University of Moratuwa, Sri Lanka

Faculty of Engineering

Department of Electronic and Telecommunication Engineering
Semester 5 (Intake 2021)

EN3160 - Image Processing and Machine Vision

Assignment 1 : Intensity Transformations and Neighborhood Filtering

Kodikara U. S. S.
210293K

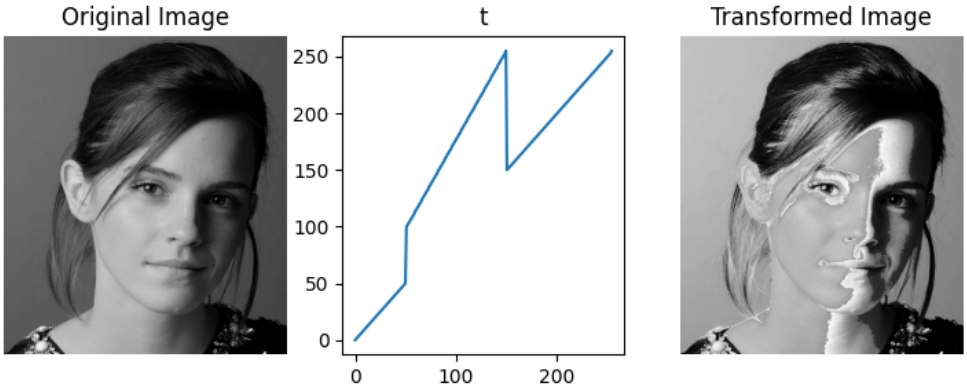
This report is submitted as a partial fulfillment for the module EN3160 - Image Processing and Machine Vision, Department of Electronic and Telecommunication Engineering, University of Moratuwa.

1 Question 01

The intensity transformation acts as an identity transformation with enhancements applied to mid-range gray pixels (50 to 150), resulting in higher contrast regions in the output image. This process introduces jump discontinuities in the transformation function, leaving high-intensity (white) and low-intensity (dark) pixels unchanged. Consequently, pixel values within the specified range are enhanced, while those outside this range remain unaffected.

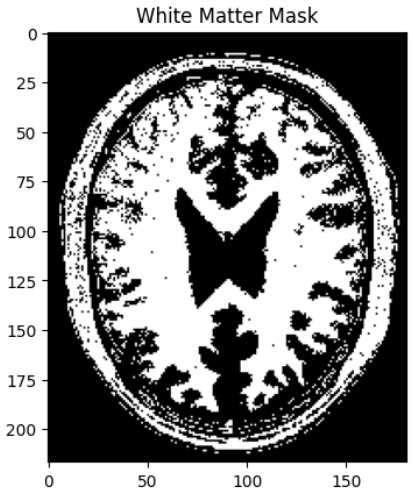
```
    t1 = np.linspace(0, 50, 51).astype('uint8')
    t2 = np.linspace(50, 100, 0).astype('uint8')
    t3 = np.linspace(100, 255, 100).astype('uint8')
    t4 = np.linspace(255, 150, 0).astype('uint8')
    t5 = np.linspace(150, 255, 105).astype('uint8')
    transform = np.concatenate((t1, t2, t3, t4, t5), axis=0).astype('uint8')
    g = cv.LUT(f, transform)
```

Listing 1: Intensity transformation

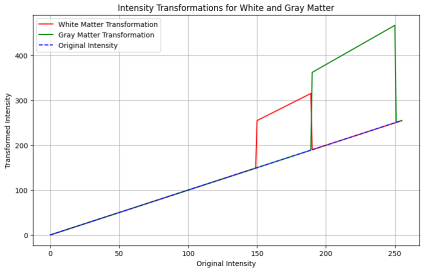


2 Question 02

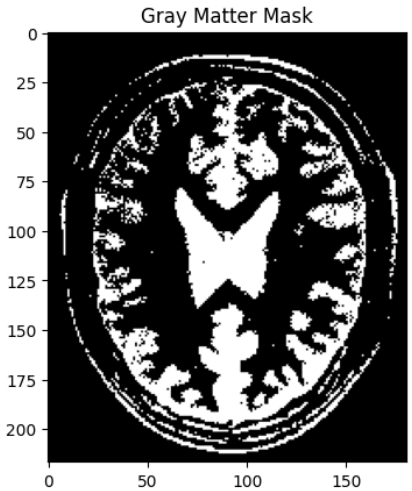
Approach: In this task, I identified the white and gray matter intensities by selecting random points within their respective areas. I then applied thresholding to create binary masks that isolate these regions.



White Matter



Transformation Function



Gray Matter

```
    # Apply transformation for white matter
    gray_matter_mask = (brain_proton_img >= 190) & (brain_proton_img <= 250)
    transformed_image[gray_matter_mask] = 1.75 * brain_proton_img[gray_matter_mask] + 30
    # Apply transformation for gray matter
```

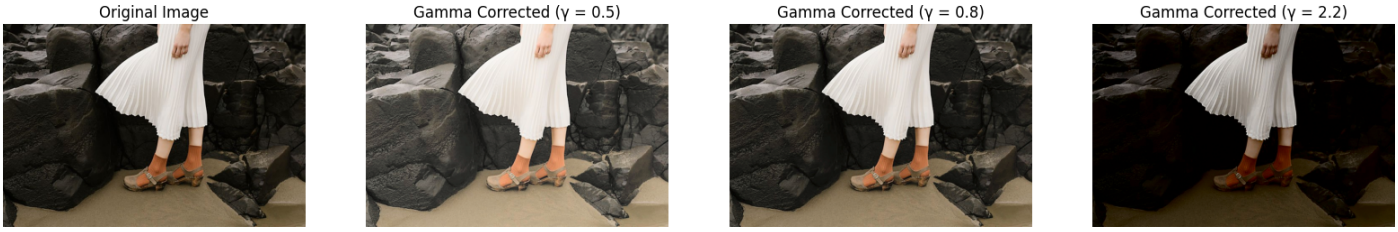
```
    white_matter_mask = (brain_proton_img >= 150) & (brain_proton_img <= 189)
    transformed_image[white_matter_mask] = 1.55 * brain_proton_img[white_matter_mask] +
        22.5
```

Listing 2: Intensity transformation

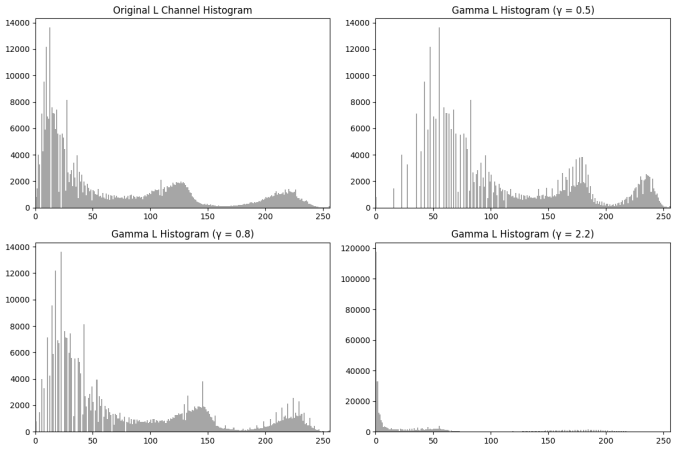
Two linear transformations were applied to distinctly enhance white and gray matter in the brain image. The thresholds for **white (190–250)** and **gray matter (150–189)** were chosen based on intensity ranges. White matter was brightened using a stronger transformation, while gray matter was subtly enhanced.

3 Question 03

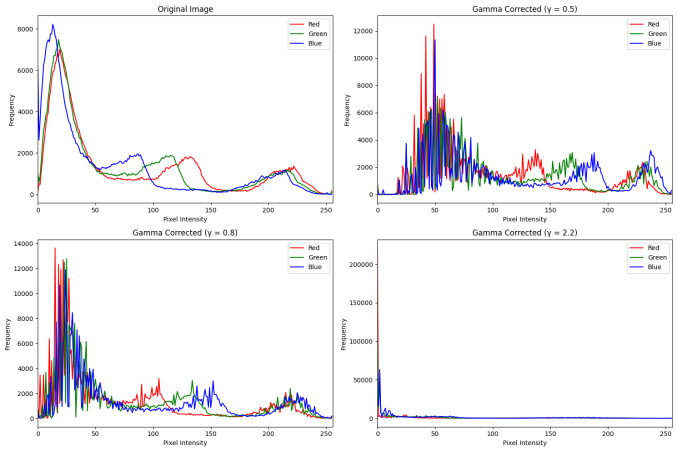
Gamma correction ($\gamma = 0.8$) was applied to the **L** channel of the image after converting it to the L*a*b* color space. This operation brightened the image, particularly enhancing darker areas such as rock hollows. The L channel represents lightness, and applying a gamma value less than 1 increases pixel values, resulting in a brighter image. This is illustrated in the histogram, where pixel intensity shifts to the right, indicating an increase in brightness.



Original and gamma corrected images



Histograms of L plane



Histograms after converting to RGB

```
    L, A, B = cv2.split(lab_img) # Split into L, A, and B channels
    L_normalized = L / 255.0 # Normalize the L channel to [0, 1]
    gamma_values = [0.5, 0.8, 2.2] # Define gamma values
    corrected_images = []
    for gamma in gamma_values: #Apply gamma values
        L_gamma_corrected = np.power(L_normalized, gamma)
        L_gamma_corrected = np.uint8(L_gamma_corrected * 255)
        # Merge the corrected L channel with A and B channels
        lab_gamma_corrected = cv2.merge((L_gamma_corrected, A, B))
        gamma_corrected_img = cv2.cvtColor(lab_gamma_corrected, cv2.COLOR_Lab2BGR)
        corrected_images.append(gamma_corrected_img)
```

Listing 3: Intensity transformation

The histograms of the gamma-corrected image show an increase in noise levels. Gamma correction, being a non-linear operation applied to the L plane (which represents lightness in color spaces like LAB), can amplify

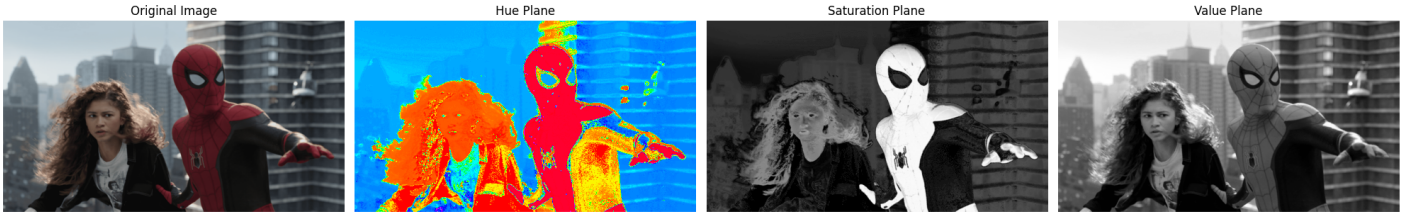
small differences in pixel intensity. This non-linear transformation tends to unevenly spread discrete pixel values, leading to quantization noise. As a result, the histogram appears noisy due to the redistribution of pixel intensities, highlighting the impact of gamma correction on image quality.

4 Question 04

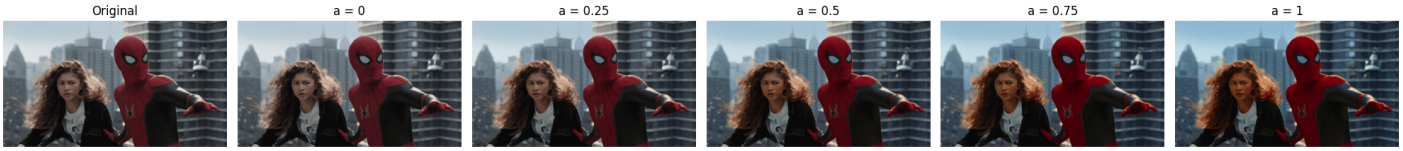
In this question, a vibrancy transformation was applied to the saturation channel of the image to enhance colorful areas while maintaining the intensity of colorless regions. The transformation uses the equation:

$$f(x) = \min\left(x + a \cdot 128 \cdot e^{-\frac{(x-128)^2}{2\sigma^2}}, 255\right)$$

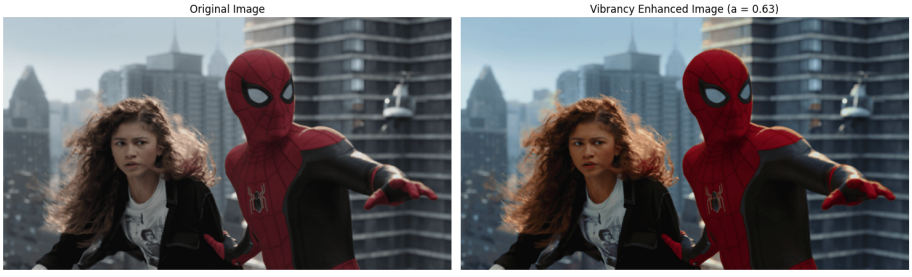
where x is the input intensity, $a \in [0, 1]$, and $\sigma = 70$. A value of **0.63** for a was selected through an interactive trackbar, allowing real-time adjustments to evaluate the impact on vibrance. This transformation effectively amplifies the vibrance of the image, enhancing colorful features while preserving the natural look of less saturated areas.



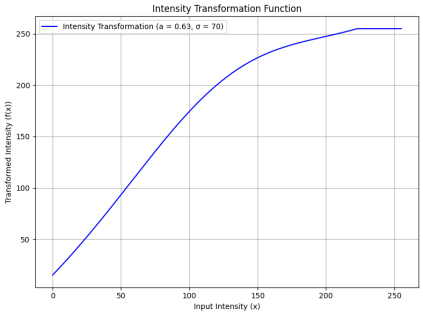
Splitted planes



Different a values



Visually pleased output



Intensity transformation

```
    # Split into HSV planes
    spider_hue_plane = hsv_spider_img[:, :, 0]
    spider_saturation_plane = hsv_spider_img[:, :, 1]
    spider_value_plane = hsv_spider_img[:, :, 2]
    # Function for vibrancy transformation
    def vibrancy_transformation(saturation_plane, a, sigma=70):
        x = saturation_plane.astype(float)
        return np.minimum(x + a * 128 * np.exp(-(x - 128) ** 2 / (2 * sigma ** 2)), 255)
    # Recombine function to merge planes after transformation
    def recombine_planes(hue_plane, new_saturation_plane, value_plane):
        new_hsv_img = cv2.merge([hue_plane, new_saturation_plane.astype(np.uint8),
            value_plane])
```

```
        new_bgr_img = cv2.cvtColor(new_hsv_img, cv2.COLOR_HSV2BGR)
        return new_bgr_img
```

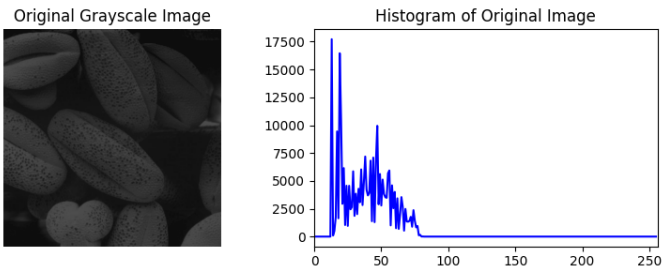
Listing 4: Important code snippets

The saturation plane reveals elevated pixel values in the vibrant areas of the image (e.g., the spider suit), while colorless regions, such as buildings and the sky, exhibit low pixel values. By selectively enhancing the saturation of these colorful areas while preserving the intensity of the colorless regions, the vibrance of the image is significantly increased, resulting in a more visually appealing representation.

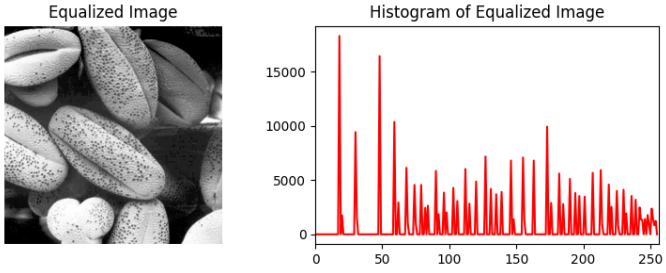
5 Question 05

```
    def histogram_equalization(image):
        # Convert the image to grayscale
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        # Compute the histogram of the original image
        hist_orig, bins_orig = np.histogram(gray_image.flatten(), 256, [0, 256])
        # Calculate the cumulative distribution function (CDF)
        cdf = hist_orig.cumsum()
        # Normalize CDF
        cdf_normalized = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())  # Normalize
        # Use the CDF to map the pixel values of the original image
        equalized_image = np.interp(gray_image.flatten(), bins_orig[:-1], cdf_normalized
           ).reshape(gray_image.shape).astype(np.uint8)
        # Compute the histogram of the equalized image
        hist_eq, bins_eq = np.histogram(equalized_image.flatten(), 256, [0, 256])
        return gray_image, equalized_image, hist_orig, hist_eq, cdf_normalized
```

Listing 5: Function for Histogram equalization



Before equalization



After equalization

The histogram of the image, as shown in the results, exhibits a more even distribution following the equalization process. However, this enhancement has introduced some quantization noise, which is evident in certain areas of the histogram. Notably, the low-intensity peaks remain unchanged, indicating that while the overall contrast has improved, the representation of darker pixel values has been preserved.

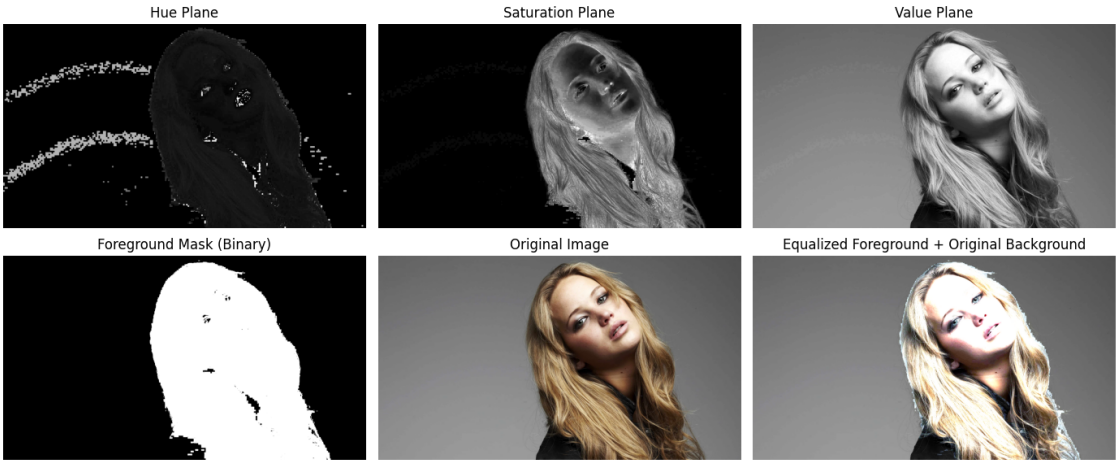
6 Question 06

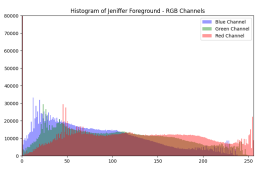
The image was split into its Hue, Saturation, and Value (HSV) planes. A binary mask based on the saturation plane was used to extract the foreground. The resulting histogram showed the pixel intensity distribution, followed by histogram equalization to enhance contrast. This process increased the visibility of dark pixels, particularly in the shadows, improving the overall detail of the foreground image.

```
    hue_plane, saturation_plane, value_plane = cv2.split(hsv_image) # Split the HSV
       image
    foreground_mask = saturation_plane > 14  # Thresholding the saturation plane
    foreground_only = cv2.bitwise_and(image, image, mask=foreground_mask.astype(np.uint8
       ))
    # Obtain the cumulative sum of the histograms
```

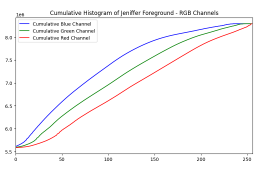
```
    cumsum_b = np.cumsum(hist_b), cumsum_g = np.cumsum(hist_g), cumsum_r = np.cumsum(
        hist_r)
    # Histogram equalization
    equalized = cv2.equalizeHist(foreground_only[:, :, i])
    # Merge equalized channels into one image
    equalized_img = cv2.merge([blue_equalized, green_equalized, red_equalized])
    # Create the inverse of the foreground mask to get the background
    background_mask = cv2.bitwise_not(foreground_mask.astype(np.uint8))
    background_only = cv2.bitwise_and(image, image, mask=background_mask)
    # Combine the histogram-equalized foreground with the original background
    combined_image = cv2.add(equalized_img, background_only)
```

Listing 6: Code snippets

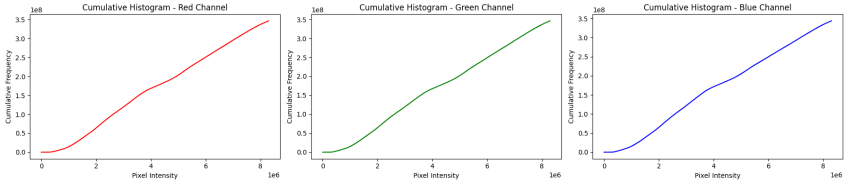




Before equalization



Before equalization



Cumalative histogram after equalization

The foreground extraction effectively isolates colorful regions, as indicated by the binary mask from the saturation plane. Histogram analysis shows improved pixel intensity distribution in the foreground, while histogram equalization enhances contrast. The combined image highlights the foreground elements, demonstrating the effectiveness of the applied transformations.

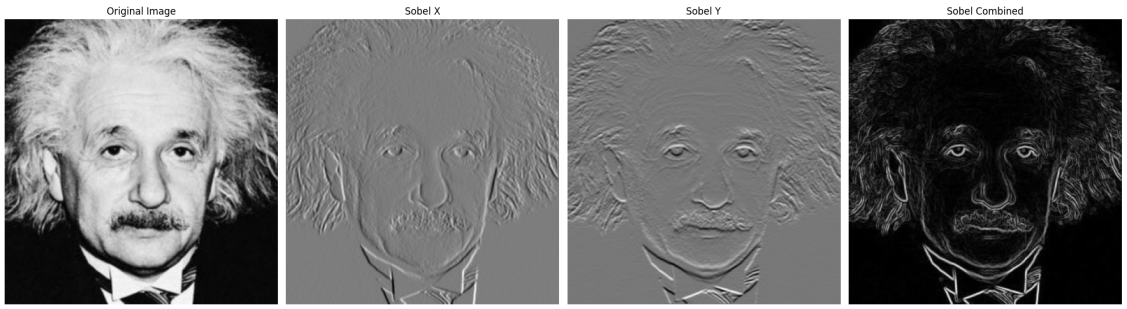
7 Question 07

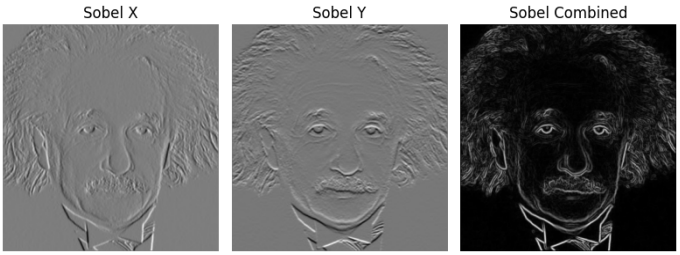
The Sobel operator was used to compute the image gradient, highlighting edges by applying the `filter2D` function. This method effectively emphasized areas of significant intensity change in the image.

```
    def apply_sobel_filter(image):
        # Sobel kernels for x and y directions
        sobel_x = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
        sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
        rows, cols = image.shape
        # Initialize output arrays for filtered images
        output_x = np.zeros_like(image, dtype=np.float64)
        output_y = np.zeros_like(image, dtype=np.float64)
        # Apply Sobel filtering manually
        for r in range(1, rows - 1):
            for c in range(1, cols - 1):
                region = image[r-1:r+2, c-1:c+2]
```

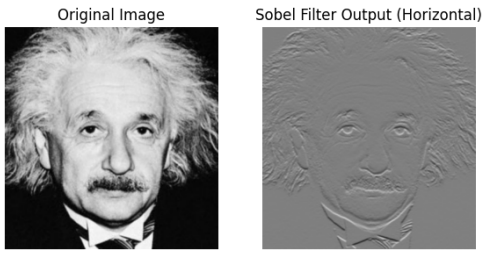
```
                output_x[r, c] = np.sum(region * sobel_x)
                output_y[r, c] = np.sum(region * sobel_y)
        # Combine the gradients for the final Sobel filtered result
        sobel_combined = np.sqrt(output_x**2 + output_y**2)
        return output_x, output_y, sobel_combined
```

Listing 7: My code for the sobel filter





My function



Part c

8 Question 08

The following function was used to zoom the images using a given interpolation method. The SSD values calculated for each image is given in the caption.

```
    def nearest_neighbor_zoom(image, scale):
    height, width = image.shape[:2]
    return image[np.clip(np.arange(int(height * scale)) / scale, 0, height - 1).astype(
        int), np.clip(np.arange(int(width * scale)) / scale, 0, width - 1).astype(int)]
```

Listing 8: Nearest-Neighbor Interpolation

```
    def bilinear_zoom(image, scale):
        height, width = image.shape[:2]
        new_height, new_width = int(height * scale), int(width * scale)
        zoomed_image = np.zeros((new_height, new_width, image.shape[2]), dtype=image.
            dtype)
        for i in range(new_height):
            for j in range(new_width):
                x, y = i / scale, j / scale
                x1, y1, x2, y2 = int(x), int(y), min(int(x) + 1, height - 1), min(int(y)
                    + 1, width - 1)
                zoomed_image[i, j] = (1 - (x - x1)) * (1 - (y - y1)) * image[x1, y1] + \
                                     (1 - (x - x1)) * (y - y1) * image[x1, y2] + \
                                     (x - x1) * (1 - (y - y1)) * image[x2, y1] + \
                                     (x - x1) * (y - y1) * image[x2, y2]
        return zoomed_image
```

Listing 9: Bilinear Interpolation

Nearest Neighbor: SSD value: 847702500.0 (Normalized: 408.807), Original size = (270, 480, 3), Zoomed size = (1080, 1920, 3)

Bilinearr: SSD value: 338416120.0 (Normalized: 146.881), Original size = (300, 480, 3), Zoomed size = (1200, 1920, 3)

The results indicate that bilinear interpolation, with a lower SSD value (338,416,120.0), offers improved image quality over nearest-neighbor interpolation (847,702,500.0). This suggests that bilinear interpolation reduces blockiness and artifacts, providing a smoother transition between pixel values, especially in areas with gradual intensity changes.

9 Question 09

```
    mask = np.zeros(flower_image.shape[:2], np.uint8)
    bg_model = np.zeros((1, 65), np.float64)
    fg_model = np.zeros((1, 65), np.float64)
    rect = (50, 50, flower_image.shape[1] - 50, flower_image.shape[0] - 50)
    # Apply GrabCut algorithm
    cv.grabCut(flower_image, mask, rect, bg_model, fg_model, 5, cv.GC_INIT_WITH_RECT)
    # Update the mask to extract the foreground and background
    final_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
    foreground = flower_image * final_mask[:, :, np.newaxis]
    background = flower_image * (1 - final_mask[:, :, np.newaxis])
    blurred_background = cv.GaussianBlur(background, (15, 15), 0) # Apply Gaussian
    enhanced_image = cv.add(foreground, blurred_background)
```

Listing 10: Part a

Reason for Dark Background Beyond the Flower Edge
During the blurring process using a Gaussian kernel, pixels corresponding to the foreground are replaced with zero values. As the blur functions as an averaging filter, the border areas near the flower are blended with these zero-value pixels, resulting in a darker appearance at the edges.