DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING
UNIVERSITY OF MORATUWA

EN2550 FUNDAMENTALS OF IMAGE PROCESSING AND MACHINE VISION

ASSIGNMENT 02 - Fitting and Alignment

Thanushan K. 190621M

This is submitted as a partial fulfilment for the module EN2550.

April 24, 2022

1 RANSAC Implementation

The given transformation is a piecewise intensity transformation that intensifies the input pixel values in the range from 50 to 150 while keeping the remaining output pixel values same as the input pixel values. The result is shown in the following figure.

```python
def RANSAC_fit(X,n):
    t = 1
    N = np.inf
    Bestfit = None
    Inliers = []
    sample_points =[]
    p = 0.99 #probability p, at least one random sample is free from outliers
    iterations = 0
    min_s = 3 #Minimum no of points required to find the equation of the circle
    e = 0.5 #Outlier ratio 0.5 was taken for the worst case
    N = np.log(1-p)/np.log(1-(1-e)**min_s)  #calculation of samples
    while N > iterations:
        random_indices= np.random.randint(n, size=min_s)
        point1, point2, point3 = X[random_indices]

        #Calculation of the center coordinates and the radius of the circle passing through the sample points
        coefficientMatrix = np.array([[point2[0] - point1[0], point2[1] - point1[1]], [point3[0] - point1[0], point3[1] - point1[1]]])
        constantMatrix = np.array([[point2[0]**2 - point1[0]**2 + point2[1]**2 - point1[1]**2], [point3[0]**2 - point1[0]**2 + point3[1]**2 - point1[1]**2]])
        invCoefficientMatrix = np.linalg.pinv(coefficientMatrix)

        center_x, center_y = (invCoefficientMatrix@constantMatrix) / 2
        center_x, center_y = center_x[0], center_y[0]
        r = np.sqrt((point1[0]- center_x)**2 + (point1[1] - center_y)**2)

        Inlier_test = []
        #Checking for inliers and appending them into the inlier set.
        for x, y in X:
            dis = np.sqrt((x - center_x)**2 + (y - center_y)**2)
            if (np.abs(dis - r) < t):
                Inlier_test.append([x,y])

        #Checking whether the number of current inliers is greater than the past inliers
        if (len(Inlier_test) > len(Inliers)):
            Bestfit = [center_x, center_y, r] #Getting the center coordinates and radius of the best fit circle
            Inliers = Inlier_test
            sample_points = [point1, point2, point3]  #Collecting the sample points of the best fit
        iterations+=1
    return Bestfit, Inliers, sample_points
```

Figure 1: Histogram plots before and after equalization

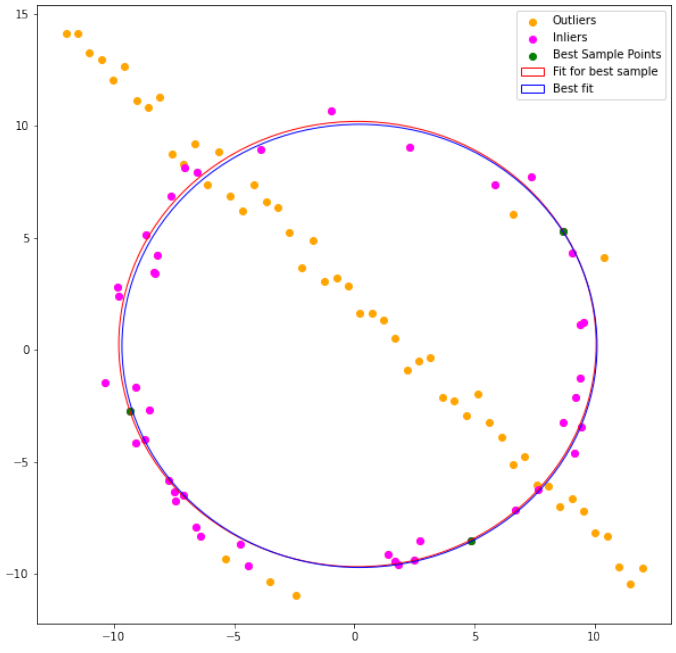
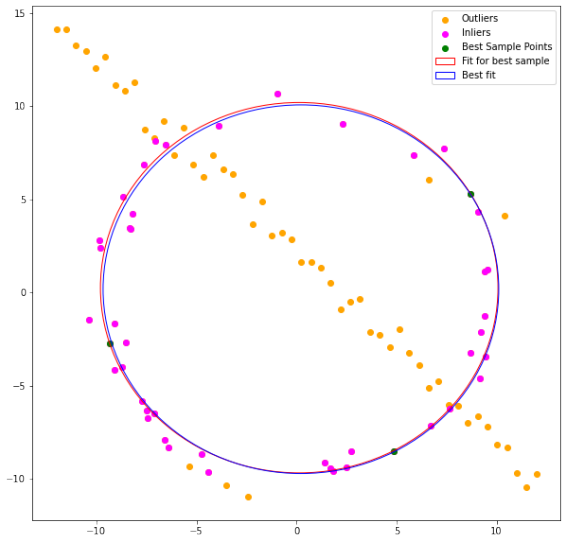


Figure 2: Code for equalizing image

```python
def RANSAC_fit(X,n):
    t = 1
    N = np.inf
    Bestfit = None
    Inliers = []
    sample_points =[]
    p = 0.99 #probability p, at least one random sample is free from outliers
    iterations = 0
    min_s = 3 #Minimum no of points required to find the equation of the circle
    e = 0.5 #Outlier ratio 0.5 was taken for the worst case
    N = np.log(1-p)/np.log(1-(1-e)**min_s)  #calculation of samples
    while N > iterations:
        random_indices= np.random.randint(n, size=min_s)
        point1, point2, point3 = X[random_indices]

        #Calculation of the center coordinates and the radius of the circle passing through the sample points
        coefficientMatrix = np.array([[point2[0] - point1[0], point2[1] - point1[1]], [point3[0] - point1[0], point3[1] - point1[1]]])
        constantMatrix = np.array([[point2[0]**2 - point1[0]**2 + point2[1]**2 - point1[1]**2], [point3[0]**2 - point1[0]**2 + point3[1]**2 - point1[1]**2]])
        invCoefficientMatrix = np.linalg.pinv(coefficientMatrix)

        center_x, center_y = (invCoefficientMatrix@constantMatrix) / 2
        center_x, center_y = center_x[0], center_y[0]
        r = np.sqrt((point1[0]- center_x)**2 + (point1[1] - center_y)**2)

        Inlier_test = []
        #Checking for inliers and appending them into the inlier set.
        for x, y in X:
            dis = np.sqrt((x - center_x)**2 + (y - center_y)**2)
            if (np.abs(dis - r) < t):
                Inlier_test.append([x,y])

        #Checking whether the number of current inliers is greater than the past inliers
        if (len(Inlier_test) > len(Inliers)):
            Bestfit = [center_x, center_y, r] #Getting the center coordinates and radius of the best fit circle
            Inliers = Inlier_test
            sample_points = [point1, point2, point3]  #Collecting the sample points of the best fit
        iterations+=1
    return Bestfit, Inliers, sample_points
```

(a) label 1



(b) label 2

Figure 3: 2 Figures side by side

```python
def RANSAC_fit(X,n):
    t = 1
    N = np.inf
    Bestfit = None
    Inliers = []
    sample_points =[]
    p = 0.99 #probability p, at least one random sample is free from outliers
    iterations = 0
    min_s = 3 #Minimum no of points required to find the equation of the circle
    e = 0.5 #Outlier ratio 0.5 was taken for the worst case
    N = np.log(1-p)/np.log(1-(1-e)**min_s)  #calculation of samples
    while N > iterations:
        random_indices= np.random.randint(n, size=min_s)
        point1, point2, point3 = X[random_indices]

        #Calculation of the center coordinates and the radius of the circle passing through the sample points
        coefficientMatrix = np.array([[point2[0] - point1[0], point2[1] - point1[1]], [point3[0] - point1[0], point3[1] - point1[1]]])
        constantMatrix = np.array([[point2[0]**2 - point1[0]**2 + point2[1]**2 - point1[1]**2], [point3[0]**2 - point1[0]**2 + point3[1]**2 - point1[1]**2]])
        invCoefficientMatrix = np.linalg.pinv(coefficientMatrix)

        center_x, center_y = (invCoefficientMatrix@constantMatrix) / 2
        center_x, center_y = center_x[0], center_y[0]
        r = np.sqrt((point1[0]- center_x)**2 + (point1[1] - center_y)**2)

        Inlier_test = []
        #Checking for inliers and appending them into the inlier set.
        for x, y in X:
            dis = np.sqrt((x - center_x)**2 + (y - center_y)**2)
            if (np.abs(dis - r) < t):
                Inlier_test.append([x,y])

        #Checking whether the number of current inliers is greater than the past inliers
        if (len(Inlier_test) > len(Inliers)):
            Bestfit = [center_x, center_y, r] #Getting the center coordinates and radius of the best fit circle
            Inliers = Inlier_test
            sample_points = [point1, point2, point3]  #Collecting the sample points of the best fit
        iterations+=1
    return Bestfit, Inliers, sample_points
```

Figure 4: label1

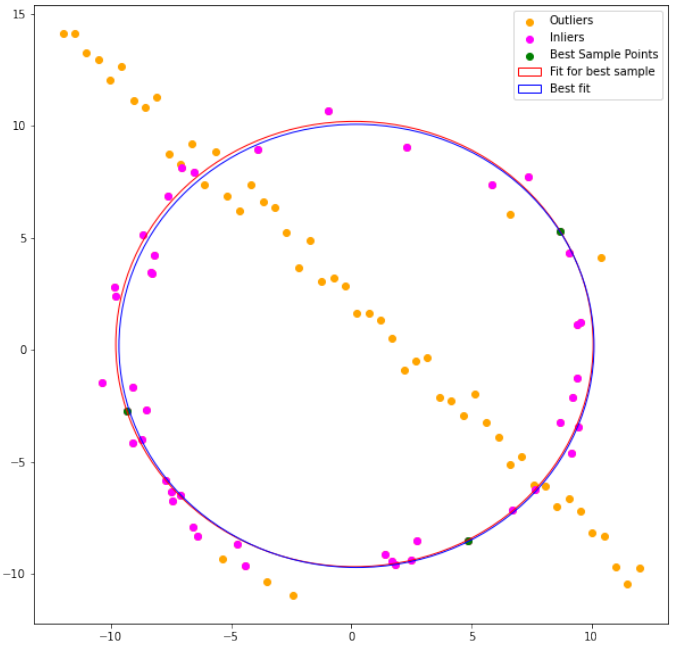


Figure 5: label2

2 Performing Gamma Correction($\gamma = 0.5$) and Calculating Histograms

Gamma correction is one of the basic intensity transformation that can be used to enhance contrast in an image. In this question we have to do gamma correction to the L plane of the image in the Lab color space. The transformation was done by considering $\gamma = 0.5$. The output pixel gets a value by going through a transformation as shown in equation 1. The results are shown below.

$$Outputpixel = 255 * \left(\frac{Inputpixel}{255}\right)^{\gamma} \tag{1}$$

Histograms were plot for the initial image and the gamma corrected image. We can observe that the peak

Figure 6: Results of doing gamma correction to a single plane ($\gamma = 0.5$)

of the histogram has moved slightly towards the right side. This shows that certain pixel values of the gamma corrected image is higher than the original image. This can be observed in the images of figure 5. The darkness of the original has reduced. This phenomenon can be obesreved when $\gamma < 1$. When $\gamma > 1$, the image gets darker than the original image and the peak of the histogram will shift towards left further to a certain extent and then pixels in the right will shift towards left. The Histograms are shown below.

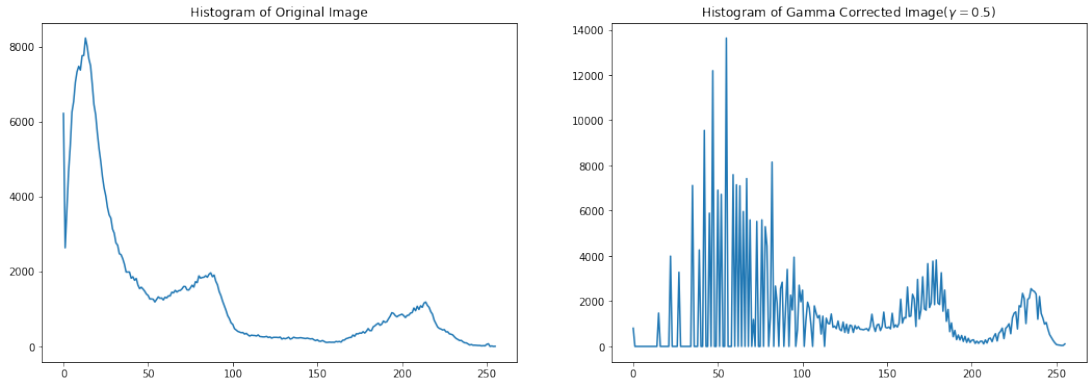


Figure 7: Histograms of the original image and the gamma corrected image

3 Histogram Equalization

Histogram equalization is also an intensity transformation that can be performed in an image such that the pixels are distributed all over the region that the pixels could take values. The result will be an improved version of the image. In this question we have to use a manual function for equalizing the image. Initially, the histogram of the input image is obtained. Then the cdf value is calculated at each point. The calculated cdf value is multiplied by the maximum possible pixel value (L-1) and divided by the product of the dimensions of the input image. The transformation is given by equation 2.

$$s_k = \frac{L-1}{MN} \sum_{j=0}^{k} n_j \tag{2}$$

Here M and N are the dimensions of the input image. The value of k ranges from 0 to L-1. Finally, the histogram of the output image is obtained. The histograms of the Equalized images created from the custom function and the in-built opencv function are plotted together and compared. We can obtain almost similar histogram plots in both cases. The results are shown in the following figure.

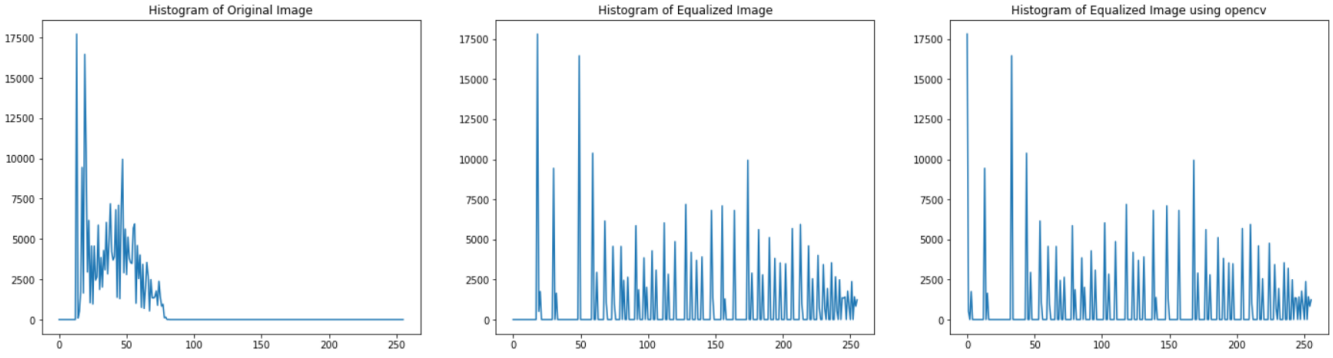


Figure 8: Histogram plots before and after equalization

4 Zooming a given image

We need to zoom images to do an analysis of these images. There are various techniques in zooming. Nearest neighbor interpolation, bilinear interpolation, bicubic interpolation are some of those techniques. In this question, we consider nearest neighbor interpolation and bilinear interpolation to calculate the normalized sum of squared difference (SSD).

```python
#Equalization function
def Equalize(Original_image, Original_image_histogram, S):
    N = len(Original_image_histogram)
    Image = np.zeros((S[0], S[1]))
    Equalize_List = []
    for i in range(N):
        Sumlist = Original_image_histogram[:i+1]
        Value = ((N-1)/(S[0]*S[1]))*(sum(Sumlist))
        Equalize_List.append(Value)
    Equalize_List = np.round(Equalize_List)
    for i in range(S[0]):
        for j in range(S[1]):
            Value = Original_image[i,j]
            Image[i,j] = Equalize_List[Value]
    Image = Image.astype(np.uint8)
    return Image
```

Figure 9: Code for equalizing image

(a) In this part, we zoom the image using nearest neighbor interpolation method. The image pixels values are determined by finding the pixel value at the nearest pixel position to the required pixel position.

```python
#Zoom functon
def NearestNeighborZoom(Image, scale):
    shape = np.shape(Image)
    Rows = int(shape[0]*scale)
    Columns = int(shape[1]*scale)
    Zoomed_image = np.zeros((Rows, Columns, shape[2]), dtype = np.uint8)
    for i in range(0, Rows):
        for j in range(0, Columns):
            X = min(shape[0]-1, round(i/scale))
            Y = min(shape[1]-1, round(j/scale))
            Zoomed_image[i, j] = Image[int(X), int(Y)]
    return Zoomed_image
```

Figure 10: Code for nearest neighbor interpolation Zooming

(b) Bilinear interpolation technique is a technique where the nearest 4 neighbors of the required point are taken and 2 values in the x direction and y direction are calculated by multiplying the input pixel values with the ratio on how the point is located and summing them up. Then the final output pixel is obtained by doing a similar step for the immediate results obtained earlier.

The SSD values calculated for 3 images under each zooming technique are shown in the table below. From this table we can conclude that bilinear interpolation zooming is better than nearest neighbor interpolation zooming, since the SSD values in bilinear interpolation method are less than that in nearest neighbor interpolation method.

Image	**SSD-Nearest Neighbor Interpolation**	**SSD-bilinear Interpolation**
Image 1	641.7879	628.1125
Image 2	268.6875	259.3884
Image 3	391.3376	369.592

Table 1: SSD Values.

```python
def BilinearZoom(Image, scale):
    shape = np.shape(Image)
    Rows = int(shape[0]*scale)
    Columns = int(shape[1]*scale)
    Zoomed_image = np.zeros((Rows, Columns, shape[2]), dtype = Image.dtype)
    for m in range(0, Rows):
        for n in range(0, Columns):
            h = m/scale
            w = n/scale
            h_prev = int(max(0, np.floor(h)))
            w_prev = int(max(0, np.floor(w)))
            h_next = int(min(shape[0]-1, np.ceil(h)))
            w_next = int(min(shape[1]-1, np.ceil(w)))
            if (h_next==h_prev) and (w_next==w_prev):
                Pixel = Image[int(h), int(w), :]
            elif(h_next == h_prev):
                Immediate1 = Image[int(h), int(w_prev), :]
                Immediate2 = Image[int(h), int(w_next), :]
                Pixel = Immediate1*(w_next - w) + Immediate2 * (w - w_prev)
            elif (w_next == w_prev):
                Immediate1 = Image[int(h_prev), int(w), :]
                Immediate2 = Image[int(h_next), int(w), :]
                Pixel = Immediate1*(h_next - h) + Immediate2*(h - h_prev)
            else:
                pixel1 = Image[h_prev, w_prev, :]
                pixel2 = Image[h_next, w_prev, :]
                pixel3 = Image[h_prev, w_next, :]
                pixel4 = Image[h_next, w_next, :]
                Immediate1 = pixel1*(h_next - h) + pixel2*(h - h_prev)
                Immediate2 = pixel3*(h_next - h) + pixel4*(h - h_prev)
                Pixel = Immediate1*(w_next - w) + Immediate2*(w - w_prev)
            Zoomed_image[m, n, :] = Pixel
    return Zoomed_image
```

Figure 11: Code for nearest bilinear interpolation Zooming

5 Sobel Filtering

Sobel Filtering is a filtering technique that helps to filter out the edges of a given image. Sobel vertical filter is used to find the horizontal edges of the image, while the sobel horizontal filter is used to identify the vertical edges.

(a) In this part, the image is sobel filtered using the opencv in-built function filter2D.

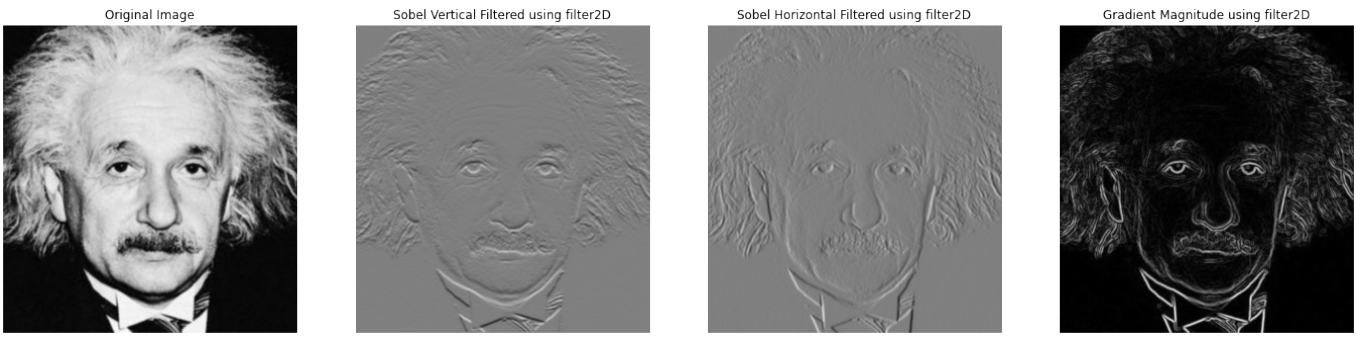


Figure 12: Sobel filtering using filter2D function

(b) In this part, we construct a filtering function code so that the image can be filtered using any given kernel. We initially determine the padding values required to pad the image. Then the kernel is run through the whole image and the pixel values located in the locations, covered by the kernel are multiplied by the values in the kernel and they are summed up to get the output pixel value. We get similar results as using the in built function.

(c) In this part, 2 arrays [1 ,2, 1] and $[-1, 0, 1]^T$ are defined and the input image is filtered using the arrays one by one for vertical filtering. The image is filtered with the transposes of the arrays for horizontal filtering. The results obtained are similar to the results obtained using the filter2D function. This shows the associative property of the filters.

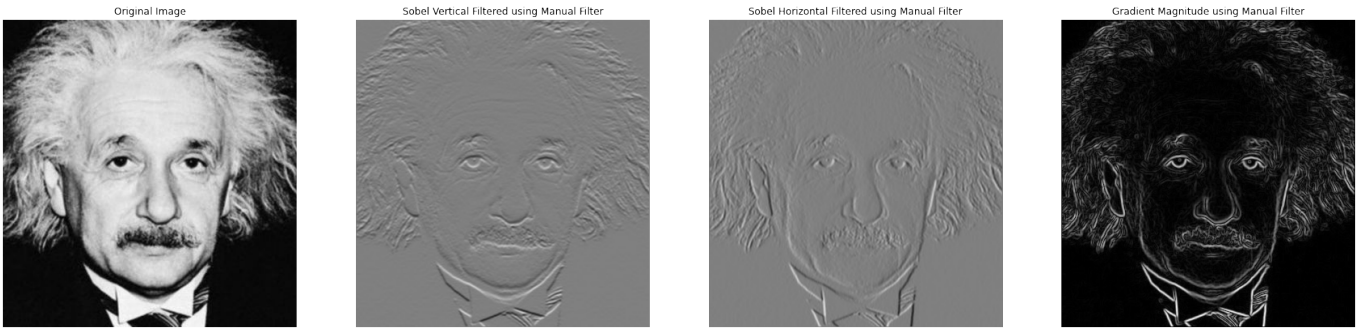


Figure 13: Sobel filtering using custom filter function

```python
#Manual Sobel Filtering
def ManualFilter(Image, Kernel):
    shape = np.shape(Kernel)
    k_hh, k_hw = int(np.floor(shape[0]/2)), int(np.floor(shape[1]/2))
    h, w = Image.shape
    Image_changedtype = Image.
    filtered_image = np.zeros(Image.shape, 'float')
    for i in range(k_hh, h - k_hh):
        for j in range(k_hw , w - k_hw):
            filtered_image[i, j] = np.dot(Image_changedtype[i - k_hh : i + k_hh + 1 , j - k_hw : j + k_hw + 1].flatten(), Kernel.flatten())
    return filtered_image
```

(parameter) Image: Any

Figure 14: Code for manual filtering

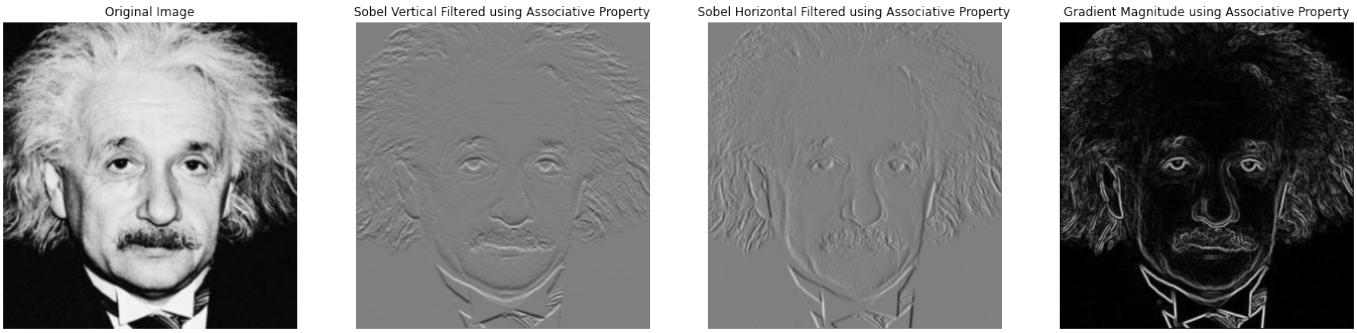


Figure 15: Sobel filtering using associative property

6 Enhancing a given Image

(a) A mask of the flower in the input image is created binary thresholding function in opencv. The mask image is read, foreground and background models are created accordingly. Finally Grabcut function was executed to separate the foreground flower and the background image. A value for binary thresholding was set such that the flower region is only shown as white. A custom mask was used instead of rectangular mask to prevent errors of selecting the exact position.

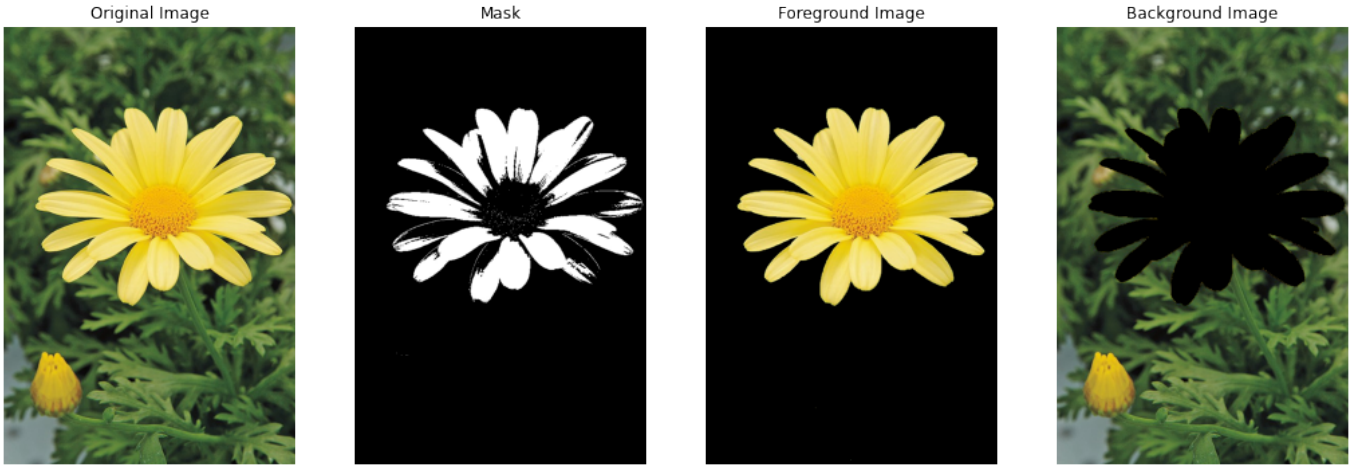


Figure 16: Separating foreground and background

(b) The separated foreground image is enhanced by increasing the saturation while the background is blurred using the Gaussian blur function. the foregroundand background sections are merged together later.

Figure 17: Enhancing the image

(c) When the kernel size increases, the edges of the foreground image region which is black in the background image gains a significant value after gaussian blur so that when the foreground image is added to the background image, the edges of the foreground image do not gain the expected pixel value. Therefore, the edges are quite dark.

Code Files : All relevant codes and files can be found in **GitHub**.