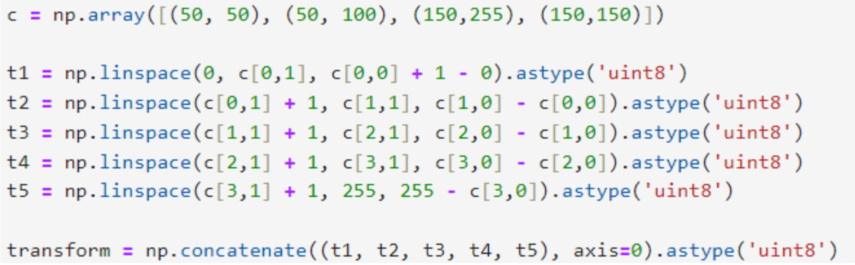
EN3160 Assignment 1
Intensity Transformations and Neighborhood Filtering

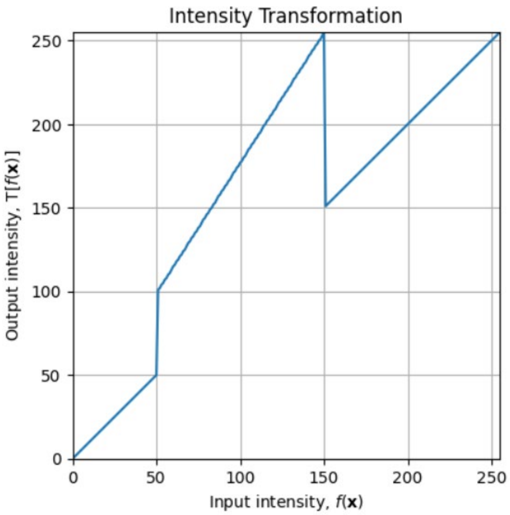
Name: Kavinda W.M.C. GitHub link: https://github.com/Chamod-Kavinda
Index No: 200301D

Question 1 - Implement the intensity transformation on the given image.

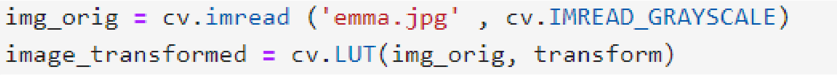
Generate the transformation.



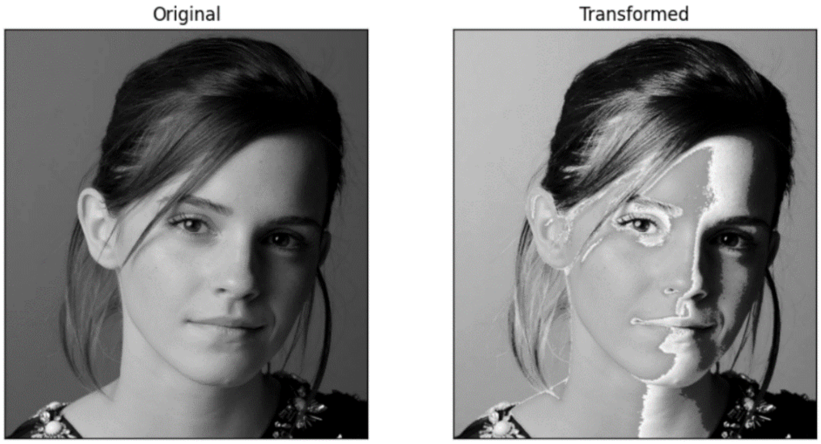
```
c = np.array([(50, 50), (50, 100), (150,255), (150,150)])

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, 255, 255 - c[3,0]).astype('uint8')

transform = np.concatenate((t1, t2, t3, t4, t5), axis=0).astype('uint8')
```



Apply the transformation to the image.

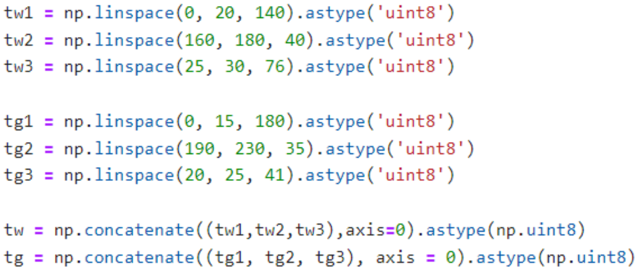


```
img_orig = cv.imread ('emma.jpg' , cv.IMREAD_GRAYSCALE)
image_transformed = cv.LUT(img_orig, transform)
```

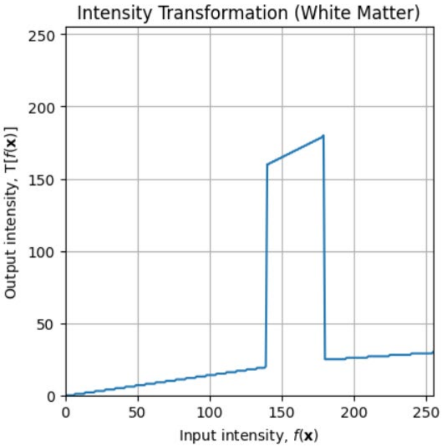


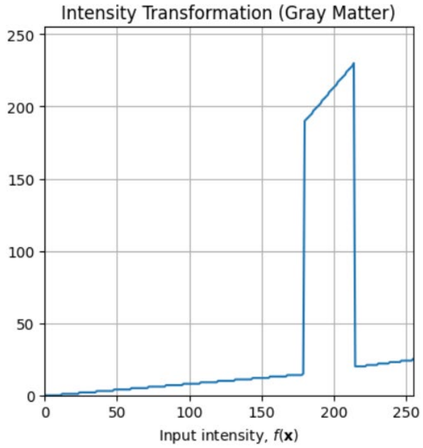
The transformation has effectively enhanced the pixels in the grayscale range between 50 and 150, causing them to shift towards higher grayscale values. As a result, these pixels appear whiter in the transformed image.

Question 2 - Apply a similar operation to accentuate white matter and gray matter.

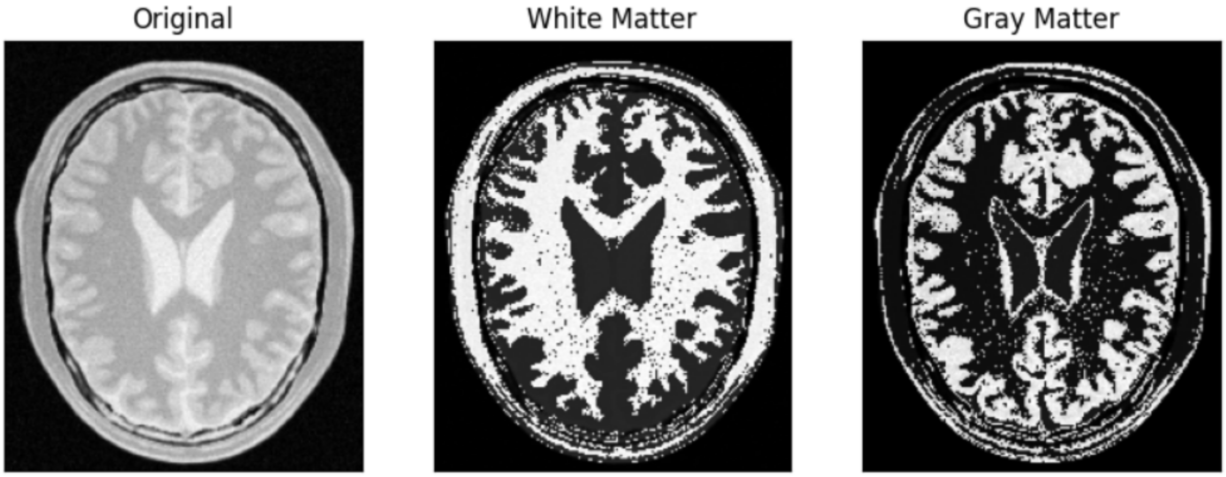


```
tw1 = np.linspace(0, 20, 140).astype('uint8')
tw2 = np.linspace(160, 180, 40).astype('uint8')
tw3 = np.linspace(25, 30, 76).astype('uint8')

tg1 = np.linspace(0, 15, 180).astype('uint8')
tg2 = np.linspace(190, 230, 35).astype('uint8')
tg3 = np.linspace(20, 25, 41).astype('uint8')

tw = np.concatenate((tw1,tw2,tw3),axis=0).astype(np.uint8)
tg = np.concatenate((tg1, tg2, tg3), axis = 0).astype(np.uint8)
```





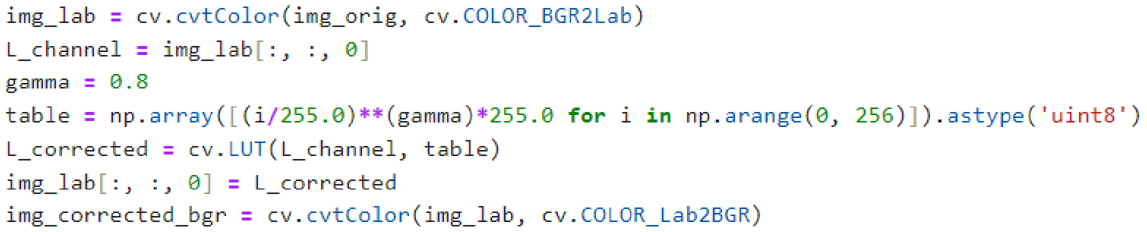
To emphasize the white and gray matter in the brain's grayscale image, the original grayscale levels should transform to higher values, making them appear bright and distinct.



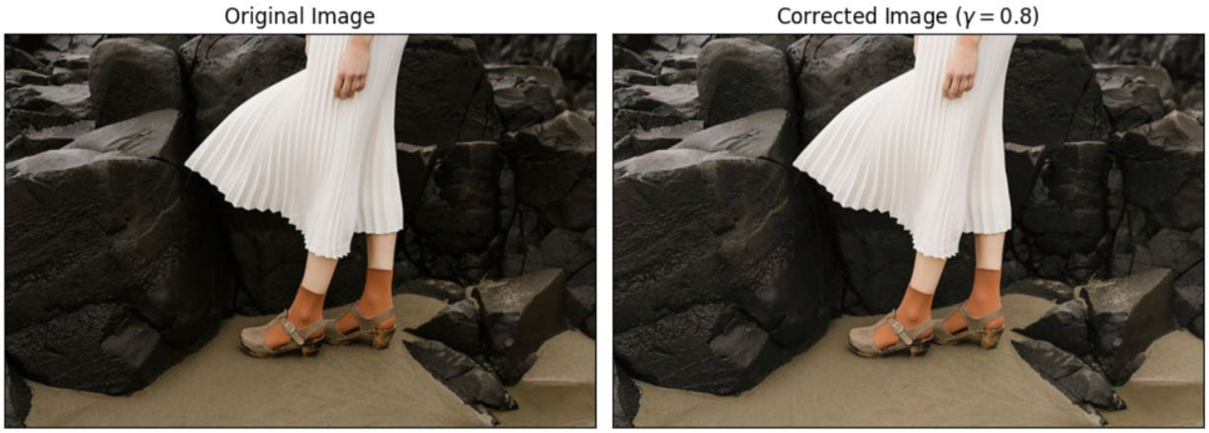
To make the white matter more prominent, the transformation ensures that pixels in that range (grayscale levels 140-180) become white in the output, enhancing their visibility. Similarly, to highlight the gray matter, the transformation ensures that pixels in the range (grayscale levels 180-215) turn into white in the output.

Question 3 - Apply gamma correction to the L plane.

a) The image is converted to the Lab color space using OpenCV. Next, the L channel is isolated from the image and a gamma transformation is designed for a specific gamma value (here 0.8). Then it's combined back with the original a and b channels.



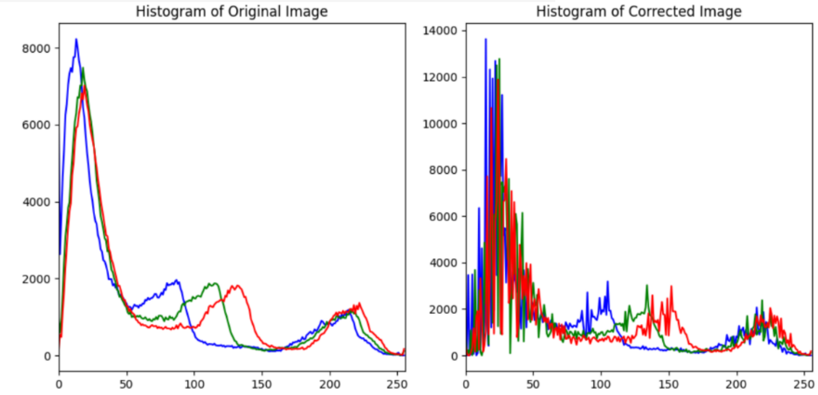
```
img_lab = cv.cvtColor(img_orig, cv.COLOR_BGR2Lab)
L_channel = img_lab[:, :, 0]
gamma = 0.8
table = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0, 256)]).astype('uint8')
L_corrected = cv.LUT(L_channel, table)
img_lab[:, :, 0] = L_corrected
img_corrected_bgr = cv.cvtColor(img_lab, cv.COLOR_Lab2BGR)
```



b) Show histograms.

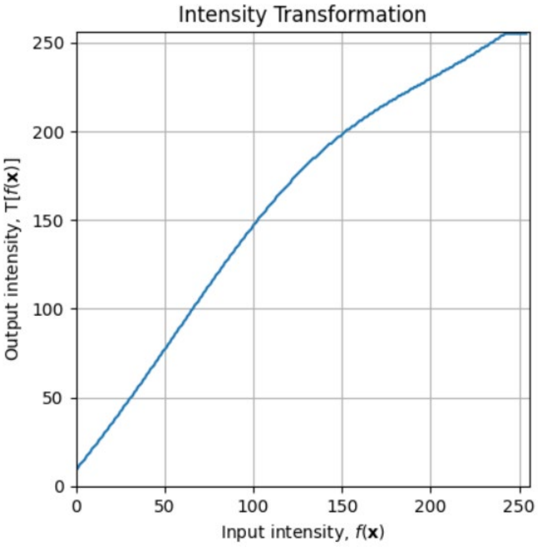
```
for i,c in enumerate(color):
    hist_corrected = cv.calcHist([img_corrected_bgr], [i], None, [256], [0, 256])
    ax[1].plot(hist_corrected, color = c)
```

Using calcHist function generate the
Histograms and plot

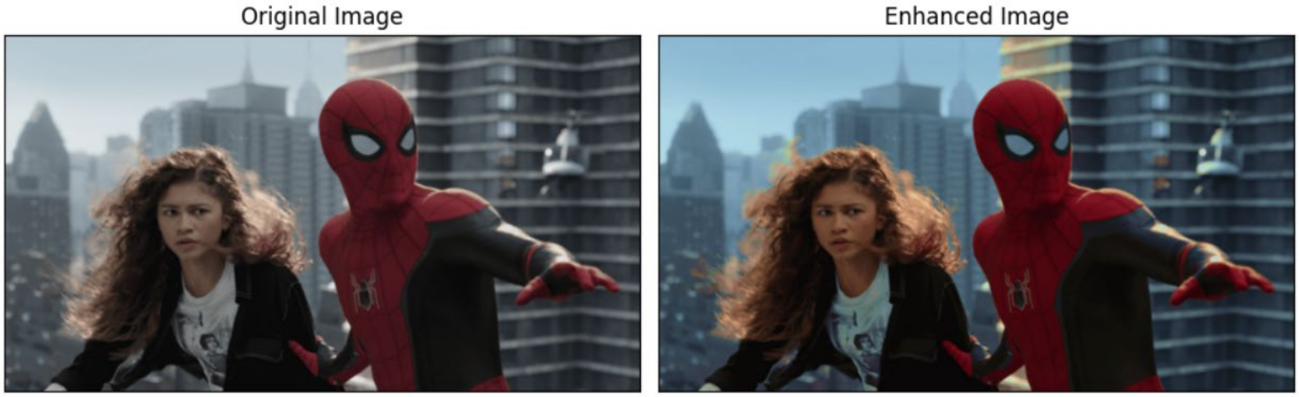


Question 4 – Increasing the vibrance of a photograph.

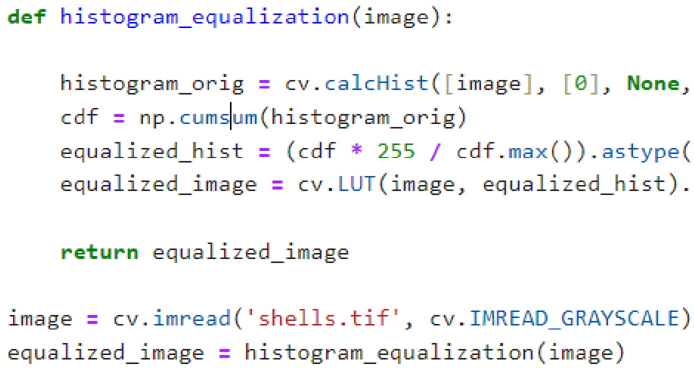
```
# (a) Split into HSV planes
hue, saturation, value = cv.split(hsv_image)

# (b) Apply the intensity transformation to the saturation plane
def intensity_transformation(x, a, sigma=70):
    f_x = np.minimum(x + a * 128 * np.exp(-(x - 128) ** 2 / (2 * sigma ** 2)), 255).astype('uint8')
    return f_x
# (c) Adjust the value of a
a = 0.4
transformed_saturation = intensity_transformation(saturation, a)

# (d) Recombine the three planes
enhanced_hsv_image = cv.merge([hue, transformed_saturation, value])

# Convert back to BGR for visualization
enhanced_image = cv.cvtColor(enhanced_hsv_image, cv.COLOR_HSV2BGR)

# (e) Display the images and intensity transformation
```

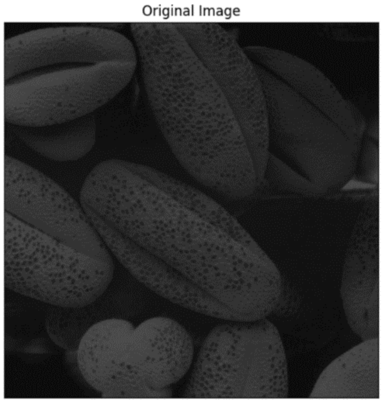


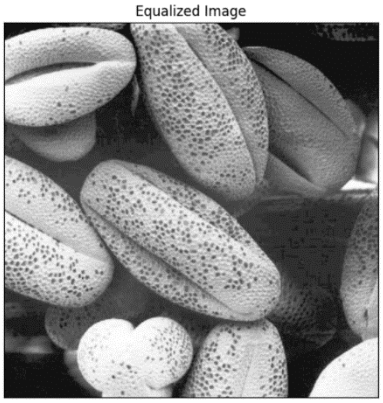
The right side of the output shows the enhanced image after applying the intensity transformation to the saturation channel. This transformation increases the vibrance of colors in the image, making them appear more vivid and saturated. The enhancement is particularly noticeable in areas with colorful elements.

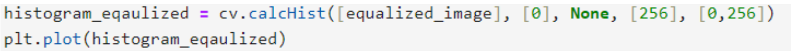


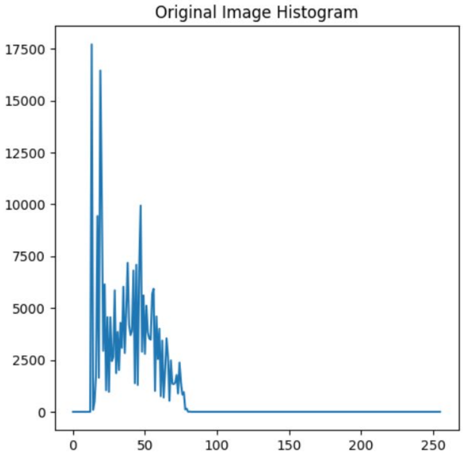
Question 5 – Histogram Equalization.

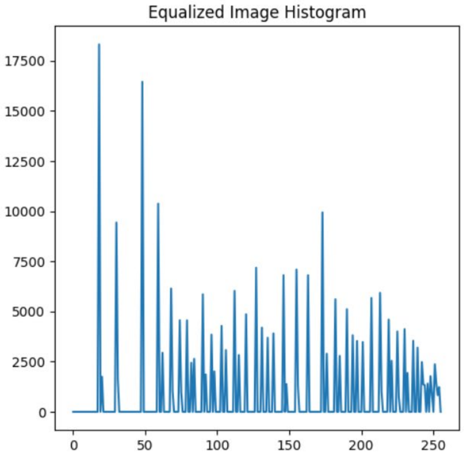

```
def histogram_equalization(image):

    histogram_orig = cv.calcHist([image], [0], None, [256], [0,256])
    cdf = np.cumsum(histogram_orig)
    equalized_hist = (cdf * 255 / cdf.max()).astype('uint8')
    equalized_image = cv.LUT(image, equalized_hist).astype('uint8')

    return equalized_image

image = cv.imread('shells.tif', cv.IMREAD_GRAYSCALE)
equalized_image = histogram_equalization(image)
```


Original Image


Equalized Image

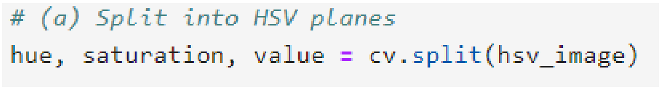

```
histogram_eqaulized = cv.calcHist([equalized_image], [0], None, [256], [0,256])
plt.plot(histogram_eqaulized)
```

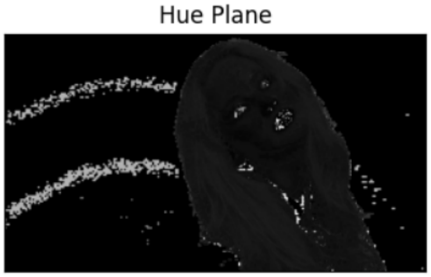

Original Image Histogram

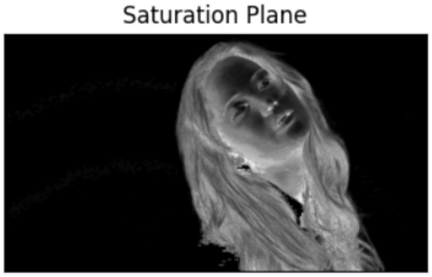

Equalized Image Histogram

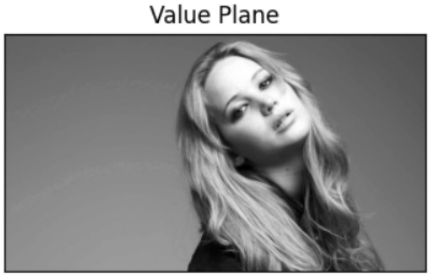
Histogram equalization enhances image contrast and visibility by reshaping pixel intensity values. This is done by calculating a cumulative distribution function (CDF) from the original image's histogram, normalizing it, and applying it as a transformation to the image.

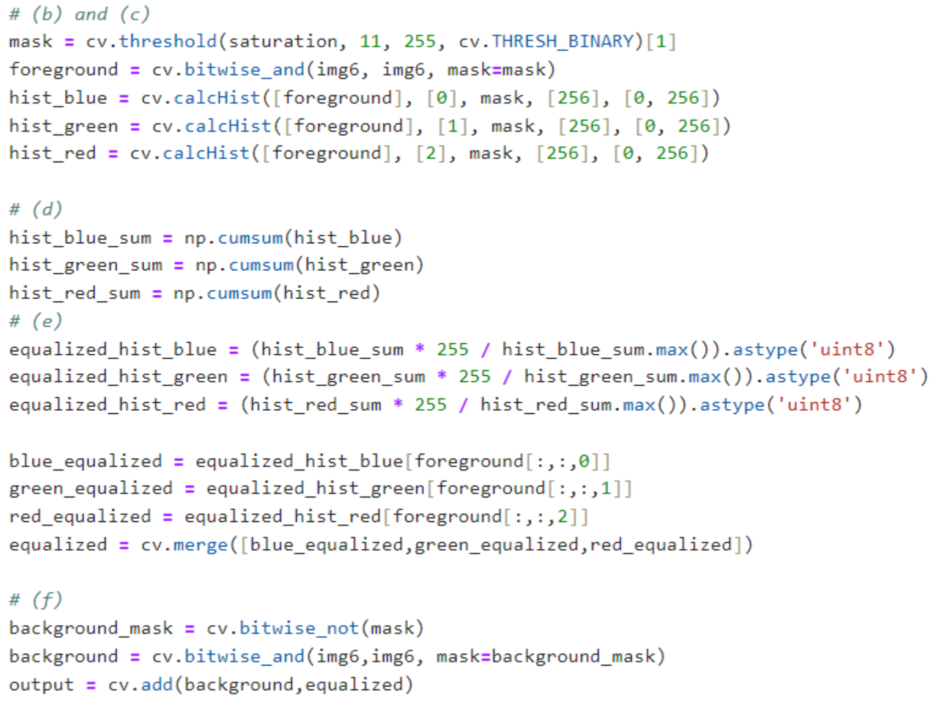
Question 6 – Histogram Equalization only to the foreground of an image.


```
# (a) Split into HSV planes
hue, saturation, value = cv.split(hsv_image)
```

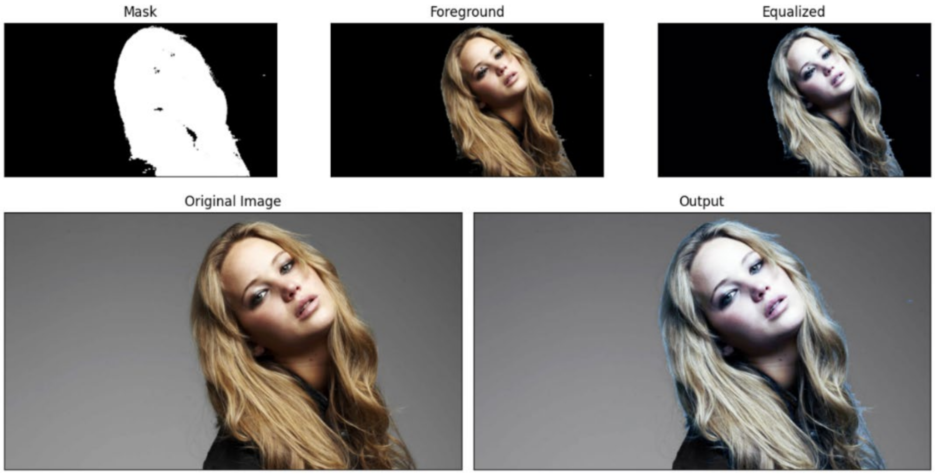

Hue Plane


Saturation Plane


Value Plane



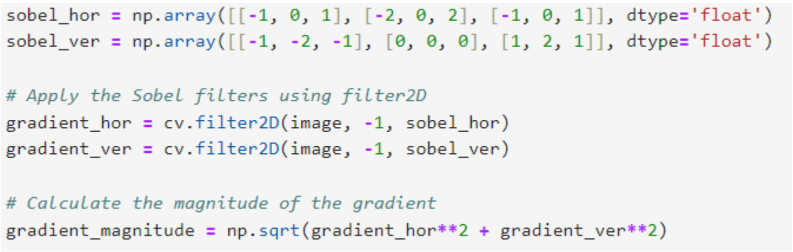
```
# (b) and (c)
mask = cv.threshold(saturation, 11, 255, cv.THRESH_BINARY)[1]
foreground = cv.bitwise_and(img6, img6, mask=mask)
hist_blue = cv.calcHist([foreground], [0], mask, [256], [0, 256])
hist_green = cv.calcHist([foreground], [1], mask, [256], [0, 256])
hist_red = cv.calcHist([foreground], [2], mask, [256], [0, 256])

# (d)
hist_blue_sum = np.cumsum(hist_blue)
hist_green_sum = np.cumsum(hist_green)
hist_red_sum = np.cumsum(hist_red)
# (e)
equalized_hist_blue = (hist_blue_sum * 255 / hist_blue_sum.max()).astype('uint8')
equalized_hist_green = (hist_green_sum * 255 / hist_green_sum.max()).astype('uint8')
equalized_hist_red = (hist_red_sum * 255 / hist_red_sum.max()).astype('uint8')

blue_equalized = equalized_hist_blue[foreground[:,:,0]]
green_equalized = equalized_hist_green[foreground[:,:,1]]
red_equalized = equalized_hist_red[foreground[:,:,2]]
equalized = cv.merge([blue_equalized,green_equalized,red_equalized])

# (f)
background_mask = cv.bitwise_not(mask)
background = cv.bitwise_and(img6,img6, mask=background_mask)
output = cv.add(background,equalized)
```



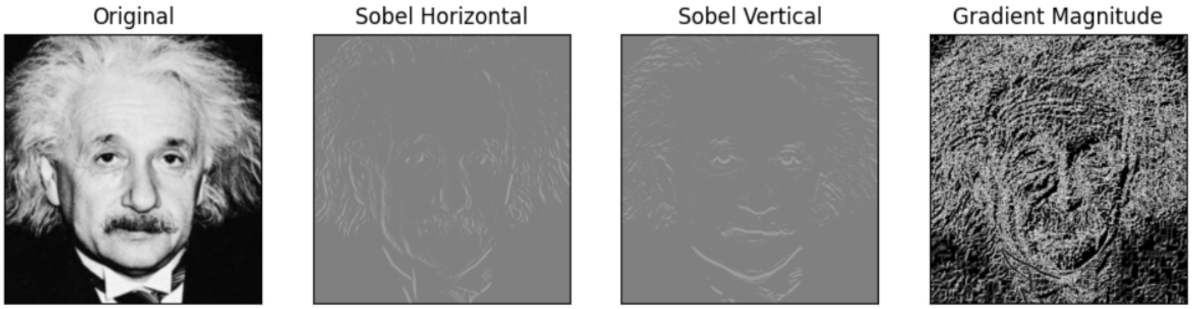
The final output shows an image with an enhanced and visually striking foreground, thanks to histogram equalization. The background remains unchanged, ensuring a balanced and appealing composition.

Question 7 – Sobel Filtering.

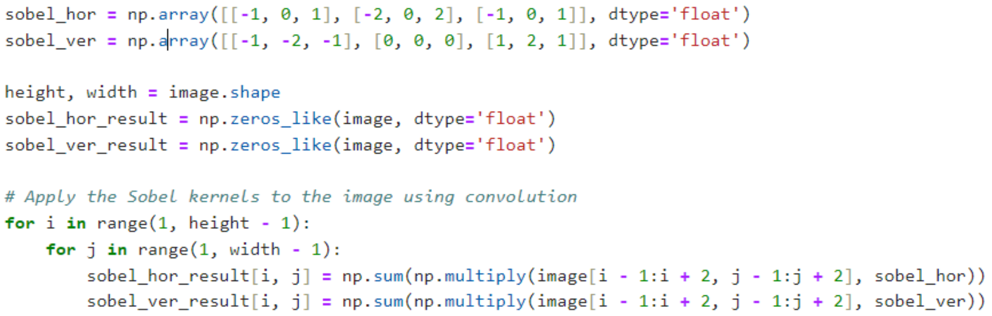
a) Using the existing filter2D



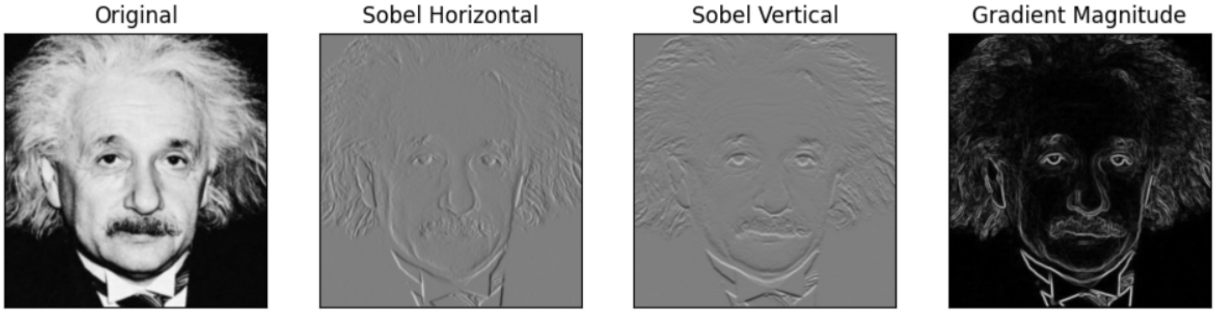
```
sobel_hor = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype='float')
sobel_ver = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype='float')

# Apply the Sobel filters using filter2D
gradient_hor = cv.filter2D(image, -1, sobel_hor)
gradient_ver = cv.filter2D(image, -1, sobel_ver)

# Calculate the magnitude of the gradient
gradient_magnitude = np.sqrt(gradient_hor**2 + gradient_ver**2)
```



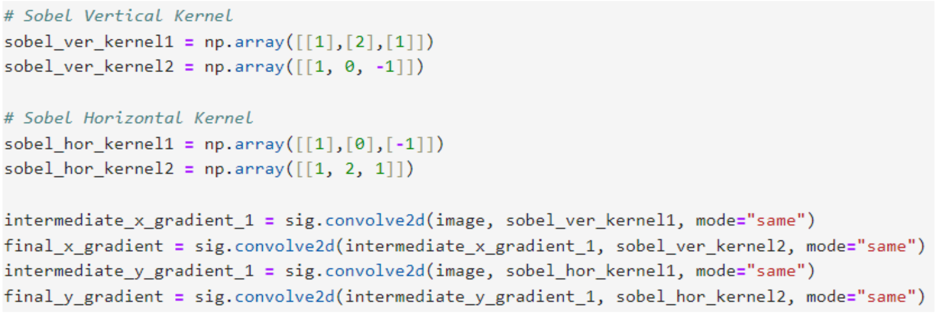
b) Using own function



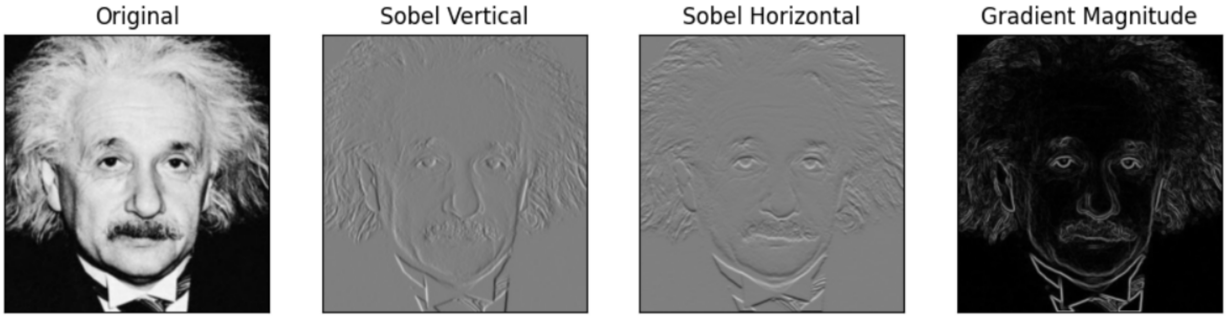
```
sobel_hor = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype='float')
sobel_ver = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype='float')

height, width = image.shape
sobel_hor_result = np.zeros_like(image, dtype='float')
sobel_ver_result = np.zeros_like(image, dtype='float')

# Apply the Sobel kernels to the image using convolution
for i in range(1, height - 1):
    for j in range(1, width - 1):
        sobel_hor_result[i, j] = np.sum(np.multiply(image[i - 1:i + 2, j - 1:j + 2], sobel_hor))
        sobel_ver_result[i, j] = np.sum(np.multiply(image[i - 1:i + 2, j - 1:j + 2], sobel_ver))
```



c) Using Property



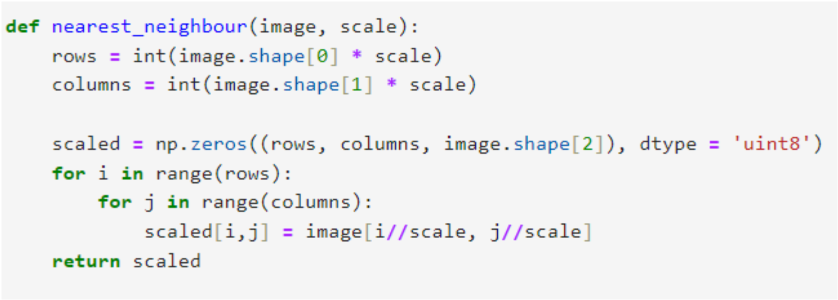
```
# Sobel Vertical Kernel
sobel_ver_kernel1 = np.array([[1],[2],[1]])
sobel_ver_kernel2 = np.array([[1, 0, -1]])

# Sobel Horizontal Kernel
sobel_hor_kernel1 = np.array([[1],[0],[-1]])
sobel_hor_kernel2 = np.array([[1, 2, 1]])

intermediate_x_gradient_1 = sig.convolve2d(image, sobel_ver_kernel1, mode="same")
final_x_gradient = sig.convolve2d(intermediate_x_gradient_1, sobel_ver_kernel2, mode="same")
intermediate_y_gradient_1 = sig.convolve2d(image, sobel_hor_kernel1, mode="same")
final_y_gradient = sig.convolve2d(intermediate_y_gradient_1, sobel_hor_kernel2, mode="same")
```



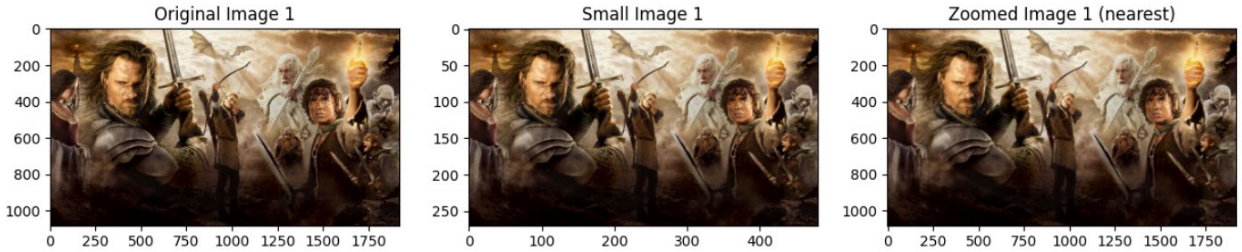
All three codes achieve the same goal of edge detection using Sobel filters, resulting in gradient magnitude images that emphasize the edges and details of the input image. However, they use different methods to apply filters, with differences in speed and efficiency.

Question 8 – Zoom Images

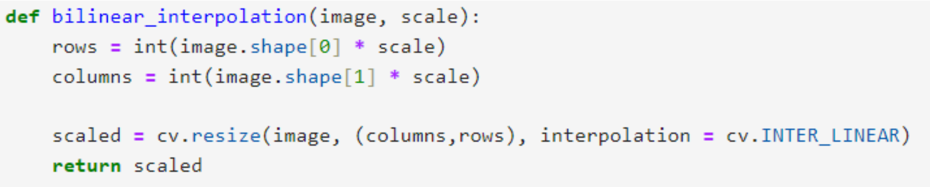
a) Nearest neighbor



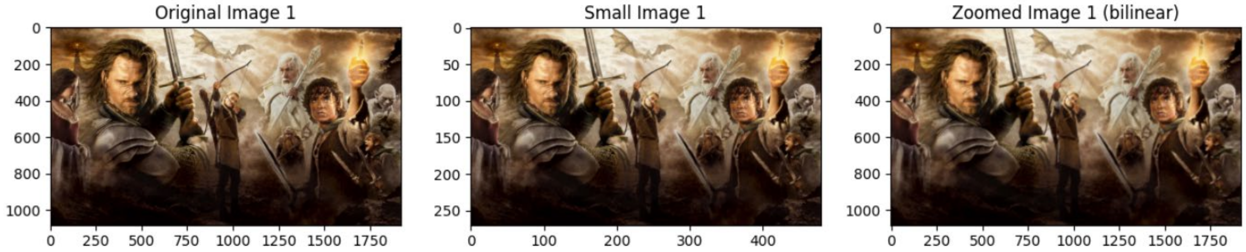
```
def nearest_neighbour(image, scale):
    rows = int(image.shape[0] * scale)
    columns = int(image.shape[1] * scale)

    scaled = np.zeros((rows, columns, image.shape[2]), dtype = 'uint8')
    for i in range(rows):
        for j in range(columns):
            scaled[i,j] = image[i//scale, j//scale]
    return scaled
```



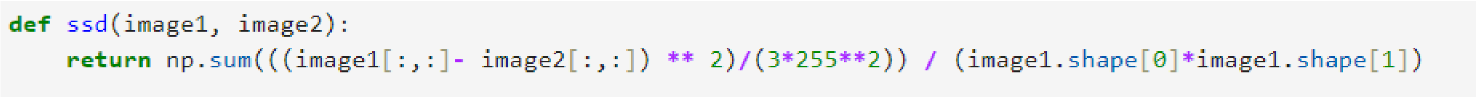
b) Bilinear interpolation



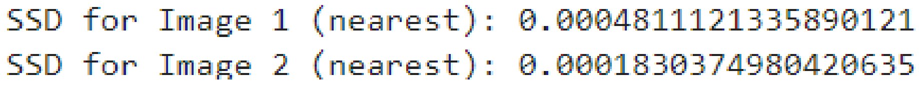
```
def bilinear_interpolation(image, scale):
    rows = int(image.shape[0] * scale)
    columns = int(image.shape[1] * scale)

    scaled = cv.resize(image, (columns,rows), interpolation = cv.INTER_LINEAR)
    return scaled
```



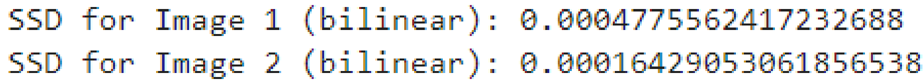
SSD



```
def ssd(image1, image2):
    return np.sum(((image1[:,:]- image2[:,:]) ** 2)/(3*255**2)) / (image1.shape[0]*image1.shape[1])
```



```
SSD for Image 1 (nearest): 0.0004811121335890121
SSD for Image 2 (nearest): 0.0001830374980420635
```

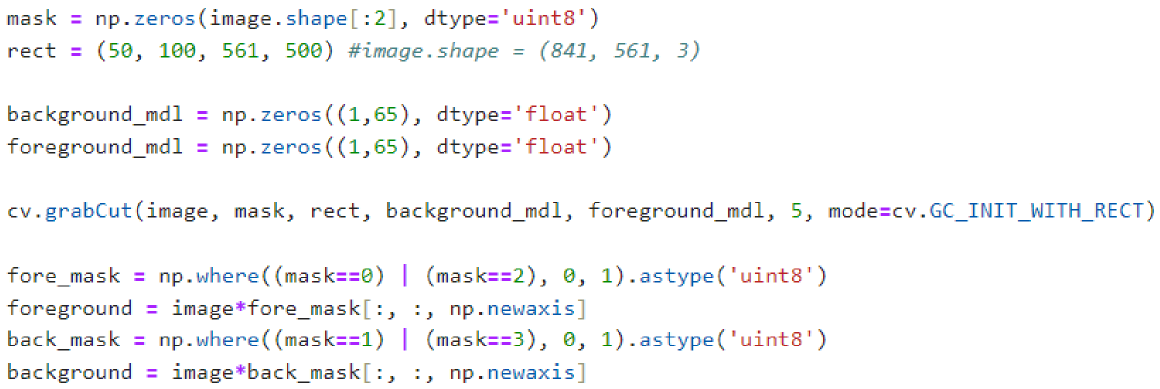


```
SSD for Image 1 (bilinear): 0.0004775624172326888
SSD for Image 2 (bilinear): 0.0001642905306185653
```

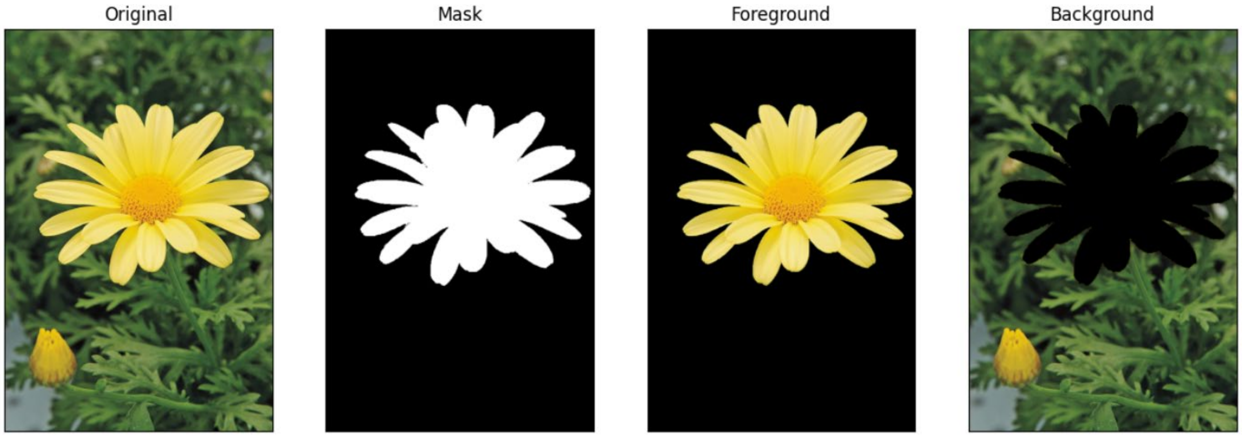
The image zoomed using bilinear interpolation looks smoother because it considers nearby pixels for a gradual transition. This results in a lower SSD compared to the pixelated appearance of the nearest neighbor method, which duplicates pixels for zooming.

Question 9 – Segment an image.

a) Use grabCut to segment the image.



```
mask = np.zeros(image.shape[:2], dtype='uint8')
rect = (50, 100, 561, 500) #image.shape = (841, 561, 3)

background_mdl = np.zeros((1,65), dtype='float')
foreground_mdl = np.zeros((1,65), dtype='float')

cv.grabCut(image, mask, rect, background_mdl, foreground_mdl, 5, mode=cv.GC_INIT_WITH_RECT)

fore_mask = np.where((mask==0) | (mask==2), 0, 1).astype('uint8')
foreground = image*fore_mask[:, :, np.newaxis]
back_mask = np.where((mask==1) | (mask==3), 0, 1).astype('uint8')
background = image*back_mask[:, :, np.newaxis]
```



Original

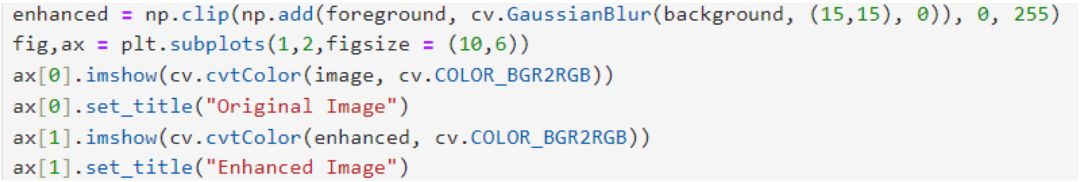
Mask

Foreground

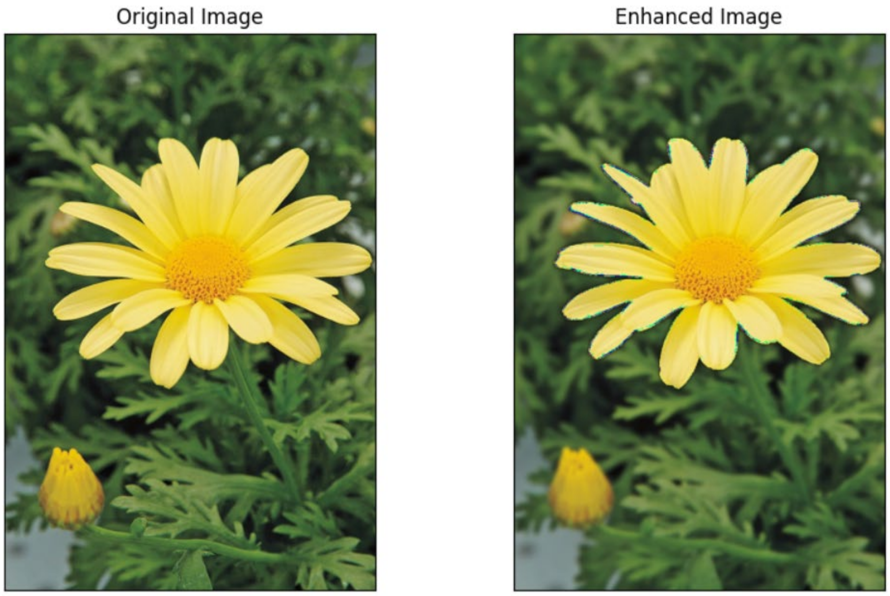
Background

The Foreground image showcases the isolated flower with transparency, while the Background image shows the removed background.

b) Produce an enhanced image with a substantially blurred background.



```
enhanced = np.clip(np.add(foreground, cv.GaussianBlur(background, (15,15), 0)), 0, 255)
fig,ax = plt.subplots(1,2,figsize = (10,6))
ax[0].imshow(cv.cvtColor(image, cv.COLOR_BGR2RGB))
ax[0].set_title("Original Image")
ax[1].imshow(cv.cvtColor(enhanced, cv.COLOR_BGR2RGB))
ax[1].set_title("Enhanced Image")
```



Original Image

Enhanced Image

The enhanced image effectively blurs the background, emphasizing the flower as the main subject.

c) The Gaussian blur increases pixel values around the edges of the black portion, making them higher than the yellow color in the foreground, resulting in a darker appearance at the edges in the enhanced image.