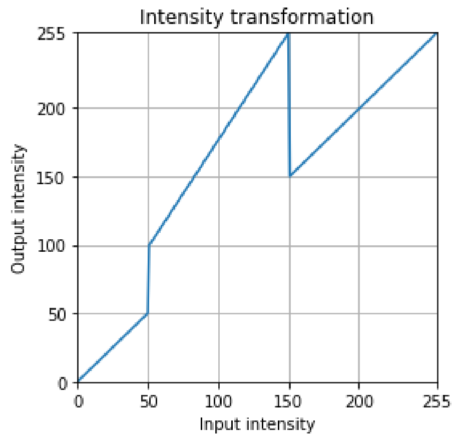
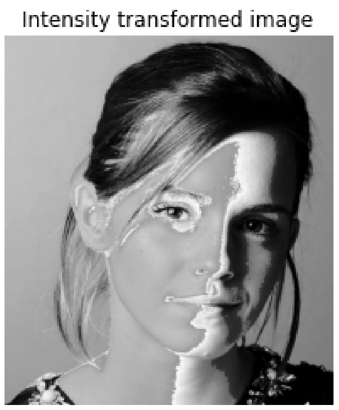
EN2550 – Assignment 01

Intensity Transformations and Neighborhood Filtering

Question 01

Initially, the piecewise intensity transformation is created by using the numpy.linspace and numpy.concatenate methods. Then, by applying the look-up-table transformation on the original image, using the **cv2.LUT** module, the intensity transformation defined by the piecewise function is applied as follows.

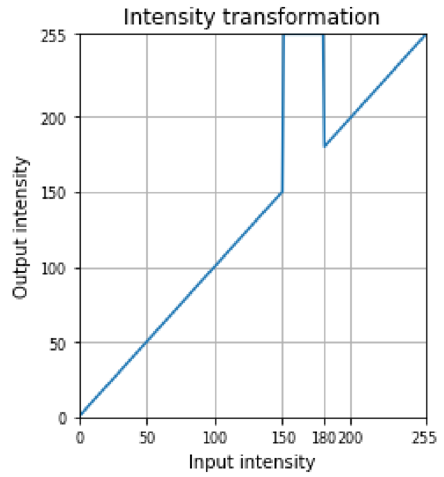


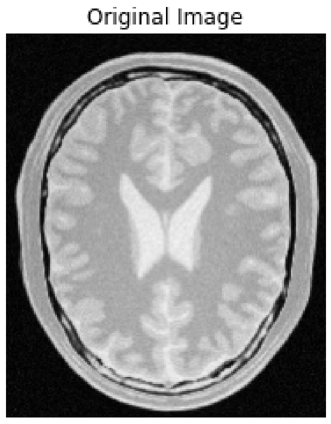


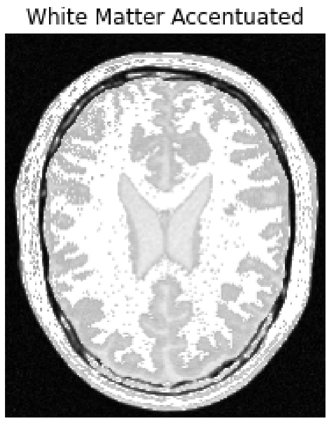
The intensity values of the original image in the range [50, 150] are mapped to the range [100, 255]. Since the intensity transformation is a many-to-one mapping in the full range [0, 255], we can observe blending of certain regions in the output image with respect to the intensity values.

Question 02

(a) White matter



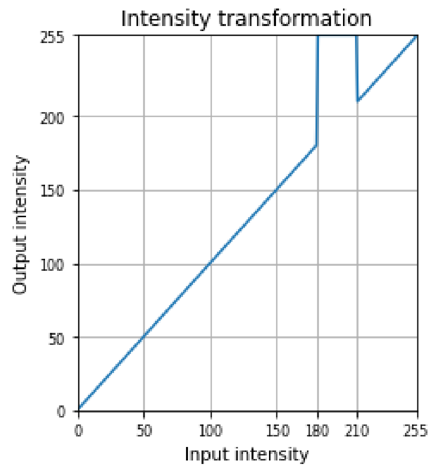


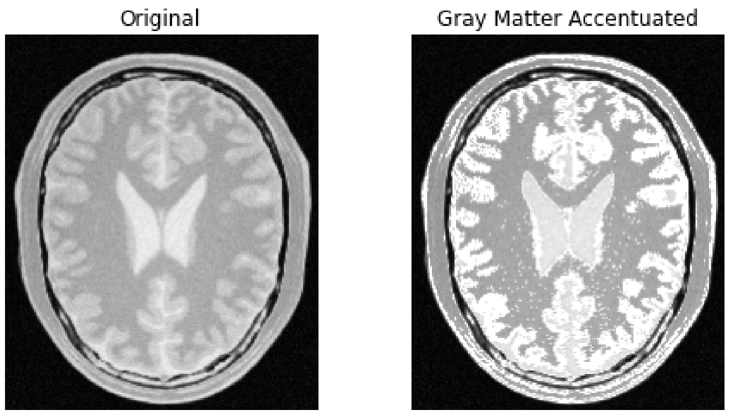


In this approach, the white matter of the image is accentuated without any alteration to the intensities of other regions. This is done by raising the intensity values corresponding to the white matter, found to be in the range [150, 180], to its maximum value of 255. With the insights gained from the previous question, the value 255 is chosen to avoid any blending of regions in the output image. This allows for clear distinction of the accentuated white matter as follows.

(b) Gray Matter

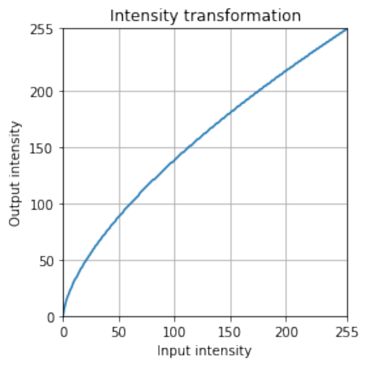
Similarly, the range of values [180, 210] corresponding to gray matter, is accentuated by raising the intensity to its maximum value of 255, to clearly distinguish the gray matter region of the brain.

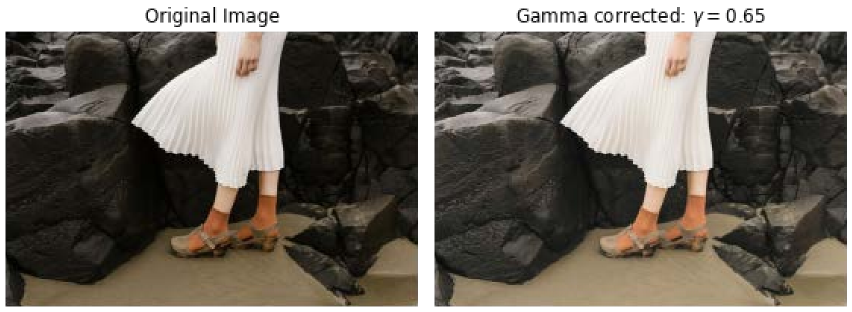




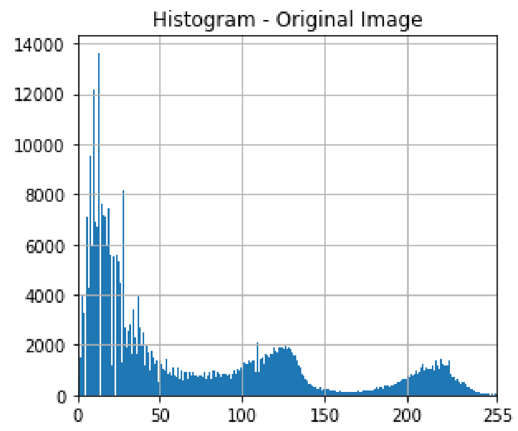
Question 03

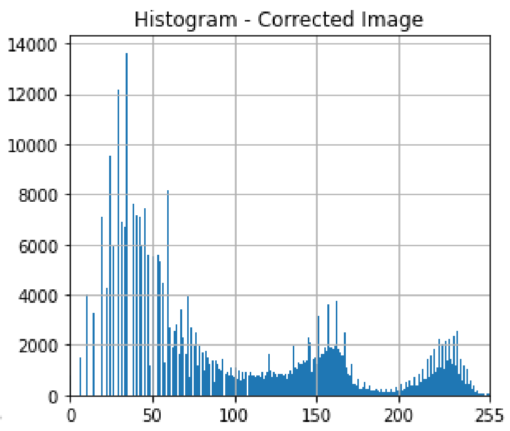
By converting the **BGR** image to **L*a*b** (CIELAB) color space, the channel **L*** is obtained which gives perceptual lightness, independent of color. By applying gamma correction with a value of **γ = 0.65** to the **L*** channel as follows, it is possible to enhance the image brightness without altering the color scheme present in the original image.





The histograms of the original and corrected images are as follows.

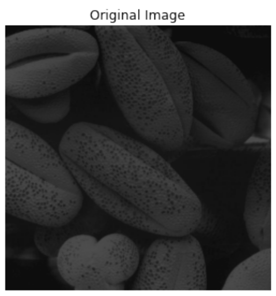


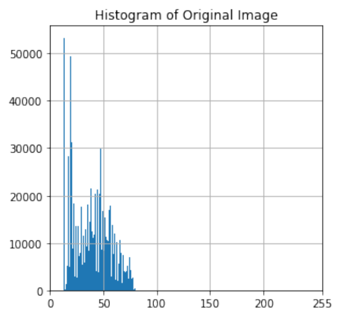


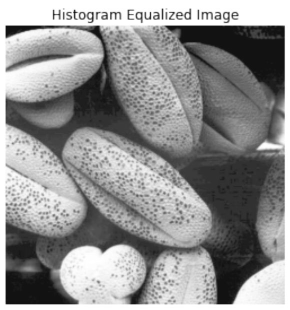
Question 04

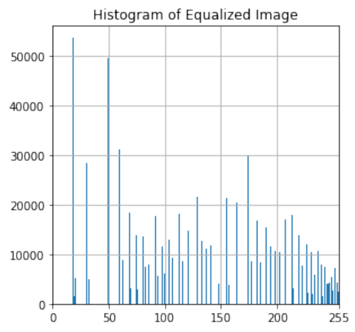
To carry out histogram equalization the following code is used. Initially the image is read in grayscale for efficient processing and the number of occurrences of each intensity level is obtained using the **numpy.histogram** module. Then by implementing the following code script, histogram equalization is carried out.

```
# read the image and find the number of occurrences

f = cv.imread(r'Images/shells.png', cv.IMREAD_GRAYSCALE).astype(np.uint8)
hist, bins = np.histogram(f.ravel(), 256, [0, 256])

# code to carry out histogram equalization

m, n = f.shape
L = 256
t = np.round((L-1)/(m*n) * np.array([np.sum(hist[0:i+1]) for i in range(0,256)])).astype(np.uint8)
assert len(t) == 256
g = cv.LUT(f, t)
```





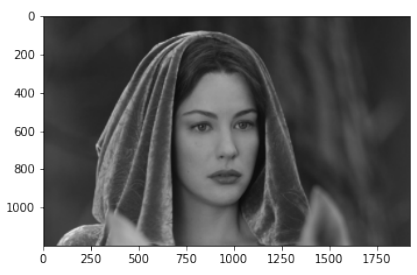


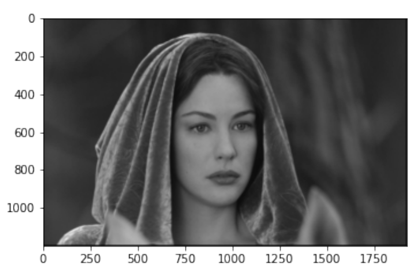


Question 05

```python
def zoom_image(image, scale, type):
""" type: 'nn' for nearest-neighbor, 'bi' for bilinear interpolation"""
    rows = int(np.round(image.shape[0]*scale))
    cols = int(np.round(image.shape[1]*scale))
    zoomed = np.zeros((rows, cols))

# zoom using Nearest-neighbor


    if type == 'nn':
        for i in range(rows):
            for j in range(cols):
                ip = int(np.round(i/scale))
                jp = int(np.round(j/scale))

                if ip >= image.shape[0]: ip = image.shape[0] - 1
                if jp >= image.shape[1]: jp = image.shape[1] - 1

                zoomed[i][j] = image[ip][jp]

 # zoom using bilinear interpolation

    if type =='bi':
        for i in range(rows):
            for j in range(cols):
                ip, jp = i/scale, j/scale
                i1, j1 = i//scale, j//scale
                i2, j2 = i//scale + 1, j//scale + 1

                if i2 >= image.shape[0]: i2 = image.shape[0] - 1
                if j2 >= image.shape[1]: j2 = image.shape[1] - 1

                val1 = image[i1][j1]*(i2-ip)+image[i2][j1]*(ip-i1)
                val2 = image[i1][j2]*(i2-ip)+image[i2][j2]*(ip-i1)

                val = val1*(j2-jp) + val2*(jp-j1)

                zoomed[i][j] = val

    return zoomed
```

The original image followed by the results from zooming by a scale of 4 using nearest neighbor and Bilinear interpolation on **im02.png** is as follows.





The sum of squared differences (SSD) values when zooming-in by a factor of 4 is obtained as below.

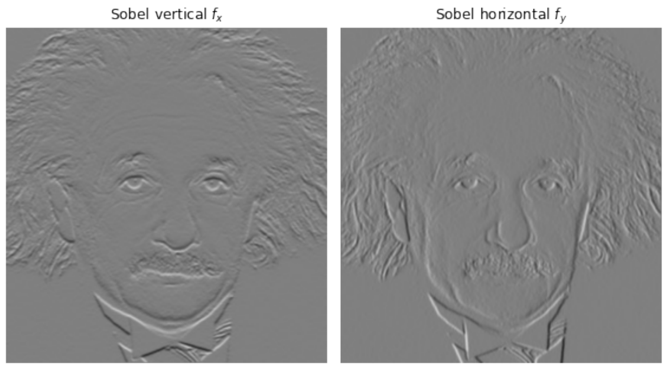
	SSD for Nearest-neighbor	SSD for Bilinear interpolation
im01.png	40	39
im02.png	16	16
im03.png	22	21

As the SSD for bilinear interpolation is lower, it is much closer to the original image provided and is therefore a better zooming algorithm.

Question 06

(a)

Using the in-built **cv2.filter2d** method, it is possible to directly implement convolution with the Sobel kernels to carry out Sobel filtering. This is possible due to the symmetry of the Sobel kernels; a 180º rotation yields the same kernel. Thus, correlation applied to the image using the filter2d results in the required convolution operation.

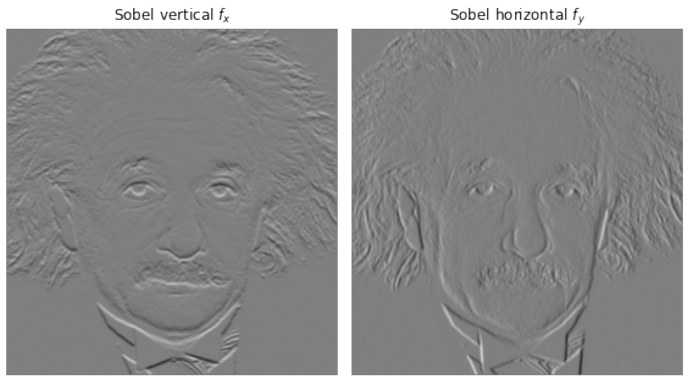


(b)

This convolution can also be carried out in a more trivial manner by using two for loops as implemented in the following code.

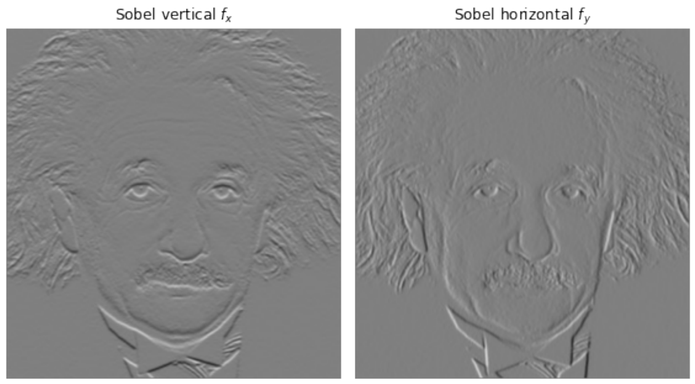
```python
def filter(image, kernel):
    assert kernel.shape[0] % 2 and kernel.shape[1] % 2
    h_gap, w_gap = kernel.shape[0]//2, kernel.shape[1]//2
    h, w = image.shape
    filtered = np.zeros(image.shape, dtype = np.float32)

    for i in range(h_gap, h-h_gap):
        for j in range(w_gap, w-w_gap):
            filtered[i][j] = np.dot(image[i-h_gap:i+h_gap+1, j-w_gap:j+w_gap+1].flatten(), kernel.flatten())

    return filtered
```

The results obtained are similar to that of using filter2D.



(c)

By using the given property, the Sobel filter can be formed by the outer product of two column vectors. Since convolution is associative, we can convolve the image with two column vectors one by one and drastically reduce the time taken (time complexity) in running the operation. The results obtained are as follows, which shows a similar gradient computation in vertical and horizontal directions.

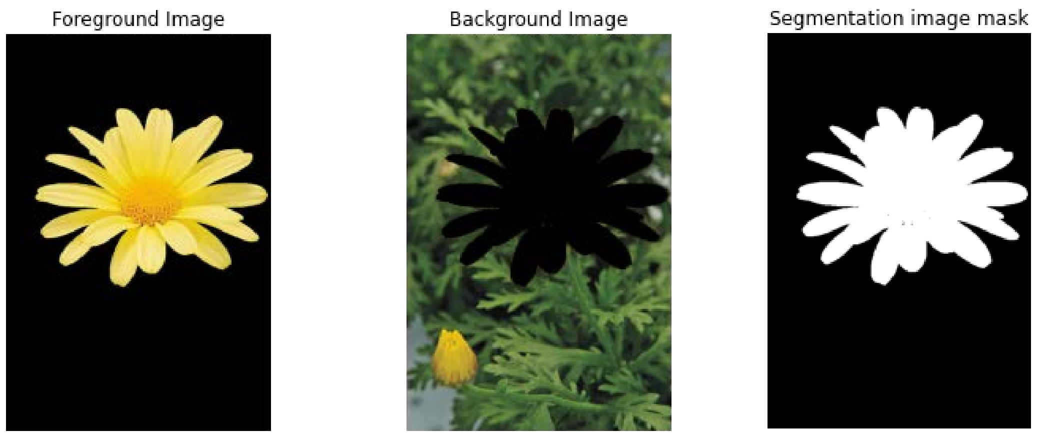


Exercise 07

(a) Using the cv2.grabCut module as in the following code, it is possible to segment the image and carry out required processing of the image.

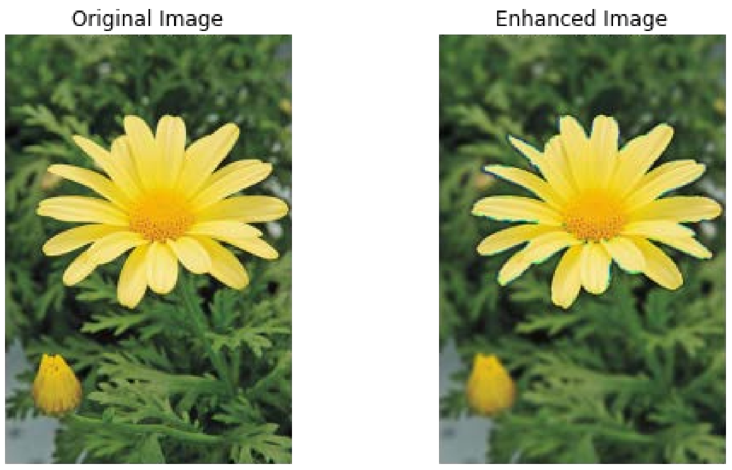
```
mask = np.zeros(img.shape[:2],np.uint8)
bgdModel = np.zeros((1,65),np.float64)
fgdModel = np.zeros((1,65),np.float64)
rect = (60, 150, 500, 400)
cv.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)

#foreground image
mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img_f = img * mask2[:, :, np.newaxis]

#backgroung image
mask3 = np.where((mask==1)|(mask==3),0,1).astype('uint8')
img_b = img * mask3[:, :, np.newaxis]
```

The foreground image, background image and the segmentation mask are as follows.



(b)

Initially, to the extracted background image, a gaussian blur is added with a kernel size of (13, 13) and a standard deviation, sigma = 7. Thereafter, the previously extracted foreground image was added to the blurred background image to obtain the enhanced image. The enhanced image obtained is observed to be similar to the "selective focus" present in most modern digital cameras.



(c)

After applying the gaussian blur to the extracted background image, the edges of the black area will end up getting values from the background, which is not black in color. Hence, along the edge the blurred image will not be completely black. This results in the background just beyond the edge of the flower to be quite dark in the enhanced image.

References

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1681575/

https://en.wikipedia.org/wiki/CIELAB_color_space

https://learnopencv.com/color-spaces-in-opencv-cpp-python/

https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html

Github Repository: https://github.com/chira99/image-processing-opencv-python.git