EN3160 – Image Processing and Machine Vision

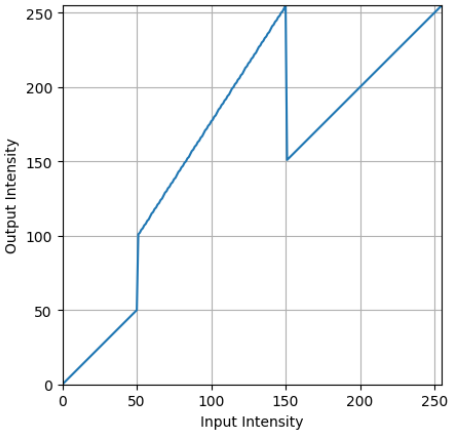
Assignment 1 on Intensity Transformations and Neighborhood Filtering

W. A. S. S. Wanigathunga

200693D

GitHub:
https://github.com/SasiniWanigathunga/EN3160_Image-Processing-and-Machine-Vision/tree/main/Assignment1

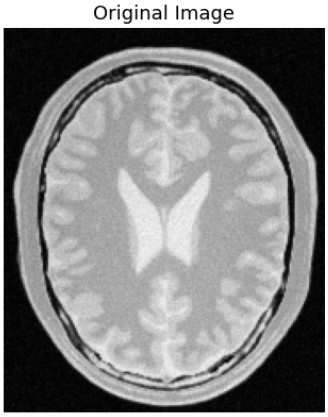
Question 01

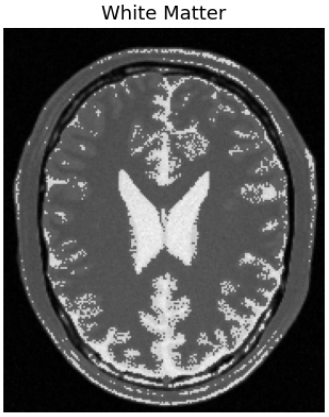


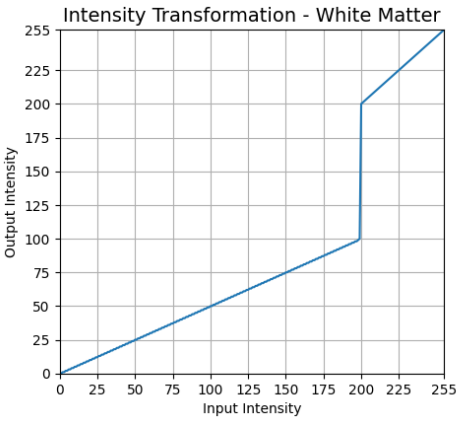
The given intensity transformation is added to the image.

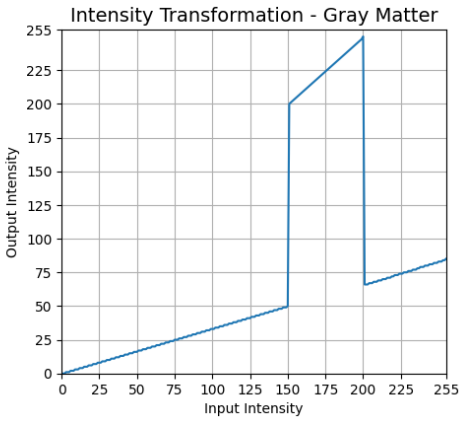
- The graph shows that only the mid-intensity values (50 to 150) are changed, while the low and high values (0 to 50 and 150 to 255) are the same.
- The mid-intensity values are mapped linearly to a higher range (100 to 255), which brightens the pixels with those values.
- The original image has mid-intensity pixels mostly in the shadows of Emma's face, so the transformation makes those parts more illuminated.
- The low and high intensity values which belong to the hair and the bright parts of Emma's face are not affected by the transformation.
- The image looks unnatural but we can adjust the range of the mid-intensity values appropriately to make the illumination more natural.

Question 02









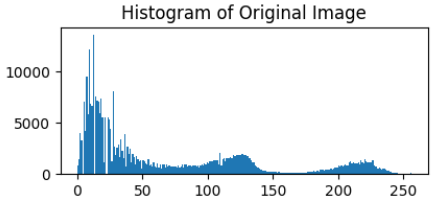
To accentuate white matter and gray matter, these transformations are used. I have reduced the pixel intensity of the other pixels which do not represent the white matter or gray matter to accentuate the white matter and gray matter.

Question 03

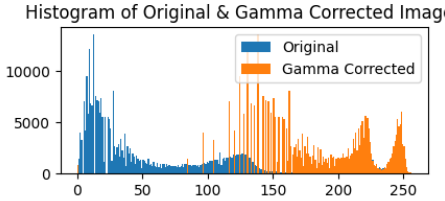
In this question the given image is converted to LAB color space and gamma correction was applied to the L plane with different gamma values.

```
img_original=cv.imread('highlights_and_shadows.jpg',cv.IMREAD_COLOR)
img_Lab=cv.cvtColor(img_original,cv.COLOR_BGR2LAB)
L,a,b=cv.split(img_Lab)
gamma = [0.2, 0.7, 1.5]
for i in range(len(gamma)):
  L_new=np.array(255*(L/255)**(gamma[i]),dtype='uint8')
  img_gamma_corrected=cv.merge((L_new,a,b))
  img_gamma_corrected=cv.cvtColor(img_gamma_corrected,cv.COLOR_LAB2BGR)
```

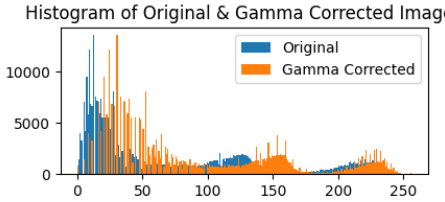
Original Image



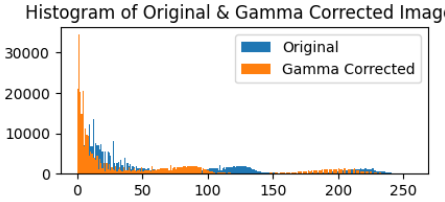
Gamma Corrected Image - Gamma = 0.2



Gamma Corrected Image - Gamma = 0.7



Gamma Corrected Image - Gamma = 1.5



The image becomes brighter when $\gamma<1$ and darker when $\gamma>1$ and brightness changes because the intensity distribution changes when the gamma value is not 1

Values of γ such that $0 < \gamma < 1$ map a narrow range of dark pixels to a wider range of dark pixels. In other words when $0<\gamma<1$, the gamma correction redistributes the colors so that black and white stay the same but colors in between move to the light edges.

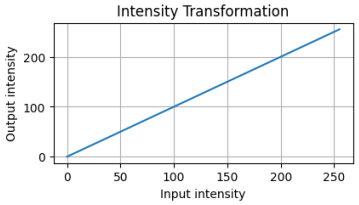
This makes the image appear brighter. Similarly for $\gamma > 1$ has the opposite effect.

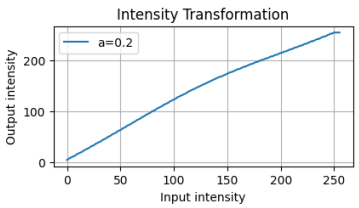
Question 04

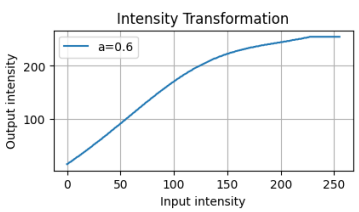
The given intensity transformation is applied to the saturation plane after the image is splitted into hue, saturation, and value planes. The value of a is adjusted to increase the vibrance of a photograph using the given transformation.

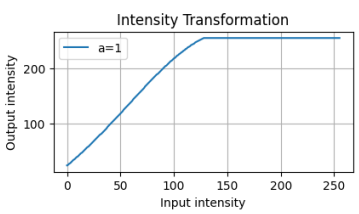
```
#intensity transformation given
def intensity_transformation(x,sigma,a):
    return np.minimum(x+a*128*(np.exp(-(x-128)**2/(2*sigma**2))),255).astype('uint8')
```

```
sigma=70
img_original=cv.imread('spider.png',cv.IMREAD_COLOR)
img_hsv=cv.cvtColor(img_original,cv.COLOR_BGR2HSV)
h,s,v=cv.split(img_hsv)#split the image into hue, saturation, and value planes
a=[0.2,0.4,0.6,0.8,1]
for i in range(len(a)):
    #apply the transformation to the saturation plane
    s_new=intensity_transformation(s,sigma,a[i])
    #recombine the 3 planes and convert back to RGB
    img_new_s=cv.merge((h,s_new,v))
    img_new=cv.cvtColor(img_new_s,cv.COLOR_HSV2BGR)
    img_new=cv.cvtColor(img_new,cv.COLOR_BGR2RGB)
```









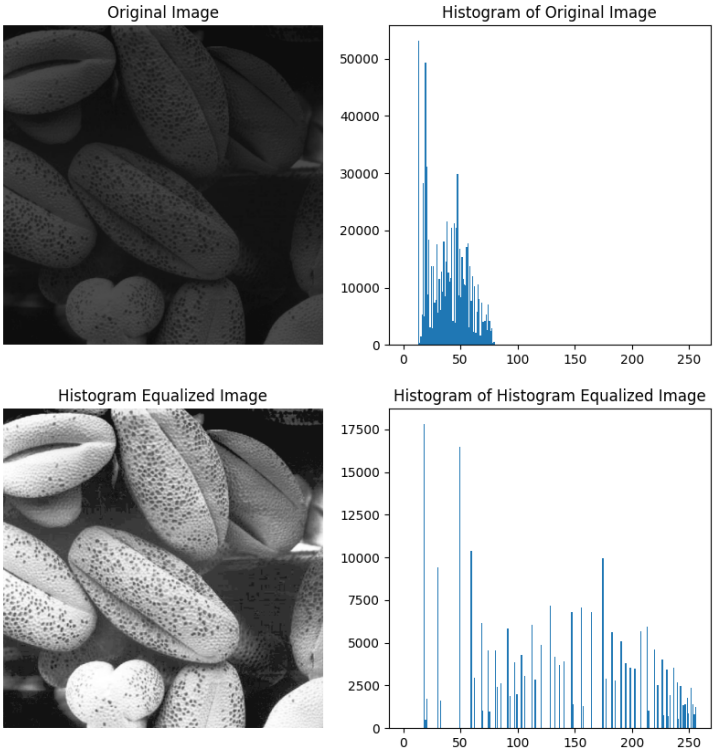
Increasing the vibrance of an image means increasing the intensity of muted colors while leaving saturated colors untouched.
Specifically, vibrance enhances colors that are more muted and mostly ignores warmer colors (yellows, oranges, and reds), while prioritizing cooler colors (blues and greens).

To get a visually pleasing output, we can say that the value of a can be adjusted to 0.6.

Question 05

The following function is used to carry out the histogram equalization on the image.

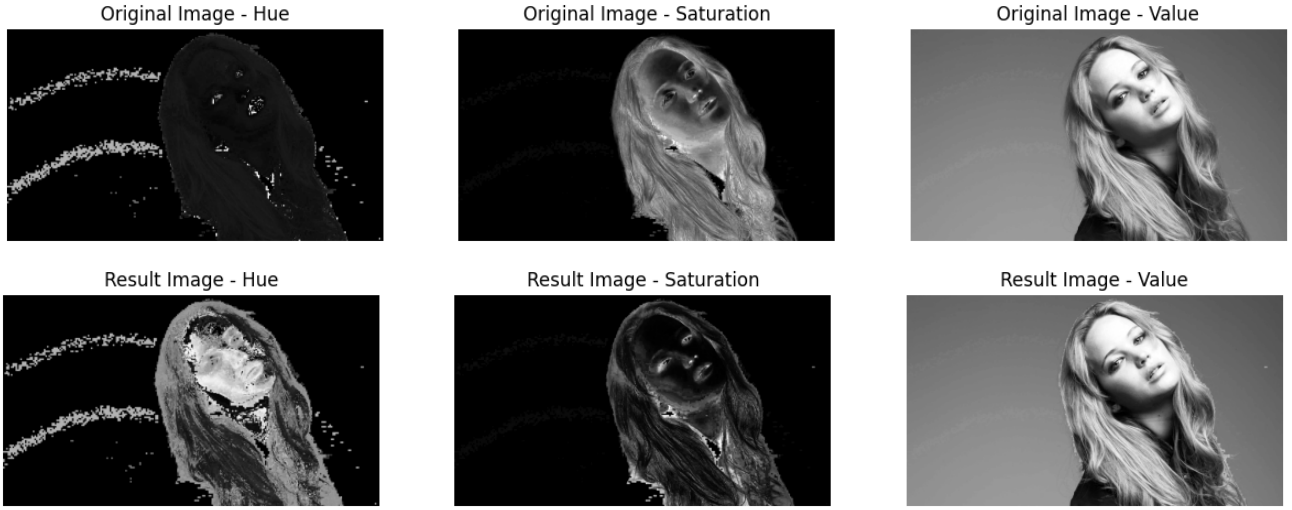
```
def histogram_equalization(image):
 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
 hist = cv2.calcHist([gray], [0], None, [256], [0, 256])
 hist = hist / hist.sum()
 cdf = hist.cumsum()
 cdf = cdf * 255
 cdf = np.round(cdf).astype(np.uint8)
 equalized = cdf[gray]
 return equalized, cv2.calcHist([equalized], [0], None, [256], [0, 256])
```



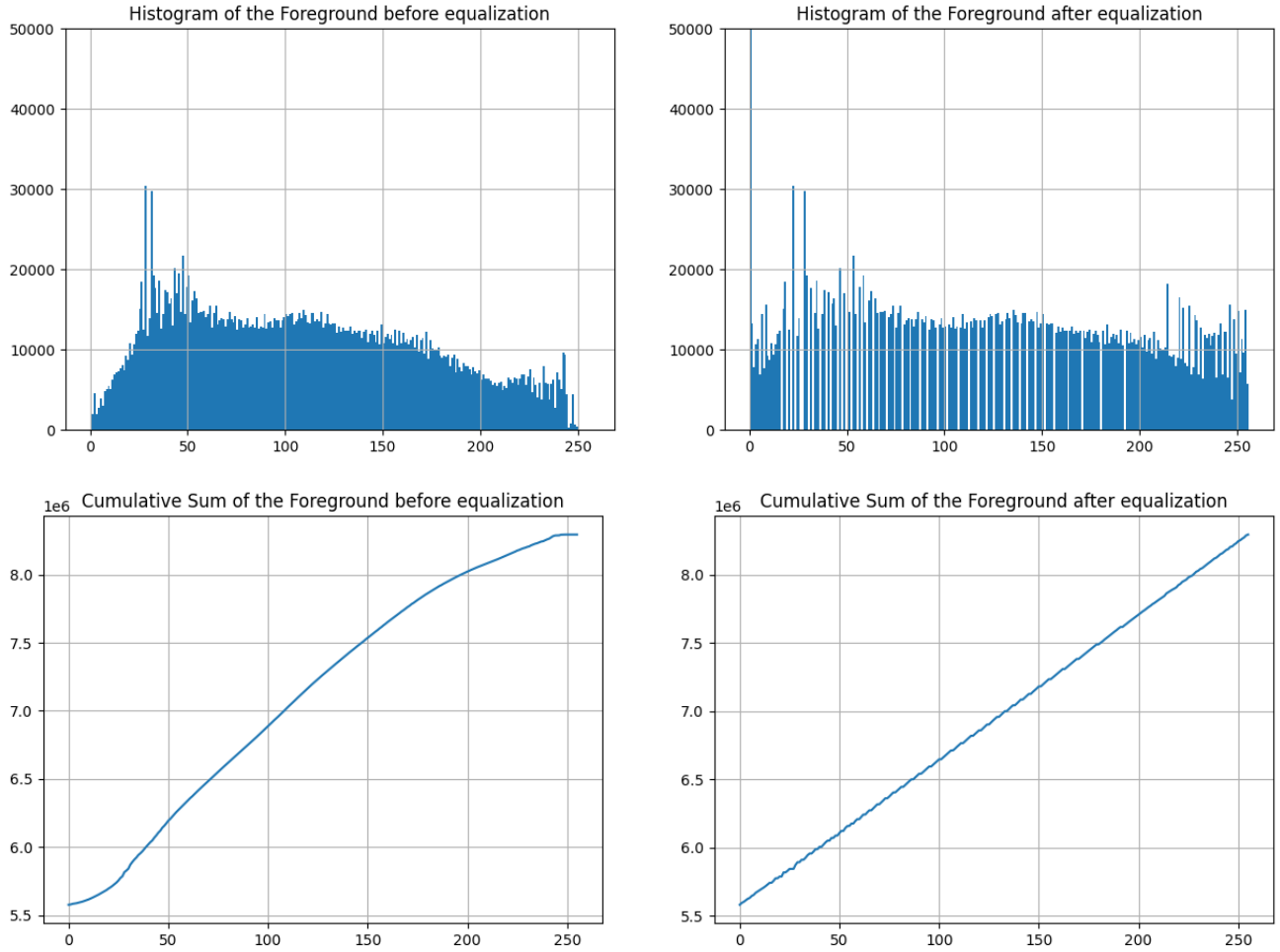
Following procedure was done in the function:
1. Calculate the histogram of the grayscale image.
2.Normalize the histogram values.
3.Compute the cumulative distribution function (CDF) of the histogram.
4.Map the CDF values to cover the full dynamic range (0 to 255).
5.Round and convert the CDF values to unsigned 8-bit integers.
6.Map the original intensity values of the image using the CDF.
7.Return the equalized image and its histogram.

After the equalization, the histogram is distributed evenly across the 0-255 grayscale values.

Question 06

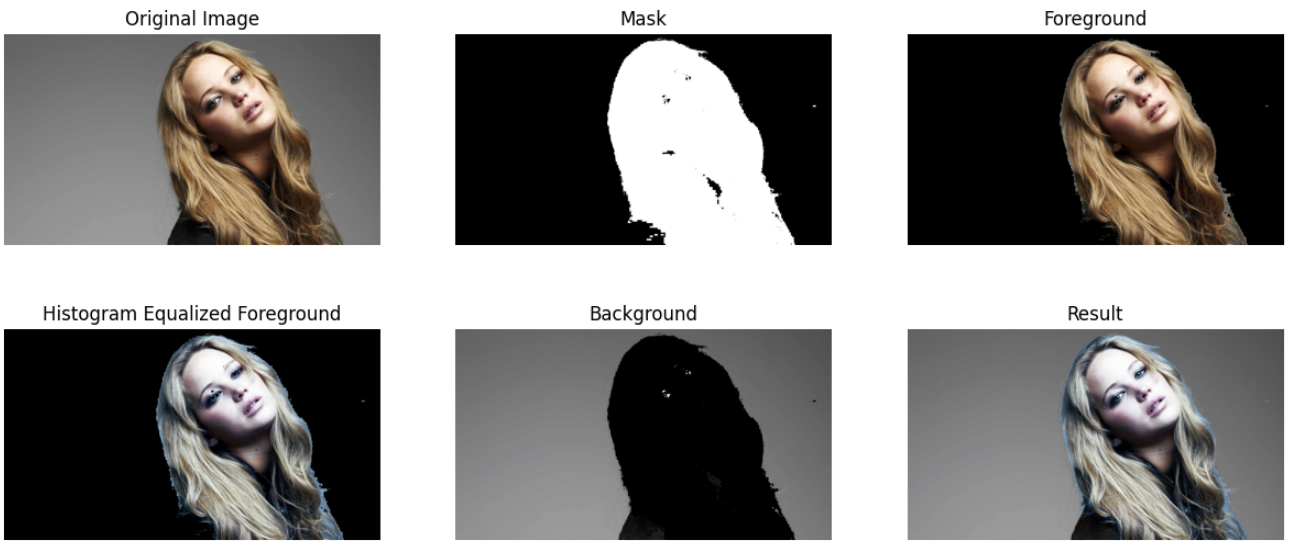


The above second row shows how hue, saturation, and value planes of the result with the histogram equalized foreground for the original image in the first row.



From the hue, saturation and value planes of the image we can see that in the saturation plane, the foreground object has comparably higher saturation values than the background. So it is easy to separate the two by thresholding the saturation plane.

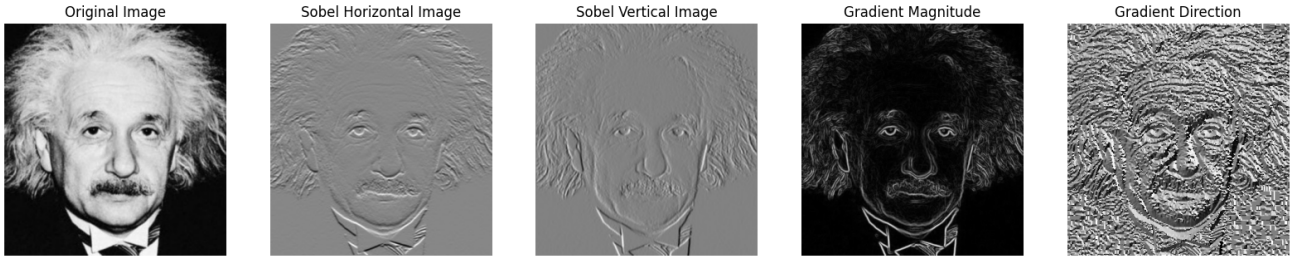
```
# Threshold value was selected by trial and error
threshold = 12
mask = (s > threshold).astype(np.uint8) * 255
mask_3d = np.repeat(mask[:, :, None], 3, axis=2)

foreground_hsv = np.bitwise_and(img_hsv, mask_3d)
foreground_rgb = cv.cvtColor(foreground_hsv, cv.COLOR_HSV2RGB)
```



Question 07

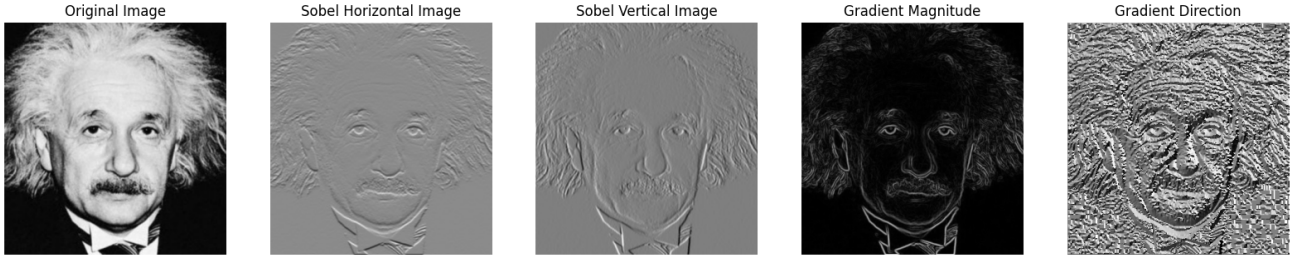
Using the existing filter2D to Sobel filter the image:

```
sobel_horizontal = np.array( [ [-1, 0, 1], [-2, 0, 2], [-1, 0, 1] ] ).astype(np.float32)
sobel_vertical = np.array( [ [-1, -2, -1], [0, 0, 0], [1, 2, 1] ] ).astype(np.float32)

sobel_horizontal_img = cv.filter2D(img_original, -1, sobel_horizontal)
sobel_vertical_img = cv.filter2D(img_original, -1, sobel_vertical)

gradient_magnitude = np.sqrt(sobel_horizontal_img.astype(np.float64)**2 +
sobel_vertical_img.astype(np.float64)**2)
gradient_direction = np.arctan2(sobel_vertical_img.astype(np.float64), sobel_horizontal_img.astype(np.float64))
```



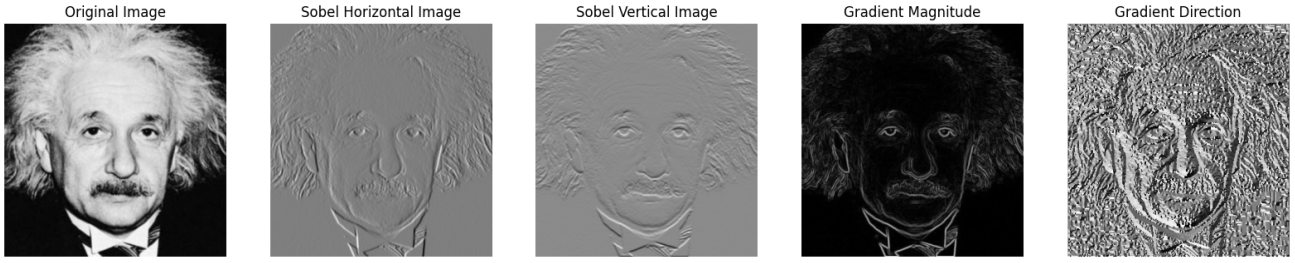
Using my own code to Sobel filter the image:

```
sobel_vertical_img = np.zeros(img_original.shape)
sobel_horizontal_img = np.zeros(img_original.shape)
rows, columns = img_original.shape
# Carry out padding
padding = 0
padded = np.full((rows + 2, columns + 2), padding, dtype=np.uint8)
# copy img image into center of result image
padded[1:rows + 1, 1:columns + 1] = img_original

for i in range(rows):
  for j in range(columns):
    sobel_horizontal_img[i,j] = np.sum(np.multiply(sobel_horizontal, padded[i:i + 3, j:j + 3]))

for i in range(rows):
  for j in range(columns):
    sobel_vertical_img[i,j] = np.sum(np.multiply(sobel_vertical, padded[i:i + 3, j:j + 3]))
```



Using the given property to carry out Sobel filtering:

Sobel Vertical Kernel	Sobel Horizontal Kernel
$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$




```
sobel_vertical_img = np.zeros(img_original.shape)
sobel_vertical_img_intermediate = np.zeros(img_original.shape)
sobel_horizontal_img = np.zeros(img_original.shape)
sobel_horizontal_img_intermediate = np.zeros(img_original.shape)

# Sobel Vertical Kernel
sobel_vertical_kernel_1 = np.array([[1],[2],[1]])
sobel_vertical_kernel_2 = np.array([[1,0,-1]])

# Sobel Horizontal Kernel
sobel_horizontal_kernel_1 = np.array([[1],[0],[-1]])
sobel_horizontal_kernel_2 = np.array([[1,2,1]])

sobel_vertical_img_intermediate = sig.convolve2d(img_original, sobel_vertical_kernel_1, mode="same")
sobel_vertical_img = sig.convolve2d(sobel_vertical_img_intermediate, sobel_vertical_kernel_2, mode="same")
sobel_horizontal_img_intermediate = sig.convolve2d(img_original, sobel_horizontal_kernel_1, mode="same")
sobel_horizontal_img = sig.convolve2d(sobel_horizontal_img_intermediate, sobel_horizontal_kernel_2,
mode="same")
```

The outputs are the same for all 3 methods.

Question 08

```
def zoom_nearest_neighbour(factor, image):
    rows = int(factor*image.shape[0])
    columns = int(factor*image.shape[1])
    zoomed = np.zeros((rows,columns,3),dtype = image.dtype)
    for i in range(rows):
        for j in range(columns):
            zoomed[i,j] = image[int(i/factor),int(j/factor)]
    return zoomed
```

```
def zoom_bilinear_interpolation(factor, image):
    rows = int(factor*image.shape[0])
    columns = int(factor*image.shape[1])
    zoomed = cv.resize(image,(columns,rows),interpolation = cv.INTER_LINEAR)
    return zoomed
```

To calculate SSD:

$$SSD = \frac{1}{3WH}\sum_{k=1}^{3}\sum_{i=1}^{W}\sum_{j=1}^{H}\left(\frac{img1 - img2}{255}\right)^2$$

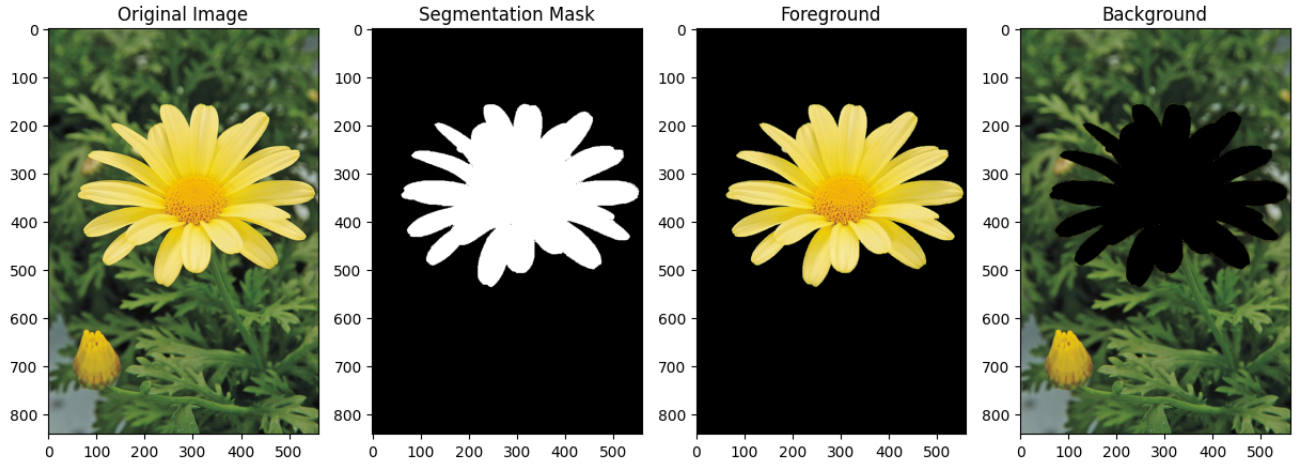
```
def ssd(img1, img2):
    if img1.shape!=img2.shape:
        return -1
    img1 = img1.astype(np.float32)
    img2 = img2.astype(np.float32)
    ssd = np.sum((img1 - img2) ** 2)/(255**2)
    ssd /= (3 * img1.shape[0] * img1.shape[1])
    return ssd
```

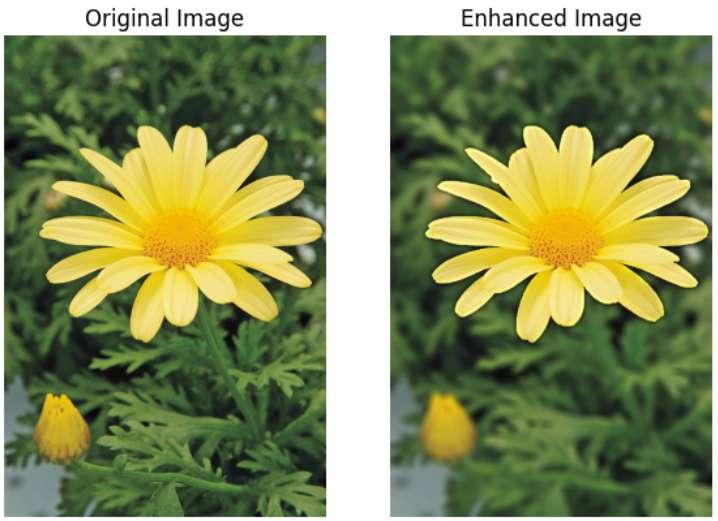
Nearest neighbor:	Bilinear interpolation:
image- 0 : 0.0020956409611371375 image- 1 : 0.0004067062169820724 image- 2 : The shapes of the two input images do not match image- 3 : 0.012111156994065278 image- 4 : 0.004395867116636051 image- 5 : 0.002885211291003935 image- 6 : 0.0028874577085736255 image- 7 : The shapes of the two input images do not match image- 8 : 0.0008133105215942586 image- 9 : The shapes of the two input images do not match image- 10 : The shapes of the two input images do not match	image- 0 : 0.0017699631195978755 image- 1 : 0.0002821362688139889 image- 2 : The shapes of the two input images do not match image- 3 : 0.01203611906151955 image- 4 : 0.00429517194241531 image- 5 : 0.0024873874087680743 image- 6 : 0.00281779251569909 image- 7 : The shapes of the two input images do not match image- 8 : 0.0005466325344952797 image- 9 : The shapes of the two input images do not match image- 10 : The shapes of the two input images do not match

Comparing the 2 methods, bilinear interpolation has lower SSDs than the nearest neighbor. This is because bilinear interpolation takes the distance to the four nearest pixel centers to calculate the output pixel value, while nearest neighbor interpolation assigns the value of the closest pixel center. But bilinear interpolation is computationally expensive.

Question 09

```
mask = np.zeros(img_original.shape[:2], np.uint8)
rect = (0, 90, 560, 500)
bgdModel = np.zeros((1,65), np.float64)
fgdModel = np.zeros((1,65), np.float64)
cv.grabCut(img_original, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)

mask_foreground = np.where((mask == 0) | (mask == 2), 0, 1).astype("uint8")
mask_background = np.where((mask == 1) | (mask == 3), 0, 1).astype("uint8")
forground_img = np.multiply(img_original, mask_foreground[:,:,np.newaxis])
background_img = np.multiply(img_original, mask_background[:,:,np.newaxis])

img_rgb = cv.cvtColor(img_original, cv.COLOR_BGR2RGB)
forground_img_rgb = cv.cvtColor(np.uint8(forground_img), cv.COLOR_BGR2RGB)
background_img_rgb = cv.cvtColor(np.uint8(background_img), cv.COLOR_BGR2RGB)

img_blurred = cv.GaussianBlur(background_img_rgb, (15, 15))
img_enhanced = cv.add(img_blurred, forground_img_rgb)
```





The reason for the background just beyond the edge of the flower quite dark in the enhanced image:

The background image has black pixels where the flower was in the original image.

The Gaussian blur filter averages the black pixels with the nearby background pixels at the border of the mask.

The blurred background image has darker pixels at the edge of the mask than the original background image.

So when the foreground image is added to the blurred background image, a darker edge around the flower is there.