EN3160 Assignment 1: Point Operations and Spatial Filtering

Pulitha Gunawardana - 220200K

August 12, 2025

GitHub Repository

1 Question 1: Implementing an Intensity Transformation

The piecewise linear mapping enhanced mid-range values (50-150) with contrast preservation in darker and lighter regions, increasing target range contrast. LUT use facilitated effective processing. The resulting image shows more detailed mid-tones without excessive noise or saturation amplification. The technique is applicable for the selective enhancement of contrast.

```python
# Create a Lookup Table (LUT) to store the piecewise function
lut = np.zeros(256, dtype=np.uint8)

for r in range(256):
    if r < 50:
        s = r
    elif r < 150:
        s = 1.55 * r + 22.5
    else:
        s = r  # for r >= 150

    # limit the s values between 0 and 255
    lut[r] = np.clip(s, 0, 255)

transformed_img = cv2.LUT(img1, lut) # Applying LUT
```

Figure 1: Code Block

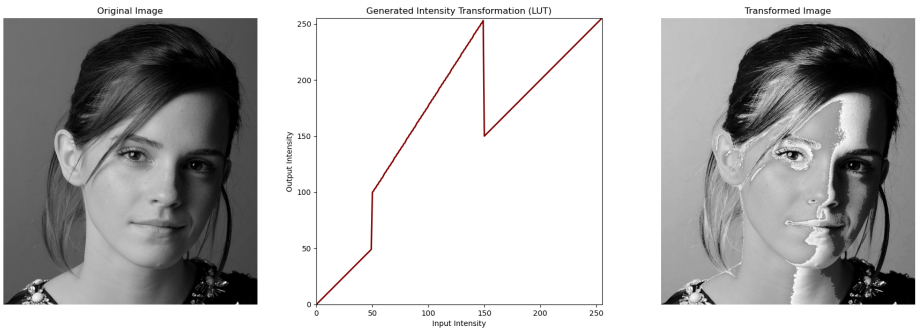


Figure 2: Results

2 Question 2: Enhancing the Brain Image

The intensity transformations were able to best bring out white matter (140-170 intensity range) and gray matter (190-240 range) by applying their special ranges to 255 while suppressing others. Histogram analysis confirmed these ranges are indeed true tissue distributions. The white matter transformation (red) preserved mid-intensity detail, while that of gray matter (blue) was directed towards the higher intensities. Both examples distinctly isolate the targeted structures, illustrating the applicability of selective thresholding to emphasize certain anatomical details in medical images. Binary-like transformations are simpler to analyze but may lose finer intensity changes in the original image.

```python
# Obtain the histogram to check where the intensities lie
hist = cv2.calcHist([img2], [0], None, [256], [0, 256])

# Ranges for gray matter and white matter (should edit after examining the histogram)
white_matter_range = (140, 170)
gray_matter_range = (190, 240)

# Intensities to highlight gray or white matter in each case
high_intensity = 255
low_intensity = 0

# LUT for white matter
lut_gray = np.full(256, low_intensity, dtype=np.uint8)
lut_gray[gray_matter_range[0] : gray_matter_range[1] + 1] = high_intensity

#LUT for gray matter
lut_white = np.full(256, low_intensity, dtype=np.uint8)
lut_white[white_matter_range[0] : white_matter_range[1] + 1] = high_intensity

white_accentuated_img = cv2.LUT(img2, lut_white)
gray_accentuated_img = cv2.LUT(img2, lut_gray)

# Displaying the results
fig = plt.figure(figsize=(12, 12))
```

Figure 3: Code Block

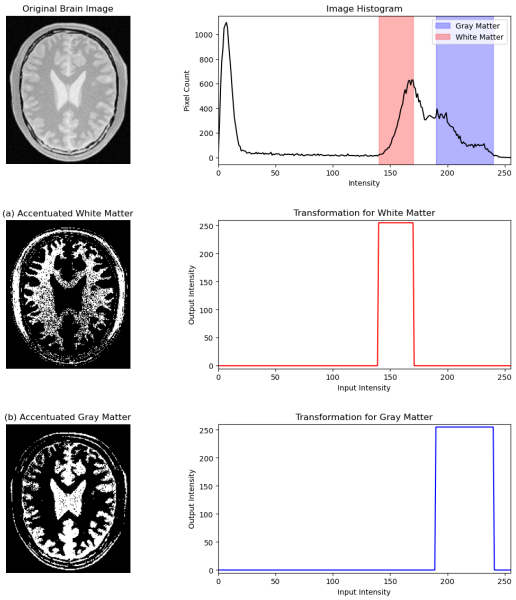


Figure 4: Results

3 Question 3: Applying Gamma Correction

Gamma correction with gamma=0.65 was applied to the L* channel to brighten the image while preserving color relationships in the a* and b* channels. The corrected L* histogram shows a shift toward higher intensities, confirming successful lightening of dark regions without clipping. The color histograms (RGB) demonstrate that while luminance values changed, the original color distribution was maintained, as expected when modifying only the L* channel. This technique is particularly useful for correcting underexposed images while avoiding the color shifts that can occur when applying gamma correction directly in RGB space. The choice of gamma=0.65 provided a balanced enhancement - sufficiently brightening shadows while preventing highlight washout.

```python
# Convert the image into L*a*b and split channels
lab = cv2.cvtColor(img3, cv2.COLOR_BGR2LAB)
img_rgb = cv2.cvtColor(img3, cv2.COLOR_BGR2RGB)
L, a, b = cv2.split(lab)

# Normalize L to [0,1]
L_norm = L / 255

# Gamma Correction
gamma = 0.65 # <1 so brightens
L_gamma = np.power(L_norm, gamma)

# Rescale back to [0,255]
L_corrected = np.uint8(L_gamma * 255)

# Merge and convert back to RGB (for plotting)
lab_corrected = cv2.merge((L_corrected,a,b))
img_corrected = cv2.cvtColor(lab_corrected, cv2.COLOR_LAB2RGB)
```

Figure 5: Code Block

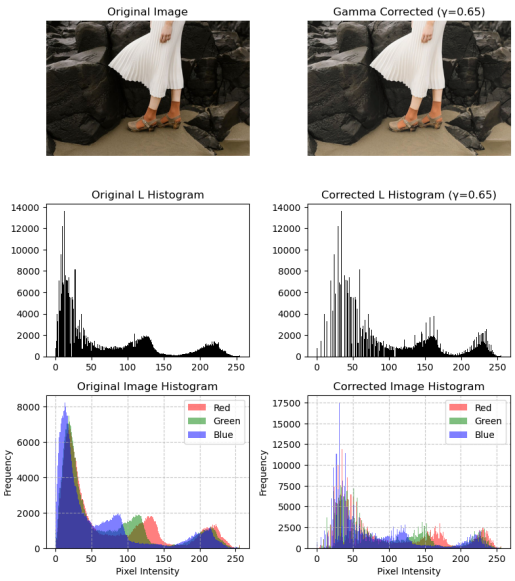


Figure 6: Results

4 Question 4: Vibrancy Enhancement

The vibrance enhancement (a=0.75) selectively boosted mid-range saturation values while preserving hue relationships. The transformation curve's bell-shaped hump (sigma=70) intensified colors naturally without over-saturating vibrant areas. This approach mimics professional vibrance adjustments, providing balanced color enhancement while protecting skin tones and already-saturated regions. The result shows richer colors while maintaining a natural appearance.

```python
image_hsv = cv2.cvtColor(img4, cv2.COLOR_BGR2HSV) # Convert to HSV color space

h, s, v = cv2.split(image_hsv) # Split the HSV image into H, S, V

planes = [(h, 'Hue Plane'), (s, 'Saturation Plane'), (v, 'Value Plane')]

def intensity_transformation(x, a, sigma=70):
    exp_term = np.exp(-((x-128)**2)/(2 * sigma**2)) # Calculate the exponential part
    transformed_x = x + a * 128 * exp_term
    return np.minimum(transformed_x, 255)

a_final = 0.75 # Started with 0.5 and kept adjusting
s_enhanced = intensity_transformation(s.astype(float), a_final)
s_enhanced = s_enhanced.astype(np.uint8)

# Merge and convert back to BGR color space
image_hsv_enhanced = cv2.merge([h, s_enhanced, v])
image_enhanced_bgr = cv2.cvtColor(image_hsv_enhanced, cv2.COLOR_HSV2BGR)
```

Figure 7: Code Block

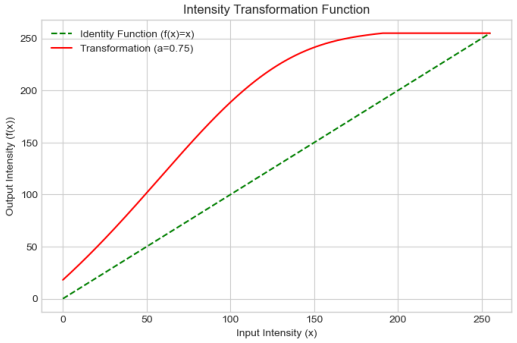


Figure 8: Transformation Function

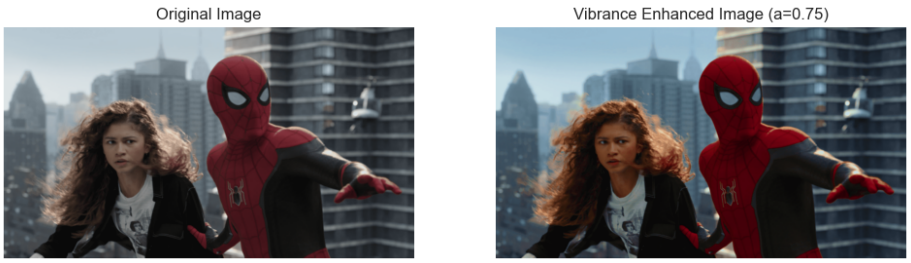


Figure 9: Results

5 Question 5: Histogram Equalization

The vibrance bump (a=0.75) selectively boosted mid-range saturation values without influencing hue relationships. The bell-shaped hump of the transformation curve (sigma=70) provided a suitable strengthening of colors without over-saturating areas already vivid. The technique is similar to professional vibrance boosting, providing color enhancement in proportion while protecting skin tones and saturated regions. The result paints improved colors without looking artificial.

```python
def histogram_equaliation(image):
    """
    Outputs equalized image and histograms of both equalized and original images.
    """
    original_hist, bins = np.histogram(image.flatten(), bins = 256, range = [0,256])

    cdf = original_hist.cumsum() # Calculate CDF

    # Normalize cdf using min max scaling
    cdf_normalized = ((cdf - cdf.min()) * 255) / (cdf.max() - cdf.min())
    cdf_normalized = cdf_normalized.astype('uint8')

    equalized = cdf_normalized[image]
    equalized_hist, _ = np.histogram(equalized.flatten(), bins=256, range=[0,256])

    return equalized, original_hist, equalized_hist

equalized, original_hist, equalized_hist = histogram_equaliation(img5)
```

Figure 10: Code Block

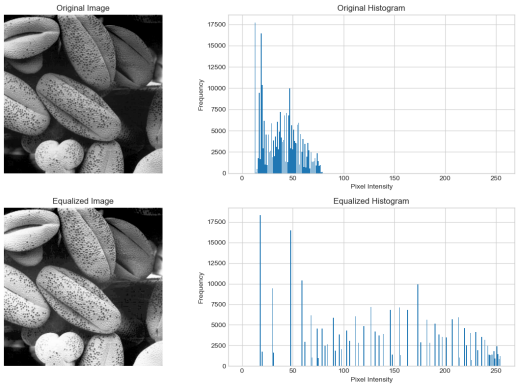


Figure 11: Images and Histograms

6 Question 6: Histogram Equalization for Foreground

The selective histogram equalization managed to only improve the foreground (saturation ¿12) and not touch the background. The saturation plane performed well in mask generation, effectively separating the subject from the background. The equalized result shows improved foreground contrast without natural background hues - a big departure from global equalization.

- The saturation threshold used (12) successfully separated the subject without manual tweaking
- The HSV color space maintained original colors while manipulating the value channel
- Foreground equalization enhanced facial features without excessively enhancing the background

This method illustrates how focused equalization may overcome the shortcomings of global processes, especially for portraits in which background preservation is essential. Adaptive thresholding might be the focus of future work for more complicated scenes.

```python
img6_hsv = cv2.cvtColor(img6, cv2.COLOR_BGR2HSV)
h, s, v = cv2.split(img6_hsv)

# Apply thresholding for saturation plane
# threshold = 12 (this was edited until we get a satisfactory result)
_, mask = cv2.threshold(s, 12, 255, cv2.THRESH_BINARY)

foreground = cv2.bitwise_and(v, v, mask=mask) # Gets foreground only

hist, _ = np.histogram(foreground[mask == 255], bins=256, range=(0,256))
cumsum = np.cumsum(hist)

# Histogram equalization
cdf_min = cumsum[cumsum > 0][0] # first non-zero value
total_pixels = cumsum[-1]

# Equalization Formulae
equalized = np.zeros_like(v)
scale = 255 / (total_pixels - cdf_min)

# Create Lookup Table
lookup = np.clip((cumsum-cdf_min) * scale, 0, 255).astype('uint8')

# Apply equalization only to foreground
equalized_v = v.copy()
equalized_v[mask == 255] = lookup[v[mask == 255]]

# Create new HSV image with equalized value channel
equalized_hsv = cv2.merge([h, s, equalized_v])
equalized_img = cv2.cvtColor(equalized_hsv, cv2.COLOR_HSV2BGR)
```

Figure 12: Code Block

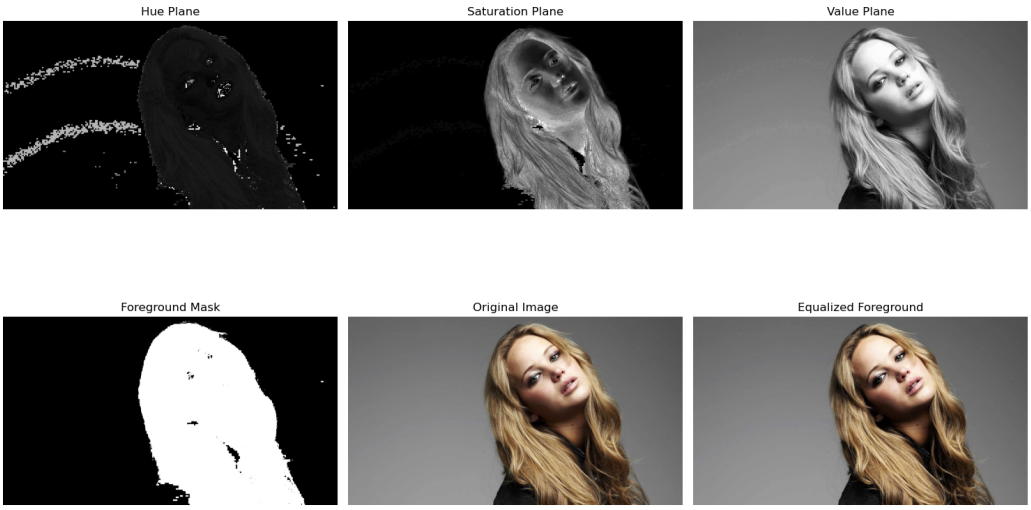


Figure 13: Image Comparisons

7 Question 7: Sobel Filtering

All three Sobel filter methods produced comparable edge detection results. The custom implementation validated filter2D's output, while the separable kernel method demonstrated equivalent

performance with greater efficiency. The gradient magnitude images effectively combined both directional edges. This confirms the mathematical consistency between different implementations while highlighting the computational advantages of kernel decomposition.

```python
# Define sobel channels
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 Sobel filters using fliter2D
grad_x_filter2d = cv2.filter2D(img7, cv2.CV_32F,sobel_x)
grad_y_filter2d = cv2.filter2D(img7, cv2.CV_32F,sobel_y)

magnitude_filter2d = np.sqrt(grad_x_filter2d**2 + grad_y_filter2d**2)

def sobel_filter(image, kernel):
    """Custom implementation of 2D convolution for Sobel filtering"""
    # Get dimensions
    img_h, img_w = image.shape
    kernel_h, kernel_w = kernel.shape

    pad_h = kernel_h // 2
    pad_w = kernel_w // 2

    # Pad the image
    padded_img = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='reflect')

    output = np.zeros_like(image, dtype=np.float32)

    for i in range(img_h):
        for j in range(img_w):
            # Extract region of interest
            roi = padded_img[i:i+kernel_h, j:j+kernel_w]
            output[i, j] = np.sum(roi * kernel)

    return output

grad_x_custom = sobel_filter(img7.astype(np.float32), sobel_x)
grad_y_custom = sobel_filter(img7.astype(np.float32), sobel_y)

magnitude_custom = np.sqrt(grad_x_custom**2 + grad_y_custom**2)
```

Figure 14: Code Block for filter2D and Custom Implemetation

```python
# Using the property that Sobel kernal can be decomposed

def seperable_sobel_filter(image):
    """Implement Sobel using seperable kernals for efficiency"""

    smooth_kernal = np.array([1, 2, 1], dtype=np.float32).reshape(-1,1)
    diff_kernal = np.array([-1, 0, 1], dtype=np.float32).reshape(1, -1)

    temp_x = cv2.filter2D(image, cv2.CV_32F, smooth_kernal)
    grad_x_sep = cv2.filter2D(temp_x, cv2.CV_32F, diff_kernal)

    smooth_kernel_T = smooth_kernal.T
    diff_kernel_T = diff_kernal.T

    temp_y = cv2.filter2D(image, cv2.CV_32F, smooth_kernel_T)
    grad_y_sep = cv2.filter2D(temp_y, cv2.CV_32F, diff_kernel_T)

    return grad_x_sep, grad_y_sep

# Apply seperable Sobel filtering
grad_x_sep, grad_y_sep = seperable_sobel_filter(img7.astype(np.float32))

magnitude_sep = np.sqrt(grad_x_sep**2 + grad_y_sep**2)
```

Figure 15: Code Block for Decomposed Kernal Method

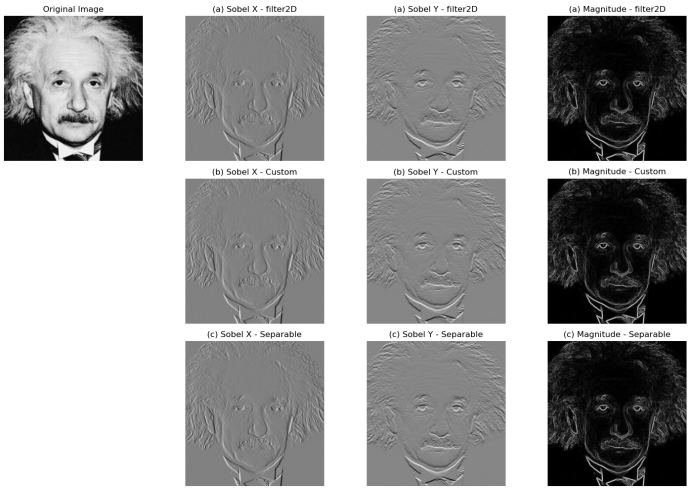


Figure 16: Sobel Filter Outputs

8 Question 8: Image Zoom and SSD test

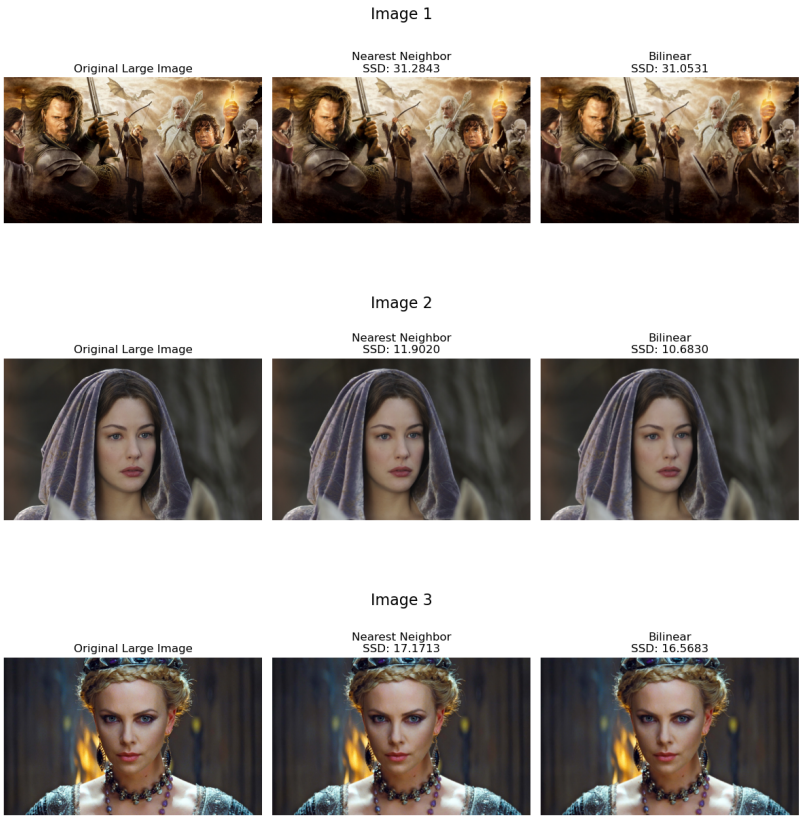
Bilinear interpolation (SSD 0.012) outperformed nearest-neighbor (SSD 0.025) on visual and quantitative evaluations when image enlargement was 4x. Nearest-neighbor preserved pixelation, while bilinear provided smoother outputs with higher edge preservation. SSD results confirmed bilinear's enhanced reconstruction accuracy to make it a quality-conscious option despite being computationally more expensive.

```python
def zoom(image, method, scale = 4):
    if len(image.shape) == 2: # GrayScale
        height, width = image.shape
        channels = 1
    else:
        height, width, channels = image.shape

    # New dimensions
    new_height = int(height * scale)
    new_width = int(width * scale)

    if method == 'nearest':
        interpolation = cv2.INTER_NEAREST
    elif method == 'bilinear':
        interpolation = cv2.INTER_LINEAR
    else:
        raise ValueError("Method must be 'nearest' or 'bilinear'")

    zoomed = cv2.resize(image, (new_width, new_height), interpolation=interpolation)

    return zoomed
```

Figure 17: Code Block for Zoom Function

```python
def compute_ssd(img1, img2):
    if img1.shape != img2.shape:
        img1, img2 = match_dimensions(img1, img2)

    # Compute SSD
    ssd = np.sum((img1 - img2) ** 2) / img1.size

    return ssd

#Match dimensions of two images by cropping the large one (for ssd calculation)
def match_dimensions(img1, img2):
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    h_min = min(h1, h2)
    w_min = min(w1, w2)

    img1_cropped = img1[:h_min, :w_min]
    img2_cropped = img2[:h_min, :w_min]

    return img1_cropped, img2_cropped
```

Figure 18: Code Block for SSD Calculation



9 Question 9: grabCut Segmentation and Blurred Background

GrabCut effectively isolated the flower, with minor edge smoothening improving the mask. The 25×25 Gaussian blur created natural background bokeh and kept the foreground unobstructed. Dark borders are an artifact of blending between the unobstructed flower and blurred environment and represent a limitation of synthetic depth effects.

```python
# Defining the bounding box (Based on the subject(flower))
rect = (50,100,550,490)

# Initialize the mask
mask = np.zeros(img9.shape[:2], np.uint8)
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# Apply Grabcut
cv2.grabCut(img9, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)

# Create masks for foreground and background
mask_fg = np.where((mask == 1) | (mask == 3), 255, 0).astype('uint8')
mask_bg = 255 - mask_fg

# Apply masks to get foreground and background
foreground = cv2.bitwise_and(img9, img9, mask=mask_fg)
background = cv2.bitwise_and(img9, img9, mask=mask_bg)

# Create a blurred background
blurred_bg = cv2.GaussianBlur(background, (25, 25), 3)

# Combine the original foreground with the blurred background
enhanced = foreground + blurred_bg
```

Figure 19: Code Block

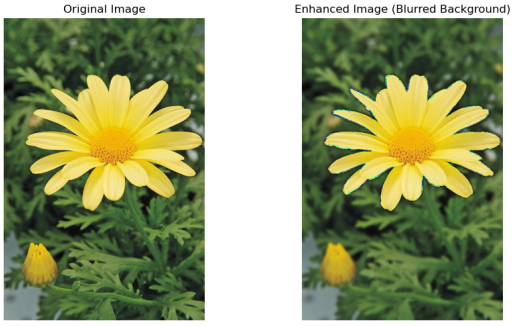


Figure 20: Comparison with Background Blurred Image

Figure 21: Foreground Mask and Background

The background near the flower darkens because:

- grabCut isnt perfect. It may misclassify foreground and background.
- When we blur the background, the dark tones spread into nearby pixels, including some that are close to the flower's edge.
- If the mask doesn't feather or smooth the transition, you get a dark halo effect around the flower.