EN3160 Image Processing & Machine Vision

Assignment 01

Name : M.P.D.N. Wickramasingha

Index No : 210705E

Github Repository : https://github.com/shan-wrench/IPMV-Assignment01

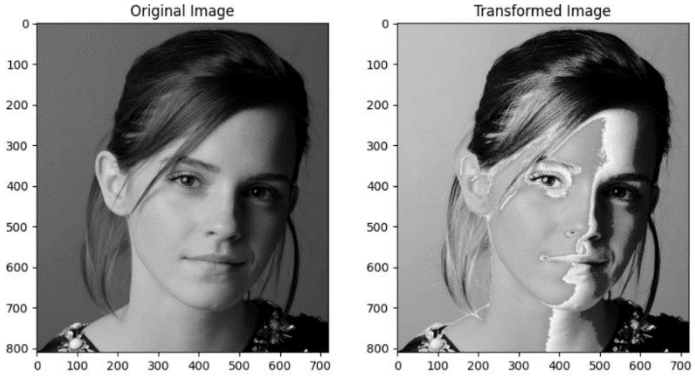
Q.01 : To convert the graph into function, Used a **intensity_transformation** function and include functions for ranges like that.

1. For pixel values less than or equal to **50**, it retains the original value (range 1).
2. For values between **50** and **150**, it applies a linear transformation, increasing brightness with a slope **m = 1.55** and a y-intercept derived from the line equation (range 2).
3. For pixel values above **150**, the function simply returns the original value (range 3).

Then applied transformation using below function. Here used **numpy vectorization** to efficiently process pixel values.

- The pixel values are transformed using the vectorized **intensity_transformation** function.
- The resulting pixel values are clipped to ensure they remain within the valid range (0 to 255) and converted back to **uint8** format. This ensures compatibility with standard image formats.

```python
# Apply the transformation to an image
def apply_transformation(image):
    # Vectorize the intensity transformation function for performance
    vectorized_transformation = np.vectorize(intensity_transformation)

    # Apply the transformation
    transformed_image = vectorized_transformation(image)

    # Convert the transformed image back to uint8 type (0-255)
    transformed_image = np.clip(transformed_image, 0, 255).astype(np.uint8)

    return transformed_image
```

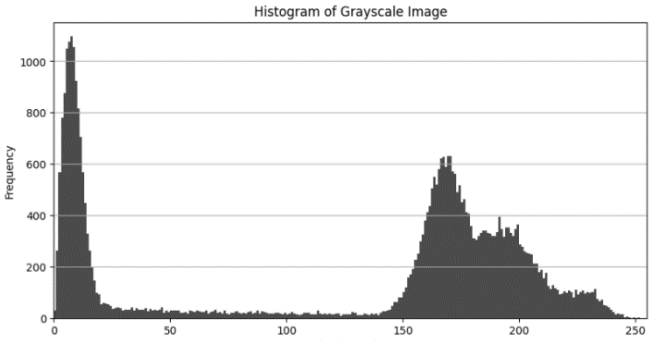


The transformation effectively enhances the image's visual appeal by adjusting the brightness and contrast. The method specifically targets the mid-range intensities, which often lack sufficient distinction in standard grayscale images. This adjustment leads to a more dynamic representation of the image, making features more distinguishable, especially in areas with previously subdued pixel values. But in here it looks like there are some issues in picture like having mutations. It is because of intensity transformation not suitable for this picture.

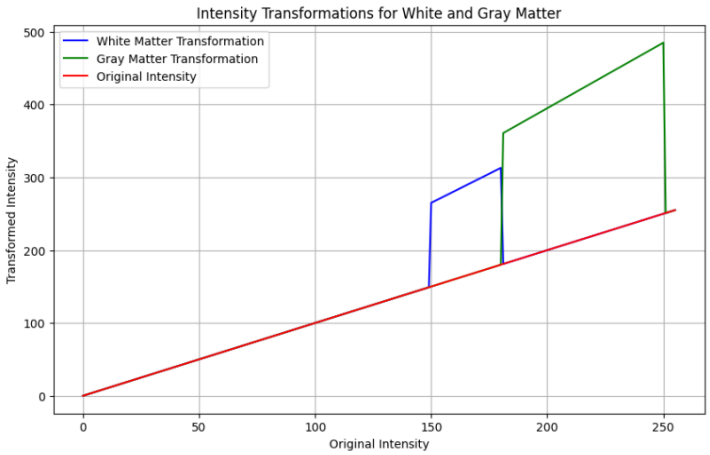
Q.02) : In this question , Trying to identify white matter and gray matter of brain. Here What I do to implement it. First roughly got an idea of intensity of white matter and gray matter of the brain. For that, I used two different points from picture. Then I got the histogram of this image to find how pixels spread on the picture with which values and found a margin value to find white and gray color. I created a function to transform pixel intensities for white and grey matter. The logic behind the intensity ranges:

- **Gray Matter**: I chose to transform intensities between 181 and 255, as grey matter tends to have higher intensity values.
- **White Matter**: I transformed intensities between 150 and 180 because white matter appears slightly darker than grey matter but still has a distinctive range.

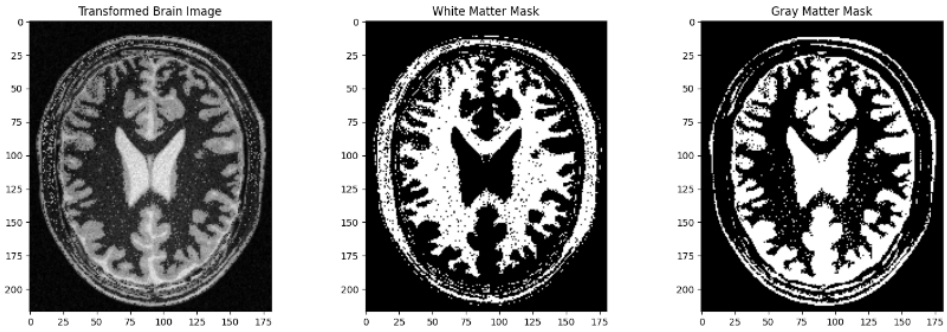
The scaling factors (1.8 for grey matter and 1.6 for white matter) were chosen to exaggerate the differences, making the regions more distinct in the final image.




```python
def Intensity_transform(image):
    # Create a copy of the image to apply transformations
    transformed_image = np.copy(image)

    # Apply transformation for white matter (150 <= pixel <= 250) # gray matter is
    gray_matter_mask = (image >= 181) & (image <= 255)
    transformed_image[gray_matter_mask] = 1.8 * image[gray_matter_mask] + 35

    # Apply transformation for gray matter (50 <= pixel <= 150) # white matter is
    white_matter_mask = (image >= 150) & (image <= 180)
    transformed_image[white_matter_mask] = 1.6 * image[white_matter_mask] + 25


    return transformed_image, white_matter_mask, gray_matter_mask
```



Then plotted the intensity graph for white matter and grey matter with original intensity in same plot. For better representation I changed the m, c values of the function.

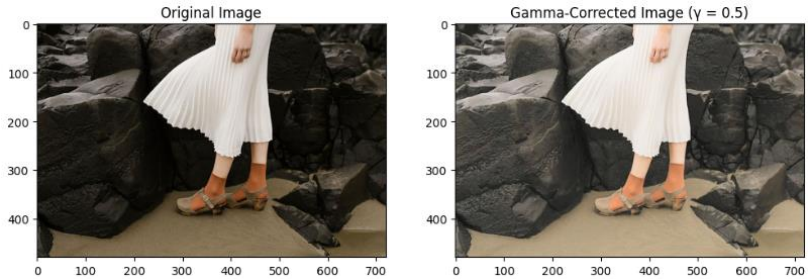


The above process effectively accentuate the white and grey matter, providing clearer differentiation between the two tissue types. The intensity transformation plots and masks show that the transformation process highlights these regions, making them easier to analyse.

Q.03 : Mainly in this question, I had to apply gamma correction to the lightness (L) plane of the L*a*b* color space in the provided image, and compare the histograms of the original and corrected images. Mainly in the code I used these steps to reconstruct the image.

- L Channel Extraction: The L channel, which represents lightness, is extracted and normalized for applying gamma correction. The L*a*b* color space is used because gamma correction is most effective on the L (lightness) plane. This isolates changes in brightness while keeping the color information (a and b channels) intact.
- Gamma Correction: The L channel undergoes gamma correction using a gamma value of 0.5 emphasize brighter regions and darken shadows, resulting in a more visually dynamic image with enhanced contrast.
- Image Reconstruction: The gamma-corrected L channel is merged back with the original a and b channels and converted back to BGR and RGB formats for display.

```python
# Convert the image from BGR (OpenCV default) to RGB for displaying
input_image_rgb = cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)

# Convert the image from BGR to L*a*b* color space for gamma correction
lab_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2LAB)

# Split the L (lightness), a, and b channels
L_channel, a_channel, b_channel = cv2.split(lab_image)

# Normalize the L channel to range [0, 1] for performing gamma correction
L_norm = L_channel / 255.0

# Apply gamma correction (adjustable gamma value)
gamma_value = 0.5  # Adjust gamma as needed
L_gamma_corrected = np.power(L_norm, gamma_value)

# Scale the gamma-corrected L channel back to [0, 255]
L_gamma_corrected = np.uint8(L_gamma_corrected * 255)

# Merge the gamma-corrected L channel with the original a and b channels
lab_corrected_image = cv2.merge([L_gamma_corrected, a_channel, b_channel])

# Convert the LAB image back to BGR color space for visualization
output_image_bgr = cv2.cvtColor(lab_corrected_image, cv2.COLOR_LAB2BGR)
output_image_rgb = cv2.cvtColor(output_image_bgr, cv2.COLOR_BGR2RGB)

# Split the original and gamma-corrected RGB images into R, G, B channels
R_original, G_original, B_original = cv2.split(input_image_rgb)
R_corrected, G_corrected, B_corrected = cv2.split(output_image_rgb)
```

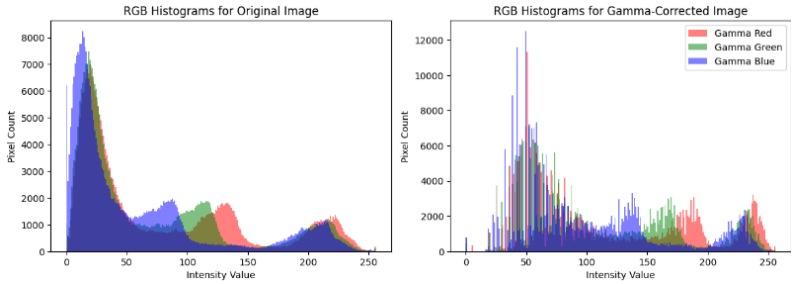


After gamma correction, the image appears brighter with more pronounced contrast between light and dark areas. If I used gamma value more than 1 I can get dark image than original value.

```
# Histogram for the original image's R, G, B channels
plt.hist(R_original.ravel(), bins=256, range=(0, 256), color='red', alpha=0.5, label='Original Red')
plt.hist(G_original.ravel(), bins=256, range=(0, 256), color='green', alpha=0.5, label='Original Green')
plt.hist(B_original.ravel(), bins=256, range=(0, 256), color='blue', alpha=0.5, label='Original Blue')
```

```
# Histogram for the gamma-corrected image's R, G, B channels
plt.hist(R_corrected.ravel(), bins=256, range=(0, 256), color='red', alpha=0.5, label='Gamma Red')
plt.hist(G_corrected.ravel(), bins=256, range=(0, 256), color='green', alpha=0.5, label='Gamma Green')
plt.hist(B_corrected.ravel(), bins=256, range=(0, 256), color='blue', alpha=0.5, label='Gamma Blue')
```

Histogram Comparison: RGB histograms are plotted for both the original and gamma-corrected images to compare pixel intensity distributions in each color channel. The histograms how pixel intensity is redistributed, providing a clear visual explanation of how gamma correction affects the image's color and brightness.



The histograms of the gamma-corrected image show a shift in pixel intensities, particularly in the L (lightness) component, indicating that the correction was successful in redistributing brightness.

Q.04 : a) The image is split into three separate planes: hue, saturation, and value. Each plane represents a different aspect of the color. The image was successfully split into three separate planes, which were visualized using Matplotlib. Each plane reveals different color characteristics, with the hue plane showing the color type, saturation showing the intensity of the color, and value showing the brightness.

```
# Split the image into hue, saturation, and value planes
spiderman_hue_plane = hsv_spiderman_img[:,:,0]
spiderman_saturation_plane = hsv_spiderman_img[:,:,1]
spiderman_value_plane = hsv_spiderman_img[:,:,2]
```

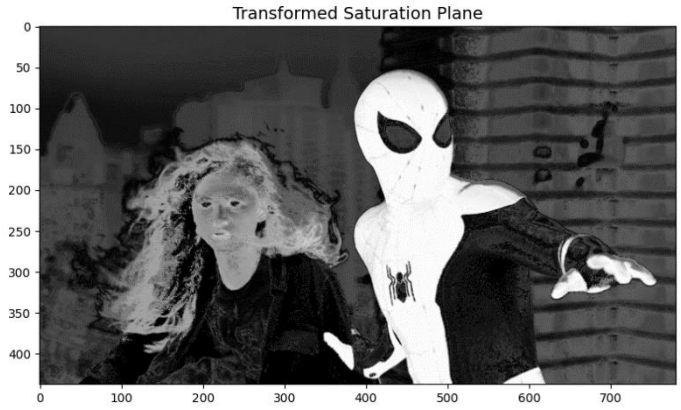
b) The intensity transformation defined by the equation below was applied to the saturation plane. This transformation enhanced the vibrance of the colors in the image.

$$f(x) = \min\left(x + a \times 128 e^{-\frac{(x-128)^2}{2\sigma^2}}, 255\right)$$

I used this code part to implement it. And the output picture is next to code.

```
# Define the vibrancy transformation function
def vibrancy_transformation_pix(input_pix_val:int, a:float, sigma:int=70)->float:
    x = input_pix_val
    return min(x + a*128*math.exp(-(x-128)**2/(2*sigma**2)), 255)

# Set the transformation parameter 'a' and apply it to the saturation plane
a = 0.5
new_spiderman_saturation_plane = np.zeros(spiderman_saturation_plane.shape, dtype=np.uint8)

# Apply vibrancy transformation to each pixel in the saturation plane
for i in range(spiderman_saturation_plane.shape[0]):
    for j in range(spiderman_saturation_plane.shape[1]):
        new_spiderman_saturation_plane[i][j] = vibrancy_transformation_pix(spiderman_saturation_plane[i][j], a=a)
```

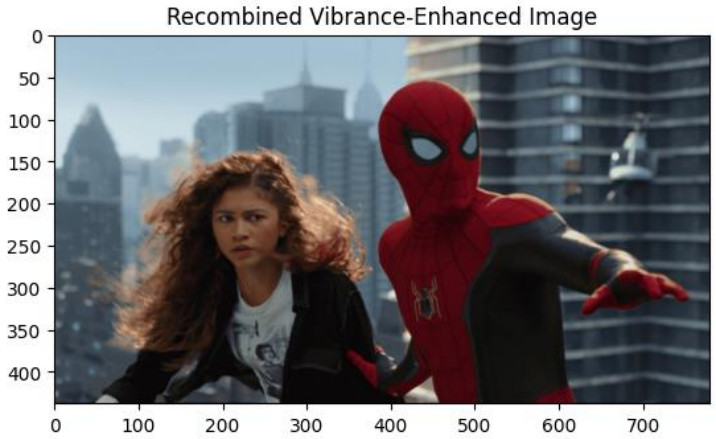
After applying the intensity transformation, the resultant saturation plane is merged back with the original hue and value planes, creating a vibrance-enhanced image. The comparison between the original and the transformed image showcases the effectiveness of the transformation, highlighting how colors are intensified and made more appealing. By carefully adjusting parameters like a and σ the transformation can be fine-tuned to achieve visually stunning results while maintaining the natural integrity of the image.



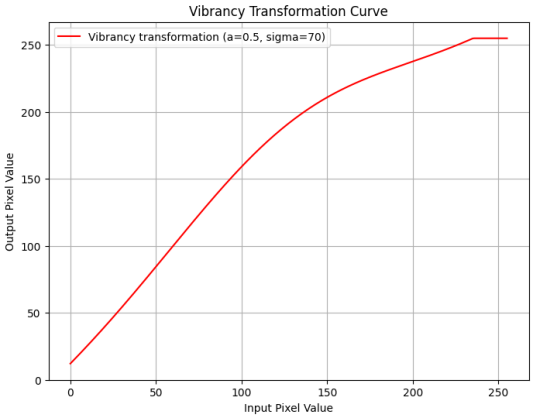
c) Adjusting a value as 0.2, 0.4, 0.6, 0.8, 1.0 and got different results and selected a good value for a to get a visually pleasing output. Through experimentation with different values of 'a', the optimal value determined was **0.5**. This value provided a visually pleasing enhancement to the saturation without overly distorting the image's natural colors.

```
# Apply vibrancy transformation to the saturation plane
new_spiderman_saturation_plane = np.zeros(spiderman_saturation_plane.shape, dtype=np.uint8)
for i in range(spiderman_saturation_plane.shape[0]):
    for j in range(spiderman_saturation_plane.shape[1]):
        new_spiderman_saturation_plane[i][j] = vibrancy_transformation_pix(spiderman_saturation_plane[i][j], a=a)

# Combine the new saturation plane with the original hue and value planes
new_hsv_spiderman_img = cv2.merge((spiderman_hue_plane, new_spiderman_saturation_plane, spiderman_value_plane))

# Convert the HSV image back to BGR color space
new_spiderman_img = cv2.cvtColor(new_hsv_spiderman_img, cv2.COLOR_HSV2BGR)

# Store the transformed image in a dictionary for visualization
hsv_images[a] = new_spiderman_img

# Recombine the hue, transformed saturation, and value planes
new_hsv_spiderman_img = cv2.merge((spiderman_hue_plane, new_spiderman_saturation_plane, spiderman_value_plane))

# Convert the recombined image back to BGR color space
new_spiderman_img = cv2.cvtColor(new_hsv_spiderman_img, cv2.COLOR_HSV2BGR)
```

d) The hue, adjusted saturation, and value planes were recombined into a final image, for a = 5, which retained the original hue and brightness while enhancing the vibrance through the saturation adjustments.



e) The transformation effectively enhances the vibrancy of the image, making colors appear more saturated and vivid. The adjustment of 'a' plays a crucial role in balancing vibrance and maintaining natural appearance. The plotted intensity transformation curve visually represents how input pixel values are altered, indicating the effectiveness of the transformation at different intensity levels.

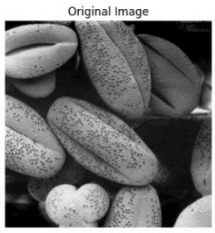


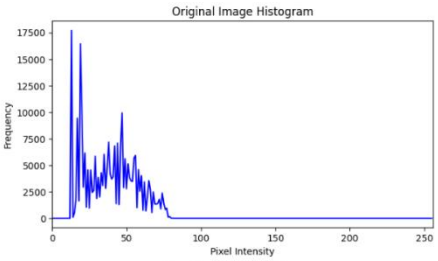
Q.05 : Histogram equalization is a powerful technique for enhancing the contrast of images, making features more visible. The function equalize_histogram encapsulate the essential steps in this process, from computing histograms and CDFs to mapping pixel values and then visualize the results. By redistributing pixel intensities, histogram equalization helps to improve the overall quality of images, especially those that are poorly contrasted or have limited dynamic range.

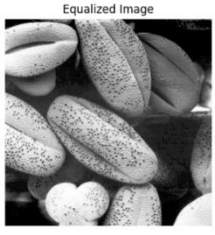
Function equalize_histogram:

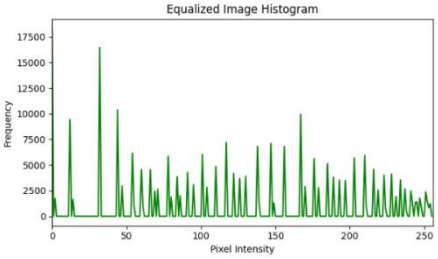
- This function accepts a grayscale image and applies histogram equalization.
- It computes the histogram and the cumulative distribution function (CDF), normalizes it, and uses the CDF to transform the original image.

```python
def equalize_histogram(gray_img):
    """Apply histogram equalization to the grayscale image."""
    # Compute histogram of the input image
    histogram, bin_edges = np.histogram(gray_img.flatten(), bins=256, range=[0, 256])

    # Compute the cumulative distribution function (CDF)
    cdf_values = histogram.cumsum()

    # Mask zero intensity values and normalize the CDF
    cdf_masked = np.ma.masked_equal(cdf_values, 0)
    cdf_masked = (cdf_masked - cdf_masked.min()) * 255 / (cdf_masked.max() - cdf_masked.min())
    cdf_normalized = np.ma.filled(cdf_masked, 0).astype('uint8')

    # Map the CDF values back to the grayscale image
    img_equalized = cdf_normalized[gray_img]

    return img_equalized, histogram

# Example usage with a grayscale image
image= "/content/drive/MyDrive/aiimages/shells.tif"
grayscale_image = cv2.imread(image, cv2.IMREAD_GRAYSCALE)

# Apply histogram equalization to the image
equalized_image_result, original_hist_result = equalize_histogram(grayscale_image)

# Compute histogram for the equalized image
eq_histogram, _ = np.histogram(equalized_image_result.flatten(), bins=256, range=[0, 256])
```



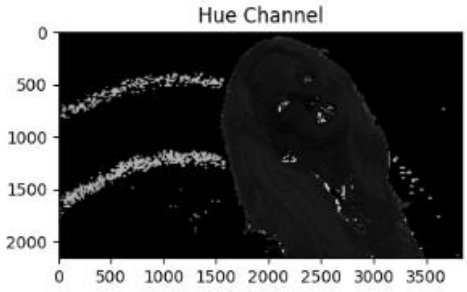


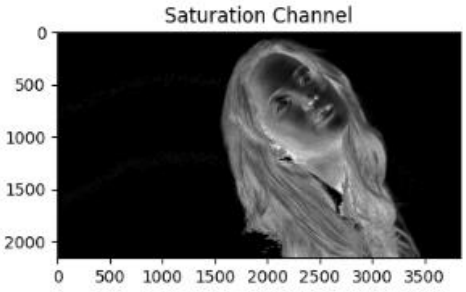


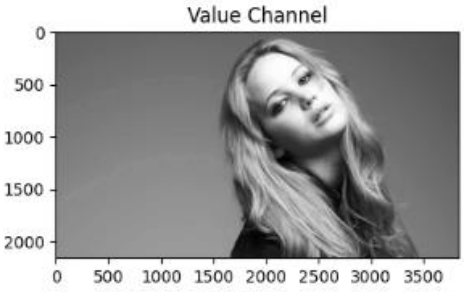


Q.06 : a) This code part is really near to first part of 4th question. The code part and result is given here.

```
# Split HSV channels
hue_channel = hsv_image[:, :, 0]
saturation_channel = hsv_image[:, :, 1]
value_channel = hsv_image[:, :, 2]
```







b) In the HSV color space, the saturation channel indicates the intensity of color. Higher saturation values correspond to more vivid colors, while lower saturation values are associated with grayscale or washed-out colors. By thresholding the saturation channel, I can effectively distinguish colorful foreground objects from a potentially background that is more washed out or less colorful. The threshold value (14) should be chosen based on the specific image and the color characteristics of the foreground and background.

```
# Create a binary mask for the foreground based on saturation threshold
foreground_mask = saturation_channel > 14

# Display the foreground mask
plt.figure(figsize=(5, 5))
plt.imshow(foreground_mask, cmap='gray')
plt.title('Foreground Mask')
plt.axis('off')
plt.show()
```

c) This code obtain the foreground only using cv.bitwise_and and compute the histogram. First line uses the bitwise AND operation to apply the foreground_mask to the original image. Only the pixels corresponding to True (or 1) values in the mask will be retained in foreground_only, effectively isolating the foreground.

Next three lines compute the histograms for the Blue, Green, and Red channels of the foreground_only image. The histograms show the distribution of pixel intensities for each color channel, which helps analyze the color composition of the extracted foreground. Last line set is used for represent the extracted foreground.

```
# Extract the foreground using bitwise AND
foreground_only = cv2.bitwise_and(image, image, mask=foreground_mask.astype(np.uint8))

# Compute histogram for each channel of the foreground
hist_blue = cv2.calcHist([foreground_only], [0], foreground_mask.astype(np.uint8), [256], [0, 256])
hist_green = cv2.calcHist([foreground_only], [1], foreground_mask.astype(np.uint8), [256], [0, 256])
hist_red = cv2.calcHist([foreground_only], [2], foreground_mask.astype(np.uint8), [256], [0, 256])

# Display the extracted foreground
plt.figure(figsize=(5, 5))
plt.imshow(cv2.cvtColor(foreground_only, cv2.COLOR_BGR2RGB))
plt.title('Extracted Foreground')
plt.axis('off')
plt.show()
```

d) This code part obtain the cumulative sum of the histogram using np.cumsum. computing the cumulative sum of the histogram is essential for transforming pixel intensity values, normalizing the distribution, and enhancing the contrast of images, ultimately leading to better visual quality and more informative images. In here I only represented the cumulative histogram for blue channel as an example.

```
# Calculate cumulative histograms
cumulative_hist_blue = np.cumsum(hist_blue)
cumulative_hist_green = np.cumsum(hist_green)
cumulative_hist_red = np.cumsum(hist_red)

# Print cumulative histogram for blue channel as an example
print("Cumulative Histogram Blue: \n", cumulative_hist_blue)
```

```
Cumulative Histogram Blue:
 [  26760.   33302.   42469.   52584.   61324.   73001.   82935.   95310.
  108524.  122178.  141746.  155494.  188722.  204406.  225838.  254304.
  276505.  301898.  320269.  352247.  371576.  395921.  419858.  441229.
  466608.  486521.  510361.  533253.  553950.  578691.  598725.  622017.
  642357.  663577.  685689.  706038.  727353.  746859.  768658.  788359.
  807603.  828442.  847972.  868143.  887213.  906423.  926086.  944983.
  964047.  982725. 1001465. 1019970. 1038253. 1056264. 1074626. 1092491.
 1110056. 1127489. 1144977. 1161866. 1178810. 1195611. 1212137. 1228870.
 1244954. 1261530. 1277489. 1293714. 1309626. 1325563. 1341265. 1357094.
 1372715. 1388053. 1403475. 1419262. 1434906. 1450662. 1466241. 1481641.
 1496705. 1511708. 1526925. 1541820. 1556876. 1571814. 1586639. 1601302.
 1615932. 1630585. 1645205. 1659880. 1674762. 1689178. 1703857. 1718639.
 1733356. 1748079. 1762526. 1777432. 1791892. 1806465. 1820971. 1835534.
 1850217. 1864521. 1878500. 1892475. 1906397. 1920046. 1933550. 1947047.
 1960418. 1973460. 1986526. 1999628. 2012279. 2024405. 2036359. 2048242.
 2060020. 2071475. 2082962. 2094319. 2105383. 2116608. 2127400. 2138163.
 2148623. 2159079. 2169388. 2179509. 2189597. 2199438. 2208996. 2218527.
 2227853. 2237094. 2246251. 2255384. 2264244. 2272926. 2281356. 2289818.
 2298081. 2306028. 2314072. 2321869. 2329506. 2337067. 2344226. 2351367.
 2358447. 2365241. 2372048. 2378583. 2385046. 2391174. 2397132. 2403000.
 2408655. 2414175. 2419751. 2425118. 2430340. 2435687. 2440771. 2445697.
 2450319. 2454850. 2459322. 2463566. 2468260. 2472443. 2476478. 2480589.
 2484639. 2488901. 2492708. 2496705. 2500458. 2504359. 2508108. 2511687.
 2516099. 2519680. 2523552. 2527600. 2532026. 2535818. 2539397. 2544147.
 2547358. 2552125. 2556065. 2560864. 2565325. 2568915. 2573826. 2577132.
 2582367. 2585804. 2589235. 2595074. 2598627. 2603129. 2606292. 2610106.
 2614235. 2618122. 2621671. 2624355. 2629727. 2631966. 2636375. 2638626.
 2644014. 2646708. 2648852. 2653609. 2655017. 2659693. 2661553. 2663123.
 2670879. 2672184. 2681009. 2682081. 2690764. 2691683. 2694789. 2696285.
 2700928. 2701560. 2701841. 2705574. 2705742. 2706549. 2706658. 2706692.
 2706725. 2706728. 2706729. 2706730. 2706730. 2706730. 2706730. 2706730.
 2706730. 2706730. 2706730. 2706730. 2706730. 2706730. 2706730. 2706730.]
```

e)

```
def equalize_cumulative_histogram(cum_hist, total_pixels):
    return ((cum_hist / total_pixels) * 255).astype(np.uint8)

# Get total number of foreground pixels
total_pixels = foreground_mask.astype(np.uint8).sum()

# Create lookup tables for each channel
lookup_table_blue = equalize_cumulative_histogram(cumulative_hist_blue, total_pixels)
lookup_table_green = equalize_cumulative_histogram(cumulative_hist_green, total_pixels)
lookup_table_red = equalize_cumulative_histogram(cumulative_hist_red, total_pixels)

# Apply equalization to the foreground
equalized_foreground = np.zeros_like(foreground_only)
equalized_foreground[:, :, 0] = cv2.LUT(foreground_only[:, :, 0], lookup_table_blue)
equalized_foreground[:, :, 1] = cv2.LUT(foreground_only[:, :, 1], lookup_table_green)
equalized_foreground[:, :, 2] = cv2.LUT(foreground_only[:, :, 2], lookup_table_red)

# Display equalized foreground
plt.figure(figsize=(5, 5))
plt.imshow(cv2.cvtColor(equalized_foreground, cv2.COLOR_BGR2RGB))
plt.title('Equalized Foreground')
plt.axis('off')
plt.show()
```

f) Finally, extract the background and add with the histogram equalized foreground using below code part and given this last result.

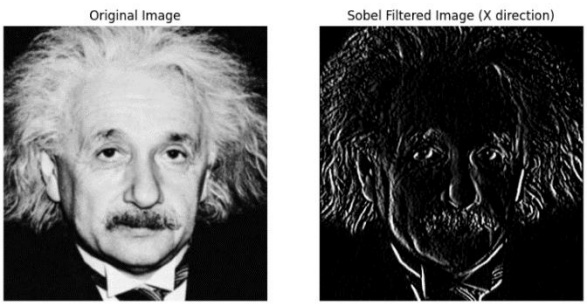
```
# Extract the background from the original image using the mask
background = cv2.bitwise_and(image, image, mask=cv2.bitwise_not(foreground_mask.astype(np.uint8)))

# Combine equalized foreground with original background
final_result = cv2.add(equalized_foreground, background)
```

Q.07) a) The filter2D function in OpenCV applies the Sobel operator, which computes the gradient of the image in the x-direction. The Sobel kernel detects edges by calculating the change in intensity between adjacent pixels. This operation enhances vertical features in the image, making edges more prominent.

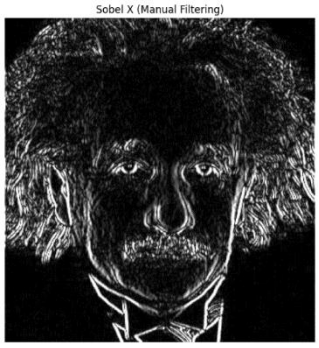
b) Manually implementing the Sobel filter involves convolving the Sobel kernel with the image. This process extracts the gradient by applying the kernel to each pixel, considering its neighbors. The manual approach helps to understand the underlying mechanics of convolution and edge detection, illustrating how pixel values contribute to the resulting gradient image.

c) The Sobel operator can be separated into two 1D convolutions: a vertical kernel for smoothing followed by a horizontal edge detection kernel. This property reduces computational complexity by breaking down the 2D convolution into two sequential 1D convolutions, allowing for more efficient processing while achieving the same edge detection effect.

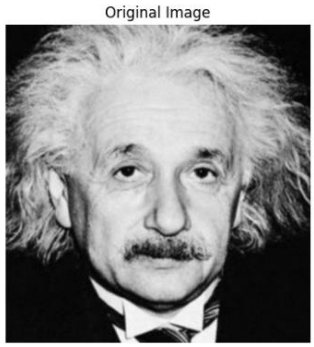
```
# Define the Sobel kernel for x direction
sobel_kernel_x = np.array([[-1, 0, 1],
                           [-2, 0, 2],
                           [-1, 0, 1]])

# Apply the Sobel filter using filter2D
sobel_filtered_x = cv2.filter2D(einstein_img, -1, sobel_kernel_x)
```

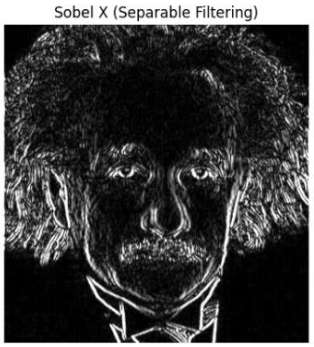



```
# Initialize empty arrays to store Sobel results
sobel_x_manual = np.zeros_like(einstein_img, dtype=np.float64)

# Pad the image to handle edges
padded_img = np.pad(einstein_img, ((1, 1), (1, 1)), mode='constant')

# Perform convolution with the Sobel kernel manually
height, width = einstein_img.shape
for row in range(1, height + 1):
    for col in range(1, width + 1):
        # Extract the 3x3 region from the padded image
        region = padded_img[row-1:row+2, col-1:col+2]
        # Apply the Sobel kernel
        sobel_x_manual[row-1, col-1] = np.sum(sobel_kernel_x * region)

# Convert result to uint8 format
sobel_x_manual = np.uint8(np.clip(np.abs(sobel_x_manual), 0, 255))
```



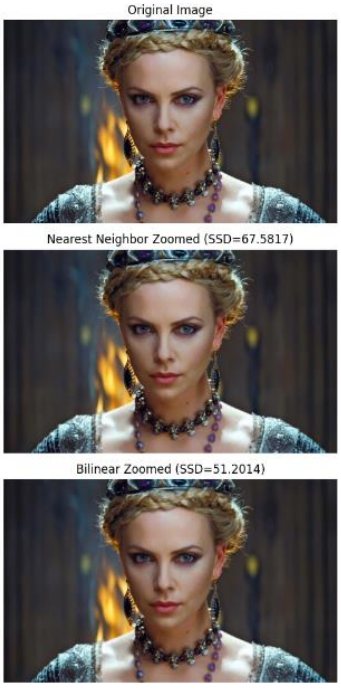
```python
# Define the 1D Sobel kernels
sobel_vertical = np.array([1, 2, 1])  # Vertical kernel for smoothing
sobel_horizontal = np.array([1, 0, -1])  # Horizontal kernel for edge detection

# Step 1: Apply vertical kernel to the image
sobel_intermediate = np.zeros_like(einstein_img, dtype=np.float64)
padded_img = np.pad(einstein_img, ((1, 1), (0, 0)), mode='constant')

for row in range(1, height + 1):
    for col in range(width):
        sobel_intermediate[row-1, col] = np.sum(padded_img[row-1:row+2, col] * sobel_vertical)

# Step 2: Apply horizontal kernel to the intermediate result
sobel_x_separable = np.zeros_like(einstein_img, dtype=np.float64)
padded_intermediate = np.pad(sobel_intermediate, ((0, 0), (1, 1)), mode='constant')

for row in range(height):
    for col in range(1, width + 1):
        sobel_x_separable[row, col-1] = np.sum(padded_intermediate[row, col-1:col+2] * sobel_horizontal)

# Convert result to uint8 format and clip values
sobel_x_separable = np.uint8(np.clip(np.abs(sobel_x_separable), 0, 255))
```






```
Sobel X Kernel:
[[ 1  0 -1]
 [ 2  0 -2]
 [ 1  0 -1]]
```

Q.08) This part implements image zooming by a specified factor using two interpolation methods: nearest-neighbor and bilinear interpolation. The `zoom_image` function scales the input image and applies the chosen interpolation method. The program then computes the normalized sum of squared differences (SSD) between the zoomed images and their corresponding original images to assess the quality of the zooming. By comparing the two interpolation methods, the program provides insights into their effectiveness in preserving image quality during enlargement. Finally, the results, including zoomed images and SSD values, are displayed to facilitate visual comparison and analysis.



(Nearest Neighbor): 136.2691	(Nearest Neighbor): 26.4461	(Nearest Neighbor): 67.5817	(Nearest Neighbor): 228.6223
(Bilinear): 115.0919	(Bilinear): 18.3459	(Bilinear): 51.2014	(Bilinear): 197.4095

```python
# Function to zoom the image using specified interpolation method
def zoom_image(image, scale_factor, interpolation_method):
    new_size = (int(image.shape[1] * scale_factor), int(image.shape[0] * scale_factor))
    return cv2.resize(image, new_size, interpolation=interpolation_method)

# Function to compute the normalized sum of squared differences (SSD)
def compute_normalized_ssd(image1, image2):
    # Ensure the images are of the same size
    if image1.shape != image2.shape:
        min_height = min(image1.shape[0], image2.shape[0])
        min_width = min(image1.shape[1], image2.shape[1])
        image1 = image1[:min_height, :min_width]
        image2 = image2[:min_height, :min_width]

    ssd = np.sum((image1.astype(np.float32) - image2.astype(np.float32)) ** 2)
    norm_ssd = ssd / np.prod(image1.shape)
    return norm_ssd
```

```python
# Function to process zooming for given images
def process_zoom_and_display(small_image, large_image, scale_factor=4):
    # Zoom using nearest-neighbor interpolation
    nn_zoomed = zoom_image(small_image, scale_factor, cv2.INTER_NEAREST)

    # Zoom using bilinear interpolation
    bilinear_zoomed = zoom_image(small_image, scale_factor, cv2.INTER_LINEAR)

    # Compute normalized SSD for both zoomed images
    ssd_nn = compute_normalized_ssd(large_image, nn_zoomed)
    ssd_bilinear = compute_normalized_ssd(large_image, bilinear_zoomed)

    # Print out SSD values
    print(f"Normalized SSD (Nearest Neighbor): {ssd_nn:.4f}")
    print(f"Normalized SSD (Bilinear): {ssd_bilinear:.4f}")

    # Display images
    titles = [
        "Original Image",
        f"Nearest Neighbor Zoomed (SSD={ssd_nn:.4f})",
        f"Bilinear Zoomed (SSD={ssd_bilinear:.4f})"
    ]
    display_images(large_image, nn_zoomed, bilinear_zoomed, titles)
```

Q.09) a) The GrabCut algorithm is a powerful image segmentation technique that separates the foreground from the background by iteratively refining an initial mask. It uses a graph-based approach to model the image pixels, classifying them as foreground or background based on color similarity and spatial information. In this task, GrabCut effectively segments the flower image by defining a rectangle around the object, allowing the algorithm to learn the characteristics of the flower and accurately extract it while leaving the background distinct.

b) The code applies Gaussian blur to the background, creating a more visually appealing image with a blurred backdrop. The enhanced image is then combined with the sharp foreground to emphasize the flower, showcasing the contrast between the clear subject and the softened background.


```python
# Convert the image from BGR to RGB color space
daisy_image_rgb = cv2.cvtColor(daisy_image, cv2.COLOR_BGR2RGB)

# Create an initial mask for GrabCut
initial_mask = np.zeros(daisy_image.shape[:2], np.uint8)

# Create models for background and foreground
bg_model = np.zeros((1, 65), np.float64)
fg_model = np.zeros((1, 65), np.float64)

# Define the rectangle for the GrabCut algorithm
rect = (50, 50, daisy_image.shape[1] - 50, daisy_image.shape[0] - 50)  # Adjusted rectangle size

# Apply the GrabCut algorithm
cv2.grabCut(daisy_image, initial_mask, rect, bg_model, fg_model, 5, cv2.GC_INIT_WITH_RECT)

# Update the mask to segment the foreground
segmented_mask = np.where((initial_mask == 2) | (initial_mask == 0), 0, 1).astype('uint8')

# Extract the segmented foreground image
foreground_image = daisy_image_rgb * segmented_mask[:, :, np.newaxis]

# Extract the background image
background_image = daisy_image_rgb * (1 - segmented_mask[:, :, np.newaxis])
```

```python
# Apply Gaussian blur to the background image
blurred_background_image = cv2.GaussianBlur(background_image, (55, 55), 0)

# Combine the sharp foreground with the blurred background
enhanced_image_blurred_bg = np.where(segmented_mask[:, :, np.newaxis] == 1, daisy_image_rgb, blurred_background_image)
```

c) The dark background beyond the edge of the flower in the enhanced image can be attributed to several interconnected factors. Firstly, the Gaussian blur applied to the background reduces its detail and sharpness, creating a softer and less defined area. This loss of clarity can lead to a perception of darkness, as blurred regions lack the vibrancy and texture of in-focus areas.

Secondly, the original lighting conditions of the image may have created a gradient in brightness, causing certain areas to appear darker. If the background was illuminated unevenly, it could lead to variations in color and intensity, particularly in the regions just behind the flower.

Lastly, the vibrant colors of the flower contrast sharply with the muted tones of the blurred background. This color contrast emphasizes the darker shades within the background, making them more pronounced. Together, these factors create a striking visual impact, drawing attention to the flower while allowing the background to recede into shadow.