EN2550-Assignment 1

Intensity Transformations and Neighborhood Filtering

Name: B.S.V.W. Munasinghe

Index Number: 190397E

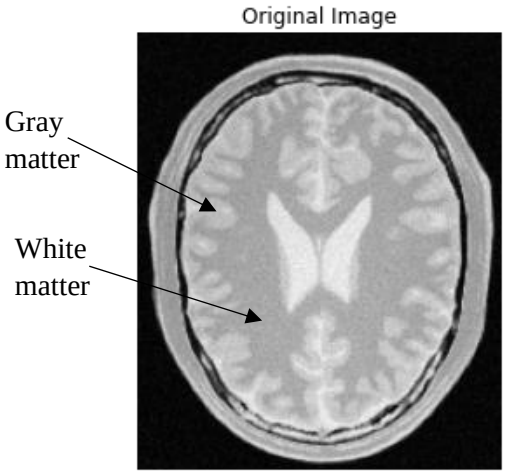
Github Repository: https://github.com/vidurawarna/EN2550_CV.git

Question 1 – Intensity Transformations

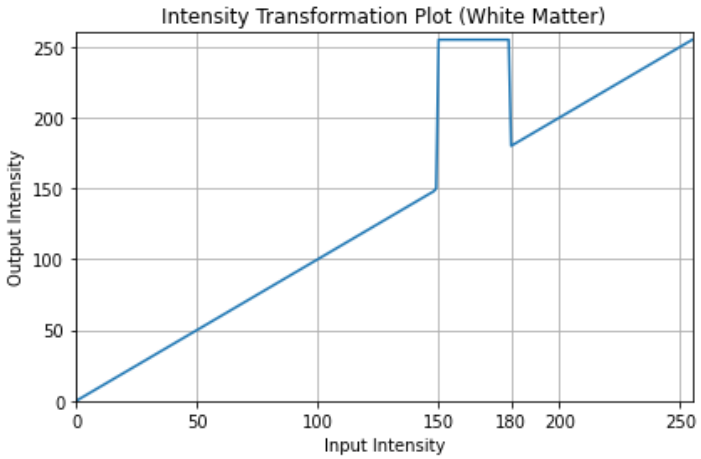
- In the given intensity transformation, the intensity values in the region [50,150] of the original image are mapped to an increased intensity value.
- Therefore, the colors correspond to the values in the region [50,150] get closer to the intensity value of white color than before.
- So those colors in that region will be displayed more whitish than the original image after the transformation.

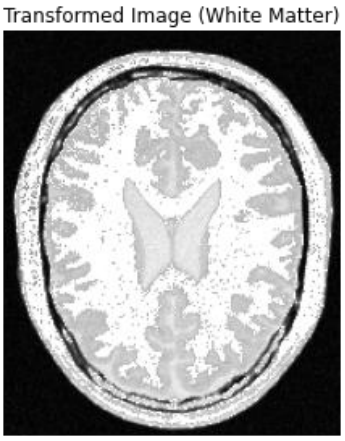
Question 2 – White and Gray Matter

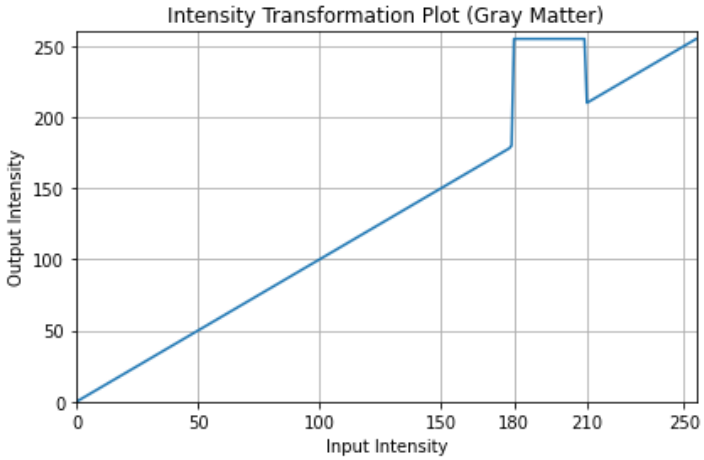
- Now we have to identify white and gray matter from a brain proton density slice image, using the same type of intensity transformations. White and gray matter are two regions in the brain. They are identified as below in the figure.

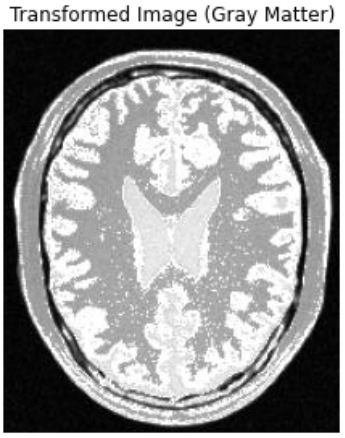


- Identifying the regions to accentuate
  - A window with intensity transformation of value 255 with some fixed width was applied to the image. Other parts of the transformation were kept linear.
  - The range which the window exist was changed to identify the correct range to accentuate.
  - Then after identifying the rough range, some slight changes were done in the width of the window to get the desired output.





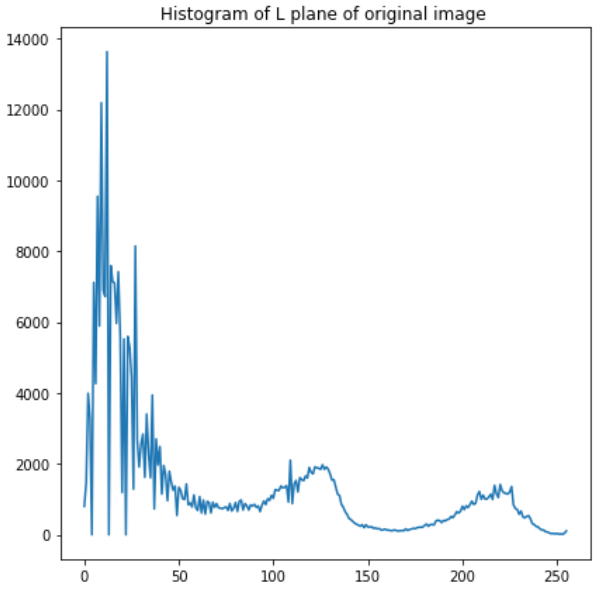


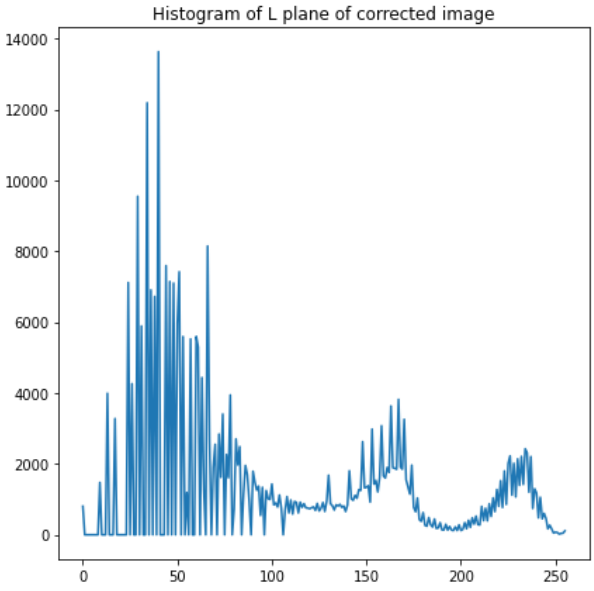


- Since we have to accentuate white and gray matter, other regions are kept linear in the intensity transformation.
    - White matter – [150,180] range is transformed to an intensity value of 255
    - Gray matter – [180,210] range is transformed to an intensity value of 255

<u>Question 3</u> – Gemma Correction on L* Plane

- To apply gemma correction to the L plane in the L*a*b* color space, we have to obtain the L, a, b planes of the image.
- `cv2.cvtColor(img,cv.COLOR_BGR2Lab)` function will return the L*a*b* color space array of the image(img).
    - L* plane – Lightness value, black at 0 and white at 100
    - a* plane – Represents the green to red opponent colors, with negative values toward green and positive values toward red
    - b* plane – Represents the blue to yellow opponents, with negative numbers toward blue and positive toward yellow
- Then an intensity transformation is generated according to the gemma correction. The value of gemma is chosen to be 0.6 in this experiment. (**gemma = 0.6**)
- After applying the transformation to the L* plane, the three planes are combined to generate the new image.





- From the corrected image we can observe that the lightness of the image can be changed without affecting the colors of the image.

Question 4 – Histogram Equalization

- We have to generate a transformation to equalize the image.
- To obtain a transformation we need the histogram of the original image. After that, we can find a transformation according to the following equation.

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

$k = 0,1,2,\dots,L-1$

$L = 256$ Number of intensity levels (Since we are using gray scale images, L = 256)

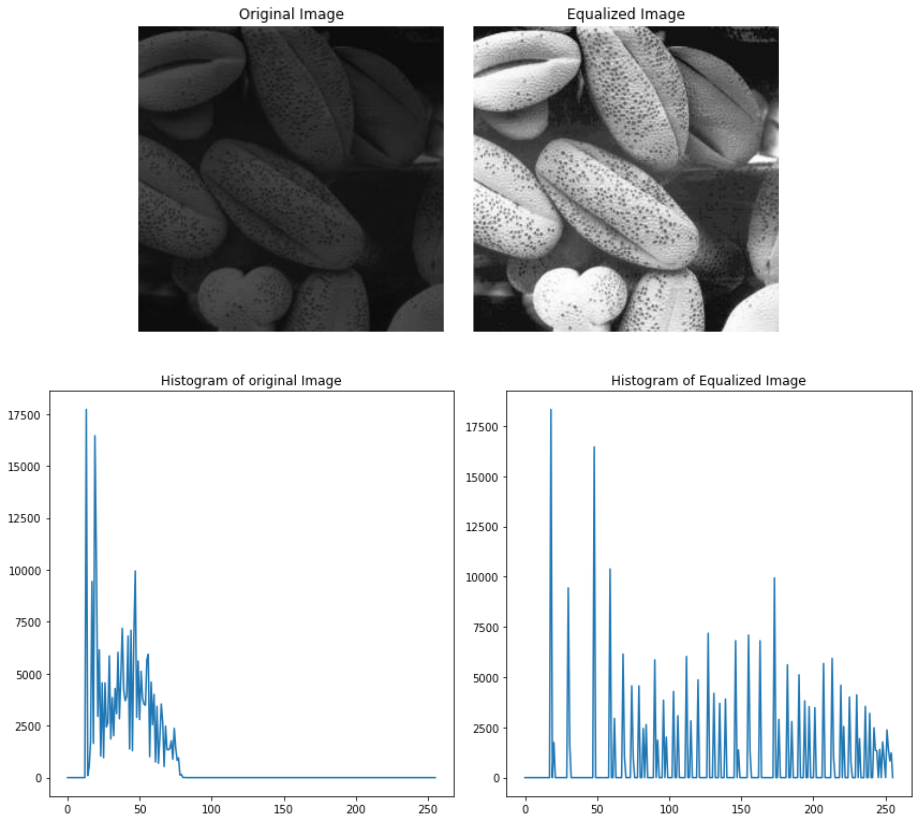
$MN = \text{image\_width} \times \text{image\_height}$

$s_k = k^{th}$ value in the transformation array

$n_j = j^{th}$ value in the histogram of the original image

- Function to carry on histogram equalization

```
def equalizeImage(img,hist):

    transf = np.zeros(256)  #Array to store new transformation for equalizer
    MN = img.size   #Image resolution (Number of pixels)
    L = 256         #Number of intensity levels

#Calculating the transformation according to the equation given above(s_k array)
    for i in range(len(transf)):
        transf[i] = int((L-1)*(sum([hist[j] for j in range(0,i+1)]))/MN)

    #Change the data type to uint8
    transf = np.array(transf).astype(np.uint8)
    #Do the equalizing transformation using lookup table
    eqalized_img = cv.LUT(img,transf)

    #Take histograme of the equalized image
    hist = cv.calcHist([eqalized_img],[0],None,[256],[0,256])

    #returning results
    return eqalized_img, hist
```



Question 5 – Zoom images

- **Nearest-neighbor method** – Single-pixel value of the zoomed image is chosen by considering the nearest pixel value when the pixel indexes of the zoomed image are divided by the scale.
- **Bilinear Interpolation** – The pixel value is determined by using linear interpolation, considering the intensity values of the nearest four points in the original image.

- Calculated normalized SSD values between each zoomed image and original large image.

	Nearest-Neighbour	**Bilinear Interpolation**
Image1	44.2050	39.2872
Image2	19.0738	16.2412
Image3	24.6596	21.3778

```
def zoomImage(img,scale,mode):
    # Scale- amount of zoom
    # mode- nearest-neighbor (mode=0) or bilinear interpolation (mode=1)
    r,c = img.shape[0]*scale,img.shape[1]*scale
    zoomed_img = np.zeros((r,c))
    for i in range(r):
        for j in range(c):
            if mode == 0:
                r_,c_ = roundNeighbour(i,j,img.shape[0],img.shape[1],scale)
                zoomed_img[i][j] = img[r_][c_]
            elif mode == 1:
                zoomed_img[i][j] = linearInterpolate(i,j,scale,img)
    return zoomed_img.astype(np.uint8)

def linearInterpolate(i,j,s,img):
    i_float,j_float,i_int,j_int = i/s,j/s,i//s,j//s
    # defining the ratios
    (p,q) = ((i_float-i_int),(j_float-j_int))
    # Handle overflowing array indexes
    if i_int>=img.shape[0] or i_int+1>=img.shape[0]: i_int = img.shape[0]-1
    if j_int>=img.shape[1] or j_int+1>=img.shape[1]: j_int = img.shape[1]-1
    if  i_int+1>=img.shape[0]: i_int = img.shape[0]-2
    if j_int+1>=img.shape[1]: j_int = img.shape[1]-2
    # Point values to do the interpolatin
    point1,point2,point3,point4 =
img[i_int][j_int],img[i_int][j_int+1],img[i_int+1][j_int],img[i_int+1][j_int+1]
    # Carry out the linear interpolation
    val1 = p*point3 + (1-p)*point1
    val2 = p*point4 + (1-p)*point2
    return val2*q + val1*(1-q)

def roundNeighbour(i,j,r,c,s):
    # Rounding the values to the nearest integer
    i_ = i//s if (i/s - i//s)<0.5 else (i//s + 1)
    j_ = j//s if (j/s - j//s)<0.5 else (j//s + 1)
    # Correcting the overflowed values
    if i_>=r: i_ = r-1
    if j_>=c: j_ = c-1
    return i_,j_
```

- The normalized sum of squared difference (SSD) gives us an idea about how similar two images are. Lesser the SSD value, the more similar the images. From the calculated SSD values we can see that the bilinear interpolation method gives a more similar image than the nearest-neighbor method. Rounding values and handling overflow might have caused the number of dissimilarities of the images.

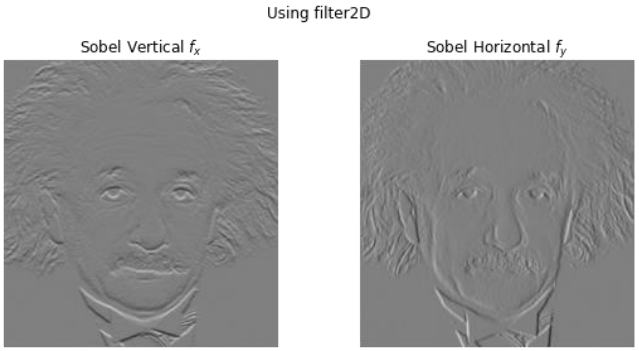
- Zoomed images for Image2.

Question 6 - Filtering with the Sobel operator

Sobel vertical kernel $= \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$

Sobel horizontal kernel $= \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$

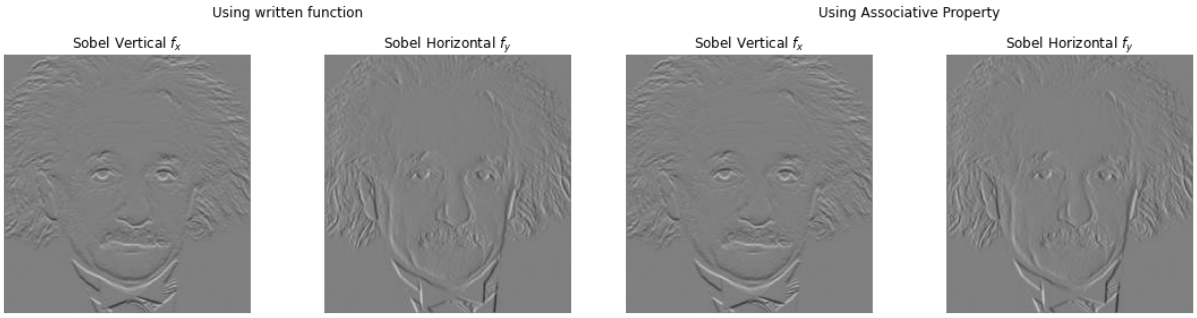
(a) Using the above two matrices as the filter kernels, we can obtain the filtered images using the inbuilt function `filter2D()` in OpenCV python. Sobel vertical filterings give us the derivative of the image with respect to vertical direction while horizontal filtering gives us the derivative of the image with respect to the horizontal direction.



(b) Instead of using the inbuilt filter2D() function, we can obtain the same results by using convolution. In normal convolution, the result will be rotated by 180 degrees. But the Sobel filter kernels are symmetric in this case. Therefore, we can use the convolution operation to obtain our result without flipping the image.

```
def Filter_conv(img,k,mode):
    # mode variable selects the kernal type
    # mode = 0 := 2D kernel
    # mode = 1 := row kernal
    # mode = 2 := column kernel

    k_height,k_width = k.shape[0]//2,k.shape[1]//2
    if(mode==1):k_height=0
    if(mode==2):k_width=0

    img_height,img_width = img.shape[0],img.shape[1]

    filtered_img = np.zeros(img.shape,'float32')

    for i in range(k_height,img_height - k_height):
        for j in range(k_width,img_width - k_width):
filtered_img[i,j] = np.sum(np.multiply(img[(i-k_height):(i+k_height+1),(j-
k_width):(j+k_width+1)],k))

    return filtered_img
```

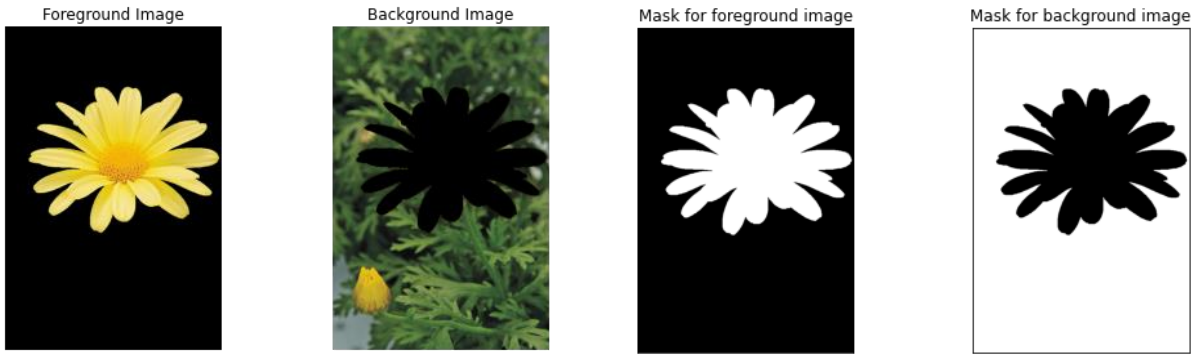
(c) We can use the associative property to do the convolution in two steps to obtain the results.



- All three methods gave the same output. Therefore we can use any of the methods to filter with the Sobel operator. But the last method has less time complexity than 2D for looped convolution.

Question 7 – Using grabCut()

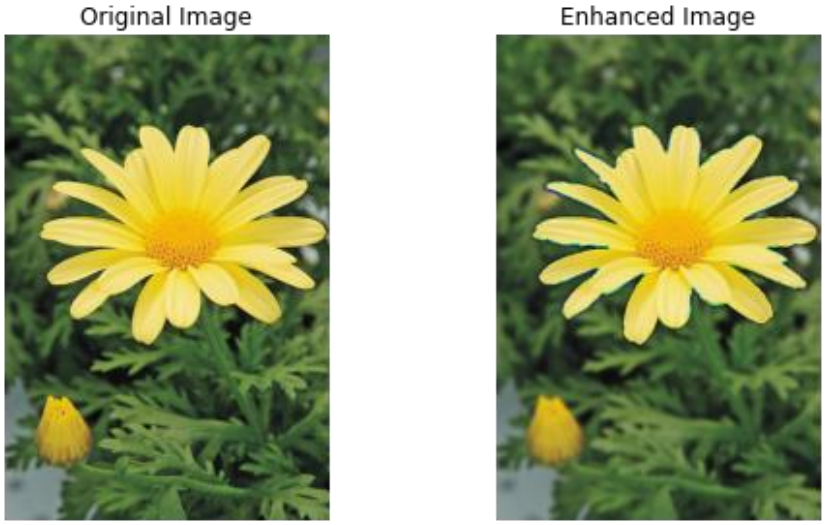
(a)



- Code where the grabCut() function is used to extract foreground and background images.

```
mask = np.zeros(img.shape[:2],np.uint8)
bgdModel = np.zeros((1,65),np.float64)
fgdModel = np.zeros((1,65),np.float64)
rect = (20,150,600,450)
cv.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv.GC_INIT_WITH_RECT)

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

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

(b) First, the Gaussian blur effect was added to the extracted background image with a kernel size of (11,11) and sigma = 5. Then the extracted foreground image and blurred background image were combined to obtain the enhanced image.



(c)

- When we apply the gaussian blur to the extracted background, the part which contains the total black color will remain unchanged.
- But near the edges of that blacked area, the blur filter kernel will contain some parts of the background image.
- Therefore, the convolved value will not be zero for some places along the edge.
- So, when we combine the blurred background and the foreground image, some parts of the edges will give darker colours that we did not expect.

References

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

https://pyimagesearch.com/2020/07/27/opencv-grabcut-foreground-segmentation-and-extraction/

https://docs.opencv.org/4.x/d8/d83/tutorial_py_grabcut.html