Department of Electronic and Telecommunication Engineering
University of Moratuwa

Assignment 01:
Intensity Transformations and Neighborhood Filtering

210294N - Kodithuwakku J.N

This assignment is submitted in partial fulfillment of the requirements for
EN3160 - Image Processing and Machine Vision module.
October 1, 2024

Note: All Python codes relevant to this assignment can be found in GitHub repository.

Question 1

In the given intensity transformation, pixel values between 50 and 150 increase, while the others remain unchanged.

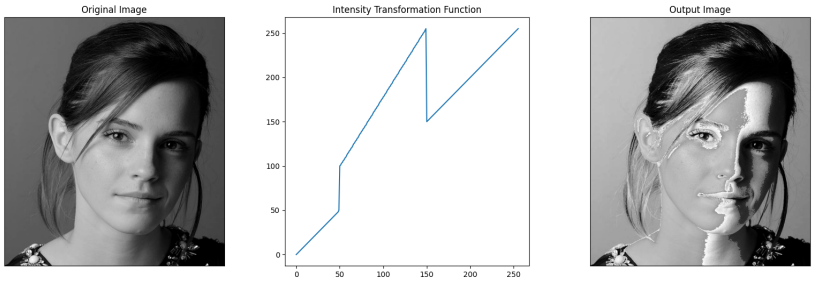


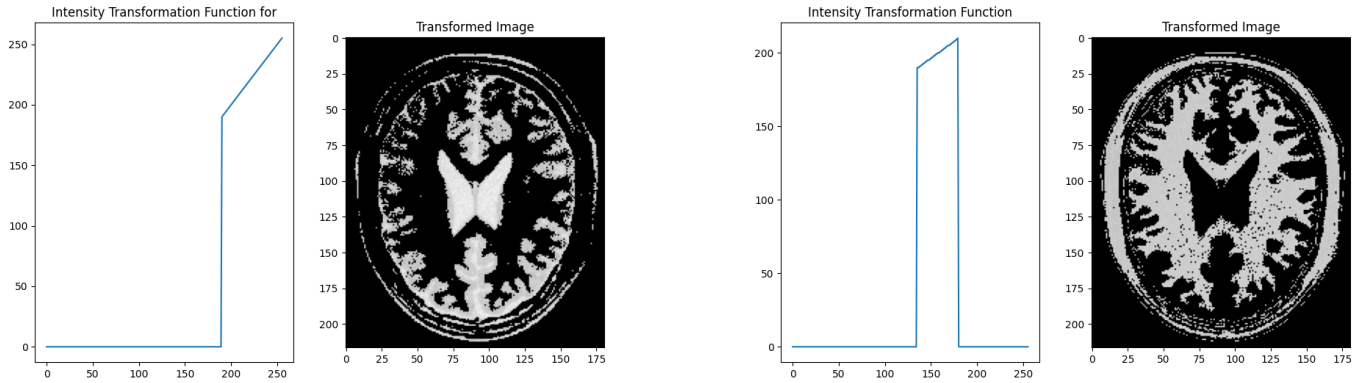
Figure 1

```
# Define the transformation
transform1 = np.concatenate((np.linspace(0, 50, 50),
np.linspace(100, 255, 100), np.linspace(150, 255,106)), axis=0)
        .astype(np.uint8)
# Apply the transformation
image_transformed = cv.LUT(image_orig, transform1)
```

Question 2

In this process, two linear transformations were applied to distinctly enhance the white and gray matter from the original image. The threshold values were determined using a trial-and-error approach, where various values were tested until a significant emphasis on the white and gray matter was achieved. Initially, values ranging from 0 to 255 were selected to isolate the desired color range, followed by the removal of the unwanted linear region. This approach started with a basic unity transformation.

- **White Matter Region**: 175 – 255
- **Gray Matter Region**: 138 – 180



(a) (b)

Figure 2

```
transform3 = np.concatenate((np.linspace(0, 0, 135),
    np.linspace(190, 210, 45),np.linspace(0, 0, 76)), axis=0).
        astype(np.uint8)
fig2_transformed_1 = cv.LUT(fig2, transform3)
```

Question 3

In this question a gamma correction ($\gamma = 0.8$) has been performed on the L plane of the given image after converting it to the L*a*b* colour space. Results are shown in figure 4.

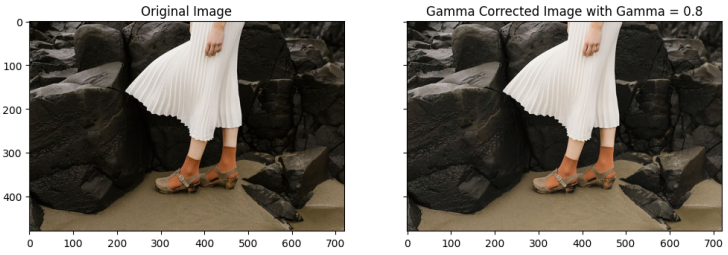


Figure 3

In the L*a*b color space, L represents the pixel's lightness. Based on Equation, applying a gamma value less than 1 results in an increased L value compared to the original. As a result, the gamma-corrected image appears brighter, enhancing the visibility of darker areas, such as rock hollows, and giving them a more appealing look.

$$\text{new L value} = 255 \left(\frac{\text{current L value}}{255} \right)^{0.8} \tag{1}$$

This can also be represented using the histograms of the two versions of the image. We can observe in figure 5, after the gamma correction, the histogram has moved slightly to the right, storing more pixels in the right-most bins.

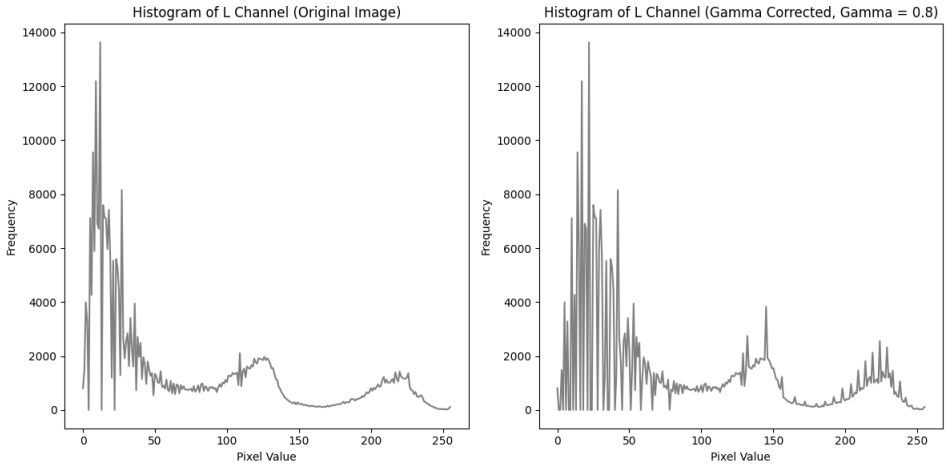


Figure 4: Histogram of L chanel

```python
# Convert the image into CEILAB color space
CEILAB_image = cv.cvtColor(original_image, cv.COLOR_BGR2Lab)
# Split the converted image into three channels
L_chanel, a_channel, b_channel = cv.split(CEILAB_image)

# Apply gamma correction to the L channel of the image
gamma = 0.8
table = np.array([(i/255.0)**(gamma)*255.0
    for i in np.arange(0, 256)]).astype('uint8')
L_chanel_gamma_corrected = cv.LUT(L_chanel, table)

# Merge L channel with other channels
img_gamma = cv.merge((L_chanel_gamma_corrected, a_channel, b_channel))
img_corrected = cv.cvtColor(img_gamma, cv.COLOR_Lab2RGB)
```

Question 4

A beautifully pleasing result is achieved when the value of 'a' is within the range of 0.5 to 0.75

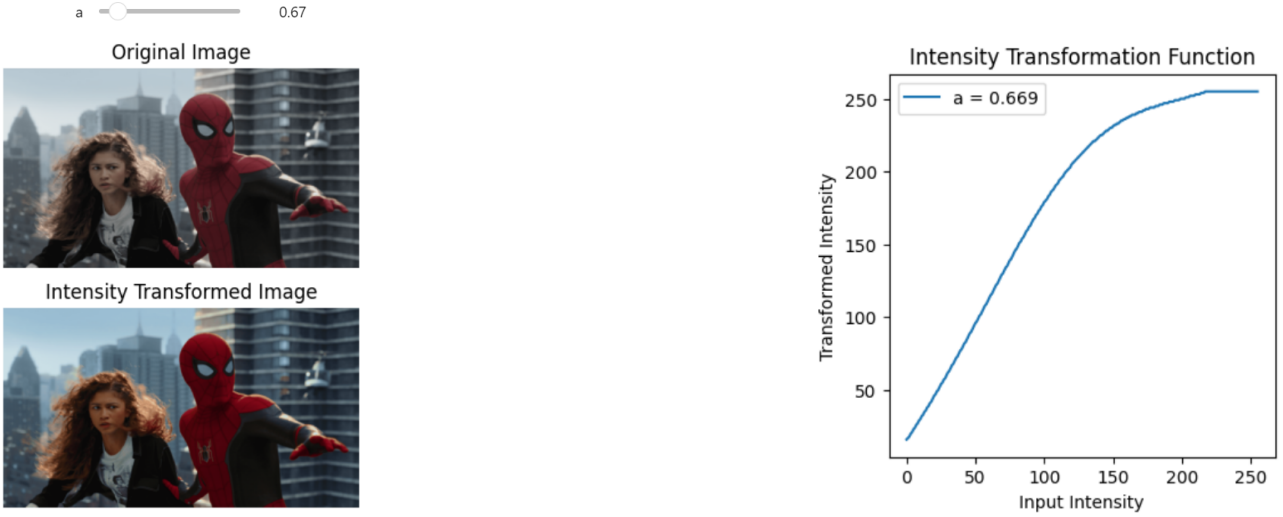


Figure 5

```python
def vibrance(x, a, sigma=70):
    # Transformfunction
    return int(min(x + (a*128)*np.exp((-(x-128)**2)/(2*(sigma**2))), 255))

def transform(a,image):
    table = np.array([vibrance(x, a) for x in np.arange(0, 256)])
        .astype('uint8')
    # Split the image
    h_channel, s_channel, v_channel = cv.split(image)
    # Apply vibrance correction to the saturation plane
    s_channel_corrected = cv.LUT(s_channel, table)
    img_corrected = cv.merge((h_channel, s_channel_corrected, v_channel))
    # Merge corrected plane
    img_corrected_rgb = cv.cvtColor(img_corrected, cv.COLOR_HSV2RGB)
    # plot the images.....
```

Question 5

In the custom equalization method, the image's histogram is first computed, followed by the calculation and normalization of the CDF. The intensity values are then mapped to the 0-255 range. Afterward, the new intensities are added to a table, which is reshaped to match the original image's dimensions. However, this custom approach may produce different results compared to OpenCV's built-in histogram equalization function (cv.equalizeHist()). Here, both the original histogram and the results from OpenCV's built-in equalization, as well as the custom method, in Figure 6.

```python
#Define the manual histogram equalization function
def histogram_equalization(image):
    hist, bins = np.histogram(image.ravel(), 256, [0, 256])
    cdf = hist.cumsum()
    cdf_normalized = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())
    cdf_normalized = cdf_normalized.astype('uint8')
    equalized = cdf_normalized[image]
    return equalized
```

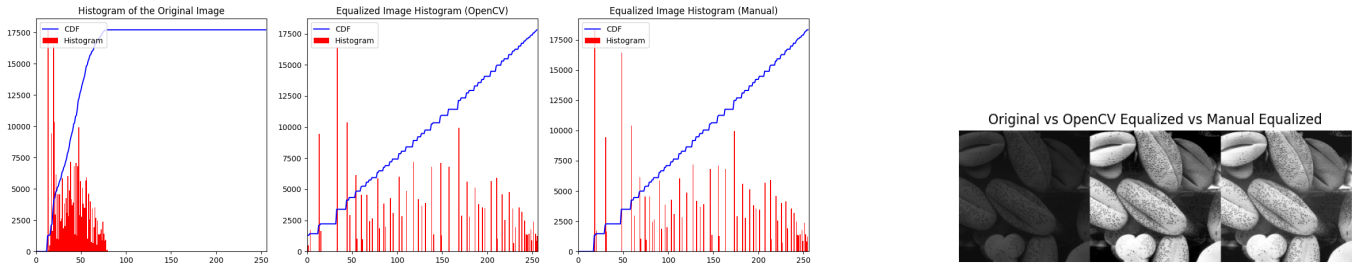
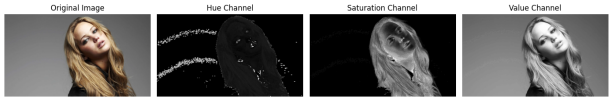


Figure 6

Question 6

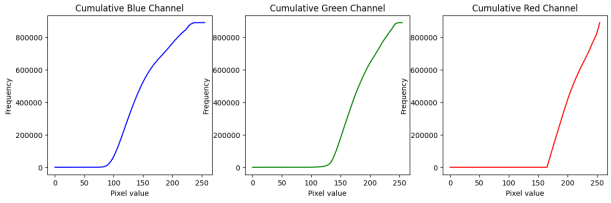
Here the threshold was set to 165

```
img_hsv = cv.cvtColor(jennifer_image, cv.COLOR_BGR2HSV)
h_channel, s_channel, v_channel = cv.split(img_hsv)
# Select a threshold value randomly
threshold = 165

# Apply thresholding on three channels seperately
ret1, foreground_mask1 = cv.threshold(h_channel, threshold, 255,
    cv.THRESH_BINARY)
ret2, foreground_mask2 = cv.threshold(s_channel, threshold, 255,
    cv.THRESH_BINARY)
ret3, foreground_mask3 = cv.threshold(v_channel, threshold, 255,
    cv.THRESH_BINARY)
# Obtain the foreground using the mask from the value channel
foreground_img = cv.bitwise_and(jennifer_image, jennifer_image,
    mask=foreground_mask3)# Calculate histograms for the foreground
b_hist = cv.calcHist([foreground_img], [0], foreground_mask3, [256], [0, 256])
g_hist = cv.calcHist([foreground_img], [1], foreground_mask3, [256], [0, 256])
r_hist = cv.calcHist([foreground_img], [2], foreground_mask3, [256], [0, 256])
# Obtain the culmulative sum of the histogram
cumulative_hist_b = np.cumsum(b_hist)
cumulative_hist_g = np.cumsum(g_hist)
cumulative_hist_r = np.cumsum(r_hist)
# Histogram equalization for three color channels
r_equalized = cv.equalizeHist(foreground_img[:, :, 0])
g_equalized = cv.equalizeHist(foreground_img[:, :, 1])
b_equalized = cv.equalizeHist(foreground_img[:, :, 2])

# Merge the equalized channels
equalized_img = cv.merge((r_equalized, g_equalized, b_equalized))

# Calculate the histograms for channels
r_equalized_hist = cv.calcHist([equalized_img], [0], None, [256], [0, 256])
g_equalized_hist = cv.calcHist([equalized_img], [1], None, [256], [0, 256])
b_equalized_hist = cv.calcHist([equalized_img], [2], None, [256], [0, 256])

# Calculate the CDF for equalized channels
r_cumulative = np.cumsum(r_equalized_hist)
g_cumulative = np.cumsum(g_equalized_hist)
b_cumulative = np.cumsum(b_equalized_hist)# Extract the background by bitwise_not
background = cv.bitwise_and(jennifer_image, jennifer_image,
    mask=cv.bitwise_not(foreground_mask3))

final_modified_img = cv.add(background, equalized_img)
final_modified_img_rgb = cv.cvtColor(final_modified_img, cv.COLOR_BGR2RGB)
```



(a) Split it into hue, saturation, and values

(b) Obtain the foreground



(c) Cumulative sum of the histogram

(d) Extract the background and add with the histogram equalized foreground.

Figure 7

Question 7

```python
def sobel_filter(img):
    # Sobel x and y kernels
    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]])
    # Get image dimensions
    rows, cols = img.shape
    # Initialize output images for x and y direction filters
    filtered_x = np.zeros_like(img, dtype=np.float64)
    filtered_y = np.zeros_like(img, dtype=np.float64)
    # Apply convolution
    for i in range(1, rows-1):
        for j in range(1, cols-1):
            # Extracting the 3x3 region around each pixel
            region = img[i-1:i+2, j-1:j+2]

            # Convolution with sobel_x and sobel_y
            filtered_x[i, j] = np.sum(region * sobel_x)
            filtered_y[i, j] = np.sum(region * sobel_y)
    # Combine gradients
    sobel_combined = np.sqrt(np.square(filtered_x)
        + np.square(filtered_y))

    return filtered_x, filtered_y, sobel_combined
```

- Here, the `openCV filter2D` function was used to carry out the Sobel filter on the Einstein image. Sobel kernels detect the edges of a given image. As can be observed in Figure 8, the Sobel vertical kernel detects horizontal edges, while the Sobel horizontal kernel detects vertical edges.
- A manual Python function was written to carry out the Sobel vertical and horizontal filtering. The function is shown above. Outputs are almost the same as those from the in-built `filter2D` function .(Figure 8)

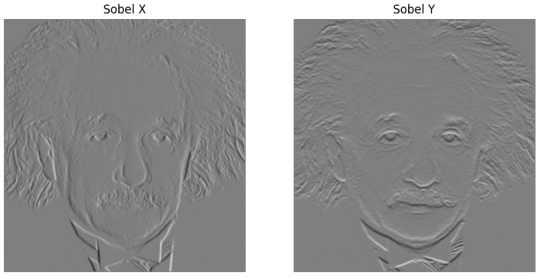


Figure 8

- Sobel filtering was carried out using the associative property. First, the image is filtered with $[-1, 0, 1]^T$ for vertical filtering, and then with $[1, 2, 1]$ for horizontal filtering. Transposes of the above matrices are applied for horizontal filtering. Again, the results are consistent with those from the in-built function.(Figure 8)

```
kernel_1 = np.array(((-1,), (0,), (1,)))
kernel_2 = np.array((1, 2, 1))
output_1 = cv.filter2D(einstein_img, cv.CV_64F, kernel_1)
output_1 = cv.filter2D(output_1, cv.CV_64F, kernel_2)
output_2 = cv.filter2D(einstein_img, cv.CV_64F, kernel_1.T)
output_2 = cv.filter2D(output_2, cv.CV_64F, kernel_2.T)
```

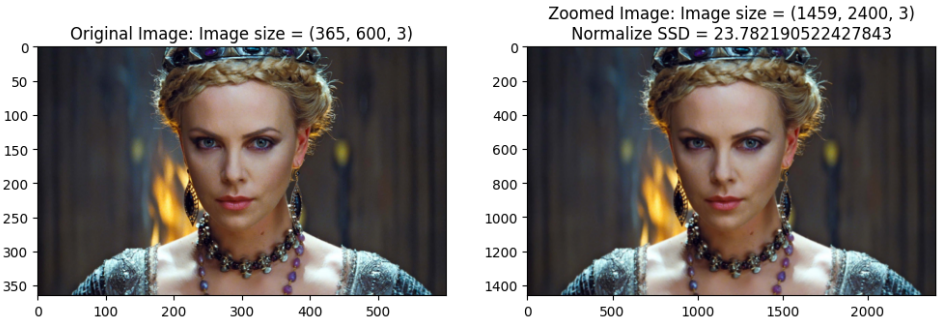
Question 8

For the Nearest-Neighborhood Method, Normalized SSD value is 31.28431, and for the bilinear interpolation method, Normalized SSD value is 23.78219.

```
    def image_zooming(image,zooming_factor):
        height,width,chanels = image.shape
        zoomed_height = int(height*zooming_factor)
        zoomed_width = int(width*zooming_factor)
        zoomed_image = np.zeros((zoomed_height,zoomed_width,chanels),dtype=np.uint8)
        for i in range(zoomed_height):
            for j in range(zoomed_width):
                x = int(i/zooming_factor)
                y = int(j/zooming_factor)
                zoomed_image[i,j] = image[x,y]
        return zoomed_image
```

```
    def Bilinear_interpolation(image, zoom_factor):
        # Get the dimensions of the image
        height, width, channels = image.shape
        # Calculate the new dimensions
        new_height = int(height * zoom_factor) - 1
        new_width = int(width * zoom_factor)
        # Create a new image with the new dimensions
        new_image = np.zeros((new_height, new_width, channels), dtype=np.uint8)
        # Calculate the scaling factor for height and width
        scale_height = height / new_height
        scale_width = width / new_width
        # Iterate through the new image and assign pixel values
        for i in range(new_height):
            for j in range(new_width):
                # Calculate corresponding pixel value in the original image
                x = i * scale_height
                y = j * scale_width

                # Find nearest pixel values
                x0 = int(np.floor(x))
                x1 = min(x0 + 1, height - 1)
                y0 = int(np.floor(y))
                y1 = min(y0 + 1, width - 1)

                # Calculate weights
                dx = x - x0
                dy = y - y0

                # Perform bilinear interpolation
                new_image[i, j] = ((1 - dx) * (1 - dy) * image[x0, y0] +
                    dx * (1 - dy) * image[x1, y0] +
                    (1 - dx) * dy * image[x0, y1] +
                    dx * dy * image[x1, y1]
                )
        return new_image
```

(a) Nearest Neighborhood Method



(b) Bilinear Interpolation Method

Question 9

The enhanced image is produced by blending the foreground image with a Gaussian-blurred background. In this process, the blurred pixels from the background merge with the high-contrast pixels at the flower's edges. This blending causes the transition from the flower's edge to the background to become smoother and less defined. As a result, the transition area appears darker in the enhanced image. However, reducing the kernel size of the Gaussian blur lessens the intensity of this darkening effect, causing the darker regions to gradually fade away.

```
img_orig = cv.imread('new_flower.png', cv.IMREAD_COLOR)
# Create a mask and foreground, background models to initialize GrabCut algorithm
mask = np.zeros(img_orig.shape[:2], np.uint8)
foreground_model = np.zeros((1, 65), np.float64)
background_model = np.zeros((1, 65), np.float64)
rect = (50, 50, img_orig.shape[1] - 50, img_orig.shape[0] - 50)
cv.grabCut(img_orig, mask, rect, background_model, foreground_model, 5,
    cv.GC_INIT_WITH_RECT)   # Apply Grabcut algorithm
new_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
# Modify the mask
# Extract the foreground and background images
foreground_img = img_orig * new_mask[:, :, np.newaxis]
background_img = img_orig * (1 - new_mask[:, :, np.newaxis])
background_blurred_img = cv.GaussianBlur(background_img, (21, 21), 0)
# Apply Gaussian blur to the background
enhanced_img = foreground_img + background_blurred_img
```

Figure 10