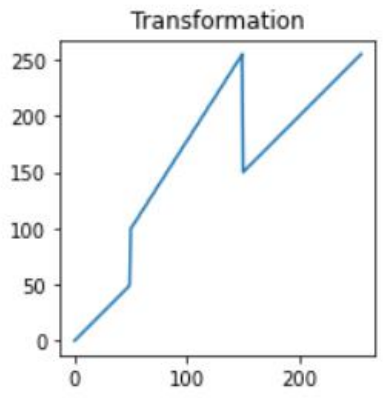
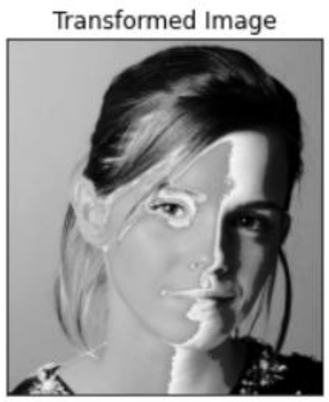
EN2550 Assignment 1

Intensity Transformations and Neighborhood Filtering

Treshan Ayesh 190443T

Q1) Applying the transformation given in the figure 1 to the original image in the figure 2. The pixels in the range 50 to 150 were transformed according to the given configuration. The output result can be seen in the final image of the following diagrams.





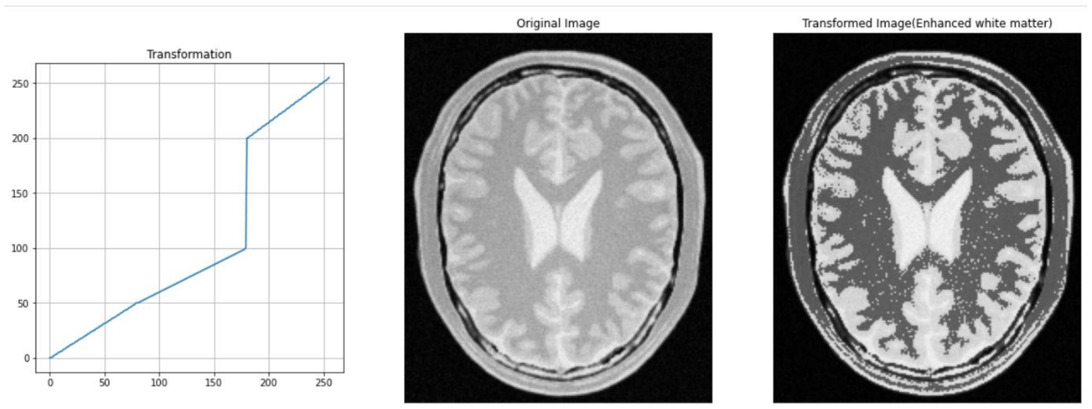
Code snippet for applying the intensity transformations using Look Up Tables.

```python
t1 = np.linspace(0, 50, 50)
t2 = np.linspace(50, 100, 0)
t3 = np.linspace(100, 255, 100)
t4 = np.linspace(255, 150, 0)
t5 = np.linspace(150, 255, 106)

transform = np.concatenate((t1,t3,t5,t2,t4), axis = 0).astype(np.uint8)
transformed_img = cv.LUT(img, transform)
```

Q2) (a)

Applying intensity transformations to accentuate the white matter of the brain image. The transformation in the figure 1 was applied to the original image in the second image, and in the resulting image, white matter area has been emphasized more

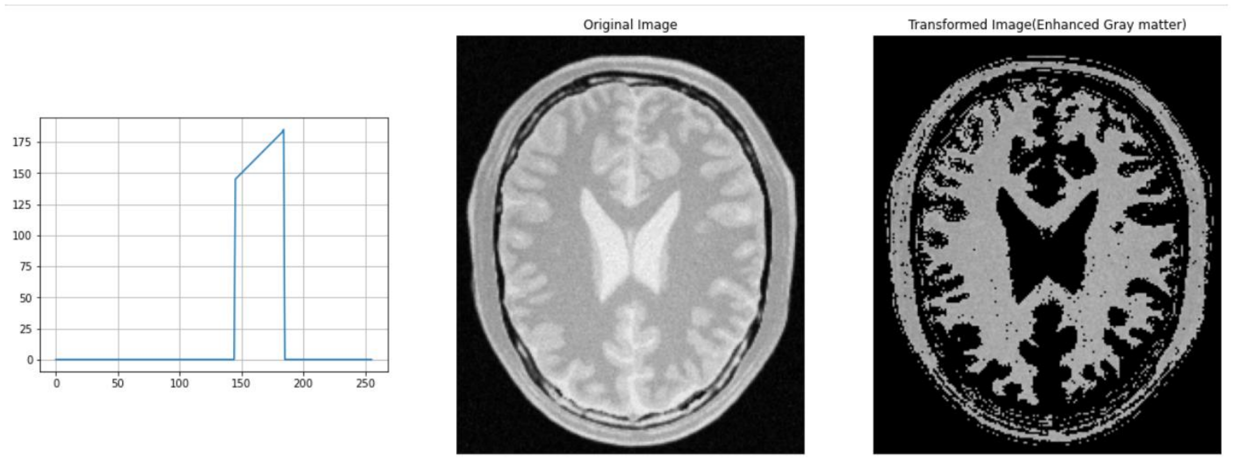
The applied transformation maps values around 185 to 255 values more towards 255 regions. So, in the resulting image white colors are emphasized more, which corresponds to the white matter in the brain.



The code snippet for setting up the intensity transformation to accentuate white matter is given below.

```
t1 = np.linspace(0, 50, 80)
t2 = np.linspace(50, 100, 100)
t3 = np.linspace(200, 255, 76)

transform_white = np.concatenate((t1,t2, t3), axis = 0).astype(np.uint8)
transformed_img1 = cv.LUT(img, transform_white)
```

b) In the second part grey matter of the brain has to be accentuated. Same has before an intensity transformation has been done and it is show in the following figure.



And the relevant code for the transformation is shown below

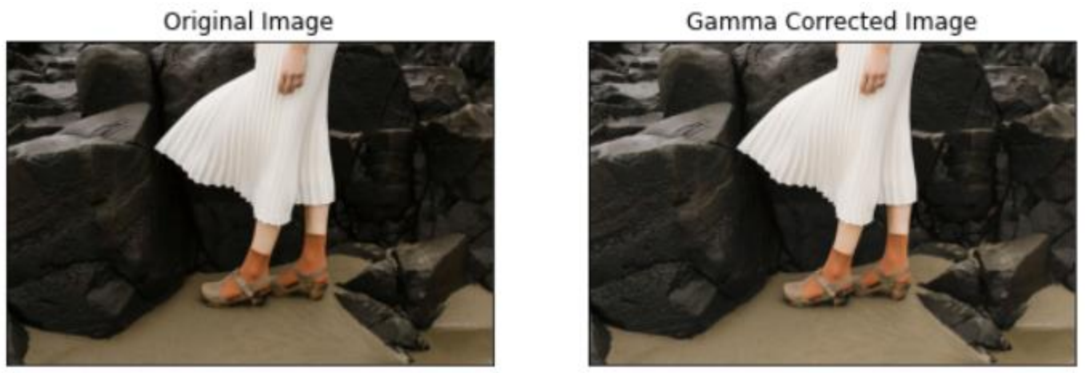
```
t1 = np.linspace(0, 0, 145)
t2 = np.linspace(0, 145, 0)
t3 = np.linspace(145, 185, 40)
t4 = np.linspace(185, 0, 0)
t5 = np.linspace(0, 0, 71)

transform = np.concatenate((t1,t2,t3,t4,t5), axis = 0).astype(np.uint8)
transformed_img = cv.LUT(img, transform)
```

Q3) Applying Gamma correction and plotting the histogram. The given image has to be separated into L, a, b color spaces. Split function is used to split the image into its respective color spaces. L plane represents the lightness of the image in LAB color space. After that gamma correction for L plane is done as follows.

```
img = cv.imread("highlights_and_shadows.jpg")
assert img is not None
img_Lab = cv.cvtColor(img, cv.COLOR_BGR2Lab)
(L, a, b) = cv.split(img_Lab)

gamma = 0.8
t= np.array([(p/255) ** gamma * 255 for p in range(0,256)]).astype('uint8')
image_transformed = cv.LUT(L , t)

merged = cv.merge([image_transformed, a, b])
```

Finally, gamma corrected L plane is merged back to a, b planes to get the final output image.

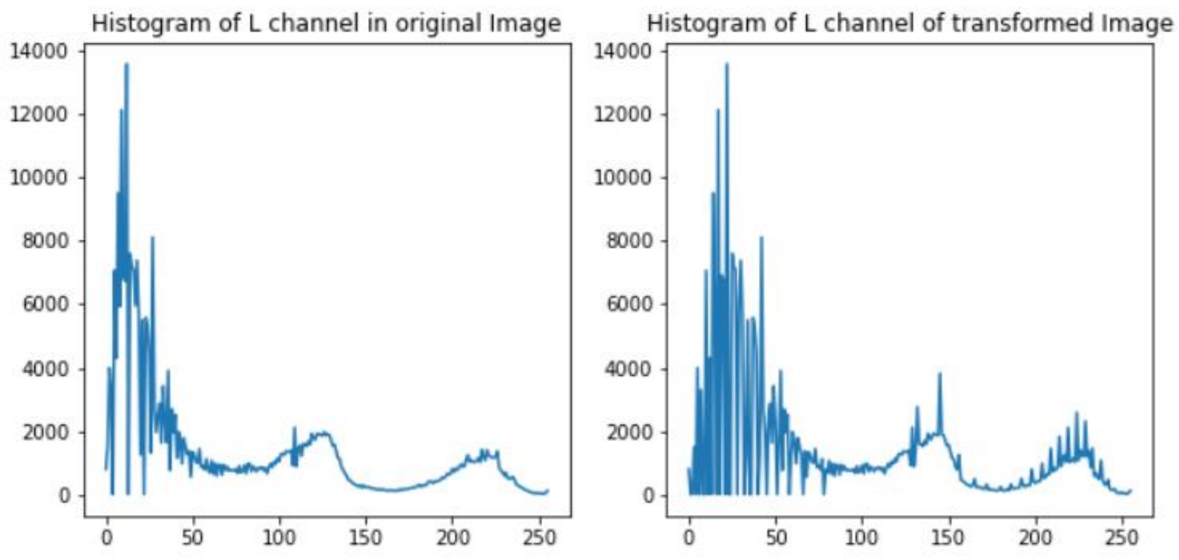
The final results are as follows. We can see that the shadows of the Gamma corrected image is much more visible now. If we decrease the gamma value more, the lightness of the image increases.



Next the histograms of the L planes of two images were obtained using ‘calcHist’ function in OpenCV as follows.

```
hist_L = cv.calcHist([L], [0], None, [256], [0,256])
ax = plt.subplot(gs[1, 0])
ax.set_title('Histogram of L channel in original Image')
plt.plot(hist_L)

hist_L_t = cv.calcHist([image_transformed], [0], None, [256], [0,256])
ax = plt.subplot(gs[1, 1])
ax.set_title('Histogram of L channel of transformed Image')
plt.plot(hist_L_t)
plt.show()
```

The results are as follows.



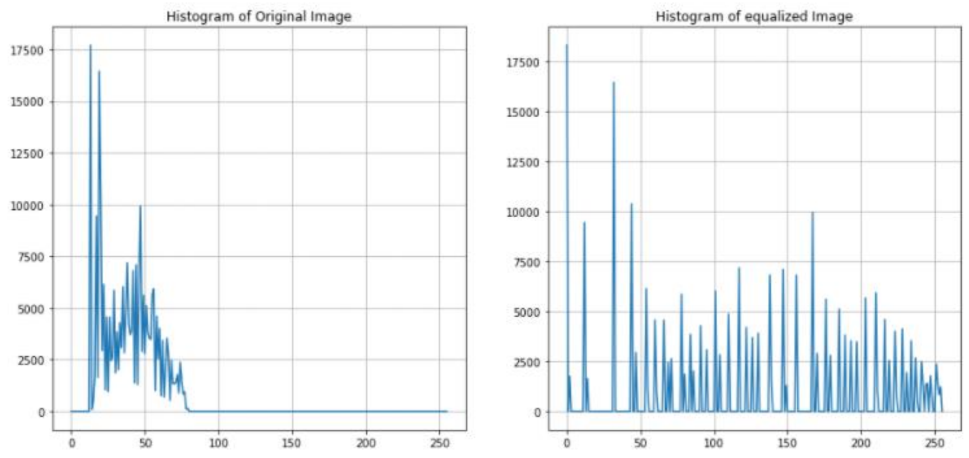
Q4) Applying Histogram equalization on an image.

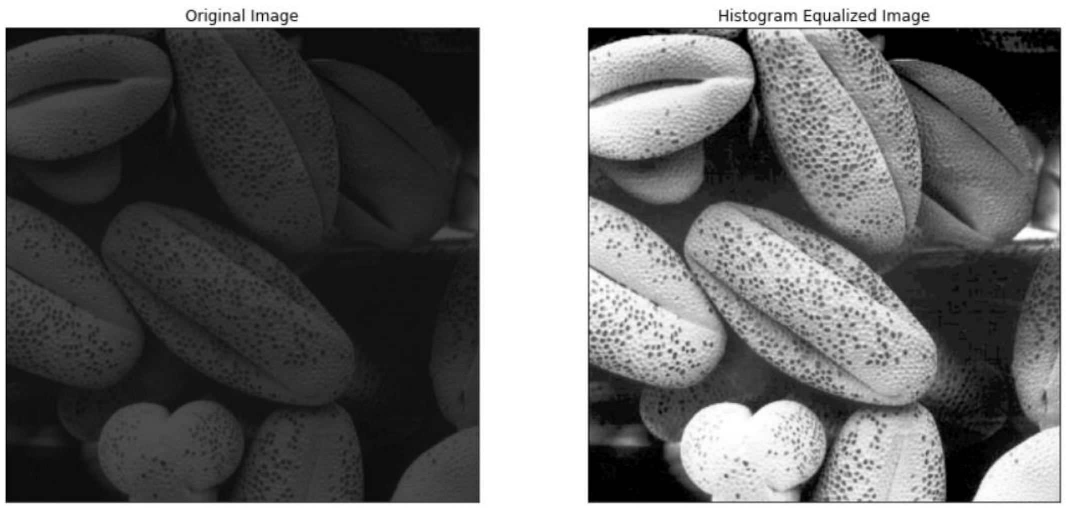
Histogram equalization will increase the contrast of the image. It will spread out the intensities of most frequent pixel values and stretches out the intensity range. Histogram equalization can be easily carries out by OpenCV function **cv.equalizeHist(),** but we are asked to write our own function for this.

The code is as follows.

```python
def hist_eq(img: np.ndarray):
    histOrig, bins = np.histogram(img.flatten(), 256, [0, 255])
    cdf = histOrig.cumsum()
    cdf_m = np.ma.masked_equal(cdf, 0)
    cdf_m = (cdf_m - cdf_m.min()) * 255 / (cdf_m.max() - cdf_m.min())
    cdf = np.ma.filled(cdf_m, 0)
    imgEq = cdf[img.astype('uint8')]
    histEq, bins2 = np.histogram(imgEq.flatten(), 256, [0, 256])

    return imgEq, histOrig, histEq
```

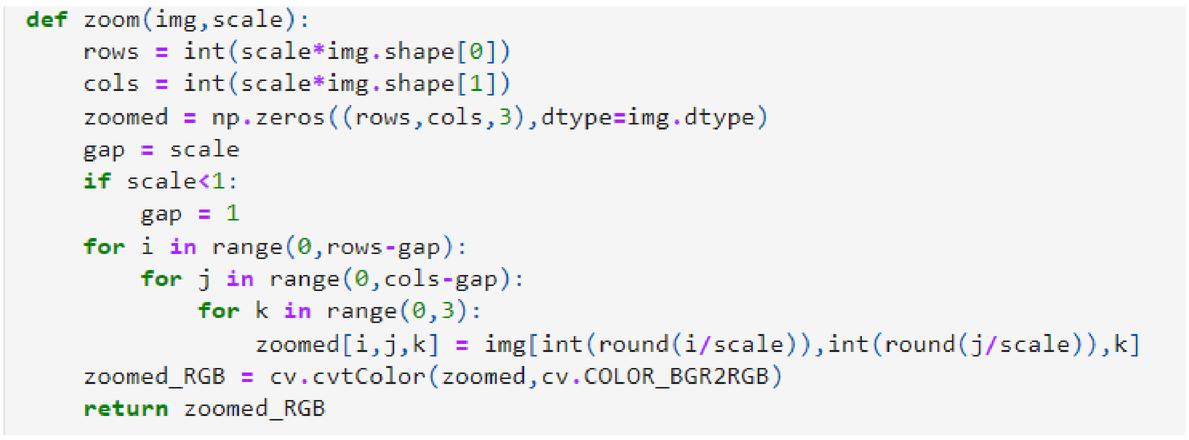
Here the cumulative sum of the histogram of the image is taken. And then it is normalized by dividing by the max value and multiplying by 255. The results can seen below.





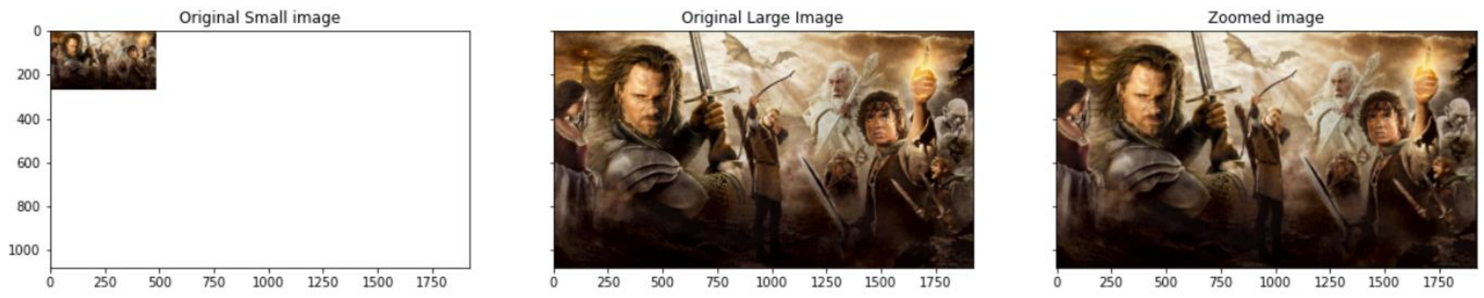
Q5) Program to zoom images by a given factor.

As given in the question, zooming has to be done in two ways. First the nearest neighbor method. Here, the pixel value of the zoom image depends on the pixel value of the pixel which is nearest to the pixel value corresponds to the original image. The code is given below.



```
def zoom(img,scale):
    rows = int(scale*img.shape[0])
    cols = int(scale*img.shape[1])
    zoomed = np.zeros((rows,cols,3),dtype=img.dtype)
    gap = scale
    if scale<1:
        gap = 1
    for i in range(0,rows-gap):
        for j in range(0,cols-gap):
            for k in range(0,3):
                zoomed[i,j,k] = img[int(round(i/scale)),int(round(j/scale)),k]
    zoomed_RGB = cv.cvtColor(zoomed,cv.COLOR_BGR2RGB)
    return zoomed_RGB
```

The results comparing the original small image, zoomed image using the above function and the given large image can be seen below.



The next method is bilinear interpolation. For bilinear interpolation, the pixel value of a pixel in the scaled image depend on the linear interpolation of pixel values of the closest pixels in the original image, to that corresponding pixel in scaled image.

The code for calculation of zoomed image by bilinear interpolation can be given as follows.

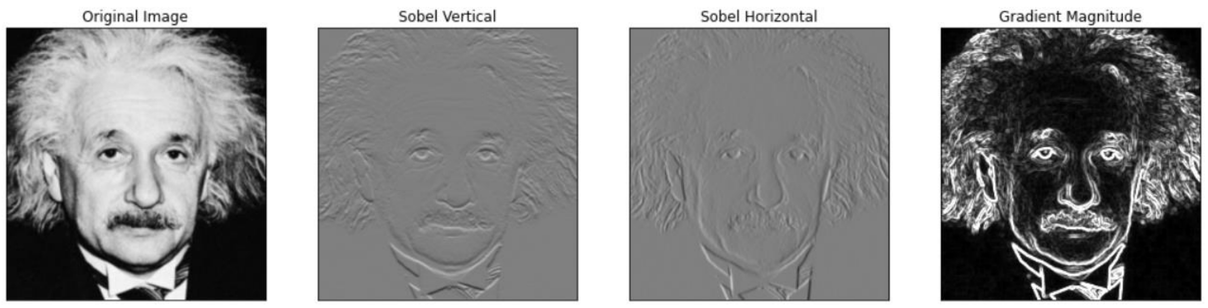
```
def zoom(img,scale,img_large):
    rows = int(scale*img.shape[0])
    cols = int(scale*img.shape[1])
    zoomed = np.zeros((rows,cols,3),dtype=img.dtype)
    gap = scale
    if scale<1:
        gap = 1
    ssd = 0
    pad_img = cv.copyMakeBorder(img, 0, 1, 0, 1, cv.BORDER_REPLICATE)
    for i in range(0,rows):
        for j in range(0,cols):
            for k in range(0,3):
                x = i/scale - math.floor(i/scale)
                y = j/scale - math.floor(j/scale)
                tl = pad_img[math.floor(i/scale),math.floor(j/scale),k]
                bl = pad_img[math.floor(i/scale),math.ceil(j/scale),k]
                tr = pad_img[math.ceil(i/scale),math.floor(j/scale),k]
                br = pad_img[math.ceil(i/scale),math.ceil(j/scale),k]
                avg_pix = int((br*(1-x)+bl*(x))*(1-y)+(tr*(1-x)+tl*(x))*(y))
                zoomed[i,j,k] = avg_pix
                ssd += (avg_pix - img_large[i,j,k])**2
    zoomed_RGB = cv.cvtColor(zoomed,cv.COLOR_BGR2RGB)
    return zoomed_RGB,ssd
```

The SSD calculated is aroun
d 1612087470 for the biline
ar interpolation.

Q6) a) Using the sobel operator.

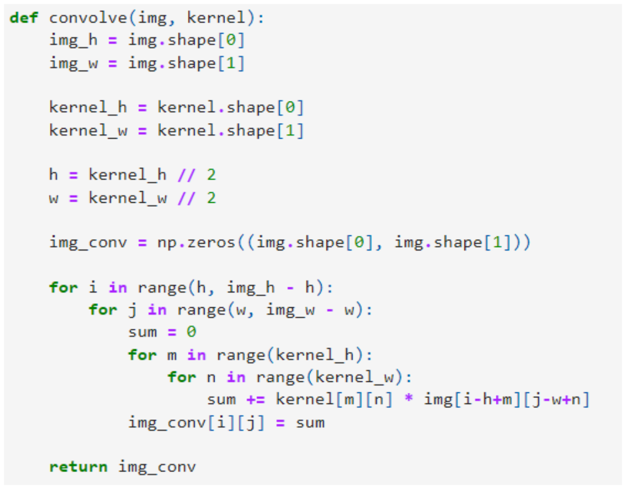
Filter2D function of OpenCV is used to sobel filter the given image.

```
img = cv.imread("einstein.png", cv.IMREAD_GRAYSCALE).astype(np.float32)
assert img is not None

#Sobel vertical
kernel_v = np.array([(-1,-2,-1),(0,0,0),(1,2,1)], dtype = 'float')
imgc_v = cv.filter2D(img, -1, kernel_v)

#Sobel Horizontal
kernel_h = np.array([(-1,0,1),(-2,0,2),(-1,0,1)], dtype = 'float')
imgc_h = cv.filter2D(img, -1, kernel_h)

#gradient Magnitude
grad_mag = np.sqrt(imgc_v ** 2 + imgc_h ** 2)
```

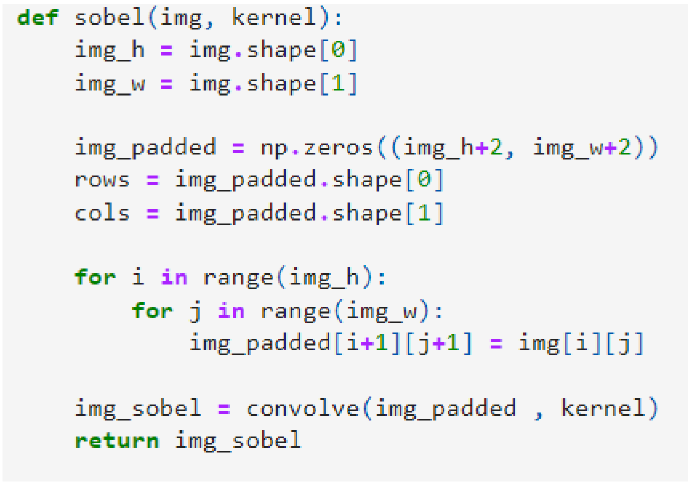


In filter 2D function the relevant sobel kernel is convoluted with the image matrix. The output result enhances the vertical edges in sobel horizontal filter while the sobel vertical filter enhances the horizontal edges. Finally the gradient is calculates by squaring and summing both and then taking the square root.

b)

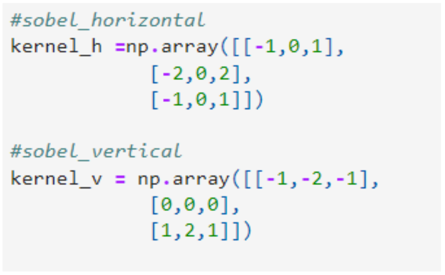


```python
def convolve(img, kernel):
    img_h = img.shape[0]
    img_w = img.shape[1]

    kernel_h = kernel.shape[0]
    kernel_w = kernel.shape[1]

    h = kernel_h // 2
    w = kernel_w // 2

    img_conv = np.zeros((img.shape[0], img.shape[1]))

    for i in range(h, img_h - h):
        for j in range(w, img_w - w):
            sum = 0
            for m in range(kernel_h):
                for n in range(kernel_w):
                    sum += kernel[m][n] * img[i-h+m][j-w+n]
            img_conv[i][j] = sum

    return img_conv
```



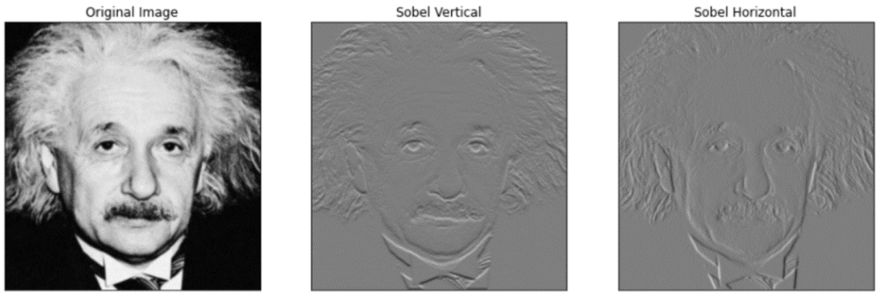
```python
def sobel(img, kernel):
    img_h = img.shape[0]
    img_w = img.shape[1]

    img_padded = np.zeros((img_h+2, img_w+2))
    rows = img_padded.shape[0]
    cols = img_padded.shape[1]

    for i in range(img_h):
        for j in range(img_w):
            img_padded[i+1][j+1] = img[i][j]

    img_sobel = convolve(img_padded , kernel)
    return img_sobel
```

Two functions are used to pad the image matrix with zeroes and then to convolve through the image. Then the relevant sobel filter are passed manually for convolution operation.

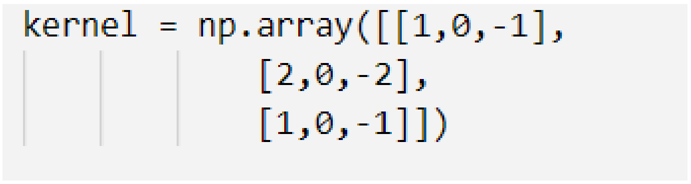


```python
#sobel_horizontal
kernel_h =np.array([[-1,0,1],
                    [-2,0,2],
                    [-1,0,1]])

#sobel_vertical
kernel_v = np.array([[-1,-2,-1],
                    [0,0,0],
                    [1,2,1]])
```

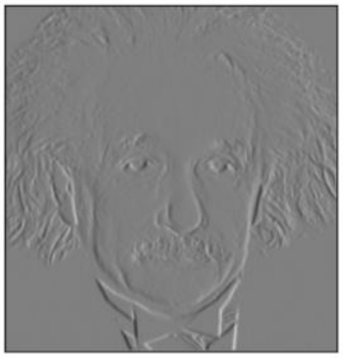
The resullts can be seen to be as same as before using the filter2D function of OpenCV.



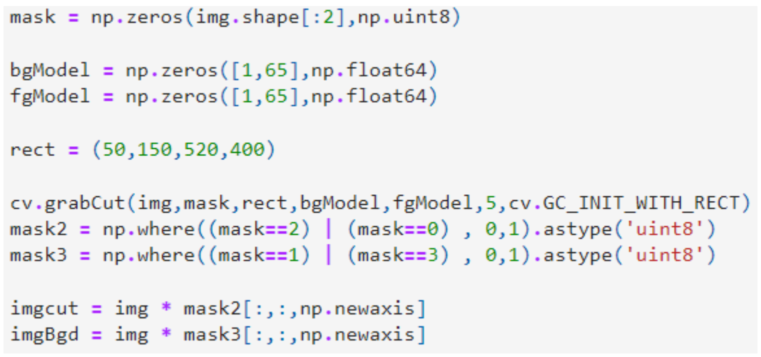
In the third part a given kernel is used to convolve with the image matrix. And the results obtained is as follows.



```python
kernel = np.array([[1,0,-1],
                   [2,0,-2],
                   [1,0,-1]])
```



Q7) Here the OpenCV grabCut function needs to be used to segment the given image as foreground and background. The code is as follows.

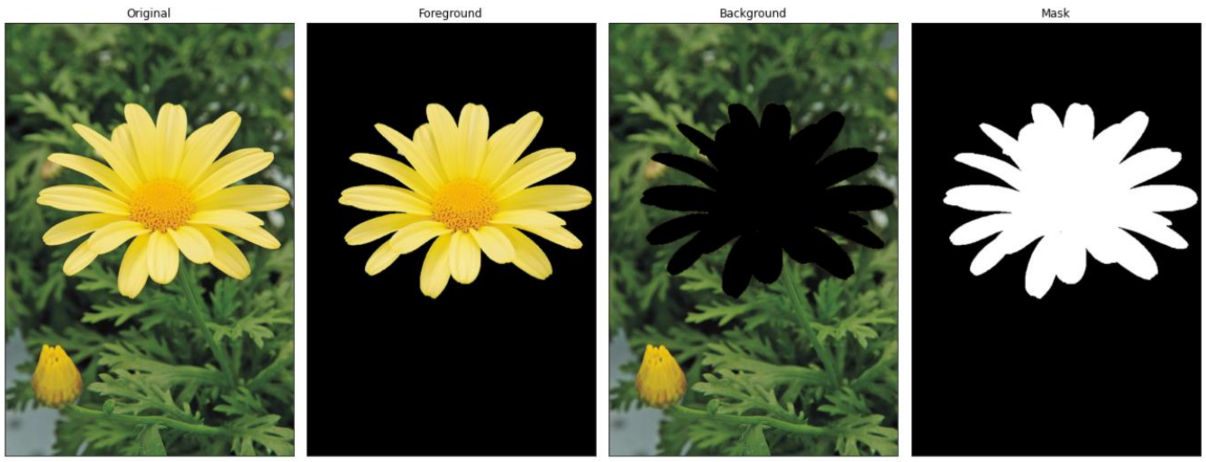


```
mask = np.zeros(img.shape[:2],np.uint8)

bgModel = np.zeros([1,65],np.float64)
fgModel = np.zeros([1,65],np.float64)

rect = (50,150,520,400)

cv.grabCut(img,mask,rect,bgModel,fgModel,5,cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask==2) | (mask==0) , 0,1).astype('uint8')
mask3 = np.where((mask==1) | (mask==3) , 0,1).astype('uint8')

imgcut = img * mask2[:,:,np.newaxis]
imgBgd = img * mask3[:,:,np.newaxis]
```

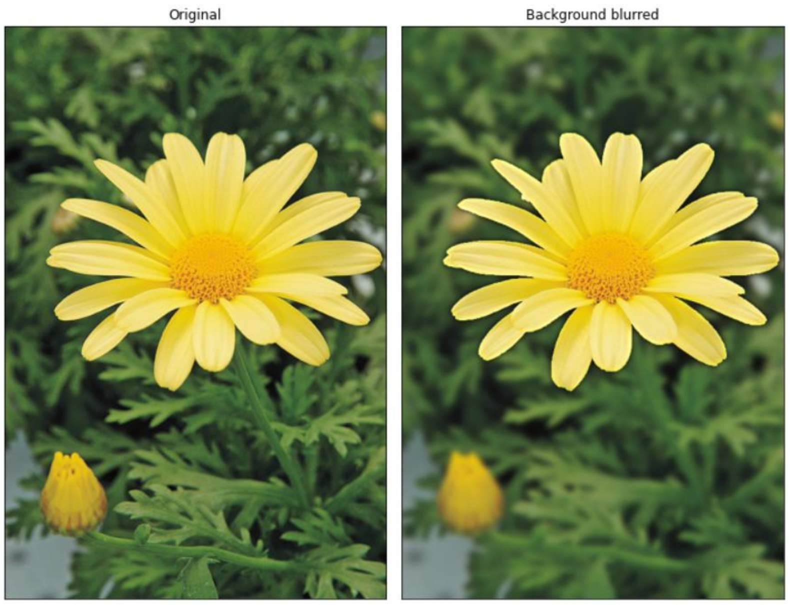
The rectangular mask is provided as a parameter for the function and the function takes pixels out of the rectangle as sure background and assign labels.

The foreground, background and the mask can be shown as below.

In the next part, gaussian blur is applied to the filtered-out background and then merged with the filtered out foreground.



The reason for having a dark edge at the background is, when applying the gaussian blur to the filtered-out background it blurs the black region as well. Therefore, at the edge of the flower the green pixels combine with black pixels and when we combine all together, it is much more visible.



GitHub link –

https://github.com/TreshanAyesh/EN2550-Fundamentals-of-Image-Processing.git