CS484: Introduction to Computer Vision

Homework I

Ali Aral Takak, EEE

22001758

Part I: Morphological Operations

The first part of the homework requires students to write their own implementations of the two fundamental morphological operations, **dilation** and **erosion**.

There are a few constraints on how the task should be handled, and is as listed as follows:

1. The structuring element should be generated as a binary image with an arbitrary shape.
2. Given the structuring element, the code should implement the dilation and erosion operations using the definitions given in the course.
3. The structuring element should be created (as a matrix) outside and given as input to the dilation/erosion functions so that the code can work with any kind of structuring element.
4. In the end, there should only exist white circles approximately the same size, without the noise.

The image that we will be working on can be seen in the figure provided below:

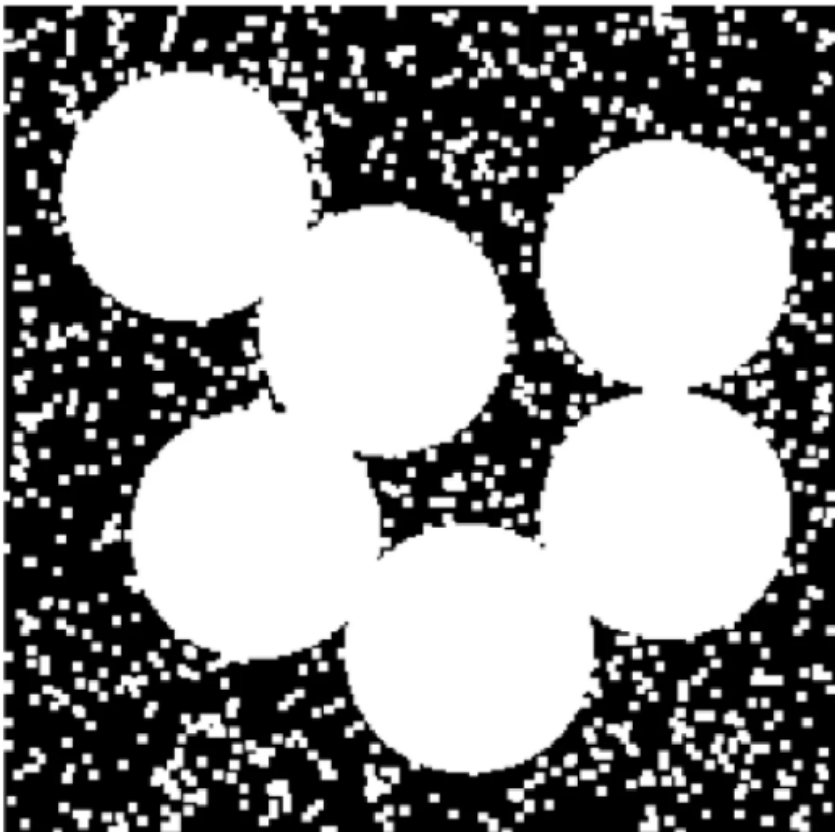

Figure 1: Reference image for morphological operations.

The Python script for this part can be seen in **Appendix A.** The figure provided below displays the final results.

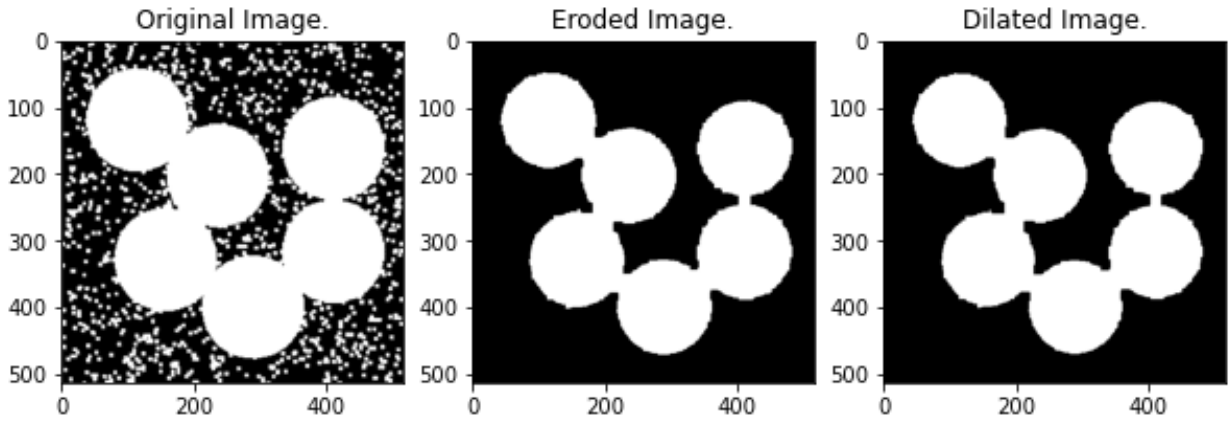

Figure 2: Results of morphological operations.

In this part, by trial and error, we have eliminated the background noise with approximately 6 erosion operations. Then, with the same number of dilation operations, we have enlarged the foreground circles back to their approximate sizes. While erosion removes pixels on object boundaries, dilation adds pixels to them. Hence, that is the reason that we have applied erosion first, and dilation last.

Part II: Histogram Based Image Enhancement

Part II.I: Histogram of a Grayscale Image

In the first part of the Histogram Based Image Enhancement, a function must be implemented to generate a histogram of a grayscale image. The function must be generic for any given grayscale image, and it should only receive the image as a matrix as the input. The test images for this part are provided below:

Figure 3: First grayscale image for histogram generation.

Figure 4: Second grayscale image for histogram generation.

The results can be observed in the figures provided below, with the respective Python script in **Appendix B.**

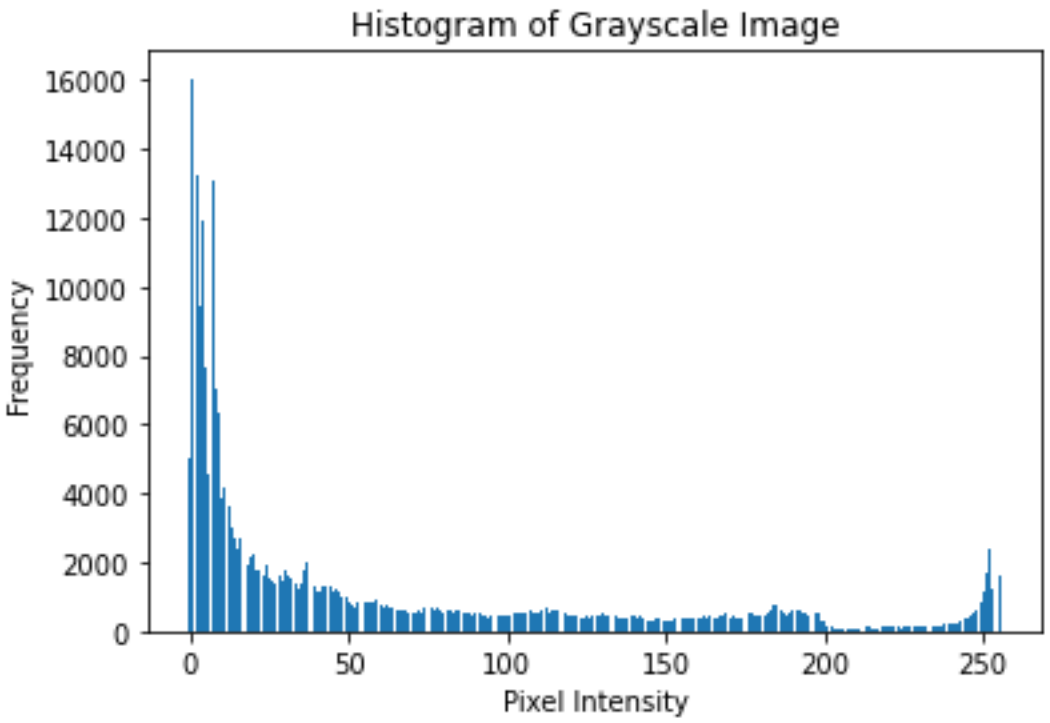

Figure 5: Histogram plot for the first image.

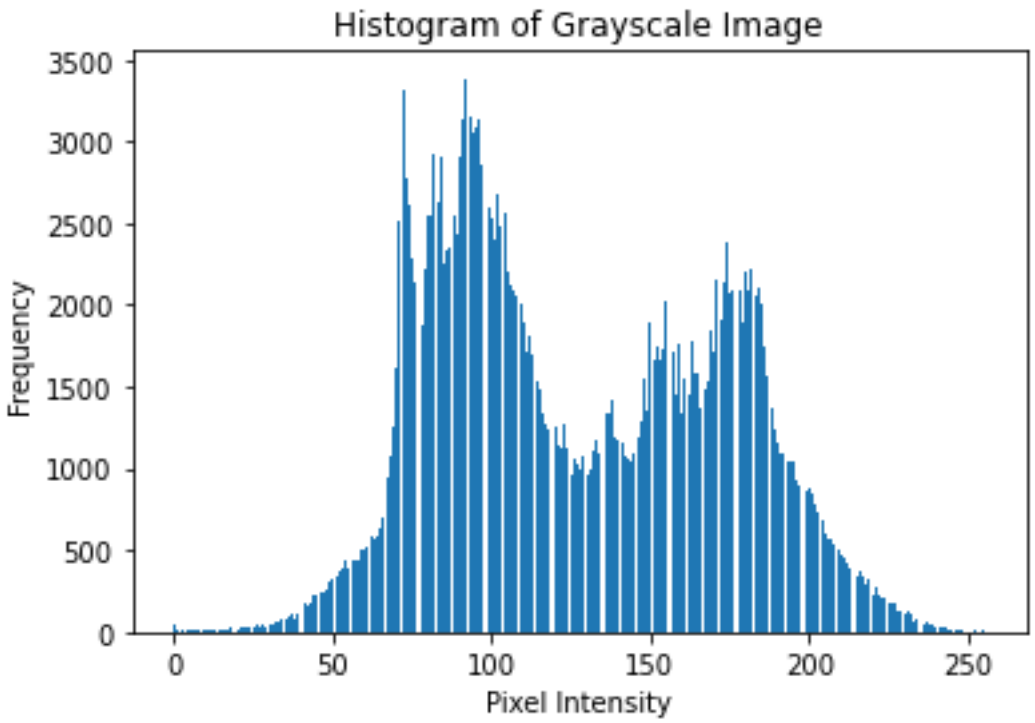

Figure 6: Histogram plot for the second image.

When we inspect the images and the histogram plots, we can conclude that we have generated accurate plots for each image. The first image contains much more pixels that are closer to black, in which they have pixel values closer to zero. Hence, the histogram plot is also skewed to the left side. On the other hand, the second image has a more uniform grayscale color distribution, with color tones similar to each other. Hence, the pixel values are more centered than the first image.

Part II.II: Contrastive Stretching

In this part of the homework, students are required to implement contrast stretching to a given grayscale image, where the formula can be seen below:

$$f(x) = \frac{(x-a)}{(b-a)} \times (d-c) + c$$

Where:

- x is the pixel intensity in the original image.
- a and b are the input image's minimum and maximum pixel intensity values respectively.
- c and d are the desired minimum and maximum pixel intensity values respectively.

The task requires us to experiment with the c and d values of $(0,255), (128,255)$ and $(0,128)$. The respective Python script can be observed in **Appendix B.**

The figures given below display the results for values $(0,255)$.

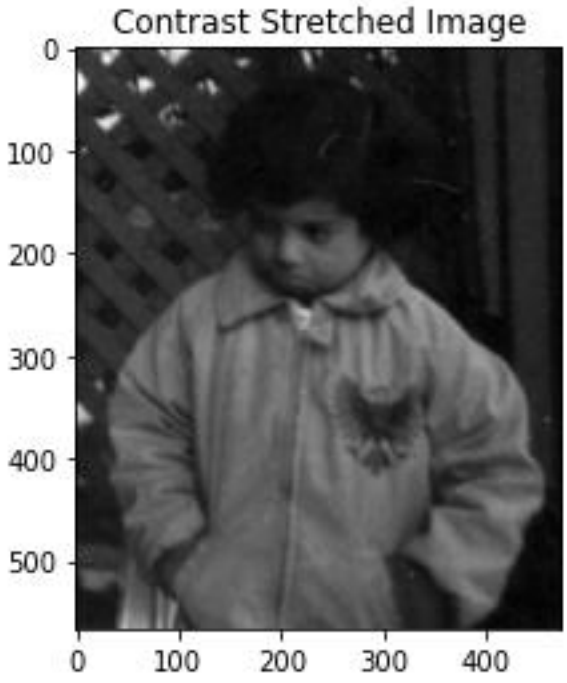

Figure 7: Contrast stretching for values (0,255).

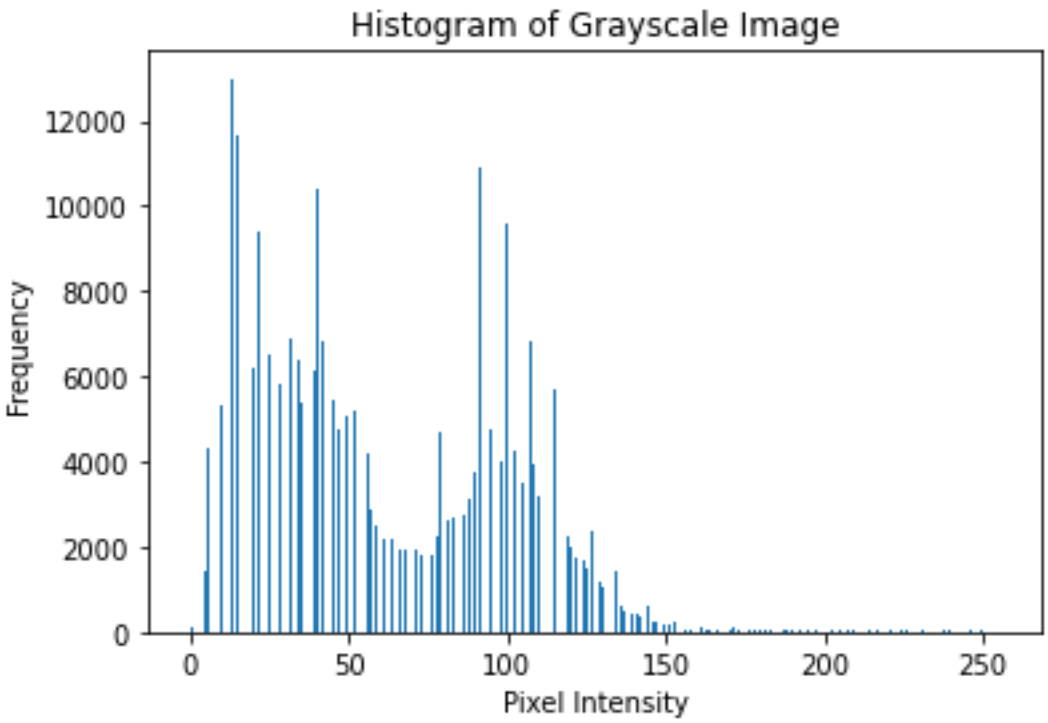

Figure 8: Histogram for values (0,255).

The figures given below display the results for values (0,128).

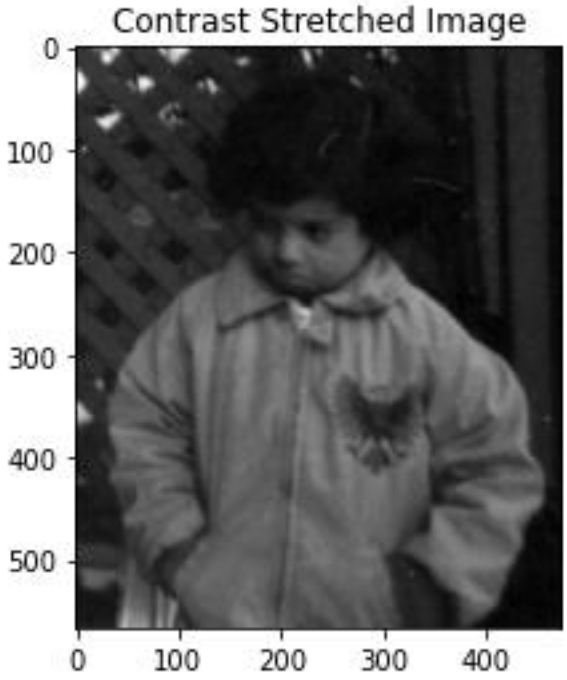

Figure 9: Contrast stretching for values (0,128).

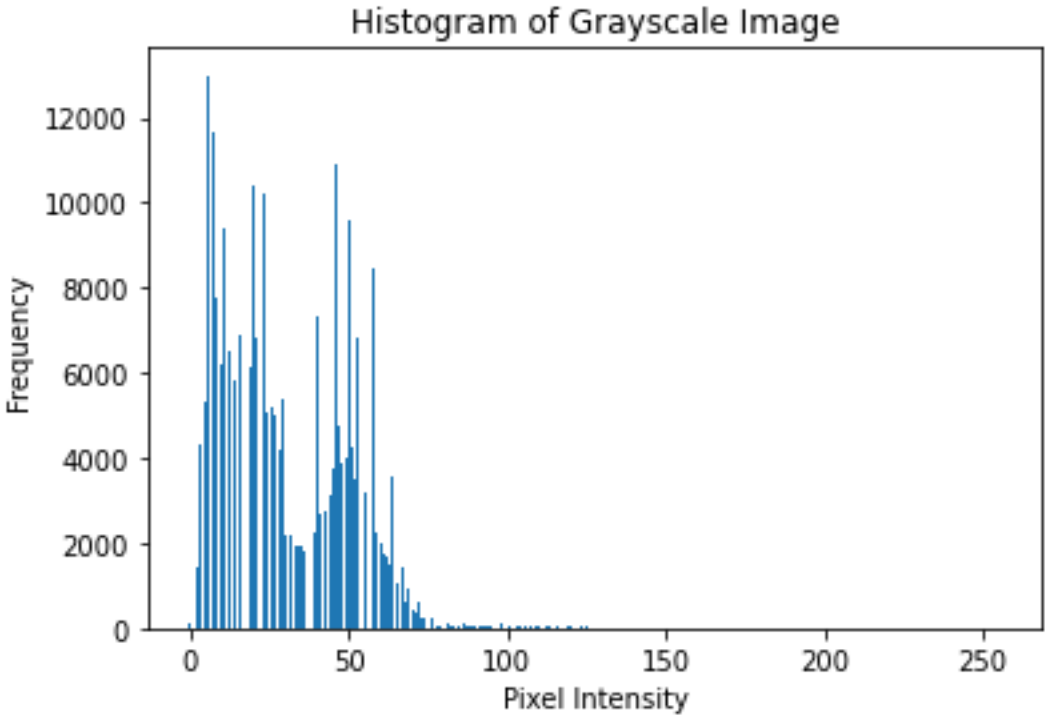

Figure 10: Histogram for values (0,128).

The figures given below display the results for values (128,255).

Figure 11: Contrast stretching for values (128,255).

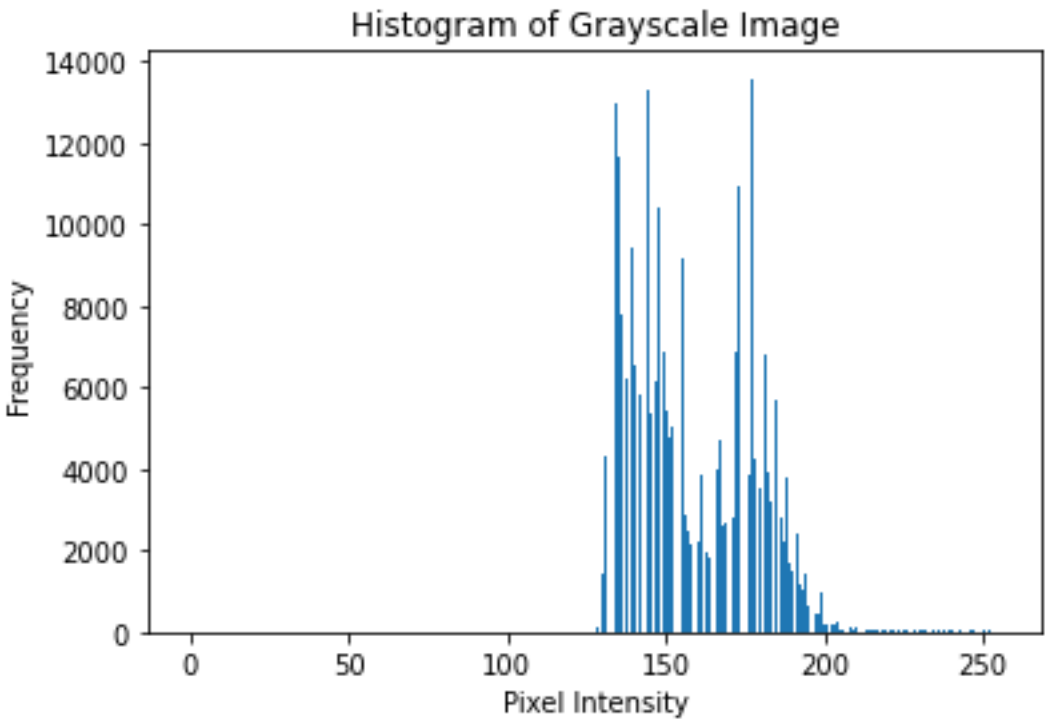

Figure 12: Histogram for values (128,255).

By observing the figures provided above, we can conclude that with the contrast stretching operation, we can manipulate the pixel values in order to adjust the contrast of a given image. Contrast is the difference between the dark areas and the light areas in an image. Increasing the contrast means making darker areas darker and bright areas brighter, hence, a greater difference. On the contrary, reducing the contrast means reducing the difference.

Part III: Otsu Thresholding

The third part of the homework requires an implementation of automated thresholding by Otsu's Method. There are again, a few constraints, such as creating one's own implementation and avoiding image processing libraries, except for reading and/or plotting the images. The function declaration should take a grayscale image as a 2-dimensional matrix as input, and produce a binary image as an output, of the same dimensions. The test images are given below:

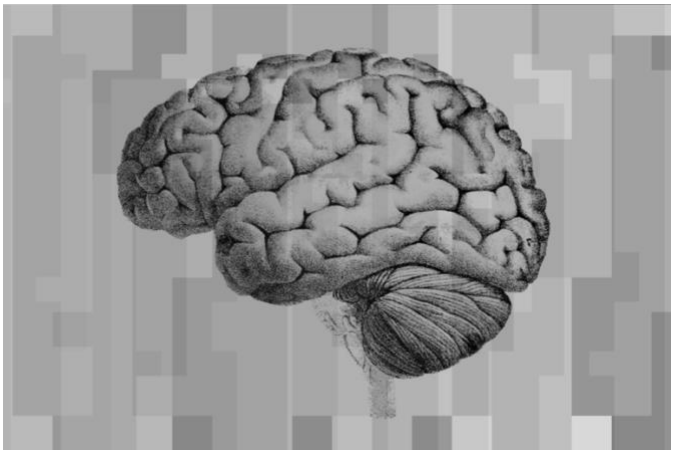

Figure 13: The first test image for Otsu's Method.

Figure 14: The second image for Otsu's Method.

The results can be seen below, with the respective Python script in **Appendix C.**

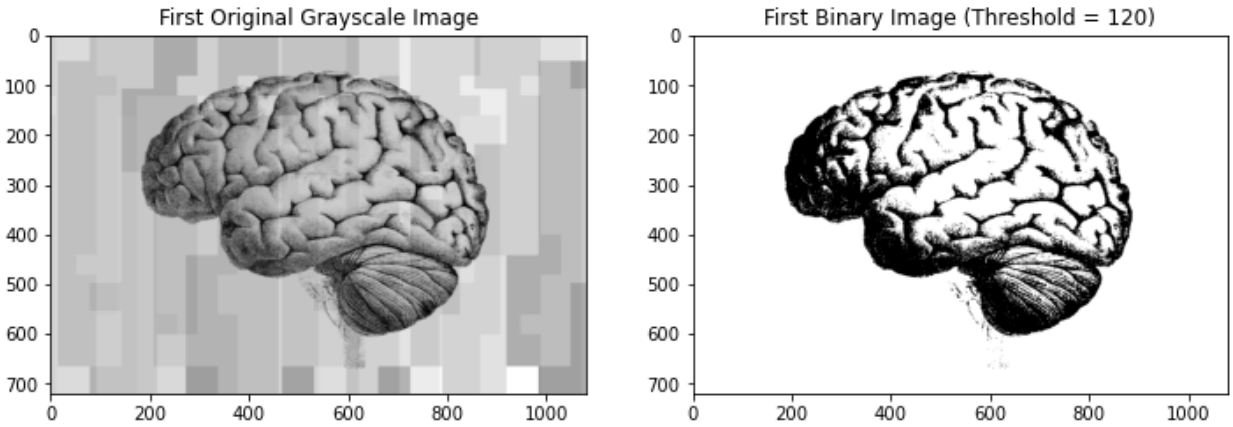

Figure 15: First original grayscale image and its separation using Otsu's Method.

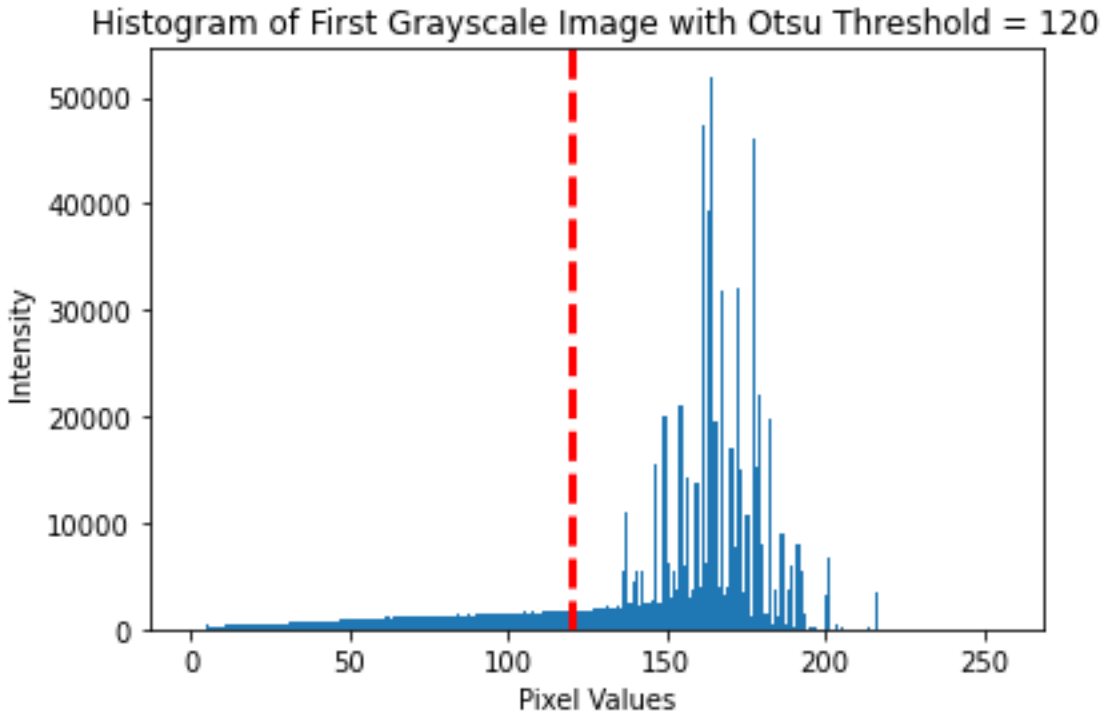

Figure 16: Pixel histogram for the first image.

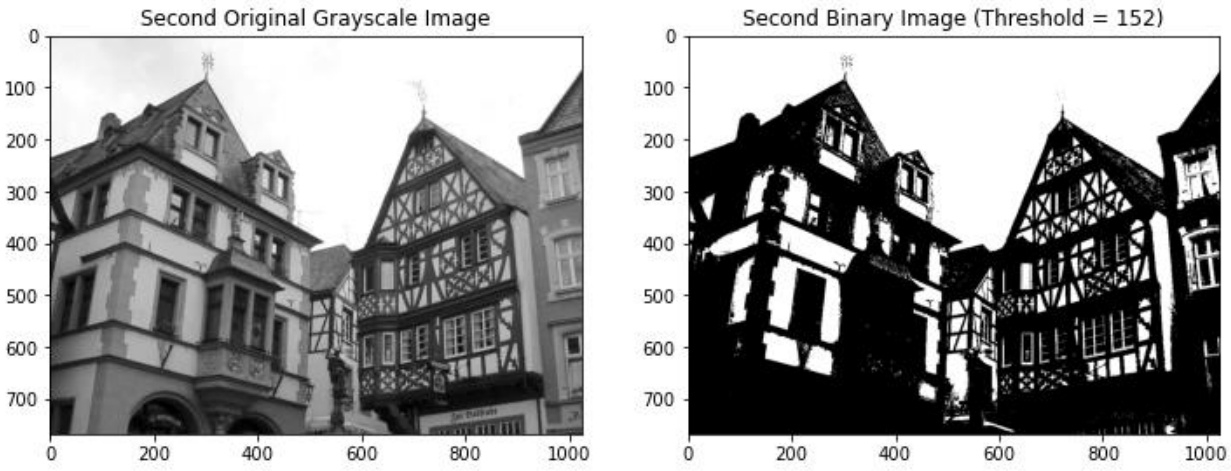

Figure 17: Second original grayscale image and its separation using Otsu's Method.

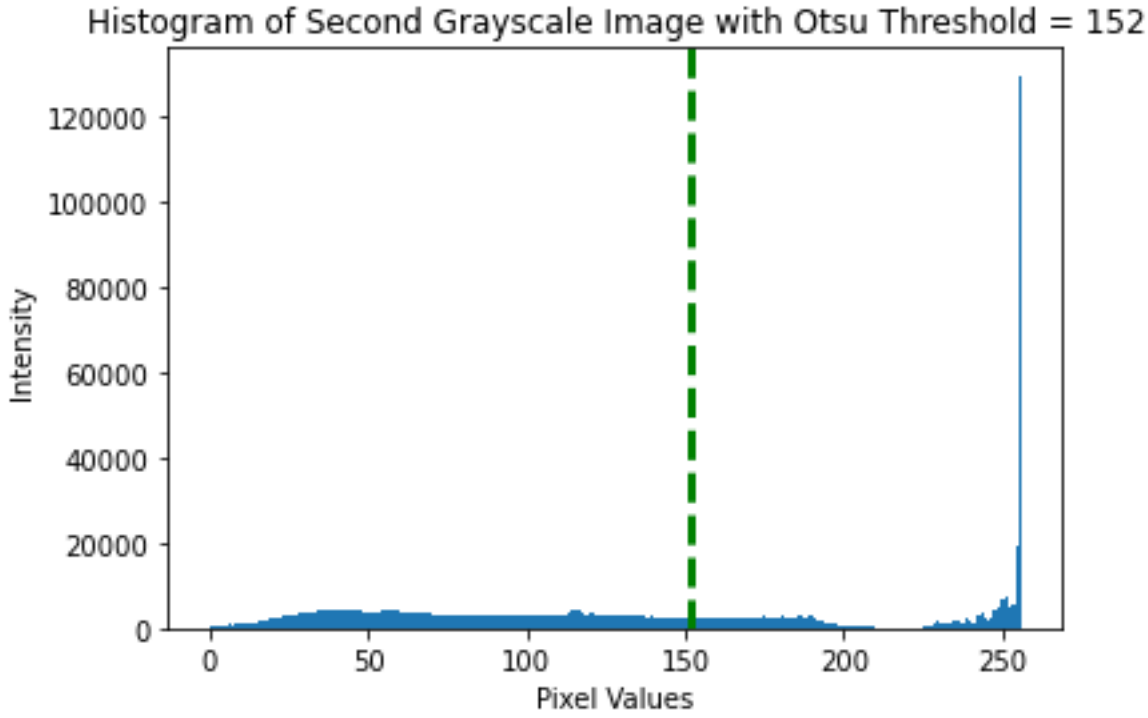

Figure 18: Pixel histogram of the second image.

Even though Otsu's Method is a simple yet effective method, it has its own limitations. Otsu Thresholding works best when there exists a deep distinction between pixel value peaks. If there exists a uniform distribution, noise, or small objects, it performs poorly. In our cases we can comment that the brain shape was segmented efficiently, yet some curves are defined poorly. The other image, on the other hand, was again successfully segmented in the case of main foreground

and background, however, color adjustment on small details were made poorly, possibly due to lighting conditions and adjustment. In conclusion, we can successfully separate the foreground and background with Otsu's Method.

Part IV: 2-D Convolution in Spatial and Frequency Domain

The final part of the homework requires students to perform two-dimensional convolution in spatial domain for edge detection using Sobel and Prewitt operators. The example image for this task can be seen below.

Figure 19: Example image for edge detection.

The second part of this task also requires students to blur an example image using frequency domain convolution and filtering in Gaussian kernel. The example image for blurring can be seen below.

Figure 20: Example image for Gaussian blurring.

The Python script for the tasks mentioned above can be observed in **Appendix D.** The results for Sobel and Prewitt operator spatial convolution can be seen in figures provided below.

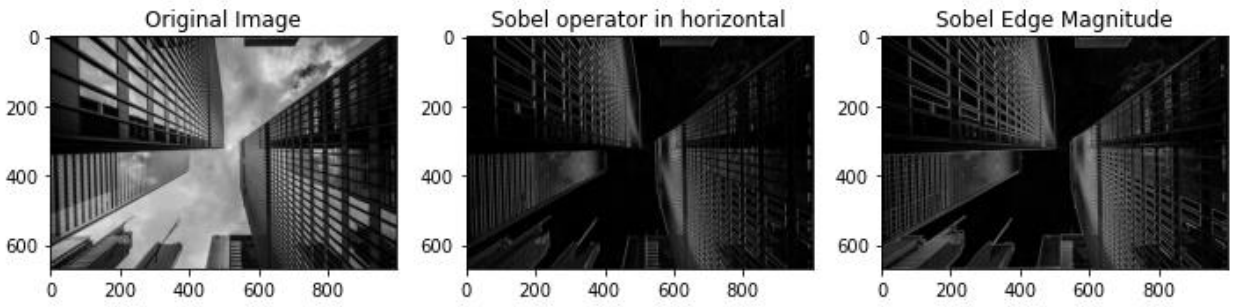

Figure 21: Sobel operator spatial convolution results.

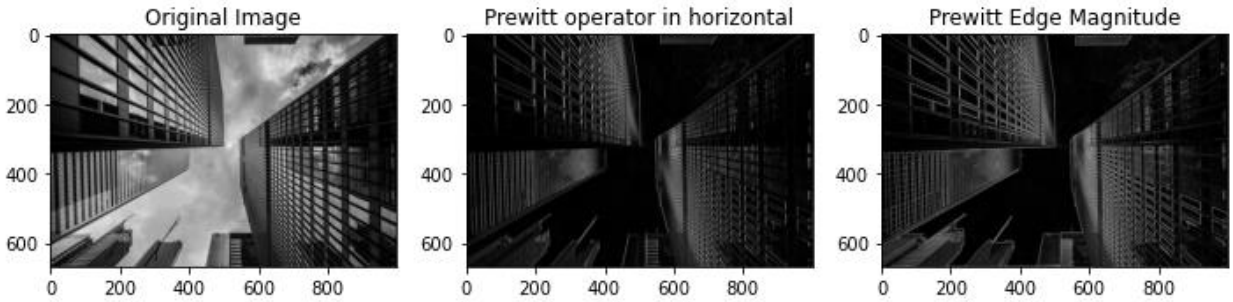

Figure 22: Prewitt operator spatial convolution results.

The results for Gaussian filtering in frequency domain can be seen in the figure provided below.

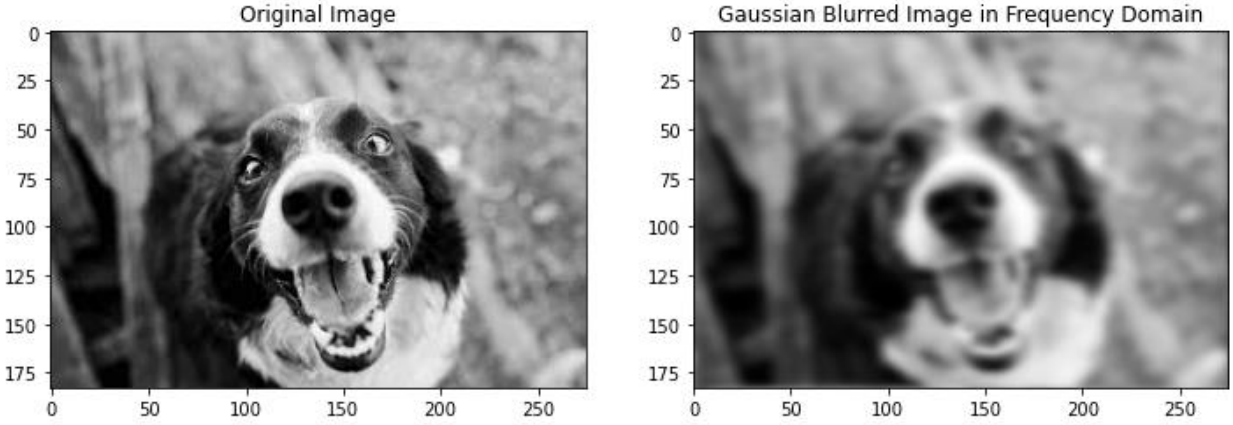

Figure 23: Results of Gaussian blurring in frequency domain.

The Sobel and Prewitt operators are both gradient-based methods for edge detection in images, but they differ slightly in sensitivity and robustness. Sobel gives more weight to pixels farther from the center, making it more responsive to stronger edges and resistant to noise, which provides bolder, clearer edges. The Prewitt operator, in contrast, uses equal weights, making it a simpler approximation of the gradient that is slightly faster but less robust to noise. To sum up, we have implemented two successful methods for edge detection, as can be observed in figures.

In Gaussian filtering, we have used the principle that convolution in time or spatial domain is the equivalent of multiplication in frequency domain. Hence, by converting the image to the frequency domain by using Fourier Transformation, and then multiplying the image with a Gaussian kernel filter, we have blurred the image, and then used inverse Fourier Transform to convert the frequency-domain image to the spatial domain. Hence, we have obtained the result provided above.

Appendices

Appendix A: Python Script for *Morphological Operations*

```
# Import the necessary libraries.
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Define a function for the dilation morphological operation.
def dilation(image, structuring_element):

    # Obtain the dimensions of the source image.
    image_height, image_width = image.shape
    se_height, se_width = structuring_element.shape
    pad_h, pad_w = se_height // 2, se_width // 2

    # Perform padding.
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
    dilated_image = np.zeros_like(image)

    # Perform dilation.
    for i in range(image_height):
        for j in range(image_width):
            region = padded_image[i:i+se_height, j:j+se_width]
            dilated_image[i, j] = (region * structuring_element).max()

    return dilated_image

# Define a function for the erosion morphological operation.
def erosion(image, structuring_element):

    # Obtain the dimensions of the source image.
    image_height, image_width = image.shape
    se_height, se_width = structuring_element.shape
    pad_h, pad_w = se_height // 2, se_width // 2

    # Perform padding.
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
    eroded_image = np.zeros_like(image)

    # Perform erosion.
    for i in range(image_height):
        for j in range(image_width):
            region = padded_image[i:i+se_height, j:j+se_width]
            eroded_image[i, j] = (region * structuring_element).min()

    return eroded_image

# Define the image path to a variable.
```

```
img_path = "/Users/aral/Documents/Homeworks/Homework 1/MorphologicalOperations/morphological_operations.png"

# Define an arbitrary structuring element.
structuring_element = np.array([
    [1, 1, 1],
    [1, 1, 1],
    [1, 1, 1]
])

# Read the example image as grayscale.
binary_image = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

# Apply erosion to the original image.
eroded_image = erosion(binary_image, structuring_element)
eroded_image = erosion(eroded_image, structuring_element)
eroded_image = erosion(eroded_image, structuring_element)
eroded_image = erosion(eroded_image, structuring_element)
eroded_image = erosion(eroded_image, structuring_element)
eroded_image = erosion(eroded_image, structuring_element)


# Apply dilation to the eroded image.
dilated_image = dilation(eroded_image, structuring_element)
dilated_image = dilation(eroded_image, structuring_element)
dilated_image = dilation(eroded_image, structuring_element)
dilated_image = dilation(eroded_image, structuring_element)
dilated_image = dilation(eroded_image, structuring_element)
dilated_image = dilation(eroded_image, structuring_element)


# Plot the results.
plt.figure(figsize=(10, 3))

# Display the original image.
plt.subplot(1, 3, 1)
plt.imshow(binary_image, cmap='gray')
plt.title('Original Image.')


# Display the eroded image.
plt.subplot(1, 3, 2)
plt.imshow(dilated_image, cmap='gray')
plt.title('Eroded Image.')

# Display the dilated image.
plt.subplot(1, 3, 3)
plt.imshow(eroded_image, cmap='gray')
plt.title('Dilated Image.')
```

```
plt.show()
```

Appendix B: Python Script for *Histogram-Based Image Enhancement*

```
# Import the libraries that will be used.
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Define a function to generate the histogram plot of the grayscale image.
def histogram(source_image):

    # Initialize an array of 256 zeros for intensity values.
    histogram_array = [0] * 256

    # Traverse the image and increase the pixel values for each pixel.
    for row in source_image:
        for pixel in row:
            histogram_array[pixel] += 1

    # Generate a histogram plot.
    plt.bar(range(256), histogram_array)
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')
    plt.title('Histogram of Grayscale Image')
    plt.show()

    return histogram_array

# Define a function to execute the contrast stretching operation on a grayscale image.
def contrast_stretching(source_image):

    # Detect the minimum and maximum pixel values in an image.
    pixel_min_val = np.min(source_image)
    pixel_max_val = np.max(source_image)

    # Apply the given contrast stretching formula.
    new_image = ((source_image - pixel_min_val) / (pixel_max_val - pixel_min_val)) * 255
    new_image = new_image.astype(np.uint8)

    return new_image

# Define the image paths.
image_path_1 = "/Users/aral/Documents/Homeworks/Homework 1/HistImageEnhancement/hist1.jpg"
image_path_2 = "/Users/aral/Documents/Homeworks/Homework 1/HistImageEnhancement/hist2.jpg"
image_path_3 = "/Users/aral/Documents/Homeworks/Homework 1/HistImageEnhancement/contrastive_strecth.png"

# Load the images and read them on grayscale.
image_1 = cv2.imread(image_path_1, cv2.IMREAD_GRAYSCALE)
image_2 = cv2.imread(image_path_2, cv2.IMREAD_GRAYSCALE)
image_3 = cv2.imread(image_path_3, cv2.IMREAD_GRAYSCALE)
```

```
# Run the histogram function on the first two images.
histogram(image_1)
histogram(image_2)


# Test with the image (assuming image is a grayscale numpy array)
stretched_img = contrast_stretching(image_3)

# Plot the contrast stretched image
plt.imshow(stretched_img, cmap='gray')
plt.title('Contrast Stretched Image')
plt.show()

# You can also plot the histogram to compare with the original
new_hist = histogram(stretched_img)
```

Appendix C: Python Script for *Otsu Thresholding*

```
# Import the libraries that will be used.
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Define a function to execute automated Otsu's Thresholding on a given grayscale image.
def otsu_method(image):

    # Compute the histogram of the input image.
    histogram = np.zeros(256)
    for pixel_value in image.ravel():
        histogram[int(pixel_value)] += 1

    # Determine the total number of pixels.
    total_pixels = image.size

    # Normalize the histogram to determine the probabilities.
    probability = histogram / total_pixels

    # Compute the cumulative sums and cumulative means.
    cumulative_sum = np.cumsum(probability)
    cumulative_mean = np.cumsum(np.arange(256) * probability)
    global_mean = cumulative_mean[-1]

    # Compute the between-class variance for each threshold.
    between_class_variance = np.zeros(256)
    for t in range(256):
        if cumulative_sum[t] == 0 or cumulative_sum[t] == 1:  # Avoid division by zero
            continue
        between_class_variance[t] = ((global_mean * cumulative_sum[t] - cumulative_mean[t]) ** 2) /
(cumulative_sum[t] * (1 - cumulative_sum[t]))

    # Find the optimal threshold that maximizes between-class variance.
    optimal_threshold = np.argmax(between_class_variance)

    # Apply the threshold to convert the image to binary.
    binary_image = np.where(image >= optimal_threshold, 255, 0).astype(np.uint8)


    return binary_image, optimal_threshold

# Define images paths.
image_path_1 = "/Users/aral/Documents/Homeworks/Homework 1/OtsuThresholding/otsu_1.png"
image_path_2 = "/Users/aral/Documents/Homeworks/Homework 1/OtsuThresholding/otsu_2.jpg"

# Load the images.
image_1 = cv2.imread(image_path_1, cv2.IMREAD_GRAYSCALE)
```

```
image_2 = cv2.imread(image_path_2, cv2.IMREAD_GRAYSCALE)

# Apply Otsu's method to the image
binary_image_1, optimal_threshold_1 = otsu_method(image_1)
binary_image_2, optimal_threshold_2 = otsu_method(image_2)


# Display the original and binary images
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(image_1, cmap='gray')
plt.title('First Original Grayscale Image')

plt.subplot(1, 2, 2)
plt.imshow(binary_image_1, cmap='gray')
plt.title(f'First Binary Image (Threshold = {optimal_threshold_1})')

plt.show()

# Plot the histogram of the image with the threshold marked
plt.hist(image_1.ravel(), bins=256, range=(0, 256))
plt.axvline(optimal_threshold_1, color='red', linestyle='dashed', linewidth=3)
plt.title(f'Histogram of First Grayscale Image with Otsu Threshold = {optimal_threshold_1}')
plt.xlabel("Pixel Values")
plt.ylabel("Intensity")
plt.show()

# Display the original and binary images
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(image_2, cmap='gray')
plt.title('Second Original Grayscale Image')

plt.subplot(1, 2, 2)
plt.imshow(binary_image_2, cmap='gray')
plt.title(f'Second Binary Image (Threshold = {optimal_threshold_2})')

plt.show()

# Plot the histogram of the image with the threshold marked
plt.hist(image_2.ravel(), bins=256, range=(0, 256))
plt.axvline(optimal_threshold_2, color='green', linestyle='dashed', linewidth=3)
plt.title(f'Histogram of Second Grayscale Image with Otsu Threshold = {optimal_threshold_2}')
plt.xlabel("Pixel Values")
plt.ylabel("Intensity")
plt.show()
```

Appendix D: Python Script for *2-D Convolution in Spatial and Frequency Domain*

```
# Import the required libraries.
import numpy as np
import cv2
import matplotlib.pyplot as plt
from numpy.fft import fft2, ifft2, fftshift


# Define spatial domain convolution function.
def spatial_convolution(image, kernel):
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape
    kernel = np.flipud(np.fliplr(kernel))
    pad_height, pad_width = kernel_height // 2, kernel_width // 2
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width, pad_width)), mode='constant')
    output = np.zeros_like(image)

    for i in range(image_height):
        for j in range(image_width):
            region = padded_image[i:i+kernel_height, j:j+kernel_width]
            output[i, j] = np.sum(region * kernel)

    return output

# Define a function to create a Gaussian kernel in the frequency domain.
def gaussian_kernel(shape, sigma):

    m, n = shape
    y, x = np.indices((m, n))
    center_y, center_x = m // 2, n // 2
    kernel = np.exp(-((x - center_x)**2 + (y - center_y)**2) / (2 * sigma**2))
    return kernel / kernel.sum()


# Define Sobel operators.
sobel_horizontal = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_vertical = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

# Define Prewitt operators.
prewitt_horizontal = np.array([[-1, 0, 1],
                               [-1, 0, 1],
                               [-1, 0, 1]])

prewitt_vertical = np.array([[-1, -1, -1],
                             [0, 0, 0],
                             [1, 1, 1]])
```

```
# Read the image in grayscale and normalize.
spatial_image_path = "/Users/aral/Documents/Homeworks/Homework 1/2DConv/convolution_spatial_domain.jpg"
spatial_image = cv2.imread(spatial_image_path, cv2.IMREAD_GRAYSCALE)

gaussian_image_path = "/Users/aral/Documents/Homeworks/Homework 1/2DConv/convolution_freq_domain.jpg"
gaussian_image = cv2.imread(gaussian_image_path, cv2.IMREAD_GRAYSCALE)

spatial_image = spatial_image.astype(np.float32) / 255.0
gaussian_image = gaussian_image.astype(np.float32) / 255.0

# Apply Sobel operator.
sobel_horizontal_result = spatial_convolution(spatial_image, sobel_horizontal)
sobel_vertical_result = spatial_convolution(spatial_image, sobel_vertical)

# Take absolute values and combine results for gradient magnitude for Sobel operator.
sobel_magnitude = np.sqrt(np.abs(sobel_horizontal_result)**2 + np.abs(sobel_vertical_result)**2)
sobel_magnitude = (sobel_magnitude / sobel_magnitude.max()) * 255  # Normalize to [0, 255]
sobel_magnitude = sobel_magnitude.astype(np.uint8)

# Apply Prewitt operator.
prewitt_horizontal_result = spatial_convolution(spatial_image, prewitt_horizontal)
prewitt_vertical_result = spatial_convolution(spatial_image, prewitt_vertical)

# Take absolute values and combine results for gradient magnitude for Prewitt operator.
prewitt_magnitude = np.sqrt(np.abs(prewitt_horizontal_result)**2 + np.abs(prewitt_vertical_result)**2)
prewitt_magnitude = (prewitt_magnitude / prewitt_magnitude.max()) * 255  # Normalize to [0, 255]
prewitt_magnitude = prewitt_magnitude.astype(np.uint8)


# Plot results for Sobel operator.
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.imshow(spatial_image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 3, 2)
plt.imshow(np.abs(sobel_horizontal_result), cmap='gray')
plt.title('Sobel operator in horizontal')

plt.subplot(1, 3, 3)
plt.imshow(sobel_magnitude, cmap='gray')
plt.title('Sobel Edge Detection')

plt.show()


# Plot results for Prewitt operator.
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
```

```
plt.imshow(spatial_image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 3, 2)
plt.imshow(np.abs(prewitt_horizontal_result), cmap='gray')
plt.title('Prewitt operator in horizontal')

plt.subplot(1, 3, 3)
plt.imshow(prewitt_magnitude, cmap='gray')
plt.title('Prewitt Edge Detection')

plt.show()

# Apply the Fourier Transform to the image.
image_fft = fft2(gaussian_image)
image_fft_shifted = fftshift(image_fft)

# Define the Gaussian kernel in the frequency domain.
sigma = 15  # Adjust the sigma value for more or less blurring.
gaussian_filter = gaussian_kernel(gaussian_image.shape, sigma)

# Apply the Gaussian filter in the frequency domain.
filtered_fft = image_fft_shifted * gaussian_filter

# Inverse shift and inverse Fourier Transform to convert back to spatial domain.
filtered_image = np.real(ifft2(np.fft.ifftshift(filtered_fft)))

# Plot the original and the filtered images.
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(gaussian_image, cmap='gray')
plt.title("Original Image")

plt.subplot(1, 2, 2)
plt.imshow(filtered_image, cmap='gray')
plt.title("Gaussian Blurred Image in Frequency Domain")

plt.show()
```