Department of Electronic & Telecommunication Engineering,
University of Moratuwa, Sri Lanka.

Assignment 1
Intensity Transformations and Neighborhood Filtering

Wickramasinghe S.D.
220700T

Submitted in partial fulfillment of the requirements for the module
EN3160 Image Processing and Machine Vision

8/12/2025

Link to the GitHub Repository: https://github.com/Sanugiw/Image-Processing-and-Machine-Vision

1 Intensity Transformation

Intensity transformation was implemented using the following code.

```
# Create a lookup table (LUT) based on the breakpoints
t1 = np.linspace(0, c[0,1], c[0,0] + 1, dtype=np.uint8)
t2 = np.linspace(c[0,1] + 1, c[1,1], c[1,0] - c[0,0], dtype=np.uint8)
t3 = np.linspace(c[2,1] + 1, 255, 255 - c[2,0], dtype=np.uint8)

# Concatenate segments to form the complete LUT
transform = np.concatenate((t1, t2), axis=0)
transform = np.concatenate((transform, t3), axis=0).astype(np.uint8)
```

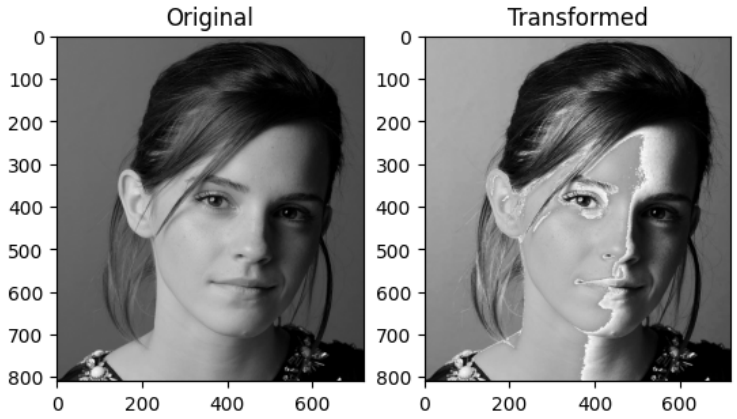


Figure 1: Comparison of Results

2 Intensity Transformation to Enhance White and Gray Matter

White matter was enhanced using the following transformation.

```
# Transformation for white matter intensities
t_white = np.zeros(256, dtype='uint8')
t_white[141:181] = np.linspace(141, 180, 40).astype('uint8')
```

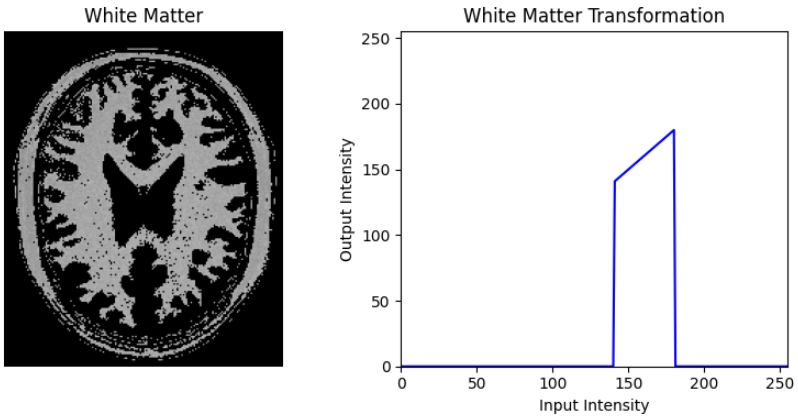


Figure 2: Result of Enhancing White Matter

The transformation has linearly intensified pixel intensities corresponding to the typical brightness of white matter in the brain proton density image.

Gray matter was enhanced using the following transformation.

```
# Transformation for gray matter intensities
t_gray = np.zeros(256, dtype='uint8')
t_gray[181:221] = np.linspace(181, 220, 40).astype('uint8')
```

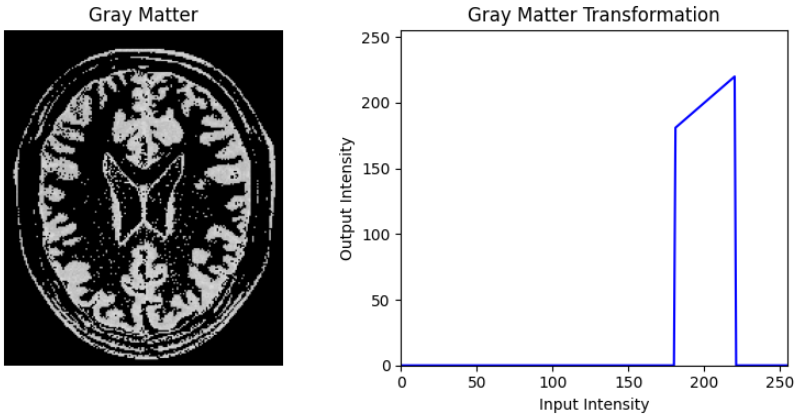


Figure 3: Result of Enhancing Gray Matter

The transformation has linearly intensified pixel intensities corresponding to the typical brightness of gray matter in the brain proton density image.

3 Gamma Correction

The image was split into L, a, and b color spaces, and gamma correction was applied to the L plane as follows.

```
# Split into H, S, V planes
h, s, v = cv.split(image4_hsv)

# build LUT for the given transformation
a = 0.6
sigma = 70.0
x = np.arange(0, 256)
f_x = np.minimum(x + a * 128 * np.exp(-((x - 128)**2) / (2 * sigma**2)), 255).astype('uint8')

# Apply the intensity transformation to the saturation plane using LUT
s_transformed = cv.LUT(s, f_x)
```

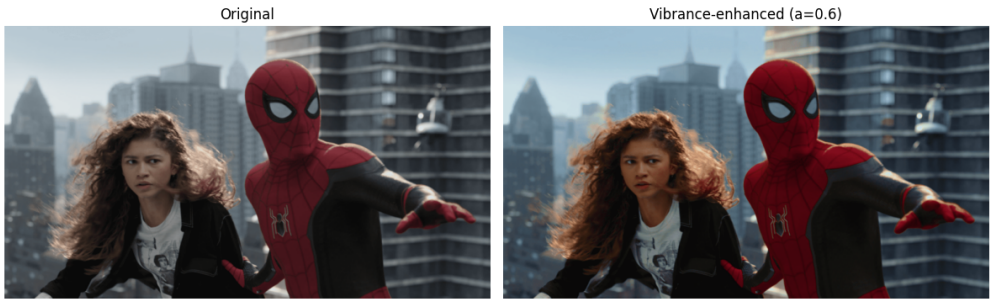


Figure 4: Results after Applying Gamma Correction

This has performed gamma correction on the L channel of an image in the L*a*b color space to enhance its brightness and contrast. The comparison between the original RGB image and the gamma-corrected version clearly shows improved visibility in shadowed areas and highlights.

4 Intensity Transformation to Increase Vibrance

```
h, s, v = cv.split(image4_hsv)

# f(x) = min( x + a * 128 * exp(-(x-128)^2 / (2*sigma^2)), 255 )
a = 0.6
sigma = 70.0
x = np.arange(0, 256)
f_x = np.minimum(x + a * 128 * np.exp(-((x - 128)**2) / (2 * sigma**2)), 255).astype('uint8')

# Apply the intensity transformation to the saturation plane
s_transformed = cv.LUT(s, f_x)
```

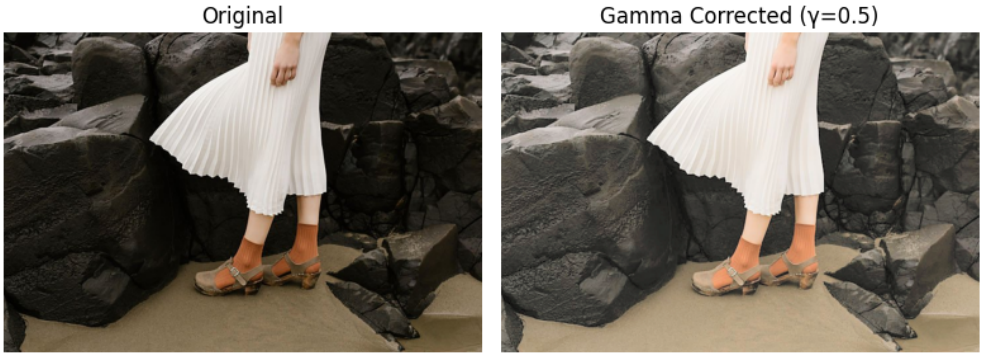


Figure 5: Results after Applying Intensity Transformation

This has boosted the saturation values around mid-range levels, instead of uniformly increasing saturation. It has applied a Gaussian-shaped intensity transformation centered on medium saturation values, avoiding over-saturation of highly saturated areas and under-saturation of low-saturated ones.

5 Histogram Equalization

```python
def histogram_equalization(f):
L = 256
M, N = f.shape

# Compute histogram
hist = cv.calcHist([f], [0], None, [L], [0, L])
hist = hist.flatten()

# Compute cumulative distribution function (CDF)
cdf = hist.cumsum()

# Normalize CDF
cdf_normalized = (cdf - cdf.min()) * (L - 1) / (cdf.max() - cdf.min())
cdf_normalized = cdf_normalized.astype('uint8')

# Map original image pixels through the normalized CDF
equalized_img = cdf_normalized[f]

return equalized_img, hist
```

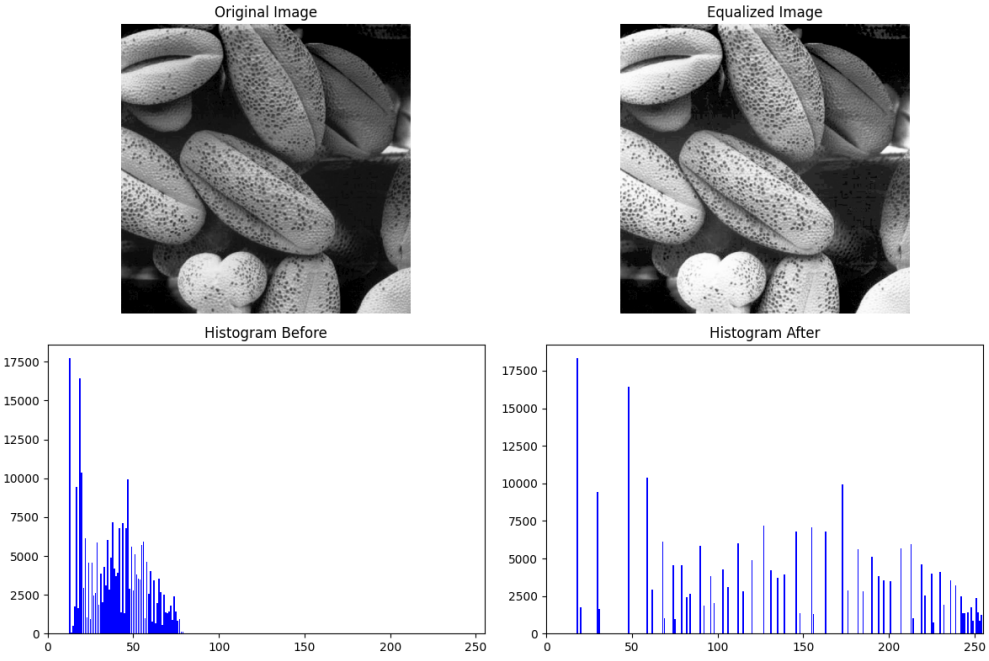


Figure 6: Results after Applying Histogram Equalization

The histogram equalization applied to the image has enhanced its contrast by redistributing pixel intensity values more uniformly across the available range. Initially, the original image's histogram values pixel values clustered within a narrow intensity band. After equalization, the histogram spreads out more evenly across the full 0–255 intensity range. This has resulted in an equalized image with brighter highlights and deeper shadows, revealing finer details than the original image.

6 Histogram Equalization to the Foreground

After splitting the image into hue, saturation, and value planes and displaying the results, it is observed that the foreground is more distinguishable in the saturation plane. The foreground is extracted by applying a binary mask created by thresholding the Saturation plane. Then the histogram equalization is applied to the extracted foreground.

```
# Threshold appropriate plane to get foreground mask using s plane
_, mask = cv.threshold(s, 15, 255, cv.THRESH_BINARY)

# Obtain foreground using bitwise_and, compute histogram
foreground = cv.bitwise_and(s, s, mask=mask)

hist_foreground = cv.calcHist([foreground], [0], mask, [256], [0,256])
hist_foreground = hist_foreground.flatten()

# Compute the cumulative sum of the histogram (CDF)
cdf_foreground = np.cumsum(hist_foreground)

# Avoid zeros in CDF and normalize to [0, 255]
cdf_nonzero = cdf_foreground[cdf_foreground > 0]
cdf_min = cdf_nonzero.min()
cdf_max = cdf_foreground[-1]

# Calculate transformation function T(r_k) using vectorized formula
T = np.clip(((cdf_foreground - cdf_min) / (cdf_max - cdf_min) * 255), 0, 255).astype('uint8')

# Apply equalization only to foreground pixels inside the mask
equalized_foreground = foreground.copy()
equalized_foreground[mask == 255] = T[foreground[mask == 255]]

# Extract background and add with equalized foreground
background = cv.bitwise_and(s, s, mask=cv.bitwise_not(mask))

# Combine background and equalized foreground
result_s = cv.add(background, equalized_foreground)
```

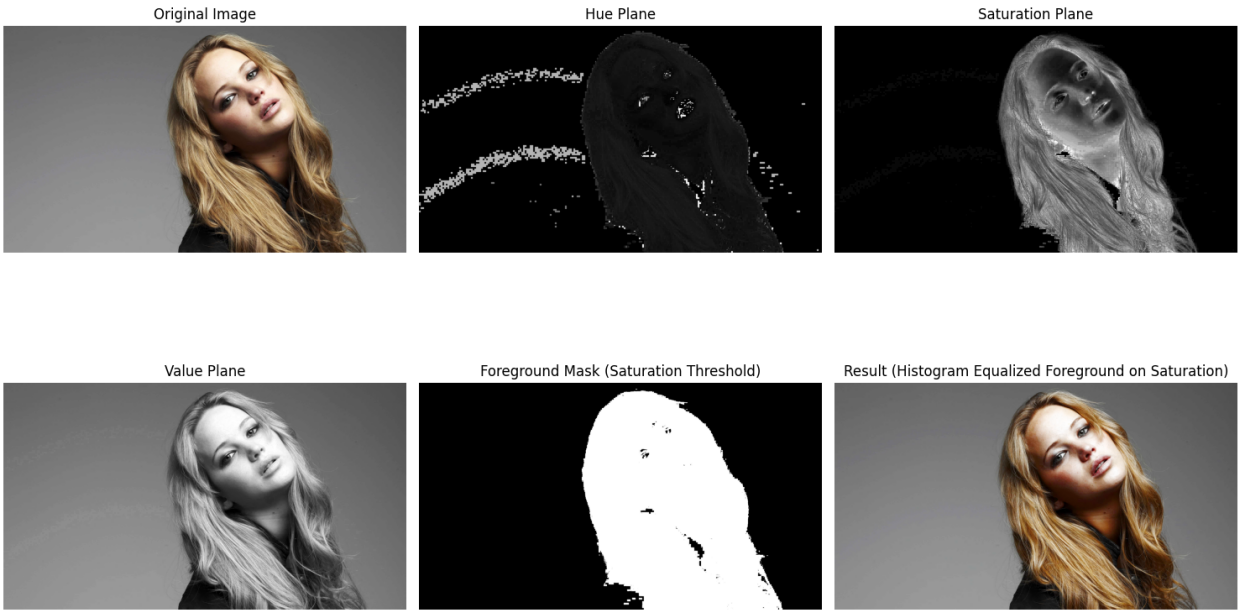


Figure 7: Results after Applying the Mask

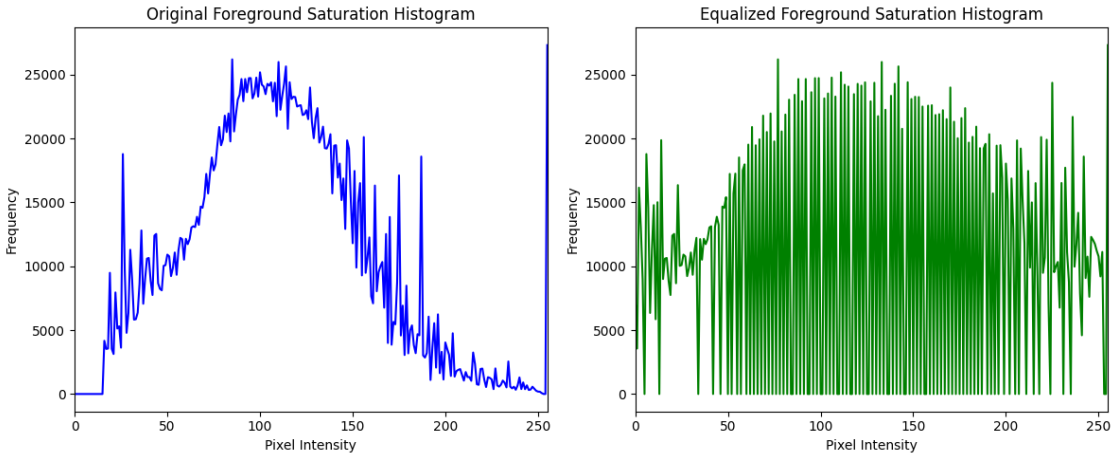


Figure 8: Histograms before and after Histogram Equalization

The results demonstrate that selectively applying histogram equalization to the saturation channel of the foreground significantly enhances color vividness without altering brightness or background colors. The foreground mask isolates the more colorful areas, and after equalization, the saturation values are more evenly distributed across the intensity range. This has led to more vibrant colors in the subject with the same background.

7 Filtering with the Sobel Operator

The existing filter2D function was used to apply the Sobel filter.

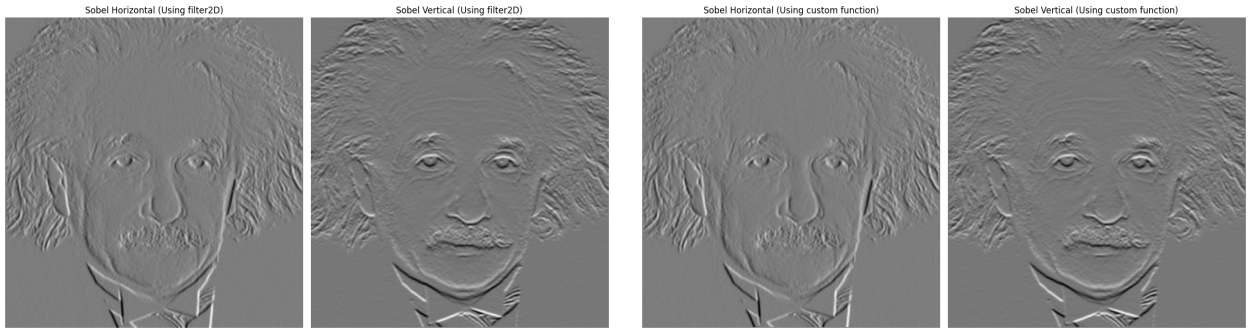
```
# Sobel kernels (horizontal and vertical)
sobel_x = np.array([[1, 0, -1],[2, 0, -2],[1, 0, -1]], dtype=np.float32)
sobel_y = np.array([[1, 2, 1],[0, 0, 0],[-1, -2, -1]], dtype=np.float32)

# Apply the Sobel filter in X and Y directions
sobel_x_filtered = cv.filter2D(image7, cv.CV_64F, sobel_x)
sobel_y_filtered = cv.filter2D(image7, cv.CV_64F, sobel_y)
```

A custom function was implemented to apply the Sobel filter.

```
def apply_filter(image, filter):
    [rows, columns] = np.shape(image)
    filtered_image = np.zeros(shape=(rows, columns))

    for i in range(rows - 2):
        for j in range(columns - 2):
            value = np.sum(np.multiply(filter, image[i:i + 3, j:j + 3]))
            filtered_image[i + 1, j + 1] = value
    return filtered_image
```



(a) Using filter2D

(b) Using Custom Sobel Filter

Figure 9: Results after Using filter2D and Custom Sobel Filter

Then, the convolution property is used for the Sobel filter.

```
# Sobel X and Y filters separated
sobel_x_vertical = np.array([[1], [2], [1]])
```

```python
sobel_x_horizontal = np.array([[1, 0, -1]])
sobel_y_vertical = np.array([[1], [0], [-1]])
sobel_y_horizontal = np.array([[1, 2, 1]])

# Apply Sobel X and Y
x_mid = cv.filter2D(image7, cv.CV_64F, sobel_x_horizontal)
x_filtered_image = cv.filter2D(x_mid, cv.CV_64F, sobel_x_vertical)
y_mid = cv.filter2D(image7, cv.CV_64F, sobel_y_vertical)
y_filtered_image = cv.filter2D(y_mid, cv.CV_64F, sobel_y_horizontal)
```

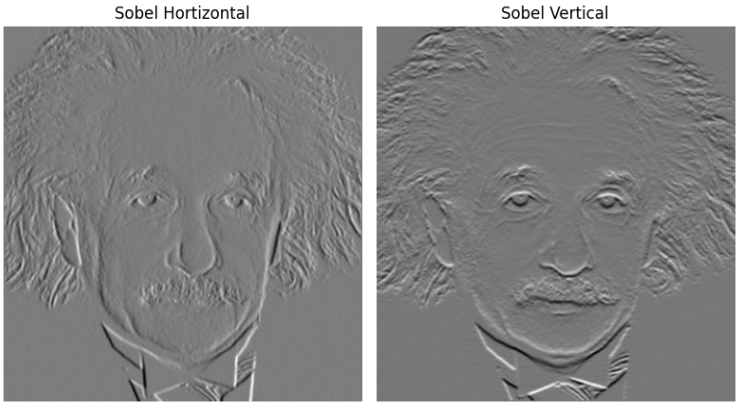


Figure 10: Results after Using the Convolution Property

8 Image Zooming

Images were zoomed by a factor of 4 using nearest neighbor and bilinear transformation. The results were compared by computing the normalized sum of squared difference (SSD).

```python
# Zoom function
def zoom(img, technique, scale=4):
    if technique == 'nn':  # Nearest Neighbor
        return cv.resize(img, None, fx=scale, fy=scale, interpolation=cv.INTER_NEAREST)
    elif technique == 'bilinear':  # Bilinear Interpolation
        return cv.resize(img, None, fx=scale, fy=scale, interpolation=cv.INTER_LINEAR)
    else:
        raise ValueError("Technique must be 'nn' or 'bilinear'.")

# Normalized SSD calculation
def norm_SSD(img1, img2):
    if img1.shape != img2.shape:
        raise ValueError("Images must have the same dimensions.")
    return np.sum((img1.astype(np.float32) - img2.astype(np.float32)) ** 2) / img1.size
```

(a) Image 1 (b) Image 2

Figure 11: Zoomed Images of Image 1 and Image 2

According to the results, bilinear interpolation outperforms nearest neighbor by producing smoother images with fewer jagged edges. It is confirmed through a lower SSD value, indicating higher reconstruction accuracy.

9 Image Enhancing

The following code applies GrabCut for 5 iterations, using a rectangle as the initial guess for the foreground, and then extracts the foreground.

```python
# Initialize mask, models, and a rectangle
mask = np.zeros(image9.shape[:2], np.uint8)
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)
rect = (50, 100, 550, 490)

# Apply GrabCut
cv.grabCut(image9, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)

# Create a binary mask for the foreground
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')

# Foreground and background extraction
foreground = image9 * mask2[:, :, np.newaxis]
background = image9 * (1 - mask2[:, :, np.newaxis])
```

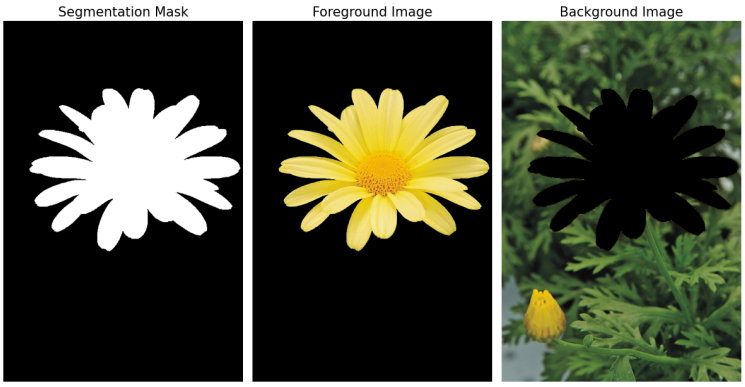


Figure 12: Image After GrabCut Segmentation and Enhancement

The GrabCut segmentation successfully isolated the daisy flower from its background, producing a clear binary mask. The foreground image retains only the flower, while the background image shows the removed surroundings, demonstrating effective separation of object and background regions.

The following code takes the segmented foreground of the image and applies a Gaussian blur to the original image to create a blurred background.

```python
# Blur the background
blurred_background = cv.GaussianBlur(image9, (25, 25), 0)

# Combine blurred background with sharp foreground
enhanced_image = blurred_background * (1 - mask2[:, :, np.newaxis]) + foreground
```

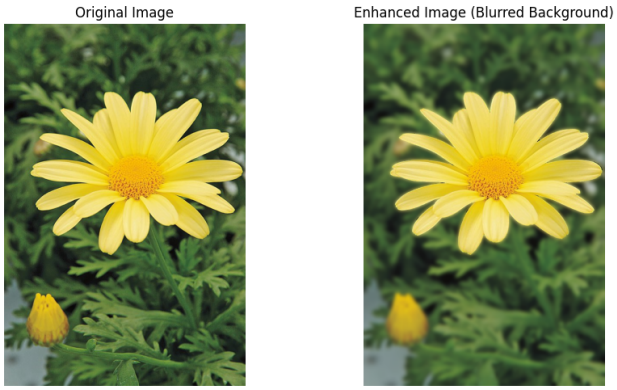


Figure 13: Image with Blurred Background

Background just beyond the edge of the flower is dark in the enhanced image.
That dark edge appears because GrabCut's segmentation mask wasn't perfectly precise at the flower's edges. In the mask, some edge pixels were mistakenly classified as background, so when the background was blurred and blended, those pixels got darkened. This effect is especially noticeable where the background color is darker than the flower's edge, creating a faint outline.