EN2550 – Assignment 1

Intensity Transformations and Neighborhood Filtering

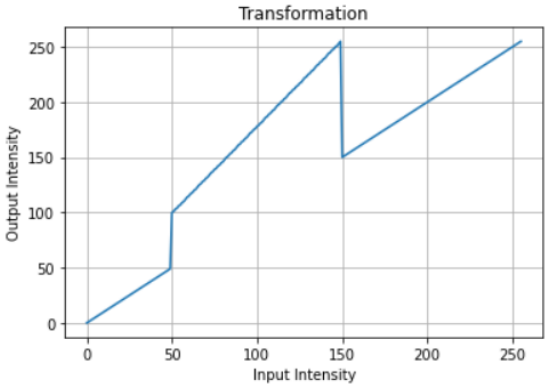
Index No: 190696U

Name: Wijegunawardana C.H.W.

Question 1 – Implementing the intensity transformation on an image

The transformation is generated as follows.

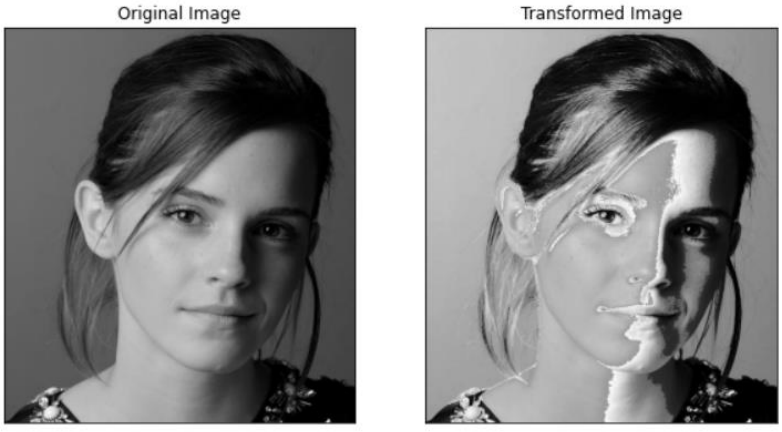
```
# Generate the transformation
transform = np.arange(0, 256, dtype = np.uint8)
transform[:50] = np.linspace(0, 50, 50, endpoint = False)
transform[50:150] = np.linspace(100, 255, 100, endpoint = True)
transform[150:256] = np.linspace(150, 255, 106, endpoint = True)
```



The *LUT* function of the OpenCV library is used to apply the transformation to the given image.

```
# Apply the transformation to the image
trans_img1 = cv.LUT(img1,transform)
```

The original image and the transformed image are depicted in the results obtained.



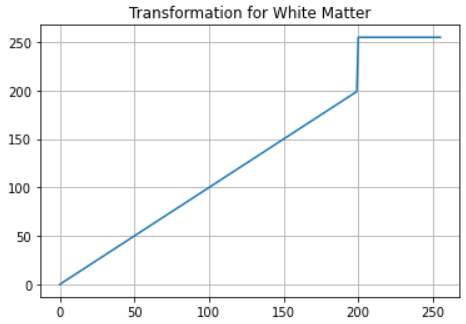
The pixels between 50-150 grayscale levels in the original image are enhanced using the transformation into higher gray-scale values. Therefore, those pixels have become whiter in the transformed image compared to the original image.

Question 2 – Accentuating white and gray matter in the image

To accentuate the white and gray matter in the grayscale image of the brain, the corresponding grayscale levels are transformed into 255 (absolute white).

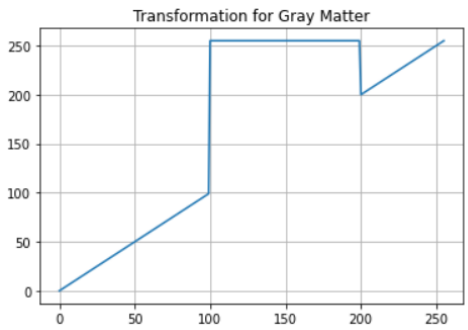
The transformation for the white matter

```
# Transformation for white matter
transform_white = np.arange(0, 256, dtype = np.uint8)
transform_white[:200] = np.linspace(0, 200, 200, endpoint = False)
transform_white[200:256] = 255
```

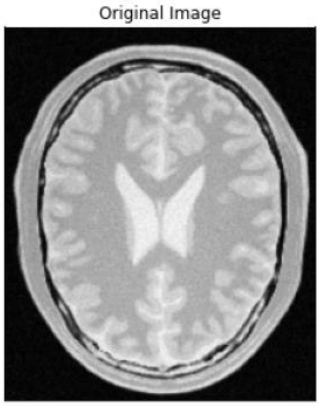


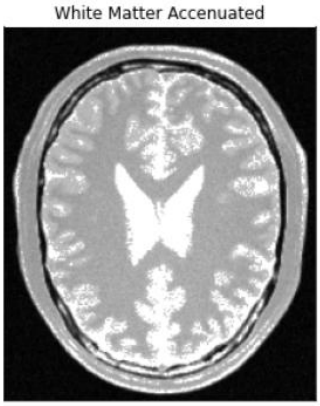
The transformation for the gray matter

```
# Transformation for gray matter
transform_gray = np.arange(0, 256, dtype = np.uint8)
transform_gray[:100] = np.linspace(0, 100, 100, endpoint = False)
transform_gray[100:200] = 255 #np.linspace(100, 255, 130, endpoint = True)
transform_gray[200:256] = np.linspace(200, 255, 56, endpoint = True)
```



The transformed images are depicted below.





To accentuate the white matter, the transformation is designed such that the output will have absolute white in near-white pixels for better visualization. The near-white grayscale range is selected as 200-255.
To accentuate the gray matter, the transformation is designed such that the output will have absolute white in gray pixels for better visualization. The gray grayscale range is selected as 50-200.

Question 3 – Gamma Correction to the L Plane

The gamma correction only needs to be applied to the L plane in the L*a*b* color space. This was done using the following segment of code.

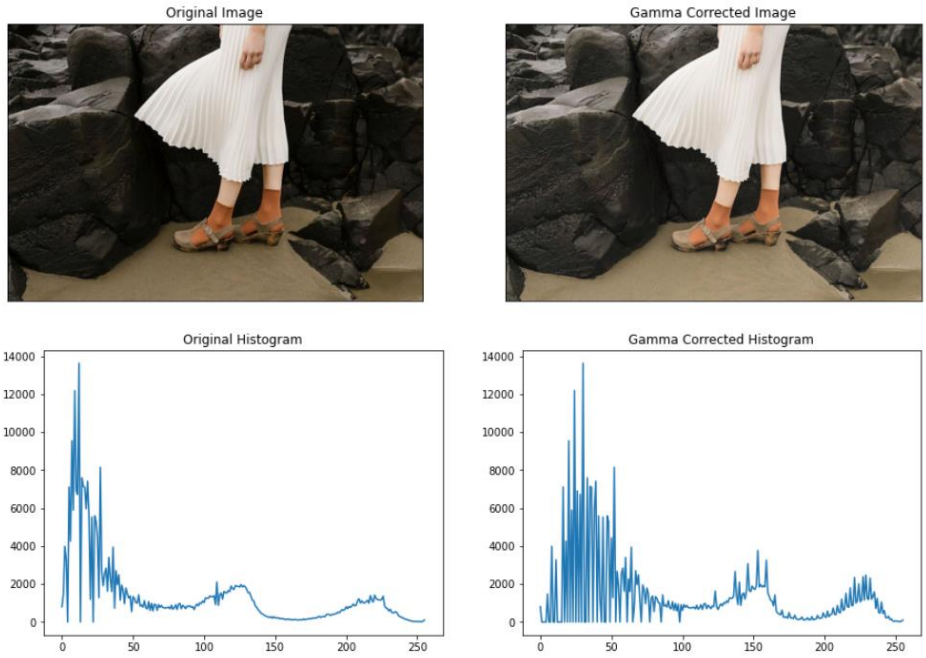
```
# Convert to L*a*b* color space
img3 = cv.cvtColor(img3, cv.COLOR_BGR2Lab)

gamma = 0.7
# Apply gamma correction
transform = np.array([(p/255)**gamma*255 for p in range(256)]).astype(np.uint8)
L_field = cv.LUT(img3[:,:,0],transform).reshape(480,720,1) # Applying Gamma correction to only the L fi
corrected = np.concatenate([L_field,img3[:,:,1:]],-1) # Reconstruct the image
```

First, the image is converted into the L*a*b* color space using OpenCV *cvtColor* function. Then the gamma transform is constructed for the selected gamma value. Then the L field is extracted from the image using array slicing, the transformation is applied and appended with the remaining a* and b* fields.

The histograms of the original image and the gamma-corrected image are generated using the OpenCV *calcHist* function.

```
# Original Histogram
histogram_ori = cv.calcHist([img3],[0],None,[256],[0,256])
# Corrected Histogram
histogram_cor = cv.calcHist([corrected],[0],None,[256],[0,256])
```

The results are depicted in the following figures.



The gamma value **0.7** is selected to obtain a spread-out histogram for the transformed image.

Question 4 – Histogram Equalization

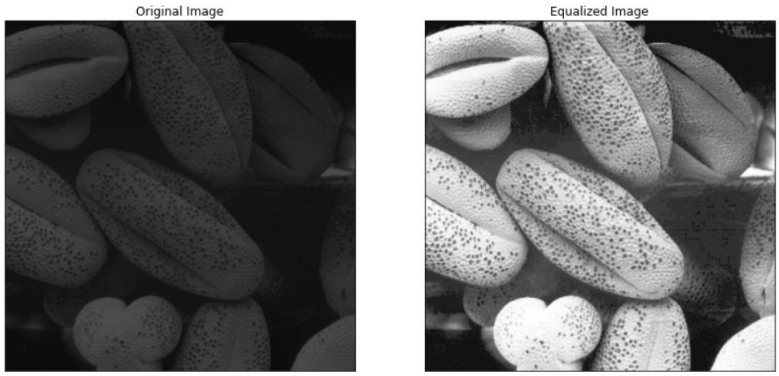
The histogram equalization is achieved by following the method below.

1. Calculating the cumulative distribution of the histogram of the original histogram.
2. Normalizing the CDF to values between 0-255.
3. Applying the CDF as a transformation to the original image.

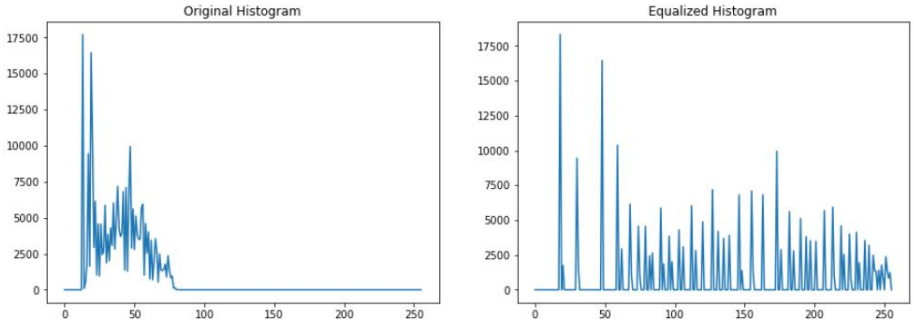
This effectively spreads out the concentrated areas of the histogram within the range 0-255.

```
# Equalize Image
histogram_ori = cv.calcHist([img4],[0],None,[256],[0,256]) # Find the original histogram
cdf_hist = np.cumsum(histogram_ori)                        # Calculate the cumulative function of the histogram
normalized_hist = (cdf_hist * 255 / cdf_hist.max()).astype(int) # Normalize the cdf such that the sum is 255
equalized = cv.LUT(img4, normalized_hist).astype('uint16') # Apply the normalized cdf as an LUT to the image
```

The original image and the histogram equalized image are depicted in the following figure.



The corresponding histograms are as follows.



It can be observed that, after the histogram equalization, the histogram is distributed evenly across the 0-255 grayscale values. The transformed image has more details.

Question 5 – Zooming

The small images given in the assignment images folder were used for zooming. They were compared with the zoomed images provided.
As instructed, the OpenCV function *resize* was used to do the zooming on the images using two different algorithms.

1. Nearest Neighbor
2. Bilinear Interpolation

To understand the behavior of the nearest neighbor algorithm, it was implemented using a written function as below.

```
def zoom_nearest_neighbour(scale, image):
    rows = int(scale*image.shape[0])
    columns = int(scale*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/scale),int(j/scale)]
    return zoomed
```

Bilinear interpolation was carried out using the *resize* function with *INTER_LINEAR* as the interpolation argument.

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

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

The small images are zoomed by a factor of 4 to be compared with the larger images.

Nearest Neighbor	**Bilinear Interpolation**
	
SSD = 0.0004811121335890121	SSD = 0.0004775562417232688

If observed closely, the image zoomed using the nearest neighbor method has a pixelated appearance because there are duplicate pixels. But the image zoomed using bilinear interpolation looks much smoother. The sum of squared differences is lower in the bilinear interpolation.

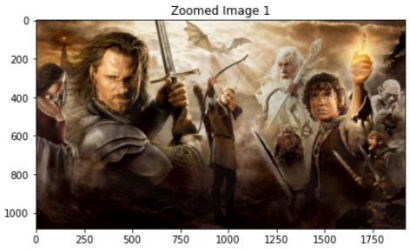
The SSD was calculated as follows.

```
# Calculating SSD for bilinear interpolation zooming
ssd_1 = np.sum(((img51[:,:]-img51_zb[:,:])**2)/(3*255**2))/(img51.shape[0]*img51.shape[1])
```

All the 3 images given were zoomed using the above two algorithms. The results are as follows.

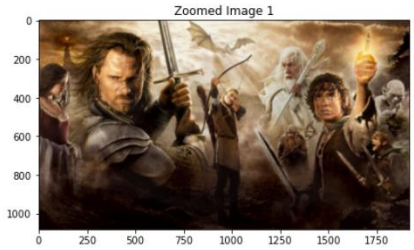
Nearest Neighbor

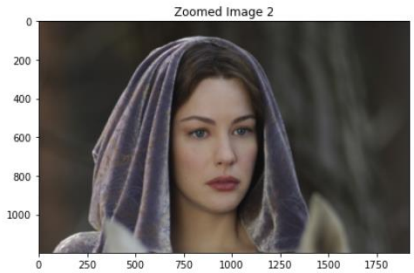
```
Original Image 1:  (270, 480, 3)
Original Image 2:  (300, 480, 3)
Original Image 3:  (365, 600, 3)
Zoomed Image 1:  (1080, 1920, 3)
Zoomed Image 2:  (1200, 1920, 3)
Zoomed Image 3:  (1460, 2400, 3)
```



Bilinear Interpolation

```
Original Image 1:  (270, 480, 3)
Original Image 2:  (300, 480, 3)
Original Image 3:  (365, 600, 3)
Zoomed Image 1:  (1080, 1920, 3)
Zoomed Image 2:  (1200, 1920, 3)
Zoomed Image 3:  (1460, 2400, 3)
```

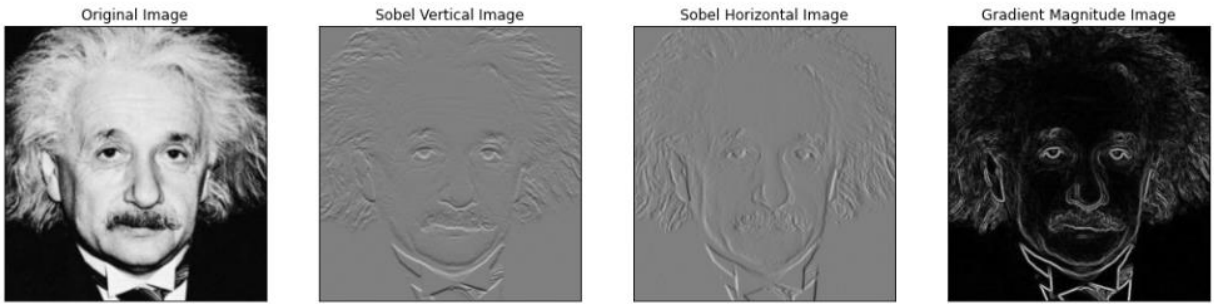




Question 6 – Sobel Filtering

Using the *filter2D* function

```
# Sobel Vertical Kernel
sobel_ver = np.array([[-1,-2,-1],[0,0,0],[1,2,1]], dtype=np.float32)
img_x = cv.filter2D(img6, -1, sobel_ver)

# Sobel Horizontal Kernel
sobel_hor = np.array([[-1,0,1],[-2,0,2],[-1,0,1]], dtype=np.float32)
img_y = cv.filter2D(img6, -1, sobel_hor)

# Gradient Magnitude Kernel
img_grad = np.sqrt(img_x**2 + img_y**2)
```



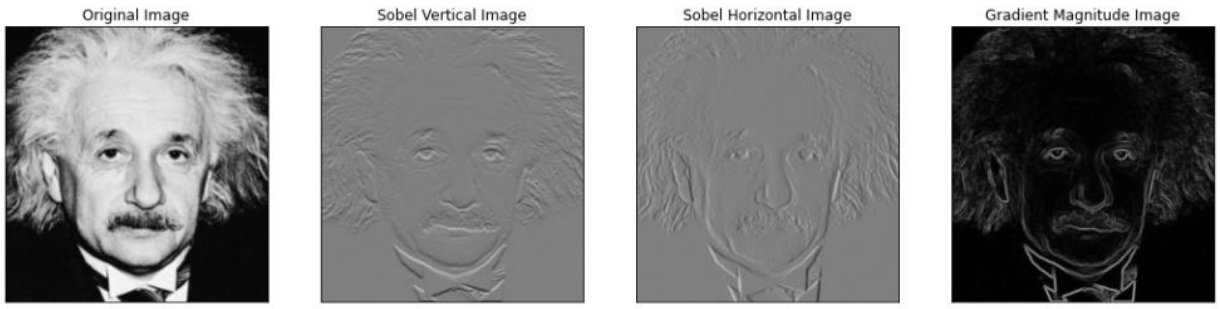
Using own function

Zero padding was added to the image before applying the filter.

```
rows, columns = img6_m.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] = img6_m
```

Filter was applied convolutionally within a loop.

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

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



Similar results were obtained when the *filter2D* function and the own function were used.

Using Associative Property

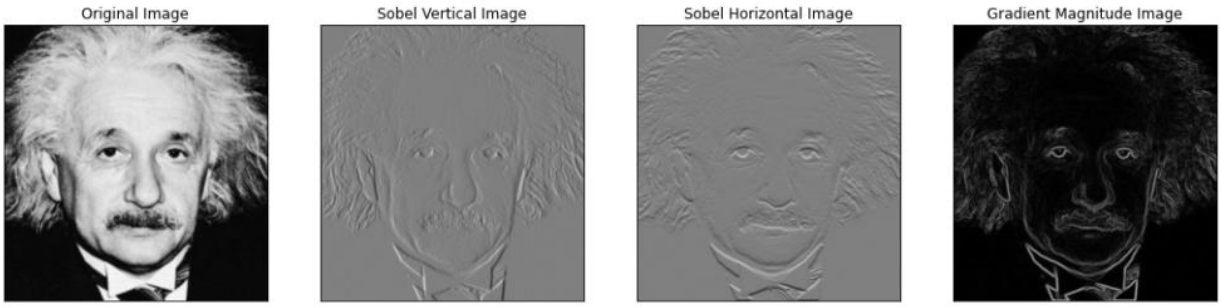
The Sobel vertical and Sobel horizontal kernels could be represented as following.

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \quad and \quad \begin{bmatrix} 1 & 2 & -1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ -0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

To generate the gradient image using this property, first the original images were convolved with the 3x1 array of the above compositions. Then, the intermediate image was again convolved with 1x3 array.

```
# Sobel Vertical Kernel
sobel_ver_array1 = np.array([[1],[2],[1]])
sobel_ver_array2 = np.array([[1,0,-1]])

# Sobel Horizontal Kernel
sobel_hor_array1 = np.array([[1],[0],[-1]])
sobel_hor_array2 = np.array([[1,2,1]])

img_xp_1 = sig.convolve2d(img6_p, sobel_ver_array1, mode="same")
img_xp = sig.convolve2d(img_xp_1, sobel_ver_array2, mode="same")
img_yp_1 = sig.convolve2d(img6_p, sobel_hor_array1, mode="same")
img_yp = sig.convolve2d(img_yp_1, sobel_hor_array2, mode="same")

# Gradient Magnitude Kernel
img_grad_p = np.sqrt(img_xp**2 + img_yp**2)
```

The same results could be obtained from this method as well.

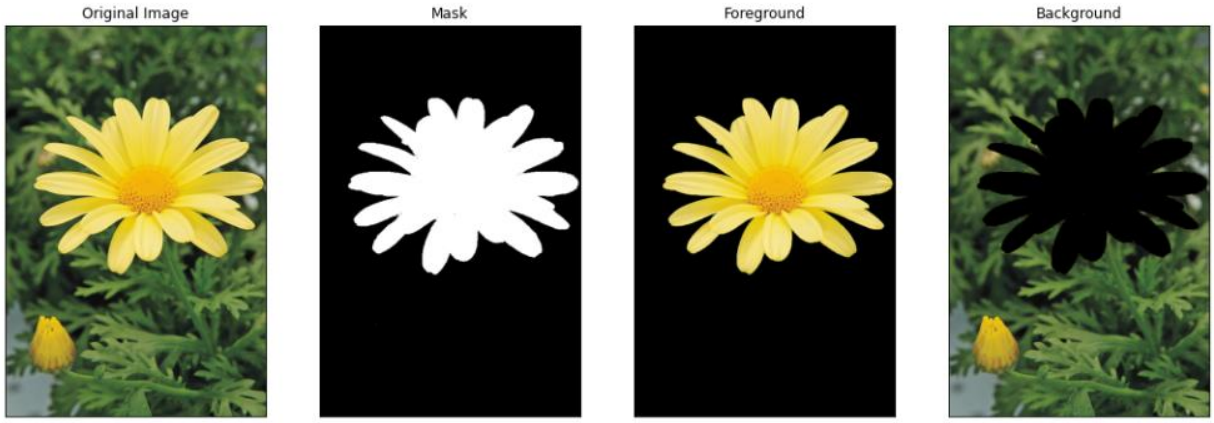


Question 7 – Segmentation

The *grabCut* function was used to segment the flower out of the given image.
First, the rectangle that completely contains the flower is defined. Then the *grabCut* function is applied and the mask is obtained where it estimates the foreground and the background.
Then a mask is obtained for the foreground and the background based on the defined mask and the mask estimated by the *grabCut* function.

Foreground and background images are extracted by applying bitwise *AND* operation between the masks and the original images.

```
mask = np.zeros(img7.shape[:2], dtype="uint8") # Mask - an empty image to store the mask
rect = (40, 150, 560, 500) # Defining the rectangle that contains the object

fgModel = np.zeros((1, 65), dtype="float")
bgModel = np.zeros((1, 65), dtype="float")

# Apply GrabCut using the the bounding box segmentation method
(mask, bgModel, fgModel) = cv.grabCut(img7, mask, rect, bgModel, fgModel, 5, mode=cv.GC_INIT_WITH_RECT)

# Mask for the foreground
fore_mask = (mask == cv.GC_PR_FGD).astype("uint8") * 255
outputMask_fore = np.where((mask == cv.GC_BGD) | (mask == cv.GC_PR_BGD), 0, 1)
outputMask_fore = (outputMask_fore * 255).astype("uint8")
foreground = cv.bitwise_and(img7, img7, mask=outputMask_fore) # Foreground Image

# Mask for the background
back_mask = (mask == cv.GC_PR_BGD).astype("uint8") * 255
outputMask_back = np.where((mask == cv.GC_FGD) | (mask == cv.GC_PR_FGD), 0, 1)
outputMask_back = (outputMask_back * 255).astype("uint8")
background = cv.bitwise_and(img7, img7, mask=outputMask_back) # Background Image
```



An enhanced image is obtained by applying *GaussianBlur* to the background image and adding it with the foreground image.

```
enhanced = np.clip(np.add(foreground, cv.GaussianBlur(background, (9,9), 4)), 0, 255)
```

When the Gaussian blur is applied to the background image, the edges of the black portion that was occupied by the flower gets blurred too. As a convolutional filter is applied, those pixels values might get increases above 0 in those edges. Then, when the foreground image is added to this blurred background, the pixel values around the edge will be increased than the yellow color in the foreground image. Therefore, they appear darker.

GitHub Link: https://github.com/chathuni1999/EN2550.git