University of Moratuwa

In19-S4-EN2550 - Fundamentals of Image Processing and Machine Vision

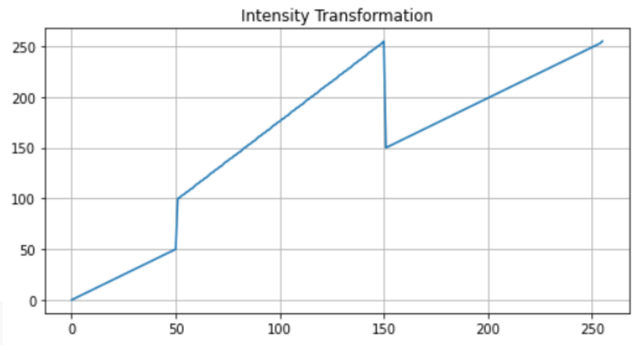
Assignment 1 on Intensity Transformations and Neighborhood Filtering

190586H

W.K.D.D. Senuruk

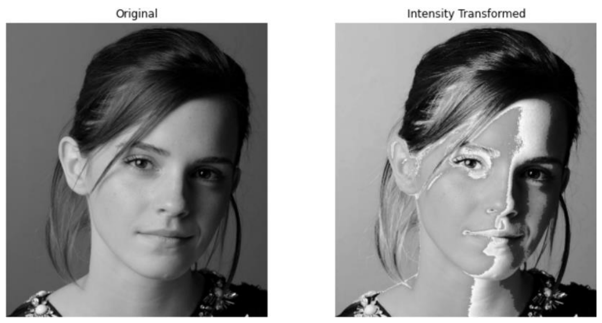
March 05th 2022

(1)



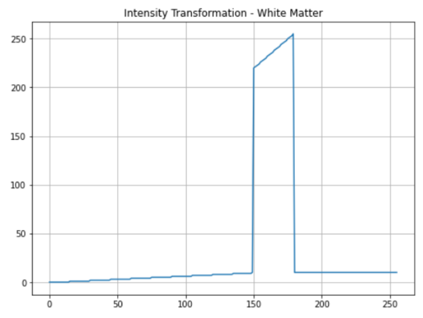
According to this intensity transformation function I gave intensities to all the pixels in the image of Emma Watson. The pixels which have intensities in the ranges (0 - 50) and (150 - 255) did not changed. Only pixels that in intensity range (50 - 150) have increased.

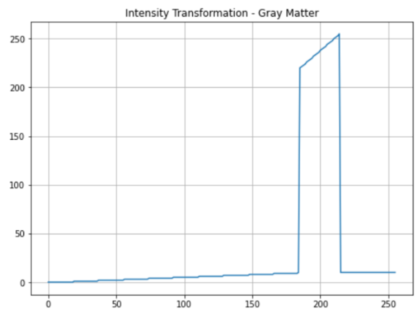
```
t1 = np.linspace(0, 50, 51)
t2 = np.linspace(100, 255, 100)
t3 = np.linspace(150, 255, 105)

t = np.concatenate((t1, t2, t3), axis = 0).astype(np.uint8)
plt.subplots(1, 1, figsize = (8, 4))
plt.title("Intensity Transformation")
plt.grid("on")
plt.plot(t)
```



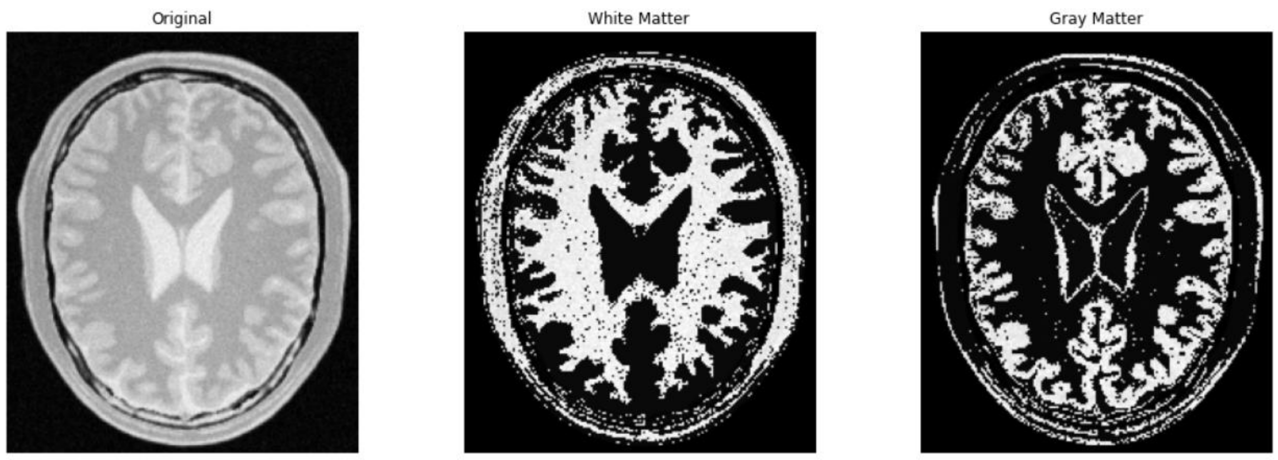
(2)

In brain there are two types of matter called white matter and gray matter. In question 2 we have to accentuate the gray matter and white matter so that we could clearly see them separated. This kinds of situations we have to first check the intensity histogram and identify what are ranges that intensities are distributed. Then we can reduced the intensities of unwanted areas and also increase the intensities of selected areas. You can see below intensity transformations functions that enhance the selected intensity ranges.





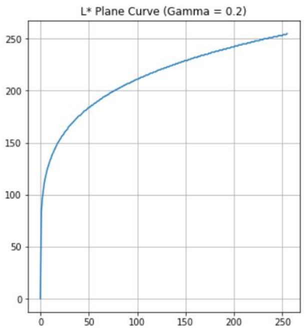
```
t1 = np.linspace(0, 10, 150)
t2 = np.linspace(220, 255, 30)
t3 = np.linspace(10, 10, 76)
t = np.concatenate((t1, t2, t3), axis = 0).astype(np.uint8)

r1 = np.linspace(0, 10, 185)
r2 = np.linspace(220, 255, 30)
r3 = np.linspace(10, 10, 41)
r = np.concatenate((r1, r2, r3), axis = 0).astype(np.uint8)

assert len(t) == 256
img2_t = cv.LUT(img2, t)

assert len(r) == 256
img2_r = cv.LUT(img2, r)
```

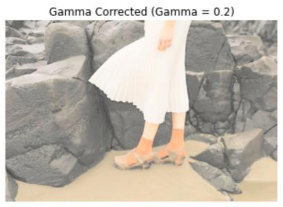


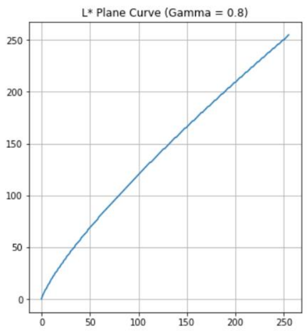
(3)(a)

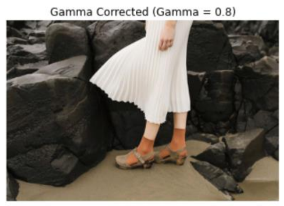
L*a*b* is a color space like BGR, RGB, HSV and CMYK. It is also known as Lab color space and L, A, B color space. It is very much familiar to HSV color space. Here we can identify three planes L, a and b. L plane refers to the lightness of the image. Here we have given the so called gamma correction to the L plane. I have plotted four graphs with gamma = 0.2, 0.8, 1.2 and 2.0. when gamma = 1 it gives the identity transformation. When gamma is less than 1 low intensities gets wide range of intensities. When gamma is greater than 1 high intensities gets wide range of inensities.

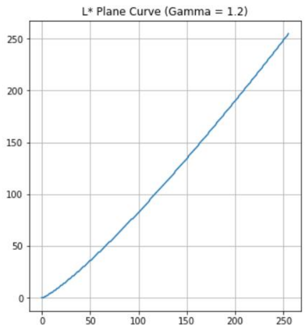
```python
gamma = [0.2, 0.8, 1.2, 2.0]
hist_img3_lab = []

lab = cv.cvtColor(img3, cv.COLOR_BGR2LAB)
lab_n = lab.copy()

for i in gamma:
    k = np.array([(p/255)**(i)*255 for p in range(0, 256)]).astype(np.uint8)
    for x in range(0, len(lab)):
        for y in range(0, len(lab[0])):
            lab_n[x, y][0] = ((lab[x, y][0] / 255)**(i)) * 255

    hist_img3_lab.append(cv.calcHist([lab_n], [0], None, [256], [0, 256]))
```

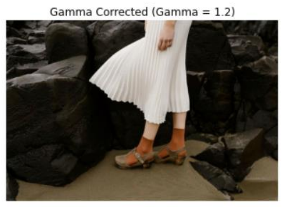


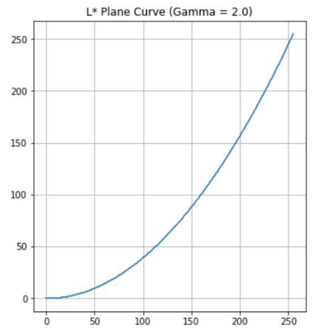


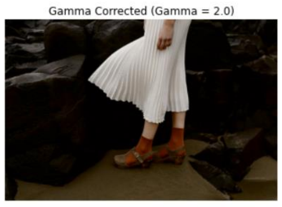






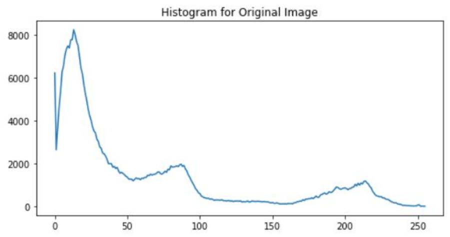


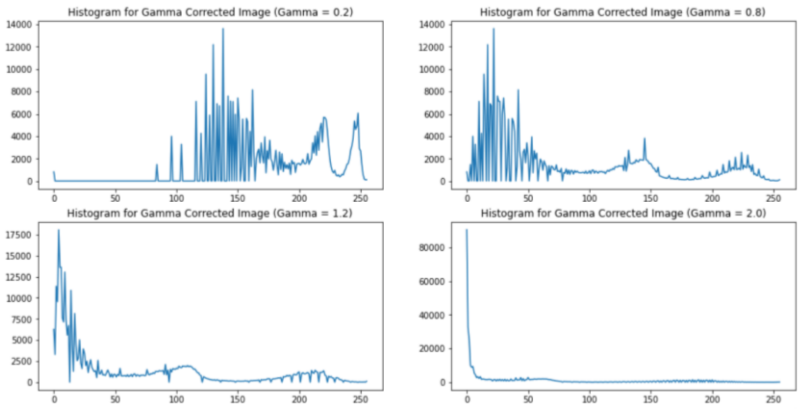




(3)(b)

Here you can see the original image and its' Intensity Histogram which shows the intensity distribution of the image. Below four intensity histograms shows the intensity distribution of L planes in the four situations where gamma = 0.2, 0.8, 1.2 and 2.0.

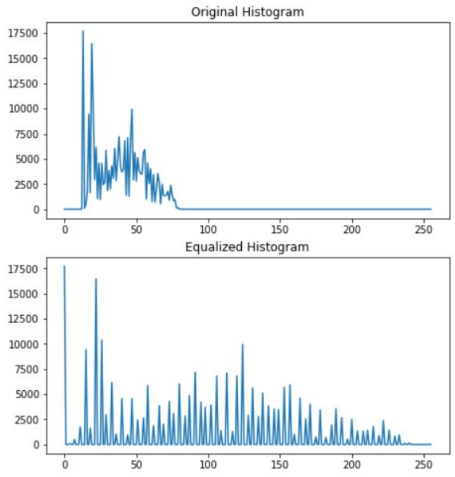


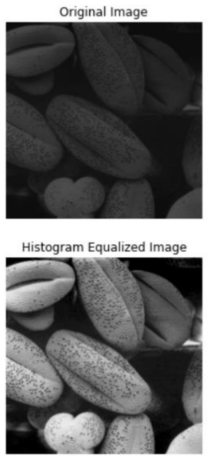


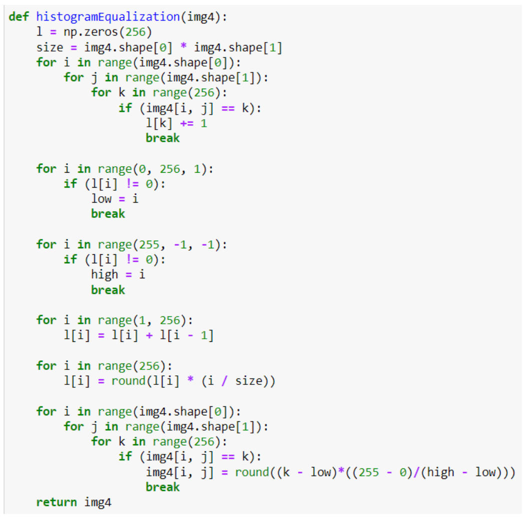
```python
hist_img3 = cv.calcHist([img3], [0], None, [256], [0, 256])
fig, ax = plt.subplots(1, 2, figsize=(18, 4))
ax[0].imshow(cv.cvtColor(img3, cv.COLOR_BGR2RGB))
ax[0].axis("off")
ax[0].set_title("Original")

ax[1].plot(hist_img3)
ax[1].set_title("Histogram for Original Image")

fig, ax = plt.subplots(2, 2, figsize=(16, 8))
k = 0
f = 0
j = 0
for i in hist_img3_lab:
    ax[k, f].plot(i)
    ax[k, f].set_title("Histogram for Gamma Corrected Image (Gamma = " + str(gamma[j]) + ")")
    f += 1
    if (f == 2):
        k += 1
        f = 0
    j += 1

plt.show()
```

(4)

We can use calcHist() function to get the histogram and also equalizeHist() function to distribute the intensity histogram in the whole 0 to 255 range. But in here I have wrote a new function to equalize the histogram of a given image. You can see the intensities of the original image is in the range of 10 to 80. In this function it is stretched so that it fits for the whole range.



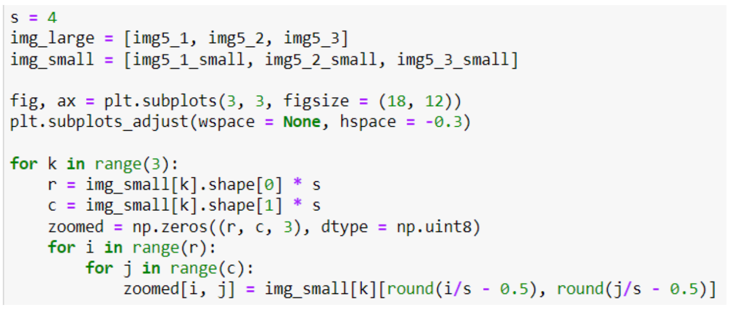




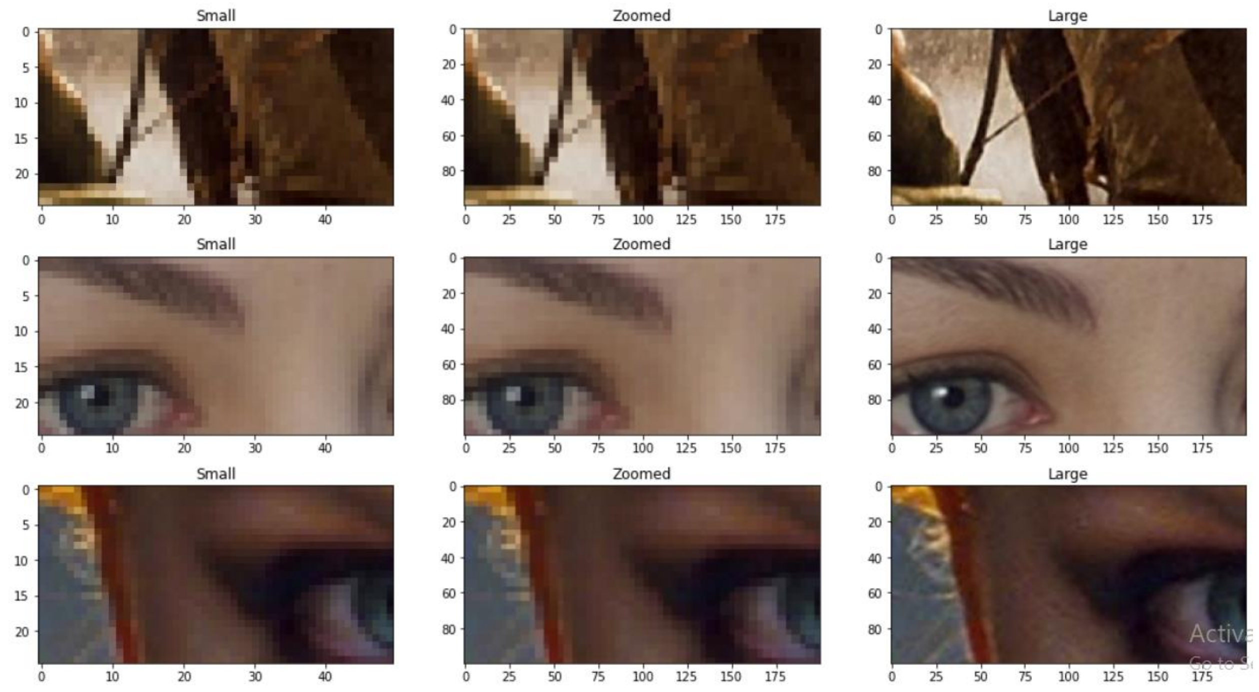
```python
def histogramEqualization(img4):
    l = np.zeros(256)
    size = img4.shape[0] * img4.shape[1]
    for i in range(img4.shape[0]):
        for j in range(img4.shape[1]):
            for k in range(256):
                if (img4[i, j] == k):
                    l[k] += 1
                    break

    for i in range(0, 256, 1):
        if (l[i] != 0):
            low = i
            break

    for i in range(255, -1, -1):
        if (l[i] != 0):
            high = i
            break

    for i in range(1, 256):
        l[i] = l[i] + l[i - 1]

    for i in range(256):
        l[i] = round(l[i] * (i / size))

    for i in range(img4.shape[0]):
        for j in range(img4.shape[1]):
            for k in range(256):
                if (img4[i, j] == k):
                    img4[i, j] = round((k - low)*((255 - 0)/(high - low)))
                    break
    return img4
```

(5)(a)

Nearest Neighbor method is used for zooming images. Here I have implement a code that goes to every pixel of a large blank image and give a relevant value according to a smaller size image. And it is done by the method of giving the value of the nearest pixel when the pixel coordinate lies among four pixels. Here three smaller size images have 4 times zoomed. Only a selected area is shown so that you can clearly identify the differences.



```python
s = 4
img_large = [img5_1, img5_2, img5_3]
img_small = [img5_1_small, img5_2_small, img5_3_small]

fig, ax = plt.subplots(3, 3, figsize = (18, 12))
plt.subplots_adjust(wspace = None, hspace = -0.3)

for k in range(3):
    r = img_small[k].shape[0] * s
    c = img_small[k].shape[1] * s
    zoomed = np.zeros((r, c, 3), dtype = np.uint8)
    for i in range(r):
        for j in range(c):
            zoomed[i, j] = img_small[k][round(i/s - 0.5), round(j/s - 0.5)]
```

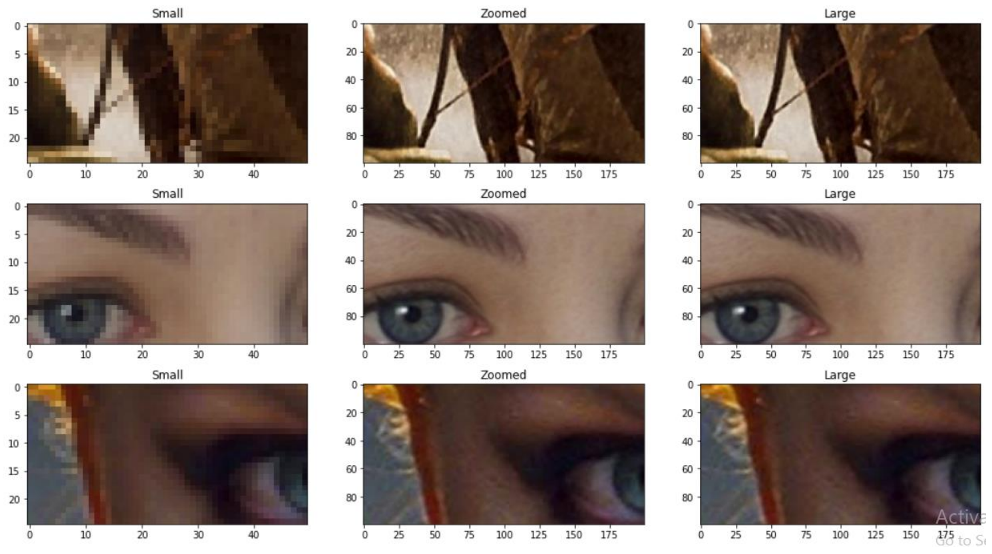


(5)(b)

Bilinear Interpolation is another method to zoom an image. In this method we calculate the values regarding the position between pixel coordinates. Value depend on how far does the imaginary coordinate lies from a relevant pixel. This way we are able to get a very good zoomed image.

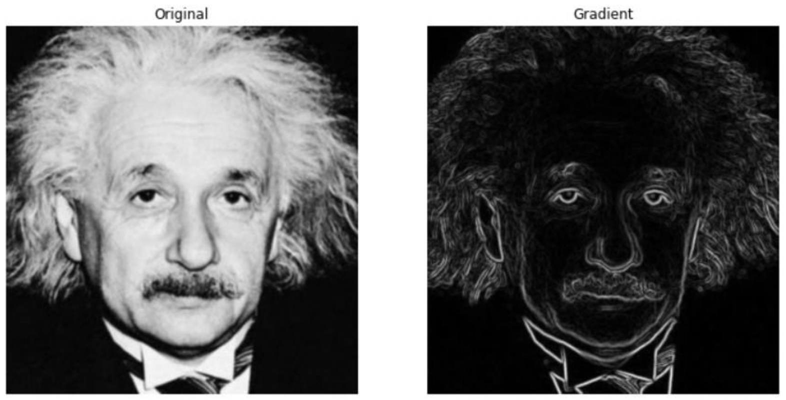
```
for k in range(3):
    r = img_small[k].shape[0] * s
    c = img_small[k].shape[1] * s
    zoomed_im = np.zeros((r, c, 3), dtype = i)
    for i in range(r-1):
        for j in range(c-1):
            e = i/s
            f = j/s

            a = int(np.ceil(e))
            b = int(np.floor(e))
            c = int(np.ceil(f))
            d = int(np.floor(f))

            lb = [b, d]
            lt = [a, d]
            rb = [b, c]
            rt = [a, c]

            if (a >= img_small[k].shape[0]):
                lt[0] = img_small[k].shape[0] - 1
                rt[0] = img_small[k].shape[0] - 1
            if (c >= img_small[k].shape[1]):
                rb[1] = img_small[k].shape[1] - 1
                rt[1] = img_small[k].shape[1] - 1

            rv = e-lb[0]
            rh = f-lb[1]
            val01 = img_small[k][lt[0]][lt[1]]*(rv) + img_small[k][lb[0]][lb[1]]*(1-rv)
            val02 = img_small[k][rt[0]][rt[1]]*(rv) + img_small[k][rb[0]][rb[1]]*(1-rv)
            zoomed_im[i][j] = np.rint(val01*(1-rh) + val02*(rh))
```

```
for k in range(3):
    ssd1 = np.sum((img_large[k][:,:,0:3] - zoomed[k][:,:,0:3])**2) / img_large[k].size
    ssd2 = np.sum((img_large[k][:,:,0:3] - zoomed_im[k][:,:,0:3])**2) / img_large[k].size
    print(ssd1)
    print(ssd2)
```



(6)(a)

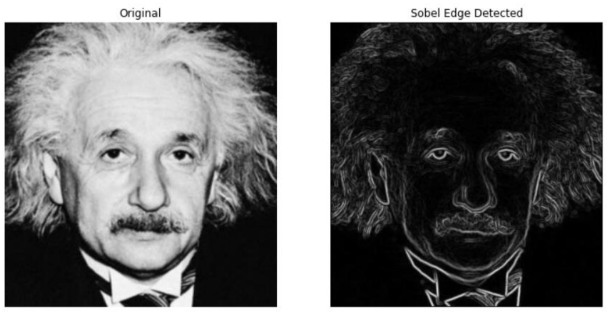
Here I have used the existing filter2D() function to do the sobel filtering on the image of Albert Einstein. First detecting the vertical edges & horizontal edges and then combining them to get the gradient image that shows the image with enhance edges.



```
sobel_v= np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.float32)
f_x = cv.filter2D(img6, -1, sobel_v)
sobel_h = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)
f_y = cv.filter2D(img6, -1, sobel_h)
grad_mag = np.sqrt(f_x**2 + f_y**2)

fig, ax = plt.subplots(1, 2, figsize = (12, 6))

ax[0].imshow(img6, cmap='gray', vmin=0, vmax=255)
ax[0].set_title("Original")
ax[0].axis("off")

ax[1].imshow(grad_mag, cmap = "gray")
ax[1].set_title("Gradient")
ax[1].axis("off")

plt.show()
```

(6)(b)

I wrote a new code that do the sobel filtering as the filter2D() function in open cv. It goes every pixel and getting values from its neighborhood and multiplying them with relevant values and then taking the cumulative summation. Like that I took two values regarding the vertical and horizontal gradients. Finally combining the two values and assigned it with a normalization to the blank image which is initially created.

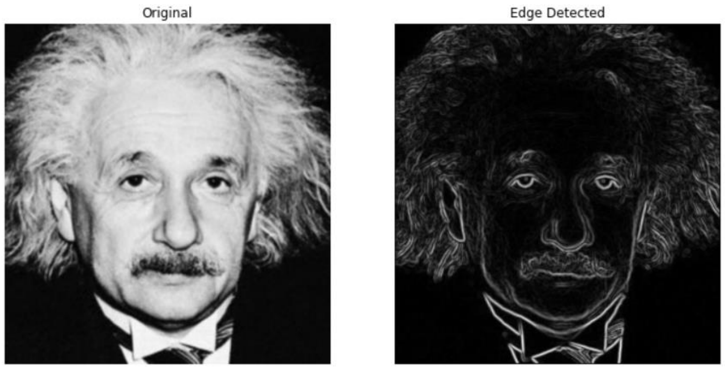


```
m = img6.shape[0]
n = img6.shape[1]
im = np.zeros((m, n), np.uint8)
for i in range(1, m-1):
    for j in range(1, n-1):
        G_x = img6[i-1,j-1]*(-1) + img6[i-1,j]*(-2) + img6[i-1,j+1]*(-1) + img6[i+1,j-1]*1 + img6[i+1,j]*2 + img6[i+1,j+1]*1
        G_y = img6[i-1,j-1]*(-1) + img6[i,j-1]*(-2) + img6[i-1,j+1]*1 + img6[i+1,j-1]*(-1) + img6[i,j+1]*2 + img6[i+1,j+1]*1
        val = np.sqrt(G_x**2 + G_y**2)
        im[i, j] = (val / 1020) * 255
```

(6)(c)

Again another code was written for the sobel edge detection considering the implementation of sobel kernel. Using the matrix multiplication methods this sobel edge detection was carried out. np.multiply(), np.sum() function are used in this method. Taking a matrix from the image with same dimensions as the sobel kernel and doing element wise multiplication and finally addition of matrix elements have carried out. This criteria was used to get horizontal and vertical values regarding gradient.

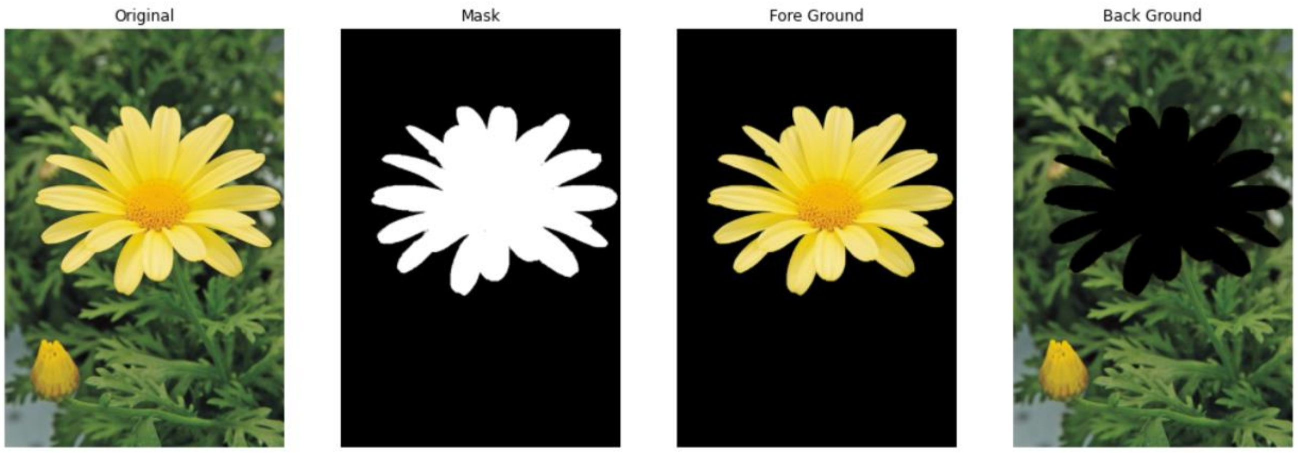
```
img = np.zeros((m, n), np.uint8)

mul = np.multiply((np.array([[1],[2],[1]], np.float32)),
                  (np.array([1,0,-1], np.float32)))
sh = (-1) * mul
sv = sh.T

for i in range(1, m-1):
    for j in range(1, n-1):
        p = np.array([[img6[i-1,j-1],img6[i-1,j],img6[i-1,j+1]],
                      [img6[i,j-1],img6[i,j],img6[i,j+1]],
                      [img6[i+1,j-1],img6[i+1,j],img6[i+1,j+1]]])
        mul_v = p * sv
        mul_h = p * sh
        cum_sum_v = np.sum(mul_v)
        cum_sum_h = np.sum(mul_h)
        val = np.sqrt(cum_sum_v**2 + cum_sum_h**2)
        img[i, j] = (val / 1020) * 255
```



(7)(a)

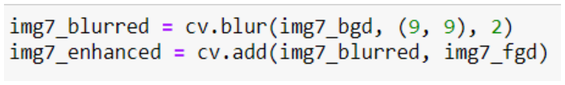
Here I have used the grabCut() function in open cv to segment the image of a beautiful yellow flower. I have given a rectangle to show the area that flower lies on the image. You can see the mask in the second image. The foreground is the yellow flower which we segmented out from the image. The other part is the background of the image. Background doesn't have yellow flower. The area of the flower is blank. Mask1 is used to identify the foreground by multiplying with the image. It carries 1 for definite foreground and probable foreground. Mask2 carries 1 for definite background and probable background so that it can be used to get the background image by multiplying.

```
img7_original = img7.copy()
mask = np.zeros(img7.shape[:2], np.uint8)
rect = (0, 100, 561, 500)
fgdModel = np.zeros((1, 65), np.float64)
bgdModel = np.zeros((1, 65), np.float64)

cv.grabCut(img7, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)

mask1 = np.where((mask==0) | (mask==2), 0, 1).astype("uint8")
img7_fgd = img7*mask1[:, :, np.newaxis]
mask2 = np.where((mask==1) | (mask==3), 0, 1).astype("uint8")
img7_bgd = img7*mask2[:, :, np.newaxis]
```

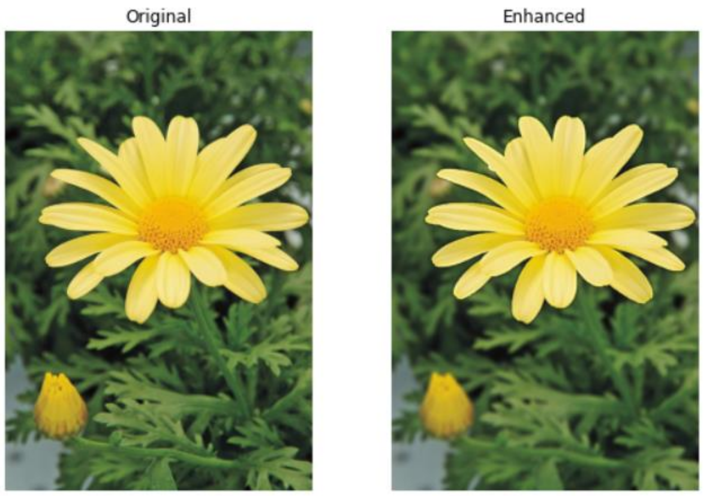


(7)(b)

Next the blur() function was used to blur the background image. After blurring the background the foreground was added to it again. That means the flower was added after blurring the rest of the image. This gives an enhanced image. You can see the difference by observing the below two images. Enhanced image has a much blurred background than the original one.



```
img7_blurred = cv.blur(img7_bgd, (9, 9), 2)
img7_enhanced = cv.add(img7_blurred, img7_fgd)
```



(7)(c)

If you observe the above two images very carefully you can identify that the background area just beyond the edge of the flower is dark than the original image. You might wonder what cause this effect. Normally the kernal for blur is a matrix of 1 which is multiplied by one over kernals' width times kernals' height.

$$\left(\frac{1}{kernals' \, width * kernals' \, height}\right)\begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix}$$

We performed blur function to the background. The area that used to be yellow flower is pitch black. So, when the convolution happens near the black area the pixels near the black area gets more darker intensities than they initially had.

Github Repository Link: *https://github.com/dilansenuruk/ML-Assignment-1.git*