# EN3160 - Image Processing and Machine Vision

## Assignment 1 - Intensity Transformations and Neighborhood Filtering

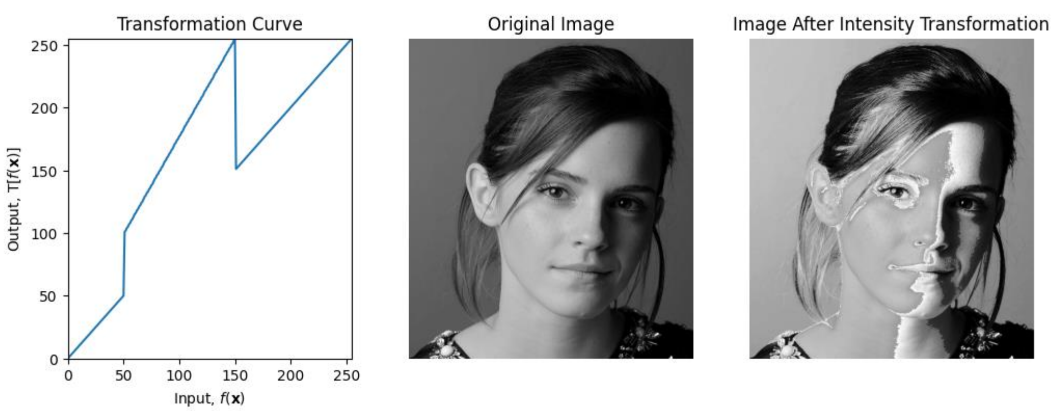

**Name** : A.A.H. Pramuditha

**Index No.** : 200476P

**GitHub Link** : [hashirupramuditha/EN3160-Image-Processing-and-Machine-Vision (github.com)]

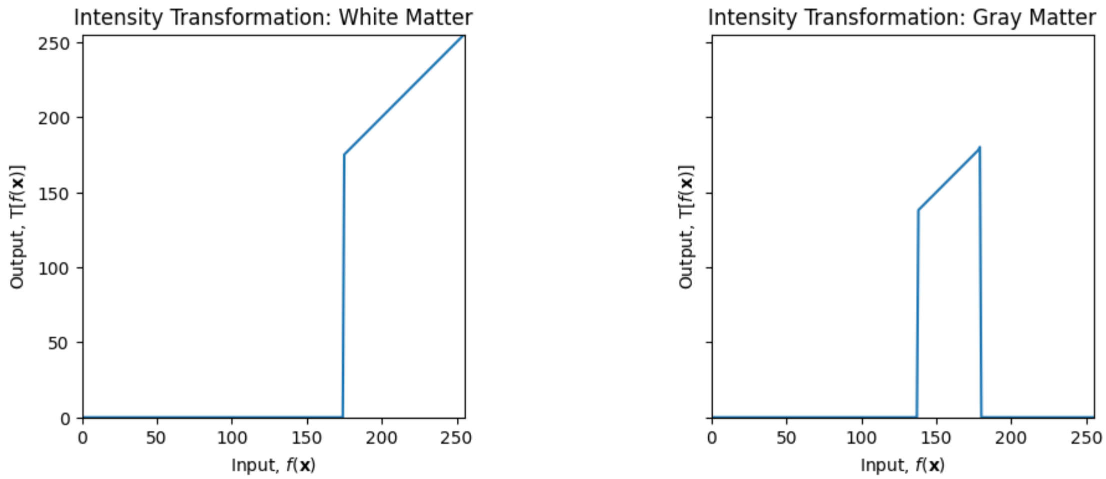

## Question 1: Intensity Transformation

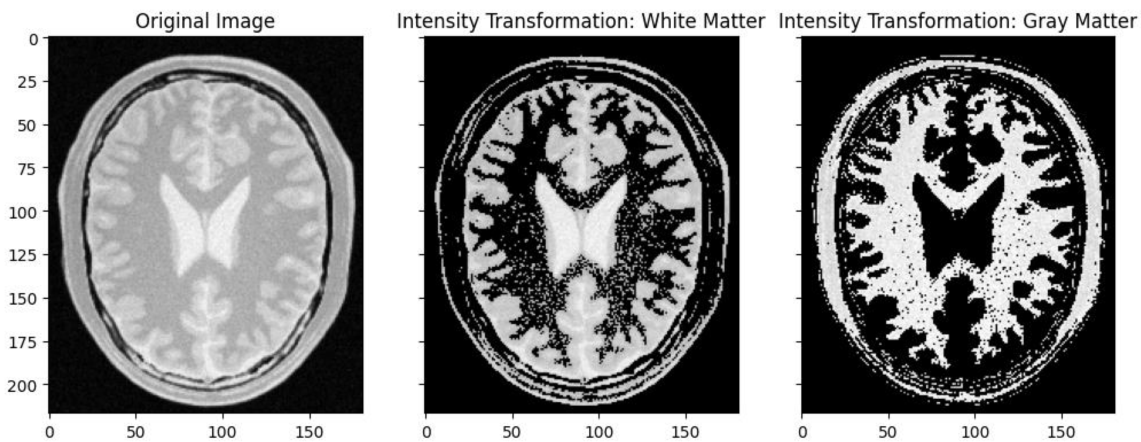
```python
c = np.array([(50, 50), (50, 100), (150, 255), (150, 150), (255, 255)])

t1 = np.linspace(0, c[0, 1], c[0, 0] + 1 - 0).astype('uint8')
t2 = np.linspace(c[0, 1] + 1, c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.linspace(c[1, 1] + 1, c[2, 1], c[2, 0] - c[1, 0]).astype('uint8')
t4 = np.linspace(c[2, 1] + 1, c[3, 1], c[3, 0] - c[2, 0]).astype('uint8')
t5 = np.linspace(c[3, 1] + 1, c[4, 1], c[4, 0] - c[3, 0]).astype('uint8')


transform = np.concatenate((t1, t2), axis=0).astype('uint8')
#region ...

img_orig = cv.imread('emma.jpg', cv.IMREAD_GRAYSCALE)

#region...

image_transformed = cv.LUT(img_orig, transform)
```




## Question 2: Intensity Transformation for a Brain Proton Density Image

```python
img_orig = cv.imread('BrainProtonDensitySlice9.png', cv.IMREAD_GRAYSCALE)

white_threshold_upper = 255
white_threshold_lower = 175
white_matter = np.linspace(0, 255, 256, dtype='uint8')
white_matter[:white_threshold_lower] = 0
white_matter[white_threshold_lower:] = np.linspace(white_threshold_lower, white_threshold_upper, white_threshold_upper - white_threshold_lower + 1, dtype='uint8')

gray_threshold_upper = 180
gray_threshold_lower = 138
gray_matter = np.linspace(0, 255, 256, dtype='uint8')
gray_matter[:gray_threshold_lower] = 0
gray_matter[gray_threshold_upper:] = 0
gray_matter[gray_threshold_lower:gray_threshold_upper] = np.linspace(gray_threshold_lower, gray_threshold_upper, gray_threshold_upper - gray_threshold_lower, dtype='uint8')

white_transform = cv.LUT(img_orig, white_matter)
gray_transform = cv.LUT(img_orig, gray_matter)
```



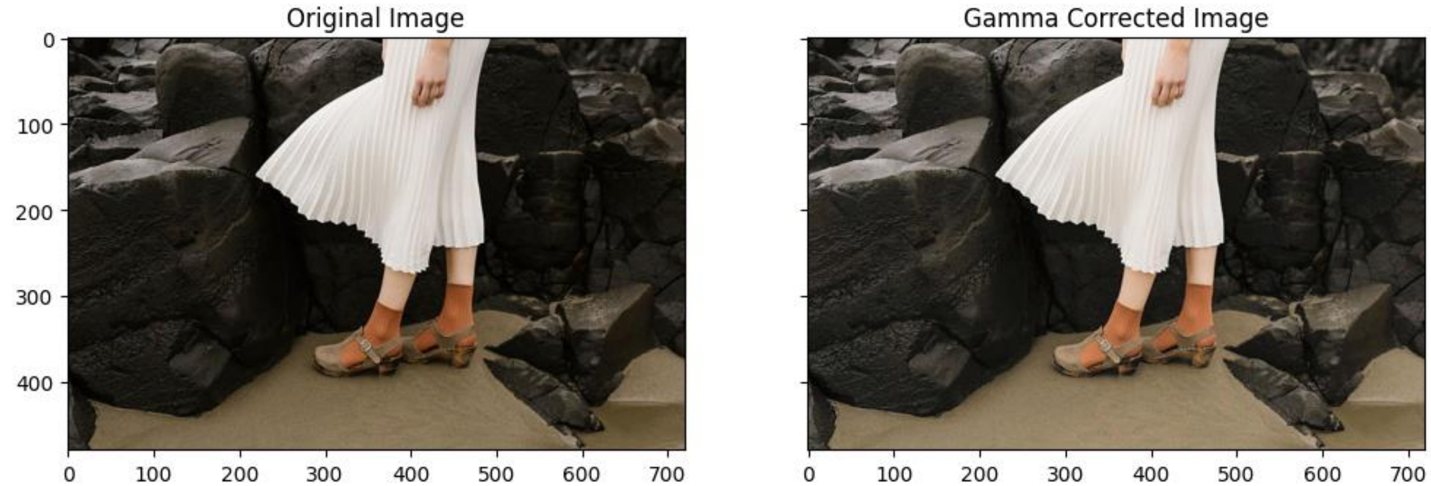



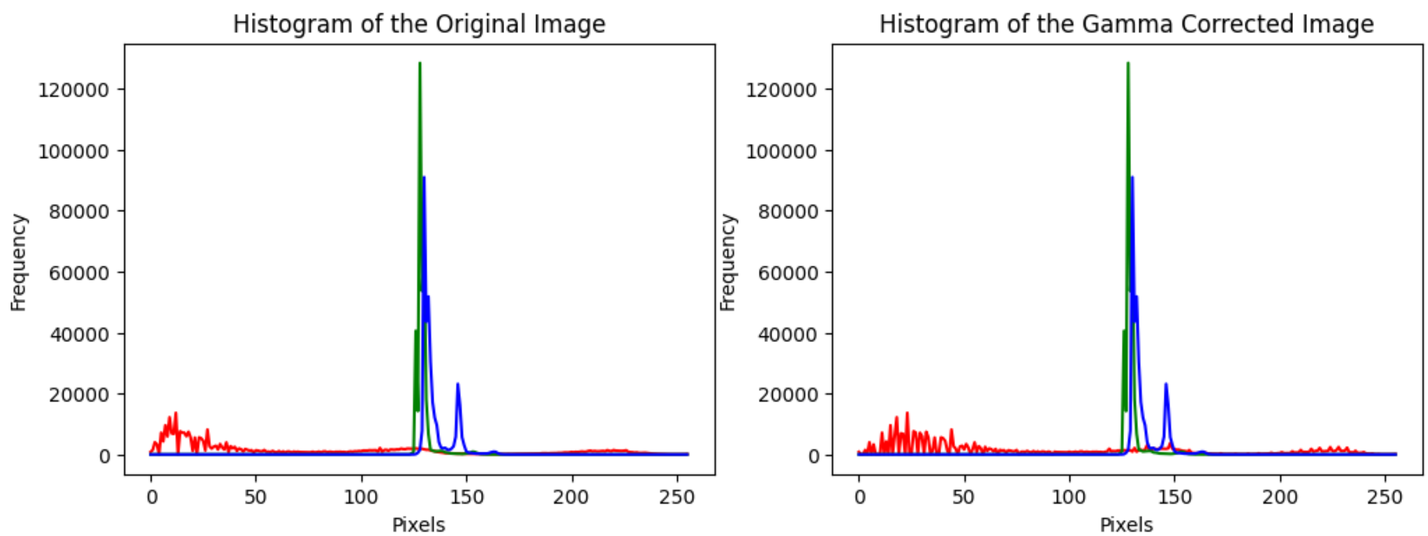

Here, two linear transformations were used to separately emphasize the white and gray matter from the original image, and the threshold values were chosen using the trial-and-error method (trying various threshold values until the white and gray matter are significantly emphasized). Different values between 0-255 were first chosen to isolate the desired color range, after which the undesirable linear region was cut. This process began with a straightforward unity transformation.

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

## Question 3: Gamma Correction

```
img_ceilab = cv.cvtColor(img_orig, cv.COLOR_BGR2Lab)         # Convert the image into CEILAB color space
l_channel, a_channel, b_channel = cv.split(img_ceilab)      # Split the converted image into three channels

> #region...

gamma = 0.78
table = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0, 256)]).astype('uint8')
l_channel_gamma_corrected = cv.LUT(l_channel, table)         # Apply gamma correction only for L channel
img_gamma = cv.merge((l_channel_gamma_corrected, a_channel, b_channel))     # Merge L channel with other channels
img_corrected = cv.cvtColor(img_gamma, cv.COLOR_Lab2RGB)

> #region...

f, ax = plt.subplots(1, 2, figsize=(12, 6))
space = ('l', 'a', 'b')
color = ('r', 'g', 'b')

for i, c in enumerate(space):
    hist_orig = cv.calcHist([img_ceilab], [i], None, [256], [0, 256])       # Calculate histogram for original image
    ax[0].plot(hist_orig, color=color[i])
    hist_gamma = cv.calcHist([img_gamma], [i], None, [256], [0, 256])       # Calculate histogram for gamma corrected image
    ax[1].plot(hist_gamma, color=color[i])
```



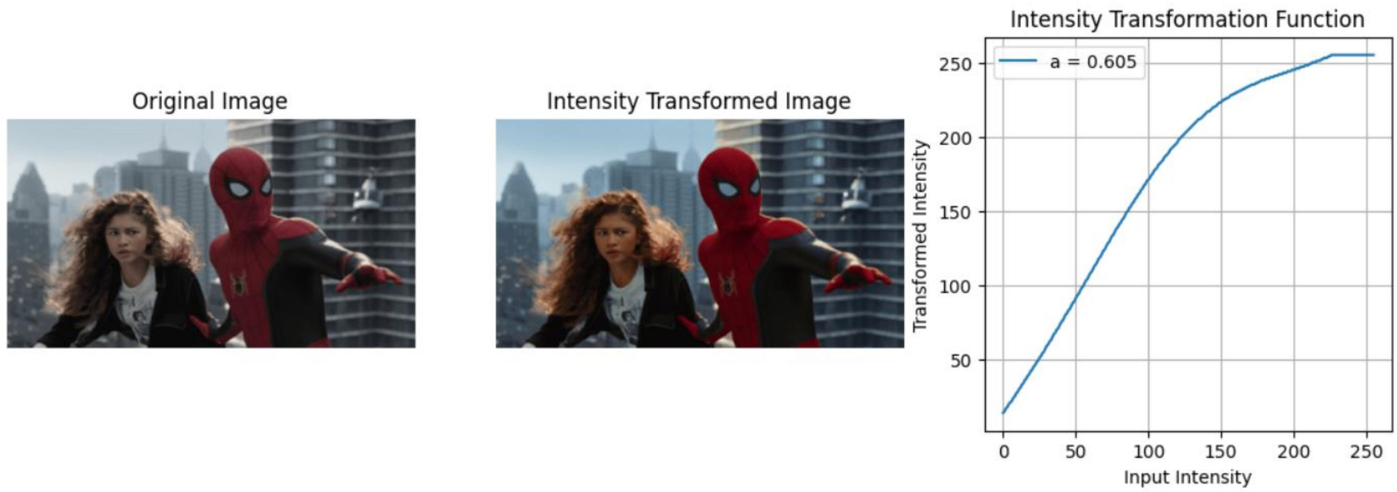

After trying different values, 0.78 is selected as the gamma value to apply gamma correction for the L plane, which is depicted in red color in the histograms.

## Question 4: Increasing the Vibrance of a Photograph by Intensity Transformation

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

def transform(a):
    # This function will apply the desired transformation to selected planes of the image
    plt.clf()
    table = np.array([vibrance(x, a) for x in np.arange(0, 256)]).astype('uint8')
    s_channel_corrected = cv.LUT(s_channel, table)          # Apply vibrance correction to the saturation plane
    img_corrected = cv.merge((h_channel, s_channel_corrected, v_channel))  # Merge corrected plane with hue and value planes
    img_corrected_rgb = cv.cvtColor(img_corrected, cv.COLOR_HSV2RGB)
    #region···

img_orig = cv.imread('spider.png', cv.IMREAD_COLOR)
img_rgb = cv.cvtColor(img_orig, cv.COLOR_BGR2RGB)
img_hsv = cv.cvtColor(img_orig, cv.COLOR_BGR2HSV)         # Convert the image into HSV color space
h_channel, s_channel, v_channel = cv.split(img_hsv)        # Split the image into hue, saturation and value planes

# Interactive Slider
final_plot = interactive(transform, a=(0, 1, 0.001))
output = final_plot.children[-1]
output.layout.height = '720px'
final_plot
```

a 0.60

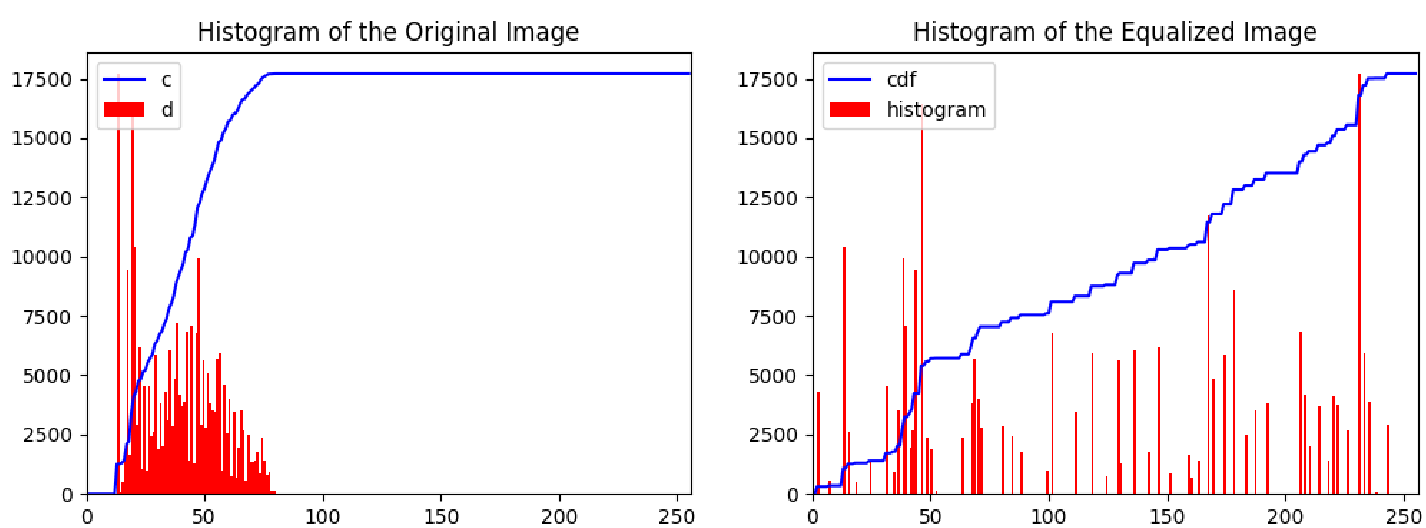
<Figure size 640x480 with 0 Axes>

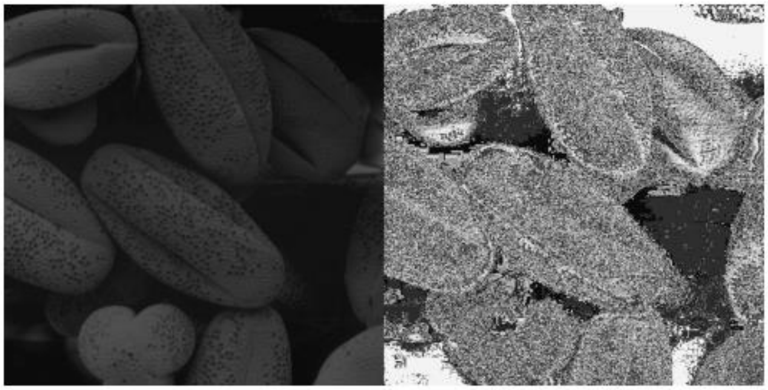



A visually pleasing output can be obtained when the value of “a” is in the range of 0.55 – 0.7.

## Question 5: Histogram Equalization

```
def equalize(image):
    histogram,  bins = np.histogram(img.flatten(), bins=256, range=(0, 256))    # Calculate the histogram of the image
    cdf = hist.cumsum()                                                          # calculate cumulative dixtribution function
    cdf_normalized = cdf * histogram.max()/cdf.max()                             # Normalize the CDF to map the range into (0-255) range
    table = np.interp(image.flatten(), bins[:-1], cdf_normalized)                # Store the mapped values into a table
    equalized_image = table.reshape(image.shape)                                 # Reshape the table into the shape of the original image
    return equalized_image.astype('uint8')


img = cv.imread('shells.tif', cv.IMREAD_GRAYSCALE)

hist,  bins = np.histogram(img.ravel(), 256, [0, 256])      # Calculate the histogram of the original image
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max()/cdf.max()
> #region ···

equ = equalize(img)
hist, bins = np.histogram(equ.ravel(), 256, [0, 256])      # Calculate the histogram of the equalized image
cdf = hist.cumsum()
cdf_normalized = cdf*hist.max()/cdf.max()
> #region ···
```

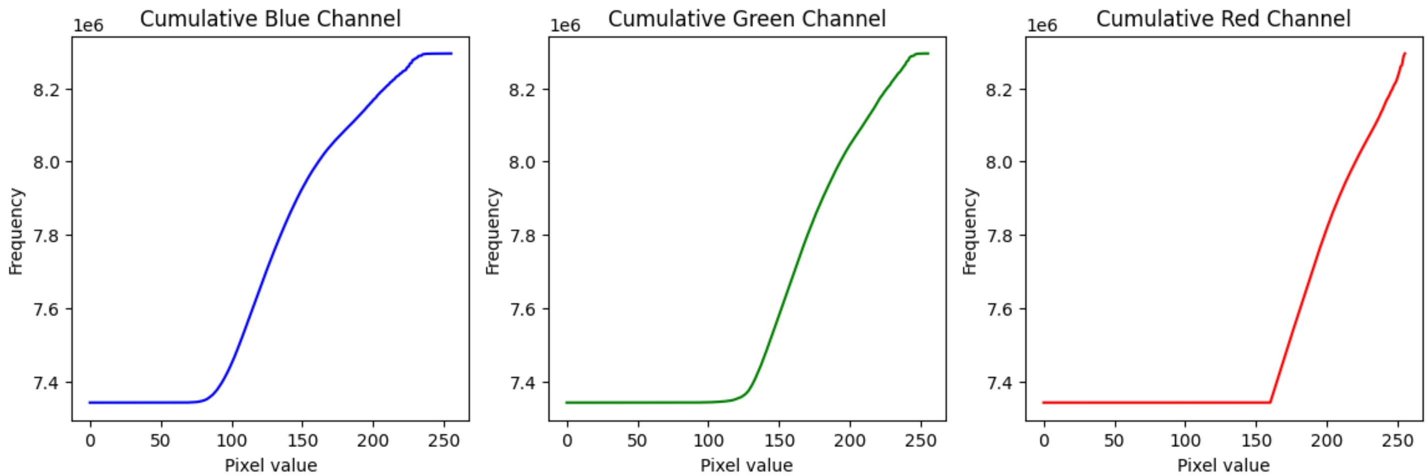




In the custom equalization method, the histogram of the image is computed, followed by calculation and normalization of the CDF. Intensity values are then mapped to the 0-255 range. Then the new intensities are included in the table and the table will be reshaped into the shape of the original image. However, this custom method may yield different results from OpenCV's built-in histogram equalization function (cv.equalizeHist()).

## Question 6: Histogram Equalization to Make Histogram Equalized Foreground

```
img_hsv = cv.cvtColor(img_orig, cv.COLOR_BGR2HSV)      # Convert the image into HSV color space
h_channel, s_channel, v_channel = cv.split(img_hsv)     # Split the converted image into hue, saturation and value planes

# Select a threshold value randomly
threshold = 160

# 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(img_orig, img_orig, mask=foreground_mask3)

# 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))

# Extract the background by bitwise_not
background = cv.bitwise_and(img_orig, img_orig, 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)
```



The threshold value for the above task was selected as 160 randomly.

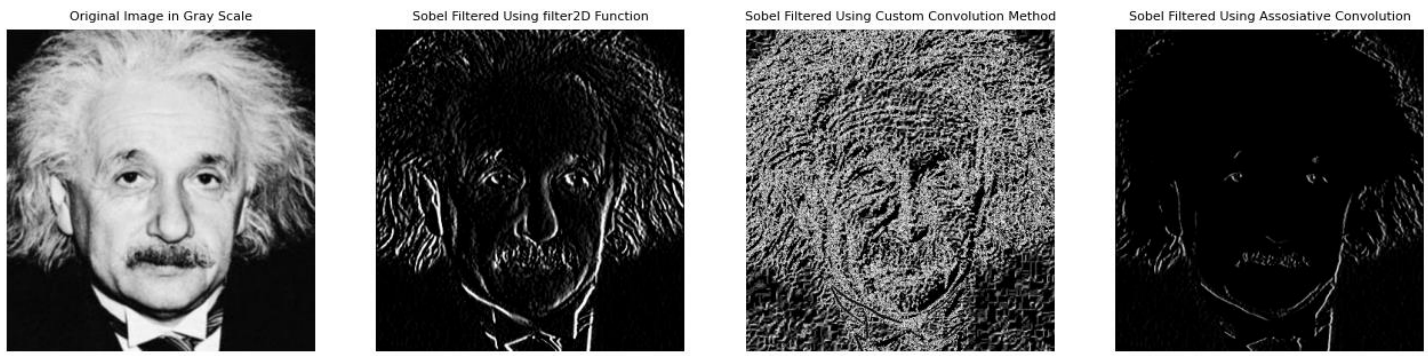
## Question 7: Sobel Filtering

```
def sobel_filter(image, kernel):
    # Sobel filtering image using convolution
    img_height, img_width = image.shape
    kernel_size = kernel.shape[0]
    output_img = np.zeros((img_height - kernel_size + 1, img_width -
    for i in range(output_img.shape[0]):
        for j in range(output_img.shape[1]):
            region = image[i : i + kernel_size, j : j + kernel_size]
            conv_result = np.sum(region * kernel)
            output_img[i, j] = conv_result
            #region···
    return output_img.astype(np.uint8)

img_orig = cv.imread('einstein.png', cv.IMREAD_GRAYSCALE)
# Introducing the kernels which will be convoluted with the image.
x_kernel = np.array([(-1, 0, 1), (-2, 0, 2), (-1, 0, 1)], dtype='flo
y_kernel = np.array([(-1, -2, -1), (0, 0, 0), (1, 2, 1)], dtype='flo
```

Two kernels were added to the custom method to carry out the convolution with the image in both the horizontal and vertical axes. From the generated horizontal and vertical gradient images, it then determines the gradient's magnitude.

```
img = cv.imread('einstein.png', cv.IMREAD_GRAYSCALE)

column_kernel = np.array([(1), (2), (1)], dtype='float')
row_kernel = np.array([(1, 0, -1)], dtype='float')

conv_1 = cv.filter2D(img, -1, column_kernel)    # First Convolute with the co
conv_2 = cv.filter2D(conv_1, -1, row_kernel)    # Convolute the above result w
```

Although the filter2D() function is utilized in this case, the convolution of two kernels was accomplished in two steps. First, a row vector and a column vector were created from the supplied kernel. We can convolution with these row and column vectors individually thanks to the associativity of convolution.




## Question 8: Zooming Images

### (a) Nearest Neighborhood Method

```python
def ssd(img1, img2):
    # Calculate the sum of squared difference
    return np.sum((img1 - img2)**2)

def zooming(original_image, zoom_factor):
    height, width, channels = original_image.shape

    # Apply zooming factor and prepare the shape of the zoomed image
    zoomed_height = int(height*zoom_factor) - 1
    zoomed_width = int(width*zoom_factor)
    zoomed_image = np.zeros((zoomed_height, zoomed_width, channels), dtype=np.uint8)

    for i in range(zoomed_height):
        for j in range(zoomed_width):
            # Zooming operation pixelwise implementation
            zoomed_image[i, j] = original_image[int(i/zoom_factor), int(j/zoom_factor)]
```

### (b) Bilinear Interpolation Method

```python
def zooming(original_image, zoom_factor):
    # Zooming by bilinear interpolation method
    height, width, channels = original_image.shape
    zoomed_height = int(height*zoom_factor)
    zoomed_width = int(width*zoom_factor)
    zoomed_image = np.zeros((zoomed_height, zoomed_width, channels), dtype=np.uint8)

    y_scale = height / zoomed_height
    x_scale = width / zoomed_width

    for i in range(zoomed_height):
        for j in range(zoomed_width):
            original_y = i * y_scale
            original_x = j * x_scale

            # Calculate the four nearest neighbours
            x1, y1 = int(original_x), int(original_y)
            x2, y2 = x1 + 1, y1 + 1

            # check boundaries
            if x2 >= width: x2 = width - 1
            if y2 >= height: y2 = height - 1

            # Interpolation weights
            weight_x = original_x - x1
            weight_y = original_y - y1

            # Apply bilinear interpolation
            pixel_interpolated = ((1 - weight_x) * (1 - weight_y) * original_image[y1, x1] +  weight_x * (1 - weight_y) * original_image[y1, x2] +
                                 (1 - weight_x) * weight_y * original_image[y2, x1] +  weight_x * weight_y * original_image[y2, x2])

            # Set the pixel value in zoomed image
            zoomed_image[i, j] = pixel_interpolated
```



For the Nearest-Neighborhood Method, SSD value is 180381523, and for the bilinear interpolation method, SSD value is 2031930775.

## Question 9: Image Segmentation

```python
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)                      # Define rectangles around the foreground

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                                # Combine blurred and original images together
```


Here we create the enhanced image by combining foreground image with a gaussian blurred background. When combining these two images, these gaussian blurred pixels will mix with the pixels representing flower's edge which contain high contrast information. When these pixels are mixed, the transition region from the flower edge to the background of the image becomes less distinct and less noticeable. That's why the transition region appears quite dark in the enhanced image.