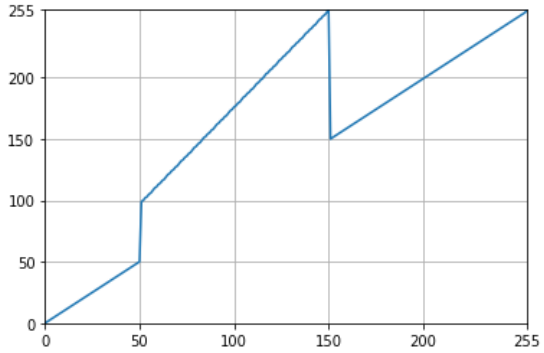
NAME: N. KRISHNAKANTH

INDEX NO: 190323C

Q1)

0-50 and 150-255 ranges map to the same range in the output indicates the highly blackish and whitish regions in the image remains same. But since 50-150 mapped to 100-255, it indicates blackish gray region is mapped to whitish gray region and within that region whitish gray is mapped to even more white as the whiteness increases. Also 50-100 gray region is avoided in the output image and

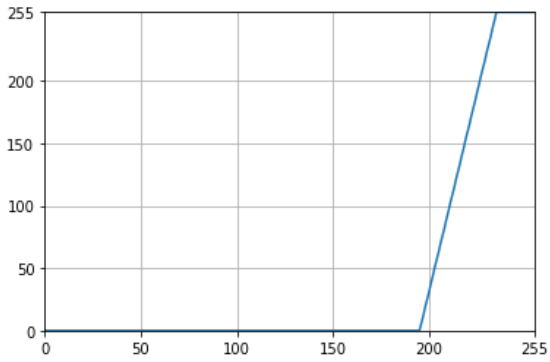
```
emma = cv.imread(r'emma_gray.jpg', cv.IMREAD_GRAYSCALE)
assert emma is not None

t1 = np.linspace(0, 50, 51)
t2 = np.linspace(99, 255, 100)
t3 = np.linspace(150, 255, 105)

t = np.concatenate((t1, t2, t3), axis=0).astype(np.uint8)

assert len(t) == 256

emma_intense = cv.LUT(emma, t)
emma_intense = cv.cvtColor(emma_intense, cv.COLOR_BGR2RGB)
emma = cv.cvtColor(emma, cv.COLOR_BGR2RGB)
```

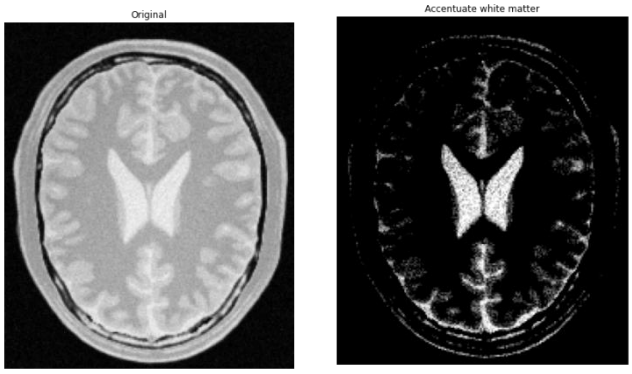


Q2)

a) To accentuate white matter, all blackish and blackish gray area are mapped to black and small range of white at the rightmost values mapped to 255. Small range of whitish gray is mapped to the whole range to make the visibility clear.

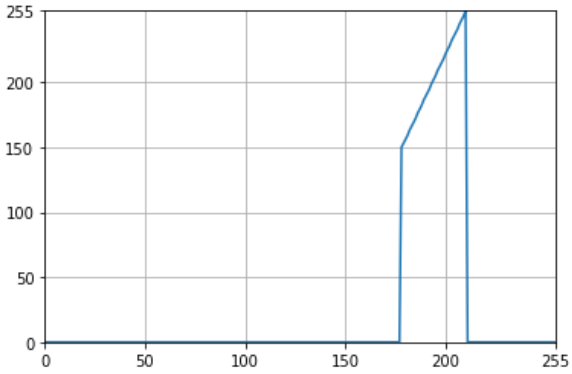
```
brain = cv.imread(r'brain_proton_density_slice.png', cv.IMREAD_GRAYSCALE)
assert brain is not None

t1 = np.linspace(0,00, 195)
t2 = np.linspace(0,255, 41)
t3 = np.linspace(255,255, 20)

t = np.concatenate((t1, t2, t3), axis=0).astype(np.uint8)

assert len(t) == 256

brian_at_white = cv.LUT(brain, t)
brian_at_white = cv.cvtColor(brian_at_white, cv.COLOR_BGR2RGB)
brain = cv.cvtColor(brain, cv.COLOR_BGR2RGB)
```

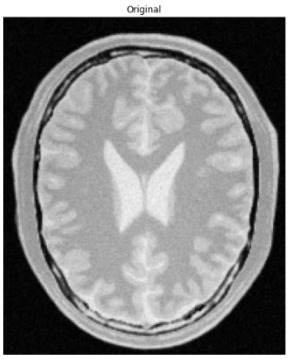


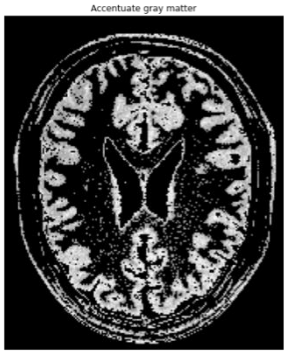


b) Gray matter lies in the mid-range. Thus, we should attenuate highly black and white regions. As shown in the graph black and white regions mapped to black. After experimentation the range correspond to gray matter became nearly 180-200. So, this region is mapped to a brighter region in a linear manner for visibility.

```
t1 = np.linspace(0, 0, 178)
t2 = np.linspace(150, 255, 33)
t3 = np.linspace(0,0, 45)

t = np.concatenate((t1, t2, t3), axis=0).astype(np.uint8)

assert len(t) == 256

brian_at_white = cv.LUT(brain, t)
brian_at_white = cv.cvtColor(brian_at_white, cv.COLOR_BGR2RGB)
brain = cv.cvtColor(brain, cv.COLOR_BGR2RGB)
```



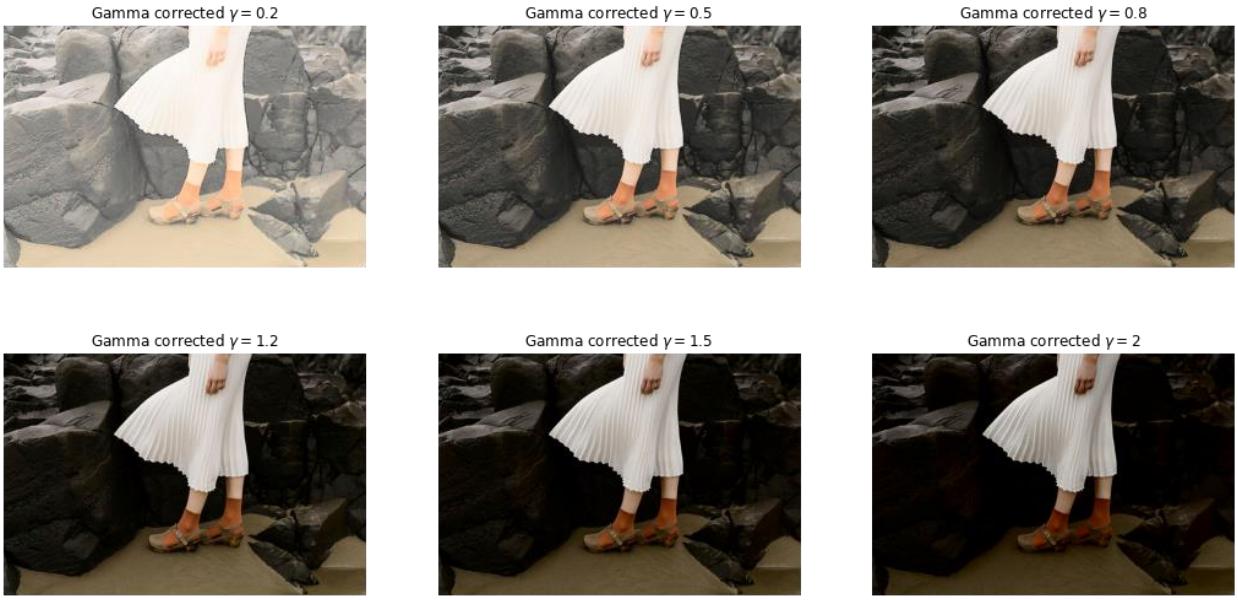




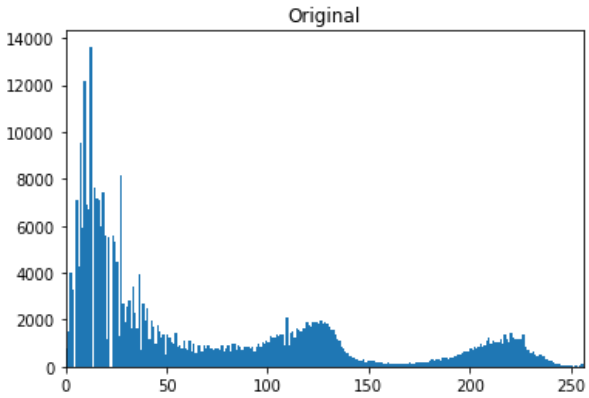
Q3)

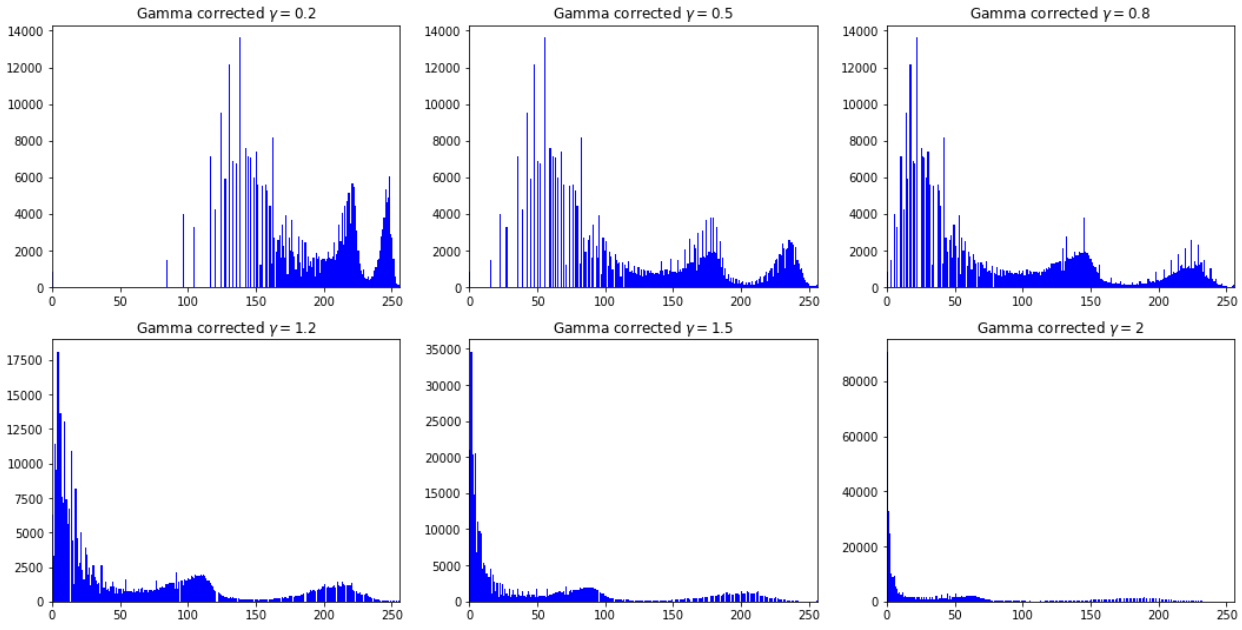
a) In the L*a*b* plane of an image the L indicates the lightness of the image. Increasing the value to maximum results in white and the lowest value represents black. Choosing the value of L, softens the brightness of the respective color. In here the L plane's intensity value is gamma corrected and observed as the gamma value ranges [0.2, 0.5, 0.8, 1.2, 1.5, 2]. Reducing the gamma increase the brightness of the image and undetectable details of the original image can be observed here. But further reducing it near to 0 results in bright image. On the other hand, increasing the gamma value darkens the image hiding the already visible details.

```
import copy
gamma_img = cv.imread(r'highlights_and_shadows.jpg')
gamma_lab = cv.cvtColor(gamma_img, cv.COLOR_BGR2LAB)
gamma_rgb = cv.cvtColor(gamma_lab, cv.COLOR_LAB2RGB)

plt.imshow(gamma_rgb)
plt.axis('off')
plt.title('Original')

gamma_list = [ 0.2, 0.5, 0.8, 1.2, 1.5, 2]
j=0
fig, ax = plt.subplots(2, 3, figsize=(18, 9))

for gamma in gamma_list:
    r=j//3
    c=j%3

    img_copy = copy.deepcopy(gamma_lab)

    t = np.array([(p/255)**gamma*255 for p in range (0, 256)]).astype(np.uint8)
    img_copy[:,:,0] = cv.LUT(img_copy[:,:,0], t)
    img_copy = cv.cvtColor(img_copy, cv.COLOR_LAB2RGB)

    ax[r][c].imshow(img_copy)
    ax[r][c].set_title('Gamma corrected $\gamma = {}$'.format(gamma))
    ax[r][c].axis('off')

    j+=1
plt.show()
```



b) Histograms below shows the number of pixels in the L plane with the given intensity. As we can see when the gamma value increases the number of low value pixels which corresponds to blackness increases (number of high pixel values decreases). It is what exactly observed above pictures. But the other two a* and b* planes won't change.





Q4)

The histogram equalization is carried out according to the following formula.

$$s_k = \frac{(L-1)}{MN}\sum_{j=0}^{k} n_j$$

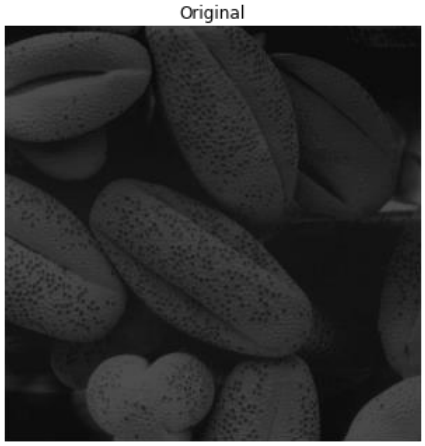
$$MN = Total\ number\ of\ pixesls,$$

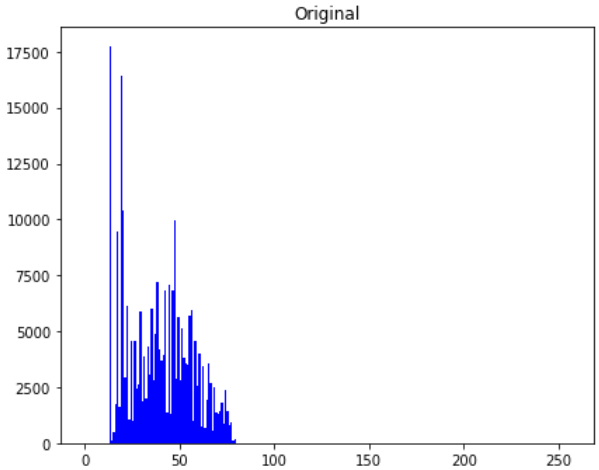
$$L = Number\ of\ possible\ intensity\ levels$$

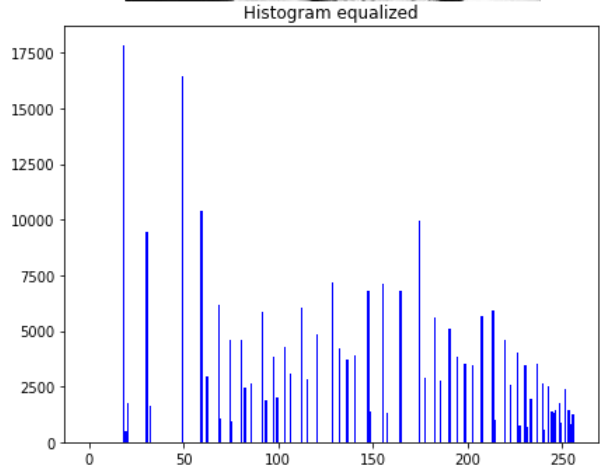
$$n_k = num\ er\ of\ pixels\ with\ k\ intensity\ level.$$

Histogram equalization produces an image with a flat histogram. Both the original and transformed image has same content but different intensity levels for different portions of the image. It makes it more visible and it enables to detect more minute details.

```python
def histogram_equalization(image):
    hist , bins = np.histogram(image.ravel(), 256, [0, 256])
    cdf = hist.cumsum()
    cdf_normalized = np.round(cdf*255/cdf.max()).astype(np.uint8)
    enhanced = cv.LUT(image, cdf_normalized)
    return enhanced

hist_img = cv.imread(r'shells.png', cv.IMREAD_GRAYSCALE).astype(np.uint8)
enhanced = histogram_equalization(hist_img)

fig, ax = plt.subplots(2, 2, figsize=(15, 12))

ax[0,0].imshow(hist_img, cmap='gray', vmin=0, vmax=255)
ax[0,0].set_title('Original')
ax[0,0].axis('off')

ax[0,1].imshow(enhanced, cmap='gray', vmin=0, vmax=255)
ax[0,1].set_title('Histogram equalized')
ax[0,1].axis('off')

ax[1,0].hist(hist_img.flatten(), 256, [0,256], color='b')
ax[1,0].set_title('Original')

ax[1,1].hist(enhanced.flatten(), 256, [0,256], color='b')
ax[1,1].set_title('Histogram equalized')

plt.show()
```

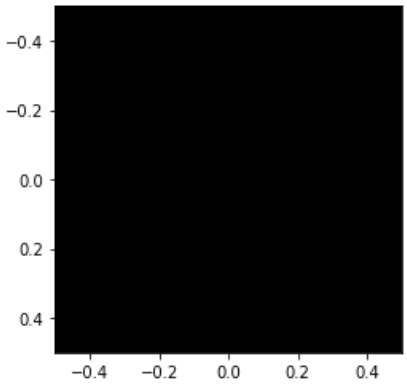






Q5)

For nearest neighbor and bilinear interpolation method the following functions were applied. In both cases the given small image is zoomed out 4 times and compared with the given zoomed out version of the image. Comparison is made by means of normalized sum of squares method. From this method if the template (reference image) and the output image are exactly same, it should produce 0 for every pixel value and thus it should be a plane black image. That is what obtained when the result is observed as shown below.




```python
def nearest_neighbour(image, scale):
    row = int(np.round(image.shape[0]*scale))
    column = int(np.round(image.shape[1]*scale))
    zoomed = np.zeros((row, column, 3), np.uint8)
    for i in range(0, row):
        for j in range(0, column):
            (x, y) = int(np.round(i/scale)), int(np.round(j/scale))
            if x == image.shape[0]:
                x = image.shape[0] -1
            if y == image.shape[1]:
                y = image.shape[1] -1
            zoomed[i, j] = image[x, y]
    return zoomed
zoom_img = cv.imread(r'./a1q5images/im01small.png').astype(np.uint8)
s=4
nn_zoomed = nearest_neighbour(zoom_img, s)
zoom_img2 = cv.imread(r'./a1q5images/im01.png').astype(np.uint8)

### calculate normalized SSD of zoomed and given reference image
R = cv.matchTemplate(zoom_img2, nn_zoomed, eval('cv.TM_SQDIFF_NORMED'))
plt.imshow(R, cmap='gray')
plt.show()
```

```python
def bilinear_interpolation(image, scale):
    row = int(np.round(image.shape[0]*scale))
    column = int(np.round(image.shape[1]*scale))
    zoomed = np.zeros((row, column, 3), np.uint8)
    for i in range(0, row):
        for j in range(0, column):
            (x, y) = i/scale, j/scale
            x_floor = int(np.floor(x))
            x_ceil = x_floor + 1
            y_floor = int(np.floor(y))
            y_ceil = y_floor +1
            if x_ceil == image.shape[0]:
                x_ceil = image.shape[0] - 1
            if y_ceil == image.shape[1]:
                y_ceil = image.shape[1] - 1
            x_floor_dist = x - x_floor
            x_ceil_dist = x_ceil - x
            y_floor_dist = y - y_floor
            y_ceil_dist = y_ceil - y
            left_weight = image[x_floor, y_floor]*x_ceil_dist + image[x_ceil, y_floor]*x_floor_dist
            right_weight = image[x_floor, y_floor]*x_ceil_dist + image[x_floor, y_ceil]*x_floor_dist
            zoomed[i, j] = np.round(left_weight*y_ceil_dist + right_weight*y_floor_dist).astype(np.uint8)

    return zoomed
```

Q6)

a) Here both horizontal and vertical Sobel filters are combined to give the gradient magnitude image for clear visibility of edges.

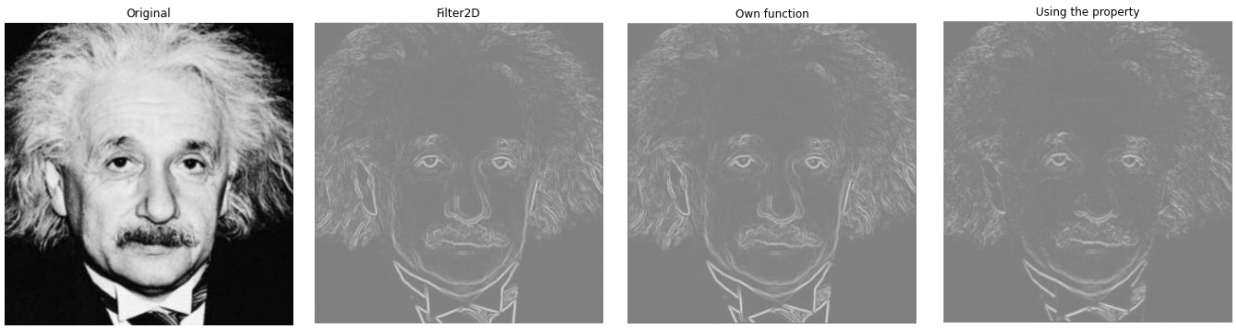
```
sobel_h = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=np.float32)
g_x = cv.filter2D(sobel_img, -1, sobel_h)

sobel_v = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.float32)
g_y = cv.filter2D(sobel_img, -1, sobel_v)

g = np.sqrt(g_x**2 + g_y**2)
```

b) Sobel filter, filters using convolution. So here for loop and dot product of matrices are used for convolution.

```
import math
def filter(image, kernel):
    assert kernel.shape[0]%2 == 1 and kernel.shape[1] % 2 == 1
    k_hh, k_hw = int(np.floor(kernel.shape[0]/2)), int(math.floor(kernel.shape[1]/2))
    h, w = image.shape
    image_float = image.astype('float')
    result = np.zeros(image.shape, 'float')

    for m in range(k_hh, h - k_hh):
        for n in range(k_hw , w - k_hw):
            result[m, n] = np.dot(image_float[m - k_hh : m + k_hh + 1 , n - k_hw : n+ k_hw + 1].flatten(), kernel.flatten())

    return result

f_x = filter(sobel_img, sobel_h)
f_y = filter(sobel_img, sobel_v)
f = np.sqrt(f_x**2 + f_y**2)
```

c)

```
sobel_v1 = np.array([[-1], [0], [1]], dtype=np.float32)
sobel_v2 = np.array([-1, 2, 1], dtype=np.float32)
img_v = cv.filter2D(sobel_img, -1, sobel_v1)
img_v = cv.filter2D(img_v, -1, sobel_v2)

sobel_h1 = np.array([[1], [2], [1]], dtype=np.float32)
sobel_h2 = np.array([1, 0, -1], dtype=np.float32)
img_h = cv.filter2D(sobel_img, -1, sobel_h1)
img_h = cv.filter2D(img_h, -1, sobel_h2)
```



Q7)

a) For ***grabcut*** function, the size of the rectangle box used to cover the foreground image is set after experimentation. ***bgdModel*** and ***fgdModel*** are used by the predefined algorithm which runs inside ***grabcut*** function. These two variables are set to ***float64*** type 1×65 array of zeros. Since a user defined mask and rectangle are given as input the mode should be given as ***cv.GC_INIT_WITH_RECT.***

```
mask = np.zeros(img_rgb.shape[:2],np.uint8)
bgdModel = np.zeros((1,65),np.float64)
fgdModel = np.zeros((1,65),np.float64)
rect = (50,120,550,450)
cv.grabCut(img_rgb,mask,rect,bgdModel,fgdModel,5,cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')
foreground = img_rgb*mask2[:,:,np.newaxis]
background = img_rgb - foreground
```

b) Here a kernel of size 19×19 is used for Gaussian blurring the background. Then the separate background and foreground images are combined to form the final image output.

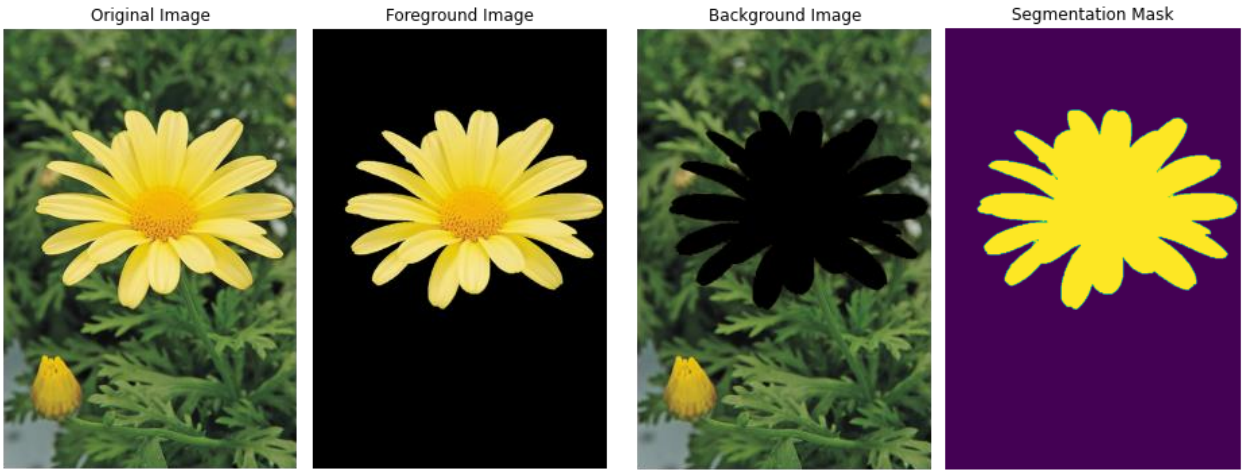
```
blurred_bg = cv.GaussianBlur(background, (19,19), 0)
enhanced = foreground + blurred_bg
```

c) As shown in the figures the background image has a black color in place of the foreground image. Since Gaussian blur is done through convolution, when blurring only the background at the edges of the foreground, black pixels are considered, and their impact causes the pixels in this edge altered. When the size of the gaussian kernel used increases, the distortion at the edges also increases.

Original Image

Blurred background

Enhanced image



GitHub link: https://github.com/Krishnakanth-lab/EN2550_Assignment1.git