EN3160: Assignment 01
Intensity Transformations and Neighborhood Filtering

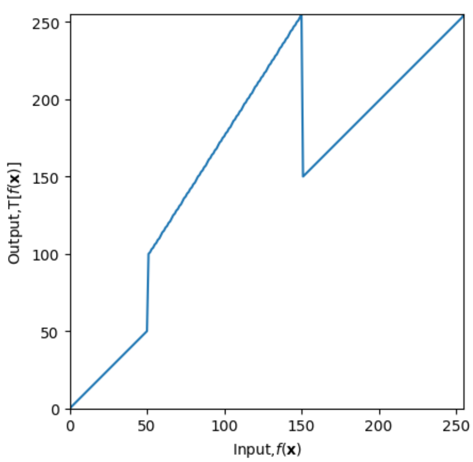
Index No. : 200462U

Name : Randika Perera

GitHub : Link to Source Code

Question 1: Intensity Transformation

```
t1 = np.linspace(0,50,50-0+1).astype('uint8')
t2 = np.linspace(100,255,150-50).astype('uint8')
t3 = np.linspace(150,255,255-150).astype('uint8')
 :
transform = np.concatenate((t1,t2),axis=0).astype('uint8')
transform = np.concatenate((transform,t3),axis=0).astype('uint8')
 :
img_orig = cv.imread('emma.jpg',cv.IMREAD_GRAYSCALE)
 :
image_transformed = cv.LUT(img_orig, transform)
```

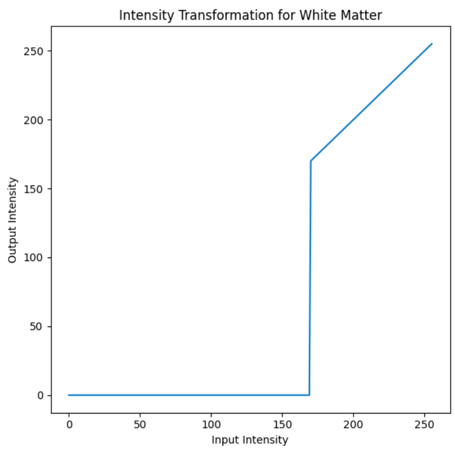


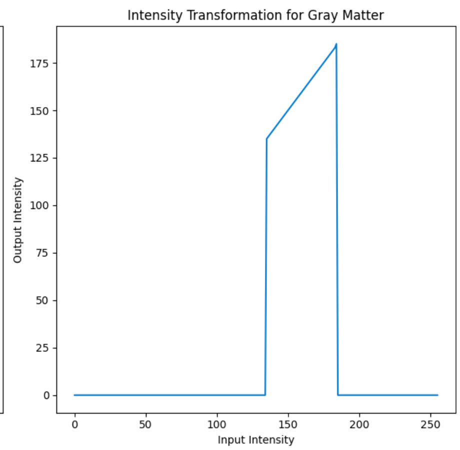
The given intensity transform is a unity transform from 0 to 50 (low intensity: close to black) and from 150 to 255 (high intensity: close to white). So, pixel intensities that fall within these 2 ranges will remain unchanged.

But for pixel intensities in the range 50 to 150, the pixel intensities have been shifted upwards. So, pixels of the image that fall within this range will appear brighter than in the original image.

Question 2: Intensity Transformation

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

# Transformation to Extract White Matter
t_white_threshold = 170
t_white_matter = np.linspace(0, 255, 256, dtype='uint8')
t_white_matter[:t_white_threshold] = 0
t_white_matter[t_white_threshold:] = np.linspace(t_white_threshold, 255, 255-t_white_threshold+1, dtype='uint8')
# The threshold value 170 was selected after trial and error

# Transformation to Extract Gray Matter
t_grey_threshold_lower = 135
t_grey_threshold_upper = 185
t_gray_matter = np.linspace(0, 255, 256, dtype='uint8')
t_gray_matter[:t_grey_threshold_lower] = 0
t_gray_matter[t_grey_threshold_upper:] = 0
t_gray_matter[t_grey_threshold_lower:t_grey_threshold_upper] = np.linspace(t_grey_threshold_lower, t_grey_threshold_upper,
t_grey_threshold_upper-t_grey_threshold_lower, dtype='uint8')
# The threshold values 135 and 185 were selected after trial and error

image_gray_matter  = cv.LUT(img_brain, t_gray_matter)
image_white_matter = cv.LUT(img_brain, t_white_matter)
```



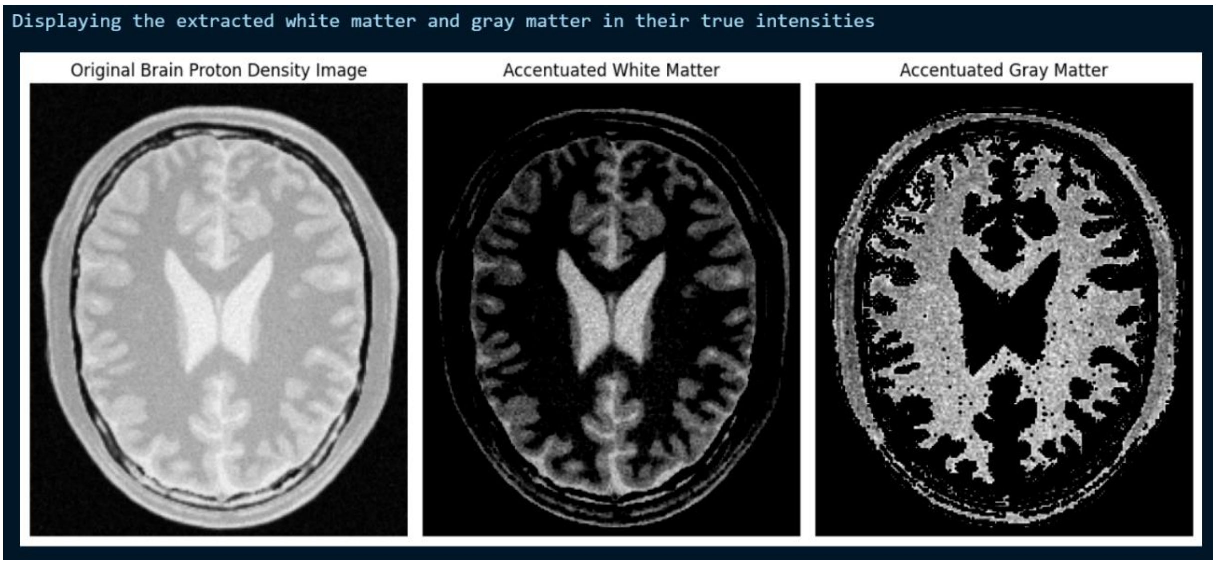


To extract white matter, I started with a unity transformation and then I made pixel intensities of all pixels below a certain threshold as zero.

To extract gray matter, I started with a unity transformation and then I made pixel intensities of all pixels above and below 2 thresholds as zero.

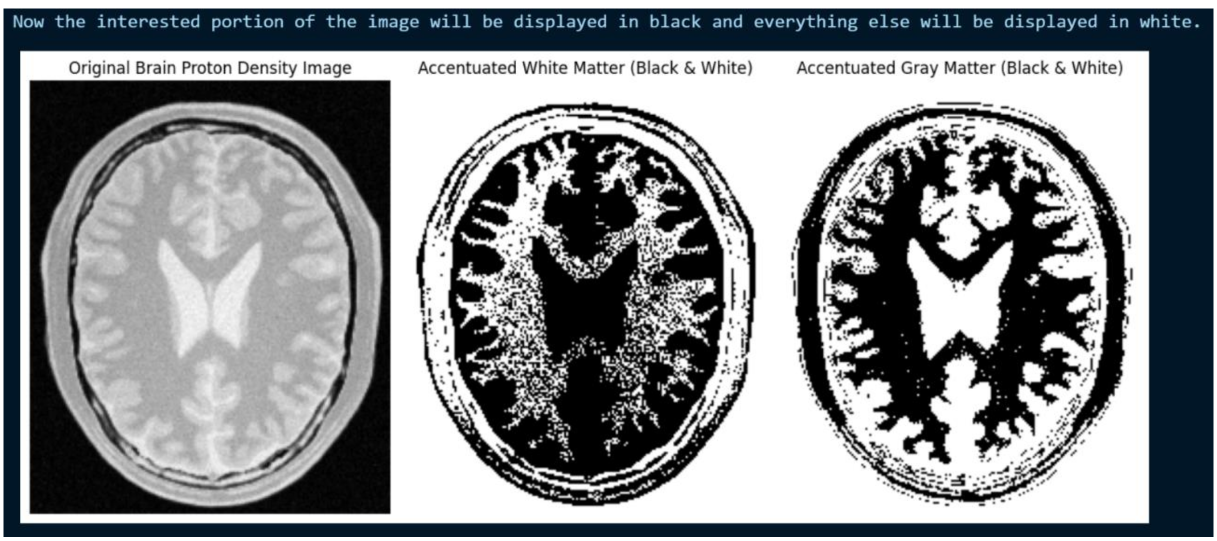
The threshold values for cutting out white matter and gray matter were selected the through trial and error. The values were selected such that the white matter and gray matter were clearly visible. The values selected are as follows,

- White Matter: Lower=170, Upper=255
- Gray Matter : Lower=135, Upper=185



Further, to visualize the interested portion of the image as a black image on a white background, I used an inverted threshold and converted the above images into black and white.

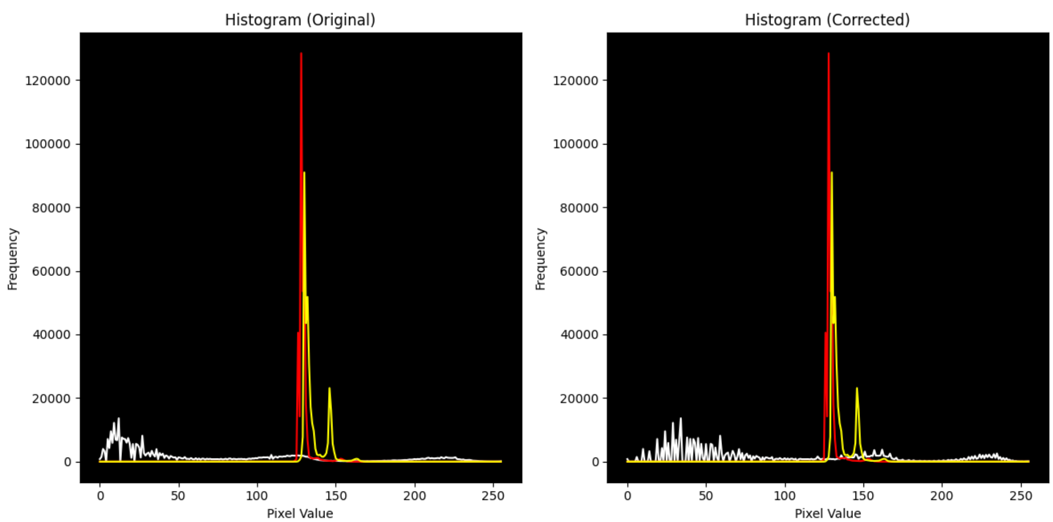
```
bw_image_white_matter = cv.threshold(image_white_matter, 1, 255, cv.THRESH_BINARY_INV)[1]    # Inverted threshold
bw_image_gray_matter  = cv.threshold(image_gray_matter, 1, 255, cv.THRESH_BINARY_INV)[1]     # Inverted threshold
```



Question 3: Gamma Correction

```
image_lab = cv.cvtColor(image, cv.COLOR_BGR2LAB)        # Convert BGR to LAB format as mentioned in the question
l_channel, a_channel, b_channel = cv.split(image_lab)   # Split the LAB image into its channels
 :
gamma = 0.65
table = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0, 256)]).astype('uint8')
l_channel_corrected = cv.LUT(l_channel, table)                                 # Apply gamma correction to the L plane using the
lookup table
image_lab_corrected = cv.merge((l_channel_corrected, a_channel, b_channel))    # Merge the corrected L channel with the A and B
channels
 :
space = ('l', 'a', 'b')                    #3 channels in LAB space
colour = ('white', 'red', 'yellow')        #3 colours that will be used to plot the histograms
# White for L channel, Red for A channel and Yellow for B channel

for i, s in enumerate(space):
# i will get values 0,1,2 and s will get values l,a,b.

    hist_orig = cv.calcHist([image_lab], [i], None, [256], [0, 256])             # Calculate histogram from original LAB image
    axarr[0].plot(hist_orig, color=colour[i])

    hist_gamma = cv.calcHist([image_lab_corrected], [i], None, [256], [0, 256])  # Calculate histogram from corrected LAB image
    axarr[1].plot(hist_gamma, color=colour[i])
```

Selected value for $\gamma = 0.65$



- After trying gamma correction for several values of gamma, 0.65 was selected as the best value.
- Note that we applied gamma correction only to the L channel.
- So only the histogram of the L channel (white graph above) has changed.
- The histograms of the other two channels remain unchanged.

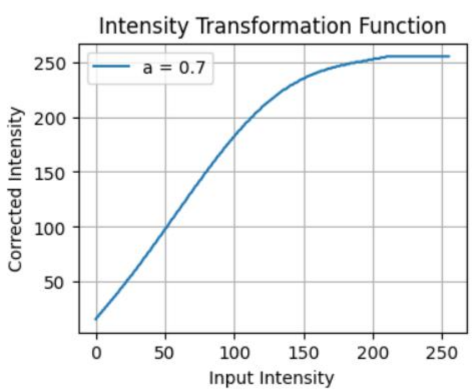
Question 4: Increasing Vibrance using the Given Intensity Transformation

For this question, I will use an interactive slider to change the value of a and update the transformed image in real time in order to select the best value for a.

```python
from ipywidgets import interactive, FloatSlider
 :
def vibrance(x,a):
    return int(min(x+a*128*np.exp(-(x-128)**2/(2*70**2)),255))
 :
def spider_update_image(a):
    plt.clf()
    table = np.array([vibrance(x,a) for x in np.arange(0, 256)]).astype('uint8')
    int_s_channel_corrected = cv.LUT(int_s_channel, table)
    # Apply vibrance correction to the S plane using the lookup table
    int_spider_image_hsv_corrected = cv.merge((int_h_channel, int_s_channel_corrected, int_v_channel))
    # Merge the corrected S channel with the H and V channels
    int_spider_image_hsv_corrected_in_RGB = cv.cvtColor(int_spider_image_hsv_corrected, cv.COLOR_HSV2RGB)
    # Convert HSV to RGB for displaying using matplotlib
 :
int_spider_image = cv.imread('spider.png', cv.IMREAD_COLOR)                # Open CV will read image in BGR format
 :
int_spider_image_hsv = cv.cvtColor(int_spider_image, cv.COLOR_BGR2HSV)         # Convert BGR to HSV format
 :
int_h_channel, int_s_channel, int_v_channel = cv.split(int_spider_image_hsv)   # Split the HSV image into its channels
 :
a_slider = FloatSlider(value=0.5, min=0, max=1, step=0.02)
interactive_plot = interactive(spider_update_image, a=a_slider)
```

a 0.70

```
<Figure size 640x480 with 0 Axes>
```



The interactive slider for a was adjusted until a visually pleasing output was obtained as the corrected image.

The best value for a seems to be around 0.7.

Question 5: Histogram Equalization

```python
img = cv.imread('shells.tif', cv.IMREAD_GRAYSCALE)
 :
hist, bins = np.histogram(img.ravel(), 256, [0, 256])
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max() / cdf.max()
 :
plt.plot(cdf_normalized, color='b')
plt.hist(img.flatten(), 256, [0, 256], color='r')
 :
equ = cv.equalizeHist(img)
hist, bins = np.histogram(equ.ravel(), 256, [0, 256])
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max() / cdf.max()
 :
```

```python
def custom_equalize_hist(image):

    # Calculate histogram and CDF. This is same as when
      using inbuilt function cv.equalizeHist()
    histogram, bins = np.histogram(image.flatten(),
    bins=256, range=(0, 256))
    cdf = histogram.cumsum()
    cdf_normalized = cdf * histogram.max() / cdf.max()

    # Instead of using inbuilt function cv.equalizeHist(),
      we will use the CDF calculated above to get the
      equalized image
    # We will use linear interpolation (np.interp()) to get
      the new pixel values
```

```
plt.plot(cdf_normalized, color='b')
plt.hist(equ.flatten(), 256, [0, 256], color='r')
 :
```

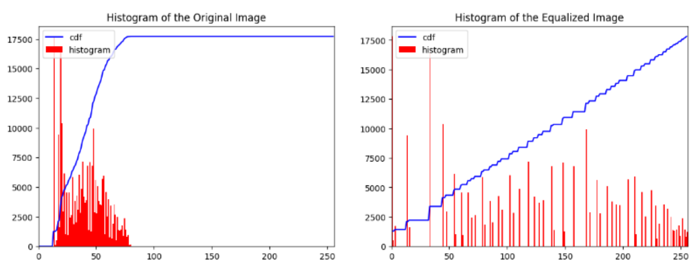
The code on the left uses OpenCV's inbuilt function cv.equalizeHist() for histogram equalization.

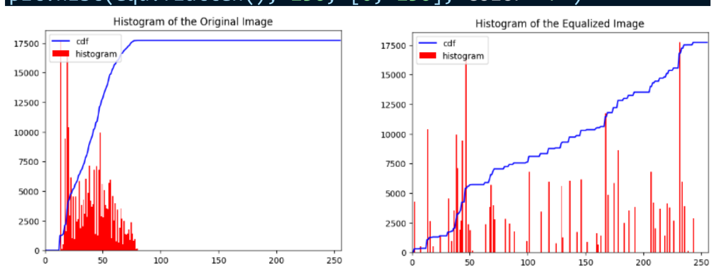
The code on the right uses a custom function for histogram equalization. It uses linear interpolation to distribute the normalized CDF values across the intensity levels and applied it the image using a lookup table.

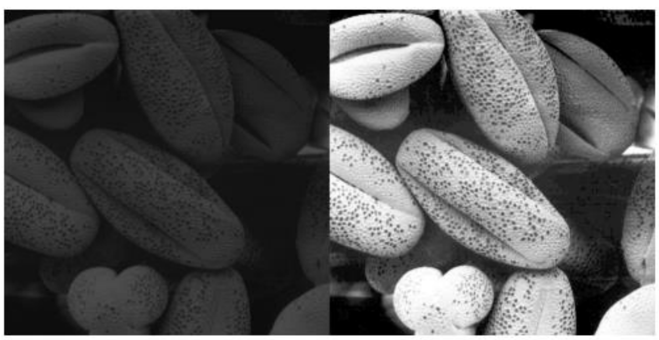
We can observe that the custom function does not provide the same results as the inbuilt function. This may be due to a number of reasons such as,

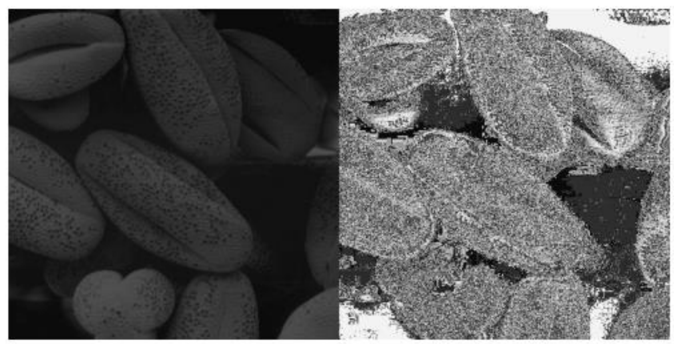
- Inbuilt function using more sophisticated normalization and mapping of the intensity levels to achieve a balanced distribution.
- Inbuilt function using additional techniques to smooth the cumulative distribution function (CDF).

```
    lookup_table = np.interp(image.flatten(), bins[:-1],
    cdf_normalized)

    # Reshape the equalized_image array into the same shape
      as the original image.
    equalized_image =
    lookup_table.reshape(image.shape).astype(np.uint8)
    return equalized_image
 :
img = cv.imread('shells.tif', cv.IMREAD_GRAYSCALE)
 :
hist, bins = np.histogram(img.ravel(), 256, [0, 256])
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max() / cdf.max()
plt.plot(cdf_normalized, color='b')
plt.hist(img.flatten(), 256, [0, 256], color='r')
 :
equ = custom_equalize_hist(img)
hist, bins = np.histogram(equ.ravel(), 256, [0, 256])
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max() / cdf.max()
plt.plot(cdf_normalized, color='b')
plt.hist(equ.flatten(), 256, [0, 256], color='r')
```



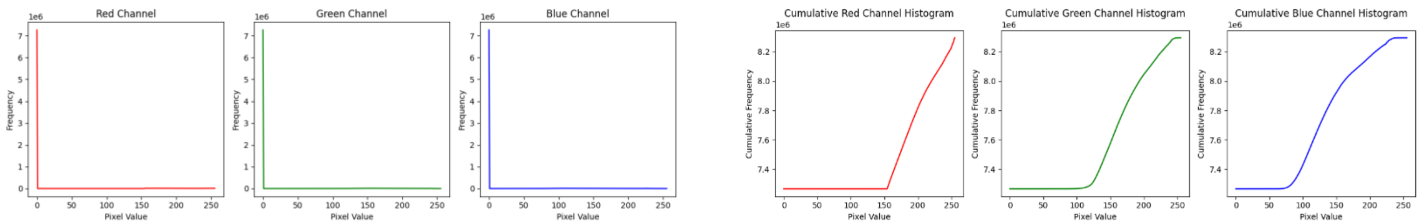


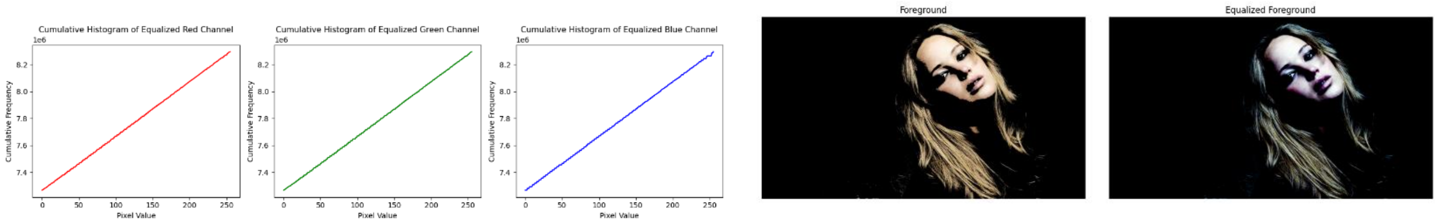


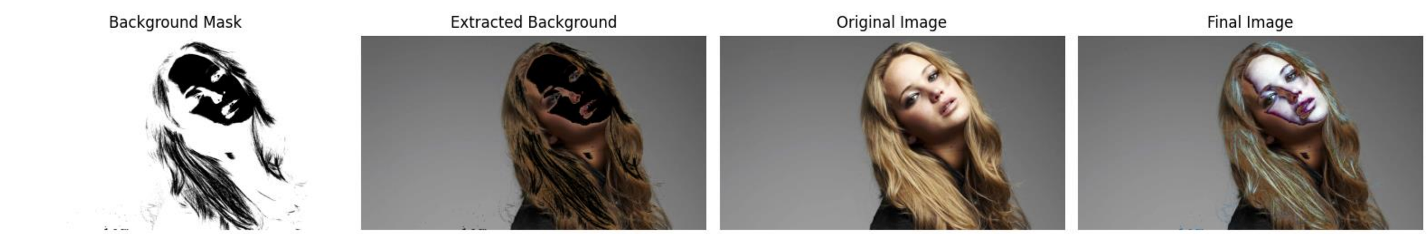


Question 6: Histogram Equalization of Foreground

```
image = cv.imread('jeniffer.jpg', cv.IMREAD_COLOR)      # Open CV will read image in BGR format
hsv_image = cv.cvtColor(image, cv.COLOR_BGR2HSV)        # Convert the image to HSV color space
hue, saturation, value = cv.split(hsv_image)            # Split the HSV image into hue, saturation, and value planes\
 :
threshold = 154
# This threshold value was selected based on trial and error to get the best possible mask from the value plane
 :
_,mask_1 = cv.threshold(hue, threshold, 255, cv.THRESH_BINARY)          # Thresholding the hue plane
_,mask_2 = cv.threshold(saturation, threshold, 255, cv.THRESH_BINARY)   # Thresholding the saturation plane
_,mask_3 = cv.threshold(value, threshold, 255, cv.THRESH_BINARY)        # Thresholding the value plane
 :
selected_mask = mask_3   # Select the mask obtained from value plane
 :
foreground = cv.bitwise_and(image, image, mask=selected_mask)
 :
# Histogram equalization for each color channel
equalized_r = cv.equalizeHist(foreground[:, :, 0])
equalized_g = cv.equalizeHist(foreground[:, :, 1])
equalized_b = cv.equalizeHist(foreground[:, :, 2])
 :
equalized_image = cv.merge((equalized_r, equalized_g, equalized_b)) # Merge the equalized channels into an equalized image
 :
background = cv.bitwise_and(image, image, mask=cv.bitwise_not(selected_mask))
 :
hist_equalized_image = cv.add(background, equalized_image)  # Add equalized foreground to background
```







Final Image = Extracted Background + Equalized Foreground

Question 7: Sobel Filtering

(a) Using the existing Filter2D to Sobel Filter the image

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

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

# Using the existing filter2D to Sobel filter the image
A_imgc   = cv.filter2D(A_image,-1,A_kernel)
```

(b) Using own code to Sobel Filter the image

```
# Define the convolution function
def B_Convolution(image, kernel):
    kernel_size = kernel.shape[0]
    image_height, image_width = image.shape
    output_img = np.zeros((image_height - kernel_size + 1, image_width - kernel_size + 1))
    for i in range(output_img.shape[0]):
        for j in range(output_img.shape[1]):
            output_img[i, j] = np.sum(image[i:i+kernel_size, j:j+kernel_size] * kernel)
            # Apply pixel value mapping to achieve black and white appearance
            if output_img[i, j] < 0:       # If pixel value is less than 0, make it 0
                output_img[i, j] = 0
            elif output_img[i, j] > 255:   # If pixel value is greater than 255, make it 255
                output_img[i, j] = 255
    return output_img.astype(np.uint8)      # Return the output image as uint8 type
```

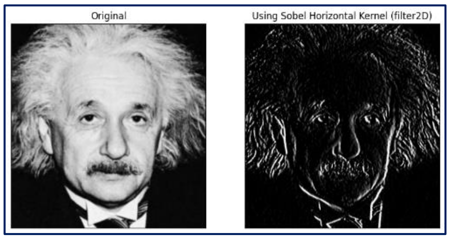
```
B_image = cv.imread('einstein.png', cv.IMREAD_GRAYSCALE)
B_kernel = np.array([(1, 0, -1), (2, 0, -2), (1, 0, -1)], dtype='float')
B_imgc = B_Convolution(B_image, B_kernel)
```

(c) Using the Property of Convolution to Carry out the Sobel Filtering

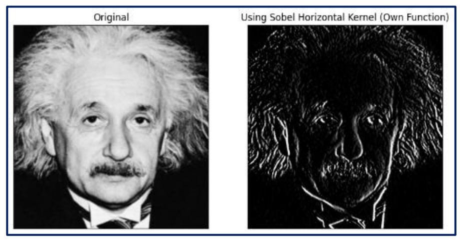
Since convolution is associative: image x (column x row) = (image x column) x row

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

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

C_imagec_1 = cv.filter2D(C_image   ,-1,C_column_kernel)    # Apply the column kernel first
C_imagec_2 = cv.filter2D(C_imagec_1,-1,C_row_kernel   )    # Apply the row kernel next
```

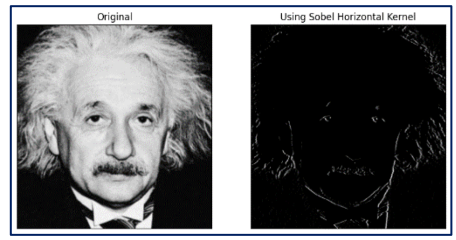
(a) Using the existing Filter2D to Sobel Filter the image



(b) Using own code to Sobel Filter the image



(c) Using the Property of Convolution (Associativity)



Question 8: Zooming

Zooming will be done using 2 methods, Nearest Neighbour method and Bilinear Interpolation. I will run the above 2 methods on 2 of the given set of images and compare the zoomed version with the original image using the normalized sum of squared differences.

(a) Nearest-Neighbour

```
def zoom_image(original_image,zooming_factor):
    height, width, channels = original_image.shape
    new_height = int(height * zooming_factor)
    new_width  = int(width  * zooming_factor)
    new_image  = np.zeros((new_height, new_width, channels), dtype=np.uint8)
     :
    for i in range(new_height):
        for j in range(new_width):
            new_image[i,j] = original_image[int(i/zooming_factor), int(j/zooming_factor)]
```

(b) Bilinear Interpolation

```
def zoom_image_INTERPOLATION(original_image,zooming_factor):
    height, width, channels = original_image.shape
    new_height = int(height * zooming_factor)
    new_width  = int(width  * zooming_factor)
    new_image  = np.zeros((new_height, new_width, channels), dtype=np.uint8)
     :
    for i in range(new_height):
        for j in range(new_width):
            y_original=i/zooming_factor
            x_original=j/zooming_factor
            y_floor=int(np.floor(y_original))
            x_floor=int(np.floor(x_original))
            y_ceil =int(np.ceil(y_original))
            x_ceil =int(np.ceil(x_original))
            top_left_pixel     = original_image[min(y_floor, height - 1), min(x_floor, width - 1)]
            top_right_pixel    = original_image[min(y_floor, height - 1), min(x_ceil , width - 1)]
            bottom_left_pixel  = original_image[min(y_ceil , height - 1), min(x_floor, width - 1)]
            bottom_right_pixel = original_image[min(y_ceil , height - 1), min(x_ceil , width - 1)]
            y_fraction_from_top  = y_original - y_floor
            x_fraction_from_left = x_original - x_floor
            new_pixel = (
                top_left_pixel     * (1-x_fraction_from_left) * (1-y_fraction_from_top) +
                top_right_pixel    * (x_fraction_from_left)   * (1-y_fraction_from_top) +
                bottom_left_pixel  * (1-x_fraction_from_left) * (y_fraction_from_top)   +
                bottom_right_pixel * (x_fraction_from_left)   * (y_fraction_from_top)
                        )
            new_image[i, j] = new_pixel.astype(np.uint8)
    new_image_in_RGB = cv.cvtColor(new_image, cv.COLOR_BGR2RGB
     :
```

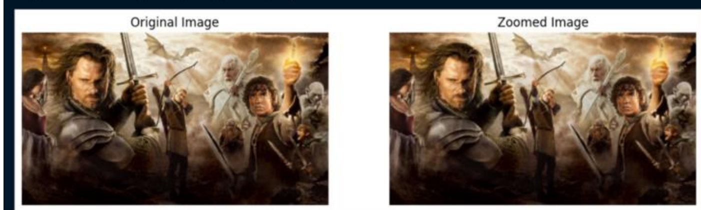
```
Zooming the small image using Nearest Neighbour method
Zooming Factor : 3
Shape of original image : (167, 250, 3)
Shape of new image : (500, 750, 3)
```


```
Normalized SSD Between Original and Zoomed Image : 21.531287111111112
```

```
Zooming the small image using Bilinear Interpolation
Zooming Factor : 3
Shape of original image : (167, 250, 3)
Shape of new image : (500, 750, 3)
```

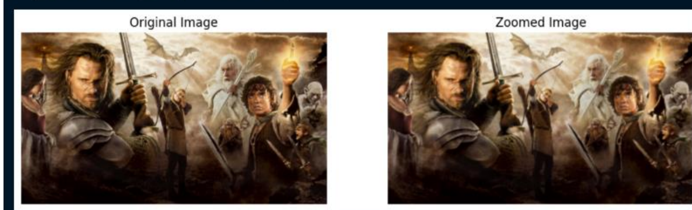

```
Normalized SSD Between Original and Zoomed Image : 23.844788444444443
```

```
Zooming the small image using Nearest Neighbour method
Zooming Factor : 4
Shape of original image : (270, 480, 3)
Shape of new image : (1080, 1920, 3)
```




```
Normalized SSD Between Original and Zoomed Image : 31.284316486625514
```

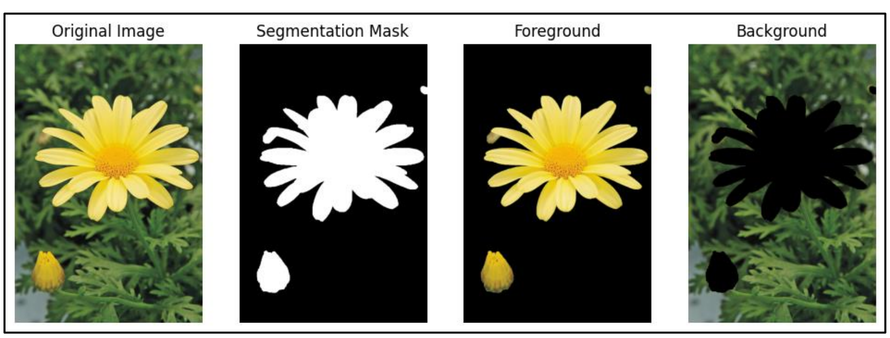
```
Zooming the small image using Bilinear Interpolation
Zooming Factor : 4
Shape of original image : (270, 480, 3)
Shape of new image : (1080, 1920, 3)
```

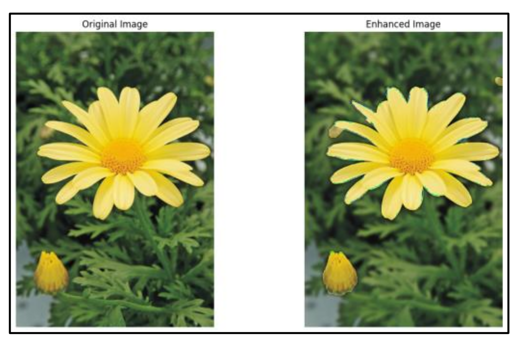



```
Normalized SSD Between Original and Zoomed Image : 39.257033179012346
```

Question 9: Segmentation

```python
# We create an initial mask, which is a black image of the same size as the input image.
# This mask will be used to store information about the foreground and background regions.
mask = np.zeros(image.shape[:2], np.uint8)

# This line defines a rectangle that roughly encloses the flower in the image.
# The rectangle coordinates are (x, y, width, height).
rect = (50, 50, image.shape[1]-50, image.shape[0]-50)

# These are matrices used by the GrabCut algorithm to internally store information about the
# background and foreground models during the segmentation process.
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# The GrabCut algorithm is applied to the input image using the cv.grabCut() function.
cv.grabCut(image, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)

# This line converts the original mask (which contains multiple labels) into a binary mask
# where foreground and probable foreground pixels are set to 1, and the rest (background
# and probable background) are set to 0.
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')

# These lines use the binary mask to create segmented images:
# a foreground image where the flower is isolated, and a background image where the flower is removed
foreground = image * mask2[:, :, np.newaxis]
background = image - foreground

# Here, we apply a Gaussian blur to the background image.
# This blurs the background, making it more visually appealing as a backdrop for the flower.
blurred_background = cv.GaussianBlur(background, (21, 21), 0)

# We add the blurred background to the foreground image,
# creating an enhanced image with the flower in focus and a blurred background.
enhanced_image = foreground + blurred_background
```





Why is the background just beyond the edge of the flower quite dark in the enhanced image?

- As we move from the flower's edge towards the background, the contrast between the flower and the background decreases.
- The blurred background pixels get mixed with the pixels that represent the flower's edge, causing the transition zone between the flower and the background to become less distinct. This reduced contrast contributes to the perception of a darker background.