DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION ENGINEERING

UNIVERSITY OF MORATUWA

EN3160 Image Processing and Machine Vision

Report

Intensity Transformations and Neighborhood Filtering

Rathnasekara T.S. 200529H

GitHub Link : *Click here to visit.*

August 30, 2023

Question 01

```
#trasformer
c = np.array([(50,50) ,(50,100), (150,255), (150,150), (255,255) ] )
t1 = np.linspace(0, c[0,1], c[0,0] + 1).astype('uint8')
t2 = np.linspace(c[1,1], c[2,1], c[1,1] ).astype('uint8')
t3 = np.linspace(c[3,1], c[4,1], 105).astype('uint8')
t = np.concatenate((t1,t2,t3), axis = 0)
```

Figure 1 : transformer code

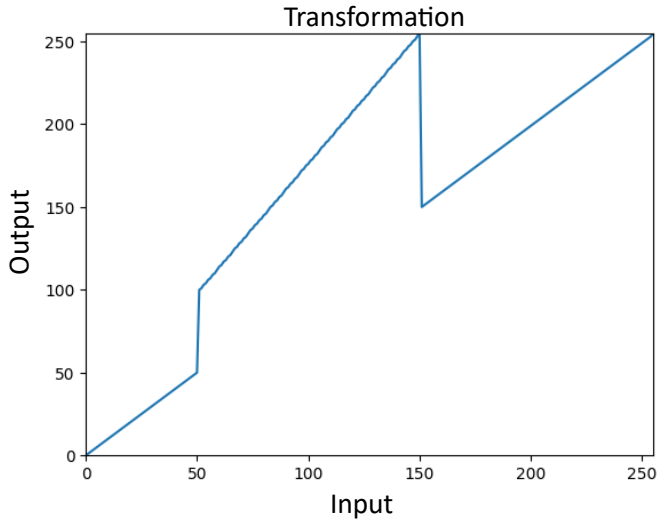


Figure 2: Plot of the transformer

According to the transformation, pixels with mid intensity level are increased suddenly with respect to the darker pixels and more bright pixels. That is why, we can intensity of some part of the image has been increased highly than original image.

Figure 3: Output Images

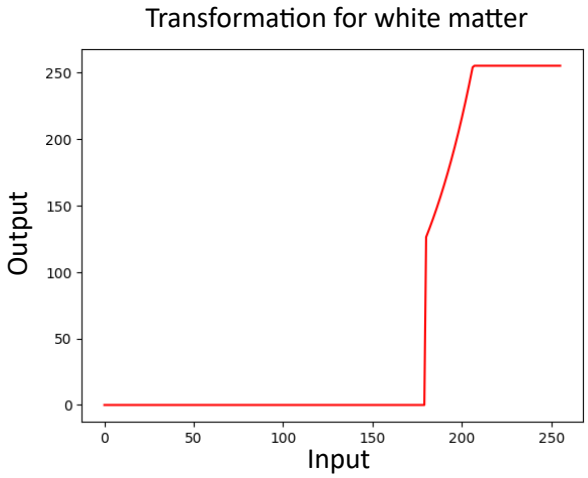
Question 02

(a) White Matter

```
t_2 = np.zeros(256)

t_2[180:] = np.clip((6**(0.015*list_1[180:])),a_min=None , a_max=255)

img_2_transformed= cv.LUT(img_2, t_2.astype('uint8'))
```

Figure 5: Code for intensity transformation of white matter

Figure 4: Transformation plot



In this brain section in order to emphasize the white matter region, pixels with high intensity level should be improved highly and low-level pixels should be surpressed to the lowest level of intensity. In order to do that transformation, above transformer was used.

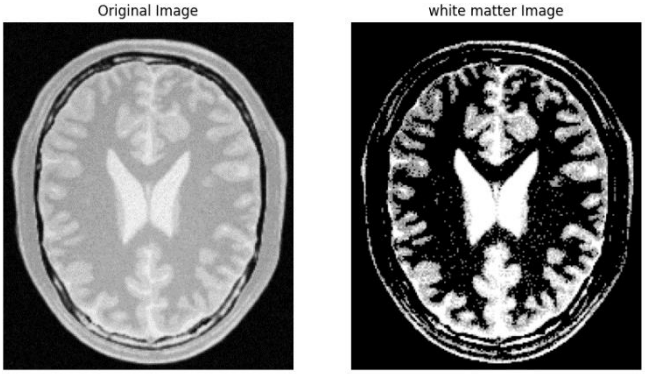


Figure 6: Original image vs Improved image

(b) Gray Matter

```
# tranformer 2
mean = 150
std_dev = 25
x = np.linspace(0, 255, 256)
y = (1 / (std_dev * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mean) / std_dev)**2)
t_3 = (y - np.min(y)) * 255 / (np.max(y) - np.min(y))
```

Figure 7: Code for intensity Transformation of gray matter

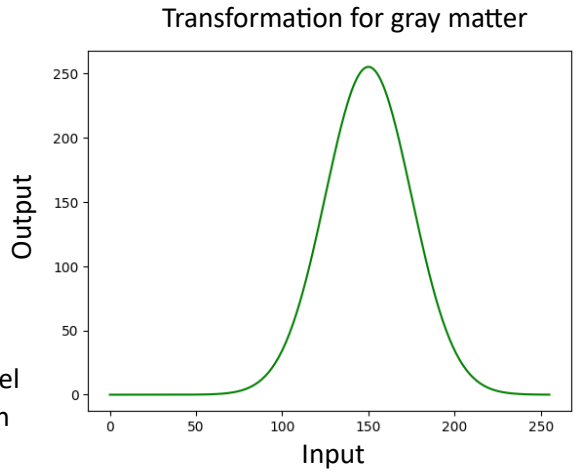


Figure 8: Transformation graph of gray matter

Here, gray matter has intensity level which is middle region of the intensity plot. So, in order to emphasize that, pixels with middle intensity level, should be improved to high intensity level and other pixels should be suppressed to '0' intensity level as in transformation graph.

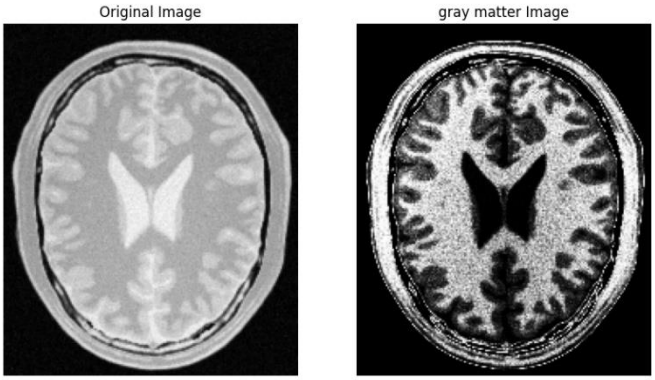


Figure 9: Original Vs Enhanced

Question 03

```
gamma = 1.5
table = np.array([((i / 255.0) ** (1.0 / gamma)) * 255 for i in np.arange(0, 256)]).astype("uint8")
L_corrected = cv.LUT(L, table)

img_3_gamma_corrected = cv.merge([L_corrected, cv.split(img_3_1)[1], cv.split(img_3_1)[2]])
```

Figure 10: code for gamma correction

According to the observation $\gamma = 1.5$ is the good gamma correction for the given image.

This will give high boost to the high-level intensity pixels and lower boost to the pixels with low intensity level.

Figure 11: original Vs gamma corrected.

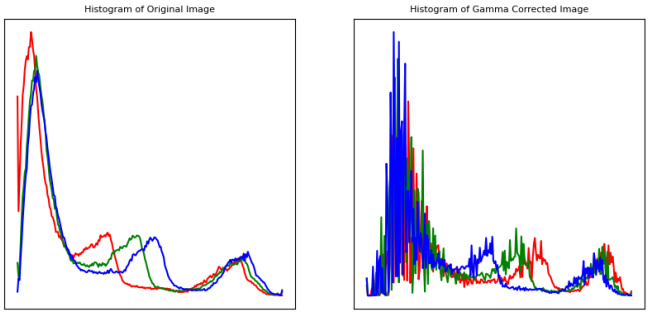


Figure 12: histograms of original image and gamma corrected image

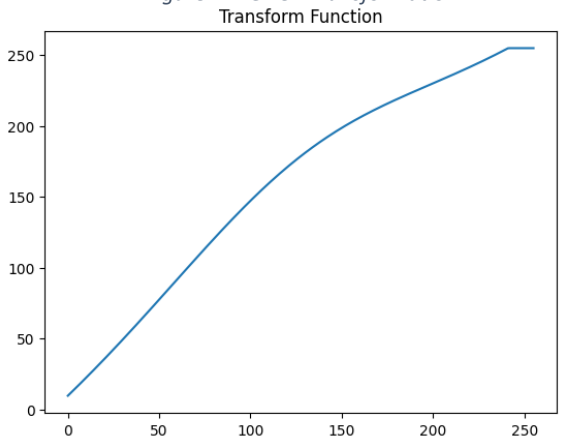
Question 04

```
# Transformation function

def transform4(arr, a, zegma):
    transformed_arr = []
    for i in arr:
        tr_list = []
        for x in i:
            t = min(x + (a * 128) * np.exp((-1 * (x - 128) ** 2) / (2 * zegma ** 2)), 255)
            tr_list.append(t)
        #converted to uint8
        transformed_arr.append(np.array(tr_list))
    transformed_arr = np.array(transformed_arr).astype(np.uint8)
    return transformed_arr

# Transform the image
saturation_transformed = transform4(saturation, 0.4, 70)
```

Figure 13: Transformation Function

After observing several values for a, 0.4 is selected as the suitable value. By adding above transformation to saturation plane, the intensity of the of image has been increased with respect to original image by giving transformation as follows.

Figure 14: Given Transformation



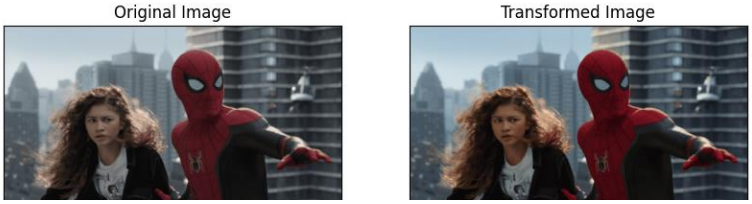


Figure 15: Original Image Vs Enhance Image

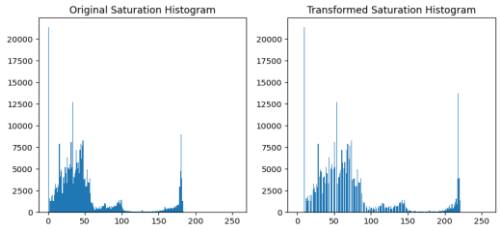


Figure 16: Variation of histograms

Question 05

```python
#histogram equalization
def hist_eq(img):
    hist, bins = np.histogram(img.flatten(), 256, [0, 256])
    cdf = hist.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).astype('uint8')
    equalized_img = cdf[img]
    return equalized_img
```

Figure 17: Code of histogram equalizer

Here, I wrote a custom histogram equalizer function using following equation.

$$\frac{L-1}{MxN}\sum_{j=0}^{k} n_j \quad \text{where } k = 1,2,3,\ldots\ldots,(L-1)$$

***L** -> number of intensity levels; **M, N** -> height and width of the image;*

Cumulative sum of histogram -> $\sum_{j=0}^{k} \boldsymbol{n_j}$

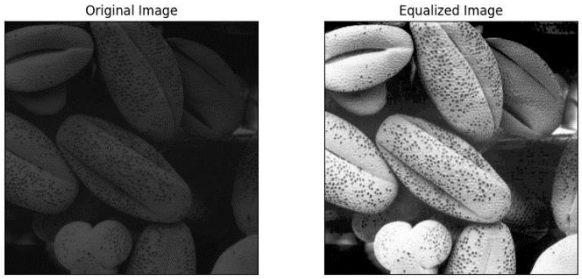


Figure18: Original image Vs Histogram equalized image

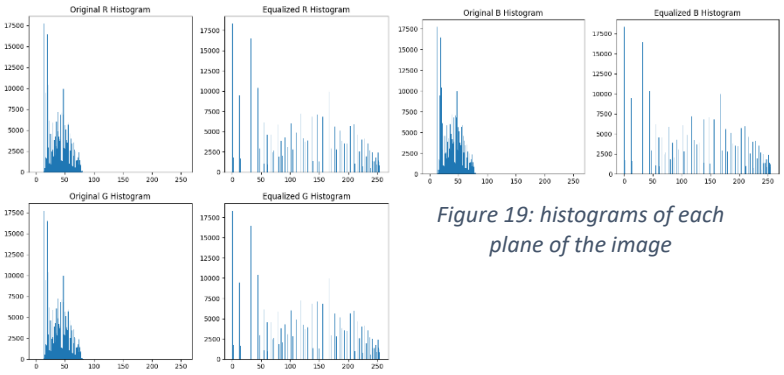


Figure 19: histograms of each plane of the image

Question 06

```python
threshold = 13
binary_mask = np.zeros_like(saturation_6)
binary_mask[saturation_6 > threshold] = 255
```

Figure 20: binary mask creation

```python
fg_h,fg_s,fg_v = cv.split(img_foreground_hsv)

fg_s_eq = hist_eq(fg_s)

equalized_hist_foreground = cv.merge([fg_h, fg_s_eq, fg_v])
```

Figure 22: histogram equalization of saturation plane of foreground

```python
binary_mask_reshape = cv.cvtColor(binary_mask.astype(np.uint8), cv.COLOR_GRAY2BGR)
img_foreground = cv.bitwise_and(img_6,binary_mask_reshape)
```

Figure 31: bitwise_and between foreground and binary mask

```python
# background mask
background_mask = cv.bitwise_not(binary_mask_reshape)
# extract background pixels
background_pixels = cv.bitwise_and(img_6, background_mask)
background_pixels = cv.cvtColor(background_pixels, cv.COLOR_BGR2RGB)

#obtain final image
final_image = cv.add(background_pixels, equalized_hist_foreground_cvt)
```

Figure 23: extracting background and add to equalized foreground.

First, binary mask was created by setting threshold to saturation plane of the image. Using extracted mask, foreground of the image was extracted and send it through the histogram equalization process in order to get vibrant foreground for the image. Next, background has been extracted using inverse of the binary mask and it was added to equalized foreground to get an image with enhanced foreground.

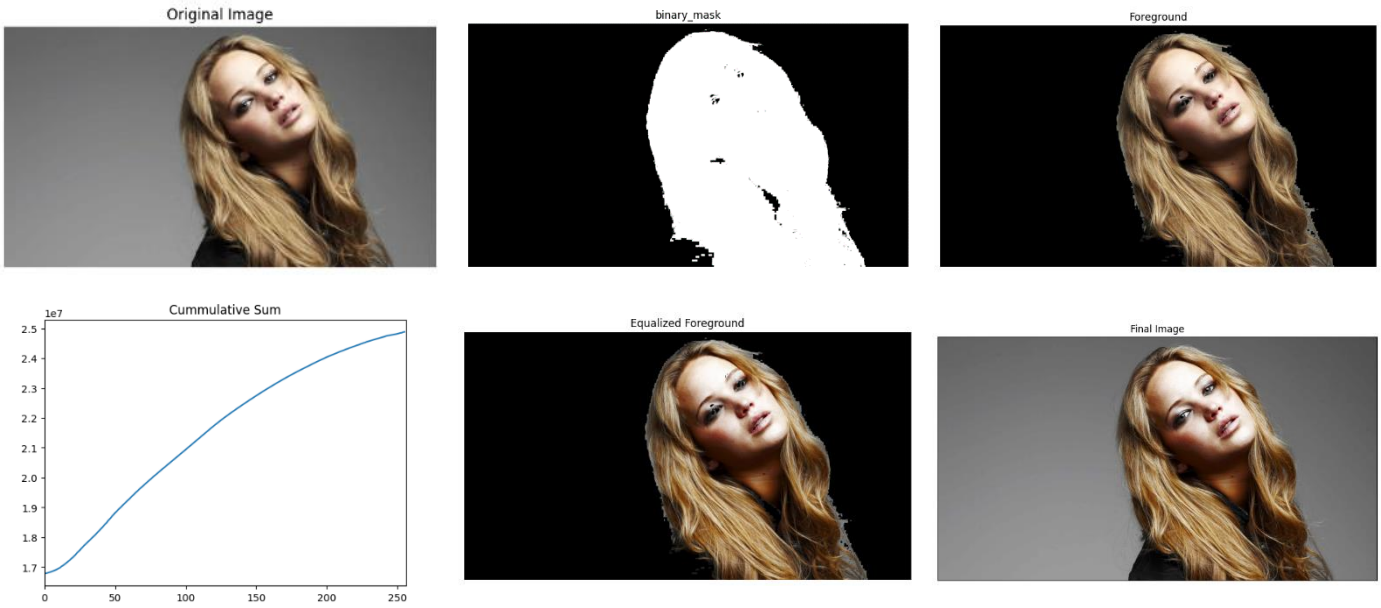


Figure 24: Process of the foreground equalization

Question 07

```
kernel = np.array([(-1,0,1),(-2,0,2),(-1,0,1)], dtype= 'float')
img_7_filtered = cv.filter2D(img_7_1, -1, kernel)
```

Figure 25: using cv.filter2D

Here, "cv.filter2D" function is used for the convolution and it gives better filter than other three methods. Because padding and other techniques are used for filtering.

```
def sobel_filtering(image):
    kernel = np.array([[-1,0,1],[-2,0,2],[-1,0,1]],dtype= 'float')
    assert kernel.shape[0] % 2 == 1 and kernel.shape[1] % 2 == 1
    k_hh, k_hw = math.floor(kernel.shape[0] / 2), math.floor(kernel.shape[1] / 2)
    h, w = image.shape
    image_float = cv.normalize(image.astype('float'), None, 0.0, 1.0, cv.NORM_MINMAX)
    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*3).astype('uint8')

#filtering
for i in range(3):
    img_7_filtered_2[:,:,i] = sobel_filtering(img_7_copy_1[:,:,i])
```

Figure 26: using custom build function.

By using custom build function, we can convolve the matrices as in code in order to filtered image.

```
# sobel_filter
sobel_kernel_c = np.array([(1),(2),(1)], dtype= 'float')
sobel_kernel_r = np.array([(1,0,-1)], dtype= 'float')
#filtering using property
img_7_filtered_3 = cv.filter2D(img_7_copy_2,-1,sobel_kernel_r)
img_7_filtered_3 = cv.filter2D(img_7_filtered_3,-1,sobel_kernel_c)
```

Figure 27: using property.

In this case, we can enhance the computational efficiency by using this property of the matrices.

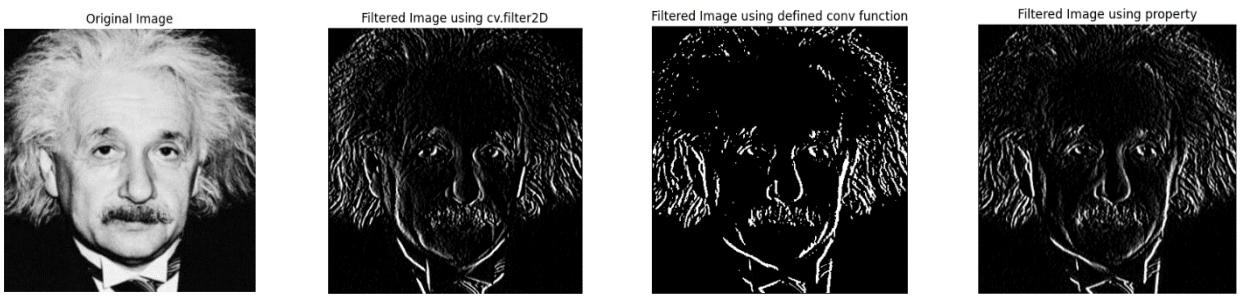


Figure 28: output images in each case

Question 08

(a) Nearest Neighbor Method

Figure 29: Nearest Neighbor Method

```python
# define zoom function using nearest neighbor method
def zoomNN(image,Scale_factor):
    rows, cols = image.shape[0]*Scale_factor , image.shape[1]*Scale_factor
    zoomed = np.zeros((rows,cols,3))
    for i in range(rows):
        for j in range(cols):
            zoomed[i][j] = image[i//Scale_factor][j//Scale_factor]
    zoomed = zoomed.astype('uint8')
    return zoomed
```

In this method, additional rows and columns are replaced using neighbor rows and columns. Width and height are determined by using scaling factor. So, NSSD values calculated for given images, are in following section.

*NSSD of zoomed image 1 (using nearest neighbor method) is **29376.036127186213.***

*NSSD of zoomed image 4 (using nearest neighbor method) is **26317.328341933513.***

Figure 10: test cases

(b) Bilinear Interpolation

```python
# define bilinaer Interpolation function
def zoomBI(image,Scale_factor):
    #scale the image
    rows, cols = image.shape[0]*Scale_factor , image.shape[1]*Scale_factor
    zoomed_image = np.zeros((rows,cols,3))

    for i in range(rows):
        for j in range(cols):
            x = i/Scale_factor
            y = j/Scale_factor
            x1 = math.floor(x)
            y1 = math.floor(y)
            x2 = min(math.ceil(x), len(image) - 1)
            y2 = min(math.ceil(y), len(image[0]) - 1)

            #calculate the value of the pixel using bilinear interpolation
            zoomed_image[i,j] = (
                image[x1,y1]*(x-x1)*(y-y1) +
                image[x1,y2]*(x-x1)*(1-y+y1) +
                image[x2,y1]*(1-x+x1)*(y-y1) +
                image[x2,y2]*(1-x+x1)*(1-y+y1)
                )

    return zoomed_image.astype('uint8')
```

Here, we use interpolation to predict the values of pixels which are added after scaling the image. In this method, we measure the distance between current pixel and the pixels which are in original images and according to that we calculate weights. Using those weight, we calculate the intensity of newly added pixels. According to the observations of NSSD, this technique is much accurate than nearest neighbor method.

Figure 11: Bilinear Interpolation

*NSSD of zoomed image 1 (using bilinear interpolation) is **27515.828999164096.***

*NSSD of zoomed image 4 (using bilinear interpolation) is **25483.006794745048.***

Figure 12: test cases for bilinear interpolation

```python
# compute normalized sum of squares difference of image
def NSSD(orig_img, zoomed_img):
    rows,cols,channel = orig_img.shape
    NSSD_value = np.int64(0)
    for i in range(rows):
        for j in range(cols):
            for k in range(channel):
                NSSD_value +=(orig_img[i,j,k] - zoomed_img[i,j,k])**2
    return NSSD_value/(rows*cols*channel)
```

This is the function for calculating **normalized sum of square differences.**

Figure 13: Normalized sum of square differences

Question 09

```
mask = np.zeros(img_9_cvt.shape[:2],np.uint8)

background = np.zeros((1,65),np.float64)
foreground = np.zeros((1,65),np.float64)

rect = (50,100,500,500)

# use grabcut function
cv.grabCut(img_9_cvt,mask,rect,background,foreground,3,cv.GC_INIT_WITH_RECT)

mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')
foreground= img_9_cvt*mask2[:,:,np.newaxis]
background = np.subtract(img_9_cvt,foreground)
```

Figure 34: code for removing flower by using grabCut

Here, we use mask to extract the interested foreground and rectangular square mentioned in code, is used to select the required area. According to these

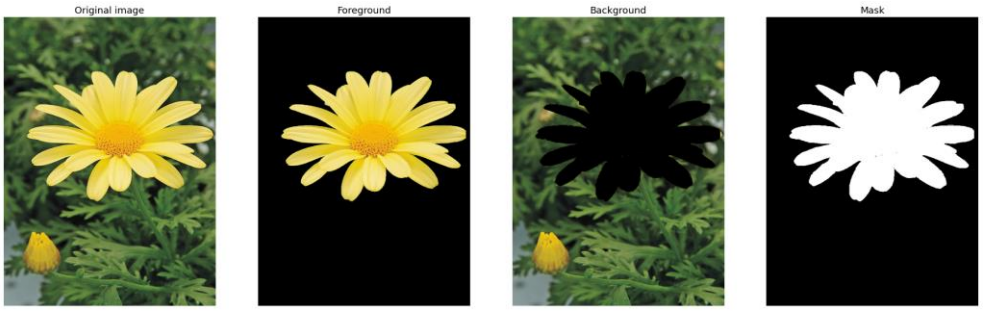


Figure 35: Output results

```
sigma = 4
Background_blurr = cv.GaussianBlur(background,(7,7),sigma)

img_9_blurr= np.add(foreground,Background_blurr)
```

Figure 36: code for blurring and generating final result.

"GaussianBlur" function is used to give intended blur for the background. Finally added that background to foreground and final image is generated.

Figure 37: Original image Vs Blurred image

(c) This happens due to imperfect object - background separation of the segmentation algorithm. Pixels near to boundary will be misclassed as background parts and go through blurring process. As a result of that, those erroneous pixels will blend with foreground and background pixels and create quit dark edges.