EN3150 Assignment 1 on Intensity Transformation and Neighborhood Filtering

200041E - Anuradha A.K

Question 1

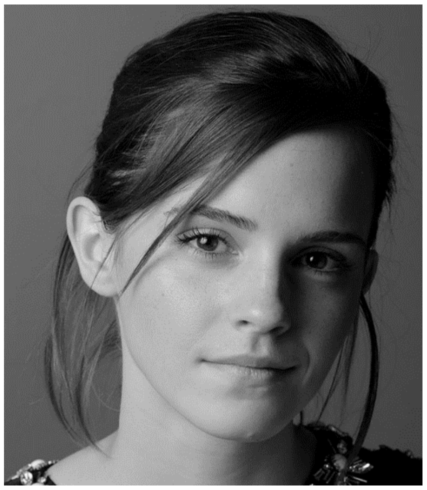
In this question, the image is read in grayscale and applied the specific intensity transformation function on it.

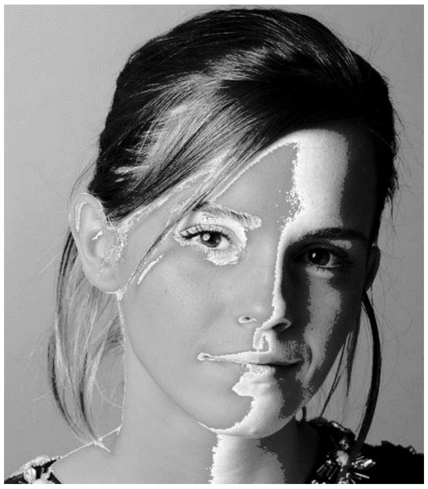
```
c = np.array([(50, 50), (50, 100), (150, 255), (150, 150)])

t1 = np.linspace(0, c[0, 1], c[0, 0] + 1 - 0).astype('uint8')
t2 = np.linspace(c[0, 1] + 1, c[1, 1], 0).astype('uint8')
t3 = np.linspace(c[1, 1] + 1, c[2, 1], c[2, 0]-c[1, 0]).astype('uint8')
t4 = np.linspace(c[2, 1] + 1, c[3, 1], 0).astype('uint8')
t5 = np.linspace(c[3, 1] + 1, 255, 255-c[3, 0]).astype('uint8')

transform = np.concatenate((t1, t2), axis=0).astype('uint8')
transform = np.concatenate((transform, t3), axis=0).astype('uint8')
transform = np.concatenate((transform, t4), axis=0).astype('uint8')
transform = np.concatenate((transform, t5), axis=0).astype('uint8')
```

Here is the output of the intensity transformation,


Original Image

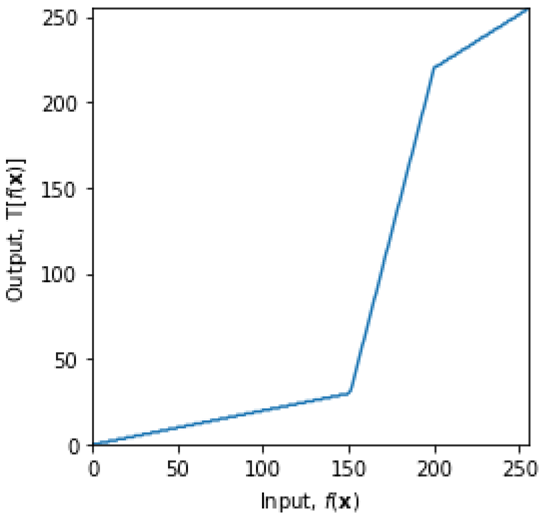

Transformed Image

Question 2

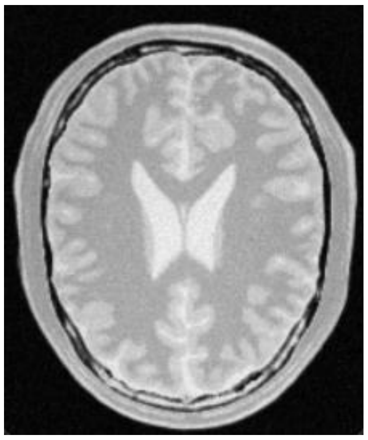
In the second question, the code needs to accentuate the white matter and gray matter on the image. To do that I used an intensity transformation function that has a lower gradient on white and gray matter.

```
c = np.array([(150, 30), (200, 220)])
t1 = np.linspace(0, c[0, 1], c[0, 0] + 1 - 0).astype('uint8')
t2 = np.linspace(c[0, 1] + 1, c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.linspace(c[1, 1] + 1, 255, 255 - c[1, 0]).astype('uint8')
```

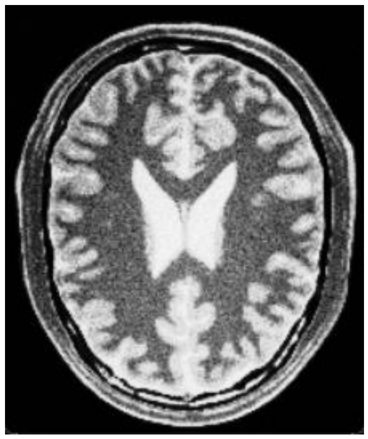
Here is the function I used and the output of the brain slice image,



Transformation Function



Original

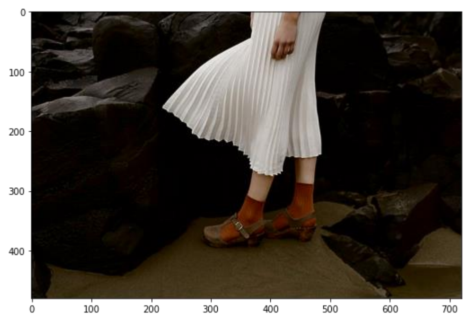


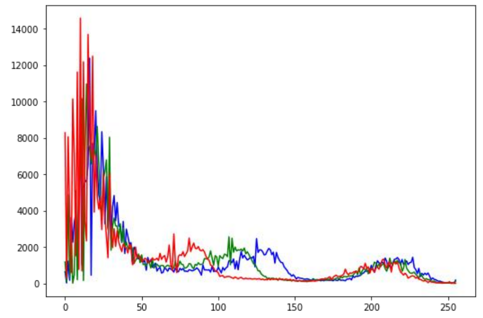
Edited

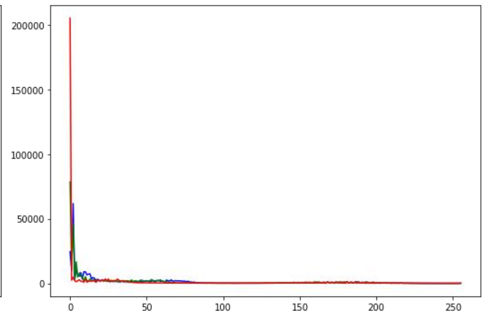
Question 3

I imported the image and made the gamma function array getting gamma as 2 for the first step of the question. Then I split the image planes into L, a, and b planes and applied the gamma function to the L plane since the question says to apply the gamma correction to the L plane only. After applying the correction merged the planes and converted them to R, G, and B planes and showed the image using matplotlib. Histograms are calculated by ***calcHist*** function on the OpenCV library.

```
gamma = 2
table = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0,256)]).astype('uint8')
img_orig = cv.cvtColor(img_orig, cv.COLOR_BGR2LAB)
L, a, b = cv.split(img_orig)
gamma_L = cv.LUT(L, table)
img_gamma = cv.merge((gamma_L, a, b))
img_gamma = cv.cvtColor(img_gamma, cv.COLOR_LAB2RGB)
img_orig = cv.cvtColor(img_orig, cv.COLOR_LAB2RGB)
```





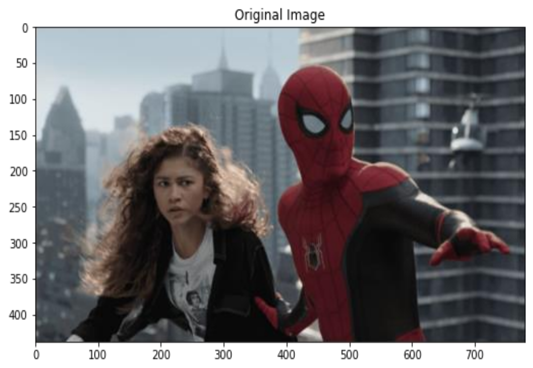


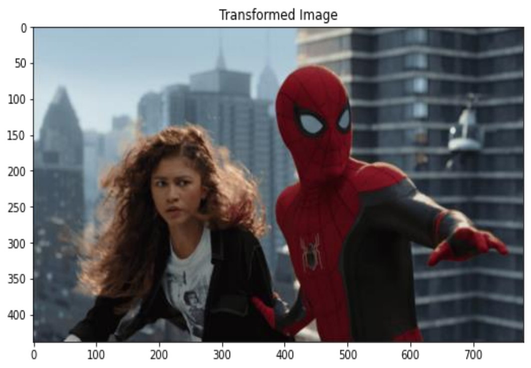
Question 4

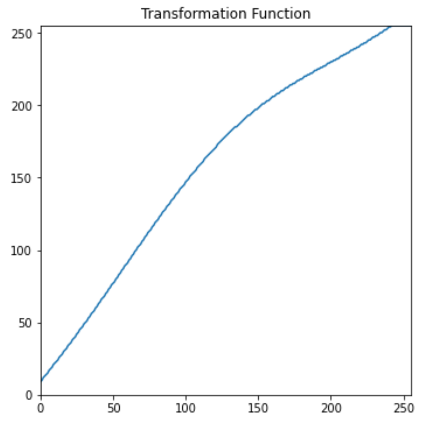
In the first step, I made an array for the given transformation function. Like in the previous question then split the image into H, S, and V instead of L, a, and b to apply the function to the saturation plane. finally merged the edited planes and showed them using matplotlib.

In the transformation function, adjusted ***a*** to 0.4 to get a visually pleasant output.

```
a = 0.4
formula = np.array([min(i+(a*128)*exp(-pow((i-128),2)/(2*pow(70,2))), 255) for i in
np.arange(0,256)]).astype('uint8')
```







Question 5

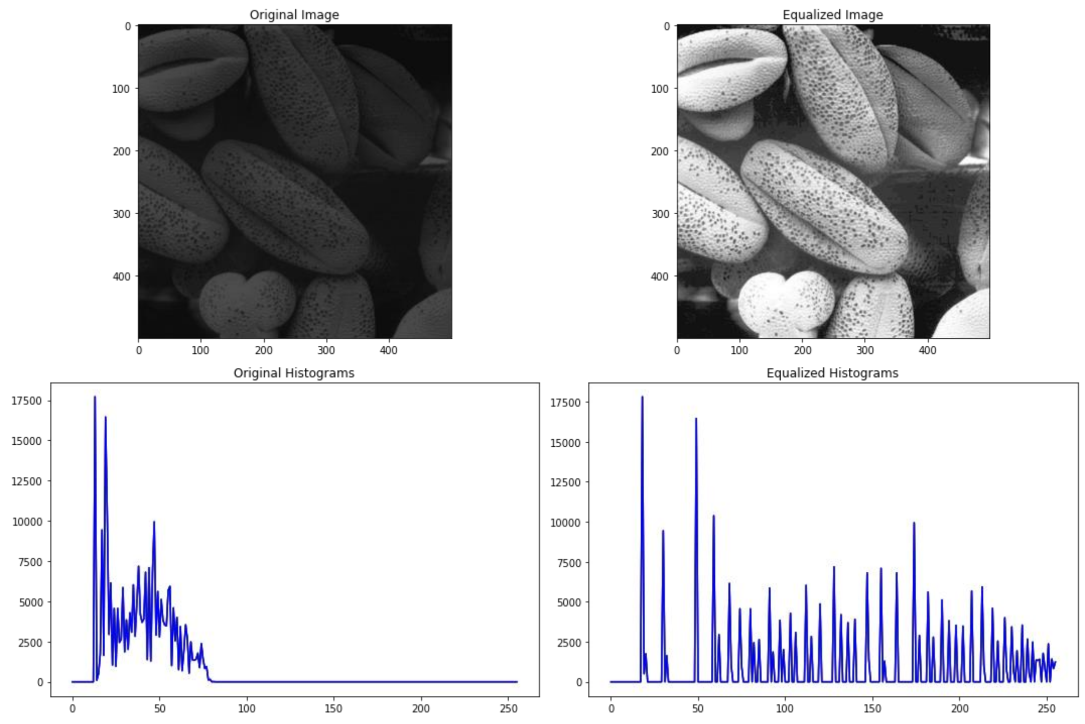
In question 5, I write a function to equalize a given image by getting the file path as the parameter of the function. In the first step of this task, I imported the image by color scale and then it split into B, G, and R planes. The equalization process is applied for each plane separately and after equalization, planes are merged and converted to uint8 data type for image showing process. The function also shows the histograms of original and equalized images. The following function is used for the equalization process.

$$Equalized\ pixel = \frac{L-1}{MN}\sum_{j=0}^{k} n_j$$

```
def histogram_equalization(filename):
    img_orig = cv.imread(filename, cv.IMREAD_COLOR)
    B, G, R = cv.split(img_orig)
    height, width = img_orig.shape[:2]
    MN = height * width
    L = 256
    color = ('b', 'g', 'r')
    color_palate = np.array([])

    for i, c in enumerate(color):
        hist_orig = cv.calcHist([img_orig], [i], None, [256], [0, 256])
        np.set_printoptions(precision=8, suppress=True)
        hist_cumsum = np.cumsum(hist_orig)
        hist_cumsum = hist_cumsum * ((L-1) / MN)
        rounded_array = np.round(hist_cumsum).astype(int)
        color_palate = np.append(color_palate, rounded_array)
    equalize_B = cv.LUT(B, color_palate[0:256])
    equalize_G = cv.LUT(G, color_palate[256:512])
    equalize_R = cv.LUT(R, color_palate[512:768])

    img_equalize = cv.merge((equalize_B, equalize_G, equalize_R))
    img_equalize = img_equalize.astype(np.uint8)  # Convert to np.uint8
```



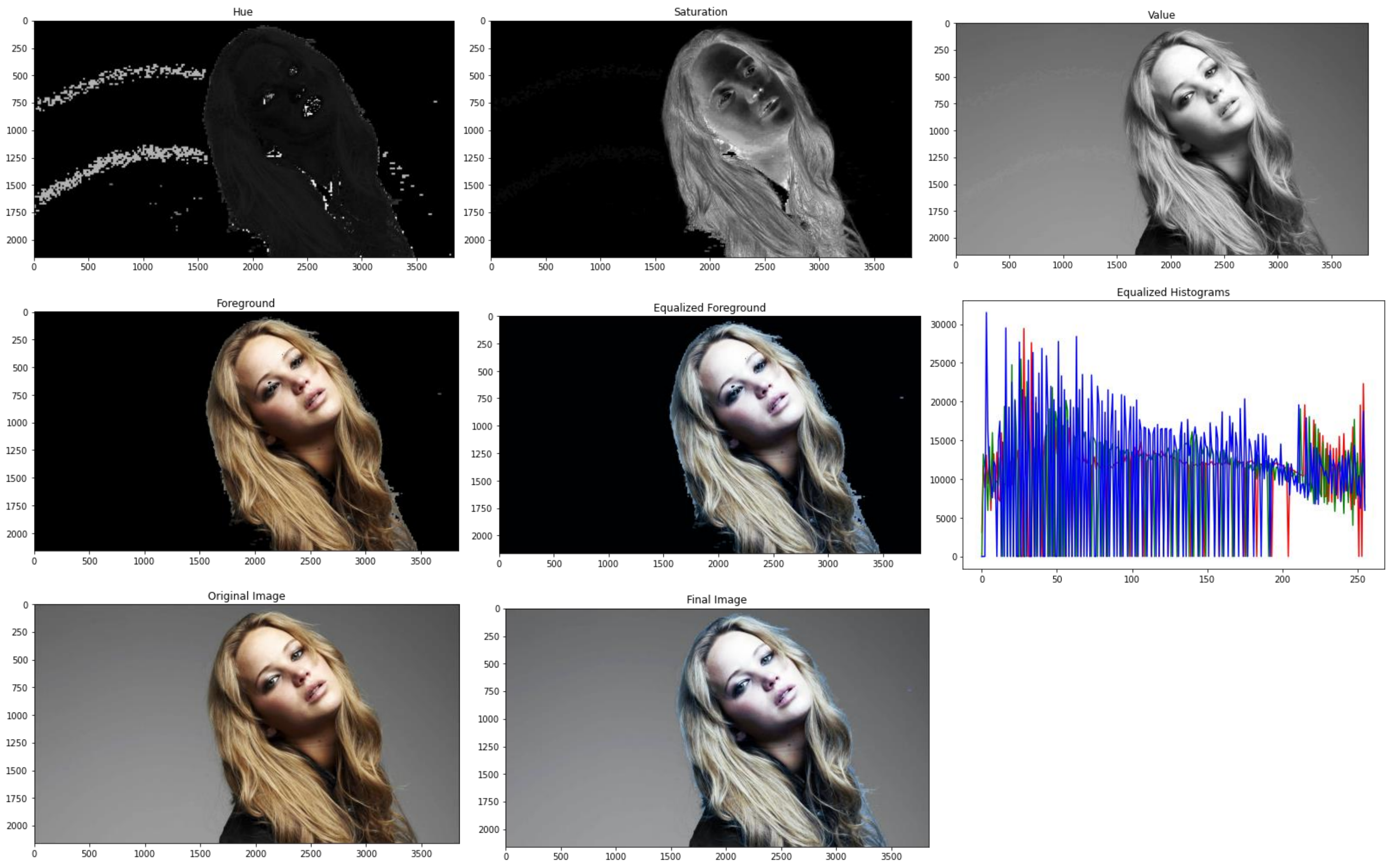
Question 6

In this question separating the background and foreground is the first step. To do this, we need to make a mask to filter out the pixels belonging to the foreground and the background. For making the mask, I split the image into H, S, and V planes and plot each plane in the grayscale. In that stage I could recognize color of the pixels of the saturation plane has a considerable difference in the background and foreground. Therefore, I used the saturation plane to make the mask and got the threshold value of 11. I used the ***biwise_and*** function to extract the image according to the mask and equalize the foreground image like in the previous question. Finally, equalized foreground and background are merged and shown using matplotlib.

```
img_orig = cv.cvtColor(img_orig, cv.COLOR_BGR2HSV)
H, S, V = cv.split(img_orig)
mask = S > 11 # saturation field selected for make the mask
mask = mask.astype(np.uint8) * 255
H_mask = cv.bitwise_and(H, mask)
S_mask = cv.bitwise_and(S, mask)
V_mask = cv.bitwise_and(V, mask)
img_fore = cv.merge((H_mask, S_mask, V_mask))
img_fore = cv.cvtColor(img_fore, cv.COLOR_HSV2RGB)
R_mask, G_mask, B_mask = cv.split(img_fore)

for i, c in enumerate(color):
    hist_fore = cv.calcHist([img_fore], [i], mask, [256], [0, 256])
    hist_cumsum = np.cumsum(hist_fore)
    hist_cumsum = hist_cumsum * (255 / np.sum(hist_fore))
    rounded_array = np.round(hist_cumsum).astype(int)
    color_palate = np.append(color_palate, rounded_array)

equalize_R = cv.LUT(R_mask, color_palate[0:256])
equalize_G = cv.LUT(G_mask, color_palate[256:512])
equalize_B = cv.LUT(B_mask, color_palate[512:768])
img_fore_equalize = cv.merge((equalize_R, equalize_G, equalize_B))
```



Question 7

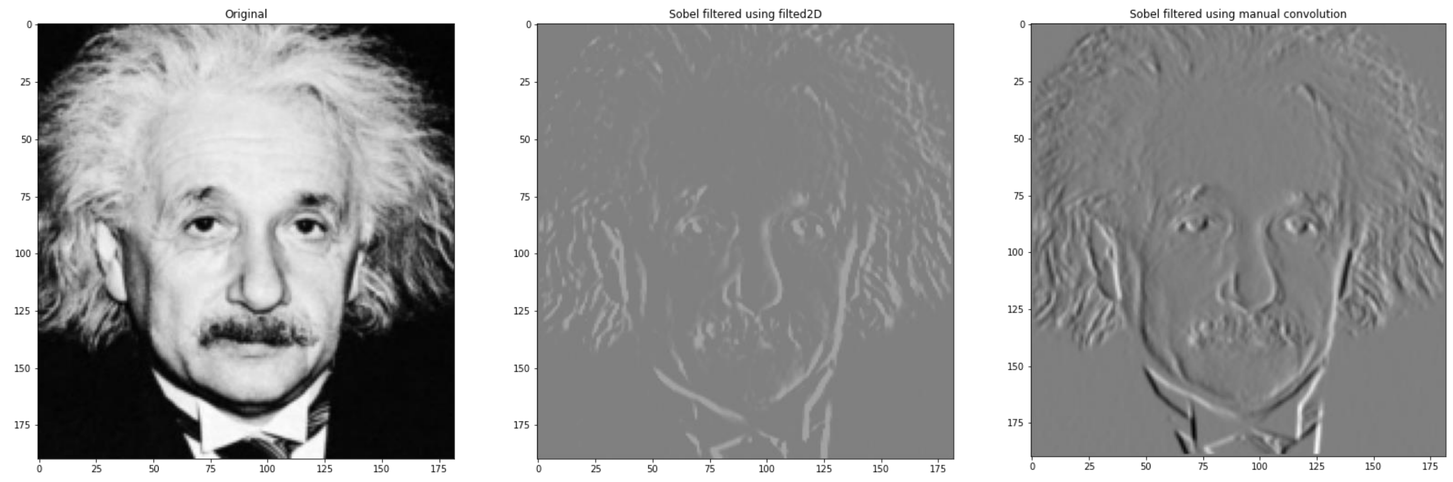
a) Used filter2D function with a given Kernal to take Sobel filtered output.

```
kernel = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]], dtype='float32')

imgc = cv.filter2D(img, -1, kernel)
```

b) Wrote a code for getting the convolution with the Kernal and image by using two ***for*** loops. After the convolution, the output array is normalized to show the Sobel-filtered image.

```
imgc = np.zeros_like(img, dtype=np.float32)
# Perform the convolution operation
for y in range(img_height - kernel_height + 1):
    for x in range(img_width - kernel_width + 1):
        img_slice = img[y:y+kernel_height, x:x+kernel_width]
        convolution_result = np.sum(np.multiply(img_slice, kernel))
        imgc[y+1, x+1] = convolution_result
imgc = (imgc - np.min(imgc)) / (np.max(imgc) - np.min(imgc)) * 255
```


Original
Sobel filtered using filted2D
Sobel filtered using manual convolution
0
25
50
75
100
125
150
175

Question 8

a) In this part I implemented the zooming method by using nearest-neighbor implementation. Using two ***for*** loops to give the same pixel color to the neighbor pixels according to the scalar factor is the implementation method in this zooming.

```
def zoom_nearest_neighbor(image, scale_factor):
    height, width = image.shape[:2]
    new_height = int(height * scale_factor)
    new_width = int(width * scale_factor)
    zoomed_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)

    for y in range(new_height):
        for x in range(new_width):
            original_y = int(y / scale_factor)
            original_x = int(x / scale_factor)
            zoomed_image[y, x] = image[original_y, original_x]

    return zoomed_image
```

b) In part 2, the image zoom implementation method is a bilinear interpolation. Bilinear interpolation is a method of zooming in on an image by estimating the value of new pixels based on the values of the surrounding pixels. It dœs this by assuming that the intensity of a pixel is a linear function of its x and y coordinates. I used the resize function with the parameter INTER_LINEAR to get the bilinear zoom.

```
# Zoom using bilinear interpolation
zoomed_bilinear = cv.resize(image, None, fx=scale_factor, fy=scale_factor,
interpolation=cv.INTER_LINEAR)
```

To quantify the difference between the zoomed image and the original image resulting from two distinct zooming methods, I computed the sum of the squared differences of the Red (R), Green (G), and Blue (B) color channel values at the pixel level.

```
def normalizedSumOfSquare(image1, image2):
    image1 = image1.astype(np.float64)
    image2 = image2.astype(np.float64)
    height, width = image1.shape[:2]
    return np.sum(np.square(image1 - image2))/(height*width*3)
```

here is the output of the code,

```
SSD between original large image and nearest neighbor zoom:  136.26904899691357
SSD between original large image and bilinear interpolation zoom:  115.0919012024177
```

We can conclude that the SSD between the original large image and bilinear interpolation zoom is lesser than The SSD value of the original large image and nearest neighbor zoom. So, the bilinear interpolation is better for zooming.

Question 9

In question 9, I used the ***grabCut*** function in the OpenCV library to get the foreground image mask and background image mask. In the first step, I created a zero array that matches the mask and then modified it using grabCut by giving a specific area that includes the flower.

```
mask = np.zeros(image.shape[:2], np.uint8)
rect = (0, 100, 560, 470) # x, y, width, height
cv.grabCut(image, mask, rect, None, None, 5, cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask==2)|(mask==0), 0, 1).astype('uint8') # mask for foreground
mask3 = np.where((mask==2)|(mask==0), 1, 0).astype('uint8') # mask for background
```

Then I did some enhancements to the image to get a visually pleasing output. I Added Gaussian blur for the background, decreased the brightness of the background, gamma corrected on the green plane of the foreground, and gamma corrected on the saturation plane of the background to get enhanced image.

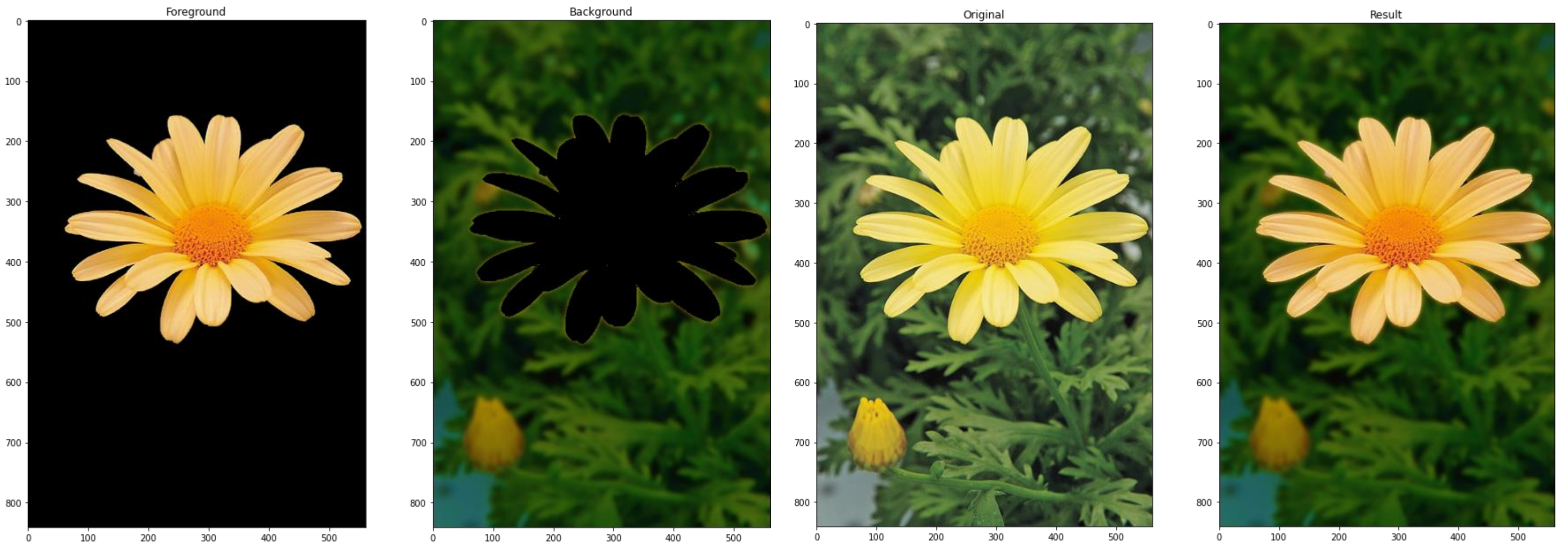
```
image_fore = image*mask2[:,:,np.newaxis]

# blur the background
blurred_background = cv.GaussianBlur(image, (35, 35), 0)
image_back = blurred_background*mask3[:,:,np.newaxis]
```

```python
# decrease the brightness of the background
brightness_factor = 0.7
image_back = np.clip(image_back * brightness_factor, 0, 255).astype(np.uint8)

# Gamma correction in green plane on foreground image
gamma_fore = 1.8
table = np.array([(i/255.0)**(gamma_fore)*255.0 for i in
np.arange(0,256)]).astype('uint8')
B, G, R = cv.split(image_fore)
gamma_G = cv.LUT(G, table)
image_fore = cv.merge((B, gamma_G, R))

# Gamma correction in saturation plane on background image
gamma_back = 0.2
table = np.array([(i/255.0)**(gamma_back)*255.0 for i in
np.arange(0,256)]).astype('uint8')
image_back = cv.cvtColor(image_back, cv.COLOR_BGR2HSV)
H, S, V = cv.split(image_back)
gamma_S = cv.LUT(S, table)
image_back = cv.merge((H, gamma_S, V))
```



After getting the enhanced image I can see there is quite dark just beyond the edge of the flower. That occurs because of the ***grabCut*** function. The ***grabCut***, while a powerful segmentation algorithm, may not always produce a perfect mask. Small errors or inaccuracies in the mask can result in the blur being applied partially to the foreground or not reaching the entire background. This can create a transition zone with partial blur, making it appear darker.

GitHub Repository Link – https://github.com/askanuradha/EN3160-Image-Processing-and-Machine-vision.git