Assignment 01 - Intensity Transformations and Neighborhood Filtering

EN3160: Image Processing And Machine Vision

Index: 210349N - Madhushan I.D.

GitHub Link for Code: Link to my GitHub repository

Question 1: Intensity Transformation

Resulting Image:

Figure 1: Result of the intensity transformation applied on the image.

Mid-range intensities(50-150) are enhanced, while the low (0-50) and high (150-255) intensities remain unchanged. Results in brighter mid-tone areas such as some parts of the right cheek and forehead, while darker and lighter regions such as shadows and highlights stay the same.

Question 2: Enhancing White and Gray Matter

Results:

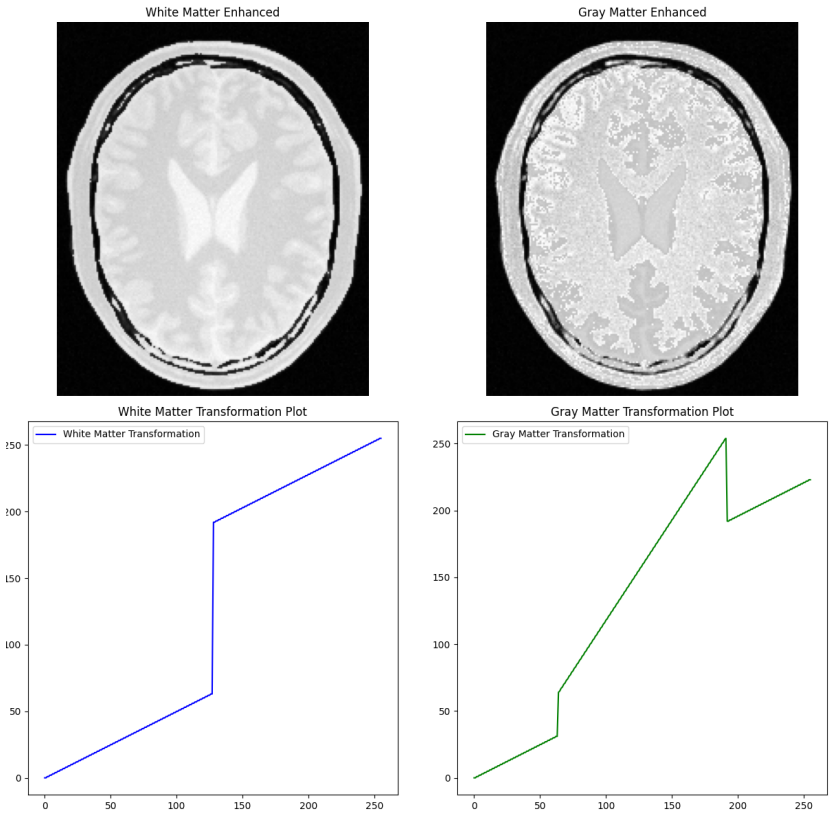

Figure 2: Enhanced White matter (left) and Gray matter (right).

White matter transformation enhances high intensities, while gray matter transformation enhances mid intensities.

Question 3: Gamma Correction

(a) $\gamma = 0.5$

Code:

```python
L, a, b = cv2.split(image_lab)  # Separate the L, a, b channels
def adjust_gamma(image, gamma=1.0):  # Function to apply gamma correction
    table = np.array([((i / 255.0) ** gamma) * 255 for i in np.arange(0, 256)]).astype("uint8")
    return cv2.LUT(image, table)

gamma_corrected_L = adjust_gamma(L, gamma=0.5)    # Apply gamma correction with gamma = 0.5
```

(b) Results:

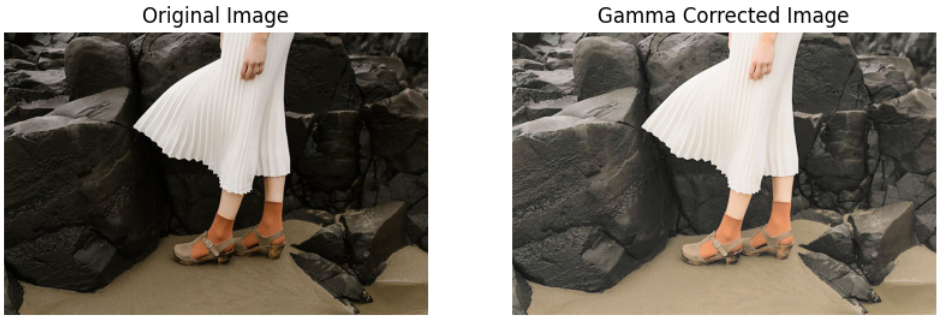

Figure 3: Original image(left) and gamma-corrected image(right).

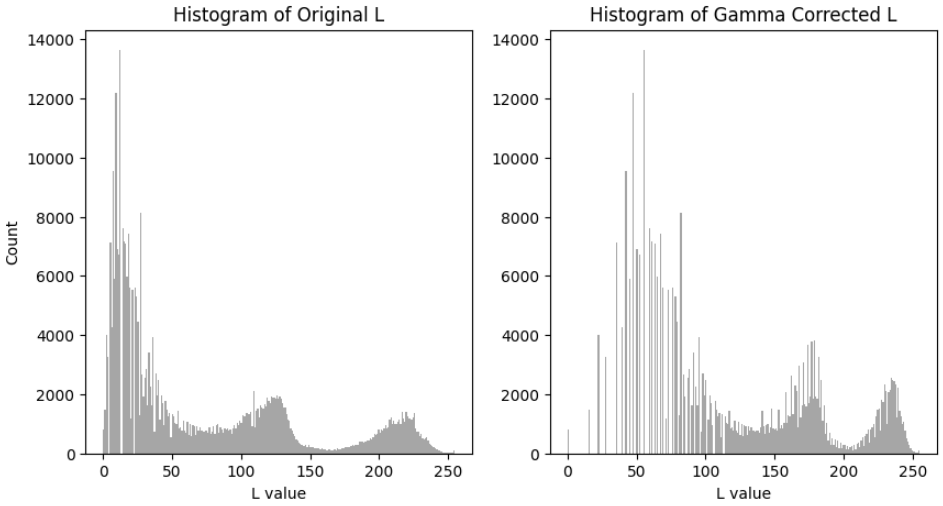

Figure 4: Histograms of the original(left) and gamma-corrected(right) images.

Question 4: Enhancing the Vibrance of a Photograph

Part (a): Splitting the Image

```python
hue, saturation, value = cv2.split(image_hsv)  # Split into hue, saturation, and value components
```

Part (b): Applying the Intensity Transformation

```python
def intensity_transform(x, alpha=0.5, sigma=70):  # Define the transformation function
    return np.minimum(x + alpha * (128 * np.exp(-((x-128)**2) / (2*sigma**2))), 255)
saturation_transformed = intensity_transform(saturation, alpha=0.6).astype(np.uint8)
```

Part (c): Adjusting the Parameter α

The optimal value found was $\alpha = 0.5$.

```
from ipywidgets import interact, FloatSlider
def adjust_saturation_plane(alpha):
    saturation_transformed = intensity_transform(saturation, alpha=alpha).astype(np.uint8)
    plt.figure(figsize=(6, 4))
    plt.imshow(saturation_transformed, cmap='gray')
    plt.colorbar()
    plt.title(f'Transformed Saturation Plane (alpha={alpha:.2f})')
    plt.axis('off')
    plt.show()
# Create an interactive slider
interact(adjust_saturation_plane, alpha=FloatSlider(value=0.5, min=0, max=1, step=0.05, description='Alpha
```

Part (d): Recombining the Planes

```
image_hsv_transformed = cv2.merge([hue, saturation_transformed, value])  # Reconstruct the image
image_transformed = cv2.cvtColor(image_hsv_transformed, cv2.COLOR_HSV2BGR)
```

Part (e): Displaying results

Figure 5: Original

Figure 6: Vibrance-enhanced

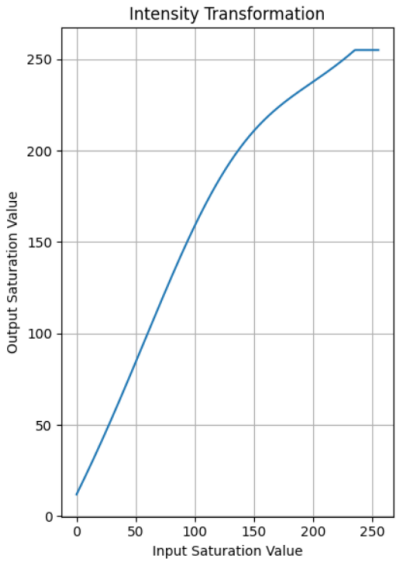

Figure 7: Intensity transformation

Question 5: Histogram Equalization

Function to carry out histogram equalization:

```
def calculate_histogram(image):
    histogram = np.zeros(256, dtype=int)
    for pixel in image.flatten():
        histogram[pixel] += 1
    return histogram
def equalize_histogram(image):
    hist = calculate_histogram(image)  # Calculate the histogram
    cdf = np.cumsum(hist)  # Calculate the cumulative distribution function (CDF)
    cdf_normalized = (cdf) * 255 / cdf.max()    # Normalize the CDF
    cdf_normalized = cdf_normalized.astype(np.uint8)
    equalized_image = cdf_normalized[image]   # Use the normalized CDF to map the original image pixels to
    return equalized_image, hist, calculate_histogram(equalized_image)
```

Result:

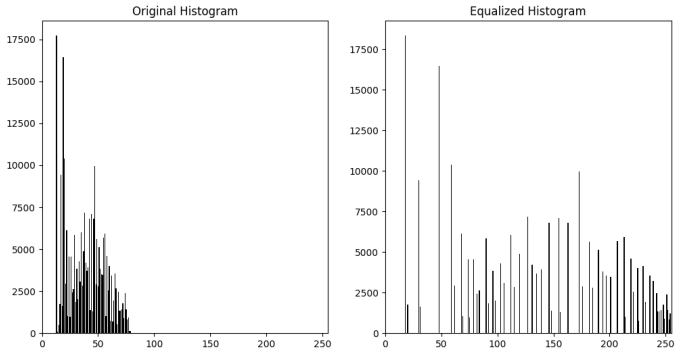

Figure 8: Histograms before and after equalization.

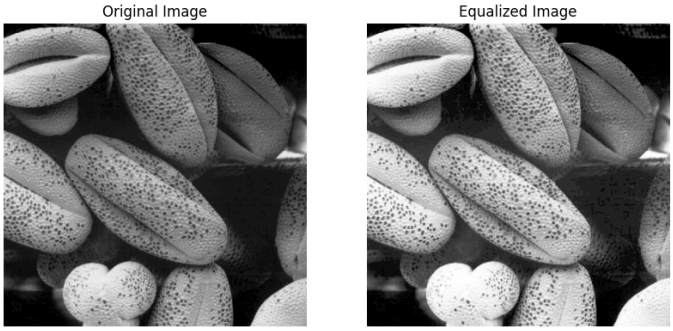

Figure 9: Comparison between the original image (left) and the histogram-equalized image (right).

Question 6: Foreground Histogram Equalization

Part (a): Splitting the Image into HSV Planes

Figure 10: Hue, Saturation, and Value components of the image.

Part (b): Thresholding to Extract the Foreground

The `saturation` component was used to threshold and extract the foreground mask.

```
_, mask = cv2.threshold(saturation, 11, 255, cv2.THRESH_BINARY) #saturation plane for extracting the foreg
```

Figure 11: Binary mask of the foreground and foreground.

Part (c): Obtain the foreground using cv.bitwise_and and compute the histogram

```
foreground = cv2.bitwise_and(image_rgb, image_rgb, mask=mask)  # Extract the foreground using the mask
histogram = calculate_histogram(foreground)  # Calculate histogram of the foreground
```

Part (d): Cumulative Sum

```
cdf = np.cumsum(histogram)  # Calculate the cumulative sum of the histogram
```

Part (e): histogram-equalize the foreground

```
cdf_normalized = (cdf) * histogram.max() / (cdf.max())  # Normalize the CDF
cdf_normalized = cdf_normalized.astype(np.uint8)
equalized_value = cdf_normalized[foreground]  # Map the value plane
```

Part (f): Extract the background and add with the histogram equalized foreground

```
background_image = cv2.bitwise_and(image_rgb, image_rgb, mask=~mask)  # extarct background from original
final_image = cv2.add(background_image, foreground)  # Combine background and equalized foreground
```

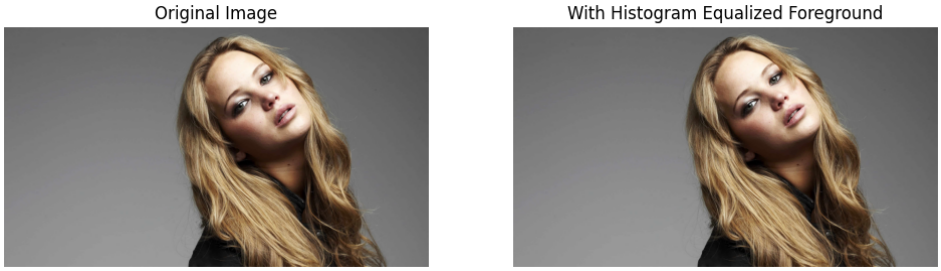

Figure 12: original image, and the result with the histogram equalized foreground.

Question 7: Sobel Filtering

Part (a): Using the Existing Filter

```
sobel_x = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype="int")
sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype="int")
sobel_filtered_x = cv2.filter2D(image, -1, sobel_x)
sobel_filtered_y = cv2.filter2D(image, -1, sobel_y)
```

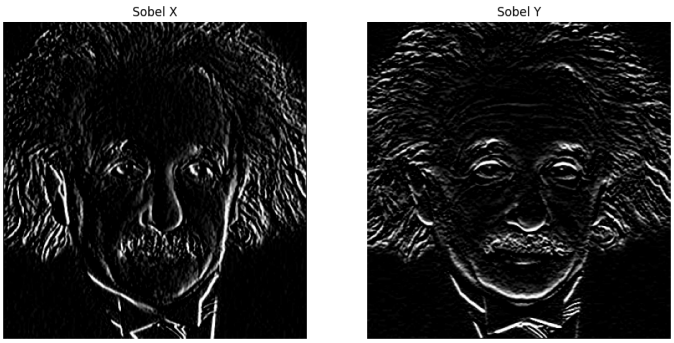

Figure 13: Sobel filtering in the X direction (left) and Y direction (right).

Part (b): Writing Custom Code for Sobel Filtering

```
def apply_sobel_filter(image, kernel):
    rows, cols = image.shape
    padded_image = np.pad(image, 1, mode='constant')  # Pad the image
    output = np.zeros_like(image) # Initialize the output image
    for i in range(1, rows-1):  # Perform convolution
        for j in range(1, cols-1):
            roi = padded_image[i-1:i+2, j-1:j+2]  # Extract the region of interest
```

```
            output[i, j] = np.sum(roi * kernel)  # Perform element-wise multiplication
    return output
sobel_x_output = apply_sobel_filter(image, sobel_x)  # Apply Sobel filter
sobel_y_output = apply_sobel_filter(image, sobel_y)
```

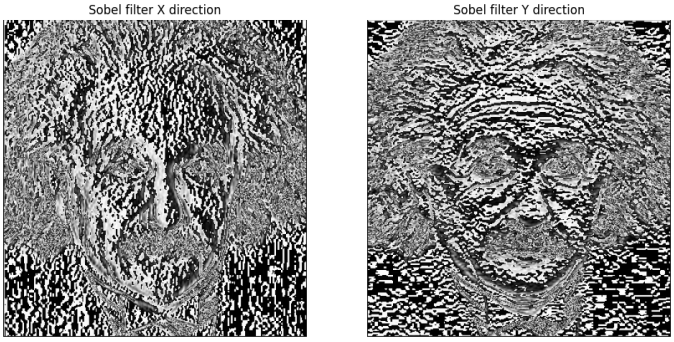

Figure 14: Custom Sobel filtering in the X direction (left) and Y direction (right).

Part (c): Using the Property of Separable Filters

```
# Define the 1D Sobel kernels
sobel_vertical = np.array([[1], [2], [1]])  # Vertical kernel
sobel_horizontal = np.array([[1, 0, -1]])   # Horizontal kernel
# Apply the vertical Sobel filter
gradient_x = cv2.filter2D(image_gray, cv2.CV_64F, sobel_horizontal)  # Using horizontal kernel
# Now apply the vertical kernel to the result
sobel_filtered_image = cv2.filter2D(gradient_x, cv2.CV_64F, sobel_vertical)
# Normalize the output image to the range [0, 255]
sobel_filtered_image=cv2.normalize(sobel_filtered_image, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

```

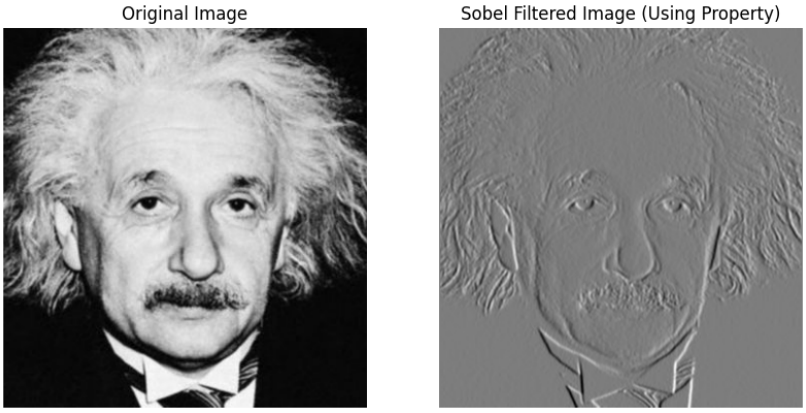

Figure 15: Result of applying separable filters for Sobel operation.

Question 8: Image Zooming

Part (a): Nearest-Neighbor Interpolation

```
def nearest_neighbor_zoom(image, scale_factor):
    src_height, src_width = image.shape[:2]
    dst_height, dst_width = int(src_height * scale_factor), int(src_width * scale_factor)
    zoomed_image = np.zeros((dst_height, dst_width, image.shape[2]), dtype=image.dtype)
    for i in range(dst_height):
        for j in range(dst_width):
            src_x = int(i / scale_factor)
```

```
            src_y = int(j / scale_factor)
            zoomed_image[i, j] = image[src_x, src_y]
    return zoomed_image
```

Part (b): Bilinear Interpolation

```
def bilinear_interpolation_zoom(image, scale_factor):
    src_height, src_width = image.shape[:2]
    dst_height, dst_width = int(src_height * scale_factor), int(src_width * scale_factor)
    zoomed_image = np.zeros((dst_height, dst_width, image.shape[2]), dtype=image.dtype)
    for i in range(dst_height):
        for j in range(dst_width):
            x = i / scale_factor
            y = j / scale_factor
            x0 = int(x)
            y0 = int(y)
            x1 = min(x0 + 1, src_height - 1)
            y1 = min(y0 + 1, src_width - 1)
            a = x - x0
            b = y - y0
            top = (1 - a) * image[x0, y0] + a * image[x0, y1]
            bottom = (1 - a) * image[x1, y0] + a * image[x1, y1]
            zoomed_image[i, j] = (1 - b) * top + b * bottom
    return zoomed_image.astype(image.dtype)
```

Testing and Comparison

```
def compute_ssd(original, scaled):
    return np.mean((original - scaled) ** 2)
original_paths = ["im01.png", "im02.png"]
small_paths = ["im01small.png", "im02small.png"]
scale_factor = 4
for i in range(len(original_paths)):
    original_image = cv2.imread(original_paths[i], cv2.IMREAD_COLOR)
    small_image = cv2.imread(small_paths[i], cv2.IMREAD_COLOR)
    nn_zoomed = nearest_neighbor_zoom(small_image, scale_factor)
    bi_zoomed = bilinear_interpolation_zoom(small_image, scale_factor)
    ssd_nn = compute_ssd(original_image, nn_zoomed)
    ssd_bi = compute_ssd(original_image, bi_zoomed)
    print(f"SSD with Nearest Neighbor for Image {i+1}: {ssd_nn}")
    print(f"SSD with Bilinear Interpolation for Image {i+1}: {ssd_bi}")
```

Results

Image & Method	SSD with Nearest Neighbor	SSD with Bilinear Interpolation
Image 1	31.284316486625514	43.3944849537037
Image 2	11.902013310185184	18.214121238425925

For both images, the nearest neighbor interpolation method resulted in lower SSD values compared to bilinear interpolation. This suggests that, nearest neighbor interpolation more closely approximates the original images.

Question 9: Image Enhancement with Segmentation

Part (a): Using GrabCut for Segmentation

```
mask = np.zeros(image.shape[:2], np.uint8)  # Initialize mask for GrabCut
rect = (65, 50, image.shape[1]-50, image.shape[0]-300) # Define the rectangle parameters that include the
bgdModel = np.zeros((1, 65), np.float64) # Create temporary arrays for the algorithm
fgdModel = np.zeros((1, 65), np.float64)

cv2.grabCut(image, mask, rect, bgdModel, fgdModel, 10, cv2.GC_INIT_WITH_RECT) # Apply GrabCut
```

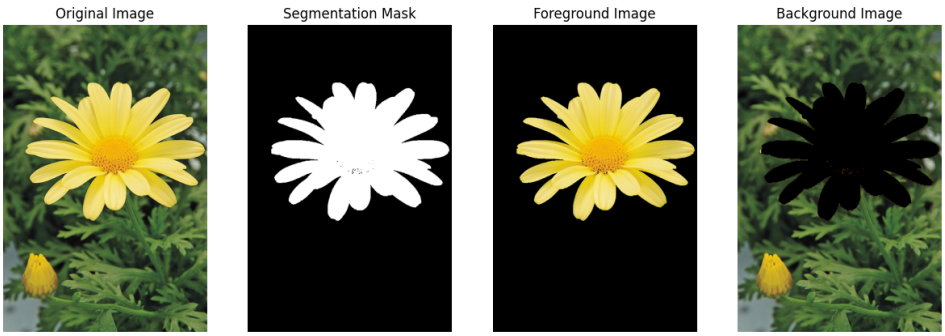

Figure 16: Original image, final segmentation mask, foreground image, and background image.

Part (b): Enhancing the Image

```
background = cv2.bitwise_and(image_rgb, image_rgb, mask=255-mask2)  # Create the background
blurred_background = cv2.GaussianBlur(background, (27, 27), 0)
alpha = 0.5; beta = 0.5; # Combine the blurred background with the original foreground
enhanced_image = cv2.addWeighted(foreground, alpha, blurred_background, beta, 0)
```

Figure 17: Original image(left) and Enhanced image with a blurred background(right).

Part (c): Discussing Background Darkness

The darkness around the flower edges in the enhanced image results from the blurring process. Blurring merges darker edge shadows into the lighter background, creating a contrast effect that accentuates the shadows, making the boundary appear darker than in the original.