Department of Electronic and Telecommunication Engineering
University of Moratuwa

EN3160 - Image Processing and Machine Vision

Intensity Transformations and Neighborhood Filtering

210204R Hansindu K.A.A.

1 Implementing Intensity Transformation

Implemented the intensity transformation given on the grayscale image in. The transformation maps specific input intensity ranges to output intensities as described in the following intervals:

- 0 to 50: Mapped to 0 to 50
- 51 to 150: Mapped to 100 to 255
- 151 to 255: Mapped to 150 to 255

```
# Define transformation ranges
t1 = np.linspace(0, 50, 51).astype('uint8')
t2 = np.linspace(100, 255, 100).astype('uint8')
t3 = np.linspace(151, 255, 105).astype('uint8')
transform = np.concatenate((t1, t2, t3), axis=0)

# Apply the transformation to the grayscale image
img_orig = cv.imread('emma.jpg', cv.IMREAD_GRAYSCALE)
image_transformed = cv.LUT(img_orig, transform)
```

Listing 1: Code Implementation (Key Snippet)

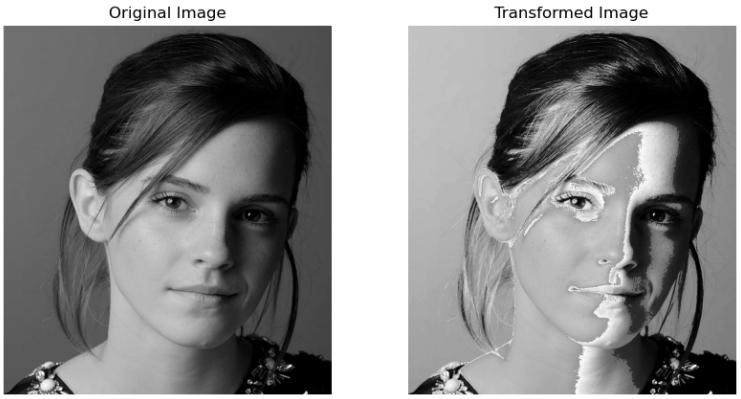


Figure 1: Image Comparison.

The transformation brightened mid-intensity pixels significantly, while preserving more of the higher intensity values. The result is an image where the middle-intensity regions are enhanced.

2 White and Gray Matter Accentuated Image Transformation

2.1 Contour Plot to Identify Intensity Levels

To identify the intensity range for white and gray matter, I plotted a contour plot of the brain proton density image.

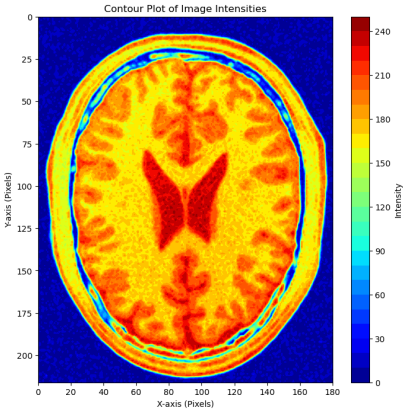


Figure 2: Contour Plot of Image Intensities.

This helped to roughly identify the intensity ranges for white and gray matter as:

- White matter: Intensities range between 150 and 179.
- Gray matter: Intensities range between 180 and 210.

2.2 White Matter Accentuated Transformation

White matter intensities were identified in the range **150-179**. The intensity transformation function preserved these intensities while setting others to 0 for better visualization.

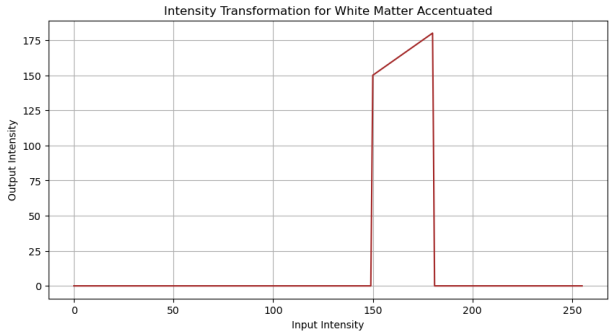


Figure 3: Intensity Transformation for White Matter Accentuated.

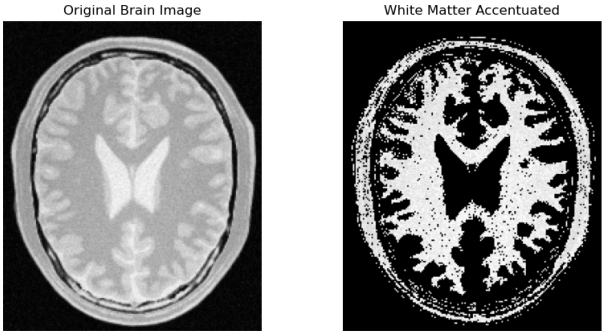


Figure 4: Original Brain Image and White Matter Accentuated Image.

```python
def white_matter_transformation():
    t1 = np.linspace(0, 0, 150).astype('uint8')
    t2 = np.linspace(150, 180, 31).astype('uint8')
    t3 = np.linspace(0, 0, 75).astype('uint8')

    transform = np.concatenate((t1, t2, t3), axis=0).astype('uint8')
    return transform

white_transform = white_matter_transformation()
image_white_matter = cv.LUT(img_orig, white_transform)
```

Listing 2: White Matter Intensity Transformation.

2.3 Gray Matter Accentuated Transformation

Gray matter intensities were identified in the range **180-210**. Similarly, intensities within this range were accentuated and the others set to 0.

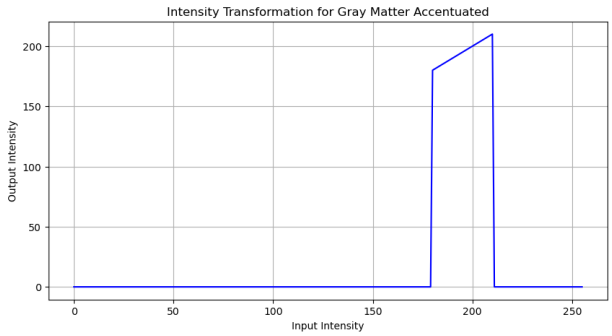


Figure 5: Intensity Transformation for Gray Matter Accentuated.

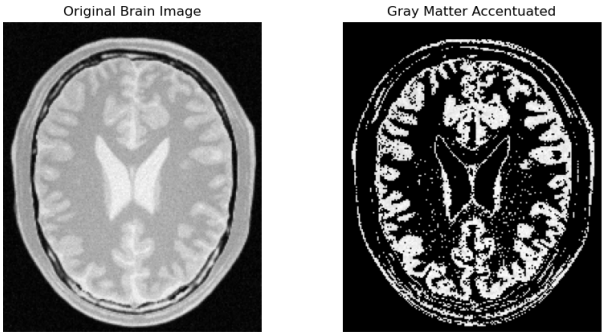


Figure 6: Original Brain Image and Gray Matter Accentuated Image.

```python
def gray_matter_transformation():
    t1 = np.linspace(0, 0, 180).astype('uint8')
    t2 = np.linspace(180, 210, 31).astype('uint8')
    t3 = np.linspace(0, 0, 45).astype('uint8')

    transform = np.concatenate((t1, t2, t3), axis=0).astype('uint8')
    return transform

gray_transform = gray_matter_transformation()
image_gray_matter = cv.LUT(img_orig, gray_transform)
```

Listing 3: Gray Matter Intensity Transformation.

3 Gamma Correction of the L Plane

The transformation uses $\gamma = 0.5$, which enhances darker regions and reduces the intensity of brighter regions.

```python
# Convert to Lab color space and split channels
lab_img = cv.cvtColor(img, cv.COLOR_BGR2Lab)
L, a, b = cv.split(lab_img)

# Gamma correction function
def gamma_correction(image, gamma):
    inv_gamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** inv_gamma) * 255 for i in np.arange(0, 256)]).astype("uint8")
    return cv.LUT(image, table)

# Apply gamma correction to L channel
gamma_value = 0.5
L_gamma_corrected = gamma_correction(L, gamma_value)

# Merge corrected L with a and b channels
lab_img_corrected = cv.merge([L_gamma_corrected, a, b])
img_corrected = cv.cvtColor(lab_img_corrected, cv.COLOR_Lab2BGR)
```

Listing 4: gamma correction on the L channel with $\gamma = 0.5$.

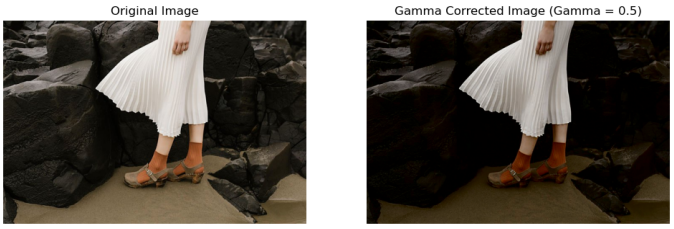


Figure 7: Part (a): Original image and Gamma Corrected Image ($\gamma = 0.5$)

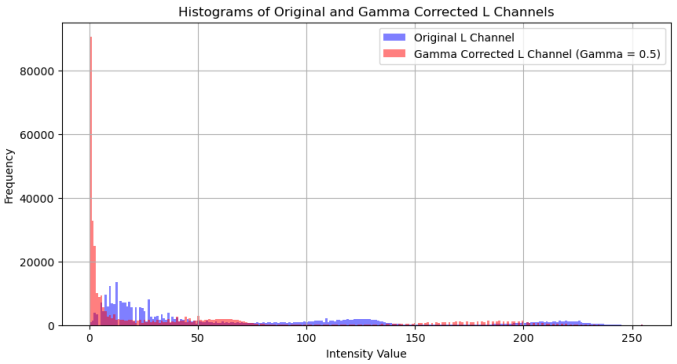


Figure 8: Part (b): Histograms of Original and Gamma Corrected L Channels ($\gamma = 0.5$).

4 Enhancing the Vibrance

To enhance the vibrance of the image, the intensity transformation is applied to the saturation plane in the HSV color space.

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

The value of α was changed in increments of 0.1 (from 0.1 to 1.0) to find the most visually pleasing output.

```python
def vibrance_transform(x, alpha, sigma=70):
    # Apply the vibrance intensity transformation function
    return np.clip(x + alpha * 128 * np.exp(-((x - 128) ** 2) / (2 * sigma ** 2)), 0, 255).astype(np.uint8)

alpha = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

for i in alpha:

    s_vibrance_enhanced = vibrance_transform(s, i)

    hsv_vibrance_enhanced = cv.merge([h, s_vibrance_enhanced, v])

    img_vibrance_enhanced = cv.cvtColor(hsv_vibrance_enhanced, cv.COLOR_HSV2BGR)

    fig, ax = plt.subplots(1, 2, figsize=(12, 6))
    ax[0].imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
    ax[0].set_title('Original Image')
    ax[0].axis('off')

    ax[1].imshow(cv.cvtColor(img_vibrance_enhanced, cv.COLOR_BGR2RGB))
    ax[1].set_title(f'Vibrance-Enhanced Image (Alpha = {i})')
    ax[1].axis('off')

    plt.show()
```

Listing 5: Code for Iterating Different α Values.

Figure 9: Original and Vibrance-Enhanced Images with Different α Values.

After experimenting with different values, $\alpha = 0.5$ was selected as the best value providing a balanced enhancement of vibrance without over-saturating the image.

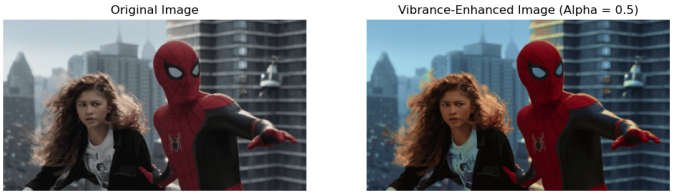


Figure 10: Part (a): Original Image and Vibrance-Enhanced Image ($\alpha = 0.5$).

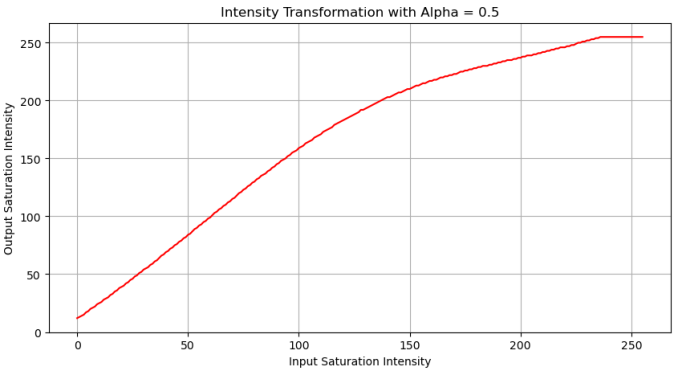


Figure 11: Part (b): Intensity Transformation for Saturation Plane ($\alpha = 0.5$).

5 Histogram Equalization of the Image

The following code used for histogram equalization to enhance the contrast of the grayscale image:

```
M, N = img.shape
hist, bins = np.histogram(img.flatten(), 256, [0, 256])
cdf = hist.cumsum()
cdf_normalized = cdf * (255 / cdf.max())

L = 256
t = np.array([(L-1)/(M*N)*cdf[k] for k in range(256)], dtype=np.uint8)
img_equalized = t[img]
```

Listing 6: gamma correction on the L channel with $\gamma = 0.5$.

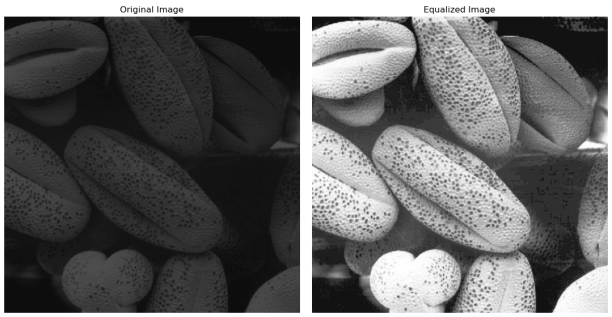


Figure 12: Comparison between the original image and the equalized image.

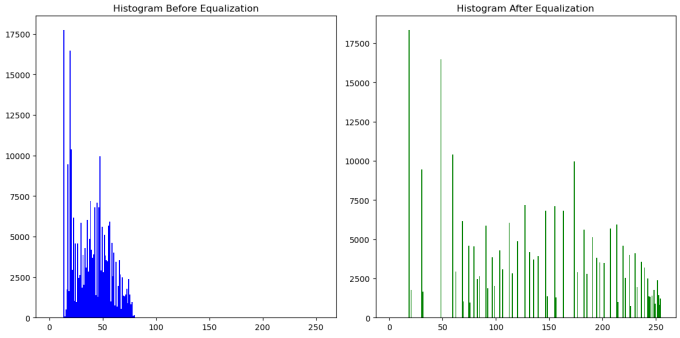


Figure 13: Histograms of the image before and after equalization.

After applying histogram equalization, the contrast of the image is significantly improved, making the finer details in the texture more visible. The redistribution of intensity levels across the image enhances clarity, particularly in the lighter areas, which were initially less visible.

6 Histogram Equalization for the Foreground

The image was converted to the HSV (Hue, Saturation, Value) color space, and the individual planes were split. Each plane were visualized to show how the intensity values vary across hue, saturation, and value channels.

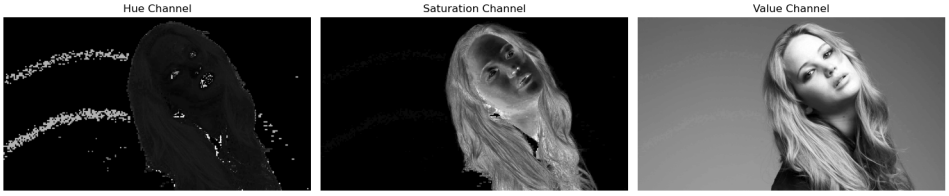


Figure 14: Display of the Hue, Saturation, and Value planes.

```
img_hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
hue, saturation, value = cv.split(img_hsv)
```

The **saturation plane** was thresholded to create a binary mask that isolated the foreground from the background.

```
threshold_value = 13
_, foreground_mask = cv.threshold(saturation, threshold_value, 255, cv.THRESH_BINARY)
```

Figure 15: Saturation channel and extracted binary foreground mask.

The foreground was extracted using the binary mask.

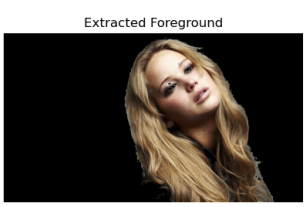
```
foreground = cv.bitwise_and(img, img, mask=foreground_mask)
```

Histogram equalization was applied to the foreground, redistributing the pixel intensities to enhance contrast within the masked region.

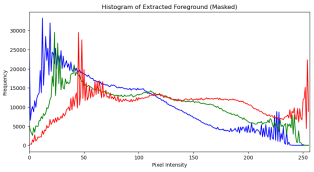
```
def hist_equalized(image, L):
    flattened_image = image.flatten()
    histogram = cv.calcHist([flattened_image], [0], None, [L], [0, L]).flatten()
    cdf = histogram.cumsum()
    M, N, c = image.shape

    hist_equalized = (((L-1)/(M*N))*cdf).astype(np.uint8)
    equalized_image = cv.LUT(image, hist_equalized)

    return equalized_image
```

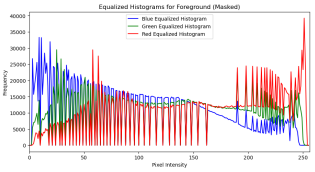


(a) Extracted foreground.



(b) Masked foreground histogram.

(c) Histogram equalized foreground.



(d) Equalized histograms.

Figure 16: Comparison of foreground extraction, masking, and histogram equalization.

The original background was extracted, and the equalized foreground was added back to it, resulting in the final image with an equalized foreground.

```
background = cv.bitwise_and(img, img, mask=cv.bitwise_not(foreground_mask))
final_image = cv.add(equalized_foreground, background)
```

Figure 17: Original image and Final image comparison.

The equalization process has redistributed the pixel intensities, leading to an increase in the number of dark pixels, as seen in the histogram Figure 16d. This can be visually confirmed by inspecting the final output image, where the shadows beneath the face have become darker, emphasizing the details. The image has gained contrast, making the dark areas more prominent while retaining the overall visual balance of the subject against the background.

7 Sobel Filtering

7.1 Using Existing filter2D to Sobel Filter the Image

Applied the Sobel operator to the image using the `cv.filter2D` function with predefined Sobel filters in the 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]])

im_x = cv.filter2D(img, cv.CV_64F, sobel_x)
im_y = cv.filter2D(img, cv.CV_64F, sobel_y)
```

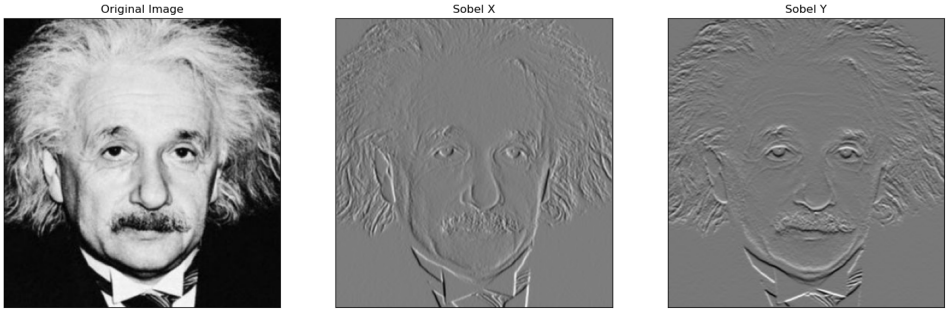


Figure 18: Using filter2D for Sobel Filtering: Original image (left) with Sobel X (center) and Sobel Y (right).

7.2 Writing Custom Code to Apply the Sobel Filter

A custom implementation of the Sobel filter was written using nested loops and convolution operations.

```
for i in range(1, height-1):
    for j in range(1, width-1):
        region = img[i-1:i+2, j-1:j+2]

        gx = np.sum(sobel_x * region)
        gy = np.sum(sobel_y * region)

        sobel_x_result[i, j] = gx
        sobel_y_result[i, j] = gy
```

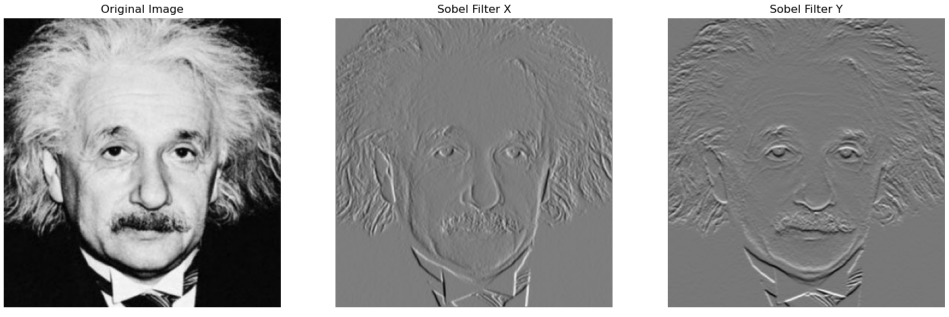


Figure 19: Custom Sobel Filtering: Original image (left) with Sobel X (center) and Sobel Y (right) using custom code.

7.3 Using Separable Property

Performed Sobel filtering by splitting the Sobel operator into two separable 1D filters. This method is computationally efficient as it reduces the number of operations.

```
sobel_col = np.array([[1], [2], [1]])  # Column filter
sobel_row = np.array([[1, 0, -1]])     # Row filter

im_col = cv.filter2D(img, cv.CV_64F, sobel_col)
sobel_filtered = cv.filter2D(im_col, cv.CV_64F, sobel_row)
```

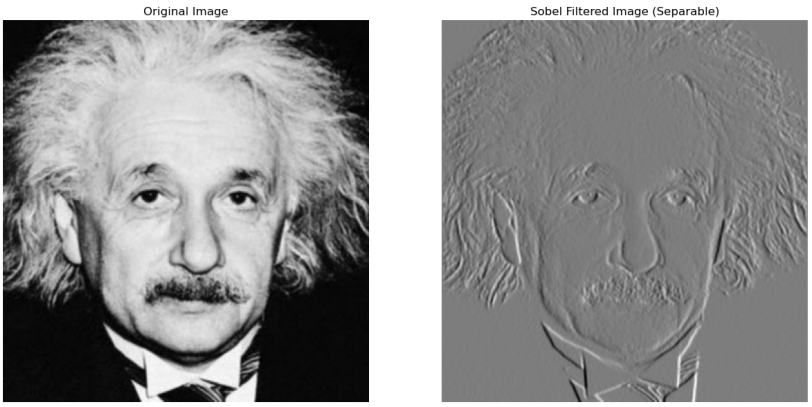


Figure 20: Separable Sobel Filtering: Original image (left) and Sobel-filtered image (right) using separable filters.

8 Zooming Images Using Nearest-Neighbor and Bilinear Interpolation

Nearest-Neighbor Interpolation: This method selects the nearest pixel to estimate the new pixel values, which leads to a blocky appearance when zoomed.

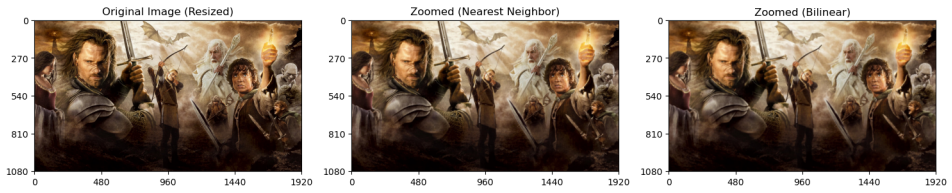
```python
def nearest_neighbor_zoom(image, zoom_factor):
    height, width = image.shape[:2]
    new_height, new_width = int(height * zoom_factor), int(width * zoom_factor)
    return cv.resize(image, (new_width, new_height), interpolation=cv.INTER_NEAREST)
```

Bilinear Interpolation: This approach calculates the pixel values by linear interpolation in two directions (horizontal and vertical), providing a smoother appearance.

```python
def bilinear_zoom(image, zoom_factor):
    height, width = image.shape[:2]
    new_height, new_width = int(height * zoom_factor), int(width * zoom_factor)
    return cv.resize(image, (new_width, new_height), interpolation=cv.INTER_LINEAR)
```

Calculated the sum of squared differences (SSD) between the zoomed image and the original image to evaluate the quality of zooming.

```python
def ssd(original_image, zoomed_image):
    return np.sum(np.square(original_image - zoomed_image)) / original_image.size
```



SSD Results:

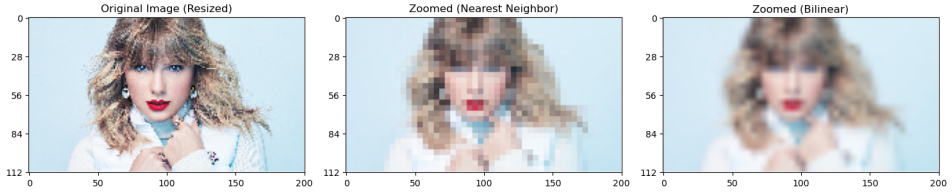
- **Nearest Neighbor:** 31.28431
- **Bilinear Interpolation:** 31.0530

Figure 21: im01 Comparison.

SSD Results:

- **Nearest Neighbor:** 11.9020
- **Bilinear Interpolation:** 10.6829

Figure 22: im02 Comparison.



SSD Results:

- **Nearest Neighbor:** 46.3475
- **Bilinear Interpolation:** 47.5353

Figure 23: Taylor_very_small Comparison.

9 Image Enhancing and Segmentation

9.1 Segmenting the image using grabCut

Used **grabCut** to separate the foreground (flower) from the background

```
mask = np.zeros(img.shape[:2], np.uint8)
bgd_model = np.zeros((1, 65), np.float64)  # Background model
fgd_model = np.zeros((1, 65), np.float64)  # Foreground model
rect = (50, 30, img.shape[1] - 50, img.shape[0] - 100)
cv.grabCut(img, mask, rect, bgd_model, fgd_model, 5, cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
foreground = img * mask2[:, :, np.newaxis]
background = img * (1 - mask2[:, :, np.newaxis])
```

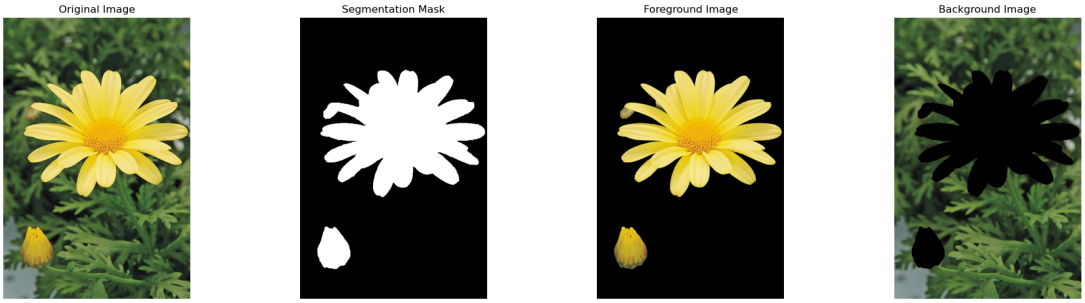


Figure 24: Segmenting the image using grabCut.

9.2 Creating an enhanced image with a blurred background

To enhance the image, applied a Gaussian blur to the background and combined it with the segmented foreground.

```
blurred_background = cv.GaussianBlur(img, (21, 21), 0)
enhanced_image = blurred_background * (1 - mask2[:, :, np.newaxis]) + foreground
```

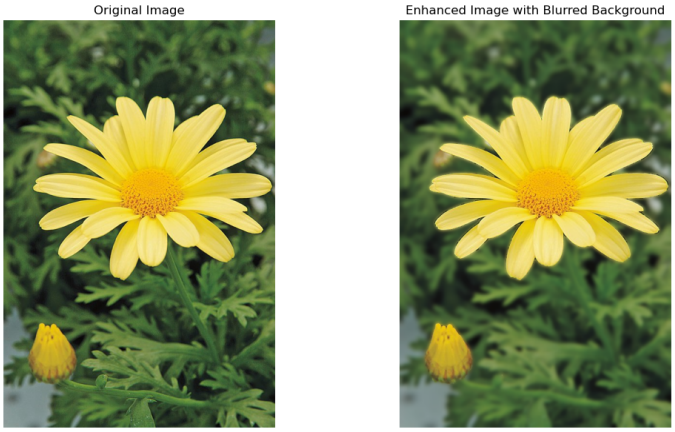


Figure 25: Enhanced image with blurred background.

9.3 Explanation for Darkened Background Near the Flower's Edge in the Enhanced Image

The area near the edge of the flower appears darker in the enhanced image because when the background is blurred, the pixels near the flower's edge are influenced by the black pixels that replace the foreground during the blur process. The blurring effect averages nearby pixels, including these black areas, which results in the darkened appearance around the flower's border.

A GitHub Link

github.com/AchiraHansindu/EN3160-Image-Processing-and-Machine-Vision/Assignment 1