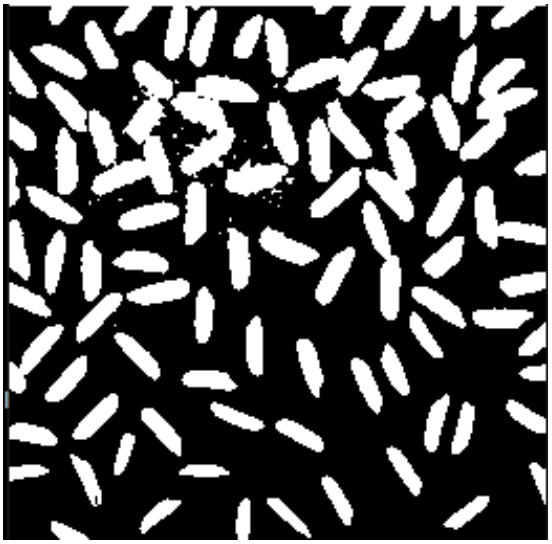

Figura 35 – Imagem após binarização

Código da questão 7:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from utils import *
from scipy.ndimage import median_filter
from q1 import two_peaks, binarize
from q2 import normalize, desnormalize
from q4 import convolve2d, correlate2d, clip_gray_image

# Images
```

```python
RICE_BMP_PATH = "./Imagens/Q7/rice.bmp"

# Kernels
GAUSSIAN_FILTER_KERNEL = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]]) * (1 / 16)


# Equalize the histogram of the image by dividing the cdf
def equalize(image: np.ndarray) -> np.ndarray:
    bin_counts, bin_edges = np.histogram(image.ravel(), 256, [0, 256])
    bin_counts = np.int32(bin_counts)

    cdf = bin_counts.cumsum()

    cdf_normalized = ((cdf - cdf.min()) * 255) / (cdf.max() - cdf.min())

    equalized_image = cdf_normalized[image]

    return equalized_image.astype("uint8")


def main():
    # Read image
    rice_image = read_images([RICE_BMP_PATH])[0]

    # Assert grey scale
    assert_images_grey_scale(rice_image)

    # Convert to work with only one matrix
    rice_image = cv2.cvtColor(rice_image, cv2.COLOR_RGB2GRAY)

    # Show rice histogram
    show_images_hist([rice_image])

    # Equalize histogram
    rice_image = equalize(rice_image)

    # Show histogram again
    show_images_hist([rice_image])

    # Normalize
    rice_image, r_max, r_min = normalize(rice_image)

    # Apply power of 2 to get a image with more separated intensities
    rice_image = rice_image**2

    # Desnormalize image
    rice_image = desnormalize(rice_image, r_max, r_min)
```

```
    # Show histogram again
    show_images_hist([rice_image])

    show_image(rice_image)

    # Uses Gaussian filter to reduce noise
    rice_image = clip_gray_image(correlate2d(rice_image, GAUSSIAN_FILTER_KERNEL))

    # Show histogram again
    show_images_hist([rice_image])

    show_image(rice_image)

    # Biniraze image
    show_image(binarize(rice_image, two_peaks(rice_image)))


if __name__ == "__main__":
    main()
```

8ª Questão

Podemos resolver esse problema de varias maneiras. Apresentarei duas delas:

Primeira: Esse tipo de ruído na imagem é bem semelhante à um ruído *salt and pepper*, uma das formas de se limpar ele é através da aplicação de um filtro de mediana. Aplicando esse filtro com uma janela 5 por 5 obtemos as images 36 e 37 abaixo que já são um bom resultado.

Figura 36 – Imagem do veado filtrada por um filtro de mediana de 5 por 5

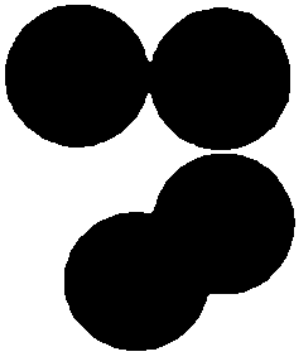

Figura 37 – Imagem dos circulos filtrada por um filtro de mediana de 5 por 5

Segunda: Podemos usar morfologia matemática para limpar a imagem através de um operações de abertura e fechamento. Como as imagens já estão binarizadas, apenas precisamos focar nos objetos e operações de interesse de cada caso. Começarei abordando a imagem do circulo. Nessa imagem, podemos fazer a limpeza do ruído apenas complementando os pixels para trabalharmos com o background ou com os circulos grandes como objetos de interesse. Inicialmente complementamos a imagem para trabalharmos com os circulos maiores como objeto de interesse, assim como mostrado na figura 38. Após isso, aplicamos uma erosão para reduzirmos os ruidos do background e obtemos imagem 39. Então trabalharemos agora com os ruidos internos que restaram. Complementaremos a imagem, pois o interesse agora são os pontos brancos internos do circulo maior e aplicamos duas erosões representadas pelas figurlas 40 e 41

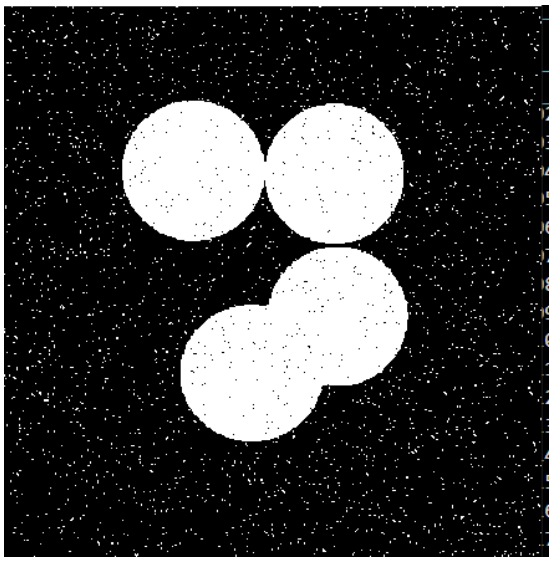

Figura 38 – Imagem do circulo complementada.

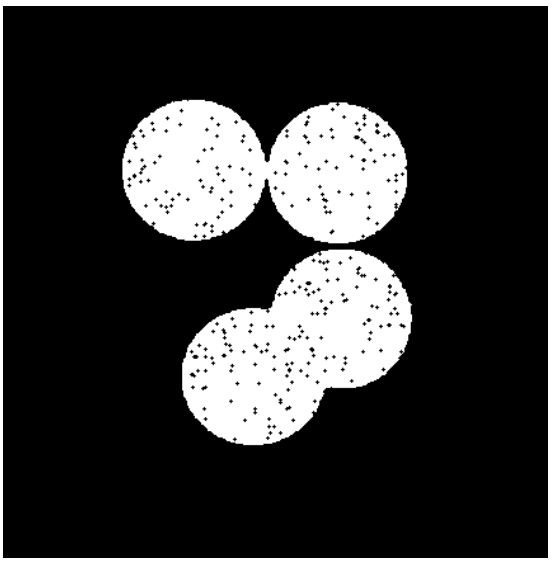

Figura 39 – Imagem do circulo complementada após uma erosão.

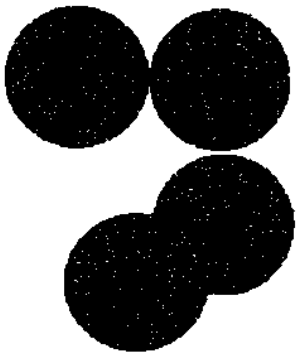

Figura 40 – Imagem do circulo complementada novamente com uma erosão.

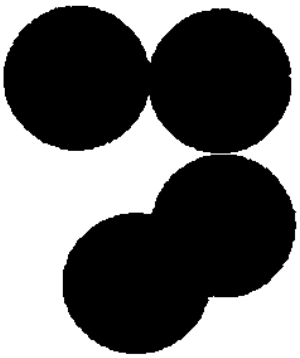

Figura 41 – Imagem do circulo com a erosão final.

A logíca para a figura do veado é a mesma, no entanto, não precisamos fazer complementos na imagem, como o objeto de interesse é bem definido pelo veado, podemos fazer operações de abertura e fechamento até obtermos o resultado desejado. Inicialmente operamos uma operação de abertura (erosão seguida de dilatação com o mesmo elemento estruturante) com um elemento estruturante de diagonal para obtermos a imagem da figura 42 e então aplicamos uma operação de fechamento (dilatação seguida de erosão com um mesmo elemento estruturante), mas agora com um elemento estruturante circular para obter o resultado da 43. A ideial é de abrir os ruídos internos do veado e depois tentar fechar aos ruídos externos. Por fim, aplicamos apenas uma erosão com o mesmo circulo como elemento estruturante para obter a image 44 mais refinada.

Figura 42 – Imagem do veado após uma abertura.

Figura 43 – Imagem do veado após fechamento.

Figura 44 – Imagem do veado após erosão final.

Código da questão 8:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from scipy.ndimage import median_filter
from utils import *

# Images
BINARY_BMP_PATH = "./Imagens/Q8/Binary_Noise.bmp"
CIRCLES_BMP_PATH = "./Imagens/Q8/Circles_Noise.bmp"


# Just complements some binary image
def complement_image(image: np.ndarray) -> np.ndarray:
    return np.abs(255 - image).astype("uint8")


# Applies a dilatation on a 3d image using some structure element
def dilate(image: np.ndarray, structure_element: np.ndarray) -> np.ndarray:
    width, height = structure_element.shape

    half_w = width // 2
    half_h = height // 2

    final_image = np.zeros(image.shape)
    for i in range(len(image)):
        for j in range(len(image[i])):
```

```
            image_height = len(image)
            image_width = len(image[0])

            is_active = 0

            for h in range(height):
                for w in range(width):
                    height_bound = i + h - half_h
                    width_bound = j + w - half_w

                    if (
                        height_bound < 0
                        or height_bound >= image_height
                        or width_bound < 0
                        or width_bound >= image_width
                    ):
                        continue

                    if (
                        structure_element[h][w] == image[height_bound][width_bound][0]
                        and structure_element[h][w] == 255
                    ):
                        is_active = 255
                        break

                if is_active:
                    break

            final_image[i][j] = is_active

    return final_image


# Applies a erosion on a 3d matrix using some structure element
def erosion(image: np.ndarray, structure_element: np.ndarray) -> np.ndarray:
    width, height = structure_element.shape

    half_w = width // 2
    half_h = height // 2

    final_image = np.zeros(image.shape)
    for i in range(len(image)):
        for j in range(len(image[i])):
            image_height = len(image)
            image_width = len(image[0])

            is_active = 255
```

```
            for h in range(height):
                for w in range(width):
                    height_bound = i + h - half_h
                    width_bound = j + w - half_w

                    if (
                        height_bound < 0
                        or height_bound >= image_height
                        or width_bound < 0
                        or width_bound >= image_width
                    ):
                        continue

                    if (
                        structure_element[h][w] != image[height_bound][width_bound][0]
                        and structure_element[h][w] != 0
                    ):
                        is_active = 0
                        break

                if not is_active:
                    break
            final_image[i][j] = is_active

    return final_image


# Tries to remove noise to the circle image
def remove_circle_noise(circle_image: np.ndarray) -> None:
    # Applies a little circle as a structure element
    circle_structure_element = (
        np.array(
            [
                [0, 1, 0],
                [1, 1, 1],
                [0, 1, 0],
            ]
        )
    ) * 255

    # Complement image, the circle aren't the object of interest right now
    circle_image = complement_image(circle_image)

    show_image(circle_image)

    # Applies erosion to remove the little circles outside the circle
    first_circle_erosion = erosion(circle_image, circle_structure_element)
```

```
    show_image(first_circle_erosion)

    # Complement the image, now we gonna remove the noise inside the big circle
    second_circle_erosion = erosion(
        complement_image(first_circle_erosion), circle_structure_element
    )

    show_image(second_circle_erosion)

    # Applies another erosion the remove indead what was dilated before
    thirdy_circle_erosion = erosion(second_circle_erosion, circle_structure_element)

    show_image(thirdy_circle_erosion)


# Tries to remove noise to the deer image
def remove_binary_noise(binary_image: np.ndarray) -> None:
    # Applies a diagonal element to flow the direction of the grass
    binary_structure_element = (
        np.array(
            [
                [1, 0],
                [0, 1],
            ]
        )
    ) * 255

    # Opens the image to try to fill the gaps inside the deer
    binary_first_erosion = erosion(binary_image, binary_structure_element)
    binary_opened = dilate(binary_first_erosion, binary_structure_element)

    show_image(binary_opened)

    # Now we gonna work with some circles to remove the noise from outside the deer
    binary_second_structure_element = (
        np.array(
            [
                [0, 1, 0],
                [1, 1, 1],
                [0, 1, 0],
            ]
        )
    ) * 255

    # Closes the image from the gaps on the background
    binary_second_dilated = dilate(binary_opened, binary_second_structure_element)
    binary_closed = erosion(binary_second_dilated, binary_second_structure_element)
```

```
    show_image(binary_closed)

    # Applies one more erosion to remove remaing noises
    show_image(erosion(binary_closed, binary_second_structure_element))


# Using median filter
def main2():
    images = read_images([BINARY_BMP_PATH, CIRCLES_BMP_PATH])
    assert_images_grey_scale(images)

    binary_image, circle_image = median_filter(images[0], size=5), median_filter(
        images[1], size=5
    )

    show_image(binary_image)
    show_image(circle_image)


# Using morphology
def main():
    # Read images
    images = read_images([BINARY_BMP_PATH, CIRCLES_BMP_PATH])
    # Assert grey scale
    assert_images_grey_scale(images)

    # Get each individual image to work different
    binary_image, circle_image = images[0], images[1]

    # Tries to remove noise in each image
    remove_binary_noise(binary_image)
    remove_circle_noise(circle_image)


if __name__ == "__main__":
    main()
```

9ª Questão

Novamente podemos usar a técnica de morfologia matematica para resolver a questão. Como as imagens ja estão binarizadas, não necessitamos repetir esse processo. O que faremos é recolher uma amostra do que é uma matriz que representa a letra 'A' maiuscula, como mostrado na image 45. Como o texto é o objeto de interesse, complementamos o objeto estruturante e a imagem do texto e então aplicamos uma erosão sobre a imagem. Se após aplicada a erosão tivermos algum pixel branco significa que naquele ponto tivemos um match perfeito com o elemento estruturante e, portanto, o texto contém a letra 'A' maiuscula. O resultado dos dois processamentos pode ser observado na figura 46.

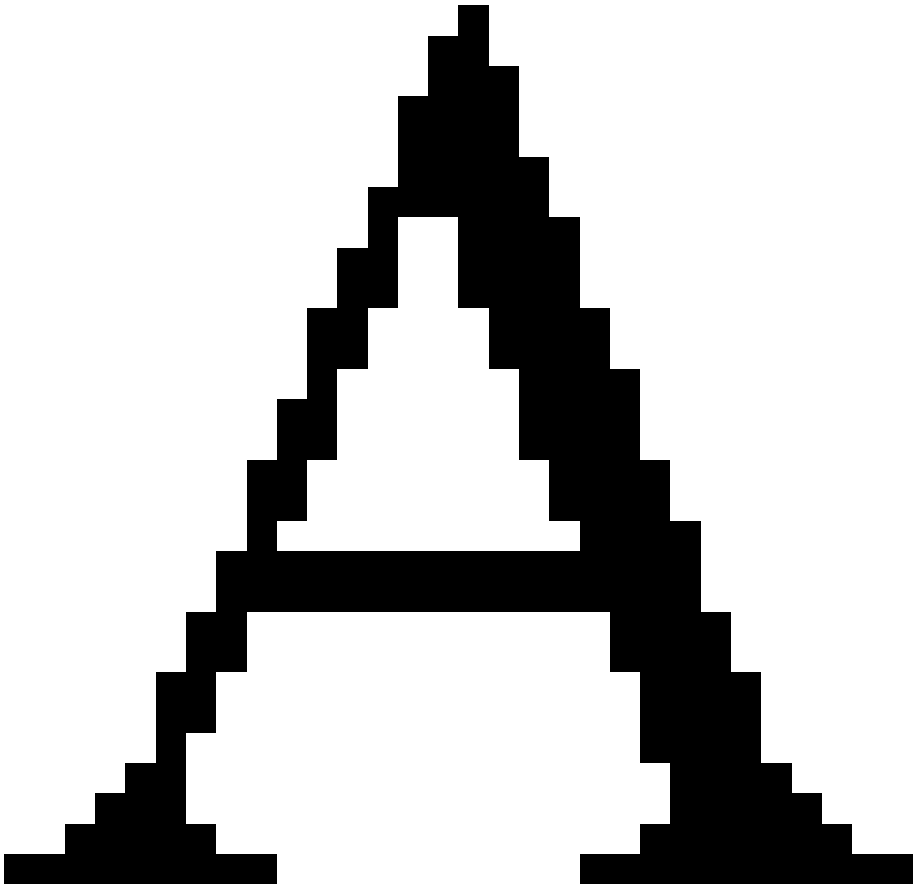

Figura 45 – Matriz que representa o A maiusculo

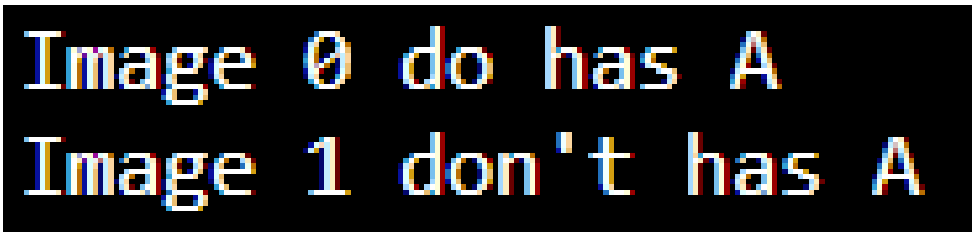

Figura 46 – Resultado da erosão nas duas imagens

Código da questão 9:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from utils import *
from q8 import erosion, complement_image

# Images
BOOK1_BMP_PATH = "./Imagens/Q9/Book_1.bmp"
```

```
BOOK2_BMP_PATH = "./Imagens/Q9/Book_2.bmp"

# Structure Element
A_BMP_PATH = "./Imagens/Q9/A.bmp"


def main():
    # Read images
    images = read_images([BOOK1_BMP_PATH, BOOK2_BMP_PATH])

    # Read structure element
    structure_element = read_images([A_BMP_PATH])[0]

    # Assert grey scale
    assert_images_grey_scale([structure_element])
    assert_images_grey_scale(images)

    # Convert to grey scale to only work with one matrix
    structure_element = cv2.cvtColor(structure_element, cv2.COLOR_BGR2GRAY)

    # Complement structure element
    structure_element = complement_image(structure_element)

    for i, image in enumerate(images):
        # Convert to grey scale to only work with one matrix
        images[i] = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        # Complement image because the letters are the object of interest
        images[i] = complement_image(image)

        # Apply erosion
        erode_image = erosion(images[i], structure_element)

        # Checks if any pixel is a match
        def has_A(image: np.ndarray):
            return np.any(image == 255)

        # Print if it has A or not
        print(f"Image {i} " + ("do has A" if has_A(erode_image) else "don't has A"))


if __name__ == "__main__":
    main()
```

10ª Questão

Para resolver essa questão necessitamos dividir nossa logica em alguns passos. Inicialmente nos vamos trabalhar apenas com a região de interesse da imagem, que será a região onde nós vimos

que está localizada a linha do mar como representada na figura 47. Após coletada a região converteremos a imagem para o modelo CIE L*A*b* e pegamos apenas a matriz do canal b*. A ideia por trás dessa conversão é trabalhar apenas com a variação de cor do amarelo (cor esperada da areia) para a cor azul (cor esperada para o mar). Depois dessa conversão podemos aplicar o algoritmo de binarização de dois picos visto em aula e implementado no projeto. Com a binarização conseguimos o resultado da imagem 48. Nesse ponto, ainda temos alguns pixels pretos na areia por conta de alguns objetos distorcidos da imagem. Então, aplicamos algumas dilatações na região de interesse (a areia) com um objeto estruturante de uma caixa, assim nos vamos fechando levemente as regiões distorcidas e obtemos o resultado da figura 49. Uma vez que dividimos areia como um objeto inteiramente branco e o mar como o fundo preto, basta aplicarmos um algoritmo de detecção de bordas para obtermos os pixels aproximados do nível do mar. Assim, aplicamos uma correlação entre a imagem e os filtros de Sobel para o eixo x e y para observarmos o fator do gradiente e obtemos a borda com a linha da praia como mostrado na imagem 50. Para saber se o nível do mar está aumentando ou diminuindo, basta percorremos a imagem e capturarmos todos os valores de y com o pixel 255 e compararmos com as imagens anteriores coletadas.

Figura 47 – Região de interesse da imagem.

Figura 48 – Região de interesse binarizada

Figura 49 – Imagem de binarizada após operações de dilatação

Figura 50 – Imagem da linha do oceano após aplicado os filtros de Sobel

Código da questão 10:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from utils import *
from q1 import binarize, two_peaks
from q4 import correlate2d, clip_gray_image
from q5 import edge_dection_kernel, SOBEL_FILTER_X_KERNEL, SOBEL_FILTER_Y_KERNEL
from q8 import dilate

# Image
BOAVIAGEM_BMP_PATH = "./Imagens/Q10/Merge_Timex_BoaViagem.bmp"


def main():
```

```
# Read image
bv_image = read_images([BOAVIAGEM_BMP_PATH])[0]

# Get only region of interest
upper_half_bv_image = bv_image[80 : bv_image.shape[0] // 2, :]
show_image(upper_half_bv_image)

# Convert to LAB representation and get only channel with yellow and blue
blue_yellow = cv2.cvtColor(upper_half_bv_image, cv2.COLOR_BGR2LAB)[:, :, 2]

# Apply binarization
binary_bv_image = binarize(blue_yellow, two_peaks(blue_yellow))

show_image(binary_bv_image)

# Merge image on grey scale just to work with my dilate algorithm
binary_bv_image = np.dstack((binary_bv_image, binary_bv_image, binary_bv_image))

# Applies some dilatation to fill the gaps
for i in range(7):
    binary_bv_image = clip_gray_image(
        dilate(
            binary_bv_image,
            np.array(
                [
                    [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],
                ],
                dtype="uint8",
            )
            * 255,
        )
    )

show_image(binary_bv_image)

# Convert the image to grey to only work with one matrix again
binary_bv_image = cv2.cvtColor(binary_bv_image, cv2.COLOR_BGR2GRAY)

# Applies correlation with the edge dection kernel and finds the ocean line
sobel_x = correlate2d(binary_bv_image, SOBEL_FILTER_X_KERNEL)
sobel_y = correlate2d(binary_bv_image, SOBEL_FILTER_Y_KERNEL)

sobel = clip_gray_image(np.abs(sobel_x) + np.abs(sobel_y))
```

```
    show_image(sobel)


if __name__ == "__main__":
    main()
```

CÓDIGO AUXILIARES DE UTILIDADE(USADO PELA MAIORIA DOS ALGORITMOS):

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


# Load some images
def read_images(paths: list[str]) -> list[np.ndarray]:
    return [cv2.imread(path) for path in paths]


# Save a image
def save_image(image: np.ndarray, path: str) -> None:
    cv2.imwrite(path, image)


# Check if some image is in grey scale
def is_grey_scale(image: np.ndarray) -> None:
    if len(image.shape) < 3:
        return True
    if image.shape[2] == 1:
        return True
    b, g, r = image[:, :, 0], image[:, :, 1], image[:, :, 2]
    if (b == g).all() and (b == r).all():
        return True
    return False


# Assert the images are in grey scale
def assert_images_grey_scale(images: list[np.ndarray]) -> None:
    for image in images:
        assert is_grey_scale(image)


# Applies two_peak algorithm to find cut point on image
def two_peaks(doc: np.ndarray) -> int:
    histogram, bins = np.histogram(doc.ravel(), 256, [0, 256])
    peak_1 = np.argmax(histogram)
```

```
    diffs = np.arange(256).astype("float64")

    for k, h_k in enumerate(histogram):
        diffs[k] = ((k - peak_1) ** 2) * h_k

    peak_2 = np.argmax(diffs)

    return (peak_1 + peak_2) // 2


# Show one image on the screen
def show_image(image: np.ndarray) -> None:
    cv2.imshow("QX", image)
    cv2.waitKey(0)


# Show images on the screen
def show_images(images: list[np.ndarray]) -> None:
    for image in images:
        show_image(image)

    cv2.destroyAllWindows()


# Show images histograms
def show_images_hist(images: list[np.ndarray]) -> None:
    for image in images:
        plt.hist(image.ravel(), 256, [0, 256])
        plt.show()
```